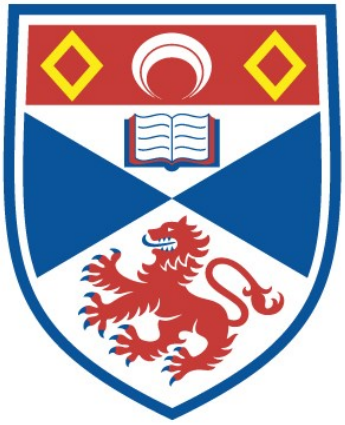
THE APPLICATION OF MESSAGE PASSING TO CONCURRENT PROGRAMMING

David M. Harland

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



1981

UNIVERSITY OF ST. ANDREWS

Thesis Copyright Declaration Form.

A UNRESTRICTED

"In submitting this thesis to the University of St. Andrews I understand that I am giving permission for it to be made available for public use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker."

B RESTRICTED

"In submitting this thesis to the University of St. Andrews I wish access to it to be subject to the following conditions:

for a period of years [maximum 5] from the date of submission the thesis shall be

a) withheld from public use.

b) made available for public use only with consent of the head or chairman of the department in which the work was carried out.

I understand, however, that the title and abstract of the thesis will be published during this period of restricted access; and that after the expiry of this period the thesis will be made available for public use in accordance with the regulations of the University Library for the time being in force, subject to any copyright in the work not being affected thereby, and a copy of the work may be made and supplied to any bona fide library or research worker."

Declaration

I wish to exercise option A [i.e. A, Ba or Bb] of the above options.

Signature

Date 22 July 1982

Abstract

The development of concurrency in computer systems will be critically reviewed and an alternative strategy proposed. This is a programming language designed along semantic principles, and it is based upon the treatment of concurrent processes as values within that language's universe of discourse. An asynchronous polymorphic message system is provided to enable co-existent processes to communicate freely. This is presented as a fundamental language construct, and it is completely general purpose, as all values, however complex, can be passed as messages.

Various operations are also built into the language so as to permit processes to discover and examine one another. These permit the development of robust systems, where localised failures can be detected, and action can be taken to recover.

The orthogonality of the design is discussed and its implementation in terms of an incremental compiler and abstract machine interpreter is outlined in some detail.

This thesis hopes to demonstrate that message-oriented communication in a highly parallel system of processes is not only a natural form of expression, but is eminently practical, so long as the entities performing the communication are values in the language.

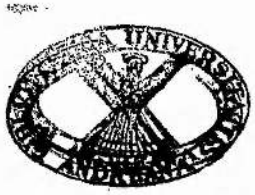
The Application Of Message Passing To Concurrent Programming

David M Harland

A thesis submitted for the degree of Doctor Of Philosophy.

Department Of Computational Science
University Of St.Andrews
Scotland

1981



Contents.

Declaration.

I declare that this thesis has been composed by myself and that the work that it describes has been done by myself. This work has not been submitted in any previous application for a higher degree. The research has been performed since my admission as a research student under Ordinance General No.12 on 1st. October 1978 for the degree of Doctor Of Philosophy.

David M Harland

I hereby declare that the conditions of the Ordinance and Regulations for the degree of Doctor Of Philosophy (Ph.D) at the University Of St.Andrews have been fulfilled by the candidate, David M Harland.

Professor A J Cole

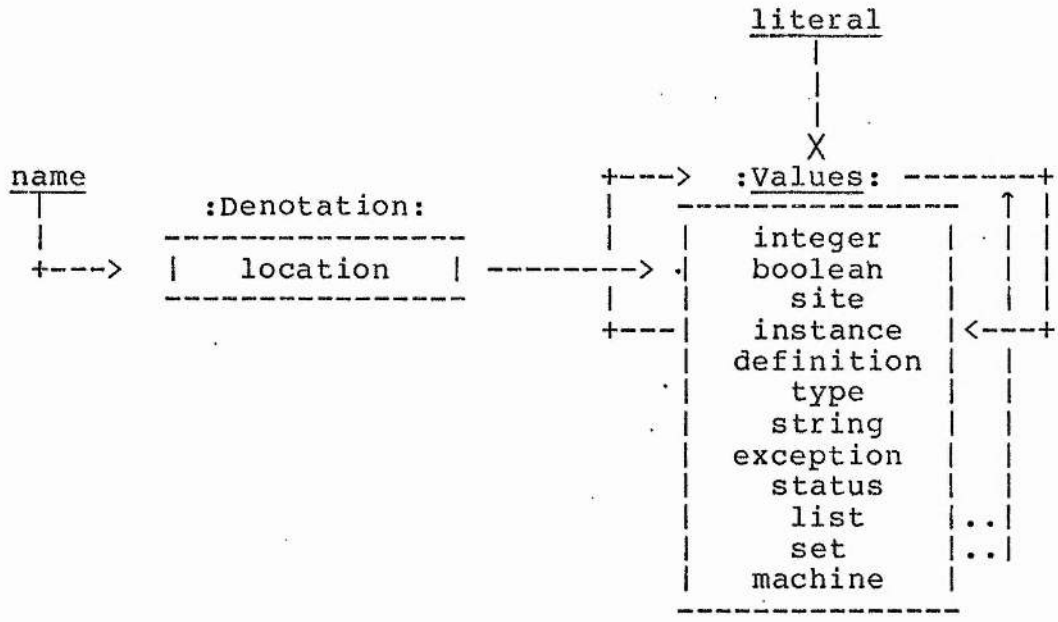
Acknowledgements.

It is a pleasure to acknowledge the assistance of Professor A.J. Cole, my supervisor during the period of this research, who, despite being bombarded with preliminary drafts and notes, was always the epitome of encouragement and improvement.

I must thank Ron Morrison and Mike Livesey, for suggesting fruitful areas for investigation, particularly at the start of this project. Mike Weatherill is also to be thanked for further discussions towards the end, and for initial readings of this thesis. I cannot sufficiently thank Hamish Gunn for providing and maintaining the language in which Protocol is written, which made the implementation considerably easier than it might otherwise have been.

Martyn Szyplewski, whose gift for finding 'bugs' and irregularities in software is also to be thanked for acting as 'user', as are my fellow research students, who often wondered what I was doing!

Finally it is a pleasure to acknowledge the financial support of the Science Research Council [UK], without which this work would have not been possible, and to thank my wife for making it all worthwhile.



Protocol : An Ontology Diagram.

1.Ø Introduction.

In this thesis we will outline the history of attempts to incorporate parallelism into programming systems; its origins in operating systems, and its evolution within programming languages.

We shall critically examine the most widely adopted techniques pointing out their deficiences, and then propose an alternative approach which seems both more natural and more powerful.

In the second section we will discuss the details of designing a programming language which captures the flavour of this new approach. Then in the third section, we will present one possible application, in terms of an on-line plug-in hardware system for laboratory experiments.

Finally, various details of an implementation of this language for the Digital Equipment PDP-11 computer, under the UNIX operating system, will be discussed.

2.Ø A Historical Perspective.

In this section we shall briefly outline the development of concurrency in computing systems, and point out the major problems encountered, and then we shall propose another, much simpler, and it is hoped better, approach to concurrency.

2.1 Concurrency In Operating Systems.

In the beginning computer systems were monolithic in structure and strictly sequential in execution. Thus when the processor initiated an external operation it simply monitored some register or other, patiently idling while it waited for its completion. This greatly under-used the capabilities of the most expensive component, the central processor, particularly when it was communicating with slow peripherals or its human operators. Not surprisingly, it wasn't long before this situation was remedied by the introduction of interrupt-driven hardware. Here the central processor was able to continue operations after requesting an external activity because it would be 'interrupted' when the task was completed. To take advantage of such parallel architectures operating systems were structured so that they could run several independent user 'processes' at the same time, interleaving them in space and time in response to peripheral performance. Later 'time-shared' systems, such as the MAC project at MIT, enhanced this illusion of parallelism by directly managing a number of interactive terminal users. These efforts were based on the dictum that

the central processor should be kept busy at all times whilst on-line users should be able to work as if they had it dedicated to their needs alone.

These developments in hardware, exploited by advances in operating systems, naturally revealed many 'new' problems. Firstly, the very nature of the interrupt mechanism produced the problem of instantaneously saving the current activity whenever an interrupt arose. This also required that system software be reentrant, so that there could be no interference between the various user processes during such suspensions, and so that the priority layering of the various interrupt levels could be undertaken directly in the hardware, with automatic 'stacking up' of pending and partly executed lower priority services. Finally the nondeterministic asynchronism of the interrupt mechanism produced severe scheduling problems, both in terms of servicing peripherals promptly and enhancing overall productivity. There are three main requirements, often mutually conflicting, which are bound up in the scheduling problem, efficient use 1) of processor time, 2) of central and backing store and 3) of user service (usually measured by response time). Each presents its own difficulties.

It would be fair to say, however, that by the early 1970's satisfactory techniques had been developed to cater for all these requirements, and that there were several multi-user time-shared systems in widespread use. The UNIX system [R8] is perhaps particularly successful, and it is noteworthy because it was one of the last such systems to be written in

a programming language which, although high-level, did not have specific constructs for creating and managing interactions between concurrent processes. In what we might call the 'classical' programming languages, these complex interactions were specified by ad hoc low-level mechanisms. It was at about this time that a new class of programming languages emerged.

2.2 Concurrency In Programming Languages.

Since Dijkstra's classic paper [D11] exposing the pitfalls of parallelism there have been numerous programming techniques put forward. Most primitive was the semaphore, with its P/V-operations. Although ultimately the most powerful they were found to be too difficult to use in general. Whereas semaphores were originally implemented by the user, as integer variables, and were manipulated directly by addition and subtraction, Algol-68 [W13], introduced them directly into the language itself. They formed values of type sema and were manipulated by special operators up and down.

This subtle change of status, from a user application to part of the language, was highly significant. By exploiting the concept of a data type the implementation was able to protect the user by checking that he did not corrupt the semaphores with illegal operations (such as multiplying them for example).

