Distributed Concurrent Persistent Languages: an Experimental Design and Implementation

H W Francis Wai

A thesis submitted to the
Faculty of Science
University of Glasgow
for the degree of Doctor of Philosophy
March 1988

Declaration

The material presented in this thesis is the result based entirely on my own independent research carried out at the Department of Computing Science, University of Glasgow under the supervision of Prof. Malcolm Atkinson. Any published or unpublished material used by me has been given full acknowledgement in the text.

Francis Wai
August, 1988.

Acknowledgements

I am indebted to my supervisor Prof. Malcolm Atkinson in many ways. He is a constant source of ideas and inspiration and many of the ideas flourished in the thesis are due to him. Through him, I learnt that doing research is not just lonely candle burning nights in front of a terminal but also the importance of the skills required in communicating ideas with others. He is also responsible for creating a stimulating research environment in terms of a comfortable and peaceful office, state-of-the-art equipment and no lack of brain storming discussion groups. Many members of the PISA team deserve gratitude more than I can express. Jack Campin's questions on my work often help to clear my mind. He also commented on drafts of several chapters in this thesis. But I bear the sole responsibility on the presentation of this thesis. When in anger, in frustration, and in desperation, Paul Philbrow and Douglas MacFarlane always lend me their sympathetic ears. Richard Cooper plays the role of being an agreeable officemate. The implementation benefits from the works of Al Dearle and Fred Brown on PS-algol at the University of St. Andrews. I also wish to thank Prof. Ron Morrison who sat patiently on a number of occasions listening to my half-baked ideas and for his encouragement throughout the project. Alexa Stewart helped in making this thesis appear more attractive than it should. The work reported here is supported by a scholarship from the SERC and grants from Alvey and ICL.

Abstract

A universal persistent object store is a logical space of persistent objects whose localities span over machines reachable over networks. It provides a conceptual framework in which, on one hand, the distribution of data is transparent to application programmers and, on the other, store semantics of conventional languages is preserved. This means the manipulation of persistent objects on remote machines is both syntactically and semantically the same as in the case of local data. Consequently, many aspects of distributed programming in which computation tasks cooperate over different processors and different stores can be addressed within the confines of persistent programming. The work reported in this thesis is a logical generalization of the notion of persistence in the context of distribution.

The concept of a universal persistent store is founded upon a universal addressing mechanism which augments existing addressing mechanisms. The universal addressing mechanism is realized based upon remote pointers which although containing more locality information than ordinary pointers, do not require architectural changes. Moreover, these remote pointers are transparent to the programmers. A language, Distributed PS-algol, is designed to experiment with this idea. The novel features of the language include: lightweight processes with a flavour of distribution, mutexes as the store-based synchronization primitive, and a remote procedure call mechanism as the message-based interprocess communication mechanism. Furthermore, the advantages of shared store programming and network architecture are obtained with the introduction of the programming concept of locality in an unobtrusive manner.

A characteristic of the underlying addressing mechanism is that data are never copied to satisfy remote demands except where efficiency can be attained without compromising the semantics of data. A remote store operation model is described to effect remote updates. It is argued that such a choice is the most natural given that remote store operations resemble remote procedure calls.

Table of Contents

Chapter 1

Universal Persistent Store

§1.1 Introduction

A universal persistent object store is a logical space of persistent objects spread over machines reachable across networks. It provides a context for the application programmers that hides the distribution of data so that the store semantics of conventional languages is provided. The manipulation of objects on remote machines is both syntactically and semantically the same as in the case of local data. Consequently, many aspects of distributed programming in which computation tasks cooperate on different processors and stores can be addressed within the confines of persistent programming.

The primary motive for this kind of store is to ease programming. Difficult tasks such as data movement to and from volatile memories and disks scattered over a network can be factored out so that each programmer does not face them. Example application systems where such a store is useful are CAD/CAM, office information systems, distributed databases, program development environments etc.

The work reported in this thesis is a logical generalization of the notion of persistence [Atkinson et al. 81] in the context of distribution. Persistence is the concept whereby programmers see only a one level store. Moreover, objects outlive the processes that created them. They remain available so long as they are still referenced. The immediate implication is that the need for overlaying and explicit transferring and translating data from a backing store into the main memory and vice versa is obviated. This has been proved to be valuable in reducing coding effort in a large class of applications. The figure often quoted is a saving of around 30%. There are other implications as well. For instance, with orthogonal persistence, values of any type may be kept in a persistent store. In PS-algol, a language for experimenting with orthogonal persistence, objects of types such as integer, boolean as well as images of rectangular arrays of pixels and procedures can be placed in a persistent store for subsequent use. This may be contrasted with a number of database programming languages where there is a restriction on the type of objects that can be kept in long term storage. In addition to the orthogonal treatment of persistent objects, a variety of typing and binding strategies are also supported to meet various needs of persistent

data [Atkinson & Morrison 85b]. Thus persistence offers a rich, flexible but at the same time secure framework for constructing large scale systems.

Derivable from the one level store notion is that the locality of data is never a programming concern. This principle of locality transparency is fundamental to persistent programming. The movement of data between a backing store and the main memory is performed automatically on a demand basis by a store management system [Brown & Cockshott 85]. This principle enables the use of the same instructions and therefore the same language constructs for the manipulation of data residing in different storage media. The locality transparency principle can be generalized in respect of distribution. The idea is based on the observation that storage media may be scattered but reachable over networks. The management of distributed data can be dealt with without involving programmers. Hence many aspects of distribution can be captured in a persistent programming language without requiring any syntactic or semantic changes. From the programmers' point of view, the same algorithms can be applied to data whether they are resident in the local physical memory, on a local disk, a remote physical memory or a remote disk.

The universal persistent object store envisaged is a logical one. It can be thought of as the union of individual persistent stores, one or more of which may be associated with a particular machine. The reason for this arrangement is to preserve autonomy which is intrinsic in a distributed environment. In such an environment, the failure of a machine is often regarded as isolated and has no consequence on the availability of other machines. Even in the case of a file server breakdown, client machines are still available as computation engines. Similarly, the addition of a machine does not affect others in any way. The universal persistent store reflects this property of a distributed system. When a machine with a persistent store breaks down, objects on that persistent store become temporarily unreachable. However, objects in other parts of the universal store remain available. Furthermore, adding a new node with a persistent store on a network merely enriches the overall store.

The universal persistent store can be provided for, independent of machine architectures, on a heterogeneous system. Although different kinds of machines may be connected over the same network, the availability of physically remote objects relies on the availability of a common communication protocol. Unfortunately computer

manufacturers do not necessarily conform to the same standard protocols even though they exist. An example is ICL Perq computers and DEC Vax computers which use incompatible protocols.

The construction and maintenance of such a universal store is feasible only if the administration of different resources is at least cooperative in nature. There may be practical difficulties as the system gets larger. But, we contend, the advantages offered by such a store are more than offset the problems that it is likely to raise. However, a centralized mechanism would be unreliable and would perform poorly when the system becomes large. Hence a cooperative architecture for the universal store is required.

Another pragmatic uncertainty arises with such a store is locality transparency. The principle is fundamental to persistent programming as explained above. But the advantages of a distributed system can not be exploited because of this. This thesis reports the construction of an experimental apparatus to test whether the principle of locality transparency should be retained or whether programmers should control data and process locality in a distributed system. A programmer may work within the constructed distributed system without knowledge of the locality of objects, this being managed entirely automatically. Equally a programmer may explicitly control the location of data and process and construct algorithms which explicitly change the locality of these objects. Consequently it should be possible to observe programmers using this system in either fashion or in some compromise combination of these styles, and hence glean information to guide the design of future languages and implementations. For example, if consistent patterns of explicit locality use are observed we may later directly support this programming with suitable constructs.

§1.2 Distributed Shared Stores

Multics [Organick 72] is one of the early well-known operating systems. One of its outstanding features is the concept of one level store. Each process is allocated a very large address space organized as a collection of 2^{14} segments each consisting of as many as 2^{18} 36-bit words. The main reason for supporting such a large address space is to avoid the necessity of overlays and therefore data movement within it [Daley & Dennis 68]. The problem addressed here is the issue of sharing. The issue is an important one as it has strong implications on building complex systems out of existing ones. The issue remains central to the design of generations of multi-user systems.

With the recent technological advances of high speed networks and VLSI, the large and expensive multi-users systems are gradually giving way to small, cheap but equally, if not more, powerful personal workstations connected over a network. In the absence of common physical stores and the lack of a central management system, sharing becomes once again the focus of research into providing distributed programming environments [Needham & Herbert 82].

One of the motivations behind persistence is the desire to facilitate reuse of software components and other data by providing a type secure yet flexible programming environment. The idea is akin to the concept of very large address space first advocated in Multics. In the universal persistent store the unit of sharing is the object whereas the unit of sharing in systems like Multics is the segment. The principal difference between the segment and the object is that objects retain their types which mediate their use even when they are created by one program and reused by another. Given that there is provision for accessing objects by name this kind of system is analogous to a filing system, except that there is a richer repertoire of structures and typing is imposed. Thus the issues that arise in distributed filing systems [Schroeder et al. 85] are extended by those of typing.

It has been pointed out earlier that the facilities of persistent stores can be provided in a distributed environment in the most natural way. The provision of a universal persistent store presents to programmers on a distributed system a single paradigm in the manipulation of both remote and local data.

§1.2.1 Overview

Locality transparency in persistence is achieved by providing very large address spaces so that data whether in heaps or on disks can be identified uniformly with persistent identifiers. This is the basis on which language constructs are insensitive to the locality of data. Such an approach is adopted here and a form of universal addressing is introduced for the purpose of accessing non-local data. An objective of this thesis is to demonstrate that such a universal persistent store can be realized in a way that has a minimal impact on programming semantics. There are three reasons for this: 1) to avoid affecting innocent by-standers who are accustomed to tradition and those who do not wish to take advantage of distribution, 2) to avoid learning costs and 3) to capitalise on existing investment.

The main difficulties to overcome in realizing such a store are:

1) to allow dynamic merging and decoupling of persistent stores with minimal impact on existing programs and data;
2) to support a universal addressing structure allowing different addressing schemes with minimal trade-offs in flexibility, space and performance;
3) to facilitate communication of data between different kinds of machines on the network under the control of either programmers or the system in a manner consistent with the programming paradigm of persistence.

It is not obvious, a priori, that with the present technologies and current theories these difficulties can be overcome. To test whether they can be overcome a universal persistent store was implemented and used. The following assumptions about requirements and contexts were made:

- A network is understood to be made up of a number of machines with individual processors, physical memories and backing stores connected over a high speed communication network such as the Ethernet. The number of machines connected and therefore the number of machines participate in the construction of a universal persistent store is restricted only by the underlying communication hardware.

- Autonomy is important, in particular, there is no notion of master and slave machines on the network. Machines may share one or more file servers but this does not cause any contradiction.

- The machines need not be homogeneous. They may have different architectures. Moreover, machines are not required to have a provision of local persistent stores in order to participate in a universal persistent store.

Primarily, we seek the advantages of shared memory programming in the context of network architectures. These advantages are derived from uniqueness of objects or pieces of store and semantic implications therefore include : 1) cheaper value equality, 2) space efficiency and visibility of updates and 3) a richer spectrum of binding strategies.

We envisaged that processes running on different machines cooperate with each other by using the universal persistent store. Objects created by one process can be shared with other processes and may become directly manipulable. For instance, if the shared object is a procedure, a remote process could call it as if it were a local procedure. Similarly, if it is a vector of integer variables, it can be assigned or used in expressions as if it were local. In either case, no extra syntax is required and possible side-effects can be seen by all those concerned. This kind of facility does not necessarily compromise the well-known engineering principle of encapsulating information. Implementation details can be hidden using scope rules or abstract data types. This idea is illustrated in some of the examples in chapter 8.

Process communication, which is essential in a distributed system, is seen in this context as a means of establishing initial links with programs or data. Once the root to a tree has been obtained, any leaf node becomes automatically reachable. For example, once an instance of an abstract data type is obtained through communication, any of its procedure components can be called without explicitly invoking communication again.

Normally there is no movement of objects between machines, rather cooperation is achieved by the implicit movement of operations to objects. An implementation may copy for engineering reasons, provided it retains the illusion of no copying. For example, immutable values such as strings are copied in communication. Hence, string operations can always be carried out locally without the overheads of communication. But care has to be taken to ensure the equality test still gives correct results. The strategy of not copying data in general is important as the scale of the problem of data inconsistency is reduced and can be dealt with in a straightforward manner. It has other advantages as well. The most important one is that synchronization primitives such as semaphores can be implemented in virtual memory. If data were copied, atomic access to semaphores could not be guaranteed. This provides a platform on which experiments with concurrency control can be carried out within the language.

Requests for the manipulation of data are always obeyed at the sites where they were created. In consequence, resilience, which can be achieved by replication, cannot be provided. Applications such as some relational database systems where replication of data at various sites is required, cannot be supported. Two primitive concepts are introduced so that particular forms of replication for reliability and recovery may be

programmed, these are *locality* and explicit copy.

§1.3 Related Work

Research on shared stores has always been in the realm of operating system people rather than language designers. Although the dividing line is becoming blurred as a consequence of persistence, it is not surprising that related work on distributed shared stores is relatively rare. Related work on programming languages is covered in chapter 3 and here we consider operating systems and stores.

The concept of distributed shared memory is absent from some distributed operating systems such as LOCUS [Popek et al. 81], the Newcastle Connection [Brownbridge et al. 82], the Network File Service of Sun [Walsh et al. 85] etc. They are concerned with providing a Unix-like computing environment over a network. The idea is to put together individual Unix file systems connected over a network. Thus one file system may appear to be a sub-tree of either another or a network-wide file system so that any file in it can be accessed in the usual Unix paradigm.

A variety of strategies in copying data can be found in some of the following distributed operating systems, although they advocate the concept of distributed shared memory.

Accent [Rashid & Robertson 81, Fitzgerald & Rashid 86], the Distributed V Kernel [Cheriton & Zwaenepoel 83], and the Apollo Aegis system [Leach et al. 83] are all message based distributed operating systems. They allow processes on different machines to share pages of one another's address space. The idea is that when a page fault occurs, the missing page can either be brought in from a local secondary storage or fetched from a remote machine. Eventually, modified pages are sent back to the home nodes to effect updates. Within this framework, there are variations on the approach and in particular the concurrency control strategy.

In Accent, there is an emphasis on the relationship between inter-process communication and the virtual memory management. To handle distribution, there is a network server process on each node to facilitate the transfer of pages between nodes. It serves as an intermediary between processes on different nodes sharing a common collection of pages. There is an interesting notion of copy-on-write so that each process concerned has a logical copy of a shared page. This can be

explained as follows. Pointers to pages may be passed in messages for the purpose of inclusion in the page table of another process. However, because of the asynchronous semantics of message passing, the sender process may alter the pages before the message is received and acted upon. When this happens, copies of the pages are then made in the sender process' address space.

The Distributed V Kernel approach is not radically different from that of Accent. The major difference is that communication is synchronous. Processes may send pointers to locations in their address spaces in messages. These sender processes are then suspended until replies arrive. In the meantime, the receiver processes may use the primitives *MoveTo* and *MoveFrom* using pointers and process identifiers passed as parameters in transferring data between the address spaces before replies are sent.

The Apollo Aegis provides processes, in the tradition of Multics and IBM System/38 [French et al. 78], with a network-wide flat address space. All processes running on different machines on a network see a single level store so that both local and remote data whether in main memory or on disks are accessed uniformly. Unlike Accent where because of the notion of copy-on-write there is no need for synchronization and the Distributed V Kernel where synchronization is built into the message passing semantics, concurrency control over copies of shared objects in Apollo Aegis is achieved by using locks. Because processes on each node can access objects on another node and because lock modes can change e.g. from read to write, each node maintains a lock database with lock records of not only local but also remote objects referenced as well. This entails a complicated procedure in realizing the locking convention. Lock and unlock requests are always sent to the home node and the lock databases at the two nodes are updated simultaneously. Locking of an object may cause local out-of-date page copies to be removed from a process address space. Up-to-date page copies will then be transferred on a demand basis. Unlocking of an object causes pages to be sent back to the home node before the remote lock database is modified.

The Monads system [Abramson & Keedy 85] appears to offer a simpler yet more attractive alternative in its locking convention. The idea is to mark each page as either read-only or read-write. An interrupt is generated if an attempt is made to write into a read-only page. There is a memory server global to all machines on a network which maintains a page status table. This table keeps track of the whereabouts of pages as well as their mark bits. It is interesting to note that concurrent requests

for a page are always granted. Any inconsistency which may arise is avoided by changing the mark bit of the most up-to-date copy. For example, if a page is used in read-write mode and the new request is read-only, the node holding the page is asked to return the most up-to-date copy of it to the memory server. At the same time, that page is marked as read-only at the site where it is currently held. The most up-to-date copy of that page is subsequently sent to the requestor node. Both nodes then have a read-only copy of the page and the memory server also has the most up-to-date copy.

There is a concept of write-through-stores on the ICL 3900 machines. Processes distributed on nodes connected by the MACROLAN [Warboys 85] may share a segment and a write into that segment causes update messages to be broadcast. In all such systems we know of this write through mechanism depends on hardware support.

The approaches taken by these operating systems represent novelties in trying to support distributed shared stores; some of them, such as the System/38 (with network-wide addresses), were even ahead of their time. However, most of them either lack generality (e.g. because special hardware is assumed as in ICL 3900 series), suffer the problem of scale-up (e.g. the memory server on the Monads system when moving to handle larger address spaces is likely to become a bottleneck) or a combination of both. Other approaches, which do not entail such rigid features, are worth exploring and the approach employed in the realization of the universal persistent store represents one such.

§1.4 Thesis Organization

Generality means different things to different people. In the case of operating systems, typically the use of a low level facility often requires expertise in different areas. For example, the use of a Unix 4.2 socket requires knowledge of address families, protocols, naming and binding of sockets, and even the size of data that can be transmitted in a single packet. To encourage the exploitation of network environments, programmers should be protected from having to use and understand such low-level details. The universal persistent store approach offers an attractive alternative whereby the underlying systems can be presented in an abstract and unobtrusive manner.

The realization of the universal persistent store relies on two mechanisms: an RPC mechanism and a universal addressing mechanism in addition to the persistence mechanism. The universal

addressing mechanism is unusual in that it is not designed to be a stand-alone addressing mechanism. It relies on the existing addressing mechanisms to locate data. The RPC mechanism is integrated into a concurrent abstract machine. The design and implementation of both the RPC mechanism and the concurrent abstract machine are described in chapter 5. The universal addressing mechanism is covered in chapter 7.

A concurrent language based on PS-algol has been designed and implemented to experiment with the idea of a universal persistent store. The design work and in particular the programming concept of *locality* are discussed in chapter 4 and the implementation is described in chapter 6. Examples using the language can be found in chapter 8.

This work grew out of the need for concurrency in PS-algol. The problems of concurrency in PS-algol are examined in chapter 2 where previous attempts in extending PS-algol with concurrency are also described. A concurrency model which fits well with the distributed store model is arrived at based on a survey of concurrency and the analysis of the strengths and weaknesses of the related mechanisms. These can be found in chapter 3.

The conclusion and a summary of related areas not covered in this thesis appear in chapter 9.

Chapter 2

PS-algol

§2.1 The Language

PS-algol was designed as a language supporting experiments into persistent programming [Atkinson et al. 81, Atkinson et al. 83, Atkinson & Morrison 85a, Philbrow & Atkinson 86, Carrick et al. 87]. It is a strongly typed language in the Algol tradition. Many of the features are similar to other conventional languages such as Algol68 but its principal innovations include:

- integrated graphics,
- higher order functions, and
- orthogonal persistence.

As a result of the adherence to the principle of data type completeness, the language is powerful yet simple to learn.

The language has a rich set of data types. The base types are integer, real, boolean, pixel, string and picture. Higher order data types are generated by the (possibly recursive) application of the type rules. The notion of orthogonal persistence is derived from the relationship between persistence and types. What it means is that any value irrespective of its type can be made persistent. This is contrasted with other related languages such as Pascal/R [Schmidt 77] in which only values of a subset of the language's data types are allowed to persist. The generality of the PS-algol persistent mechanism has proved to be invaluable in many respects. For instance, since procedures are first class values, they can be made persistent. An implication of this is that modules or objects (as in Smalltalk) can be constructed, without further language support, as persistent structure instances whose fields are procedure values.

The application domain includes that of conventional languages and is also aimed at database related utilities. Experience gained in the use of the language has proved to be fruitful. It has been used as the basis of an Alvey project [Atkinson et al. 87, Hepp 83, Kulkarni 83] and several others including Prolog [Gray et al. 87] and a distributed relational data system [Norrie 85]. At the time of writing, there are also plans to use it in several large scale projects e.g. support of Flagship's data store [Sparks 88]

and the store for IPSE 2.5.

In retrospect, it was felt that the language represents a milestone in the pursuit of persistent programming languages. However, some of its features have been overlooked in the process. In particular, its most powerful feature is where one of its weaknesses is most felt.

§2.2 Persistence and Concurrency

PS-algol is one of the first store semantics languages with the provision of a persistence store. The notion of a store is much more general than in other conventional languages. A persistent store is an abstraction over physical memories [Cockshott 83] in that:

- from the programmers' point of view there is no distinction between RAM and the secondary storage,
- it is meant to be a data depository and there is no limit on the amount of data that can be put into it,
- it is reliable and transactionally secure, and
- it is object oriented.

In the present implementation, the persistent store is organized as a collection of databases. There is no constraint on the kinds of objects that can be put into any one of these databases. There is no limit on the amount of data that the store can accommodate as the number and the size of databases are not constrained either. Thus it appears to the programmers that the underlying store is a very large and secure one-level store. Data whether in RAM or on disks are manipulated uniformly as if they were in RAM.

On a multi-user system such as Unix™, a number of PS-algol processes may run simultaneously against the same persistent store. In order to resolve conflicts that may arise, concurrency control is necessary. However, since concurrency is not defined in the language, concurrency control has to be introduced at the store level. In order to ensure the integrity of data in store, each PS-algol process is seen as a single transaction by the store management system. A transaction is considered to have two parts: atomicity and stability. Atomicity is supported by a locking convention which guarantees that the effect of a transaction is seen by others as a whole. That is to say either it takes place or nothing happens at all. Stability is provided through a language interface to the store. The effect of calling this interface is to cause copies of objects reachable from a root from which persistent objects hang to be sent back

to the persistent store. The entire operation itself is atomic. Stability here means that once in store, data are free from anomalies such as power failures. Moreover, only inverse operations can nullify the effect of stability.

The locking convention applies to all processes running PS-algol programs and is enforced implicitly by the store management system. It is rudimentary: multiple readers and single writers and the unit of locking is a database. Locks are acquired at the time when the required databases are open and are not released before the process has terminated. In retrospect, the efficacy of the system with respect to the utilization of the persistent store has not been found satisfactory. There are a couple of reasons for this.

First of all, the granularity of lock is unacceptably coarse. The unit of locking is a database, disregarding the ratio of the amount of data in it and that actually used in a transaction. Consequently, some transactions are locked out unnecessarily for arbitrarily long periods. The degree of concurrency attainable is not as high as one would like. Logically speaking, there is no reason why transactions manipulating a non-intersecting set of persistent objects cannot proceed in parallel.

The consequence of organizing the persistent store as a collection of databases and the locking convention was a grave one. A database could contain objects which have pointers to objects in other databases. De-referencing such pointers automatically causes these databases to be locked in the same mode as the one containing those pointers. Under such a circumstance, the system could deadlock yet there is no trace of it anywhere in the source codes. This makes understanding programs much more difficult. In principle, a separate PS-algol 'database' could be constructed for each object or group of objects that require a lock, though the present implementation would make that inefficient.

It is beyond any doubt that concurrency is essential as the volume of data increases in persistent stores. The lower the utilization of data the less attractive persistent stores appear to be. The present arrangement in PS-algol is an ad hoc expedient. The issue has not been addressed properly.

The core problem is the lack of coordination among transactions. The compiler does not have enough information on the extent of the effects of transactions. Because of this, the unit of locking as imposed at the persistent store level could not have been finer than its structural

unit viz. a database. Although this guarantees the atomicity of transactions, it is at the expense of locking out transactions unnecessarily. Ideally, the subject of locking should be at a lower level, for example, on the basis of either individual objects or a group of objects. If an object has been locked out exclusively then any acquiring transaction will have to wait until it is released. Research in this area have found optimistic approaches of such nature often more favourable [Kung & Robinson 81] in sparsely populated databases as the chance of running into a deadlock situation is relatively small.

One approach we could take is to do away with the locking convention imposed at the persistent store level altogether. Instead synchronization primitives and concurrency are introduced into the language to allow customized locking conventions.

There are other reasons for wanting to introduce concurrency into the language. Firstly, the expressiveness of the language is enhanced as some inherently concurrent applications such as simulation of discrete events, handling i/o events etc. can be described in a very natural way. One may argue that it is possible to use a sequential language to emulate pseudo-concurrency. However, the resulting programs are often less structured and the logic more complicated.

Furthermore, concurrency is one of the means whereby it is feasible for the language designers and implementors to explore the underlying machine architecture. Concurrent programs give rise to the opportunity that their fragments can be executed in parallel on multiprocessor machines.

Last but not least, concurrency is inseparable as a component of a distributed system. Concurrent processes are building blocks from which distributed applications can be realized.

§2.3 Previous Work

Four attempts have been made in tackling the concurrency problem in PS-algol. Larry Krablin developed CPS-algol [Krablin 85a] in his investigation into transactions and persistence during a visit in 1985 from Burroughs Corporation, USA. Chris Barter from University of Adelaide, Australia, proposed language extensions in experimenting with the idea of polymorphic persistent processes in 1987 [Morrison et al. 87]. Richard Connor, a member of the PISA team at University of St. Andrews, is pursuing the same line as Krablin but using PS-algol as the

platform for experiments. Michael Guy at ICL implemented a concurrent version of PS-algol on ICL 3900 series computers.

In CPS-algol, concurrency is introduced by a form of dynamic processes. A dynamic process is created using the language construct **fork** which acts on procedures. Unlike in Unix, the forked process executes in the same environment as the executing process. Furthermore, there is no dependency between child and parent processes. Parent processes are not suspended because of the creation of child processes. The termination of one type of process is not conditional on the other. There are two pragmatic reasons for allowing this. Firstly, there is a restriction on procedures that can be used in a **fork** statement; they must not return results. Secondly, the representation of processes is a chain of stack frames which are objects in the heap. Thus environments remain reachable even if processes have terminated.

Krablin favoured this form of concurrency because of its generality. But he also pointed out that because of the dynamism, more care and discipline are required on the part of the programmers. The interaction between forked processes and the PS-algol notion of transaction is also a subject of concern. The problem is that the effect of a commit is global to all processes sharing the same environment.

The focal point of Krablin's interest is in transactions [Krablin 85b]. Concurrency is a means whereby parallel transactions can be obtained and therefore their study is made possible. In particular, he is concerned with atomicity and deadlocks as persistence already provides the provision for stabilizing the effects of transactions. In his experiments, he proposed using some form of critical region (see Chapter 3 for a more general description) as the synchronization primitive. The following simple example sketches the idea.

```
resource a(int b; bool c)
let d = a(10, true)
...
atomic on d do {await d(c) = true; d(b) := 0; d(c) := false}
```

Protected data are encapsulated in resource expressions such as *d* which is created as an instance of the resource class identifiable as *a* in the example. Accesses to them can be made only through an **atomic** statement. If *d* is in use, the executing process is blocked. Otherwise, it continues to carry out the assignments if furthermore the boolean field *c* indicates a **true**. If *d(c)* evaluates to **false**, the executing process is then

blocked and *d* is made available to other processes. It is resumed when *d* can be seized, the condition expression is evaluated again prior to the assignments.

The intention is that a transaction can be realized with an **atomic** statement where atomicity is guaranteed and stability can be achieved by performing a commit in scope. However, the problem of deadlock is left open. It remains the responsibility of the programmers who organize the order in which resources are accessed.

In contrast, Barter is interested in the concurrent use of a large volume of data which may be distributed over different machines on a network. The usefulness of the Krablin approach is considered to be inappropriate in the absence of shared stores. Barter's idea is centered around communicating processes. Each process encapsulates protected data which can only be manipulated by procedures local to the process. These procedures are invoked by sending requests. Mutual exclusion is achieved by ensuring that requests are served one at a time. There is a strong resemblance in spirit with monitors in his approach.

The language extensions put forward by Barter come close to that of Ada. The proposal is also close to some of the ideas which I had arrived at independently. In his proposal, processes are also dynamic and independent from one another as in CPS-algol, but processes are created as instances of process classes. A process class can be introduced into the type system. Since processes do not share data even if they share the same store, communication is achieved by calling each other's procedures.

Krablin and Barter addressed the same problem of the concurrent use of persistent data but from two extreme perspectives. Krablin concentrated on the construction of concurrent transactions on a local store and Barter was more concerned with the distribution of data. The work of Krablin's was experimental in nature supported by an implementation and the concurrency model did not appear to be more attractive than that of Barter's. On the other hand, Barter seemed to have left open the issue of distributed transactions. The work on concurrency reported in this thesis, which was undertaken concurrently with other work, is an attempt to tackle the same problem again but with a flavour of the two approaches.

Connor has experimented with the use of a CPS-algol-like language to examine its efficacy in implementing various transaction models.

Michael Guy has built a system which adds monitors to PS-algol and uses object level locking.

It is not yet clear which models of concurrency will work best for persistent languages, particularly with respect to programmer comprehension and use. It is therefore worthwhile conducting the experiment reported in this thesis, and other experiments, to explore this issue. The sub-issues include locking and concurrency control, process synchronization and communication and matters arising from the distribution of data such as distributed transactions and distributed commit.

Chapter 3

A Survey of Concurrency

§3.1 Objective

Concurrency is perhaps the most talked-about subject in Computer Science. It has been studied in many fields: programming languages, theory of computations, database systems etc. The fields are diverse and the wealth of knowledge that has been generated is probably too much to be comprehensible for any single person. The principal objective of this chapter is to understand what concurrency means particularly in the areas of programming languages and database systems. This is for the purpose of introducing concurrency into PS-algol with a flavour of distribution.

The study of existing concurrent programming languages is aimed at the understanding of different concurrency models; their strengths and weaknesses as well as the state-of-the-art technologies. But there is no intention of evaluating them against some problem domain nor do we wish to suggest a universal one.

There are a large number of concurrent programming languages; the survey is conducted based on a selection of representative languages: Algol68 [van Wijngaarden et al. 76], Concurrent Pascal [Brinch Hansen 75], CSP [Hoare 78], Distributed Processes [Brinch Hansen 78], Ada [ANSI 83], Amber [Cardelli 85], Occam [May 83] and Modula-2^+ [Rovner et al. 85].

A word of warning is in order at this point. We take the liberty to use the term process in the following discussion. It is not properly defined and it is meant to be some kind of concurrent activity. This will give us the necessary freedom to talk about various aspects of concurrency without committing to a particular form of concurrency.