A 'critical section' was a segment of program which manipulated a data structure 'shared' with another part of a

concurrent program. By placing semaphore operations around such critical sections of a program it was possible to synchronise the requests of the various parts of a concurrent program for access to common data. However, this was extremely tricky in practice, and potentially disastrous in any non-trivial application.

This insecurity arose principally because the critical regions were apparent only to the user and the implementation was unable to protect him in any way. For example, it could not verify that he associated the appropriate semaphore with a given critical section; it could not guarantee that he performed the up/down operations in the correct sequence and indeed it could not even verify that he had remembered to protect the critical section at all! It was in this light that Brinch Hansen [B20,B21] introduced the attribute shared to declarations of data, together with a control structure called the region to augment the semaphore variables (which he called conditions). In this way the critical text was explicitly delimited by the region block, and access to it was regulated by a semaphore, whilst shared items could only be accessed within a region block. The advantage of all this to the user was that he could be protected from his own follies whilst being relieved of much of the burden of the programming. However, even here, he could still manipulate the semaphores incorrectly and there was no way for the implementation to know which semaphore guarded which region/data combination. There was still considerable room

for error.

It was at this point that Hoare [H13] and Brinch Hansen [B21,B22] independently proposed the 'monitor' construct. This gathered together all the above individual ideas into one single easy-to-use facility for programming access to data shared within a concurrent program. The monitor drew on the concept of a class in Simula [D1], by isolating the declarations of the data structures to be shared, together with all their critical sections, from the main program text, by enclosing them in a monitor-block. This could then be given a name. Access to such data was strictly controlled by the monitor itself. Apart from the purely lexical advantage of having all the relevant code and declarations gathered together into one place, the implementation was now in a position to protect the user more thoroughly. For example, the 'controlling semaphore' which guards the monitor was implicit in the monitor block structure - it was invisible to the user and so could not be corrupted, furthermore it operated entirely automatically, leaving little scope for error. The monitor was a tool since the user needed only to declare it for the tricky details of the implementation to be provided for him. It was a logical development of the concept of sharing data between concurrent processes.

First in the new genre of languages were Wirth's Modula [W16] and Brinch Hansen's Concurrent Pascal [B22], both exploiting the monitor concept. In addition, to capture the

flavour of concurrency, they introduced a block-like construct to delimit the text of the various processes in the concurrent program. This, they argued, was more modular than the simple 'concurrent statement' of Algol-68. Many languages have been implemented using the process/monitor mechanism, consider Simone [K2], CCNPASCAL [N1], Pascal-Plus [W6,W7] and Concurrent SP/k [H16] for example, and in some cases monitors have been provided as applications packages to much lower level languages, such as in the case of BCPL [L12].

2.3 Review.

It would be fair to say that considerable success has been achieved with the monitor-based view of concurrency, but it conceals several critical issues, some fundamental to its entire philosophy. The monitor is about sharing. It guarantees secure access to its data structures by permitting at most one process to be active within it at any given time. However, as its designers recognised, there is a need to be able to 'delay' processes within a monitor. Suspended processes need to be queued up, temporarily releasing the monitor so that some other process can enter it. Once this process departs, conditions will hopefully be such that one of the suspended processes can be reactivated. This procedure is essential if request/release operations are to avoid deadlock. The significance of this delaying tactic is that the monitor enforces sequentiality in order to implement concurrency! The enforcement of sequential access has far reaching implications for scheduling

strategies (see, for example, the discussion of Legally [L1]). When one considers the consequences of nested monitor calls [see H2,L1Ø,L11,L12 and P1 for example], it becomes obvious that monitors are not the ideal solution that they appear to be. Thus, for example, should nested monitor calls be permitted as being essential [B2Ø] for hierarchical systems, or forbidden [K2] as entirely unnecessary?

3.Ø Another Look At Concurrency.

During these developments, from semaphore and regions in monolithic programs to monitors and processes in modular programs, the unquestioned assumption adopted by all, was that processes actually need to interact via shared data structures, and so efforts have always been directed towards providing more secure mechanisms for such interactions.

The monitor approach is essentially oriented towards the development of new control structures, enhancements of block-structure itself, with the emphasis on encapsulation via the Simula notion of block-retention, treating concurrency as an issue in 'flow of control'. In this thesis we hope to show that concurrency is not a matter of sequentiality via control structures, but of communication.

The real problem is therefore much more fundamental than simply providing secure access to shared data structures, it lies in our entire approach to the concept of interaction. Previous efforts have been geared towards interaction via side-effects (by way of an updatable store) but interaction,

in abstract terms, is the communication of messages, and in terms of computing such messages are values. The implementation of concurrency by imposing sequentiality, with all its scheduling effects and bottlenecks, seems to be counterproductive, and despite its data sharing it conceals the assumption of a common store in the implementation (to be controlled by the monitors) - a fact that will probably not be the norm in the distributed multi-computer systems of tomorrow. A more natural approach would seem to need the specification of individual behaviours (the definitions of processes) together with a generalised, polymorphic message passing system which is completely asynchronous (so that parallelism can be exploited without imposing sequentiality). Furthermore, such a system must not be tied to current centralised architectures, or else it will offer no alternative at all to the unwieldy systems currently being 'extended' to distributed and embedded networks.

Since the basic problem is embedded in our approach to programming we shall now examine the language design process itself, as opposed to specific extensions to its present form. We will endeavour to reveal the principles which lead to a 'well designed' language, in order that we can then exploit them whilst designing a message-oriented programming language.

3.1 The General Problem.

When concurrency is incorporated directly into a programming language several quite distinct problems arise. All of the

languages onto which concurrent 'features' have been grafted suffer from these problems, and each shows its own scars, as its designers tried to overcome them.

Here we shall begin by introducing the most obvious areas in which the traditional freedoms conflict with the discipline demanded by the inclusion of concurrency into a language. Then we shall attempt to discover the root of the ailment, and go on to suggest a less painful antidote. In later sections we shall develop this new approach, and show how it offers a more natural and higher-level framework in which to programme concurrent activities. But for now we discuss the problems.

As we have seen most of the languages so far supporting 'concurrent processes' permit them to interact via shared data structures. Direct sharing is therefore the most obvious problem in such languages. This is so for several reasons. Firstly, there is the problem of actually addressing shared data structures. How are they to be recognised and where are they to be held? Secondly there must be secure access to any such objects in a concurrent environment.

Sharing occurs for two reasons, environmental and by parameterisation. Environmental sharing arises because free variables inherited by any particular process will be shared with other activities created in this environment. Similarly, items passed as parameters upon initialisation of an instance of a process (in the case of explicitly declared

process-blocks which make use of parameter lists) may be accessible to other parts of the system. We must therefore take account of the different variable and parameter passing mechanisms. If we refer to objects by-reference then all objects (inherited or passed) are actually shared. If however we use a pass by-value mechanism then only inherited free variables are shared with surrounding activities, the initialising parameters being copied into local (and so private) space at the time of creation.

It could be reasoned that all that need be done is to enforce by-value passing and arrange for context-shared objects to be declared in some special way, and incorporate some special access mechanism to ensure secure access. It is this approach that gave rise to the concept of the monitor. The isolated scope is exploited to hide the shared data structures from their users, and the monitor's procedures guarantee that they cannot be corrupted. Aesthetically, however, the monitor leaves a lot to be desired, because whilst it takes advantage of the scope rules on the one hand it violates them by making its procedures visible outside! Undoubtedly the monitor can be forced into a language, and used, but it does not really solve the problem.

3.2 The Problem Reconsidered.

So far we have seen that the problem of shared objects involves inherited free variables (generally arising from block structured scope rules) and have countered this by

introducing the monitor-like construction which takes advantage of very elegant and powerful program structuring rules. But there was another problem, supposedly swept away by enforcing a pass by-value strategy for parameters. Here the important issue is not so much that they are passed by-value, but that in most such languages rules are introduced to restrict parameter types to those which are easily handled. The values discriminated against are the compound, or 'structured', data types. Such restrictions are directly counter to the design principles of Correspondence and data type Completeness, so eloquently expounded by Tennent [T1] and Strachey [S12].

These principles of good design require that all data types have the same 'civil rights' [S12] (so that values of any type can be built into any other data structure, passed as a parameter or returned from a function) and that there be a strict correspondence between properties of names introduced by in-line declarations and names introduced as parameters to abstractions (so that they may all have the same potential for constancy or type restriction).

Simple, or 'primitive' data values (integers and the like) pose no problem - they can be passed by-value quite easily. Compound data structures (arrays, vectors, structures etc.), however, if implemented in harmony with the principles of data type completeness and correspondence, usually reintroduce the very problem just 'solved' by the monitor.

Consider, for example, a process initiated with a compound

data structure as one of its parameters. This, by-value, may (for reasons of efficiency) be a pointer to the base of the composite value, which means that its components are effectively being passed by-reference. This process will then share the contents of such a data structure with surrounding activities -- without the protection afforded by the monitor concept. It is apparent, therefore, that in a clean programming language the sharing problem will usually still exist, despite the discipline of the monitor construct.

The usual escape is to start making restrictions as to what the programmer can do here as opposed to there, or if you do this here it isn't the same as if you do it there, or these values cannot be passed here, but they can there.... The story is all too familiar. Such rules only serve to complicate the language, lengthen its formal specification and irritate the user.

It is noteworthy that this situation is not a particular characteristic of the new concurrent languages. It exists to some extent in all popular programming languages; the consequences are simply more pronounced when combined with concurrency. The solution of a problem by the introduction of a special case, or a localised restriction is simply a short term 'fix'. Such restrictions are entirely unnecessary. Compound data structures tend to fall foul of such strategies rather often. Such things are a matter of language <u>design</u>, not pragmatic afterthought. The lesson is clear: if your language cannot handle a concept uniformly,

then live without it altogether. Concurrency is an excellent example of this dilemma. Unfortunately few of the languages into which concurrency has been incorporated have accepted it harmoniously. This is probably due to the fact that such concepts have been 'added-on' to existing languages, extending their facilities. In most cases the host languages were designed to a quite different set of criteria, often conflicting with the needs of concurrency. Such efforts, because they try to retain the facilities of the host language, resolve conflicts at the expense of the newer facilities.