§3.2 Synchronization Mechanisms

As soon as concurrency is admitted, the notion of atomicity becomes prominent. We are not aware of it because everything is ordered in sequential programming. When something goes wrong in a program we are inclined to think that the sequence of events (e.g. declarations, assignments, procedure calls etc.) is not in the correct order or an event is missing. Atomicity of change is always guaranteed and to correct a problem can be a matter of re-arranging events. The ability to control the

ordering or timing of events is lost once concurrency is admitted. Concurrency in this context is usually taken to mean that the ordering of events is not important; that is to say two events could happen at the same time or in either order.

A favourable interpretation of concurrency is that there is a partial order on the happening of events. What this means is that there are occasions when some subsequences of events do require to be ordered. A simplistic example which serves to illustrate the point is the simultaneous execution of two identical statements viz. "x := x + 1". If there is no synchronization, the effect of one of the updates is likely to be lost. The problem is that we can never be certain about it. We therefore cannot tell whether or not a bug exists in the program. Partial ordering of events can be achieved by the provision of concurrent processes which are used to delineate unrelated sequences of events and synchronization mechanisms which serialize selected events between processes.

One should notice that there is already some primitive synchronization mechanism at the store level. The parallel execution of the two statements "x := 2" and "x := 3" results in x being assigned a value of either 2 or 3 and nothing else. The fact that we are uncertain which value x has taken is more a semantic than an integrity concern. Synchronization mechanisms are used to guarantee the integrity or consistency of shared resources.

§3.2.1 Semaphore

The example on the parallel execution of "x := x + 1" above is not an arbitrary choice. It highlights the core problem of synchronization. The statement represents a sequence of three events: inspection, addition and assignment, in that order. Two processes could do the inspection at about the same time without being aware of one another. Instead of incrementing the value of x twice, the effect of the one of the pair of addition and assignment is simply duplicated. A remedy is to arrange the sequence of events to be carried out indivisibly. One way this can be achieved is to abstract the sequence of events into a single operation and to make sure that such an operation can only be carried out serially. Two practical examples are: 1) a test-and-set store instruction and 2) non-interruptable system calls.

It turns out that an indivisible operation such as the test-and-set instruction is very useful. It can be used to organize synchronization in more general settings.

In [Dijkstra 68], Dijkstra proposed the use of semaphores as a synchronization mechanism in the accompanied proposal on introducing concurrent programming into Algol60. The idea of semaphore is based on a convention of signals whose origin can be traced to the early railway systems. A common definition of semaphore is described in [Ben-Ari 82] as:

> A *semaphore* *s* is an integer variable which can take on only non-zero values. Once *s* has been given its initial value, the only permissible operations on *s* are to call the procedures *wait(s)* and *signal(s)* which are primitive operations. ... The definition of these operations is as follows:
>
> *wait(s)* : If $s > 0$ then $s := s - 1$ else the execution of the process that called *wait(s)* is suspended.
>
> *signal(s)* : If some process P has been suspended by a previous *wait(s)* on this semaphore *s* then wake up P else $s := s + 1$.

There are two weaknesses in semaphores. Firstly, careless use involving more than one semaphore can cause deadlocks. Two processes are in a deadlock, for instance, when two semaphores are seized in different order by each.

Secondly, the use of semaphores is merely a convention. The problem with conventions is that they are often not enforceable. A process could sneak in by executing a V operation and throw other processes into disarray. Apart from this, a matching pair of P and V operations could be applied to different semaphore variables unintentionally without being detected.

The mechanism is not as grim as it looks. A language could support semaphores by allowing variables of some base type to be declared and providing two P and V like operations which act on variables of that type. The purpose of supporting semaphores as a base type is to let the compiler enforce the use of those semaphore variables (distinct from integer variables) with only the two operations permitted. Moreover, the compiler, in some cases, can enforce the consistent use of the two operations by static analysis. For example, it can ensure that P and V are always used in pairs and on the same semaphore. The implication of these rules is the loss of flexibility making programming more awkward. But the compiler checks can easily be bypassed by data dependent code so that the responsibility of accessing protected resources rests entirely with the programmers.

We list below three conventional languages that support semaphores or their variants:

Language	Type	Operations
Algol68	**sema**	level, down and up
Ada	none	pragma shared
Modula-2+	**mutex**	acquire, release and InitMutex

The proposal on parallel processing by Dijkstra was adopted in Algol68 [van Wijgaarden et al. 76]. Consequently, Algol68 allows variables to be declared of type **sema** which can be initialized using a standard function *level*. The function *level* takes an integer parameter and thus allows general semaphores. The two operations *down* and *up* are the P and V operations respectively.

In Modula-2+ [Rovner et al. 85] monitors (see §3.2.3) are dropped in favour of synchronization facilities at a more primitive level. Variables can be declared of type **mutex** and they are initialized by the standard function *InitMutex* and used as binary semaphores. The functions *Acquire* and *Release* are supported as the P and V operations respectively. However, for the sake of conveniency the language supports a **lock** construct which ensures that *Acquire* and *Release* are called at the appropriate places. Thus the following statement,

```
lock aMutex do whatever() end
```

is a convenient shorthand for the following program fragment in Modula-2+,

```
var &t: POINTER TO Threads.Mutex;
&t = System.Adr(aMutex);
Threads.Acquire(&t^);
whatever();
Threads.Release(&t^);
```

In addition to mutexes, condition variables are also supported. The latter together with *wait*, *broadcast* and *signal* operations that release mutexes under different circumstances, are meant to be used in avoiding deadlocks.

In Ada [ANSI 83], some (scalar and access) types of variables can be protected against uncoordinated concurrent accesses. This is achieved

using **pragma shared** which is followed by the variable to be protected. An execution of a **pragma shared** statement causes a semaphore to be attached to the variable. The semaphore convention is enforced in that the compiler generates the appropriate code whenever there is a read or an update access to such variables.

§3.2.2 Conditional Critical Region

An alternative to semaphores was proposed by Hoare in [Hoare 72].

The suggestion is considered more useful than semaphores in that it aids structured programming. The notations Hoare suggested were:

```
r: record ... end;
...
resource r;
...
with r when cond do CriticalRegion;
```

Here *r* is declared as a variable of some record type. The *resource* statement indicates that *r* is to be protected against concurrent accesses. The **pragma shared** statement in Ada is reminiscent of this in that they serve the same purpose. However, *r* has to be accessed in a conditional critical region statement (CCR) where the scope of *r* together with its field is constrained to be within *cond*, a Boolean expression, as well as *CriticalRegion* in the example. The compiler makes sure that only when the condition expression *cond*, whose purpose is to peek at the state of *r*, is true does *CriticalRegion* get executed. Otherwise the executing process is blocked until the condition becomes true due to a change of state by other processes. Note that the condition expression *cond* is optional. By default, it is assumed to be true.

Given a low level semaphore support, the implementation of CCR is straightforward. A resource statement causes a semaphore to be created. For every CCR statement naming that resource, the object code of the critical region of a CCR statement is preceded by seizing the semaphore and is followed by releasing it. Moreover, the evaluation of the condition expression is performed only after the semaphore is acquired. The situation when two processes finding their condition true and both entering into their critical regions is avoided. In any case, if the condition is false the executing process is suspended and the semaphore is released.

We demonstrate here the inter-definability between semaphores and CCRs. We have casually outlined a possible implementation of CCR

using semaphores. We show here how to obtain semaphores using CCRs. In any language that supports higher order functions, semaphores can be obtained as instances of an abstract data type. Suppose we have the necessary facilities in PS-algol (cf. CPS-algol), a general semaphore ADT would then appear as:

```
structure semaphore(proc() P, V)

let Sema = proc(int range -> pntr)
           begin
           structure Mutex(int capacity)
           if range <= 0 do raise NegativeNullException
           let mutex = resource Mutex(range)
           semaphore(
                      proc()                          ! P
                        with mutex when mutex(capacity) > 0 do
                          mutex(capacity) := mutex(capacity) - 1,
                      proc()                          ! V
                        with mutex do
                          if mutex(capacity) < range
                            then mutex(capacity) := mutex(capacity) + 1
                            else raise AbuseException
                    )
           end
```

In the example, the resource expression returns an object of a recognizable type which is used in the following CCR statements. This is in keeping with the tradition of strong typing in the language that every expression has a type. Also, the range check is in accordance with the definition of semaphores. In particular, the second range check eliminates a potential misuse of the V operation though it still can be called inadvertently.

§3.2.3 Monitor

The invention of monitors is more-or-less a response to the need for structuring large and complex programs e.g. operating systems. Low level resources such as memory allocation must be centrally administered to prevent malicious uses while allowing concurrent accesses. In this respect, it would be a tiresome task to rely on semaphores or CCRs. A semaphore is meant to be a gateway to protected resources but it could be bypast intentionally or otherwise. CCR facilities put a fence up around protected resources but once the door is open the resources are liable to misuses. Furthermore both semaphores and CCRs are scattered all over program sources making debugging difficult.

The idea of monitor was first outlined in [Dijkstra 72] and was

further expounded in [Brinch Hansen 73, Hoare 74]. A monitor encapsulates resources and at the same time limits the ways in which they can be manipulated. This is achieved by disallowing direct accesses to the protected resources altogether. Instead they can only be accessed through procedures which are made visible. Synchronization is achieved by disallowing more than one process calling a monitor procedure.

Since the protected resources are completely hidden, it is not possible to peek at the state of the resources. But the nature of some concurrency problems such as i/o buffering requires the state of the resources to be inspected before an operation can be carried out. For this reason, condition variables are introduced to serve as signals between processes. The idea is that if the condition does not permit, the calling process is suspended and the monitor is released for the benefit of other processes. A problem in the early proposal [Hoare 74] is that a signal is not remembered so that a process may be suspended for a condition not knowing that it has already been made favourable by other processes. A taste of this can be found in the example below. The problem has been the subject of a score of proposals. These are further discussed in the two papers above and the following [Brinch Hansen 75, Rovner et al. 85, Lampson & Redell 80].

In the monitor paradigm there is no global variables. Accesses to resources are always through monitor procedure calls. This means a monitor procedure may call other monitor procedures. Since a monitor procedure call is conducted in mutual exclusion, mutually recursive calls could result in deadlocks. The problems of monitor procedure calls are further discussed in [Lister 77].

Concurrency control with monitors is quite stringent. Only one process is allowed to be inside a monitor at any time. This could prove to be rather restrictive if a large collection of small objects such as records or relations is protected by a single monitor. Unless monitors are cheap, the degree of concurrent accesses to resources is not likely to be high.

The implementation of monitors is again trivial given primitive semaphores. A binary semaphore suffices to ensure mutual exclusion to monitor procedure calls. However an implementation of condition variables requires the support of a queue for suspended processes and the two operations *wait* and *signal* for suspending the executing process and resuming an enqueued process respectively. The following is an example showing an ADT of a mutex monitor. The construction of the

ADT capitalizes on Sema defined earlier.

```
structure MutexMonitor(proc() acquire, release)

let MMonitor = proc( -> pntr)
                begin
                structure semaphore(proc() P, V)
                let notBusy = Sema(1)
                let Pp = notBusy(P)
                let Vv = notBusy(V)
                let mutex = Sema(1)
                MutexMonitor(
                    proc()
                      begin
                      mutex(P)()
                      Pp()
                      mutex(V)()
                      end,
                    proc()
                      begin
                      mutex(P)()
                      Vv()
                      mutex(V)()
                      end
                    )
                end
```

In the example, *mutex* is used to ensure that there is only one caller to either one of the two procedures. This is in accordance with the definition of monitors. The semaphore *notBusy* serves to realize a mutex being implemented here in a monitor setting. This example is for didactic purposes only and serves to illustrate the fact that it is possible to obtain some general concurrency control devices based on simple, low level primitives.

We have shown here that monitors can be constructed using semaphores and vice versa. However, the example above is unrealistically simple particularly in process scheduling. In this case, all processes calling *acquire* have to wait until *release* is called. It could be more appropriate to support explicit condition variables for the sake of program clarity and to relieve the programmers from details irrelevant to their real problems.

§3.2.4 Path Expression

The provision of synchronization is to ensure reader and writer processes do not interfere with one another. Reader processes which do not alter the state of the resources can execute in parallel whilst a writer

process always executes in isolation. The way in which synchronization is achieved in monitors is to treat readers and writers alike. They all execute in isolation. In this way, it can be said synchronization is achieved at high expenses. Monitors are useful in dealing with concurrency and in structuring large, complex programs but they may not be the ideal tool.

An alternative approach towards synchronization was put forward by Campbell and Habermann [Campbell & Habermann 74]. In their approach, synchronization is expressed explicitly. The setting is similar to that in monitors in that resources are encapsulated and they can only be manipulated through visible procedures. The order or rather the lack of order of execution of these procedures is described by a set of relations.

Ordering and the lack of ordering are captured in the two kind of relations: *sequencing* `;` and *selection* `,`. For example, "P; Q" means the execution of Q should be preceded by P and "P, Q" means either P or Q can be selected for execution but not at the same time. In addition, *association* `()` is possible so that "P; (Q, R); S" describes two possible sequences of execution viz. P then Q then S or P then R then S. Futhermore, there are *repetition* `**path** ... **end**` and *simultaneous execution* `{}`. These concepts and notations are more than sufficient to express multiple readers and single writers as in "**path** {read}, write **end**". In general, priority of execution and simultaneous execution of writer procedures which do not interfere with one another can be expressed.

Despite the attractiveness of this approach, it has not been adopted in any high level concurrent languages the author is aware of. One reason for this may be due to the fact that the state of the protected resources is completely left out. The problem is manifest in some intricate situations where there are two or more kinds of writer procedures operating on shared resources such as buffers. It is less than straightforward to find a set of path expressions that is flexible enough to allow these procedures to be called in mutual exclusion, arbitrarily and infinitely often without hitting the bounds of the buffer. The approach is a victim of its own descriptive power in that there can be no compensatory auxiliary synchronization mechanism at the same notational level.

§3.3 Interprocess Communication Mechanisms

The contribution of monitors lies in the fact that both modularity and process synchronization are achieved at the same time. There is little doubt that modularity is useful in structuring large, complex programs.

But the synchronization side is not considered to be satisfactory as it is founded upon a very strict access regime which results in less parallel execution that is logically and practically achievable. The approach of path expressions represents an effort in trying to remedy that aspect but fails to take into account the state of protected resources. It is therefore not as versatile as one may have wished. The essence of path expressions is to describe some execution relations among procedures. It was observed that this could be achieved, in part, by granting autonomy to monitors and programmers are then responsible for organizing when and which call should be accepted. Whether or not two non-interfering calls can be executed in parallel depends on the process model which will be covered in a later section.

The observation gives rise to monitor-like entities that not only encapsulate data resources but also have a separate thread of control; just like processes. This idea was first employed in Distributed Processes [Brinch Hansen 78]. Since the only means to interact with such processes is by calling their procedures, they can be characterized as communicating processes. Communication through parameter passing is generally known as message passing. A distinctive feature here is that communicating processes need not necessarily be sharing the same store, which makes distributed programming feasible. It should be pointed out that communication and synchronization are two different concepts although they are often related in this context. The distinction will be made clear later.

This section discusses interprocess communication mechanisms designed and/or implemented hitherto for a number of programming languages. Each of these has its own emphasis which gives its flavour. In keeping with the underlying themes, we highlight the main features of individual mechanisms and avoid discussion of the languages wherever possible.

§3.3.1 Distributed Processes

In Distributed Processes (DP) [Brinch Hansen 78], the structure of a process is similar to that of a monitor. It encapsulates data and provides procedures which can be called upon to manipulate the former. In addition, there is an initial statement which is executed on process start up.

Although a process has a thread of control of its own, the execution of procedures upon requests is by separate threads of control. In a sense, this is the same as a procedure being executed by whoever calls it. In DP,

each process runs on a separate processor with local storage. But there is a surrogate process for each potential caller on each processor. When a request is received, the appropriate surrogate process is called in for the execution of the called procedure. These surrogates are transparent at the language level so that process communication can be thought to be accomplished by procedure calls.

Execution of procedures need not happen immediately upon requests. This is due to the need for synchronizing the process and the surrogates. Some form of CCRs are introduced for that purpose. Synchronization is achieved as follows. On start up, the initial statement is executed. The execution either terminates or halts at a CCR statement. The surrogates which then executes is chosen non-deterministically. Again the execution either terminates or halts at a CCR statement. The process or one of the other surrogates then has a chance to run. This continues indefinitely as the processes are not meant to terminate. One limitation is that the set of processes is statically determined.

§3.3.2 CSP, Occam, Ada and Amber

The communication style in CSP is modelled upon assignments. In fact the way processes communicate is reminiscent of the execution of an assignment. There are two complimentary actions involved viz. a ? for input and a ! for output mimicking the evaluation of the expressions on the left and right hand sides of an assignment. Two processes engaged in a communication are synchronized until both are ready and a value has been exchanged.

This concept of CSP brought the semantics of communication in programming languages into focal interest. Synchronous communication as opposed to asynchronous communication has a simple and concise semantics. It has simple semantics because it does not require buffering of data and hence there is no "hidden" state associated with messages in transit, nor are there flow control problems. It has concise semantics because it is easier to reason about communicating programs as an event at one end of the communication channel triggers exactly one event at the other.

The elegance in which process communication is accomplished is a reason why CSP has such a profound influence on rendezvous communication mechanisms found in subsequent languages such as Occam, Ada and Amber.

Occam [May 83] is a language subset of CSP. It takes the rôle of an assembly language for the INMOS transputers. An INMOS transputer is a computer made up of a CPU and local memory on a single chip with eight hardwired channels for external communication. A cluster of transputers can be programmed in a variety of ways to carry out specific but concurrent computational tasks. Unlike CSP, processes in Occam do not name one another for the sake of communication. Instead they name channels which can be shared only between two processes mimicking the connectivity nature of transputers.

One of the most talked-about features of Ada is rendezvous; the process communication mechanism. It is the most comprehensive piece of language communication machinery ever designed covering as wide a spectrum as: timed and conditional calls, selective waits and extended rendezvous. Basically, rendezvous is a synchronous communication mechanism, just like that advocated originally in CSP. Processes communicate through entities known as entries. Entries are rather like procedures. They have headers and executable bodies similar to that in procedures. Messages are sent to entries where they are queued awaiting acceptance. A process can have more than one entry associated with it. A non-deterministic selection of entries is possible. When an entry is selected, the longest waiting message will be selected and the body of the entry is executed by the callee. In comparison with CSP, callers and callees are synchronized for longer than just for the sake of exchanging values. Callers wait until callees have finished execution of entry bodies. *Extended rendezvous* (p. 370 [Watt et al. 87]) allows callers to be resumed earlier than this. There is an arrangement for propagating exceptions to both processes if they are triggered during a rendezvous. The juxtaposition of all these facilities could lead to programs which are difficult to understand [Hoare 81].

The communication mechanism in Amber is very similar to that in Occam. In both cases, process communication is synchronous and is facilitated by channels. The difference between the two languages is that Occam is very close to the machine and Amber is a strongly typed high level functional language. In Amber, channels are typed first class objects and can be shared by an unspecified number of processes. Processes communicate by naming common channels and the CSP notations i.e. ? and ! are adopted which operate on channels and values whose types are restrained to that of the channels to which they are transmitted. For example, only integers can be sent down a channel of integer. It is interesting to note that channels are first class objects in the language so

that a channel can be passed down another channel. Consequently, the communication capability of a process is not a static property.

§3.3.3 Mesa

Distributed Processes is one of the first languages to have supported remote procedure call (RPC) as the process communication mechanism. RPC differs from the communication mechanisms in CSP, Occam and Amber in that communicating processes do not name one another and the communication medium is transparent. These make RPC a more general communication mechanism particularly in a distributed environment. The *modus operandi* of the mechanism in DP is rather restrictive. Processes can communicate with one another only if they are parts of a single program. This is mainly because the targets aimed at were static systems whose structures seldom altered.

The communication mechanism in Mesa [Birrell & Nelson 83] is also RPC. But the model of computation is more general than that in DP. It is based on the concept that each communicating process can be regarded as either a client or a server. In the context of communication, servers do not know the identities of clients but the latter must name the former. This allows library server packages which are not possible in CSP. Furthermore, clients and servers may be compiled separately and on different machines.

The RPC in Mesa is seen to be an extension of the ordinary procedure call mechanism. The aim was to make RPCs appealingly similar to ordinary procedure calls. Consequently, it tries to hide network and machine failures. It was believed that such an approach contributes towards simplicity in distributed programming.

§3.3.4 Argus

Argus [Liskov 85b] is derived from of CLU but with particular emphasis on distributed systems which lead to a consideration of the manipulation and preservation of long-lived data. A distributed system is modelled as a collection of entities known as guardians executing on different machines. A guardian is structured more-or-less like a monitor or a class in that data are encapsulated and can only be accessed by locally defined procedures known as handlers. Instances of guardians may be created at any site. Interaction between guardian instances is by invoking handlers which is akin to calling remote procedure.

Concurrency is inherent in the communication mechanism. Concurrency is therefore implicit. For every handler call, a process is

spawned to serve it. These processes are synchronized only if they try to access global data. Concurrency control is built into the semantics of data types.

One interesting semantic aspect of handler calls is that the calling semantics is at-most-once. This is in keeping with the view that a handler call has the rôle of a transaction; either it succeeds or not and the caller is informed. This, they claim, makes the construction of reliable distributed systems less complicated than it would otherwise be.

An interesting aspect of Argus is that although it is for programming distributed applications, there is no concept of shared stores. It supports the construction of distributed transactions by means of handler calls. Each handler call is regarded as a transaction. Handlers may call one another or be mutually recursive. A distributed transaction is depicted as a tree whose nodes are sub-transactions scattered over machines on the network. A commit decision made at the root is rippled down through the leaves thereby causing local commits. Central to this theme is that images of transactions are saved on local storage media to get around the problem of node crashes during a global commit. It is then required that objects remain where they are. Consequently, the passing of objects in handle calls results in copies installed at the receiver ends. These copies are regarded as separate objects. Consistency among copies of an object is therefore not a responsibility of the system.

It is interesting to contrast this approach with that of Linda. In [Carreiro & Gelernter 86], Linda is described as a sub-language with a small number of parallel operators. A space of ordered tuples shared by processes running on different nodes on the S/Net [Ahuja 83] underlies its communication mechanism. Primitives are provided for the injection and extraction of tuples to and from the space. The tuple space is a logical transit area for holding messages exchanged among processes. The realization of the tuple space is by requiring each node on the S/Net to keep a complete copy of the space. So the injection of a tuple into the space causes the installation of a copy in each of the nodes. Similarly the extraction of a tuple causes the deletion of its copy from each of the nodes. The consistency among copies of the tuple space on each node appears to rely heavily upon the bus nature of the S/Net and in particular the broadcast protocol used.

§3.4 Non-determinism

Dijkstra was (again) responsible for introducing non-determinism by

means of guarded commands into programming languages [Dijkstra 75]. The intention was to allow a number of (trivially) different programs to be expressed succinctly in a single program.

A guarded command is a statement made up of a Boolean guard and a list of statements. It is only when a Boolean guard is evaluated to become true that the associated list of statements gets executed. Guarded commands are rather like the if-then construct in Pascal or the if-do in PS-algol. They alone are not meant to introduce non-determinism. It comes from the alternative and repetitive constructs both of which have guarded commands as constituents.

<alternative command> ::= **if** <guarded command> {[] <guarded command>} **fi**
<repetitive command> ::= **do** <guarded command> {[] <guarded command>} **od**
<guarded command> ::= <Boolean expression> -> <statement> {; <statement>}

The if-then and the if-do constructs specify a left to right order of evaluation. The guards of the guarded commands in the alternative or the repetitive constructs are evaluated, so to say, in parallel. When a guard is evaluated to become true, the associated list of statements then has a chance to be executed. If there is more than one eligible list, an arbitrary choice is made. The difference between the alternative and the repetitive constructs is that the former will cause an error if there is no choice possible while the latter keeps evaluating until there is no choice and at which point it terminates with no effect.

An area in which non-determinism is found useful is in expressing the behaviour of communicating processes. Non-deterministic constructs allow a communicating process to be able to behave differently according to the pattern of external requests or conditions of some internal states.

However, in this connection non-determinism need not be explicit at the language level. Argus is a classic example: a flavour of non-determinism can be introduced implicitly through concurrency. A guardian, although executed like a process, does not dictate the acceptance of calls. Each call causes the creation of a new process to serve it. These processes are synchronized only when they interfere with one another.

CSP and DP were the first two languages to support non-deterministic constructs for the sake of communication. CSP adopted the alternative and repetitive constructs with minor syntactic alterations to those

above. In DP, in addition to these, there are two constructs collectively called guarded regions. These are the when and the cycle constructs. They are similar to the other two in that the only constituents are guarded commands. The when construct specifies, subject to eligibility, the selection of only one of the guarded commands and this means the executing process may have to be suspended. The cycle construct specifies, subject to eligibility, the selection of the guarded commands one at a time. In either case, the executing process skips if no selection is possible. The other languages included in this survey do not adopt all four constructs in their repertoire.

The non-deterministic behaviours of a communicating process can be captured very neatly. For example, a semaphore process can be expressed in CSP notations as follows:

```
Sema :: [ room : integer;
          room := 10;
          *[ room > 0 -> [ Any ? V -> room := room + 1
                         [] Any ? P -> room := room - 1]
           [] Any ? V -> room := room + 1
           ]
          ]
```

The Sema process accepts *P* and *V* repeatedly and the choice of which is conditional on the state of the integer variable *room* using the repetitive construct. *P* is accepted only if *room* is non-zero. On the other hand, *V* can be accepted at any time. As it is shown here, input and output guards can be used in guarded commands.

Occam and Amber both chose to support the when construct which has exactly once semantics. The executing process is blocked if no selection can be made. It is a little confusing in Occam in that the construct is called the alternative construct. This may be due to the fact that only input guards are allowed in the construct. In Amber the construct is known as the select construct. In contrast to Occam, both input and output guards are allowed and can be freely mixed in the same statement.

By and large, the select construct in Ada has a far richer semantics than the one in Amber. Its effect is equivalent to the alternative and the when constructs combined. The construct takes several forms handling committed events, non-committed or conditional events (the else part) and timing of events (delay statements) thus allowing both clients and servers the necessary freedom to backoff and/or try again later.

§3.5 Processes

Concurrency is introduced by means of processes each with a separate thread of control. Typically, the following questions can be asked:

1) Are processes explicit or implicit? If explicit, what form does a process take?
2) Can processes be dynamically created?
3) What is the relationship between parent and child processes?

Answers to these questions characterizes the process structure of a concurrency model. The discussion here does not cover Argus not because it is the only one favouring the implicit approach but because this has already been done.

Algol68 is probably the first algorithmic language to support concurrent programming. Concurrency is introduced through the use of collateral clauses or parallel clauses. There is a minor difference between them and that is the constituents of collateral clauses are a sequence of simple statements whereas constituents of parallel clauses are sequences of statements which may be synchronized by means of semaphores. Moreover, collateral clauses may yield values but not parallel clauses. The execution of a collateral clause or a parallel clause causes the creation of child processes and the suspension of the parent process. The parent process is resumed only when all child processes have terminated. This style of concurrency seems to have little influence on later language designs.

In contrast to Algol68, there is a strong distinctive flavour of modularity in processes in Concurrent Pascal [Brinch Hansen 75]. A process is a self-contained unit which has a private workspace which is initialized by local declarations with an independent thread of control. This mimics the CPU and local memory structure of a computer. A process is introduced as a template into the type system. This allows more than one instance of a process to be executed in parallel. The language construct **init** is used to spawn new processes. Thus there is an element of dynamism in the process structure. Child processes once started are independent from the parents.

As procedures or functions represent some sort of executable codes, a number of languages took a short cut in supporting concurrent programming by simply introducing a concurrent operator which acts on

procedures or functions. In Mesa, it is **FORK** and in Amber it is **process**. The purpose of such operators is to introduce a thread of control for the execution of the procedure or function in question. Since procedures or functions may produce results, it must be possible for the parent processes to synchronize with its child processes in order to obtain results. In Mesa, this is achieved with a **JOIN** operator which acts on the results of FORK and returns the result of the child process. In Amber, functions upon which processes can be launched are those that return void results and processes are expected to communicate using channels.

Tasks in Ada have similar structures to processes in Concurrent Pascal. The process structure in Ada is mostly static. A task is initiated at the point where it is declared and reached during execution. However, a task can be declared as one of the access types thus almost achieving the same degree of flexibility as in Concurrent Pascal in allowing more than one instance of the same task to be executed in parallel. Hence the number of tasks is not statically determinable. Child processes once started, execute in parallel with the parent processes. However, parent processes do not terminate until all child processes have terminated. This is mainly because child processes may reference variables in the parent's environment.

§3.6 Conclusions

Essential to a concurrency model are processes, synchronization mechanisms and/or communication mechanisms. There are variations on each one of these. For instance, whether processes should be explicit or implicit or both. Should synchronization be built into the communication mechanism and not supported separately? In view of the store semantics of PS-algol and the desire for distribution, we come to the following observations.

We believe processes should be explicit so that programmers are aware of where concurrency is introduced. It is intended that processes can be used as building blocks in constructing large, possibly distributed, programs.

We believe synchronization and communication are separate issues. On the other hand, it is not denied that synchronized communication is attractive. The separation offers room to accommodate different programming styles. Processes may be synchronized without appealing to the communication mechanism. This is true even if they do not share the same store as we are developing a shared store technology.

We also believe processes should be dynamic so that programmers are allowed to specify when concurrency is to be introduced. The combination of explicit processes and dynamic process structure is believed to enhance the expressive power of a language given that communication is synchronous [Liskov 85a].

In facilitating the construction of distributed systems, we believe a remote procedure call mechanism is the desirable communication mechanism. This should enable processes executing on different machines to cooperate with one another in the most natural way. It also permits the components of the system to evolve and change as it is an interface which supports delayed type checked binding.

The successive improvements in synchronization mechanisms are motivated by the desire for structured programming. We believe this can be achieved through other means e.g. higher order functions. The drawback of monitors is in their strict access regime resulting in a lower degree of concurrency in some cases. We intend to support the more flexible mutexes for mutual exclusion and condition variables for process scheduling so that sophisticated synchronization mechanisms such as monitors can be realized. It remains a challenge in the language design to minimize problems such as incorrect use of mutexes, by high level constructs which encourage the use of well defined simple cases in most instances.