It is easy to carry out character assassinations with particular points in particular languages which, with hindsight, highlight the problems. It is less easy to actually recognise the ailment (as opposed to its symptoms) and even more difficult, counter it.

In each case above, be it by-reference parameters (even if disguised in a by-value language), or items made available via a block structuring mechanism, the problem appears to concern shared data in a concurrent environment.

3.3 The Actual Problem.

Here we suggest that shared data, per se, is not so much the problem, but merely the most obvious symptom of the problem, this being the degree to which the notion of the store influences the language. The elementary notion of a store is not trivial, its effect on a language can dominate its overall appearance. Witness, for example, the differences

between the store-oriented algol tradition and the more elegant of the applicative languages [M2,T6]. It is worthwhile digressing for the moment, to underline the importance of including a store in the basis of a programming language.

A store manifests itself in a language as a location into which values can be placed (by assignment). A name, or identifier is taken to bind to or denote the location itself, not the (current) value held within it. This then introduces the notion of the l-value (address) and r-value (contents) of a name [S12], so that assignment can be thoroughly explained. That is, a name has different meaning when on the right of an assignment to when it is on the left side.

This, however, is only the most obvious consequence of having the store built into the model of the language (rather than hidden in its implementation). It has a more subtle effect. The store, and its ability to be updated, enforces sequentiality of execution. This may appear to be obvious - a program starts at the top and works its way to the end, but it need not always be so. This is because, when a store can be updated a program must be executed in a particular sequence in order to achieve consistent results. It introduces the notion of 'flow of control'. Thus repetitive constructs like while-loops and for-statements, which do not actually produce values but alter the store, appear and are accepted as natural, even though they are not essential.

The third consequence of introducing the store into a language is its effect on compound structures, which as we have seen, if they are implemented as values in their own right, are generally taken to be pointers to amalgamated components. The store has become a value in its own right (as a pointer value to a location). A data structure value is therefore a reference [R4] to the amalgamation of locations holding the actual data values.

This use of pointers to link together regions of the store is simply one aspect of the 'efficiency syndrome'; it permits large data structures to be manipulated without copying (ie efficiently), but it also introduces the problem of side-effects. Whilst this tradeoff may be quite acceptable in traditional languages, such as the algols, it is the crux of the concurrency problem.

Data structures such as arrays and the like, embody an abstraction, and in these cases this relies primarily upon selective updating of components for its effect - ie on selective assignment. In such languages the notion of the store is very strong.

We conclude, then, that it is the presence of the store in a language's realm of data values, together with the need to update locations, which is at the heart of the problem. The two combine to cause the problem, but it is assignment that is dominant. Assignment cannot take place without the store, but the store can exist and be exploited (for efficiency) by a concurrent language so long as selective

assignment is forbidden. That is, so long as the data structures are composed of values and not of locations holding these values (see the ontology diagram in the preface).

If a motto is to be coined then surely it should be this: if you have a language in which the store is already dominant, then don't try to include concurrency (it will involve lots of special cases and ad hoc restrictions and even then probably a lot of hard work). How might we overcome this problem?

Firstly we can decree that all operations on compound data types be either selectors, which do not change anything, extractors, which pull out values while leaving the data structure intact, or mergers, which take copies of data structures and merge them to produce new values, leaving the operands unchanged.

To reinforce this psychological change of emphasis we can decline to use the traditional side-effect oriented data structures. Strictly speaking this is not necessary because they could safely be manipulated by the above operations. However, it is felt that there are other data structures with even more powerful abstractions. To replace them we have selected lists and sets, which have sensible abstractions independent of selective updating. Lists are the traditional composite structure of the applicative languages, we use them in much the same way. In the case of sets, however, we are being slightly more adventurous,

because they are implemented as unordered collections of values, where the various values can be of any arbitrary mixture of types. Both compound data structures are values in their own right and both may contain other compound data structures as components.

There is no need to update the components of such data structures, so we will forbid it. In fact we lose little by selecting alternative data structures, since we can always index into a list, just as with a vector, whilst we gain much in the form of 'extra' properties, such as sublists (the equivalent of slicing vectors). It is not the particular data structures which we implement that is important, but rather the underlying principle that is the crux; namely that selective updating is an indirect form of sharing and should therefore be banishhed.

What of direct sharing of data ?

It seems that the well trodden road towards the monitor could be a wrong turn! At least in the sense that it is not so much a solution but a convenient 'ad hoc' technique which hides the problem (for another day), and seems to hinder the development of a 'clean' language, by itself imposing and then breaking rules (the violation of scope rules being most blatant).

Here we shall abandon the monitor construct and its precursor in the form of critical regions, and we shall consider afresh the real problems, without requirements to 'extend' an existing language. We will therefore build up

an entirely new approach, incorporating only those concepts which support, rather than hinder the expression of interactions in a system of concurrent processes.

Before we move on however, it is interesting to note that the originators of the monitor have also had second thoughts concerning direct data sharing. Both Brinch Hansen and Hoare have recently put forward proposals for languages in which there is no shared data. Brinch Hansen [B24] proposes that processes be lexically distinct, but in order that they be able to interact he proposes that procedure 'entries' be visible outside their native scope (so breaking the scope rules). This idea, also at the heart of the Ada approach, calls for processes to call on one another's procedures, and because a process can only carry out one activity at one time it again enforces sequential access to 'local' data indirectly accessed from outside.

It is significant that Brinch Hansen's scheme does not include procedures as 'first class' objects, they are not values, and so cannot be passed around from process to process. They are made visible by composite naming conventions. Brinch Hansen's strategy actually relies upon this fact, because the desired sequentiality enforced by entry procedures would not exist if the procedures were passed out to other processes so that they execute them, while the 'owner' does something else! This, incidentally, is why we will have no procedures in our language - if they were to be included at all then, in keeping with our adopted principles of good design (Correspondence and Completeness),

they would have to be included as values, not denotations. Recall our tenet about only including a concept if it can be handled uniformly. Designers, we suggest, would do well always to apply this version of Occam's Razor.

Whereas Brinch Hansen retained much of the flavour of the earlier approach, merely replacing the monitor with another process, Hoare [H13] has proposed something more radical. He suggests that input/output is a fundamental operation and that all processes should interact only by specific i/o operations. He goes on to suggest a syntax for sending and receiving values - messages - between processes. To simplify matters he recommends that related input and output be synchronised, so that either a sender or a receiver is forced to wait until the other is ready, at which time the values being passed are instantly exchanged. Many have heralded Hoare's CSP, as it has come to be known, as a major step forward. This is so. Unfortunately it, like its predecessors, imposes a 'second class' status on its processes. We shall return to Hoare's proposals in a later section.

3.4 A New Approach.

It is a characteristic of the languages produced so far, that the processes they envisage are introduced as denotations rather than as values in the language. That is, they are names representing static textual definitions, the task-bodies, rather than the dynamic incarnations of such definitions. This use of the term process is undesirable,

because it should really denote the dynamic execution of that definition. If there should be more than one such entity then they should be distinct processes, even though they execute the same textual definition. There is generally no means provided for identifying particular members of the system when it is running. This anonymity is actually regarded as advantageous by some (such as Modula [W16], and Ada [B5,I2]).

The language proposed herein includes both the static definitions and their individual incarnations as separate data types. That is, executing entities as values of type instance, are part of the value domain of the language. An instance is therefore different from its specification, being of type definition. Both are data types and, in accordance with good design, they are both 'first class' values (in the sense of Strachey [S12]). This fact is clearly seen in the Ontology diagram in the frontispiece.

The implementation of values of type instance is absolutely fundamental to the proposed approach to concurrency in a programming language. No restriction whatsoever is made on the numbers of simultaneous instances of a given definition, each is distinct and is uniquely identifiable. Such values are produced when a definition is initiated. An instance of a definition can be formed at any time, it can even occur recursively.

If we accept the critical distinction between the usual meaning of the term process, or task, and the envisaged

concurrent instances [H4], then we can go on to investigate and exploit message passing as a means of communication. We introduce message passing as a fundamental mechanism in a language, here called Protocol, which elevates processes to 'first class citizenship' [S12].

Message systems, as such, are not untried. Several implementations have been reported [for example G1,H1,R6 and L9], but each has tended to superimpose it upon some other system, rather than introduce it as part of the fabric of the system. Perhaps the best of these proposals is the MIT 'guardian'-based system [L9], but even this passes its messages to ports (developed originally by Balzer [B4]) instead of directly between the processes.

Such systems have appeared to be rather ad hoc, with arbitrary restrictions on the nature and quantity of messages being passed. In some [eg B1Ø,E4,H13 and W9], where the message passing is fully synchronous, they have also tended to have adverse effects on the underlying system's scheduling strategies. The 'bad press' often directed towards message passing systems is considered to be very much a criticism of their various implementations, rather than as a mechanism for communication.

4.Ø A Means Of Communication.

Concurrent instances must be able to communicate, with each other and with the outside world. Most existing languages promote process interaction via shared data structures, within the system, and have separate i/o statements for

communication with the outside world. Since message passing is a more natural method of communication we will use it to unify internal interprocess communication with traditional i/o operations. Since the instances and their definitions are an integral part of the proposed language itself, we suggest that the message passing constructs be part of that language too, and not some operating system provided facility, added later on.

The message system proposed here is therefore fundamental to the language, in much the same way as parameter passing is in the case of procedures. Indeed, the analogy is quite appropriate, as it is suggested that a message system for parallel instances is the concurrent equivalent of the sequential procedure's parameter mechanism.

All communication is undertaken by message passing, as there are no shared data structures (and so no need for complex mechanisms for arranging exclusive access). This philosophy is similar to that which suggests that procedures should only interact via their parameters, not via global variables. Indeed, the case against indirect interaction is even more justifiable when concurrency is incorporated.

A message passing facility within the fabric of a language for a system of concurrent instances, based on the analogy with the parameter passing mechanism of sequential procedures, with values being passed as messages is considered to be quite new. Its uniformity makes it far more intellectually manageable than previous systems. This is

especially true if all objects in the language are values, and if all values have equal citizenship.

The guiding tenet is simple - anything can be passed anywhere -- and the entities doing the passing are themselves values. The principle of data type completeness [R4] is rigidly enforced, so any data value can be passed as a message. The writing of general purpose programs, ie software tools, is encouraged, simply by this uniformity.

4.1 On Type Checking.

The topic of type checking is rather contentious. There are various schools of thought, and matters of type checking should be resolved very early on in the process of a language's design, as such decisions can either free or strangle it.

To some, types are unnecessary and so 'typeless' languages appear [for example G17,M4,R4,R5,W21 and W22], in which the user specifically models the types he needs from raw bit patterns. There is obviously no checking, so the matter of compile time versus run time action is immaterial. Others [such as K5 and I1] maintain that types are merely a means of specifying to the compiler the kind of storage required for any given object, and so employ them while encouraging relative freedom of use. To others again [M6,H10,H12,S2,T5 and W15], types are of paramount importance in a language, and they recommend that a compiler should check programs thoroughly, before they are run.

The pros & cons of type checking are open to controversy. There is a case for having the compiler detect errors, and prevent a faulty program from running, but there is the point that if taken to extremes, complete checking undertaken by the compiler begins to restrict the language quite severely. It is, of course, feasible to check types dynamically, if the appropriate mechanisms are built in properly, and this has the advantage that it can impose strict checking without restricting the language so much. If an operation makes sense at run time then carry it out. The old banner of efficiency can, of course be raised [B13], but there are measures which can be taken to ensure that total checking can be achieved largely unobtrusively.

There is therefore a balance, between the two extremes, and this is where we find our language. Type checking is considered to be very important, but as it would be almost impossible to check types at compile time in such a generalised message passing system, type checking is undertaken mainly at run time.

Checking is carried out dynamically by operators, which verify the compatibility of their operands. In contrast, the message system itself is polymorphic, since it allows values of any type to be passed as a message between any pair of instances. This combination of freedom (to pass any values anywhere) when combined with the protection (validity of operations on values) of strong type checking is highly desirable. By distinguishing between passing values and operating on values we can avoid the complexities of the

typed-channels or typed-ports which have plagued previous attempts to combine message passing with strict type checking (see the Thoth system [G1] for example).

Most computers are constructed along the lines of the original von Neumann architecture [N4], in which memory is organised as a contiguous area of identical locations, the contents of which are indistinguishable. Here we implement dynamic type checking via a tagged architecture, the tags being used to hold type markers. The tagging of memory locations, and the widening of the data busses to accommodate them, is not a new concept, but few real computers have been built with this feature. Notable exceptions are the Rice R-2 research machine [F4] and the Burroughs B5000 machines [O1].

Feustel [F3] and Myers [M8] have suggested that such tagging could make high-level languages both easier to implement and more reliable, with more compact code. It is a pity that most machines follow the von Neumann design. Several workers [B8,C4,C5,D5,H8 and M3 for example] have discussed the degree of support given by machines to programming languages. The general reluctance to build 'better' machines has been analysed by Frailey [F6], who concludes that as manufacturers are wanting volume sales of compatible machines they have been unwilling to invest large amounts of capital in non-standard architectures. The relatively flexible and aggressive market of the microcomputers might well see such experimental work brought out into the open. It is to such a tagged machine that our language is aimed.

If the computer being used is not of such design then this must be simulated. As we shall see later tagging provides an extremely simple means of producing run time diagnostics.

4.2 Definitions, Instances & Messages.

A system [H4] is considered to come into existence complete, with an infinite number of instances, although initially only one of these, the user's instance, is active. Most of the instances have nothing to do, they are dormant, awaiting a definition to execute. Because they have not been allocated a definition, most instances in the system, are anonymous. An instance only becomes known (ie its value is made available) when an already active instance requests the system to allocate another instance to execute a particular definition. This can occur at any time, by using the new operator, which takes a definition value and returns a new identity.

For example:

```
let new demo -> x
```

will simultaneously declare "x" and initialise it with a value of type instance. The entity in the system to which the value in "x" refers then starts executing the definition known as "demo".

The symbol "demo" is therefore a literal of type definition. This literal will have been defined earlier on, by requesting the system to store a statement away in its library and return a value which will thereafter denote it.

The definition values can be regarded as an incrementally enumerated finite set, the members of which are known by unique literal symbols.

Concurrent instances may send and request messages at any time, unlike the sequential procedures which take parameters once, on entry, and may produce one value, on exit. Instances may thus interact with each other while executing.

Naturally messages have to be directed to their destination and, correspondingly, instances awaiting a message may specify where it is to come from. A message transaction employs a value of type instance to indicate the desired member of the system. The system automatically transfers the messages, all that need be specified is the target and the value.

We propose the following concrete syntax. Sending a message takes the form:

instance.value _ message.value

where the two expressions are evaluated to produce the target instance and the actual message value. The "_" is the concrete symbol chosen to represent the send operation. Requesting the acquisition of a message takes the form:

(instance.value) -> xyz

where the expression in the "(" and ")" brackets evaluates the source of the message. These brackets form the concrete representation of the read-in operation. The "->" symbol is

the assignment operator. A complete language definition will be given later on. Here examples are merely intended to give a taste of the concurrency related constructs. The message so acquired will then have to be used elsewhere, so this forms an expression, here it is assigned to the variable "xyz".

Such a system can only be achieved if the instance's identities are values in the language, unless, as in Ada, there is a restriction so that there is at most one incarnation of any given definition at any one time, in which case the task 'name' degenerates to identify that single instance by default.

To create multiple coexistent copies of given tasks languages in the Ada tradition invoke 'families' by using special array constructions, where the subscripted names select the tasks required. This feature is, however, not as general as truly first class processes since the individual members of such families cannot be passed around within the system: they are not really values in arrays, but subscripted names, borrowing the syntax of an array.

4.3 Networks.

Since all values have equal status, and so can be passed as messages within the system, it is perfectly feasible to send an instance's identity across the system to place statically unrelated instances in direct contact with each other. For example, if an instance is to set up a pair of other instances, initialise them and arrange for them to

communicate with each other, then all it needs do is initiate each, recording its instance value, and then send to each any initialising message, together with the identity of the other. They can then communicate directly, leaving the creator free. It can proceed independently, interact with each if necessary or may even terminate.

It is thus possible to build a dynamically variable network-like system in which new communications paths can come into being spontaneously without the need for such artificial facilities as the virtual ports [B4] implemented in the 'guardian' system at MIT [L9] and in an extension to Hoare's CSP by Silberschatz [S8].

In our case instances have two predefined literals, "input" and "output", forming the default communications paths for messages. These are both initialised dynamically to the creator of an instance, so that different incarnations of a given definition can have different default communications paths, depending upon where they were initiated from. The system's communications system is, therefore, initially hierarchical, but as pointed out it can become as convoluted as necessary merely by passing around the identities of the members of the system.

Two-way communication with the outside world is achieved by sending to, and receiving messages from, the "world", a predeclared instance literal, thus unifying internal communications with i/o activities. Standard inlet and outlet for the initial instance are dynamically initialised

to the "world" value.

4.4 Message Queuing.

If the system were so primitive as to require a receiver to restart as soon as it was sent a message, interrupting the sender then overall performance of the system would be very difficult to manage. It is suggested, therefore, that there be queues attached (invisibly) to each instance, into which messages are delivered. Each message is tagged with the identity of its sender. It is now possible to have a stream of messages arrive at an instance, possibly being sent by different instances in the system, all intermixed in the queue. When that instance eventually requests input from a given source the system looks along its input queue, matching the identities. If an input is found then it is 'read in', otherwise that instance is suspended until after such a message actually arrives.

4.4.1 Asynchronism.

It is obvious that the instances can communicate asynchronously, rather than by enforced synchronised rendezvous, whenever it is a simple one-way send that is required, leaving the sender free to continue immediately, having dispatched its message. Each instance can therefore run at its optimum rate (processor sharing aside).

4.4.2 Synchronism.

Equally obviously, in interactions where synchronisation is essential, the send/receive operations incorporate the

necessary scheduling requirements, as can be seen in:

```
( instance.value _ message.value ) -> xyz
```

for example. The implicit synchronisation in the operations, due to the relative precedences of sending messages by "_" and requesting messages be read-in, by "(" ")" brackets, sending occurring before the wait, makes message passing constructs 'high-level'.

4.5 On Storage Management.

As all instances are active 'simultaneously' a simple stack mechanism is insufficient. Although a cactus stack would be sufficient its linkage is not actually necessary, since instances have no shared data, so a heap based implementation is employed, with the heap being carved up into independent miniature stacks, one per instance, so that each can expand and retract without interference.

4.6 The Message Pool.

Messages, being sent asynchronously, can accumulate in queues attached to instances. These are implemented simply as linked lists in the heap, each link holding an 'input'. Each input consists of the actual message value, its type tag, and the identity of its sender. Naturally the numbers of messages permitted to pile up depends upon how liberal the system is with its heap space.

4.7 Garbage Collection.

A garbage collector runs in the system, effectively as one

of the instances, known only to the scheduler, reclaiming used space in the heap. This works in collaboration with the scheduler. The scheduler's strategy can be dynamically variable, to take into account the garbage collector's activity. This is a matter for fine tuning.

5.Ø Communication Facilities.

We will see that by elevating the concurrent instances to first class status, by making them values in the language, it is possible to construct systems of parallel processes which are truly dynamic. By adding message passing operations as primitives too, we have been able to incorporate asynchronous interactions and, where necessary, the automatically synchronised transactions of two-way communication.

As the system evolves dynamically, and the interactions can take place asynchronously, we must provide facilities which permit the instances to act non-deterministically. This we will achieve by permitting instances to interrogate the system, to find out about each other, and of the origin of the messages in their input queues. It is by examining their environments that the instances can dynamically match their activities to a varying workload. As we shall see, it is possible to express local scheduling algorithms directly in their definitions.