Chapter 4

Language Design

§4.1 Introduction

A language based on the distributed stores outlined in chapter 1 is described here. The technologies required in realizing such stores will be described in subsequent chapters. The principal features of the language are distribution and concurrency. Concurrency is admitted through lightweight processes which are computational tasks that share the same address space. Distribution is supported by allowing pointers to remote, as well as, local data. The effect of manipulating both remote and local data is the same giving the illusion of shared address spaces. There is a language concept of *locality* to facilitate specific coding of distributed algorithm to give higher performance computations when necessary. The language still allows programs to be written according to the principle of locality transparency. Process communication comes in two varieties: store-based and message-based. Processes may interact across localities using a remote procedure call mechanism (chapter 5). On the other hand, they may also interact with one another through global environments which may be on remote stores. Process synchronization over accesses to shared environments may be achieved using concurrency control primitives provided. In addition to these, the language has a notion of non-determinism and supports separate compilation of communicating programs. The latter is characterized by the signature matching algorithm so that changes in one software component has minimal impact on its dependencies thus reducing the amount of re-compilation necessary.

The language is a descendant of PS-algol, from which, persistence, higher order functions and other features of PS-algol are inherited.

§4.2 PS-algol

The new language is designed based on PS-algol. A brief introduction to the syntax and semantics of PS-algol is given in this section for readers not familiar with the language. PS-algol is described in detail in [PPRG 87a].

PS-algol is one of the first algorithmic languages in which the concept of persistence is supported. Persistence is defined to be the length of time for which data exist and are usable. It is a property of data

independent of its type. Any value in the language can be retained in secure storage media known as persistent stores for an arbitrary period. The notion of a store is much more general compared with other conventional languages and is fundamental to the semantics of the language. A PS-algol program is executed in a persistent store so that persistent data are always available. Moreover, there is no distinction in accessing, say, data created at the inner most block of a recursive evaluation, and data in a persistent store. This is in contrast with other approaches to long term data where the manipulation of such data is facilitated by an embedded data manipulation language. Because of the generality on the concept of store and its implications on long term data, PS-algol is a very attractive tool in database applications.

Apart from persistence, the language is a powerful, general purpose, conventional language in its own right.

It is a strongly typed language. The type system is sound in the sense that it does not leave any loophole in it and every expression carries a type known to the type checker (in contrast to, say, variant records in Pascal). Type checking is performed mostly at compile time. Occasionally, type checking is also performed at runtime. The execution of a PS-algol program therefore may not be as efficient as, say, an ML program which is always statically typechecked. But runtime typechecking is deemed necessary in PS-algol because of persistence [Atkinson et al. 88]. In general, a type error is always reported at the earliest possible opportunity. The type system admits a rather limited form of (universal) polymorphism. This is achieved with the use of the data type `pntr` which represents the union of all labelled cross products. Examples of the use of `pntr` can be found in [Cooper 87].

The language is designed based on the principle of data type completeness; values of all types receive equal treatment as parameters to procedures, as results returning from procedures, as fields of structures, as elements of vectors etc. The result is a powerful yet simple to learn language.

The language supports higher order functions. They enjoy the same rights as integers, reals, boolean etc. In particular, functions can be returned as results of functions hence the term higher order functions. Higher order functions are very useful as a programming tool. A conventional language equipped with higher order functions may provide programmers with a richer set of programming utilities than

otherwise. In the case of PS-algol, for instance, higher order functions together with structures and persistence permit the construction of abstract data types and modules without any language support [Atkinson & Morrison 85a]. Examples on these concepts can be found in chapter 2.

The language has built-in graphic facilities in line with the wide availability of bitmap graphic terminals. Displaying graphical images is generally accepted to be an efficient way of communicating data between programs and users. Such facilities permit the construction of, for example, graphical interfaces to databases, editors and window manager systems.

Flanked by a callable compiler, PS-algol can be considered as a total language in the sense that activities such as editing, linking, compiling and storing of long term data can all be done without resort to mechanisms outwith the language, e.g. operating systems.

§4.2.1 BNF Syntax and Type Matching Rules

The syntax rules of PS-algol specify a context-free grammar which is then restricted by a set of type rules. For the remaining part of this chapter, the shadow font style is used for the language types and the generic types, which are a meta-physical concept, to avoid confusion with some of the key words (shown in bold) in the language.

The number of data types in PS-algol is infinite. They can be defined recursively by the following rules.

1) The primitive data types are integer, real, boolean, picture, pixel, file and string.
2) #pixel is the type of an image made up of pixels arranged as a rectangular matrix.
3) For any data type T, *T is the data type of a vector with elements of type T.
4) The data type pointer comprises a structure with any number of fields, and each field consists of a binding of a name and a value of any type.
5) For any data types T_1,..., T_n and T, proc(T_1,..., T_n -> T) is the data type of a procedure taking parameters of type T_1 to T_n and producing a result of type T. The type of a similar result-less procedure is proc(T_1,..., T_n).

In addition to these data types there are other objects in PS-algol to which it is convenient to give a type in order that the compiler may check their use for consistency.

6) Clauses which yield no value are of type void.

7) The class of a structure with fields of type T_1,..., T_n is of type (T_1,..., T_n)-structure and its fields are of type T_i-field.

The world of data types in PS-algol can be categorized. They are useful in the understanding of the enforcement of syntax rules by the typechecker in screening out some semantically meaningless programs. Generic types are used to identify type categories. Constancy of value is expressed at the level of type and is indicated by the letter 'c' preceding a type expression.

```
type arith        is  int | real
type comparable   is  arith | string
type printable    is  comparable | bool | pixel
type literal      is  printable | pntr | proc | file
type image        is  #pixel | #cpixel
type nonvoid      is  literal | image | *nonvoid
type type         is  nonvoid | void
```

In the following syntactic rules syntactic categories, such as <identifier> that are obvious or the omission of which do not jeopardize the understanding of other rules, are automatically dropped from further discussion.

The structure of a PS-algol program is governed by the following rules expressed in BNF notations in which [...] denotes zero or more occurrence.

```
<program> ::= <sequence>?
<sequence> ::= <declaration> [; <sequence>] | <clause> [; <sequence>]
```

A PS-algol program is a sequence of declarations and clauses. Unlike Cobol, Algol60, Algol68, Pascal and Ada, declarations and clauses in PS-algol can be mixed freely. The advantage of this is that a declaration can be made where the binding is first used and the initializing expression can be evaluated, improving lexical locality.

```
<declaration> ::= <let_decl> | <structure_decl> | <handler>

<let_decl> ::= let <identifier> <init_op> <clause>
<init_op> ::= = | :=

<structure_decl> ::= structure <identifier>[(<field_list>)]

<handler> ::= when <ex_id_list> [as <identifier>] do <clause>
<ex_id_list> ::= any | <identifier_list>
```

A declaration permits the introduction of either a binding of a type-value pair, a labelled cross product or an exception handler. In the case of a binding, the type of the value is always inferred by the typechecker. A binding can be a value binding or a location binding depending on whether '=' or ':=' was used. A value binding means the identifier always denotes the same value and a location binding allows the denotable value to be changed. A structure declaration does not introduce any value. It is a means of introducing a labelled cross product type into the type system. A declaration can introduce an exception handler for the enclosing blocks. The terminal **any** is for use as a catch-all exception identifier.

Types are inferred whenever possible. Type inference is not restricted to base types whose literals are known to the typechecker. It also applies to some compound types whose data constructors are defined in the language. Data constructors do not introduce new types.

The set of clauses in PS-algol is defined by the following syntactic rule:

<clause> ::= **if** <clause> **do** <clause> |
if <clause> **then** <clause> **else** <clause> |
repeat <clause> **while** <clause> [**do** <clause>] |
while <clause> **do** <clause> |
for <identifier> = <clause> **to** <clause> [**by** <clause>] **do** <clause> |
raise <identifier> [(<clause_list>)] |
case <clause> **of** <case_list> **default**: <clause> |
<raster> | <print> | <write> |
<name> := <clause> | **abort** | <expression>

The conditionals, the unbound recursion operator **while**, the bound recursion operator **for** and the **case** constructs are fairly familiar. The more interesting ones include raster operations, exceptions and the device independent **print** construct.

§4.3 Distributed PS-algol

Distributed PS-algol (DPS-algol), or DPS for short is a descendant of PS-algol. All features of PS-algol are retained so that a PS-algol program is a legal program in the language. The converse is not true. The object in designing the language has been to facilitate concurrent programming and at the same time admit a flavour of distribution. The design is influenced by the desire to retain the flavour of PS-algol. Because of this, the basic principles of PS-algol are observed. The simplicity and generality of PS-algol proved to be a constant source of inspiration as well

as a guidance in the process. Many of the new features are more general than when first conceived. For instance, a communication port can be shared, though in a controlled fashion, by more than one process. A server process may thus delegate its functions to other processes in a neat manner and without client processes being aware of it.

In the language description below, whenever a term with an unusual meaning is introduced a different font is used i.e. thisFont.

§4.3.1 Entry

Processes can affect one another either by modifying global variables or by message passing. In general, message passing is the more attractive mechanism of the two in the context of distribution since it does not assume common stores. With it, messages can be exchanged through ports which can be addressed from any process anywhere over a network. Examples on the use of communication ports in process communication can be found in DP, CSP, Ada etc. The approach taken here represents yet another one although it is rather close to that in Ada. The differences will become clear later.

An entry is a bi-directional communication port. It is where a process receives messages and replies are sent. An entry can be constructed based on the following syntactic rules:

<entry> ::= **entry**([named_param_list] [<arrow> <type>]); <proc_clause>

<named_param_list> ::= <proc_param_type> [; <named_param_list>]
<proc_param_type> ::= <type1> <identifier_list> | <structure_decl>
<proc_clause> ::= <clause> | **nullproc**

The type rule for the syntactic category <entry> is simple.

t : **type**, **entry**([<named_param_list] [<arrow> <type> : **t**]); <clause> : **t**
=> **entry**(t_1, ..., t_n -> **t**)

The interpretation of this type rule is as follows. If <arrow> <type> are specified, then the return type can be any nonvoid type t and the type of the entry body must be t as well. Otherwise the type of the entry body must be of type void. The resultant type is an entry type. Entry types are akin to procedure types and they share the same type matching rule. However, they are not compatible to one another. In particular a procedure can never be assigned to an entry or vice versa.

```
let anEntry := entry( -> int); 3
```

anEntry in the example above denotes a communication port which, when a null message is received and accepted, returns an integer 3.

Entry variables and constants can be declared whenever and wherever they are needed just like any other values in the language but the entry literals may only appear in the scope of the **process** construct. For the reason of modularity entries then belong to one or a group of processes within whose scope they were created. Such modularity is necessary since it is our desire to allow processes to be as independent as possible. This permits their distribution to be straightforward both from the programming and implementation point of view.

There is no limit as to the number of entries a process can associate with. An entry may take on different values. For example,

```
anEntry := entry( -> int); 103
anEntry := x
```

As in the case of procedures, such assignments are possible only if the types on the left- and right-hand sides of the assignment match.

In essence, an entry resembles a procedure. It has a procedure-like header and an executable body. Like procedures, entries are first class values so that they can be shared. The primary reason for introducing entries is because procedures are abstraction over either statements or expressions and entries are for synchronous communication. They serve different purposes. Processes communicate using entries and are synchronized until the executions of the entries have been completed. This contrasts with procedure calls where callers are not suspended. One can argue, as a consequence of such semantics, that the degree of parallelism attainable with procedures in process communication can be higher. However, this can be achieved in the language. An example can be found in chapter 8.

§4.3.2 Process Template and Start

Concurrency is introduced by means of lightweight processes [Doeppner 86, Rovner et al. 85]. Lightweight processes are those that share the same set of resources such as the heap and the i/o streams. However, a lightweight process is considered to be an independent activity in carrying out a specific computation task. A lightweight process is introduced based on a process template. There is a hint of the concept of

class in process templates. A process template declaration results in a binding to a process template. A process template is a passive object. Such an arrangement permits a number of identical but distinct processes to operate in parallel. In this way, a concurrent program may appear to be more concise than otherwise possible. Moreover, concurrency can be introduced when and where it is needed. A process template has the following form:

<process_template> ::= **process** [**with** <signature_list>]
begin <clause> **end**
<signature_list> ::= <identifier> <init_op> <entry_type>
[, <signature_list>]
<init_op> ::= = | :=
<entry_type> ::= **entry**([<proc_param_list>] [<arrow> <type>]
<proc_param_list> ::= <proc_param_type> [, <proc_param_list>]
<proc_param_type> ::= <type1> | <structure_decl>

A process template has an optional specification header and an executable body. The specification header is made up of a list of labelled entry types. This list is known as the signature of the process template. The behaviour of a process is specified partially by the signature of its process template. The presence of a signature indicates a willingness to communicate with other processes. In the absence of a signature, a process template mimics a PS-algol program.

The type rule for <process_template> can be specified as follows,

process [**with** <signature_list>] **begin** <clause> : **void end**
=> **process**($id_1 : t_1$, ..., $id_n : t_n$)

The executable body must have the type void. The resultant type is a compound data type **process**($id_1 : t_1$, ..., $id_n : t_n$) where id_1, ..., id_n are identifiers and t_1, ..., t_n are entry types. The type matching rule is the same as with procedure and entry types. The signature specifies a collection of entries some of which a process instance may listen to during the course of execution. Each of these entries must be defined in the body. The compiler verifies this.

A process template may reference data in an outer environment. This allows a very efficient and neat way of communicating data among concurrent processes which may or may not be sharing the same address space. On the other hand, data local to a process template cannot be shared. This allows a process to have overall control of its data.

Process creation is dynamic. The number of lightweight processes that can be made to run is not statically determinable because of conditional clauses, etc. As such the language does not impose an upper bound on the number of lightweight processes that can be created. The actual upper bound is determined by the availability of memory which is a runtime property. The syntax to spawn a process is:

<process_handle> ::= **start** <clause> [**as** <clause> [**at** <clause>]]

The type rule for <process_handle> is:

start <clause> : **process**($id_1 : t_1, \ldots, id_n : t_n$) [**as** <clause> : **string**
[**at** <clause> : **string**]]
=> **ph**($id_1 : t_1, \ldots, id_n : t_n$)

If the type of the left-most clause is **process**($id_1 : t_1, \ldots, id_n : t_n$) the resultant type is then a new compound data type **ph**($id_1 : t_1, \ldots, id_n : t_n$). In other words, the signature of a process template is inherited by its process instances. Values of the type **ph** are known as process handles. The type matching rule for process handles is unusual and is based on a notion of inclusion. This can be illustrated as follows:

let p1 := p2

If *p1* is of some process handle type then *p2* must also be of some process handle type. In addition, the signature of *p2* must be a superset of that of *p1*. The idea is that an assignment of process handles such as the one above is allowed if and only if there is an enrichment of the signature concerned so that the new process handle can perform all that was expected of the previous value.

The second clause in a **start** expression is taken to be the symbolic name of the lightweight process created. The lightweight process created is registered with this symbolic name at the locality specified by the third clause. By default, it is taken to be the local one. A registration is required if the process is to receive messages from processes outwith the address space it is executing. In the absence of the second (and therefore the third) clause, no registration will take place and the process created can only communicate with other processes in the same address space. In any case, the lightweight process is made to run wherever it happens to be. It could be in the local address space or on a remote machine. In general, a process can be created locally or remotely with or without registration which

could be on any machine. If for any reason a registration cannot succeed e.g. name clashes or machine failure, a system event is generated.

A DPS program consists of at least a top-level process which is made to run implicitly. All lightweight processes are children of it. A PS-algol program would simply be run as a single top-level process. A DPS program terminates when all processes have terminated. But the termination of parent and child processes is not conditional on one another. A number of separate DPS programs may interact as explained below.

The result of a **start** expression is a process handle. Process handles are immutable. They are used in denoting the target process in process communication.

§4.3.3 Communication

In order to facilitate process communication, the language supports a remote procedure call (RPC) mechanism. Such a mechanism allows processes to interact with one another wherever they happen to be. The use of the mechanism is not restricted to processes that are compiled together. They can be separately compiled and this could happen on remote machines. However, from a programmer's point of view, it makes no difference in communicating with a local or a remote process. In either case, the syntax is the same. The only notable differences are slower responses and possible machine or network failures in communication over a network.

An RPC is a synchronous activity. Processes which initiate RPCs are always suspended. They are resumed when some processes indicate a willingness to serve calls and the execution of entry bodies have terminated. There is no guarantee that suspended processes will ever be resumed since the entries they are calling may be ignored or the execution of an entry body may go into an infinite loop. However, if there is machine or network failures or premature process terminations, system events are raised and suspended processes concerned are resumed in order to handle them.

§4.3.3.1 Remote Procedure Call

The syntax for initiating an RPC is:

<rpc_call> ::= <clause> @ <clause> ([<clause_list>])
<clause_list> ::= <clause> [, <clause_list>]

The type rule for <rpc_call> is:

<clause> : **ph** @ <clause> : **entry** ([<clause_list>]) => **type**

where the type rule for <clause_list> is:

<clause> : **nonvoid** [, <clause_list>]

In the <rpc_call> category, the first clause must be of some process handle type. The second clause must be of an entry type which must correspond to a labelled entry type of the signature of the process handle denoted by the first clause. Typing of parameters is exactly the same as in procedure calls. The resultant type of an RPC is the return type of the entry or void.

The syntax for initiating an RPC is like a procedure call except that in addition a process has to be specified. It is convenient to call that process the server of the RPC and the executing process the client. The fact that the server may be a remote process is of no concern to the programmers. This is the basis for syntactic uniformity. Such syntactic uniformity is a convenience in distributed systems where the locality of a communicant may either vary over a period of time or simply be immaterial. We describe later how process handles to remote processes can be introduced into the environment.

The semantics of parameter passing is pass-by-value; the same as in procedure calls. Pass-by-value is different from pass-by-copy. In the latter case objects are always copied. Consequently, objects are considered to be bound to the locations where they were created. This we consider a hindrance in distributed programming. The main reason is that the propagation of side-effects, which is fundamental in conventional programming, has to be the responsibility of the programmers. Primarily we wanted a programming style that is consistent with persistent programming. This was achieved in PS-algol in which the store semantics enables a blackboard view (-- many processes or blocks of code determined by the scope rules may see a part of the store/blackboard simultaneously) over different types of physical storage. This blackboard view of the underlying object stores is extended in the language. Of course, even on a single processor machine the simultaneity is not realizable due to bus contention and store arbiters. In a distributed system the approximation to simultaneity is less achievable.

§4.3.3.2 Accept

The occurrence of an RPC is in part initiated by a client and in part by the willingness shown by a server. A server expresses its willingness to communicate by accepting calls from its entries. The syntax for accepting calls is:

<accept > ::= **accept** <clause> [|| <clause>]* [**otherwise** <clause>]

The type rule for <accept> is:

accept <clause> : **entry**(...) [|| <clause> : **entry**(...)]*
[**otherwise** <clause> : **void**] => **void**

The first and all the clauses, if any, following the '||' symbol must be of some entry types. The clause following the terminal **otherwise** must be of type void. The resultant type is void.

The semantics of the **accept** construct is less than straightforward and is given informally here in English. All the entry clauses are evaluated in parallel. An entry is said to be eligible for selection if it has a message. If there is more than one eligible entry, a non-deterministic choice will be made. The longest waiting message of the chosen entry is selected. The content of the message is pushed onto the stacks and the entry body is entered as if the server is making a procedure call. If a choice cannot be made and there is no **otherwise** part, the current process is suspended until a message for any one of its entries has arrived when the evaluation of the **accept** clause is repeated. If there is an **otherwise** part, it is executed and the process continues.

The **accept** construct is the only means whereby non-determinism can be introduced. It is considered to be less general than other non-deterministic constructs in the literature in the sense that the only constituents allowed in the construct are entry clauses rather than guarded input and output commands. Apart from the consequence in the asymmetric treatment in communication of clients and servers, the construct is no less powerful than the alternative, repetitive, when and cycle constructs reported in the literature (see §3.4). The effects of these constructs can be simulated with the **accept** construct combined with the loop construct. Allowing a client to retract its call after the lapse of a certain period of time requires a global clock which is difficult to maintain on a distributed system. Some sort of timeout effect can be obtained using a local clock process. We supported a primitive non-

deterministic construct here in keeping with the simplicity tradition of PS-algol.

§4.3.4 Concurrency Control and Scheduling

The organization of the persistent store, which underlies the language, is in the form of a graph whose nodes are collections of local and remote data. Although data are guaranteed to be transactionally secured or stabilised in a persistent store, accesses to nodes require coordination to guarantee atomic changes. Primitive concurrency control is supported in the language. Our approach is oriented towards optimism. We do not wish to impose a strict access regime so that the degree of concurrency attainable may not be uniform. It is allowed to vary according to the nature of applications.

§4.3.4.1 Mutex and Lock

A simple data type and a construct are introduced to facilitate concurrency control. It is intended that concurrency control is achieved based on conventions and it is the responsibility of the programmers to observe the conventions they chose. Because of higher order functions and persistence, it is believed that methods of accessing data can be packaged and users only need to know about interfaces.

The new data type is **mutex** and the only value of this type is the literal mutex. A mutex is generally known as a binary semaphore.

```
let x = mutex
```

In the declaration above, *x* is introduced as an object of type **cmutex**. As with other values in the language, mutex objects are first class. A mutex object is used in a construct whose syntax is:

```
<lock> ::= lock <clause> [, <clause>] do <clause>
```

The type rule for <lock> is:

```
lock <clause> : mutex [, <clause> : mutex] do <clause> : void
        => void
```

The right-most clause must be of type void and other clauses must be of type mutex. The resultant type is void.

The evaluation of the right-most clause in a **lock** statement is conditional upon the seizure of all the mutexes specified. The evaluation

of the mutexes follows a left to right order of evaluation. The executing process is suspended if any one of the mutexes cannot be seized. In this case, all the mutexes acquired hitherto are released. Subsequently, the release of any one of these mutexes triggers the re-evaluation of the **lock** statement. Upon completion of the right-most clause, all the mutexes are released. Any abnormal exit during the evaluation of the right-most clause, such as exceptions, will cause the release of all mutexes acquired.

If mutexes are always acquired in the same order (when in different nested **lock** clauses), the **lock** clause is useful in avoiding deadlocks. Note that a mutex may appear more than once in a **lock** clause but the effect is the same as if it appears once; no deadlock will result. Note also that the following two programs are not equivalent.

```
lock m1, m2 do S
```

and

```
lock m1 do
  lock m2 do S
```

The first program guarantees that *m1* and *m2* are seized simultaneously whereas the second does not. In the second program, if *m2* cannot be seized, *m1* is not released.

The familiar P and V operations are not supported in the language. Instead they are built into the semantics of the **lock** construct. They are not supported in the language because their uses are often subject to abuses. Since the two operations can be used in isolation, a process can force its way into a critical region by executing a V operation. Moreover, their erroneous uses can affect the proper working of other processes e.g. a matching pair of P and V operations are applied onto different mutex objects. The provision of the **lock** construct is intended to eliminate these problems. This is ensured in that 1) the P and V operations are always used in pairs and 2) a pair of these operations always acts on the same mutex object. Although these two rules can be ensured by the compiler, they can easily be by-passed by variable renaming so that supporting the P and V operations at the language level remains problematic.

The provision of only the **lock** construct does prevent some intricate use of mutexes. For instance, it is not possible to construct general semaphores which require isolated uses of the P and V operations as shown in the example below.

```
structure semaphore(proc() acquire, release)

let makeSema = proc(int room -> pntr)
                begin
                let cond = mutex
                let atomic = mutex
                let wait = proc()
                           begin
                           P(cond)
                           P(cond)
                           V(cond)
                           end
                let signal = V
                semaphore(
                              proc()
                              begin
                              if room = 0 do wait()
                              P(atomic)
                              room := room - 1
                              V(atomic)
                              end,

                              proc()
                              begin
                              P(atomic)
                              room := room + 1
                              signal(cond)
                              V(atomic)
                              end
                          )
                end
```

In the example, two different mutexes are used for different purposes. One is to ensure atomic changes to information concerning the number of processes allowed in their critical sections; the other is for blocking processes, if the limit on processes admitted has been reached, until some of the latter exits from its critical section. Blocking and resumption of processes are achieved by the procedures *wait* and *signal*. Callers of *wait* are suspended and this is ensured by two calls of P on the same mutex object. One of those suspended callers is resumed when some process calls *signal* which effectively provides an extra V to nullify the effects of the two P's. The solution does not always work, however. The subtlety here being that there is no guarantee that a process resumed by a signal will be first to execute a P on the mutex object *atomic* before others. After signal is called, *room* is greater than 0 and therefore any process could seize *atomic* without calling the procedure *wait*. A notion of priority is required here. The provision of the simple data type **cond** and two

operations aims to alleviate situations like this. Should programming experience show that direct operations on mutexes are desirable then it would not be difficult to introduce them.

§4.3.4.2 Condition, Wait and Signal

The set of base types in the language is enriched with the data type **cond**. The only value of this type is the literal **cond**. An object of this type can be introduced into the environment as in:

```
let x = cond
```

Objects of this type are first class. A cond object is intended to be used to signal events e.g. changes of state of some resource. In addition to equality, there are two operators for cond objects viz. **wait** and **signal**.

<wait> ::= **wait** <clause> : **cond** [, <clause> : **cond**] => **void**
<signal> ::= **signal** <clause> : **cond** [, <clause> : **cond**] => **void**

The specification here is similar to that in Modula-2+ [Rovner et al. 85].

The execution of a **wait** clause may cause the suspension of the executing process if no signal has been received by any one of the cond objects in the list. When this happens, all the mutexes, if any, of the inner most block are released. The process will only be resumed after a signal on the appropriate cond and all the released mutexes have been re-acquired.

The execution of a **signal** clause may cause the resumption of one or more suspended processes in the near future; there is no processor switch. A process suspended due to a **wait** has higher priority in acquiring mutexes it may have released. A **signal** is remembered if no process can be resumed immediately.

```
structure semaphore(proc() acquire, release)

let makeSema = proc(int room -> pntr)
                    begin
                    let notFull = cond
                    let atomic = mutex
                    semaphore(
                                proc()
                                lock atomic do
                                begin
                                if room = 0 do wait notFull
                                room := room - 1
```

```
                    end,

                    proc()
                    lock atomic do
                    begin
                    room := room + 1
                    signal notFull
                    end
                )
end
```

Because we have shared store semantics and access to remote entries, it would be possible to define a procedure "cond" which had in its block a mutex and a list of processes which yielded a pair of entries, signal and wait. This would have nearly the same semantics -- the difference being that the release of other mutexes and the higher priority restart would not apply. These differences combined with the programmer convenience justify the introduction of **cond** as a primitive type in DPS.

§4.3.4.3 Stop and Kill

Two operators are included to cause termination of some process. They are useful in discarding useless processes.

<termination> ::= **stop** | **kill** <clause>

The type rule for <termination> is:

stop | **kill** <clause> : **ph** <lsb> ... <rsb> => **void**

The operand to **kill** must be of some process handle type. The resultant type is void.

stop causes the executing process to terminate. **kill** is more general and it causes the termination of the process denoted by the operand. If the process has already terminated, it has no effect.

If there are outstanding messages when a process is terminated, an exception is propagated to each of the client processes concerned. Similarly, RPC communication with terminated processes will cause an exception to be propagated to the communicating processes. In general, **stop** and **kill** are dangerous operators and should be used with considerable care.

§4.3.5 Locality

The main concern in the design of the language is to hide away distribution as much as possible. The rationale for doing this is to keep in line with the style of persistent programming in which locality is never a programming concern. This is achieved in the language by supporting uniform access to both local and remote data. The same syntactic constructs can then be used on both local and remote data. For example, it is possible to start a process running on a remote machine using the same construct to spawn a local process as in:

```
let remoteProcess = start remoteProcessTemplate
```

where *remoteProcessTemplate* is a process template on a remote machine.

The lack of a notion of locality leads to a programming style in which distributed programming can be no more difficult than programming in conventional languages. This is certainly desirable as programmers are not distracted by distribution. Besides being conventional, our language also has a notion of persistence. This leads to a realm where distributed database systems can be realized in much the same way as centralized database systems have been [Hepp 83, Kulkarni 83]. Distributed stores are accessed in the same way a central store is accessed. In other words, distributed stores appear to be logically centralized. The crux of the matter here is that some interesting aspects of distribution are not addressed e.g. resilience and resource utilization. Thus the application domain of the language is not as rich as that which can be achieved. We want to introduce a notion of locality into the language but only in an unobtrusive manner. Thus programmers need not know the whereabout of data unless they wish to program an explicit version of resilience and resource utilization.

Two new base types are introduced. These are **loca** and **node**. They belong to the generic type **locality**. A node is a space where lightweight processes may be started. A loca denotes a collection of nodes and other locas. It is the intention that a loca mimics a machine or a network representing a distinctive set of resources such as persistent stores, processors or devices. There are a number of primordial loca values defined in the language. From time-to-time, new primordial loca values may be created or changed without logical impact on existing programs or data. Node corresponds to a single address space and loca are an abstraction over these. From time-to-time we expect new loca to appear as machines are inserted (by means outside the scope of a programming language). Although this may have no impact on the semantics of

programs, mechanisms must exist to a) allow such new locas to be discovered, and b) for nodes to be withdrawn.

The notion of locality is a relative one so that loca values that are not referenced and are not primordial (i.e. is in a 1 to 1 relationship with a loca) cease to exist. Consequently, nodes belonging to discarded loca values may become unreachable.

A loca value can be constructed or discovered. The syntax is:

<loca> ::= **newlocality** | **locality** <clause>

The clause following the terminal **locality** must be of type nonvoid. The resultant type is loca.

A hierarchical structure of localities can be composed with the following construct.

<localities> ::= **add** <clause> : **locality to** <clause> : **loca**

The result of the construct is void. In addition to those language defined loca values, there are two distinct points in the hierarchical structure viz. **universe** which refers to the root and **here** which refers to the locality of the executing process.

As an example, **locality** 4 returns **here**. In general, immutable values exhibit the same characteristic.

Three relational infix operators are defined for loca values. These are:

[equality]	$l_1 = l_2$
[in]	l_1 **in** l_2
[within]	l_1 **within** l_2

Two loca values are said to be equal if and only if their denotations are the same. A loca is said to be in another if and only if the former is a member of the collection of the latter. A loca is said to be within another if and only if there exists a sequence of loca values $l_i, l_j, \ldots, l_n$ such that l_1 **in** l_i & l_i **in** l_j & ... & ... **in** l_2.

As an example, for any l of the generic type **locality**, l **within universe** always returns **true**.

A node value can be created or discovered. The syntax is:

<node> ::= **newnode** [**in** <clause>] | **node** <clause>

The clause after the terminal **in** must be of type loca. The other clause must be of type nonvoid. The resultant type is node.

A node is always created in some locality; the default is **here**. A node can be in one or more localities at the same time. We assume that the operation **newnode** either follows some external action introducing a machine/address space or that action in some way happens at the same time. Node operations refer to actions external to the language and are part of its relationship with its environment whereas loca operations are defined and implemented within the language. **newnode** may also declare a new loca which then matches the space correspond to the node.

The only relational operator defined for node values is equality. Two nodes are said to be equal if and only if their denotations are the same. Furthermore, a node value may appear as the left expression in **in** and **within**, the two relational operators for loca values.