Having established the basics of starting up an instance from a definition and of sending and receiving messages, we can exploit the simplicity and conciseness to accommodate

the new requirements [H5].

5.1 Identifying the activity being undertaken.

The instance values are unique and serve to identify particular members of the system; this is why we use them to direct messages. The definitions, being distinct from their incarnations as instances, can be related to individual instance values to determine whether an instance is executing that particular definition. It is therefore possible to find out, dynamically, what any particular instance is doing (in terms of the available definitions). For example:

```
if x is demo then ... else ...
```

where "x" is a value of type instance and "demo" is a value of type definition. This is self explanatory.

5.2 Data flow in the system.

Instances communicate via messages, and messages are sent asynchronously. Messages accumulate in queues attached to the target instances, from where they can be 'read in' when needed. These queues serve to buffer the transfer of data within the system. By absorbing variable transaction rates by holding temporary overruns the queue mechanism implements the synchronisation. It is the flexibility of the message queue which makes the system tolerant of unpredictable data rates, and it is this which enables us to separate out user-oriented scheduling requirements from system scheduling constraints. As we shall see later, the user can write

private scheduling algorithms directly.

The asynchronous nature of a queue means that messages can arrive from a variety of sources, in any order, whilst those from a particular source may be read in sequence, irrespective of any other messages, from other sources, already in the queue. By examining the queues, to determine the origin of the messages, and the nature of the senders, nondeterministic interactions can be undertaken.

5.2.1 A Continuous Record.

In situations where a continuous stream of data is required it is merely necessary to request that the next message from that source be read in (with an automatic wait should it not yet have arrived). This is the default action, but can be explicitly specified by using the keyword first, eg:

```
( first x ) -> xyz
```

5.2.2 An Up-To-Date Record.

In some cases it is essential that only the most recent data be used, and that all intermediate messages from that source be discarded (but leaving messages from other sources undisturbed). This can be specified quite easily by the keyword latest, eg:

```
( latest x ) -> xyz
```

These simple mechanisms provide quite a powerful data handling facility.

If an instance, such as a line printer driver, is neither

interested in who it serves, nor in what they are doing, then it can use the predefined instance identity "system", which matches against any instance. This permits an instance to serve others in general, without taking specific action to find out the identities and requirements of all the members of the system.

The "system" instance valued literal, when used for input of a message with either the first or the latest options, accepts either the first or the latest message from any source. When used for output this instance identity serves to send a copy of the given message to every active instance in the system, providing, as a result of the uniformity of the system, a 'broadcast' facility:

system _ xyz

Since every instance has an identity of its own, it is only natural that an instance is able to access its own identity, by way of a local constant called "identification", which it can then pass around the system, to other instances, and so identify itself explicitly. Because the system is orthogonal it is even possible, although somewhat superfluous, for an instance to send a message to itself!

5.3 Interrogating Message Queues.

Merely requesting that a message be received incorporates an automatic delay until such a message becomes available, if one is not already in the queue. There are situations, eg in an instance serving a variety of other instances, where this

enforced delay is unsatisfactory. In such cases it is more sensible to cycle, polling the queue, in order to decide what action to undertake next. It is therefore necessary to provide a means of interrogating a queue to determine whether or not there are messages available from particular sources, or classes of sources.

This can be achieved simply by the message predicate:

if message x do ...

which, given an instance value reports whether there are any messages currently in the queue from that source or, given a definition value, reports whether there are any messages from any instance executing that definition. If no argument is given then it simply returns whether there are any messages at all, from any source. We shall have more to say on extracting information from the message queues in a later section.

5.4 Scheduling.

Being able to interrogate message queues, identify messages, and select the order in which to consume them provides the basis for a user-oriented scheduling mechanism.

For example, if an instance services messages from a variety of producers, then it can select its work according to some in-built priority scheme, written by the user. Although the system automatically buffers messages, the user could simulate this if he wished. The classic reader/writer problem could be written so that the 'common buffer' is

actually managed by a separate instance; this taking requests from a "reader" and a "writer" instance. The buffer manager then serves the writer before the reader simply by:

if message writer then ... else ...

This is simple for the trivial case of a single writer and reader instance pair. If, however, there were many readers and writers then the buffer manager would have to discover which of its messages came from instances executing a write-oriented definition, and which were from a read-oriented definition. This can be done using the is operator mentioned above, or by using the write-oriented definition value, to find out whether there are any messages from any writer instances, whoever they may be, before serving read-oriented instances. This is extremely concise!

So far all communication has been with either instances which know of each other personally, or with instances of known definitions. If interacting directly they must have been placed in contact with each other either by being directly related, via default communications paths, or by having been sent their mutual identities via the message system, so that they could communicate directly.

Any instance can send a message to any other instance, so long as it has been given its identity. However, that instance might not know of, and so will not be expecting messages from, such an instance, and because it does not know its identity it cannot specifically ask for its message

to be read in! If the definition that such an instance is executing is known in advance then, as in the case of the extended reader/writer problem, the message could be retrieved by asking for the message from an instance of that definition. If however the receiver does not know of the sender directly, and does not know which definition it is executing, then it simply knows that it has a message from somewhere, but has no way of directly asking for it. The generality of the message scheme being outlined permits even this sort of situation to be managed sensibly.

Apart from polling the queue to see if there are any messages, either at all or from specific sources or classes of sender, it is possible for an instance to request the identities of all senders of messages currently in its queue. The system primitive census achieves this. It returns a set (sets are one of the data types in the language) of identities.

This set value can then be examined to decide how to manage the messages in the queue at that time. Unlike the above cases, where only previously known instances could be served, the receiver is now given the identities of all of the senders, from which it can find out what they are doing, and so ask for their messages in whatever order is deemed appropriate.

The mechanism is completely general purpose: if census is given an argument, corresponding to a currently active instance in the system, then it samples the input queue

attached to that instance, otherwise it examines that of the current instance.

A two-way link can establish itself automatically, as once one instance knows another, the second can find out the identity of the sender of the 'anonymous' message. This is quite an attractive capability, and it permits a system of interacting instances to 'grow' new communications paths as and when desired.

As a data type a set is merely an unordered collection of values. The set returned by census is a set which is composed entirely of instance identities. There are various operations on a set, one is a simple membership predicate:

if census has x then ... else ...

where 'x' is again a previously known instance value, would test for messages from the above mentioned instance. This is as before. The advantages are only realised when the set of identities is used in conjunction with the is operator discussed earlier, or the classof operator. The is operator reports whether a particular instance is executing a particular definition. The classof operator takes an instance and returns its definition value. This definition value may not have been known to that particular instance before. By testing elements of the census set the senders, even if previously unheard of and so with unknown requirements, can always be identified in terms of 'what they are doing' as well as 'who they are'. This permits scheduling-by-activity, as well as scheduling-by-identity.

For example, if we take a census of the queue and save it, by:

```
let census -> ids
```

then by using an iterative set element selector (which applies a statement to each member of a set, in some 'randomised' order) we can service all of those messages from instances performing a certain function, whatever their identities:

```
forall sender in ids do
if sender is tape.driver do ...
```

In this way it is possible to test the senders against known definitions, if it was necessary to find out the definition being executed by any given instance then the classof operator will report this. It is possible, therefore, to write a definition which records the names and activities of all instances which send it messages, even if they were unheard of before the message arrived. It is even possible to write a definition which, depending upon some test, is required to invoke a new instance of its creator:

```
if ... do new classof input -> x
```

and so replicate some arm of the system hierarchy.

Note that such high level scheduling is based on the nature of the concurrent entities and their activities, rather than on some 'ad hoc' priority scheme. This mechanism enables the user to write his scheduling algorithms directly into general purpose message managing systems.

These facilities, taken together, support many data handling and communications requirements, and permit instances to schedule their activity in accordance with dynamically variable workloads.

Having outlined how concurrent processes can interact with each other, both asynchronously and nondeterministically, we will move on to explain how they can widen their horizons by examining their environment, inspecting one another directly.

6.Ø Managing A System Of Processes.

Here we shall develop facilities for handling tasking exceptions arising in and propagating thru such an interacting system. These permit processes to examine one another, in order to avoid failure, and enable supervisory processes to manage the activities of others. We shall then investigate the need for a means of dynamically detecting deadlock, and show how this can be removed by exploiting such task-oriented operations. It will be shown that these facilities can be provided inexpensively in a message-based language provided the concurrent processes are values in their own right.

6.1 States And Tasking Exceptions.

Whenever an instance wishes to send a message to, or receive a message from, another instance in the system it specifies that instance's value. At this time the system automatically

checks that the companion still exists in the system and that it is running normally (ie that it has not suffered an exceptional condition). If the specified instance has either terminated, or has aborted prematurely due to some run error, then the instance requesting a transaction is itself aborted; it being an error to attempt to interact with a faulty part of the system. Other than by being preempted for this reason, an instance will abort if it encounters a dynamic type mismatch or other such data-oriented error.

It is apparent, therefore, that a system of highly interacting instances will fall apart if one of its member instances falls foul of even a type mismatch in its data!

In order to make a system more robust, so that it is not so sensitive to such 'trivial' and highly localised errors as mismatching data, types are themselves values in the language. For example, the literal int is a value of type type, with value integer, as opposed to something of type int with an integer value. An instance can therefore examine its inputs, in order to verify them, or to select which of several actions to take on this basis. A user who misfeeds data can then be prompted for more appropriate input, and the system will survive the error, rather than this causing some part of the system to collapse in a chain reaction of tasking errors.