Since we now have a proper notion of locality in the language, the type rule for <process_handle> can be augmented as follows:

start <clause> : **process**(...) [**as** <clause> : **string**
[**at** <clause> : **loca**]] => **ph**(...)

The semantics of this construct remains unchanged. But if the locality specified is not in the hierarchy, the registration will fail.

§4.3.6 Transcopy and Assign

Two store-to-store operations are supported for the atomic physical transfer of data between localities.

The operation for copying a piece of data to a locality has the following syntax:

<copy> ::= **transcopy** <clause> [**to** <clause>]

The type rule for <copy> is:

t : type, transcopy <clause> : **t [to** <clause> : **locality]** => **t**

where t belongs to the generic type **type**. The resultant type is t.

The polymorphic operation initiates the transfer of a copy of the data specified in the left-most clause to the designated node or loca specified in the right-most clause. The default locality is **here**.

The term **transcopy** suggests a notion of atomicity and is distinguishable from the graphic construct **copy** which is used to update an image by another. The **transcopy** operation fails if for any reason a copy cannot be installed in the locality indicated. In this case, an exception is raised. If the destination is a loca no assumption can be made as to which node has the copy installed.

The amount of data transferred is determined by the type of the expression to be copied.

1) Values of the base types are immutable. Some of them are transferred as if they were declared in the remote stores. These include integer, real, boolean, picture, pixel, and string. For loca and node, only pointers to them are copied. For mutex and cond, suspended processes are not copied.
2) Process handles are transferred as immutable values. Copies resulting from such transfer are handles to processes on remote machines. The referred processes and their entries are not copied.
3) Images are rectangular matrix of pixels and they are transferred in their entirety.
4) Vectors are always transferred with their top-levels installed in the remote stores. For example, transfer of ***int** could result in a vector of integers installed whereas transfer of ****int** could result in a vector of ***int** installed.
5) Procedures, entries and process templates are copied in such a way so that their environments remain at the original site and are sharable.
6) Structures are copied in a similar fashion to vectors.

The rules above are meant to avoid phantom copying. In particular, there is no danger of copying an entire persistent store without programmers being aware of it. Copying of data structures several levels deep can be achieved by copying individual levels.

The **transcopy** operation preserves the properties of the objects copied. The effects of the same sequence of operations when applied on copies and their originals should be identical. However, it is the

responsibility of the programmers not to destroy circularity and sharing during the copying process.

There is a complimentary operation to **transcopy**. The syntax for this is:

<assign> ::= **assign** <clause> **to** <clause>

The two clauses must be of the same type t where t belongs to the more restrictive generic type **mutable**. This is because the meaning of, for example, **assign** 3 **to** 4 is dubious. The resultant type is void.

The semantics of the construct is straightforward. It resets the value of the second clause to the value denoted by the first clause. The effect of **assign** is the same as **transcopy**; only the top level of an object is copied.

The two values in an **assign** statement must occupy the same amount of space in their immediate denotations. This means, for instance, the bounds of the first dimension of two multi-dimension vectors in a store-to-store assignment must conform to each other. The rest are immaterial. Access exceptions will be raised if objects no longer conform to the sizes expected. Replacing the first dimension vector altogether would require updating outstanding references on all process stacks. This is not possible in a distributed environment. If for any reason the operation fails, an exception is raised. The effect of such assignment takes place in the locality of the value of the second clause.

§4.3.7 Miscellaneous

In the context of communication, processes are either clients or servers. It is not suggested that a process cannot be both but only one at a time. In order that clients can communicate with servers, binding and typechecking must be resolved. Our approach to these matters is a static one but augmented with runtime support. The language does not specify any order of compilation of clients and servers. They can be compiled whenever and wherever it is deemed necessary. The rationale for this is because we do not have, nor do we think it is appropriate to have, the facilities outwith the system to maintain information on types, names and locations of clients and servers on a distributed system.

Process communication is always initiated with the stipulation of a process handle to a server. A handle to a separately compiled process can be introduced into the environment by a construct whose syntax is:

<install> ::= **for** <identifier> = <clause> [**at** <clause>]
with <signature_list> **do** <clause>

The accompanied type rule is:

for <identifier> = <clause> : **string** [**at** <clause> : **loca**]
with <signature_list> **do** <clause> : **void** => **void**

The resultant type is void.

A process with the symbolic name specified in the string clause is to be found in the locality designated. If found, the signature of this process is matched against that specified. Type matching of signatures is based on a notion of inclusion. The signature specified is required to be a subset of the one expected. This allows servers to evolve independently between program executions without affecting existing clients unnecessarily. If type matching succeeds, the identifier denotes the process handle to the server. Exceptions are raised if the process cannot be found or the type matching fails.

The scope of the identifier is basically confined to the clause following the terminal **do**. However, the process it denotes can be passed out to a global environment by assignments.

Once a handle to a process is installed, communication with it does not require further binding and typechecking. Thus the runtime overhead of binding and typechecking is constant independent of the frequency of RPCs. This is achieved since the required typechecking and binding are already performed at compile time for the majority of code (i.e. after the **do** above) and the remaining delayed check occurs once at installation time.

§4.3.8 Summary

The universe of types of DPS can be described as follows:

type **arith**	is	**int** | **real**
type **comparable**	is	**arith** | **string**
type **printable**	is	**comparable** | **bool** | **pixel**
type **locality**	is	**loca** | **node**
type **literal**	is	**printable** | **locality** | **pntr** | **proc** | **file** | **entry** | **ph** | **process** | **mutex** | **cond**
type **image**	is	**#pixel** | **#cpixel**
type **nonvoid**	is	**literal** | **image** | ***nonvoid**

type **type** is **void** | **nonvoid**

In addition, it is convenient to define the generic type **mutable** for store-to-store operations. However, it does not contribute anything new to the universe of types described above.

type **mutable** is **pntr** | **image** | ***nonvoid**

Note **pntr** represents the union of all labelled cross-products. The literal for this type **nil** is, however, immutable.

Language features on top of those found in PS-algol can be summarized by the following syntactic rules.

<clause> :: = ... | <accept> | <lock> | <termination> | <localities> |
<assign> | <install> | <wait> | <signal>
<accept> ::= **accept** <clause> [|| <clause>] [**otherwise** <clause>]
<lock> ::= **lock** <clause> **do** <clause>
<termination> ::= **stop** | **kill** <clause>
<localities> ::= **add** <clause> **to** <clause>
<assign> ::= **assign** <clause> **to** <clause>
<install> ::= **for** <identifier> = <clause> [**at** <clause>] **with**
<signature_list> **do** <clause>
<wait> ::= **wait** <clause> [, <clause>]
<signal> ::= **signal** <clause> [, <clause>]

where the type rules for each of the constructs are,

<accept> :
accept <clause> : **entry** [|| <clause> : **entry**] [**otherwise** <clause>
:**void**]
<lock> :
lock <clause> : **mutex** [, <clause> : **mutex**] **do** <clause> : **void**
<termination> :
stop | **kill** <clause> : **ph**
<localities> :
add <clause> : **locality to** <clause> : **loca**
<assign> :
assign <clause> : **mutable to** <clause> : **mutable**
<install> :
for <identifier> = <clause> : **string** [**at** <clause> : **locality**]
with <signature_list> **do** <clause> : **void**
<wait> :
wait <clause> : **cond** [, <clause> : **cond**]
<signal> :
signal <clause> : **cond** [, <clause> : **cond**]

In all cases, the resultant type is always void.

```
<expression> ::= ... | <entry> | <process_handle> | <process_template> |
                 <rpc_call> | mutex | cond | <loca> | <node> | <copy>
<entry> ::= entry([<named_param_list>][<arrow> <type>]);
<proc_clause>
<process_handle> ::= start <clause> [as <clause> [at <clause>]]
<process_template> ::= process [with <signature_list>] begin <clause>
end
<rpc_call> ::= <clause>@<clause>(<clause_list>)
<loca> ::= newlocality | locality <clause>
<node> ::= newnode | node <clause>
<copy> ::= transcopy <clause> [to <clause>]
```

where the type rules are,

```
<entry> :
   for any t : type, entry([<named_param_list>][<arrow> <type> : t]);
      <proc_clause> : t  => entry
<process_handle> :
   start <clause> : process [as <clause> : string[at <clause> :
locality]]
      => ph
<process_template> :
   process [with <signature_list>] begin <clause> : void end
      => process
<rpc_call> :
   <clause> : process @ <clause> : entry(<clause_list>)
      => type
<loca> :
   newlocality | locality <clause> : type
      => loca
<node> :
   newnode | node <clause> : type
      => node
<copy> :
   t : type, transcopy <clause> : t to <clause> : t
      => t
```

For <rpc_call>, the second clause must be of some entry type which is in the signature of the process denoted by the first clause. The types of parameters of the entry must match with those in the call. The resultant type is the result type of the entry.

Chapter 5

Remote Procedure Call

§5.1 Introduction

A remote procedure call (RPC) mechanism is understood here to be a synchronous process communication mechanism. It shares the idea of the conventional procedure call mechanism through which data can be conveyed between different parts of a program that is otherwise impossible in the normal flow of control. An RPC mechanism, furthermore, enables processes to communicate with one another across address spaces. Since the mechanism does not require the introduction of a radically new concept into a programming language and because it is amenable to typechecking, it is an attractive basis on which process communication over a network can be realized within the confines of a conventional language. Our ultimate goal is to induce a style of distributed programming into the realm of persistent programming.

Our RPC mechanism is an essential element in the proposed concurrency model assisting processes to communicate with one another. It allows process communication to occur over a network, within the same machine as well as within the same address space. In any case, there is no syntactic difference and we try to hide semantic differences as far as possible. What we are trying to achieve is an arrangement whereby geographic differences need not be a concern in distributed programming. In fact, the term "distributed programming" is not a well-defined one in such a context. That is to say it is not apparent from the code of a program whether it or some part of it will be executed in a distributed or localized context.

The most unusual aspect of our RPC mechanism is in its semantics of parameter passing. It has to be pass-by-value whereby pointers are allowed to be exchanged. This is essential for the realization of distributed shared stores. Such a parameter passing semantics is possibly the first of its kind employed in a communication mechanism. An added bonus for such a semantics is that it conforms with that in ordinary procedure calls in PS-algol. There are other points of interest as well. These include:

1) the support for lightweight processes,
2) typechecking, binding and separate compilation,

3) semantics of parameter passing,
4) calling semantics and exceptions,
5) the abstract machine, and
6) the communication protocol.

Each of these will be discussed in the present chapter. In the following discussion, "the mechanism" is understood to be referring to our RPC mechanism unless stated otherwise.

For the sake of efficiency and self-sufficiency, our RPC mechanism is implemented as an integral part of the host language abstract machine. In this case, the host language is DPS, a descendant of PS-algol (see chapter 4). Minimal additions and structural re-organization are made to the original machine. The mechanism is an orthogonal component so that changes to other components do not affect the mechanism and vice versa.

§5.2 Lightweight Process

The mechanism is an essential part of the proposed concurrency model. Its main function is to facilitate communication between processes wherever they happen to be. These processes may be within the same address space or different address spaces on the same machine or different address spaces on different machines. Moreover, the concurrency model allows processes to be created dynamically. Thus a newly created process may communicate with any existing processes; they may even be on different address spaces on different machines. Although there is no assumption that the communication partners of a process are statically determinable, in some cases they are e.g. a child and a parent processes within the same address space only communicate with each other. We require that process communication is always accomplished, under such diverse circumstances, in a manner that is transparent to the programmers. We described the approach taken to achieve this.

In order that processes can communicate with one another across address spaces, each will have to be associated with a reusable unique network address to which messages can be delivered. Because of the dynamic nature of processes, these addresses are allocated at runtime whenever a process is created. All processes of an address space belong to a family of network addresses. Note that whether a process will communicate cannot be statically determined. Note also that whether it is capable of communicating cannot be statically determined either. But

as a consequence of persistence, a non-communicating process could extract from a persistence store a piece of code selected on data dependent computation which when executed starts a communication.

Each instance of the abstract machine has a single communication port. This is so that all incoming and outgoing messages can be channelled through this port. De-multiplexing of messages is done on the basis that the network addresses of processes are structured in such a way that they share the same denominator i.e. the network address of the port but differ by a process identifier. The allocation of network addresses to processes is simply assigning to each one of them a process identifier. It can be said that the network address of a process is composed of a low (communication port address) and a high (process identifier) address. In principle at least, the low address can itself be constructed in this way, and so on recursively, allowing various network architectures and dynamic change of network configuration.

The use of the network facilities in process communication local to an instance of the abstract machine is avoided. This is an efficiency consideration only. One way this can be achieved is to compare the low address of the network address of the target process with that of the communication port. If they are the same, the message is directed to the process immediately based on the process identifier of the target process' network address without resort to the underlying network. We chose a more efficient way -- optimization. Such a technique is supported with the provision of two opcodes. The compiler keeps track of the processes involved in communication whenever possible. It generates the opcode for local communication to accomplish the call when the static information indicates a local call. In case of doubt or known remote communication, the compiler emits the other opcode which involves the dynamic check anyway. Such optimization applies only to the calling but not the returning part of an RPC.

§5.3 Typechecking, Binding and Separate Compilation

We require that any process can communicate with another process anywhere in the system provided it is running and reachable and that the type system is not jeopardized. The problems of typechecking and binding must be resolved prior to any process communication. Because of distribution existing compiler technologies are insufficient. In the context of communication, processes behave like servers or clients. It is not suggested that a process can only send messages or receive messages during its life time; it can do both, but one at a time. In a truly distributed environment, it must be possible for clients and servers to be compiled

independently on different machines and perhaps in arbitrary order. In principle the two problems can be resolved with the help of some distributed database systems. Clients can be compiled in the context of servers whose type information and locations are kept in a distributed database system with consistency guaranteed. The use of such distributed database systems is out of the question in our case. They are exactly what we are trying to realize. The two problems have to be resolved by some other means.

Processes communicate with one another by exchanging messages through communication ports using RPCs. These ports known as *entries* (see §4.3.1) are not to be confused with that used by the mechanism itself. Entries are objects created as a result of declarations in user programs. They are rather like procedures in that they are first class objects and have procedure-like headers and executable bodies. There is a semantic difference between entries and procedures; that is: an entry is a synchronization point for two processes engaged in a communication and they are resumed only upon completion of its execution.

For the reason of modularity, an entry is associated with a particular process although it can be shared. There is no limit as to the number of entries that a process can associate with. In a sense, entries characterize the observable behaviour of a process. For this reason, the type of a process is taken to be a list of its labelled entry types which is known as a *signature*(§4.3.2). Signatures are the basis on which typechecking of messages and binding of communication ports are resolved.

Typechecking of messages is achieved by matching entry types with parameters to be passed similar to that in ordinary procedure calls. Due to the lack of a depository where type information can be kept, it may appear to be necessary to perform typechecking every time a message is received. We avoid this with the aid of a device provided in the language which allows typechecking to be performed mostly at compile time.

In the case of process communication local to an address space, typechecking is always performed statically at compile time. This is possible because the compiler keeps track of the signatures of all processes.

In the case of servers and clients being compiled separately, typechecking is split into two stages. First of all, clients wishing to communicate with a non-local server have to establish a context to allow

the compiler to perform static typechecking on remote procedure calls. The language provides a construct for this purpose. The context is constructed by means of a *to-be-verified* signature. The extent of the context is limited by the normal scope rules and such contexts can be nested. The second stage involves a runtime verification. At runtime, the signature of the server is matched against the to-be-verified signature to ensure the validity of typechecking performed at compile time. An exception is raised if they do not match. Signature equivalence is further explained below. The advantage of such an arrangement is that the cost of typechecking is constant independent of the number of calls. More precisely, the dynamic cost of verifying message type compatibility is proportional to the number of interfaces used, not the number of times they are used. Furthermore, typechecking performed every time a new to-be-verified signature is encountered is an effective way of signalling changes in an evolving environment.

Binding of communication ports occurs at compile time whenever possible. In the case of remote servers, binding occurs after the runtime verification.

Signature equivalence required in the runtime verification is unusual in that it has an element of flexibility to account for the dynamism of a distributed system. Recall that the behaviour of a server may be characterized by its signature. Changes in the observable behaviour of a server may affect its clients. The runtime verification will ensure those affected are notified by exceptions. However, there are changes which do not affect existing clients at all. These are re-ordering of declarations of entries, re-declaration of existing entries, and introduction of new entries. These changes can be accommodated easily with the use of *indirection tables*. There is one such table per remote process introduced in a context. The size of the table is directly proportional to the length of the to-be-verified signature or the number of interfaces used and is determined by the compiler. In compiling a to-be-verified signature, the compiler assigns to each entry in it a number corresponding to an entry to the indirection table. These numbers are used to index the indirection table to obtain the true stack addresses of the target entries required at runtime. The rule for signature equivalence is simple. For every labelled entry in the to-be-verified signature there must be a corresponding one of equivalent type in the signature of the server. If there is one, the stack address of that entry is put into the corresponding table entry. A type exception is raised otherwise. The stack addresses of target entries are deduced since they occupy the bottom

addresses of the process' stack and are in the order they appeared in the signature.

The remaining problem is how to locate a remote server? This is achieved with the use of a collection of process name spaces. We do not believe maintenance of a global name space is manageable due to frequent dynamic changes in a large network. Our arrangement is to have a process name space per machine on the network. A server wishing to communicate with processes outwith its address space registers itself with a process name space on start up; the default being the local process name space. The information required in a registration is:

- its signature,
- its network address, and
- a symbolic name.

A registration can be refused on the grounds of name clashes. A client names its server in a particular process name space. If the server exists in that name space, its signature and its network address are returned for the purpose of typechecking and binding. When a server terminates, its name becomes reusable in the appropriate process name space. Consequently, all previous bindings to its entries become invalid. Any attempt in communication with that server results in a system event [Philbrow & Atkinson 86]. In principle, again, and independent of the architecture of the network, name resolution may be subdivided by a recursive structure of name spaces and name resolution agents.

§5.4 Semantics of Parameter Passing

The semantics of parameter passing is unusual as it supports pass-by-value for pointer values as well as pass-by-value for scalars. Logically speaking, referends are not copied though the implementation may make copies where this achieves an optimization without jeopardizing the semantics. Interprocess communication mechanisms designed hitherto [Cardelli 84, Liskov 85b, Birrell & Nelson 83] implicitly copy data across address spaces. Our mechanism employs such a semantics for the purpose of emulating distributed shared stores.

Of course, it is not possible to pass ordinary pointers from one address space to another. They are context sensitive and their interpretation in a foreign address space would be catastrophic. Our solution to this problem is to introduce *remote pointers* which are universally recognizable. The representation of remote pointer requires more space on stacks than ordinary pointers as the former contains more

information on the location of data which can be anywhere on a network. However, the seemingly necessary architectural change is avoided by introducing remote pointers in the heap rather than pushing them directly onto the stack. Their existence is invisible to the programmers, though. This is fundamental to our approach towards distribution. The management of remote pointers and aspects of the distributed stores that arise are discussed further in chapter 7.

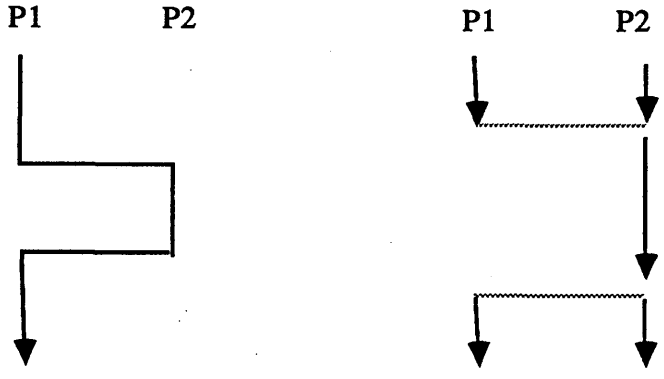
There are two types of pointers in the system: local pointers to objects in the same heap and remote pointers to objects outwith the local address space. All references that are represented by a local pointer could be represented by a remote pointer (as they are in System/38 [French et al. 78], Poppy [Cockshott 85], Monads [Abramson & Keedy 85] and Thompson machines [Gallagher 85]). We therefore need to justify the complexity of supporting both representations. It is expected, and we later present preliminary evidence, that there are many more objects created for local computations than those addressed by remote processes. It will not be cost-effective to assign each object ever created a universally recognizable address, they will require additional bandwidth on channels and additional de-reference time. But two address representations requires time in very de-reference to determine the type of pointer representation in use, and information per reference indicating the type. This will be explained in more detail in chapter 7. Since objects can be addressed remotely only if they had been exported, the mechanism is the only place where remote pointers can be introduced. A pass-by-value semantics allows these to be exchanged across address space boundaries.

An important motive for the encapsulation of remotely addressable data in remote pointers is that it avoids the need for a single addressing mechanism for both local and remote data. Consequently, this enables the interpretation of the locality information in a remote pointer to vary due to network re-configuration so that the remote referend remains addressable. Furthermore, it allows an easy adaptation without requiring any significant change in the addressing structure of a high level language.

§5.5 Calling Semantics and Exception

It is desirable that RPCs resemble, as far as possible, ordinary procedure calls. This can contribute towards the simplicity of the host language. Employing the same semantics of parameter passing in both cases represents a significant part of this. Although there is a difference in synchrony, callers of the two mechanisms always wait for the calls to

return. The diagram below illustrates this concept.



A Procedure Call A Remote Procedure Call

In the diagrams, the solid lines represent the flow of control in the direction indicated. Note that in the right diagram, there are two threads of control which are synchronized at the point of the upper dotted line. The lower dotted line represents the point where both the caller and the called may continue their execution depending on whether the result or a null value has been exchanged. In some situations, the difference in the time waiting for the process P2 in the right diagram to complete execution of the appropriate entry may not be significant from that in the left diagram.

A pragmatic difference between the two mechanisms is in reliability. In ordinary procedure calls, a call either succeeds with exactly one invocation of the procedure, fails with an exception being raised or never returns because of non-termination. The difficulty in accomplishing RPCs is that there is a plethora of low level, unreliable mechanisms involved. For instance, a packet may be corrupted due to electrical noises along the physical transmission medium before being received. Any hiccup of such nature will cause a message to be undeliverable. This is aggravated by machine or network failures. The chances of failure in an RPC, discounting exceptions and non-termination, are higher than a procedure call. Such failures are unpredictable and unpreventable. The best strategy is to bypass them by re-trial tactics (in the case of corrupted packets) or to detect and report them (in the case of machine or network failure).

The problem caused by corrupted packets can be solved by re-

transmission. Normally, each packet sent will have to be acknowledged. Otherwise, the same packet is transmitted repeatedly until it is acknowledged. However, retransmission by itself creates another problem. It will cause multiple delivery of messages. A consequence of retransmission of, for example, call messages is multiple invocations of the remote procedure with possible multiple side-effects only one of which is expected by the programmer. What is required here is that message delivery is not only reliable but also not duplicated. This can be achieved by assigning every message a sequence number and the receiver maintaining a sequence counter. A message is received and acted upon only if its sequence number is greater than the sequence counter. The counter is then set to that sequence number. This scheme works on the assumption that retransmitted packets carry the original sequence numbers assigned to them.

It is often possible to take advantage of a transport level protocol which meets some of these requirements [Postel 80]. Their design is not part of this work, though an efficient combination of transport protocols and RPC protocols may require that they are considered together. On the other hand, a fully-fledged RPC protocol may dispense the support of the underlying protocols. This may be more convenient as the RPC mechanism is not dependent on the availability of certain protocols.

The sequencing technique guarantees what is known in the literature as at-most-once semantics. Such semantics ensures that an invocation is made once, if at all possible. Despite this technique, an RPC may still fail because of network or machine failure. The server and the client are notified. The means of notification is through exceptions. It is expected that server processes receiving such exceptions are responsible for undoing side-effects of entry invocations.

There is no notion of timeout in the mechanism. It is believed that timeout is inappropriate in this context since it is absent from procedure calls. Furthermore, a timeout mechanism would require a global clock which is hard to maintain on a distributed system. The absence of a timeout facility is compensated for with several kinds of failure being detected and the appropriate processes being informed through exceptions. Detectable failures include not only machine and network failures but also pre-mature process termination as well. Timeout can be achieved in the language by using a local clock process.

§5.6 Abstract Machine

The RPC mechanism is integrated into an abstract machine. Apart from

structural differences and the capability to communicate with the outside world, the resultant abstract machine is basically compatible with the PS abstract machine [PPRG 85a]. The PS abstract machine has:

- been designed for efficient execution of reverse Polish codes,
- two stacks: one for scalars and one for pointers,
- a heap for persistent objects,
- built-in graphics, and
- supports for higher order functions.

The provision of a heap for persistent objects characterizes one approach towards persistence. Persistent objects are copied between a heap and a persistent store incrementally on need. The movement of persistent objects occurs when either the heap is full after a garbage collection, a persistent identifier is dereferenced or a commit is performed. Persistence is defined in terms of reachability consistent with the view that such a property of data is orthogonal to their types.

The separation of a scalar and a pointer stacks is for conveniency. This allows pointers to be found quickly without resort to a tag architecture. The task of garbage collection retaining objects used by the current block is very much simplified because of this. As a consequence of the separation, the number of registers is slightly higher than usual and therefore the time and space required in dumping and restoring during a context switch is relatively speaking more expensive.

The support for higher order functions significantly enhances the expressive power of the host language. The realization of higher order functions itself requires the abolition of stacks in favour of stack frames as heap objects. Stack frames are chained together to model the enter-exit nature of blocks and the call-return nature of procedures. As it turned out, such an arrangement renders the abstract machine suitable for implementing concurrency.

The concurrency model does not impose an upper bound on the number of processes that can be created. In reality, this is limited by the available space on the heap. Free space is returned by a compacting garbage collector which is invoked on a demand basis. Potentially objects in the heap are sharable by all processes. The scope rules of the host language governs the set of directly addressable objects of individual processes. A process can affect other processes either by modifying objects *in situ* or by RPCs.

All processes share the same resources e.g. the heap, the I/O streams, the screen and the mouse etc. Concurrency is realized by interleaving and a pre-emptive scheduling strategy. However, such a technique does not preclude the possibility of implementing the abstract machine on multiprocessor machines with a common store. This is because the number of processors available cannot be assumed to be greater than the number of processes. Hence a scheduler remains necessary. However, it is then necessary to ensure the execution of some abstract machine instructions to be atomic.

Scheduling is basically pre-emptive which allows I/O bound and computation bound processes to be handled differently according to needs. On the other hand, there are instructions which may cause the suspension of the executing process and thereby yielding a processor. The overhead in context switch is comparable to procedure call since all processes are always kept on the heap.

The implementation of the mechanism and the concurrency model requires some 21 primitives to be added to the basic instruction set. The operational semantics in terms of the state of the abstract machine of these primitives is given below. The same technique can be applied to other instructions which are ignored here because they are not essential for the understanding of the mechanism and the concurrency model.

§5.6.1 States

The state of the abstract machine is determined by ten pointers, a system event register, a mutex register together with their denotations and the heap. The two registers are specific to certain instructions and they are not included in the general discussion below. The pointers are the Code Pointer, the Local Main Stack Pointer, the Local Main Stack Base, the Local Pointer Stack Pointer, the Local Pointer Stack Base, the Event Main Stack Pointer, the Event Pointer Stack Pointer, the Current Process, the Ready Processes Queue and the Suspended Processes Pool. System events are raised by the abstract machine if it is forced to abandon an evaluation whose result would cause inconsistency in the state e.g. stack overflows. Some system events can be caught and handled by user programs. System events are not exceptions which are raised and handled by user programs.

The Code Pointer (CP) points to the next abstract machine instruction of the currently executing compilation unit. A compilation unit is either a block, a procedure or a process template.

The Local Main Stack Pointer (LMSP) and the Local Main Stack Base (LMSB) point to the top and the bottom of the same stack which holds scalar values. LMSB is part of the machinery in providing the environment for the currently executing unit. It is reset on every context switch. The values between LMSB and LMSP are the denotations for some of the free variables of the current block. LMSP always point to temporary values; for example, parameters to be passed, results to be returned or results of subexpressions. LMSP can be raised, in which case the values below become permanent for the current block.

The Local Pointer Stack Pointer (LPSP) and the Local Pointer Stack Base (LPSB) point to the top and the bottom of the same pointer stack and behave and function in a similar fashion as the pair above. Values on the stack delimited by the two are pointers to objects in the heap.

The Event Main Stack Pointer (EMSP) and the Event Pointer Stack Pointer (EPSP) point to the stacks for holding excepted values which can be scalars or pointers. They are reset once the event has been handled. The stacks are shared by all processes.

The Current Process (CPr) points to a process handle representing the currently executing process. A process handle is a record of the state of the machine necessary for the resumption of a process.

The Ready Process Queue (RPQ) and the Suspended Process Pool (SPP) point to a queue of readily executable processes and a collection of suspended processes respectively.

For the rest of this document, we take a somewhat high level view of the state of the machine. We describe the following abstract machine instructions in terms of their effects on CPr, RPQ and SPP only. Unnecessary details have been either abstracted away or omitted altogether. In the figures below, the state of a process is represented by an ellipse and any other object by a rectangle with a name. For instance, a process in the state p with a vector on its execution stack will be represented as:

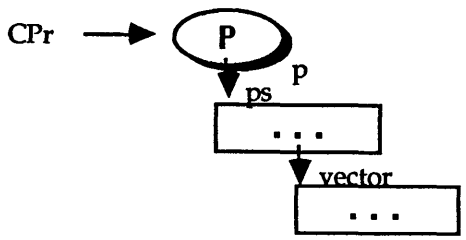


Fig. 5.1 A process with a vector

A process can be in one state at a time. But it is possible that a process suspends the present state and enters into a new state or resumes a previous state. In both cases there is a context switch but the processor remains allocated to the executing process. In the following figures, only the state of a process prior to entering into a new state is shown. In the same vein, it is understood that the current state is discarded (as far as that process is concerned) upon resuming a previous state. Moreover, a process handle with an italic character represents a foreign process not in the heap. The state identifier of a process is left out from most of the figures below if it is not necessary. In the diagrams below, a grey line indicates a separation of address space.

§5.6.2 Communication

There are ten communication related primitives. They can be divided into three categories: RPC call, RPC return and accept RPC. Operands that can be determined statically are embedded in the code stream e.g. the number of parameters in a procedure call. Operands that cannot be determined at compile time can be found on the top of the stacks. For example, the procedure to be called. Primitives in the RPC return category differ in the type of the operand on the stack but otherwise they have exactly the same operational semantics.

rpcLocal ms ps offset

This instruction is for an optimization in RPCs. It is for accomplishing process communication local to an address space. *ms* and *ps* are the number of scalar and pointer parameters to be passed. *offset* specifies the stack offset required to locate the target entry. The stack on which entries are kept can be found down a chain of frames, if necessary. The top of these frames can be found from a process handle which is beneath the pointer parameters on the stack. A call packet containing the parameters and the local or network address, whichever is appropriate, of the caller is constructed and put at the back of the queue of the target entry. *ms* and *ps* number of parameters and the process handle are

popped off from the stacks. The executing process is put into the SPP. If the target process is in the SPP, it is put back onto the RPQ immediately. A process from the RPQ is selected as the current process. In this case, there is always a readily executable process that can be selected as the current process.

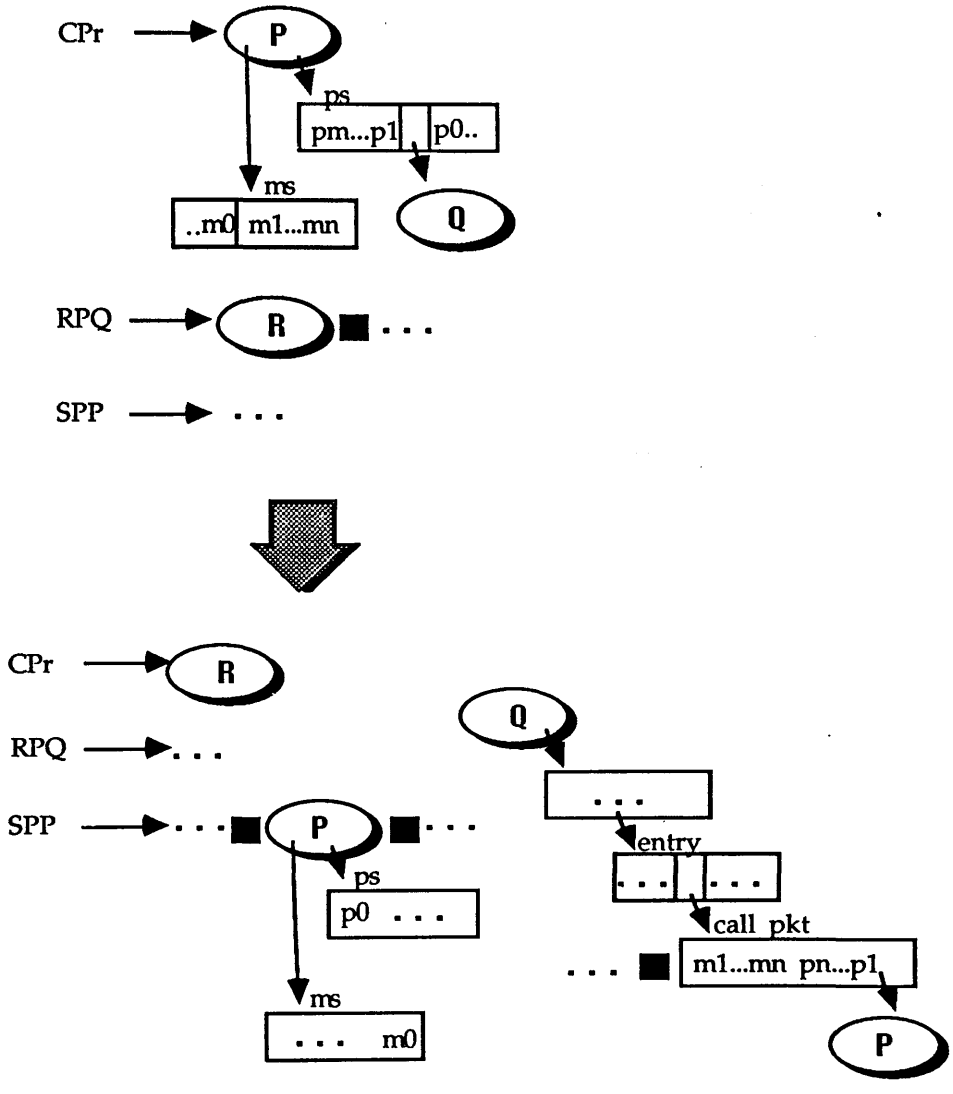


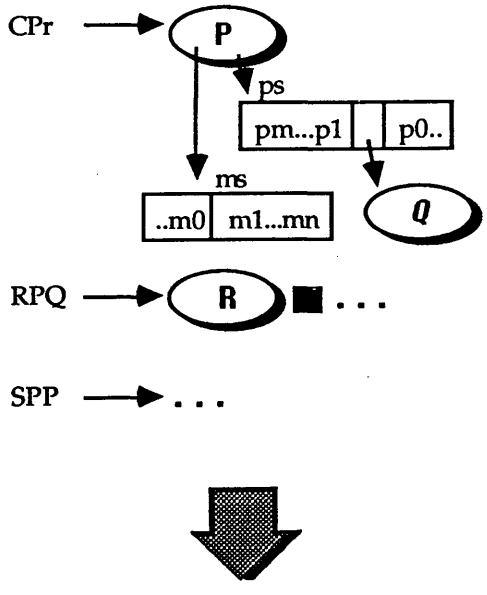
Fig. 5.2 rpcLocal

rpcRemote ms ps offset

ms and *ps* are as above. In contrast with the instruction above, the *offset* here is not the stack address of the target entry. It is used to index an indirection table where true stack addresses of entries can be obtained. There is an indirection table associated with each remote process communicating with any one of the lightweight processes in the heap.

An indirection table can be accessed via the process handle which is beneath the pointer parameters on the stack. A call packet is generated for dispatch to the target abstract machine. However, the dispatch of the call packet does not involve the network if the receiver machine is the same as the sender machine. The distinction can be made by comparing the low addresses of the communication port of the two instances (sender and receiver) of the abstract machine.

ms and *ps* number of parameters are popped off from the stacks. The executing process is put into the SPP. A process is selected from the RPP as the current process. If there is no selectable process, the control is passed to the protocol handler. It is guaranteed that when it returns, there is an executable process on the RPP. For this reason, we assume that it is always possible to select a process to be the current process. In extreme cases, the selected process is just the process suspended prior to the execution of this instruction. In the following discussion, we assume that there is always a process selectable as the current process on any context switch. The abstract machine terminates only when instructed to do so.



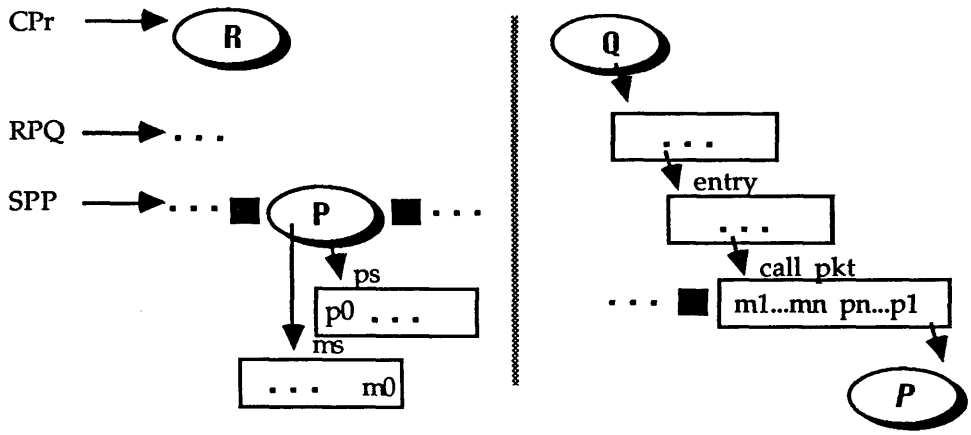
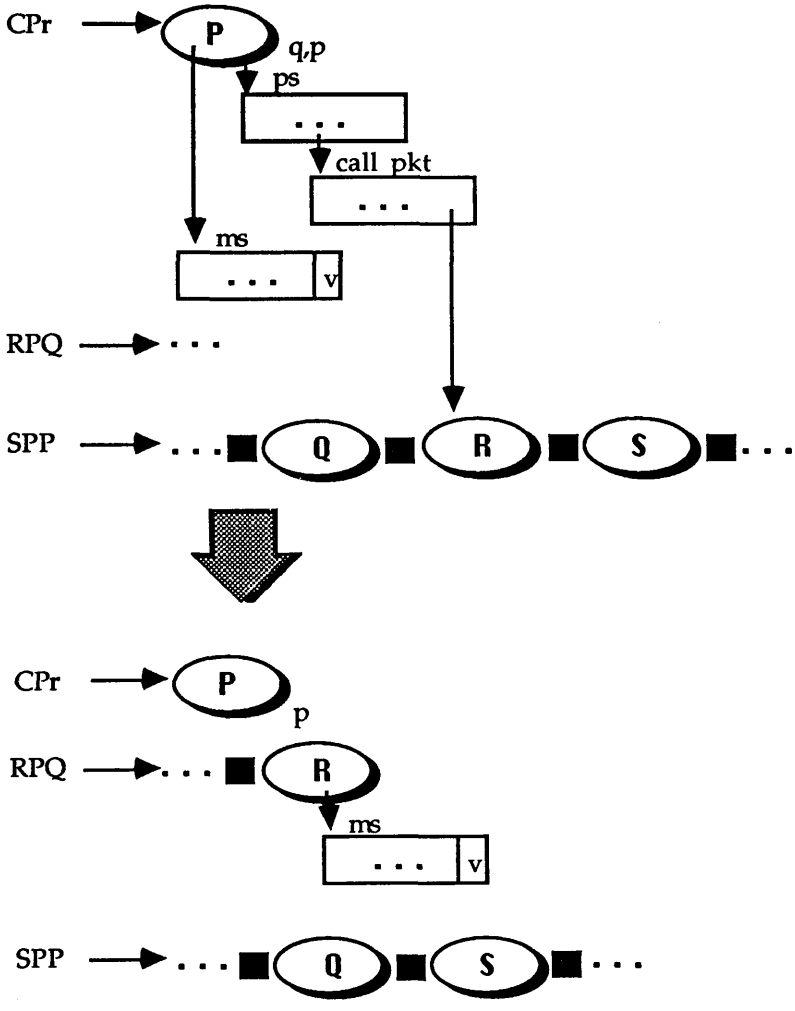


Fig. 5.3 rpcRemote

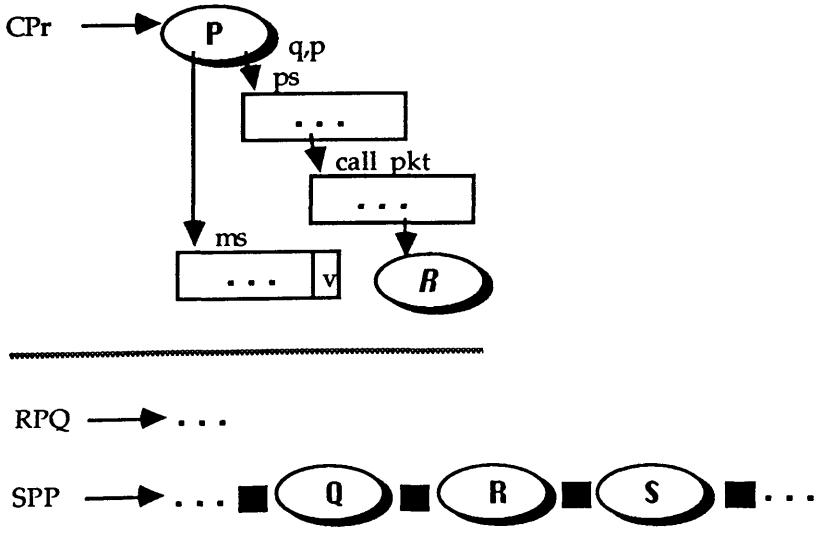
In the category of RPC return, there is one abstract machine instruction for each of the data types understood by the abstract machine. They include integer, real, pointer, procedure and void. In each case, the same instruction is used for returning an RPC to a process both within or outwith the address space. It is because the model allows a process to accept calls both from within and outwith the address space in which it is executing. The locality difference of callers cannot be determined at compile time. It can only be established at runtime. For didactic purposes, only one of the RPC return primitives is described here.

rpcRtnIB

The only operand can be found at the top of the scalar stack. The return address can be found from the original call packet which is kept on the pointer stack. The return address can be either a local address or a network address. In the case of a local address, the caller must be in SPP. It is then put at the back of the RPQ. In the other case, a message containing the result is sent back to the caller. In the same vein, the caller is put back onto the RPQ of the abstract machine in which it resides. The process that owns the entry called is resumed via a dynamic link.


CPr
P
q,p
ps
call pkt
ms
v
RPQ
SPP
Q
R
S
CPr
P
p
RPQ
R
ms
v
SPP
Q
S

or


CPr
P
q,p
ps
call pkt
ms
v
R
RPQ
SPP
Q
R
S

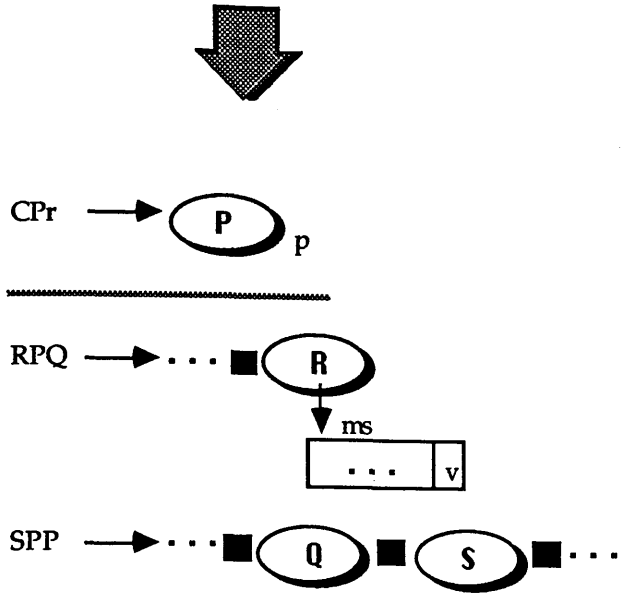
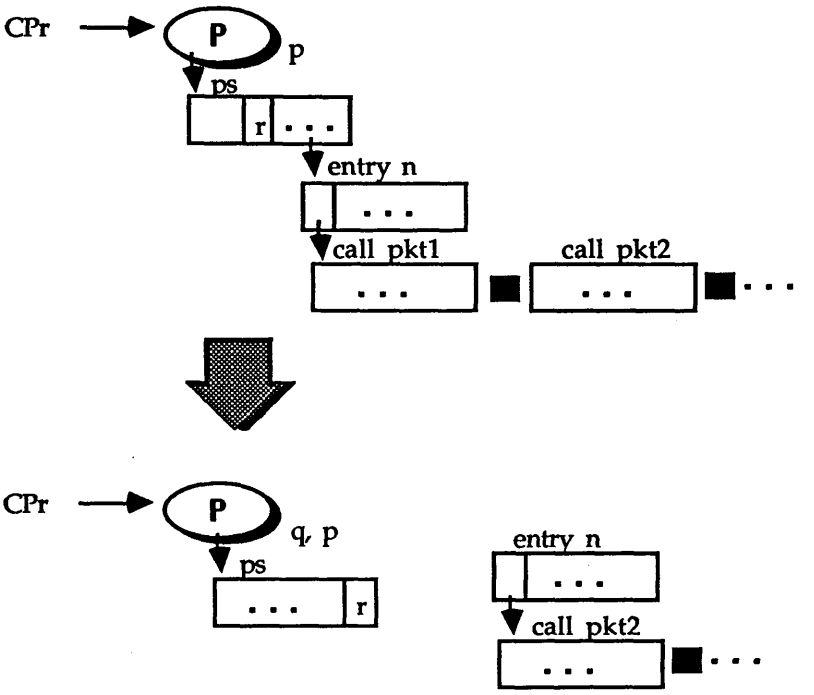


Fig. 5.4 rpcRtnIB

acceptElse m

m is the number of entries at the top of the pointer stack. An entry is selected if it has a call packet. If there is more than one candidate entry, one is chosen at random. This is the only means whereby non-determinism is introduced. When an entry is selected, the executing process suspends its present state and enters into a new state to serve the call. This is akin to procedure calls. If there is no call packet in any one of the entries, the executing process continues. In any case, *m* entries are popped off from the pointer stack. The executing process continues either in its present state or a new state to serve the call.



or if there is no call packet on any entry,

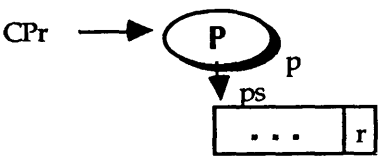
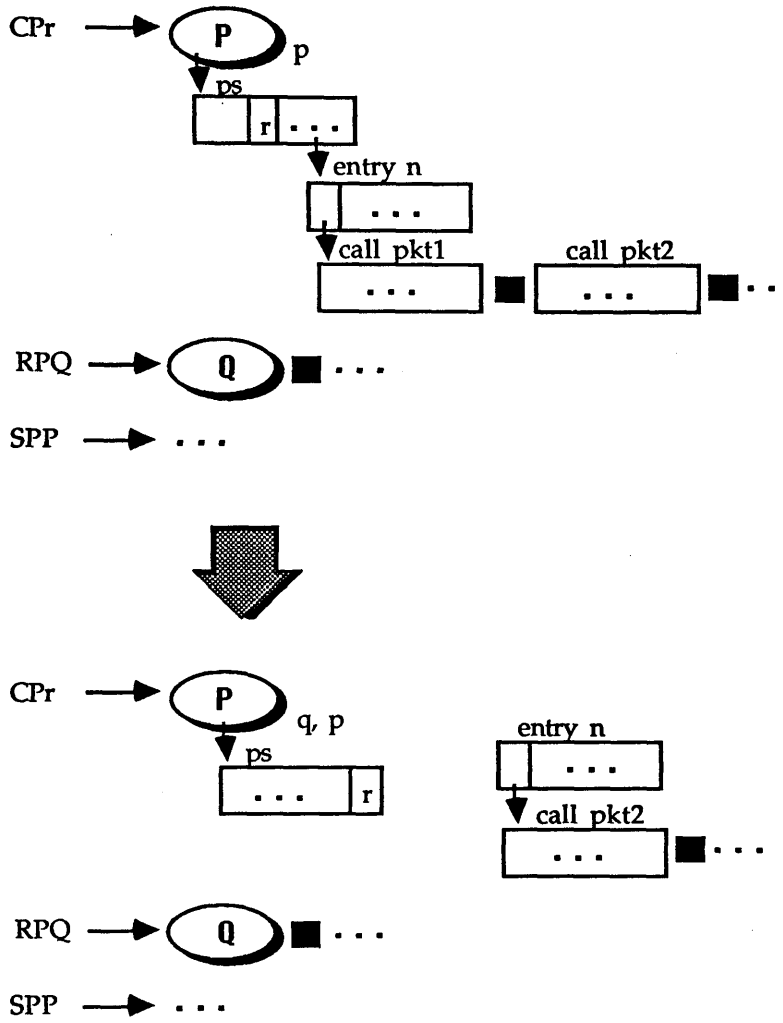


Fig. 5.5 acceptElse

accept m

This primitive behaves exactly the same as in the previous case. However, if there is no eligible entry, the executing process is suspended and another process is selected as the current process. *m* entries are popped off from the top of the pointer stack only if a call packet can be found among them.



or if there is no call packet on any entry,

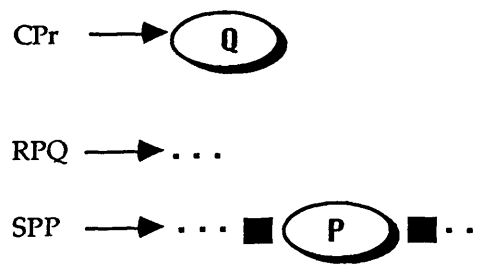


Fig. 5.6 accept

§5.6.3 Concurrency

A process can be created dynamically. A process is created based on a process template whose structure is well-defined. Basically, it is a vector

of codes. Apart from this, it has a vector of strings, a vector of indirection tables, a vector of procedures and entries together with an indication on how many entries there are to be initialized.

start

A process template, a symbolic name and a machine name are at the top of the pointer stack. A process is created and put at the back of the RPQ. If the symbolic name is not null, the process is meant to be registered as a server preparing to accept calls outwith the address space. In this case, the signature and the network address of the process together with its name are sent to the name resolution agent specified. By default, it is the local one. A runtime error is reported if there is a name clash. Once a process is started, other processes can communicate with it immediately. It is therefore necessary to initialize the entries so that call packets are not lost. The number of these entries can be found in the process template. The process template is popped off the stack and the current process continues execution.

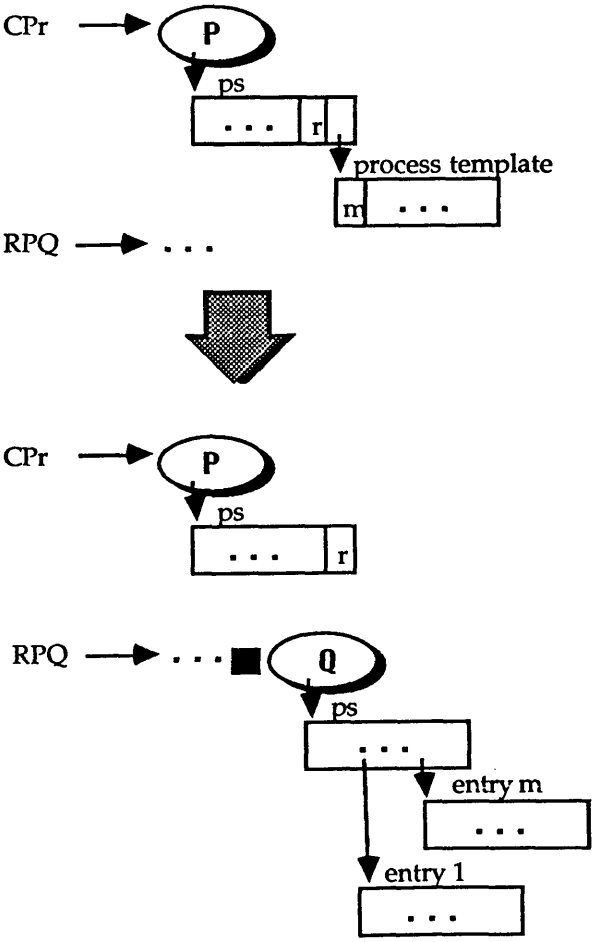
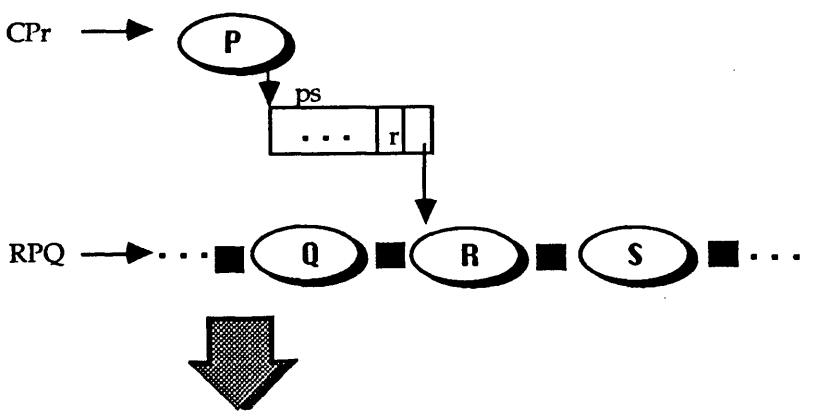


Fig. 5.7 start

There are three ways in which a process can be terminated. It is terminated upon completion, voluntarily, or by request from another process. In any case, if there is any outstanding call packet, the caller is notified through a system event. The terminated process is eventually garbage collected. For each program, there is a top level process, the two instructions -- *endProcess* and *kill* -- do not apply to it although the other one *stop* does. The two abstract machine instructions will never be generated for the top level process. We will describe *kill* only. The other two are similar except that the operand to each one of them is always taken to be the current process. *stop* is useful for voluntary termination and the execution of *endProcess* signals the natural termination of the current process.

kill

A process handle representing the process to be terminated is found at the top of the pointer stack. The process can be found either in the RPQ or the SPP of this or another instance of the abstract machine (Fig. 5.8 shows one case only). It is discarded from whichever it is in. The process handle is popped off from the stack and the executing process continues. The operation is a dangerous one. It is supported to prevent useless processes from hogging the resources. Other processes communicating with the terminated process will be notified through a system event in due course. When the process is a registered server, a message indicating its name can be reused is sent to the process name server where it was registered.



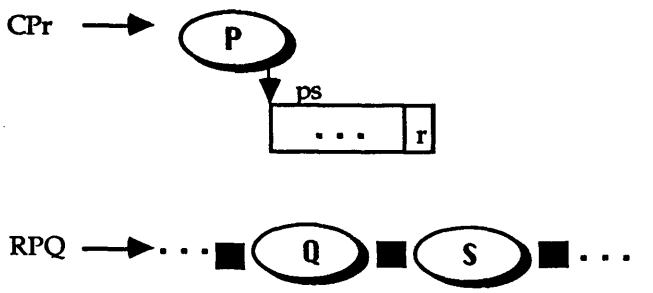


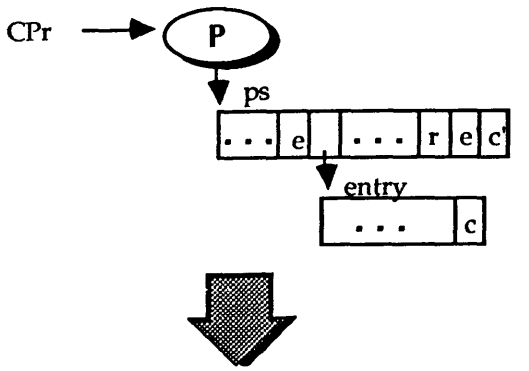
Fig. 5.8 kill

§5.6.4 Load and Assignment

We describe load and assignment of entries and process templates here. There are three pairs of instructions. The two instructions in a pair have exactly the same operational semantics. One instruction with a larger operand field (2 vs. 1 byte) is generated to handle exceptionally large numbers of entries or process templates. In essence, there is an assignment instruction for entries and two loading instructions for entries and process templates respectively. There is no special assignment instruction for process templates. An existing instruction suffices to handle this.

entryAss offset

offset is the stack address of the target entry. A closure i.e. a code vector and an environment is found at the top of the pointer stack. An entry is an object consisting of three fields viz. a queue of call packets, a code vector and an environment in which free variables can be resolved. The effect of this instruction is to replace the code vector and the environment parts of the target entry. The queue of call packets remains unaffected. The closure at top of the pointer stack is then popped off.



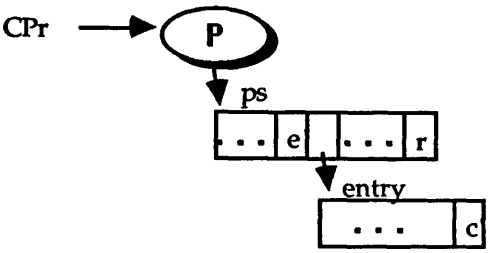


Fig. 5.9 entryAss

loadEntry offset

offset is used as an index into the vector of procedures associated with the process template of the current process. The closure found is then pushed onto the top of the pointer stack.

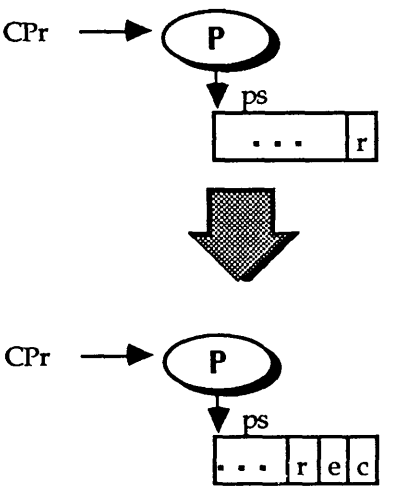


Fig 5.10 loadEntry

§5.6.5 Miscellaneous

In order to communicate with processes outwith an instance of the abstract machine, process handles representing such processes must be loaded onto the stack. On such an occasion, runtime typechecking is performed and the indirection table for the current lexical level is filled with the true stack addresses of entries of the foreign process.

rpcGet itIndex

itIndex is used to locate the indirection table for the current lexical level in a vector of indirection tables. The dimension of the vector and therefore the degree of nesting of blocks where remote process handles are introduced is limited to 256. The process name, the machine name

and a signature are at the top of the pointer stack. The protocol handler sends a message to a name resolution agent on the machine specified. The reply contains the network address and the signature of the target process. If this signature matches that on the stack, a process handle is created and pushed onto the top of the stack. The process name, the machine name and the signature are popped off from the top of the pointer stack. If the two signatures do not match, a runtime type error is reported as an exception; otherwise, the current process continues execution.

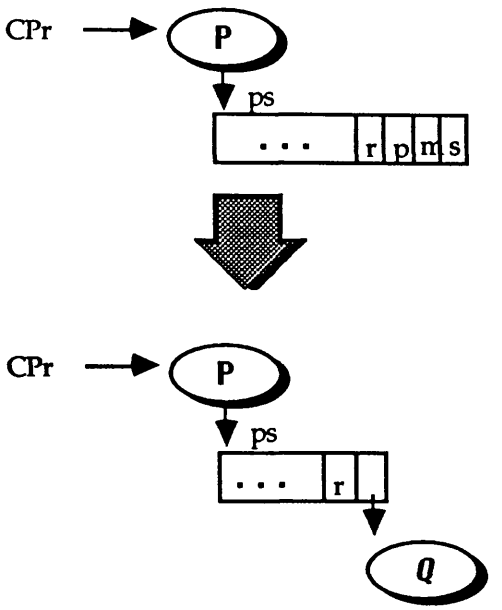


Fig. 5.11 rpcGet

The signature matching algorithm is straightforward. It can be expressed formally as follows.

$$tbvSig = \{ n_i : T_i \}$$
$$retSig = \{ n_i' : T_i' \}$$
$$\forall (n : T) \in tbvSig \; \exists (n' : T') \in retSig : n = n' \text{ and } T' \leq T$$

The two signatures are: a *to-be-verified* signature and the returned signature of the remote process. The two signatures match if and only if the former is a subset of the latter. Note that the order within a signature is not significant.

The true stack addresses of each entry in the *to-be-verified* signature are deduced as follow. In searching for a corresponding type equivalent entry in the other signature, the number of entries scanned so far

including the current one is remembered. If one is found, the number is entered into the appropriate slot in the indirection table. It is the true stack address since all entries occupy the initial slots of the process stack in the order in which they were declared.

There are two instructions for acquiring and releasing mutex objects. These are *Get* and *Release* which are reminiscent of the P and V operations of semaphores. *Get* takes only one operand and that is a mutex object which is at the top of the pointer stack. It sets the mutex register to point to the mutex it is operating upon. If the mutex is not currently in use, it is set to busy state and the Boolean value "true" is left on the top of the main stack. The mutex at the top of the pointer stack is not removed and the executing process continues. *Release* also takes a mutex operand which is at the top of the pointer stack. It resets the mutex to indicate it is free only if it has not been seized by a process. If there is a process waiting for the mutex, it is made runnable. The mutex at the top of the pointer stack is removed and the executing process continues.

Note that the *Get* instruction does not cause process suspension. It is caused by the execution of the *Suspend* instruction which suspends the current process on the mutex pointed to by the mutex register. The operational semantics of these instructions are a consequence of a compilation technique used for the **lock** clause in the language (§6.6).