In the same way as a location can be examined to find out the type of its current contents, a value of type instance can be interrogated. It is possible to determine what it is

doing, in terms of its definition, by using the classof operator mentioned earlier. Its current status, that is whether it is running normally, is finished or has aborted, can also be determined. We see, therefore, that just as an instance could protect itself against faulty input values, it can protect itself against faulty transactions too. If an instance examines its companion immediately before it interacts with it then it can be forewarned of a fault propagating thru the system, and so avoid contact with the ailing instance. Having detected this failure it can then voluntarily abort or follow up some contingency strategy. For example, if it was a line printer spooler, serving an entire system, and it noticed a failure in the instance that it was currently serving, it could simply abandon that particular activity and service the next request. Such 'service' oriented facilities would therefore have to protect themselves, by being cautious, and examine their environment before acting.

Whether or not a system is protected depends upon the user, because he must write definitions which check before acting, and he must specify what they are to do when failure is detected. If he naively writes his system such that it assumes valid data and no unforseen events then any error will propagate. The important point is that the facilities for providing protection are there and they are present in a thoroughly natural fashion, in a high-level form, in the programming language itself.

6.2 The Status Of An Instance.

To acquire the status of an instance the operator stateof is provided. Given an instance value this will return its current state. For example:

```
let stateof worker -> x
```

The value returned is of type status. The values of this type comprise a finite set, one for each of the possible states of an instance. Each value has its own literal symbol, for example:

active : running normally

complete : completed normally

aborted : terminated abnormally

waiting : as yet undelivered message

are all values of type status, and like all other values they can be passed around freely. This is rather similar to the set of boolean values, comprising true and false. A separate type was introduced for the status of an instance, rather than employing some integer values, say, because the distinct type serves as a measure of protection and also enhances the degree of abstraction.

There is no need for 'nil' values corresponding to as yet uninitialised instances, because instance value only come into existence when a definition is started up, at which time it assumes the active state. An alternative explanation could proceed as follows: there are actually an infinite number of instances in the system, most of which have not been assigned definitions to execute, and so have not had their instance values made available - so they are

essentially unknown and cannot be asked about. The two approaches are equivalent.

This interrogative operator now enables the user to write definitions which examine their environment before attempting any interaction. Consider, for example:

```
if stateof worker = active then ... else ...
```

or, more interestingly:

```
if stateof worker = waiting do ...
```

which leads on to the enquiry holdup. Given an instance value that is currently in a waiting state, this reports the identity of the instance in the system upon which the worker instance is waiting. Thus:

```
world _ holdup worker
```

will show the user at his terminal exactly why the worker instance is waiting. This can be built into a loop to follow a trail of waits to its source. It is now possible to trace out a chain of waiting instances, in order to identify the cause of the delay.

6.3 Tasking Exceptions.

When the system aborts an instance it is because it suffered a 'hard' failure. Such faults include type mismatches which the user did not check, or an overflowing stack, or an attempt to interact with a faulty part of the system. The exact nature of the error is recorded by the system as it

deletes the main part of the ailing instance. This information is made available to the user, should he want to use it. Such values are of type exception. There is a predeclared exception value for each possible hard failure. The exceptions form an enumerated type, as did the status and boolean values, and are available as literals, eg Overflow, Mismatch and Interaction.

It is therefore possible, on discovering that an instance has the aborted state, to enquire as to the nature of its fatal act. This is achieved by the faultof operator which, given an instance value, will return its exception condition. For example:

```
if stateof worker = aborted
    do world _ faultof worker
```

will display the exception to the user. An exception is raised when an instance aborts. This may happen because the instance commits an error, or voluntarily, because it finds itself unable to continue.

The user can introduce his own exceptions, and is free to pass them around within the system as necessary; exceptions have the same rights as all other types in the language. A new exception literal is created by introduce which is a predefined literal for an instance which updates a list of user-defined literal symbols associated with the requesting instance. Thus:

```
introduce _ Too.Many.Items
```

will add a literal of type exception with Too.Many.Items as its basic symbol.

Since an instance can only generate an exception as it fails, there is an abort mechanism which takes an exception value and terminates an instance signifying that as the fault. The name "abort" is another predefined literal for an instance, in this case it kills off the indicated member of the system. Unless an instance value is specifically given, abort terminates the current instance. For example:

```
abort _ Too.Many.Items
```

will terminate the current instance with the stated exception. Similarly:

```
abort _ worker, Too.Many.Items
```

will evaluate worker and terminate that instance. It is possible for any instance to abort any other instance that it knows of, with any available exception value. The mechanism is quite uniform.

This provides the user with an application dependent means of introducing failure, for cases where progress is no longer possible.

In the language envisaged here the exception is simply a value indicating the reason for failure of an instance to terminate properly. The exception is not a general purpose shared signal mechanism by which processes can synchronise. This limited application contrasts with the PL/I-like 'events' [Il], which can be used for explicit interaction. We provide a completely buffered message passing facility for such communication.

Values of type exception are more like the exceptions in

Ada [I2], to the extent that when raised they terminate a process, but here exceptions cannot be trapped by 'handlers'. That is, there is no when-clause, or on-condition. The philosophy advocated here is rather different: the user should be able to interrogate every aspect of his data, and the system itself, so that he can, if he wishes, make his programs 'fail-soft'. To achieve this he has various primitives, including stateof, holdup and faultof which together with the typeof operator and the message, census, first and latest facilities enabling an instance to select its work according to a private scheduling strategy, provide a much more powerful facility.

The language is specifically designed to exploit the underlying system, and that system relies on the expressibility of the language - it is a symbiotic relationship. The user is given full access to the available information by way of operators which extract specific values from the system.

The simplicity of the mechanism encourages the user to develop more sophisticated system-probing operations, and the orthogonality of the constructs ensure that there are no artificial restrictions placed upon him in his efforts. Such constructs fit into a language in which the running entities, the instances, are first class values; and they harmonise extremely well with the concept of message passing. Such a scheme is both high level and intellectually manageable, because the user is more concerned with the interactions in his system of processes,

than its implementation.

Consider the following system-interrogating definition:

```
let survey -> sys
world _ "Survey Of The Entire System"
forall id in sys do
{
   world _ id
   world _ stateof id
      if stateof id = waiting do world _ holdup id
   if stateof id = aborted do world _ faultof id
}
```

which, apart from elementary layout of the output, will report on the current state of all the instances in the system. Note in passing that the survey operation was used. This is similar to census introduced earlier, but it works on the system as a whole, rather than on a particular input queue. If it is given an argument of type definition it returns the identities of only those instances executing that definition, otherwise it returns all the instances in the system.

The design tenet is clear - all constructs should be orthogonal, and every value should have the same rights, so that it can be passed around in safety. Any irregularities immediately undermine the generality of the mechanism.

6.4 Deadlock.

One of the major problems facing the designer of a system of concurrent processes is the potential for deadlock, or the 'deadly embrace' [D11]. Current works, eg Hoare's [H14,H13], have suggested that a static analysis should be employed to generate 'partial proofs' that deadlock is

absent from a system. Early efforts along axiomatic lines [A4] seem to be encouraging. Techniques based on graph theory [J3] are also being studied.

Such a strategy is, however, entirely inappropriate in a concurrent system of the sort proposed here. This is because of a fundamental difference between this and previous languages developed to express interactions in systems of concurrent processes. In Hoare's CSP, probably the best so far proposed, the 'processes' he employs are names in the program, not values generated dynamically within the system to denote the running instances.

It is this distinction which provides the power of our approach, as the processes, or instances, can be passed around within the system, making it possible to establish direct communications links between any members of the system, as and when necessary. The advantages of awarding the parallel processes full civil rights were discussed earlier. This freedom renders a full static analysis of likely interactions quite impossible, as the members of the system develop their message routes dynamically. It is therefore essential that such a system has deadlock detection built-in at run time.

6.4.1 Detecting Deadlock.

Here we are not so badly off as most other systems which have the problem of dynamically detecting deadlock. The complexity and expense of deadlock-related schemes is probably attributable to the lack of suitable information on

the state of the system. In our case, however, this difficulty is somewhat eased. As a by-product of providing access to the current status of instances in the system (and if they are waiting, then who they are waiting for, together with the nature of the hold up), the system always has sufficient information about the state of its members to permit it to detect some elementary forms of deadlock (an Eulerian chain for example). Moreover, all this information is readily, and inexpensively available to it.

The time to look for deadlock is system dependent. It might only be undertaken when the system scheduler finds that all its instances are waiting; or it might, like the garbage collector, be an instance in the system, known only to the scheduler, running incrementally. This is a matter of detail, and is of no real interest here. The important point to note is that the system holds all the information needed to detect trouble, and this is already in a form suitable for surveying the system.

The overheads incurred by any deadlock-detecting system, as with garbage collection, should be absolutely minimised, as they are due to a 'scanning' process which may strike often, while contributing nothing to the thruput of the system. Static analysis for detecting deadlock depends upon many simplifying assumptions, most notably the ability to deduce communications paths by examination of the source text, and as this is frequently not the case it is often necessary to resort to dynamic detection of deadlock.

The costs of such a scheme then depends upon how readily the necessary information can be collected. In this system, where the message passing and exception facilities exploit the fact that the processes are values in their own right, this information is readily available, and so the costs of detecting and removing deadlock can be minimised.

6.4.2 Removing Deadlock.

Furthermore, the user can dictate how the system should attempt to recover from deadlock, because the system (in the experimental implementation developed for this project) looks for a specific definition, from which it will create a deadlock-scanning instance. This definition can be written by the user, and so be application dependent. In this definition he can interrogate the system to discover which instances are involved, what they are doing, and the cause of the deadlock. This instance can then selectively delete all or part of the crippled section of the system. Those left running will, of course, have to have been written so as to recover from such an interruption. In this way the user can preserve 'service'-oriented instances caught up in system failures. The system's default action, if such a definition does not exist, is to abort all of those instances involved in the deadlock.

6.5 On Correctness And Survivability.

As we have seen, the subtle shift in attitude, making processes values in the language's universe of discourse rather than named pieces of text, has the effect of making

it generally impossible to deduce the participants in any particular interaction until it occurs. This expressibility is at the heart of the power of the system. It is however, in direct conflict with the much coveted goal of language designers of being able to submit a program to a static correctness check before execution starts.

In the same way the polymorphism of the message system, which permits us to write general purpose definitions which handle all data with equal ease, is together with the ability to have general purpose variables, in direct contrast to the growing support for static type checking systems.