§5.7 Communication Protocol

Any RPC mechanism has to be realized upon some kind of communication protocol capable of message delivery. Some of the requirements on the underlying communication protocol have already been outlined above. A connection oriented or virtual circuit protocol is adequate to meet those requirements viz. reliable and un-duplicated message delivery. However, as Larus [Larus 83] observed a remote procedure call mechanism (in his case Courier) when realized upon a datagram protocol has a considerable performance advantage over a virtual circuit protocol.

With a virtual circuit protocol, a substantial amount of time is spent in maintaining state information at both ends when transmitting a large number of messages. The benefits of virtual circuit protocol occur when large data volumes are to be transferred. High volume data transfers are rare in our system. They could only occur if strings were passed (by copying as they are immutable) to emulate file transfers or the language constructs **transcopy** and **assign** were used. Furthermore, when considering the fact that an RPC can be accomplished by the exchange of

four messages, the cost of establishing and maintaining a virtual circuit for every RPC seems rather unjustifiable. Given that the existing communication protocols (on Unix) are either datagram or virtual circuit, it was justifiable to design and implement our own protocol which basically is datagram oriented but augmented with a minimum housekeeping for the purpose of transferring large volumes of data. The main design goals were:

- *Portability* In a distributed system, it is unreasonable to assume a homogeneous environment. The mechanism and therefore to a certain degree the concurrency model are rendered useless if a certain protocol is not supported on a particular machine.

- *Flexibility* Transmitting parameters whose number varies from call to call and copying parameters such as strings which have variable characters require messages to carry with them size information. The interfaces to the protocol do not place restriction on the number and size of parameters passed.

- *Fragmentation Transparency* A communication protocol often imposes an upper bound on the amount of data that can be sent in a single transmission. Transmitting large objects requires more than a single transmission. In our case, under normal circumstances this will not happen. But because of objects such as images which can be very large, fragmentation of data occurs when they are required to be transmitted for a remote image operation. Although complex logic is required in any case, it is more efficient to build it into the protocol. The senders and receivers are therefore not aware of fragmentation.

- *Efficiency* Our protocol can be tailored to our own needs. Although this means the protocol is highly specific, it is a worthwhile trade-off for efficiency. Efficiency is probably the most important factor in deciding whether or not an RPC mechanism is an acceptable one.

§5.7.1 The Protocol Handler

The protocol is designed for connectionless communication. All RPC messages are delivered using datagrams which can arrive in random order. The functionality of the protocol roughly subsumes that of the network layer and the transport layer of the ISO 7-layer model. It is responsible for (re-) transmission, acknowledgement, fragmentation and addressing. Moreover, it is reliable and guarantees that all messages directed to processes are never delivered more than once. The overall aim of the protocol is to hide away low level communication mechanics and to make RPCs indistinguishable as far as possible from ordinary procedure calls.

The interfaces to the protocol place no limit on the amount of data that can be sent or received in a single attempt. There is no possibility of buffer overflow since the protocol handler has unrestricted accesses to the heap. All data can be passed to the protocol very efficiently through pointers. Furthermore, results are pushed directly onto the stacks of the appropriate processes.

The protocol is administered by a protocol handler. Control is passed to the handler from time to time. There are two interfaces to the handler. They are defined by the following C routines.

```
rpcWait()
{
        int nread, rmask;
        struct timeval wait;

        wait.tv_sec = Timeout;                    /* a universal constant */
        ...
Again:
        rmask = 1 << sock;                        /* a global variable */
        nread = select(32, &rmask, 0, 0, &wait);
        if (rmask & (1 << sock)) {
          if (getMsg() != 1) goto Again;
        } else {
          sleep(1);
          ...
          goto Again;
        }
}

rpcTry()
{
        int nread, rmask;
        struct timeval wait;
```

```
        rmask = 1 << sock;
        wait.tv_sec = 0;
        ...
        nread = select(32, &rmask, 0, 0, &wait);
        if (rmask & (1 << sock))
          getMsg();
}
```

The only difference between the two is that rpcWait() guarantees when it returns there is a readily runnable process for selection. It is called when there is no more runnable process but some process is suspended due to communication. rpcTry() is called to ensure fairness among communicating and non-communicating processes. It is called at regular intervals only when there is at least one process suspended due to communication. The routine makes an attempt to receive a message but it never waits. A process may become runnable as a consequence.

The protocol itself can be specified by the following C routines. The small number of interfaces can be attributed to the fact that it is highly specific. For instance, a client process, once made an RPC, is suspended and awakened automatically when the reply comes. In other words, there is never a need for a client process to attempt to receive messages. The same applies to server processes as well. They are suspended when there is no message available and automatically awakened when there is one.

```
int rpcGivePort(process)
int *process;

rpcReleasePort()

rpcGetProcess(itLevel)
int itLevel;

rpcSend(userPacket, offset, to)
call_packet *userPacket;
int offset;
rpc_addr *to;

rpcRegister(pname, mname, process)
int *pname, *mname;
proc_handle *process;

rpcRaiseExcept(pkt)
call_packet *pkt;

rpcDeleteProcess()
```

The protocol has a layered architecture. The above routines are called as a result of executing some abstract machine instructions. The following routines which constitute a lower layer are called by those above. This separation of layers arises out of functional requirements. The modification of the lower layer does not affect the one above and vice versa. The lower layer serves two functions. It is the only means whereby messages are sent and received. It also concerns the representation of packets transmitted over a communication medium. The representation of RPC messages is different from those actually sent.

```
ctlSend(from, to, type, len, off, tid, msg, keep)
rpc_addr *from, *to;
int type, len, off, tid, keep;
char *msg;

ctlRecv(msg, time)
char *msg;
int time;
```

Fragmentation occurs transparently at this layer. It happens when the message size is found to be too big to be transmitted with a single datagram. Moreover, messages are retained in case they need to be re-transmitted.

§5.7.2 Packet Format

The layout of a protocol packet is given here as a C type definition.

```
typedef struct {
        u_short     versionNo;
        u_short     fragment;
        u_short     totalSize;
        u_int       checkSum;
        u_int       seqNo;
        u_int       type;
        rpc_addr    src, dst;
        u_int       pOffset;
        u_int       tid;
        char        msg_buf[BUFFERSIZE];
} Packet;
```

The functionality of each field is explained below. First of all, the fragment field is further broken down into two subfields: fragmentBit and fragmentOffset. If the fragmentBit is set it indicates that the packet is a fragment of a large message. fragmentOffset indicates which section of the original message the fragment belongs to. Every fragmented packet carries the size information of the entire message. Under normal

circumstances, the size value should be equal to the number of bytes actually received. Otherwise, this is used to calculate the expected number of packets required for the transmission of the message. When all the expected packets have arrived, the message is re-constructed and the fragmentOffsets are used to establish the total order of the packets.

All packets carry a versionNo. This must occupy the first 2 bytes of the packet. A packet is discarded if it carries an out-of-date versionNo. A change in the format of the packet will be reflected by the versionNo it carries.

totalSize is the sum of the length of the header and the size of user data. Size information is always expressed in bytes.

checkSum is normally redundant because Interface Message Processors (IMPs) often perform Cyclic Redundant Code (CRC) checks to detect corrupted packets. But it is included here in case serial lines where no CRC check is performed are used.

A message may be transmitted more than once. seqNo is used to prevent messages from being received more than once. This is important for an at-most-once semantics. The seqNo of a packet is normally unique. However, fragmented packets of a large message carry the same seqNo so that it can be determined they all belong to it.

The type field is used to determine how the packet is to be interpreted by calling upon the appropriate service. The services supported are listed below as manifest constants in C.

```
#define Declare          1
#define Confirm          2
#define Delete           3
#define Failure          4
#define Inquiry          5
#define IReply           6
#define CreateRequest    7
#define ReturnHandle     8
#define CreateProcess    9
#define Call             10
#define Ack              11
#define ReturnIB         12
#define ReturnR          13
#define ReturnP          14
#define ReturnPr         15
```

```
#define ReturnV         16
#define RexRequest      17
#define RexReturn       18
#define Probe           19
#define Alive           20
#define Except          21
#define ImRequest       22
#define ImReply         23
#define AckFrag         24
```

The src and dst are the network addresses of the sender and the receiver respectively. These addresses are protocol specific and are understood by the protocol handler only.

pOffset specifies an offset into the user data area where pointer parameters can be found. This field gives the protocol a particular favour in the transmission of pointer and scalar parameters.

The tid field is for holding a transaction identifier. It is reserved for future use.

§5.7.3 Semantics

The protocol handler that determines which service to call upon the receipt of a message, characterizes the functionalities of the protocol. It is where messages with the exception of certain types are acknowledged. Although its behaviour directly reflects the semantic requirements of the remote procedure call mechanism, it has no state. In particular, it never waits.

This appears to be contradictory to what is required, for instance, by the at-most-once semantics. A remote procedure call is accomplished by the exchange between the two sites of four messages of the following types: Call, Ack, Return, Ack in that order. After the transmission of a Return message, a process cannot continue until an Ack message has been received. This is explained below. The protocol handler cannot afford to wait since no assumption can be made about the time of arrival of an expected message. Other processes may be held up if the protocol handler waits. The solution is to require the process to remain suspended while other processes have a chance to run. If an Ack has arrived, the process concerned is then made runnable. On the other hand, it is guaranteed that a process is never suspended forever.

Furthermore, in this case, both the Call and the Return messages are remembered for the purpose of retransmission. If the expected Ack did not arrive within a reasonable period, an exception is generated and propagated to the process concerned. The functionality of the second Ack is less than obvious. It is required since an execution caused by a Call message may have side-effects. Hence, if it did not arrive in time an exception is propagated to the process (which causes the transmission of the Return message in the first place) for remedial actions.

Chapter 6

Language Implementation

§6.1 Introduction

The implementation of DPS is in two parts: the compiler and the interpreter which simulates a hypothetical machine designed for the efficient execution of programs written in the language. The designs of DPS and the hypothetical machine have been covered in two previous chapters. The present chapter describes an implementation overview of some of the intriguing features of the language.

The compiler is based on the PS-algol compiler which is a recursive descent compiler [Davies & Morrison 81]. The advantage of recursive descent compiling is that there is no need for the construction of explicit trees or tables for the different phases of a compilation. Typically, recognition, typechecking and code generation for a construct are all together in a single routine and the structures that are currently being processed are represented by the nested set of frames of these routines. Consequently the recursive descent technique yields a compiler conveniently structured for development and experiment.

The compiler generates DPS codes of an abstract machine. The abstract machine is implemented in C. The implementation is organized as a collection of modules; adding a new module or changing an existing module is relatively easy with automatic recompilation of dependencies facilitated by 'make'.

§6.2 Process Template

A process template can be considered to be a sequential PS-algol program which may be able to communicate either by remote procedure calls or through global environments. The compilation of a process template is concerned with its communication capability and is otherwise straightforward.

The communication capability through RPCs is indicated by the optional specification part. The name and type of every entry encountered in the specification part of a process template is remembered in a signature list. A stack address is allocated to each entry encountered so that all entries occupy the starting addresses of the stack of a process. The reason for this is to allow automatic deduction of stack addresses of entries required in the

dynamic binding of separately compiled processes.

The declaration of an entry in the specification part is reminiscent of a forward declaration in S-algol. But an entry has to be defined before it can be referenced. The order in which entries are defined need not be the same as they were declared, though. Although it is not statically feasible to assure an entry will be used in interprocess communication, the compiler ensures that all entries declared are defined. If an entry does not match its type indicated in the specification, it is not considered to be defined. An error is raised in either case. The constancy of an entry binding is determined at the time it is defined.

The compilation of an entry is similar to that of a procedure except that codes generated are slightly different reflecting the semantic difference between the two.

The successful compilation of a process template results in the generation of a code vector of a fixed format. The code vector consists of, in addition to the code for the body, an indication of the number of entries defined, a vector of procedures and entries defined in the body, a vector of string literals introduced in the body and a vector of indirection tables as arranged in the following diagram.

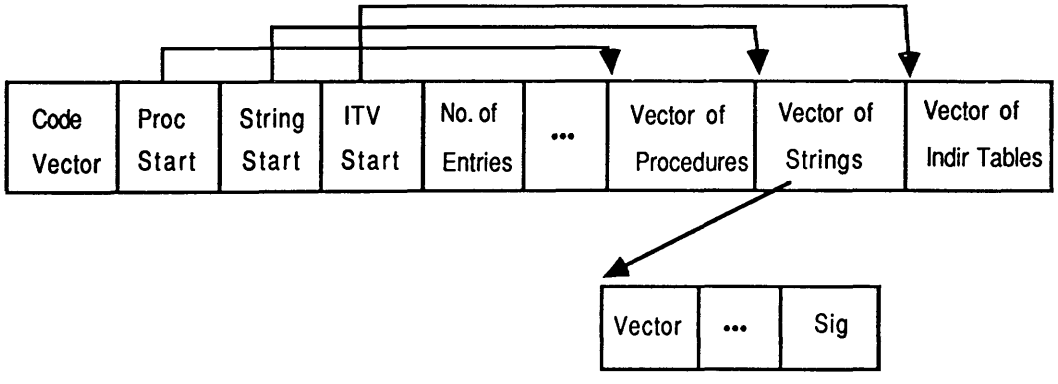


Fig. 6.1 Code Vector

An indication of the number of entries is required for the following reason. In order that communication may occur as soon as a process is spawned, it is imperative that messages sent to it are not lost. Messages are kept in queues which are associated with entries. Because entries always occupy starting addresses of a process stack and because the number of these entries is known, it is possible to initialize those queues during the creation

of a process.

In order to facilitate dynamic typechecking of communicating but separately compiled processes, the signature of a process has to be made available at run time. It is embedded in the code vector of the associated process template. It is placed as the last element of the vector of string literals for conveniency.

Indirection tables are used to obtain true stack addresses of entries of separately compiled communicating processes. This has already been covered (§5.6.2). There is a vector of indirection tables associated with a process in whose environments handles to remote processes are introduced. The size of each table is proportional to the signature of the remote process concerned. These tables are initially empty and they are filled at run time. For the sake of more compact codes, the vector of indirection tables is represented by a vector of integers containing size information of individual tables. These tables are then created at runtime and the vector of integers is replaced by the vector of these tables.

§6.3 Remote Procedure Call

An RPC is initiated by the stipulation of a process handle expression, an entry expression and parameters to be passed. Depending on the locality of the process handle, different codes are generated. In the case of process communication within the same address space, messages can be directly queued without involving the underlying network. This is considered to be an optimization only since in general a message destined for a local process will be routed back by the underlying network. The compiler keeps track of locality information of process handle variables in the symbol table. A process handle assignment results in an update on the locality information in the symbol table. In case the locality of a process handle cannot be resolved statically, it is regarded as remote. The following is an example of such a case.

```
let ph := if boolExpr then localProcessHandle else remoteProcessHandle
```

At runtime, calls to local processes are trapped and re-routed accordingly.

The type of a process is the signature inherited from the process template upon which it is created. During compilation, a signature is represented by a list of nodes each of which contains information on an entry including name, type and an offset from the start of a stack. In the case of remote processes, the offset is deduced from a signature which is verified

at run time.

For local communication, the typechecker ensures that the process to which a message is to be delivered supports the target entry by inspecting its signature list. If the signature list contains the entry named, the offset is used as an operand in the code generation so that the message can be directly queued. A type error is raised otherwise.

For remote communication, the same typechecking routine is applied. Although communicating processes may be separately compiled, there is enough type information available at compile time. Because of this, typechecking of messages received at run time is factored out. This makes process communication over the network more efficient than otherwise possible.

The offset of a remote target entry is deduced from the signature of the remote process. Signature matching occurs whenever two separately compiled processes try to communicate with one another. It is based on a notion of inclusion to accommodate certain types of changes between program executions. Although any such changes could require alterations to stack addresses of entries, this is overcome with the use of indirection tables as explained in a previous chapter. The offset of a remote entry established by the compiler is in fact an index into the appropriate indirection table where true stack addresses of entries can be found. The compiler keeps track of the indirection table for a remote process. This is accomplished simply by using an integer variable as only one remote process handle is introduced in one <install> statement. The variable is incremented when entering into such a block and decremented upon exit. The integer value is used as an index into the vector of indirection tables. The index to a particular table in the vector of indirection tables is generated as an operand to the instruction *rpcGet*. The execution of this instruction results in true stack addresses of entries being recorded in the appropriate indirection table prior to a handle to the remote process being pushed onto the stack.

The extent of a remote process handle is not limited to the block in which it is introduced. It can be passed to an outer environment by assignments or become persistent. Because of this, indirection tables are retained at all time.

§6.4 Install

In DPS, the separate compilation of communicating processes may occur

autonomously and without appeal to an external medium for type information. In order to access type information distributed over a network, a compiler for DPS written in DPS itself will be required to take advantage of distribution provided in the language. The following arrangement is considered to be a means leading to the bootstrapping of DPS.

In order to communicate with a non-local process, a handle to it has to be introduced into the environment. This can be achieved with an <install> clause described in §4.3.7. In such clauses, the types of the remote processes have to be specified as signatures to be verified at runtime. The type information of these signatures is then used in the compilation of subsequent clauses. In order that these signatures can be verified at run time, they are stored as string literals and loaded onto the stack prior to the execution of clauses which may depend on them. The true signatures of the remote processes which are received at run time are then matched against those on the top of the stack. If they match, handles to remote processes are then loaded into locations on the stack to effect the bindings. A run time type error exception occurs otherwise. Strings are used for the representation of signatures, even though another data structure could retain more of the parse information, because the signatures have to be sent over the network and using strings then avoids a bootstrap problem.

§6.5 Accept

A communicating DPS process receives messages by executing **accept** statements which specify some collection of entries of interest during the course of computation. The effect of such executions is either the acceptance of exactly one message, the suspension of the process if there is no message or the continuation of the process even if there is no message. Whether or not a process is suspended when there is no message available depends on which one of the two forms of **accept** statement was used. Different codes are generated to produce the desired effect.

In order to ensure that any entry has the same right in respect of communication, the process of selecting an entry with messages cannot be biased. Non-determinism is used as a means to obtain a certain degree of fairness and is built into the semantics of the construct. The implementation uses a simple method, which may be expensive in terms of space and time on rare occasions, to simulate the effect of non-determinism. However, the method always succeeds with the selection of a message in a single attempt. The method consists of two phases: collection and extraction. First of all, all entries which have at least one message queued are collected in a set. The cardinality of the set is then determined. In order

to obtain a random number within the range of the cardinality, the number returned from a random number generator is multiplied by the cardinality of the set and a ceiling function is applied on the result. The random number is then used in picking out an entry from the set. If the cardinality is found to be zero at the end of the first phase, the second phase is not carried out. The executing process may then be suspended.

Parameters passed in an RPC are not directly pushed onto the stack as in the case of procedure calls. They are kept in data structures of a recognizable format which imitates the stack for efficient data transfer. Such structures cannot be directly queued at the target entry in the case of remote communication. They have to be transmitted in a linear form and are reconstructed at the target end into the appropriate format. Besides the parameter to be passed, the message transmitted also includes the network address of the target process and a stack address so that the target entry can be located. Architectural differences among machines on the network can be overcome by translation. Assuming a set of canonical representations of values, the translations to and from them occur at the sender and the receiver ends respectively. The format of the data structures containing parameters may vary from one machine to another.

§6.6 Lock

The **lock** clause offers a tool for constructing atomic actions. Because of concurrency, the simultaneous execution of atomic actions may therefore result in deadlocks. A solution to the problem of deadlocks is deadlock avoidance. If some collection of atomic actions exhibit deadlockable behaviour, only one of them can be carried out at any one time. The use of the **lock** clause over **mutex**es is a cost-effective means of reducing the probability of deadlock. Such a reduction makes it reasonably easy for the programmer to adopt a programming method that cannot introduce deadlock, and is not as expensive as complete deadlock avoidance.

A **mutex** is represented by a queue of processes together with an indication as to whether or not it has been acquired. Processes that cannot acquire a **mutex** on behalf of some atomic actions are put onto the queue. Upon completion of an atomic action, including exceptions and errors, all the **mutex**es acquired for the executing process are released and a process in each of the queues concerned may then scheduled to run. If a process cannot acquire all the required **mutex**es in a single attempt, it is placed in the queue where the associated **mutex** cannot be acquired and all the **mutex**es acquired so far are released. A release of a **mutex** will, eventually, trigger one of the queued processes to make another attempt in acquiring all **mutex**es

required.

Two atomic instructions, Get and Release, are required in the code generation for the compilation of a **lock** clause. The atomicity of the two instructions is realized in the current implementation by virtue of the fact that concurrency is implemented by interleaving. The codes generated illustrate the method employed in effecting a simultaneous seizure of a collection of **mutexes** under such circumstances.

lock $m_1, ..., m_n$ **do** S

is translated into the following sequence of codes:

```
fjump(n)                    ! jump over release sequence and suspend
release ... release         ! (n - 1) releases
suspend                     ! wait for a mutex to be released
m1 get
bjumpf(cardOp(1))
...                         ! code sequence for getting m2 to mi-1
mi get                      ! claim mutex mi  leave false if busy
bjumpf(cardOp(i))           ! jump to release mutexes claimed so far
...
mn get
bjumpf(cardOp(n))           ! if no jump then have claimed them all
S
release ... release         ! normal and exceptional exits pass this way
```

where $\mathbb{S}$ and $\mathbb{m}_i$s stand for the codes generated for S and m_is respectively. The function *cardOp(i)* returns an integer which is the total number of opcodes plus their operands generated so far with respect to the previous $(n - i + 1)^{th}$ *release* instruction or the *suspend* instruction and there are (n - 1) *release* instructions generated prior to $\mathbb{m}_1$. So that the compiled backward jump, when executed, releases all the mutexes acquired so far and there can only be (n - 1) mutexes to be released in the extreme case. There is a **mutex** register holding the **mutex** a process is currently trying to acquire. It is set by the instruction Get. If a **mutex** cannot be acquired, the value **false** is pushed onto the stack. The instruction bjumpf removes the top element of the stack and if it is **false**, the control is transferred to the location specified. The instruction suspend deactivates the current process and places it in the queue associated with the **mutex** held by the **mutex** register. Any exception raised during the evaluation of $\mathbb{S}$ will cause the control to transfer to the

beginning of the following sequence of release instructions. This requires a proper model of finalization in the exception mechanism [Philbrow & Atkinson 87]. In practice more compact code is generated as just one release sequence is used via jumps.

§6.7 Locality

With locality as a programming concept, a machine on a network appears to be a nameable DPS machine. As in the case of any instance of the abstract machine, a process may thus start running on it as a result of a remote request. The process in question need not be known locally but a copy of its executable codes may have to be shipped prior to that request. The question is: where on a machine does the copy get installed?

The implementation keeps one or more perpetually running servers capable of launching concurrent processes. These servers are instances of the abstract machine but they never terminate automatically. Copies of values including process templates destined to a machine but not a particular address space are installed in one of these servers. All subsequent requests on these copies are directed to where they are installed. Consequently, a process started as a result of a remote request runs in one of these servers.

Localities exhibit hierarchical structure. A locality may be embedded in another locality representing the fact that a network may only be reachable from a gateway. The implementation of the relational operations on locality-related values is based on structural information used in the representation of network addresses. Recall that a network address contains a high and a low address. The high address is a process identifier and the low address is a communication port address which can further be broken down into network number, machine address etc. Relationship between localities can be discovered by comparing the network addresses of the two to a certain depth and by additional structural information which is made available as a result of embedding localities. The locality value **universe** is represented by an entity which serves as a wild card. However, locality is a relative concept and therefore there is no absolute **universe**. Hence the expression **universe**$_1$ **within universe**$_2$ yields **true** only if the former has been embedded into the latter.

A locality is represented by one of the following two PS-algol data structures, and therefore required no new abstract machine instructions.

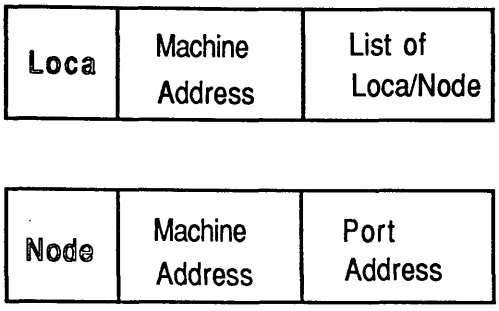


Fig. 6.2 Loca and Node

The relationship between locas and between locas and nodes can be established by inspecting a field or a combination of fields of the data structures and by the method employed in a search over the list of loca/node.

Chapter 7

Distributed Store Management

§7.1 Introduction

The distributed stores management system described in this chapter is a means to achieve the effect of shared persistent stores over a network. It differs from other approaches such as [Cockshott 85] and [Gallagher 85] in that it does not require architectural changes. Although it does address allocation, it is not meant to be a fully-fledged addressing mechanism. It takes a subsidiary role complementing existing persistent addressing mechanisms to locate data which may be on a local or a remote persistent store. Except for a minor change in the representation of objects, its existence is felt only within the existing addressing mechanism. Moreover, the existing addressing mechanism remains unchanged but enhanced. Thus the compiler has not been affected in any way. On the other hand, many features in the language DPS are based on this addressing capability.

Notionally, the distributed store management system makes the underlying network appear to be an extension of a memory bus. The effect is that the addressable scope of a process is unrestricted. Its addressable data include both local data and those on remote machines. The more volatile nature of the communication medium is made accountable by enriching the set of system events.

The allocation of universal addresses is a focal point of interest. There are two types of addresses; one for local data and the other for non-local data. The addressing mechanism for remote data depends heavily on the addressing mechanism for local data. The main characteristic of address allocation is that objects are guaranteed to be addressable from a remote site and yet not every object is given a universal address. A process which references only local data pays a very small space penalty in the representation of objects and a small time penalty of a 1-bit test on every pointer dereference.

Persistence is essential to this type of work where the export of a universal address means data residing at that address must be kept as long as it is reachable from one of a local root, a local active process, or from a remote address space. We explain below how the universal addressing mechanism interacts with the persistent mechanism. Two

methods of distributed garbage collection are also described here and each one is useful in reclaiming space occupied by obsolete objects no longer referenced from remote sites.

The manipulation of remote data is by sending requests to the sites where those data reside. This model of remote operation is elaborated further in a later section. The reasons for this choice were presented earlier (§1.2.1).

A potential weakness of the address management system is efficiency. A reason for this is the fact that it is simulated without hardware or micro-code support. As a deliberate design decision, the format of a remote address is an arbitrary length bit string with no locally known internal structure, to allow system evolution. This makes it difficult to provide hardware support for its manipulation. But the frequency of de-reference of remote pointers is observed to be a very small proportion of the total instructions and it is not yet clear that hardware or microcode support is warranted, particularly as the de-reference time is likely to be dominated by network delays. Perhaps the most useful place for hardware support is to aid the formation and handling of network messages. Reasonable performance has been obtained on a cluster of Sun-3 workstations using only the standard Unix 4.2 protocols and software written in C.

We begin by examining two approaches towards realizing shared persistent stores.

§7.2 Universal Addresses

In order to address data in store, the representation of pointers is normally encoded with some sort of locality information. This concept has so far been applied in a rather limited fashion. Traditionally, a process is constrained to access data within a dedicated virtual memory for reasons of security and protection [Organick 72]. Pointers are represented by virtual memory addresses which when given to an addressing mechanism yield the data residing at a corresponding location in the physical memory. Persistent addressing is similar as it takes pointer representations to be Persistent Identifiers (PIDs). Objects in store which includes both main and secondary memory can be addressed using these PIDs. Thus the addressable scope of a process now has a finer grain as it refers to objects. The benefit of this is that programmers see a one-level store without being distracted by the fact that objects addressed during the course of a computation may be resided in a random access

memory or on a sequential file store.

It is observed that the essence of persistent programming is in locality transparency. This facet is useful in distributed computations since there will be no difference in accessing local and remote data. In order to allow remote data to be addressed, the representation of pointers must be encoded with network addresses on top of object identifiers. In addition, a mechanism for the distribution of pointers is also required. One such mechanism has been described in chapter 5 and is therefore assumed for the rest of the discussion. The receipt of these pointers represents a right to access the remote data they refer to so that the addressable scope of a process which has already included both main and secondary memory may further be expanded in a controlled manner.

The representation of pointers extended with network addresses is considered to be more general than those without them. In realizing shared stores, a logical choice therefore will be to use them to represent pointers on stacks. Compilers at different sites will then generate such pointers with different network address values. This approach has a number of drawbacks. Firstly, the scheme calls upon a universal addressing mechanism that can interpret any pointers that can ever be generated. The widespread use of persistent stores by more than one language is likely to be hampered by the imposition of a single addressing mechanism. Secondly, local and remote data are addressed using the same pointer representation. An error in the representation inflicted during transmission over a network may be disastrous. Although transmission errors can be avoided with checks, the overhead incurred is likely to be high.

Thirdly, there is a space overhead since the network addresses in pointers to local data are redundant. It is observed that the number of pointers to local data is much higher than pointers to remote data so that the space overhead is significant.

Finally, the approach is problematic with respect to garbage collections. It was claimed that garbage collections may be avoided by providing address spaces so large that there is never a need to reclaim space. But this is unrealistic as the rate of object creation is not predictable. Assuming garbage collections can take place whenever and wherever necessary, a solution is to use indirection so that the exact whereabouts of data can be established at any time. However the attendant overhead is proportional to building on top of a virtual memory yet another virtual memory mechanism.

A slightly different approach, which incurs less overheads and more flexible, is to permit different representations of pointers; one for local data and another for remote data. This approach is strongly influenced by the belief that in general the number of pointers to local data is much higher than to remote data. For the sake of clarity, pointers to remote data are known as *remote pointers*. The problem of representation on stacks due to the difference in sizes can be overcome by disallowing remote pointers on stacks altogether. Instead they exist only as objects in the heap. An implication that follows from this is the flexibility of the representation of these remote pointers.

The existence of remote pointers off the stack is of no concern to the programmers. A slight modification to the existing addressing mechanism suffices to present to the programmers a uniform view on pointers. When a referend happens to be a remote pointer, a subsidiary addressing mechanism is employed to locate the remote referend.

Remote pointers are heap objects. They can be made persistent and used subsequently; just like any heap object. Since the existence of remote pointers is of no concern to the programmers, it follows that a logical universal persistent store is obtained but participating persistent stores remain autonomous.

It is important to note that as long as a remote pointer exists, its referend must be available. Persistence provides the necessary facilities for keeping data referenced from remote sites.

For the rest of this chapter, it is understood that the representation of remote pointers contains three values:

- a network address,
- an object identifier and
- a persistent identifier.

The network address identifies an instance of the abstract machine on a particular machine in a network. The object identifier is used to locate a piece of data in the heap of that remote instance of the abstract machine. An instance of the abstract machine ceases to exist when all processes running on it have terminated. The persistent identifier is a key to locate the same piece of data in a remote persistent store if that particular instance of the abstract machine ceases to exist. It should be pointed out

that the persistent identifier and the object identifier do not address the same object. This will be explained in due course.

The representations of each value and their space requirements are specific for an implementation and they can be changed without impact on existing programs. Indeed they are allowed to differ when different remote localities are addressed so that the network may be heterogeneous and may evolve. However, this aspect is immaterial for the discussion which follows.

§7.3 Address Allocation

It is implicit in the discussion above that remote pointers could not be generated by the compiler which only allocates pointers to data in the local heap. In fact, the compiler requires no adaption for remote pointers. Recall that pointers to non-local data can be obtained through a distribution mechanism, which in our case, is an RPC mechanism as the distribution of data is initiated under the control of the programmers. It follows that the only occasion when remote pointers are required is when data are exchanged in communication across address spaces. Ordinary pointers are replaced by remote pointers which are encoded with more locality information at foreign sites, but immutable values such as integers and strings are copied. The rationale for this is efficiency and we arrange that the semantics, including equality, remains unchanged.

There are pitfalls in this lazy generation approach. An object which is passed as a separate parameter or on different occasions could be given different remote pointers referring to the same referend. This conflicts with the store semantics in which each heap object is uniquely identifiable. If objects are uniquely identifiable, equality can be defined very cheaply in terms of references without requiring examination of the behaviour of objects. For instance, function equality can be defined, albeit not very satisfactorily, by comparing the closures. Although it is still possible to establish equality by inquiring at the original sites, this appears to be rather clumsy and expensive. A more efficient solution is to ensure that a unique remote pointer is associated with each object. Once a remote pointer has been generated for an object, it is remembered and will be reused whenever a pointer to it is encountered again as a parameter or result in a communication.

There is still a further complication. A remote pointer is represented on the stack by a local reference and thus could be passed as a parameter

in an RPC. If such is the case, a chain of remote pointers one referring to another could result. Consequently, an object can only be located, by tracing back the path through which it was imported. This leads not only to more messages, slower responses but also the situation where an intermediate node along the path becomes in-accessible while the root where the referend resides is reachable. The problem is easily solved by copying remote pointers, when passed as parameters or results, as if they were immutable objects instead of generating new ones. This scheme works since remote pointers are meant to be universally recognizable. This contrasts with Hurst's approach [Hurst 87] in which indirection is possible in case the root itself becomes unreachable.

An associative structure at the sender is used to keep track of remote pointers generated: Its used to prevent re-generation of remote pointers. This structure is shared by all processes within the same address space. Pairs of the form <xid, local reference>, where xid is a context sensitive object identifier, are held in the structure. Local references refer to data in the local heap. The validity of these local references is guaranteed despite garbage collections.

The structure at the sender end holds remote pointers generated and references with which they are associated. It is called the Export Table. The organization of the table is such that there is a partition for a group of communicating processes whose network addresses refers to a common instance of the abstract machine. A communicating process here is the one to which a remote pointer was or is about to be exported. Such an organization is for the purpose of efficient searching and distributed garbage collection, and has no semantic implications. It is speculated that export of remote pointers exhibits a locality of reference similar to the one associated with paging systems.

A remote pointer may be exported many times. Thus multiple occurrences of the same remote pointer may be found in the same heap. It is possible to keep track of remote pointers imported in a table to reduce the number of identical copies of a remote pointer to one. This is not considered because of its implication on distributed garbage collections. Distributed garbage collections are discussed in a separate section.

The algorithm for the generation of remote pointers works as follows. For every RPC made by a lightweight process, every pointer parameter X is used as a key in searching for an object identifier which may have been generated for it. The search is conducted first of all in the

partition associated with the communicating process, if allocated, and then the rest of the Export Table. If found, a remote pointer based on the partition and the object identifier is constructed and then returned. Otherwise, a partition is allocated if necessary and X is entered into the first available slot in the partition and the offset of the slot relative to the partition is used as the object identifier in generating a remote pointer as above.

§7.4 Remote Store Operation

A remote pointer contains information to allow the identification of a referend at a remote site. In this section, we describe how updates on remote referends are carried out. First two strategies that were investigated but rejected are described:

> 1) When a remote pointer is "dereferenced", the remote referend can be copied into the local address space. A local reference to a temporary object holding the copy is then used to replace the remote pointer and the execution of the dereferencing instruction is carried on this temporary object. The resultant copy is then sent back to the original store to effect the changes. This method transfers a message containing the object in each direction every time it is required. This may affect performance in, for example, a tight-loop situation where the same object is iterated over again and again.
>
> 2) Instead of sending back the copies to effect the updates, they can be cached. This sort of approach has been used in various guises in different problem domains. The usual problem is multiple copies update. A number of solutions have been discussed in [Holler 81] for maintaining various degrees of consistency among copies. A simple one is not to allow more than one copy to be made at any one time by using a lock protocol. A request for a copy of a locked object is ignored until the lock on it has been released.

Note that copying an object at this level is less tricky than copying the referend of a pointer parameter in a remote procedure call. If a pointer parameter happens to be a root to a large database, the entire database might be copied across. On the other hand, a store level instruction always has a target object to operate upon. For example, the

operands to a vector update instruction are a reference to the vector, an offset and a value to replace an existing element of the vector. The only value which needs to be copied is the vector. So there is no danger of copying data irrelevant to the computation.

The copying approach has no major engineering obstacles. However, it is considered to be less attractive compared to the remote store operation approach for the following reasons. Firstly, copying requires translation in a heterogeneous system, but a substantial amount of translation work done is wasted as typically only one field of a data structure is the focus of interest. Secondly, the lock-copy-return-release approach is a questionable policy. The release of locks cannot be determined at the site where they are held. The return of a modified copy is subject to network partition or machine failures. The time elapsed before locks are released may be arbitrarily long thus holding up other requests. Furthermore, instructions requesting copies need not be eagerly executed as there may be processes waiting for execution. It is therefore impossible to impose a time-out period upon which locks can be released unilaterally.

Finally, there is an efficiency problem. A tight-loop manipulation of an object, say, incrementing individual elements of a sizeable, one dimensional array of integers, would require the sending to and fro of the same object over and over again. However, this may be overcome by cacheing as an optimization.

The remote store operation approach avoids copying. A similar model was described in [Spector 82]. Instead of copying operands, the de-referencing instructions are sent and executed at the sites where the referends reside. There is an assumption here that the same set of instructions are used at all sites participating. This assumption is not invalid even on heterogeneous systems since the same abstract machine can be used. This approach still requires some copying to be done but the amount is insignificant. For example in the vector update instruction above only the operand to be used to effect the assignment is required to be copied. In keeping with the policy of not copying objects, the value may be replaced by a remote pointer. Indeed, what is interesting here is that a store operation can be seen as a primitive procedure supported by the store. A remote store operation resembles an RPC with exactly the same kind of semantics.

The remote store operation approach is attractive as there is only one copy of data at any time and therefore the problem of multiple copies

inconsistency simply does not exist.

In the remote store operation approach, there is still a need for a locking protocol. A request for a store operation to be executed is not served unless the operand in question is not locked. Locks are acquired and released according to the execution behaviour of programs. The language described earlier does not support explicit locking of individual objects. The effect of locking is achieved by seizing and releasing mutexes. The idea is that such objects are meant to be keys the seizure of which permits exclusive accesses to some collection of data. Accordingly, requests for store operations are carried out without observing any locking protocol because it is assumed that appropriate interlocks have been arranged by the programmers' use of mutexes. If some data is meant to be accessed sequentially, it should not be exported on its own. They can be packaged up e.g. in an expression in which some locking convention is observed.

In terms of implementation, the remote store operation approach is appealingly simple in its own right. It has been observed that about 14% of the 256 abstract machine instructions require a remote implementation. Of course, a much smaller percentage of the instructions executed refer to remote store. It is believed that the small percentage is true in general since most machine instructions are for control jumps, loading literals, comparison and arithmetic, i/o and stack manipulation. The 14% remote executable instructions, in the case of our abstract machine, includes those manipulating heap objects eg. structures, vectors, and graphical objects. We outline the operational semantics of remote execution of store instruction on structures and vectors for didactic purposes. For images, it is slightly more complicated but the principle is the same.

Operands to those instructions are typically an object whose reference is near the top of the stack, an offset, and in the case of assignment another object whose reference is at the top of the stack. If the object in question is a remote pointer, the executing process is suspended. The dereferencing store instruction is identified which is the current value of the program counter. The program counter is incremented to point to the next instruction and the stack operands are popped off. Now the state of the process is dumped for use in subsequent resumption. The remote pointer object contains the network address of where the referend resides together with a context sensitive object identifier. A message containing all the operands and an opcode which identifies the

dereferencing instruction, is sent to the target machine. All operands except those for identifying the target object and field, are treated the same as parameters of an RPC

The referend in question can be located through the Export Table. The object identifier in the remote pointer is made up of a partition number and an offset. The partition number is used to locate the partition in the Export Table and the offset is used to index into the partition to obtain a reference to the referend. The instruction requested is carried out as it would have been at the site where the remote pointer is dereferenced had the referend been available locally. The execution is carried out by some surrogate process of the instance of the abstract machine where the true referend resides. An RPC message is sent back upon completion of the instruction.

§7.5 Persistence

A remote pointer dereference may occur at any time. In particular, it may occur after the process which exports the remote pointer has terminated. It is therefore imperative that once a remote pointer to an object has been exported, the object should be kept available for as long as some other process has the capability of referring to this object. This requirement is easily met with the provision of persistence.

When an instance of the abstract machine terminates, all the objects reachable from the Export Table are sent back to a local persistent store. This is accomplished by making use of persistent facilities which already exist. Since an Export Table provides a context for locating an exported object, it is also kept in the persistent store as well. Under normal circumstances, a request for the execution of a store operation is sent back to the instance of the abstract machine where the referend resides. If this is not possible, the request is re-directed to a perpetual server process. Upon receipt of such requests, the server process pulls out the referends from a local persistent store and carries out the required operations as it would have done in the normal case.

The Export Table is not global to a persistent store for efficiency reasons. There is one per instance of the abstract machine. The persistent identifier in the representation of remote pointers is used to locate an Export Table in a persistent store. The persistent identifier is not redundant even when objects can be located without resort to server processes. This is because network addresses are re-usable but the confusion over the identity of an instance of the abstract machine is

avoided as persistent identifiers, which identify the Export Table that is (was once) associated with the process, are unique.

Once an Export Table is located, the object identifier can be used to locate the referend. It should be obvious that the PID for an Export Table is required to be obtained prior to any construction of remote pointers originating from an incarnation of the abstract machine.

§7.6 Distributed Garbage Collection

It has been mentioned that once an object is exported, it remains available as long as it is accessible. Persistence ensures that objects reachable from an Export Table are retained forever. Garbage collections, on the other hand, are used to free space occupied by obsolete objects. During the course of a distributed computation, garbage collections may occur autonomously in the heaps at different sites. This method of operation has a counterpart in some non-distributed garbage collection algorithms. For instance, Bishop [Bishop 77] proposed heap spaces to be separated into compartments in which objects are garbage collected if they are not referenced from objects in any other compartment. Distributed heaps with remote pointers referring to objects in them are, logically speaking, a form of compartment described in Bishop's algorithm. In our case, garbage collectors at these sites retain objects reachable from the Export Tables in addition to those reachable from the stacks. Exported objects are purged from an Export Table only when all of their outstanding remote pointers are garbage collected at remote sites.

The number of remote pointers constructed in remote heaps is kept in a reference count on the Export Table. The count is incremented every time a copy of the remote pointer is installed at a foreign site. It is decremented on receipt of messages from garbage collectors at remote sites. Messages are sent (in batches, perhaps) at the end of a garbage collection when remote pointers which are no longer referenced are garbage collected. If the count drops to zero, all outstanding copies of the remote pointer have ceased to exist and therefore the referend can be garbage collected locally. Note that only exported objects are garbage collected based on reference counts. Other heap objects are garbage collected based on a constant space, pointer reversal, mark-and-slide compacting algorithm.

Garbage collections based on reference count normally suffer from the problem of not being able to detect circular objects. Hughes [Hughes 84] proposed a distributed garbage collection algorithm based on time-stamping which can detect circular objects spanning across sites. The idea

is that each heap object is time-stamped with either the time it is created or the time which the local garbage collection is started if it is reachable from the stacks. The time-stamps of remote pointers are propagated to their referends. An object is considered marked if its time-stamp is later than or equal to the time which the current garbage collection is started. At some point in time, objects not referenced and in particular circular objects spanning across processors will lapse behind in time and therefore can be garbage collected. The algorithm requires a global clock which is hard to maintain on a distributed system, space per object to hold time-stamp information and a large number of messages to be exchanged between all sites and is therefore expensive if the occurrence of circular objects is rare. Also, in pathological cases, it may take a very long time before circular objects spanning across sites can be garbage collected. A reasonable approach is to run it occasionally to pick up circular objects not detectable in the other method of distributed garbage collections described.

Neither of these distributed garbage collection algorithm has yet been implemented, though we recognize that it is ultimately necessary to avoid space creep.

Chapter 8

Distributed PS-algol

§8.1 Introduction

Distributed PS-algol or DPS for short is a conventional, persistent programming language with supports for concurrency and distribution. The semantics of language constructs are concise with particular emphases on simplicity. The language is aimed at a wide audience. It inherits from PS-algol a rich set of data types, store semantics, persistence and higher order functions. With persistence in particular, programming efforts can be accumulated and made use of subsequently with relative ease. This chapter serves to illustrate the use of the language in the areas of concurrency, synchronization, process communication and distribution. The programming style in the examples below is our initial idea as to how such languages may be used.

Measurements on process communication are described to give an indication on the performance of the RPC mechanism.

§8.2 Examples

§8.2.1 Concurrency and Process Communication

The following is an example of a program that never terminates. It (always) contains a process that does nothing but spawn itself.

```
let forever := process
                 begin
                 end
forever := process
           begin
           let x = start forever
           end
let y = start forever
```

This simple program consists of two declarations and an assignment. The first declaration serves to introduce a forward reference. It is used in the body of the process template in the assignment. The second declaration introduces a process handle which is never used. The purpose of the second declaration is to initiate a chain of self-spawning processes.

A simple example of concurrency and process communication is given below.

```
let sema = process
           with
             P = entry(),
             V = entry()
           begin
           let P = entry(); {}
           let V = entry(); {}
           while true do {accept P; accept V}
           end

let Mutex = start sema
let outfile = open("afile", 1)
let writer = process
           begin
           Mutex@P()
           for i = 1 to 10 do
             output outfile, "This is meant to be a long sentence to be output'n"
           Mutex@V()
           end

for i = 1 to 10 do {let x = start writer}
```

In the example, output to a file is atomic with the use of a semaphore process. The semaphore process serves to illustrate the synchronous nature of process communication and is otherwise redundant since the language already supports **mutex**. Communication between *writer* processes and the *Mutex* process is initiated by expressions such as *Mutex@V()* and accomplished with clauses like ***accept*** *V*. Atomicity is guaranteed by virtue of the fact that the semaphore process accepts a message from the entry *P* followed by a message from the entry *V* and the discipline exhibited in the *writer* processes. Note the way in which entries are specified in the **accept** statements; no brackets are required.

Concurrent processes can be created dynamically whenever the need arises. This facet can be utilized to create some sort of asynchronous effect in process communication. The concept of extended rendezvous in Ada is an example in which clients can be resumed earlier than otherwise possible. The same effect can be emulated in the language. The following simple example sketches the idea.

```
let p = process
          with
            ep = entry(-> int)
        begin
        let ep = entry(-> int)
                 begin
                 let x = process
                         begin
                         whatever()
                         end
                 let y = start x
                 3
                 end
        accept ep
        end
```

It is obvious that if an entry does not return any tangible result, clients can be resumed even earlier. It is interesting to note that in contrast with Ada, the server can be resumed earlier as well since the call is served by a third process.

It is possible to exploit dynamic processes further in this vein. A server may spread its workload to delegate processes. In extreme cases, it is even possible to create the impression that a server appears to be able to handle messages for different entries at the same time.

```
let p = process
          with
            ep = entry(...)
        begin
        let ep = entry(...); ...
        let x = process
                begin
                while true do accept ep
                end
        let y = start x
        other()
        end
```

It should be realized here that it is not possible to create a process just for the sake of serving an RPC because of the semantics of **accept** and in the absence of a facility to peek at the arrival of messages. On the other hand, there is not much gain in economy in doing so. This is because in the absence of messages, an **accept** causes the executing process to be suspended until a message has arrived. Moreover, a server process like *y* in the example above avoids the need to create a process every time a message arrives.

The example below illustrates how a process can be started on a remote machine:

```
let p = process
          with
            ep = entry(-> string)
        begin
        let ep = entry(-> string); "hello from paama'n"
        while true do accept ep
        end

let rp = start transcopy p to paama
print rp@ep()
```

where *paama* is a predefined loca value in the language. A process template *p* is declared, and a copy of which is sent to the destinated locality. A process is started where the copy happens to be. The result of the **start** expression is a handle to a remote process. Communication with the process is by RPC.

In the example above, the process denoted by *rp* can only communicate with the process which started it. In general, a server process may communicate with a host of processes some of which may be separately compiled. A public process can be declared as follows:

```
let rp = start transcopy p to paama as "server"
```

The effect of the **start** expression is the same as before but in addition the process created is known as "server". Any process can communicate with it as in:

```
for s = "server" at paama with ep = entry(-> string) do print s@ep()
```

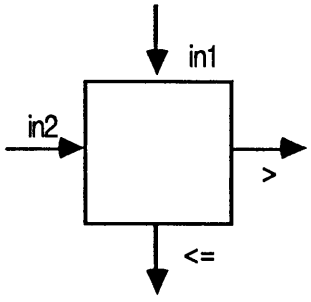
A handle to the process known as server is introduced into the environment with a **for** clause. The remote process must support an entry named *ep* which does not take any parameter but returns a string as the result. If it does, a binding occurs so that *s* denotes a handle to that process. Communication with it can be accomplished as in the example above.

§8.2.2 A Concurrent Sorting Algorithm

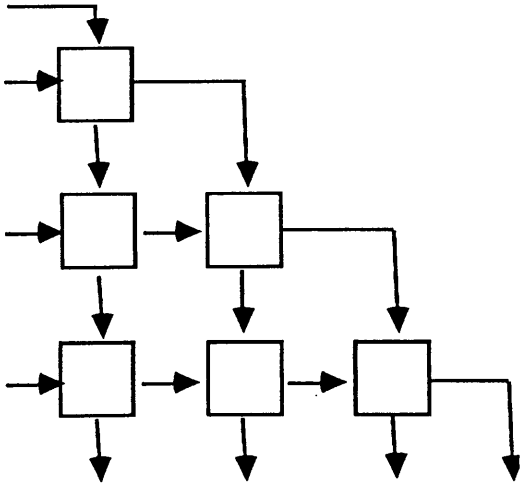
The sorting of a sequence of comparable values can be accomplished by a network of concurrent sorting processes [Sheeran 85]. The idea is that the network of processes produces a sieving effect and the larger the value,

the longer it stays in the network. Smaller values are sieved out through the lower echelon of processes of the network. The number of comparisons required in such a sorting network tends to be higher than normal. But the main advantages are that there is no need to remember previous results and that hardware parallelism may be exploited.

The task of a sorting process is simple. Upon receipt of two values, it makes a single comparison and passes them on to its neighbour accordingly:



A network of efficient sorting processes may be difficult to construct in general. The concurrent sorting network presented here can be constructed algorithmically according to the number of inputs because of its geometrical shape. For example, a network of sorting processes for an input of 4 values looks something like:



The following program implements a concurrent sorting algorithm using these sorting processes. The algorithm capitalizes on the first class nature of process handles.

```
let sorter = process
               with
                 in1 = entry(int),
                 in2 = entry(int),
                 init = entry(int, int)
             begin
             let min := 0; let max := 0
             let x := 0; let y := 0
             let in1 = entry(int a); max := a
             let in2 = entry(int b); min := b
             let init = entry(int u, v); {x := u; y := v}
             accept init
             while max = 0 or min = 0 do accept in1 || in2
             if max <= min do {let tmp = max; max := min; min := tmp}
             case true of
               x = 1 and y = 1        : write min, max, "'n"
               x = y                  : begin
                                          let s := select(x - 1, y - 1)
                                          s@in1(max)
                                          s := select(x, y - 1)
                                          s@in1(min)
                                          end
               y = 1                  : begin
                                          write min
                                          let s = select(x - 1, y)
                                          s@in2(max)
                                          end
               default                : begin
                                          let s := select(x - 1, y)
                                          s@in2(max)
                                          s := select(x, y - 1)
                                          s@in1(min)
                                          end
             end
```

Each sorter process is made aware of its position in the network. The compared values are then either output directly or passed on to the appropriate neighbour. Note that a sorter process is not committed to receive a value from one particular entry. Any process in the network is thus allowed to fire off whenever two input values become available. The procedure *select* returns a process handle to the process at the specified position in the network. The network itself is held in a data structure.

```
structure node(pntr this, that)
structure box(ph(in1=entry(int), in2=entry(int), init=entry(int, int) sort;
              pntr next)
...
let s := start sort
s@init(1,1)
let network := node(box(s, nil), nil)
...
```

The sequence of comparable values is input to the network on the left where the number of processes is one less than the number of values in the sequence.

§8.2.3 Distributed Programming

There is no restriction on the types and number of parameters that can be passed in an RPC. The effects on parameters passed in RPCs are consistent with that in ordinary procedure calls. The following example illustrates this concept:

```
let p = process
          with
            ep1 = entry(**int),
            ep2 = entry(string -> string)
        begin
        let ep1 = entry(**int x)
                    for i = lwb(x) to upb(x) do
                      for j = lwb(x(i)) to upb(x(i)) do x(i, j) := x(i, j) + i + j
        let ep2 = entry(string s -> string); s ++ "'n"
        while true do accept ep1 || ep2
        end
let thisP = start p as "aServer"
```

Suppose *thisP* is running on *paama.*

```
let aVector := vector 1::10 of vector 1:: 20 of 0
for thatP = "aServer" at paama with ep1 = entry(**int) do
      thatP@ep1(aVector)
```

The effect of this program is the declaration of a one dimensional integer vector *aVector* whose elements are initialized to zero and it is passed to the entry *ep1* of the process *thisP*. The integer vector, *aVector,* is subsequently modified so that individual elements have values identifiable to the sum of their indexes. This is the same kind of effect normally expected in procedure calls. It is safe to assume that address spaces of communicating processes are shared even if they are on different machines. The first program, in particular, does not contain any

indication of distribution. Indeed, if the two programs are brought together on *paama* as separate programs or otherwise the same result can be obtained. Note that although *thisP* supports two entries, the program above merely uses one of them. In general, a program is required to name those entries of interest only and in whatever order these may appear in the specified signature. This allows certain changes in servers between executions without any impact on existing clients.

Distributed programming in the language is not radically different from conventional programming. The only requirement is the establishment of a "link" between processes. This can be achieved by parameters passed in RPCs. Once a parameter is received it can be accessed as if it were local. The following program capitalizes on this fact. The two programs together realize distributed free-hand drawing. The objective here is to permit whatever drawn on a screen of a machine to appear immediately on a screen of another. The two programs share the same free-hand drawing package which is stored in a persistent store. In addition to setting up the screen into quadrants, the package provides a set of paint brushes and supports pop-up menu. The display uses two of the quadrants; one for local drawing and the other for displaying remote images. The main body of the package looks something like:

```
...
while true do begin
   moose := locator()
   if moose(the.buttons)(the.button) do begin
     let x = moose(X.pos)
     let y = moose(Y.pos)
     if x < bum.r then if y >= bum.b do controls(x, y) else
     if inBrushSet() then change.brush() else
     if inLocalDisplay() do ror brush onto limit paper at x, y
     end
   copy their(the.paper) onto rpaper
   end
...
```

Depending on the position of the cursor, the package performs various activities such as popping up menu, changing the brush or displaying the brush along the path of the cursor movement. The two display areas are represented by images of type **#pixel**. They are identified as *paper* for local drawing and *rpaper* for displaying remote images. Distributed free-hand drawing is realized basically by running the package on different machines at the same time and that the expression *their(the.paper)* is made synonymous to *paper* of the other side. This is accomplished by the following two simple programs:

```
let root = open.database("demos", "friend", "read")
...
let make = s.lookup("sketch pad", root)(content)
structure paper.box(#pixel the.paper)
let their.paper := image 1 by 1 of off
let my.paper := image 1 by 1 of off
let their = paper.box(their.paper)
let mine = paper.box(my.paper)
let sketch = make(their, mine, ...)
my.paper := mine(the.paper)

let p = process
          with
            init = entry(#pixel -> #pixel)
        begin
        let init = entry(#pixel theirPaper -> #pixel)
                   begin
                   their(the.paper) := theirPaper
                   my.paper
                   end
        accept init
        end
let aP = start p as "server"
sketch()
```

and

```
let root = open.database("demos", "friend", "read")
...                                        ! same as above
my.paper := mine(the.paper)
let x = process
         begin
         sketch()
         end
let y = start x

for p = "server" at vanuata with init = entry(#pixel -> #pixel) do
   their(the.paper) := p@init(my.paper)
```

Instances of the structure class *paper.box, their* and *mine,* are used to convey pointers to display areas between the programs and the package. The package is retrieved from a persistent store and is initialized with *their* and *mine* as two of the parameters. After the initialization, *mine* contains a pointer to the local drawing quadrant. One of the programs then initiates a communication with the other passing to it the pointer and the other responses by replying with the counterpart which is made synonymous with the expression *their(the.paper).*

The free-hand drawing package is not oriented towards distribution. Indeed, if there is no communication between the two processes, free-hand drawing is still possible except that the image will not be reflected on the other machine. The objective of the exercise is essentially accomplished simply by the clause ***copy*** *their(the.paper)* ***onto*** *rpaper.* Note that **copy** is the usual graphic construct in the language.

This concurrent program was constructed by making minor changes to a freehand drawing package written as a demonstrator by Paul Philbrow. The whole programming task, design, and implementation was done in a single day.

§8.2.4 Input and Output

Lightweight processes share the same set of resources of the underlying machine. In particular, they share the input (keyboard by default) and output (screen by default) streams together with the mouse. In some cases, the concurrent uses of these streams and the mouse need to be coordinated. This can be achieved with a lightweight process whose sole function is to regulate the uses of the two streams and the consumption of mouse events. Any other process wishing to perform coordinated input or output activity must communicate with it. Because communication is synchronous, the sequential nature of input and output is ensured. Because of graphics, however, the concurrent uses of (different parts of) the screen in some cases can only be organized by the programmers.

```
let io = process
           with
             iStart = entry(),
             iStop = entry(),
             iLocator = entry(-> pntr),
             oStart = entry(),
             oStop = entry()
         begin
         let i := false
         let o := false
         let iStart = entry(); {i := true}
         let iStop = entry(); {i := false}
         let iLocator = entry(-> pntr); locator()
         let oStart = entry(); {o := true}
         let oStop = entry(); {o := false}
         while true do begin
           accept iLocator || iStart || oStart
           if i do accept iStop || iLocator
           if o do accept oStop || iLocator
           end
         end
let ioController = start io
```

The function of the process *ioController* is to oversee the proper use of the input and output devices. Any coordinated input and output operation must be preceded with a *Start* request and followed by a *Stop* signal to the process. In the case of the mouse, the process guarantees that mouse events are not shared by more than one process.

§8.2.5 Dining Philosophers

Here is another solution to the ubiquitous dining philosophers problem:

```
let fork = process
            with
              pickUp = entry(),
              putDown = entry()
           begin
           let pickUp = entry(); {}
           let putDown = entry(); {}
           while true do {accept pickUp; accept putDown}
           end
let Fork = start fork
let forks := vector 1 :: 5 of Fork
for i = 2 to 5 do {forks(i) := start fork}

let seats := vector 1::5 of true
let x = mutex
let ok = cond
let join = proc(int s)
           begin
           lock x do begin
             while ~(seats(s) and seats(s + 1 mod 5)) do wait ok
             seats(s) := false
             seats(s + 1 mod 5) := false
             end
           end
let leave = proc(int s)
            lock x do begin
              seats(s) := true
              seats(s + 1 mod 5) := true
              signal ok
              end

let think = proc(); {...}
let eat = proc(); {...}

let phil = process
            with
              init = entry(int)
           begin
           let s := 0
           let init = entry(int seat); {s := seat}
           accept init
           while true do begin
             think()
             join(s)
             forks(s)@pickUp(); forks(s + 1 mod 5)@pickUp()
             eat()
             forks(s)@putDown(); forks(s + 1 mod 5)@putDown()
             leave(s)
             end
           end

for i = 1 to 5 do {let dum =  start phil; dum@init(i)}
```

The dining philosophers problem is a classical example of race condition with the usual pitfall of deadlocks. In the example above, the behaviours of identical forks and identical philosophers are captured by their respective process templates. A fork can either be picked up or put down. The life of a philosopher is either to think or to eat. In order to eat, she is given a seat at a round table. She also needs two forks and one of them is shared with her neighbour. Thus a philosopher cannot proceed with her meal until two forks can be acquired at the same time. The synchronized use of forks is achieved using the communication mechanism. Five forks are created whose identities are stored in the vector *forks*. These identities are required in determining which two forks a philosopher needs as each one of them is allocated a particular seat.

The deadlock situation where all philosophers dine together creating a shortage of forks is avoided. A philosopher will not join the table while her neighbour is still proceeding with her meal. This courtesy is observed by each philosopher by calling the procedure *join* before proceeding to pick up the two forks and the procedure *leave* upon finishing the meal. The procedure *join* may cause the caller to be blocked unless two forks in the vicinity of the seat required are free.

§8.2.6 Locality

The concept of locality is useful in a number of ways. For example, data can be replicated at various sites for resilience; processes can be downloaded to idle machines; files can be output to a particular printer on the network etc. There are other applications; efficiency and atomicity. The following program fragment has been used in a previous example:

```
let p = process
          with
            ep1 = entry(**int),
            ep2 = entry(string -> string)
        begin
        let ep1 = entry(**int x)
                    for i = lwb(x) to upb(x) do
                      for j = lwb(x(i)) to upb(x(i)) do x(i, j) := x(i, j) + i + j
        let ep2 = entry(string s -> string); s ++ "'n"
        while true do accept ep1 || ep2
        end
```

Here the assignments in the entry *ep1* are the subject of concern. Assuming it is called by a remote process, the evaluation of the expression on the right and the actual assignment in the inner most part

of the **for**-loop cause 6 messages (3 for loading the vector element and 3 for the assignment) to be exchanged. The total number of messages required for the entire operation is therefore 6 times the total number of elements in the vector. Moreover, each assignment is carried out as an atomic operation. A failure will invalidate previous assignments. Logically speaking, all the assignments should be carried out as a single transaction. The situation can be alleviated as follows:

```
let p = process
          with
            ep1 = entry(**int),
            ep2 = entry(string -> string)
        begin
        let ep1 = entry(**int x)
                  begin
                  let y := vector 1:: 1 of vector 1:: 1 of 0
                  if locality x = here
                    then y := x
                    else begin
                      y := transcopy x
                      for i = lwb(y) to upb(y) do y(i) := transcopy x(i)
                      end
                  for i = lwb(y) to upb(y) do
                    for j = lwb(y(i)) to upb(y(i)) do y(i, j) := y(i, j) + i + j
                  if locality x ~= here do assign y to x
                  end
        let ep2 = entry(string s -> string); s ++ "'n"
        while true do accept ep1 || ep2
        end
```

The body of *ep1* is slightly modified here. The locality of the parameter is discovered. If it originated from a remote address space, a copy of it is installed locally. Upon completion of the assignments, the copy is transferred back to where it originates. Except in abnormal circumstances, a copy of a reasonable size vector can be sent in a single message. Thus the entire operation requires 6 or more messages but the number is an order of magnitude less than the total number of elements in the vector. Furthermore, since the operations **transcopy** and **assign** are atomic, the assignments can be considered atomic as well.