This is the classical conflict between language expressivity on the one hand and data security on the other. In the past this has been tackled by increasing security at the expense of expressivity. Here, however, we take the view that strict adherence to static techniques is, in the long term, detrimental to the development of advanced programming languages, and that this conflict of interests should be resolved not by sacrificing either of these highly desirable goals, but of something else. We propose to trade these off against run time efficiency, by undertaking comprehensive dynamic checks. Dynamic type checking is not new, of course, but the extent to which it has been built into our system, in all its aspects, and particularly with regard to the implementation of instances as values, is quite unprecedented. Protocol is extremely expressive yet totally secure.

The consequent inability to perform static correctness proofs lead to the development of a somewhat different philosophy towards failure. Instances can be written so as to embody a high degree of intelligence, because they can inspect their inputs and be suspicious of their partner's ability to perform. Indeed the safest members of the system would be over-inquisitive, in self-protection, perhaps to the point of appearing rather neurotic, since more often than not their suspicion would be quite unnecessary! The basic requirement when writing the definitions of the instances which will form the system is to check immediately before acting. This contrasts with the static analysis which leads to a certificate of correctness before execution begins, or the somewhat ungainly recovery techniques proposed for languages such as Ada.

All this naturally leaves the naive system wide open to accidents or malicious attack by irresponsible members, but in a well designed self-sufficient system such damage or sabotage would be detected by the proposed victims and evasive action initiated, perhaps even leading to the demise of the perpetrator!

We therefore advocate dynamic checks, with a comprehensive but orthogonal set of probing operations to permit instances to examine their environments, so that they may assume responsibility for their own survival. In the rather expressive system that we have designed this is deemed preferable to restricting that freedom simply to permit rigorous static analysis. Remember that even in certified

systems accidents occur! At least an inquisitive system will notice this instead of blindly carrying on.

The emphasis here is towards robustness, survivability rather than pronounced correctness. It is only by acknowledging that failures occur in real life that fail-safe systems can be constructed.

6.6 Distributed Systems.

The system, in the form of a infinite number of instances, working concurrently, has so far been discussed without reference to its physical implementation. For example, we have not determined if the instances do, or indeed can, execute truly concurrently, or whether the parallelism is simulated, on a single processor. We will now show that the system does not depend upon simulated concurrency, as it can take advantage of highly parallel multi-computers, to form a truly distributed system.

If we acknowledge the existence of multi-processors, by introducing a new type, "machine" say, with as many values of this type as there are processors available, then the notion of an instance value can be seen to be sufficiently high-level to form a useful abstraction of the activities in such a system.

This set of machine values, like the sets of definition and exception values, can be incrementally extended, to accommodate dynamic variations in a hardware configuration. This could easily share the syntax for specifying new

exception literals:

```
introduce _  Z.8Ø , machine
```

for example. This would also require that the new machine value be bound to some particular piece of hardware. Naturally the hardware drivers for such processors would have had to have already been included in the implementation.

Any instance starting up a companion could then have the option of specifying which machine it wished it to be executed on. This would be a natural extension of the new operation, eg:

```
let new demo on Z.8Ø -> x
```

If none is specified then it is left to the system itself to decide where to execute the new instance. This might well be on the same processor as the creator, or it could be any processor which happened to be less-loaded.

It is only sensible, having introduced these machine values, to take full advantage of them, as we did with the message queues, and permit them to be interrogated, to determine which instances are on which machines. We should be able, therefore, to ask an instance value which machine it is executing on, and ask a machine value for the set of instances which it supports. We should also associate status values with machines, so that machines can go 'off-line', assuming the completed state, or if they fail beyond recovery then they would take the aborted state, possibly

even displaying some exceptional condition. This permits machines, and the activities running on them, to be handled in a single uniform, and thoroughly natural form. This is discussed in greater detail elsewhere [H7].

The instance value is, therefore, not simply a numerical activation index, it is an abstract representation of a particular entity running on a particular machine. Messages directed towards it are automatically routed across the hardware channels, between processors, by the implementation of the system. The interpreter on any given processor, scheduling its local instances, detects transactions for instances not resident on its own machine, and transfers the messages to the appropriate hardware, where another interpreter takes it and delivers it to its target's queue. This is similar to the way i/o is incorporated into the message passing system, by having the interpreter detect the instance value associated with the terminal.

7.0 Conclusions.

We can summarise the major failings of previous attempts to incorporate concurrency into programming languages as follows:

1) In such languages processes were included as denotations, names not values, so making it difficult to identify particular members of the running system,

2) they supported, and often relied upon, shared data for communication. This required that they impose sequentiality in order to implement concurrency! In addition,

3) nearly all involved attempts to 'extend' existing languages, designed to different requirements. This fact was often noted because

4) most of those previous languages contained concepts antagonistic to the incorporation of concurrency. In particular

5) they were languages in which the store was dominant. That is, the data structures relied upon pointers and selective updating for their usefulness.

It is interesting to note that thus far there have been no attempts (?) to include concurrency in the applicative languages. This is rather puzzling, as such languages, with referential transparency established, offer a much more hospitable base on which to build a system of interacting concurrent processes. Perhaps the reason for this obsession with what we might call the algol-like languages (leaving aside the attempts to graft concurrent features onto Fortran), is more rooted in tradition than in good sense.

The proposals put forward herein are also distinctly algol-like. This is so largely because of familiarity with the store-oriented languages, together with more than a touch of reticence concerning functional languages. Nevertheless careful selection of concepts has enabled us to avoid the abovementioned pitfalls.

7.1 What Has Been Achieved?

Firstly, we have avoided the trap of grafting on 'extensions' to an existing language. This has enabled us to step back so as to examine dispassionately some of the problems encountered in earlier efforts by others, and to deduce the underlying cause of those problems. The presence of the store in the realm of data values and abstractions exploiting selective updating are prime examples. Thus we have tended to remove the source of the problem rather than to live with it and hide its effects in return for some 'essential' feature. To replace the traditional side-effect oriented arrays and such we adopted lists and sets, which have useful abstractions without the need for selective updating, even if implemented as pointers. This switch of data structure is made simply to reinforce our awareness of the root of the problem.

Secondly where previous attempts were lacking we have tried to extend all concepts to fulfil the same general principles. In particular processes, or instances and definitions as we call them here, have been elevated to 'first class citizenship', and all values can be passed as

messages.

Since several message-oriented systems have already been implemented it is fair to ask how that proposed here is an improvement on such systems. Message passing systems can be categorised in various ways. Most use fixed sized buffers, into which the messages must be squeezed. Naturally once a buffer size is specified one has the problem of messages which are larger than that buffer size. Here all messages are the same size because each is a value, which can be an arbitrarily complex data structure.

In a single address space this can be highly efficient for 'large messages' because data structures are passed as pointers. If messages are to be passed between processors, to instances on other machines, then direct i/o is performed, and the operation is correspondingly slower, as copying would be required for large messages. In this multi-processor system, however, all the buffering and i/o transfers would be undertaken by the interpreters on the respective machines. All that the instances (on separate processors) would do is pass a message as normal. Thus instance values are not simply numerical values (as in most other systems), they are more abstract, embodying the identification of their processor (ie its machine value) and their relationship to it. In a system running (possibly on a variety of machines) the various interpreters, one per machine, manage all the detailed buffering of physical i/o transfers, to provide the level of abstraction on which the instances operate. The language, therefore, is free of such

matters, ie it is truly high-level.

One can also categorise message systems by their type-checking mechanisms. Most are either typeless, as in TRIPOS [R6], or heavily type-checked, as in Thoth [G1]. To permit polymorphism the heavily checked systems introduce extraneous syntactic constructs, such as the modified Pascal variant-records in Thoth. We take the view that whilst types and type-checking are important, static checks are too restrictive, so we implement a dynamic type-checking mechanism via a tagged architecture. This ensures complete freedom of action, yet it is fully protected.

The message system is inherently asynchronous, permitting each instance to run at its optimal rate (processor sharing aside). Instances only need to wait when they request inputs which have not yet been sent. This involves queues attached to instances, in which as yet unwanted messages accumulate. Because queues reside in the same heap-based address space as the rest of the system they can grow arbitrarily long. These queues implement the synchronisation implicit in sending and receiving messages. Our use of instance values both for receiving and sending messages, together with the no-wait send and queues contrasts with the blocking sends of CSP [H13], and the need for special reply-messages provided by the Thoth system. Storing messages in queues also removes the need to reschedule every time some process tries to send or to receive a message, unlike the fully synchronous proposals of Hoare's CSP system.

Separating out internal system scheduling from localised user-oriented scheduling requirements is considered to be particularly well implemented, as it is based upon the identities of the participants and their activities, not on some arbitrary priority scheme.

Then we brought everything 'out into the open' because all of the information held within the system about the state of the instances and their input queues, is available by way of interrogative operators. This permits the user to write inquisitive programs which examine their environment before acting.

As demonstrated the current system offers considerable scope for nondeterminism, because an instance can evaluate its workload (in its queue) and schedule its own activities accordingly. It can always identify the senders of its messages, and can read the messages in as necessary. It can respond at any time, without being delayed. This system is extremely orthogonal, unlike its predecessors. Indeed, many of the Thoth restrictions and 'extra' concepts are avoided here, and the dynamic type-checking, instance values and asynchronism makes our system much easier to use.

In short, we have here an interactive facility which is both a programming language and its own 'shell' [R8], in which concurrency is a fundamental concept, not an add-on feature, and a completely uniform generalised message system, passing arbitrarily complex values, is the only means of communication.

It is quite clear that these communication primitives provide considerable freedom of action. All of the constructions embody the essence of the desired activity without involving the user in the details of the interactions. It is felt that it is only by abstracting the essential purpose of a construct, and by pitching it at such a high level of participation, that the immensely complicated interactions taking place in a system of concurrent instances can be reduced to an intellectually manageable form.