§8.3 Performance Measurement

The measurements in this section are meant to give an indication on the performance of the RPC mechanism. The machines used in this experiment are a Sun 3/260 and a Sun 3/50 connected over a 10 MBits/sec ethernet. Both machines are running Unix 4.2 BSD. In addition, the 3/260 also runs as a network file server. Measurements

were taken under normal conditions but the machines were lightly loaded at the time the experiments were carried out.

Timing for 350 RPCs in a tight loop was made since the resolution of the clock 60Hz is considered low for our purpose. The timing for the tight-loop itself is negligible and is therefore ignored. Because of variance over the load of the machines, timing is performed repeatedly 25 times and the average is taken.

The test programs are a client and a server communicating over an entry which takes either no argument or one whose type for the purpose of the measurement is either **int** or ***int** representing immutable and mutable type respectively. In any case, the entry does not return any result and the body of the entry is null. Since the RPC mechanism is a very general communication mechanism, the test programs are run in the same address space, in different address spaces on the same machine (3/260) as well as on different machines (3/260 & 3/50). Times in the tables below are expressed in clock ticks. The figures in the brackets are the corresponding number of calls per second. Table 8.1 shows the figures for passing zero or one parameter in RPCs.

	3/260 <-> 3/260	3/260 <-> 3/50	*
none	119.0 (176.4)	202.6 (103.7)	13.1 (1605.5)
mutable	123.6 (169.9)	213.0 (98.6)	13.6 (1539.6)
immutable	119.6 (175.6)	207.3 (101.3)	13.4 (1567.2)

Table 8.1

	3/260 <-> 3/260	3/260 <-> 3/50	*
immutable	117.0 (179.6)	213.4 (98.4)	14.1 (1487.2)
mutable	134.5 (156.1)	204.4 (102.7)	14 (1500)

Table 8.2

Figures in Table 8.2 are the result of repeating the experiment. This time the entry takes no parameter and returns a result whose type is either mutable or immutable.

The set of figures for making RPCs in the same address space is presented in the third column of the tables above. It shows a factor of at least 10 gain in stark contrast with others. The performance difference here is not surprising since communication within the same address space is accomplished without involving the network.

A number of observations can be made. The performance in communication across the network is between 50% - 80% worse than communication within the same machine. This is not unreasonable and the variation may be due to uneven loads on the machines concerned and the network at the time the experiments were carried out. There is also little difference in communicating values of difference types. This is mainly because mutable objects are not copied. Passing a mutable object costs no more than passing an immutable object. In the worse case, the number of RPCs that can be made is well beyond 60 per second. The figures obtained from this experiment may be contrasted with that of Courier [Larus 83] where the performance (obtained between a Vax 11/780 and a Sun (M68000) connected by a 10 Mbit/sec ethernet) ranges from 4 calls per second for an RPC with 4000 bytes argument and a result to 60 calls per second for an RPC with 0 bytes argument and no result.

The cost of loading or updating a field of a remote reference is roughly equal to making an RPC with no parameter and one result or one parameter and void result respectively. It is interesting to compare local pointer dereference with remote pointer dereference. This is not covered here because requests for loading and updating remote referends are sent in batches[1] in order to reduce latency.

[1]Requests are gathered together according to their destinations and are sent by the system which looks into these periodically.

Chapter 9

Conclusions

The idea of persistence has been accepted as a sound language concept [PPRG 85b, PPRG 87b]. The provision of a persistent store together with a set of binding and typing mechanisms provides an attractive programming environment for the construction of a rich repertoire of applications. Within the PISA project, these include a functional database system [Kulkarni & Atkinson 86], a relational database system [Hepp & Norrie 85], a bibliographic database [Cooper 86], an interactive browser [Dearle & Brown 87], and a callable compiler. The next step is to provide concurrency and distribution in persistent programming languages so that the larger scale, distributed systems such as office information systems, computer integrated manufacture, etc. can be implemented.

§9.1 Summary of Work

It was observed that the concept of persistence can be generalized to include distribution; in the sense that, just as a programmer may program without knowing whether data is on backing stores or in volatile stores, equally he or she may not know the geographic location of the data to be manipulated. Consequently, the facilities of persistence can be provided in a distributed environment. The idea of a universal persistent store as a logical space of objects in individual persistent stores reachable over networks is proposed. Programmers on a distributed system see this logical space rather than distinct local stores. This thesis demonstrates that such an idea can be realized with the present day technologies showing that universal addressing does not require architectural changes. Moreover, autonomy is not lost so that existing addressing mechanisms are not given away in favour of a single universal addressing mechanism.

An ideal universal persistent store provides a conceptual framework in which persistent data anywhere in a network can be accessed and manipulated uniformly and with a guarantee of consistent semantics. To approach this ideal, a distributed store design was proposed and demonstrated based on the strategy that data are never copied between machines except where efficiency can be obtained without compromising semantics. This is significant in a number of aspects. For instance, the size and number of messages required to be exchanged in accessing non-local data are both sufficiently small to permit implementation over

existing local area networks. This strategy allows us to implement semaphores in virtual memory without hardware support in a distributed environment.

Data are never copied to satisfy remote demands. Instead, the demanding instructions are obeyed at the sites where the data are kept and results are sent back to produce the desired effect. This strategy is not new and was discussed in [Spector 82]. This approach was adopted because it has a neat semantics without requiring the maintenance of consistent copies over a network, and because the technology developed for RPCs can be used to realize it.

In some circumstances the overall number of messages could be high compared with that which could be achieved by an expert programmer who understood message costs. To allow such programmers to achieve efficiency the language concept of *locality* and the two constructs **transcopy** and **assign** are introduced. They allow the discovery of relative *locality* of data and then for copies of data to be made atomically between localities so the programmer may relocate data to achieve lower costs. A second motivation is of course to allow data to be replicated for resilience. With a good implementation of the basic mechanisms these facilities should be needed only rarely. It seems appropriate to allow the programmers to judge when this exceptional style of coding is needed.

Another motivation for the introduction of *locality* is to avoid embedding machine names in the codes for process communication (chapter 4). Machine names are interpreted by an external mechanism and therefore the language, which admits them, loses control over their use. *Locality* allows us to present to the programmers the underlying network in a consistent manner.

The process model is designed to take advantage of networks. The separation of process declarations and their executions gives rise to the possibility of starting a process on any *locality*. This can be achieved by downloading a process template using **transcopy** and the result is used in a subsequent **start**. The **transcopy** construct does a one level copy and is sensitive to the type of data. For example, in this case it does not copy the environment of the process template but merely its code. This is chiefly because environments are shared in DPS. Changes in store can be seen by those processes concerned and their localities are irrelevant. There are outstanding problems with the definition of **transcopy** and **assign**,

discussed later.

Coordinated accesses to shared resources in general can be achieved using semaphores. The choice of semaphores as the store-based (vs. message-based) synchronization mechanism may be controversial. It is chosen to avoid imposing a single locking regime. Instead a primitive is developed out of which appropriate synchronization schemes may be built. These are outlined in chapters 2 and 8. Chapter 4 introduced syntactic support to discourage mistakes in the use of semaphores where they are being used in simple ways. For example, a block may be associated with a list of semaphores so that they are all claimed before it is entered, their claiming cannot cause deadlock, and however the block is left (by end of block, exceptions or machine events) none of them are inadvertently retained.

Remote procedure calls provide synchronous transfer of computation. The calling process is halted immediately, and some time later the called process may run the called procedure. If it returns then the original process is resumed. This is a reasonable compromise between feasible implementation and achieving identical semantics for remote and local procedure calls. For instance, the semantics of parameter passing was made consistent for local and remote procedures. The difference is that the called process may never accept the call (chapter 8). Synchronous semantics has a counter-productive effect on the degree of concurrency achievable. Ada addressed this problem by introducing extended rendezvous. In DPS, no new concept is required since dynamic processes can be used to achieve an equivalent behaviour; examples can be found in chapter 8.

The remote procedure call mechanism is part of the abstract machine. Its efficient implementation is a most important factor in deciding whether or not the language is acceptable. Even without microcode support, the statistics gathered in chapter 8 suggests that the performance is acceptable and at least as good as other RPC mechanisms [Larus 83]. Of course, as mutable parameters are not copied in RPCs the cost of the call is reduced.

Another factor which contributes towards the efficiency of the RPC mechanism is that typechecking is factored out by signature matching prior to any communication. Signature matching is performed dynamically and once only so that typechecking every message sent and received is avoided. Static signature matching is unacceptable for two reasons. One is to avoid using a distributed database to keep type

information. This would require global coordination, contradicting the aim of utilizing only local evolving information about behaviour of neighbouring processors. The other is to allow binding to be delayed so that software components may evolve independently [Atkinson et al. 88]. Moreover, there is an element of flexibility in the signature matching algorithm so that changes in one software component do not invalidate the type checks where they do not intersect the signature.

§9.2 Critique of Achievements

As powerful workstations and high speed networks are becoming more available, it is clear that paradigms, tools, programming languages or programming environments for distributed computations will be in demand. Aspects of distribution have already been subjects of research in many different areas. This thesis presents a demonstration that distribution can be presented to the programmers in a manner consistent with persistence without requiring them to master new concepts. Larger scale experiments based on DPS are now required to assess whether such an approach is practical for typical applications and adequate to achieve performance when large scale systems are built.

The principle objective of this project is to demonstrate that it is both desirable and feasible to realize a universal persistent store. An implementation of such an idea exists which took about 8 man-months to construct with the kind of technologies already on hand. The language, DPS, the programmer view of that implementation in which the locality of data is hidden (but in which it may be discovered when necessary), was used in a number of demonstrations. Two of these -- concurrent spinners and a distributed free-hand drawing package (§8.2.3) -- were demonstrated in the Alvey Conference in July, 1987. The two programs were written by a colleague, Paul Philbrow who had no experience in the use of a distributed language. The coding effort in each case was less than a man-day from an initial discussion to the finished program. The distributed free-hand drawing package, in particular, confirms the view that distribution need not be a subject of concern and the programmer is able to concentrate on developing an algorithm in a simple and easy to understand framework.

A small number of language concepts and constructs were introduced in order to take advantage of distribution in a manner that is coherent with the programming paradigms advocated in PS-algol. The notion of a universal store is a novelty to most programmers but, they quickly learn to exploit its advantages in distributed programming since

algorithm definition can be insensitive to the locality of data. The experiments that have been possible since the implementation was complete have had programmers who were part of the encompassing PISA project. This is obviously not an unbiased group and they probably understand the model more readily than an average programmer. A student project has just started which will provide evidence from less experienced programmers. An important consequence of allowing the same constructs to be applicable on both remote and local data (and therefore consistent semantics) is that the coding effort required and the complexity of these algorithms are unchanged. However, it is possible to access the additional robustness of a distributed system by writing more complex algorithms utilizing explicit control of locality.

The concepts of dynamic lightweight processes and synchronous process communication are useful and permit easy programming. The concurrent sorting network (§8.2.2), for instance, was programmed in less than an hour once the problem was well-understood. The coding itself is small - about 40 lines of DPS. The algorithmic construction of a network of sorting processes constructs the geometric formation of the network of the appropriate scale. This depended on the ability to spawn processes when needed and the first class nature of process handles allowing freedom to arrange dynamically interprocess connections. Some examples in chapter 8 serve to illustrate the expressive power of the language capitalizing on the synergy of the two facilities.

The language design might benefit from a formal definition of its semantics. Most of the ideas are simple and easy to understand and are believed to be well defined in our informal terms. But their shortcomings and any pathological interactions between them might be discovered by more formal definition. Although large scale experiments may provide valuable feedback in the long term (which is not necessarily going to be exposed by formal treatment), a formal theory is a more versatile, economic and efficient means of obtaining potentially different and strategically important verification of the design.

The concept of *locality* and the related constructs: **transcopy** and **assign** serve their purpose well for the present implementation. However, they will certainly be improved by further research and experiment. Jack Campin [Campin 88] pointed out some of the debatable areas of the present model. For example, since loca can be defined as a collection of nodes (see chapter 4), the base type **loca** is extraneous. But care is required to avoid allowing a node to be within another node

leading to a situation which has no counterpart in the real world. The main motivation for providing the **transcopy** operator is for efficient and atomic transfer of data over the network and to avoid the danger of pulling more data than that of which the programmer is aware. Hence **transcopy** does a one-level copy and falls short of a proper treatment of circular objects. (We noted that this was the compromise reached in FAD [Danforth et al. 87].)

The investigation of supporting persistence in a network environment has lead to the idea of a universal persistent store. This thesis has shown that such a store is both feasible to realize with the present day technologies and that distributed programming with a distributed language based on such a store need not be more difficult than the conventional style of programming. Moreover, with the concept of *locality* the advantages offered by the underlying network can be obtained without jeopardizing the integrity of the framework underlying the language.

§9.3 Future Work

The following list illustrates some aspects of the system requiring further investigation:

- *Iterator* Although it was stressed that it is only necessary to pass the root to an object in communication, there is no provision of an iterator for the purpose of discovering what that root provides. There is a parallel development on namespaces in Napier [Atkinson 86]. A polymorphic iterator construct is provided there for traversing hierarchical namespaces. It is envisaged that objects in store will be organized into such namespaces, and when that is done it will be possible to investigate whether this meets the need.

  Another class of iterations is used by diagnostic and statistical tools on stores. So far these have not been considered, as the difficulty is that they would break protection and information hiding. However, such tools will have to be built for any production system.

- *Distributed Transactions* This issue is not addressed in the present implementation. The author did not believe the requirements were sufficiently identified to permit a solution to be built into the system. Krablin [Krablin 85b] suggested that concurrent processes and the **lock** construct

are sufficient to program transactions meeting the requirements. This should be investigated by practical implementations using DPS. It is possible that this would introduce cumbersome code and be onerous for each programmer. The problem seems to be with distributed commit. At the present stage, this can be done by the programmers using the reliable communication, synchronization and stable storage already provided. The research may be continued by building systems using these constructs and assessing the difficulty presented to programmers. If it then appears to be too complex to require implementation out of primitives, syntactic support to commonly coded cases could be investigated. This might lead to built-in optimizations.

- *Locality* The concept of locality is important in our system. It serves the purpose of allowing coding of transactions, archiving, merging of persistent stores to form a new universal persistent store etc. The present version falls short of arranging for new locality values to be introduced without stepping outside the language. It serves to be used to develop an understanding of the way programmers would use such a concept. When its benefits and deficiencies have been discovered a better notation or semantics may be developed.

- *Canonical Representation of Values* This is required in the context of **transcopy**. It was not a problem in the current implementation as a cluster of Sun-3s were used. To overcome the heterogeneous problem, either a set of canonical representations of values of the base types is required so that the receiver end of a communication knows what to expect or automatic data translation. The latter (which is more efficient) can be invoked as a result of exchanging the appropriate messages. Fred Brown is working on canonical representation of values for PS-algol data types [Brown 87]. There remains the problem of representation of abstract data types. In [Herlihy & Liskov 82] Herlihy and Liskov described one possible approach.

- *Transcopy* This construct is provided for the sake of transferring a copy of some piece of data from one locality to another in an atomic fashion. As explained above, it does a

one level copy and falls short of proper handling of circular objects. Thus if a complete copy of a graph-like object is required, the programmer will have to traverse the entire graph and repeatedly use **transcopy** on each node encountered. The point is that he or she will have to retain circularity by keeping track of all nodes copied so far. It may be appropriate to provide another construct to accomplish this. Even with structures that the programmer considers as 'one-level' there is a difficulty, as the implementor may see many levels. A particularly difficult case is the procedure or the ADT because of shared environments. Furthermore we do not yet know how to give the programmer the power to construct traversals over these without compromising semantics, scoping and protection.

- *Store Management* Finally, there is an intriguing engineering problem. The concurrency control mechanism is store-based so that all processes sharing the same resources can cooperate among themselves by employing conventions based on mutexes. The realization of one level store in the PS-algol system is by copying lazily the necessary data between the heap and the secondary store. Consequently, there is a momentary disparity between the state of data in the secondary store and the heap. It is a deliberate dichotomy so that there is always a copy of data in the stable store. But this also leads to situations where some collection of processes observing the same convention is synchronized whilst others are not, depending on whether they are accessing the same version of data. Some variant of the Monads approach on this matter may be useful here.

The motivation behind the language concept of persistence is to factor out difficult tasks in the manipulation of data in different storage media so that coding effort can be reduced. By hiding the locality of machines in a network, distribution can be subsumed by persistence. As the investigation of this project has shown, it is necessary to relax the locality transparency requirement in persistence so that programmers are allowed to arrange the appropriate strategy for their applications. There is a balance to be struck in providing the sort of facilities described in this thesis in a distributed programming language. It is expected that more research is required to confirm or revise this balance now that a working

system is available.

References[1]

[ANSI 83] America National Standards Institute, Inc. *The Programming Language Ada Reference Manual* ANSI/MIL-STD-1815A-1983 LNCS 155

[Abramson & Keedy 85] Abramson, D. A. and Keedy J. L. *Implementing a Large Virtual Memory in a Distributed Computing System* Proc. of the 18th Ann. Hawaii Int. Conf. on System Sciences

[Ahuja 83] Ahuja, S. *A High-speed Interconnect for Multiple Computers* IEEE Selected Areas in Communication

[Atkinson 86] Atkinson, M. P. *Private Communication*

[Atkinson et al. 81] Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. *PS-algol: An Algol with a Persistent Heap* ACM SIGPLAN Notices 17(7):24-31

[Atkinson et al. 83] Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, W. P. and Morrison, R. *An Approach to Persistent Programming* The Computer Journal 26(4):360-365

[Atkinson et al. 87] Atkinson, M. P., Morrison, R. and Pratten, G. D. *Persistent Information Architectures* PPRR-36 Universities of Glasgow and St. Andrews

[Atkinson et al. 88] Atkinson, M. P., Buneman, O. P. and Morrison, R. *Binding and Type Checking in Database Programming Languages* Computer Journal 31(2):99-109

[Atkinson & Morrison 85a] Atkinson, M. P. and Morrison, R. *Procedures as Persistent Data Objects* ACM TOPLAS 7(4):539-559

[Atkinson & Morrison 85b] Atkinson, M. P. and Morrison, R. *Types, Bindings and Parameters in a Persistent Environment* in Data Types and Persistence (Eds: Atkinson, Buneman and Morrison) Springer-Verlag Berlin

[Ben-Ari 82] Ben-Ari, M. *Principles of Concurrent Programming* Prentice/Hall International

[Birrell & Nelson 83] Birrell, A. D. and Nelson, B. J. *Implementing Remote Procedure*

[1] In the following, LNCS stands for Lecture Notes in Computer Science published by Springer-Verlag, Berlin; PPRR stands for Persistent Programming Research Report available from the Department of Computing Science, University of Glasgow or Department of Computational Science, University of St. Andrews.

Calls CSL-83-7 Xerox PARC

[Bishop 77] Bishop, P. B. *Computer Systems with a Very Large Address Space and Garbage Collections* TR-178 MIT

[Brinch Hansen 73] Brinch Hansen, P. *Operating System Principles* Prentice-Hall

[Brinch Hansen 75] Brinch Hansen, P. *The Programming Language Concurrent Pascal* IEEE Trans. on Software Engineering SE-1, 2

[Brinch Hansen 78] Brinch Hansen, P. *Distributed Processes: A Concurrent Programming Concept* CACM 21(11):934-941

[Brown 87] Brown, A. L. *A Distributed Stable Store* Proceeding Workshop on Persistent Object Systems: their Design, Implementation and Use PPRR-44 Universities of Glasgow and St. Andrews

[Brown & Cockshott 85] Brown, A. L. and Cockshott, W. P. *CPOMS* PPRR-13 Universities of Glasgow and St. Andrews

[Brownbridge et al. 82] Brownbridge, D., Marshall, L. and Randell, B. *The Newcastle Connection or Unixes of the World Unite* Software Practice and Experience 12

[Campbell & Habermann 74] Campbell, R. H. and Habermann, A. N. *The Specification of Process Synchronization by Path Expressions* Operating Systems LNCS 16

[Campin 88] Campin, J. *Leibnizian Cyberspace* Internal Document, The PISA project, Department of Computing Science, University of Glasgow

[Cardelli 84] Cardelli, L. *An Implementation Model of Rendezvous Communication* Seminar on Concurrency LNCS 197

[Cardelli 85] Cardelli, L. *Amber* Technical Memo, AT&T

[Carreiro & Gelernter 86] Carreiro, N. and Gelernter, D. *The S/Net's Linda Kernel* ACM Trans. on Comp. Systems 4(2)

[Carrick et al. 87] Carrick, R., Cole, A. J. and Morrison, R. *An Introduction to PS-algol Programming (third edition)* PPRR-31 Universities of St. Andrews and Glasgow

[Cheriton & Zwaenepoel 83] Cheriton, D. R. and Zwaenepoel, W. *The Distributed V Kernel and its Performance for Diskless Workstations* ACM Proc. on the 9th Symp.

on Operating System Principles

[Cockshott 83] Cockshott, W. P. *Orthogonal Persistence* PhD Thesis, Department of Computer Science, University of Edinburgh

[Cockshott 85] Cockshott, W. P. *The Persistent Store Machine* PPRR-18 Universities of Glasgow and St. Andrews

[Cooper 86] Cooper, R. L.*Using a Persistent Environment to Maintain a Bibliographic Database* PPRR-24 Universities of Glasgow and St. Andrews

[Cooper 87] Cooper, R. L. *Applications Programming in PS-algol* PPRR-25 Universities of Glasgow and St Andrews

[Daley & Dennis 68] Daley, R. C. and Dennis, J. B. *Virtual Memory, Processes, and Sharing in Multics* CACM 11(5)

[Danforth et al. 87] Danforth, S., Khoshafian, S., Valduriez, P. *FAD -- A Database Programming Language* submitted to ACM TODS

[Davies & Morrison 81] Davies, A. J. T. and Morrison, R. *Recursive Descent Compiling* Ellis Horwood

[Dearle & Brown 87] Dearle, A. and Brown, A. L. *Safe Browsing in a Strongly Typed Persistent Environment* PPRR-33 Universities of Glasgow and St. Andrews

[Dijkstra 68] Dijkstra, E. W. *Cooperating Sequential Processes* in Programming Languages (Ed: Genuys) Academic Press, New York

[Dijkstra 72] Dijkstra, E. W. *Hierachical Ordering of Sequential Processes* In Operating System Techniques (ed: Hoare & Perrot) Academic Press

[Dijkstra 75] Dijkstra, E. W. *Guarded Commands, Non-determinancy and Formal Derivation of Programs* CACM 18(8)

[Doeppner 86] Doeppner, T. *Towards a Workstation Operating System* Proc. of the 19th Hawaii Int. Conf. on System Sciences

[Fitzgerald & Rashid 86] Fitzgerald, R. and Rashid, R. F. *The Integration of Virtual Memory Management and Interprocess Communication in Accent* ACM Trans. on Computer Systems 4(2)

[French et al. 78] French, R. E., Collins, R. W. and Loen, L. W. *System/38 Machine Storage Management* IBM System/38 Technical Developments, IBM General System Division

[Gallagher 85] Gallagher, R. T. *Thomson Pursues 70-Bit Addressing Scheme* Electronics November 4

[Gray et al. 87] Gray, P. M. D., Moffat, D. S. and Paton, N. *A Prolog Interface to the FDM with Modular Commitment* Internal Report Dept of Comp. Sci. University of Aberdeen

[Hepp 83] Hepp, P. E. *A DBS Architecture Supporting Coexisting Query Languages and Data Models* PhD Thesis, Department of Computer Science, University of Edinburgh

[Hepp & Norrie 85] Hepp, P. and Norrie, M. *Raquel User Manual* CSR-188-85 Dept of Comp. Sci. University of Edinburgh

[Herlihy & Liskov 82] Herlihy, M. and Liskov, B. *A Value Transmission Method for Abstract Data Types* ACM TOPLAS 4(4)

[Hoare 72] Hoare, C. A. R. *Towards a Theory of Parallel Programming* Operating System Techniques (ed: Hoare & Perrot) Academic Press

[Hoare 74] Hoare, C. A. R. *Monitors: An Operating System Structuring Concept* CACM vol. 17(10):549-557

[Hoare 78] Hoare, C. A. R. *Communicating Sequential Processes* CACM 21(8):306-317

[Hoare 81] Hoare, C. A. R. *The Emperor's Old Clothes* CACM 24(2):75-83

[Holler 81] Holler, E. *Multiple Copy Update* Distributed Systems - Architecture & Implementation An Advanced Course LNCS 105

[Hughes 84] Hughes, J. *A Distributed Garbage Collection Algorithm* Functional Programming Languages and Computer Architecture LNCS 201

[Hurst 87] Hurst, A. J. *A Context Sensitive Addressing Model* PPRR-27 Universities of Glasgow and St Andrews

[Kulkarni & Atkinson 86] Kulkarni, K. G. and Atkinson, M. P. *EFDM: Extended Functional Data Model* The Computer Journal 29(1):38-45

[Kung & Robinson 81] Kung, H. T. and Robinson, J. T. *On Optimistic Methods for*

Concurrency Control ACM TODS 6(2)

[Krablin 85a] Krablin, G. L. *Experimental Concurrency in PS-algol* Private Communication

[Krablin 85b] Krablin, G. L. *Building Flexible Multilevel Transactions in a Distributed Persistent Environment* PPRR-16 Universities of Glasgow and St Andrews

[Kulkarni 83] Kulkarni, K. G. *EFDM - User Manual* PPRR-7 Universities of Glasgow and St. Andrews

[Lampson & Redell 80] Lampson, B. W. and Redell, D. D. *Experience with Processes and Monitors in Mesa* CACM vol.23(2)

[Larus 83] Larus, J. R. *On the Performance of Courier Remote Procedure Call under 4.1c bsd* UCB/CSD 82/123

[Leach et al. 83] Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L. and Stumpf, B. L. *The Architecture of an Integrated Local Network* IEEE Journal on Selected Areas in Communications SAC1(5)

[Liskov 84] Liskov, B. *Overview of the Argus Language and System* Programming Methodology Group Memo 40 MIT

[Liskov 85a] Liskov, B. et al. *Limitations of Synchronization Communication with Static Process Structure in Languages for Distributed Computing* CMU-CS-85-168

[Liskov 85b] Liskov, B. *The Argus Language and System* LNCS 190

[Lister 77] Lister, A. *The Problem of Nested Monitor Calls* Operating System Review 11(3)

[Morrison et al. 87] Morrison, R., Barter, C. J., Brown, A. L., Carrick, R., Connor, R., Dearle, A., Hurst, A. J. and Livesey, M. J. *Polymorphic Persistent Processes* PPRR-39 Universities of Glasgow and St. Andrews

[May 83] May, D. *Occam* ACM SIGPLAN Notices 18(4)

[Needham & Herbert 82] Needham, R. M. and Herbert, A. J. *The Cambridge Distributed Computing System* Addison-Wesley

[Norrie 85] Norrie, M. *The Edinburgh Node of the Proteus Distributed Database System*

CSR-191-85 Dept. of Computer Science University of Edinburgh

[Organick 72] Organick, E. I. *The Multics System: An Examination of its Structure* MIT Press

[Philbrow & Atkinson 86] Philbrow, P. and Atkinson, M. P. *Exception Handling in a Persistent Programming Language* PPRR-26 Universities of Glasgow and St Andrews; to appear also in the Computer Journal in 1989

[PPRG 85a] -*PS-algol Abstract Machine Manual* PPRR-11 Universities of Glasgow and St Andrews

[PPRG 85b] - *Persistence and Data Types: Papers for the Appin Workshop* PPRR-16 Universities of Glasgow and St. Andrews

[PPRG 87a] - *PS-algol User Manual - fourth edition* PPRR-12 Universities of Glasgow and St. Andrews

[PPRG 87b] - *Proceeding Workshop on Persistent Object Systems: their Design, Implementation and Use* PPRR-44 Universities of Glasgow and St. Andrews

[Popek et al. 81] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G. and Thiel, G. *LOCUS: A Network Transparent, High Reliability Distributed System* ACM Proc. on the 8th Symp. on Operating System Principles

[Postel 80] Postel, J. *User Datagram Protocol* Request For Comment 768

[Rashid & Robertson 81] Rashid, R. F. and Robertson, G. G. *Accent: A Communication Oriented Network Operating System Kernel* ACM Proc. on the 8th Symp. on Operating System Principles

[Rovner et al. 85] Rovner, P., Levin, R. and Wick, J. *On Extending Modula-2 For Building Large, Integrated Systems* DEC SRC Research Report

[Schmidt 77] Schmidt, J. W. *Some High Level Language Constructs for Data of Type Relation* ACM Trans. on Database Systems 2(3)

[Schroeder et al. 85] Schroeder, M. D., Gifford, D. K. and Needham, R. M. *The Caching File System for a Programmer's Workstation* Research Report, DEC SRC Palo Alto

[Sheeran 85] Sheeran, M. *Designing Regular Array Architectures Using Higher Order*

Functions Proc. Int. Conf. on Functional Programming Languages & Computer Architectures LNCS 201

[Sparks 88] Sparks, D. J. *Interface between Flagship and its Object Server* Research Report STC Technology Ltd, Newcastle-under-Lyme, UK

[Spector 82] Spector, A. Z. *Performing Remote Operation Efficiently on a Local Computer Network* CACM 25(4)

[Walsh et al. 85] Walsh, D., Lyon, R. and Sager, G. *Overview of the Sun Network File System* Proc. on Usenix Winter Conference, Dallas

[Warboys 85] Warboys, B. C. *VME Nodal Architecture: A Model for the Realization of a Distributed System Concept* ICL Technical Journal 4(3)

[Watt et al. 87] Watt, D. A., Wichmann, B. A., Findlay, W. *ADA Language and Methodology* Prentice/Hall International

[van Wijngaarden et al. 76] van Wijngaarden, A., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T., Fisker, R. G. *Revised Report on the Algorithmic Language Algol68* Springer-Verlag