7.2 Is This Sufficient?

We have argued that the approach adopted herein avoids, rather than actually solves, many of the trickiest problems faced by previous attempts to incorporate parallelism into a programming language. We developed the notion of passing messages and found it to be an extremely uniform and flexible mechanism. But is it sufficient?

Well, there is always a tremendous temptation to regress, to reinstate some degree of selective updating within the structured objects. In short there is a lingering desire to bring the store back into play. As things stand the store is relatively inconspicuous, being visible only in the case of individually named variable locations. We could do away with the store altogether, bind names directly to values, and replace iteration by recursion to control evaluation. In the longer term this might be the best way forward, but such programming techniques are relatively little understood by

the computing world at large. Most computational problems are stated in terms of, and solved by exploiting, an updatable store. Thirty years of hard learned programming technique and style presents a tremendous inertia, and poses a formidable obstacle to any radically different approach. The applicative programming languages have been around for over twenty years, but have made relatively few inroads into the establishment.

The domination of our programming languages by the store is founded in a desire, sometimes almost obsessive, to run our programs efficiently on our machines. Similarly new machines are generally nothing more than faster versions of the previous generation, because that is the design which best supports the most widely used languages. It is a vicious circle. Unfortunately most attempts to incorporate concurrency have been based on languages thereafter intended for 'systems programming', where their desire for direct access to the underlying sequential machine conflicts most with their planned parallelism. The two levels of abstraction have very little in common. It is hardly surprising, therefore, that all such efforts have run into difficulties almost immediately.

It is only natural to wonder whether we could write an operating system more or less easily in such a new language? To contemplate this would, however, be a futile exercise, because the implementation of such a language is itself a complete system. It is visible as an incremental compiler. As a built-in mechanism it provides a polymorphic message

system. There is therefore no reason to expect that its basic mechanisms should then be re-implementable in that language! The language depends on the implementation for its effect whilst the implementation depends upon the the expressivity of the language - it is clearly a symbiotic relationship.

Most previous languages for writing systems of processes have deliberately provided only very primitive operations, and their data structures have extended right down into the realms of the implementation (consider the otherwise unnecessary access to bits and device registers of Modula for example). Inherent in such languages has been the 'bootstrap syndrome' which demands that any language be so widely based that it can write its own implementation. In the world of concurrency we should give more than a passing thought to the old adage regarding the performance of the 'jack of all trades'. The two extremes, of low-level implementation and the abstractions of high-level systems of concurrent systems, are at opposite ends of the language spectrum, and since there are so many areas of conflict it is not really advisable to try to merge them.

The language presented here is, therefore, not intended for writing operating systems and the like, because its implementation is itself a complete system. That implementation is written in a language more suited to this task, in which there is no trace of parallelism. It is doubtful whether we will ever see a truly general purpose programming language!

Is this sufficient? There are many computing problems, eg large data bases, in which concurrency is desirable, but the nature of a solution, within our language, is not immediately obvious. The language should suggest a solution! We may well find an indication of a solution in the multi-computer systems of tomorrow. In such a system a data base would inevitably be distributed, with separate processes managing different catalogues of information. The structure of the file system would therefore have to be changed from a central memory threaded with location-links to a distributed memory threaded with process(or) linkage. Information would then have to be extracted by sending a message to its managing process. As this would be running in parallel with its enquirers together with its companions in other sections, it could service its requests in whatever order it prefers, whilst simultaneously sending out other requests for information to its companions to satisfy its own needs. The data base is therefore nothing more that a distributed system of concurrent processes. Any restriction on either the mode of communication or the material permitted as a message will immediately complicate such a dynamically evolving network. An asynchronous message system such as that proposed here might prove to be ideal, since it enhances parallelism, and the fact that it is polymorphic means that there are no restrictions on the content of messages.

The greatest obstacle in the way of better solutions to such massive systems is inherent in our traditional approach to

the problem, because all of our traditional techniques are geared towards "efficient" solutions, and these are always thought of in terms of central inter-related storage.

There are two outstanding requirements which must be fulfilled before the benefits become evident. Firstly, that the design of our language continues, and hopefully improves. Secondly there is an urgent need for programming techniques to exploit it. Since the field of the applicative languages is not bound by the limitations of present-day machines, this is probably a fertile area in which to search.

7.3 The State Of The Art!

In this thesis we have tried to argue the case that there was something drastically wrong with the early attempts to build concurrency into a programming language. Certainly this is all too obvious in the case of PL/I [N3], and Algol-68 [W13]. These particular languages added it to their list of features almost before it was realised that it involved a whole new approach to programming. Both languages also include free pointers - a sure recipe for disaster.

One might well expect therefore that the concurrent languages of later years had learned from this. Apparently not. Wirth introduced Modula [W16,W20], building into it 'processes' which interact by fairly well defined interfaces based on the monitor of Hoare and Hansen. Wirth deliberately designed Modula as a static-storage system, there were no explicit pointers, and compound objects had restricted

rights. This situation did not last for long, however. Wand [W2] proposed that numerous 'extensions' be grafted on, including, almost incredibly, pointers. Naturally he also suggested a list of rules to try to restrict their use. This is precisely what is not needed, even in Modula.

Modula is not particularly offensive, it is selected simply because it is typical. Perhaps the worst offenders are those who unilaterally 'extend' an existing language out of all recognition, while retaining its name. In this category Pascal, also by Wirth [W17], has probably suffered most. Of late, apart from Modula itself, there have been the processes, monitors and classes of Hansen's Concurrent Pascal [B19,B22], the envelopes and modules of Welsh & McKeag's Pascal-Plus [W6,W7], and the assorted extras in CCNPASCAL [N1]. There are even secondary extensions, with Silberschatz et al enhancing Concurrent Pascal to permit dynamic structures [S9], and later [S6] a 'capability'-oriented mechanism for specifying access-rights and exceptional conditions.

Most intriguing of all, however, is the Pascal-based language described by Andrews and McGraw [A3] from Cornell University. They present what they call "a unified set of language features" designed to control process interactions. Of four possible developments which they considered, they incorporated three, all in the same language! They have 'resources', which they say is "an extension of Hoare's monitor", 'protected variables' and 'shared reentrant procedures' (which are naturally second class objects, made

visible by violating scope rules). Interestingly enough their other option, considered first and rejected, was message passing. This was rejected "because policy decisions [concerning what could be passed and how] would then have to be made [by the language designer]." Obviously this did not fit into their "unified set of language features". Instead their users are presented with a bewildering set of language features ranging from Dijkstra's P/V operations [D9] and Hansen's condition variables [B25], to a form of Habermann's path expressions [C1,H1], together with so-called 'capabilities' [F1,M9]. As if this was not unified enough they also brandish global variables, call by-value and call by-reference and pointers! They further declare that it is essential that everything be verifiable at compile time. At the end of their paper there is a discussion on the possibilities for detecting deadlock in a system of such interactions!

Against this background it is hardly surprising that the latest international collaborative effort, the DOD's Ada project [see B5 or I2 for a summary], is Pascal-based, with pointers (access types) and shared variables (which are "not recommended" as a means of parallel interaction), together with every other 'feature' that might possibly be of some use. For various critical comments on the Ada tasking facilities the reader is referred to [B14], E[6], H[3], [J4], [S15] or [W8], and to [M1] for a general evaluation of its real time aspects.

Standing out in this wilderness is Hoare's CSP proposal

[H13]. This, as was indicated earlier, is probably the most radical rethink of the concurrency problem so far published. Apart from the fact that his system is fully synchronised, with either partner waiting until the other is ready to interact, Hoare's proposals are different from those developed herein mainly in that we treat processes as values, whereas he regards them as names. The main limitation, noted by Hoare himself (see his line-printer spooler example [H13]) stems directly from the fact that processes are names in CSP. The obvious solution therefore is to make processes values, as we have done here, but what do we see in the literature? We see an 'extension' to CSP by Silberschatz [S8] proposing that this particular problem be overcome by making processes communicate via ports, rather than directly with each other.

The state of this particular part of the art is an odd one indeed! It was the purpose of the work reported herein to reexamine this view, in an attempt to devise some truly unified set of language features capable of exploiting the multi-computer systems now becoming available. Our overriding concern, at every stage, was to keep it simple, while including only those concepts which were directly related and could be implemented uniformly. Complete uniformity is of paramount importance - any special case, no matter how trivial, serves only to undermine the generality of any mechanism. The message passing scheme presented here, unlike others before it, is completely uniform. It is a tool. With this tool one can express arbitrarily complex

interactions. As a mechanism for programming concurrency it is therefore a success.

8.Ø The Design Of Protocol.

What constitutes a 'well designed' programming language? This is a contentious issue with, it seems, as many views as there are workers in the field. The methodology adopted here is relatively new, as a basis for the process of designing a language, although it has been frequently employed as a means of reviewing the failings of existing languages.

There are several important principles, forming a kernel of good design, which if adhered to rigidly, provide essential language properties. These are due originally to Landin [L3,L4,L5], Reynolds [R4] and Strachey [S11,S12,S13], but are frequently attributed to Tennent [T1,T2] because of his strong criticisms of Pascal. Morrison [M6] has recently demonstrated that these principles of design can lead to a language which is powerful and expressive precisely because of its simplicity, uniformity and lack of special cases.

The notion of a store is important, because the degree to which this infiltrates a language sets its overall characteristics. The concept of scope, first introduced by the designers of Algol-6Ø [N2], is perhaps the most significant feature of the languages in the Algol-family. Scope determines the lexicographical properties of user-defined symbols (literals, names etc) as defined by block structure. The designers of Algol-68 [W13] emphasised the need for orthogonal concepts, ie that the concepts bound together to form a language should interact uniformly in all cases, without peculiar conventions and restrictions to

control their combination. The principle of Completeness [R4] requires that values of all types be 'first class citizens' [S11,S12] of the language. That is that all values can be passed around, or assigned or built into data structures with equal freedom. The Correspondence principle dictates that all user-defined names in a program should have the same potential attributes, ie names introduced as parameters, or components of data structures should have the same constancy options, modes of initialisation etc, as names introduced by in-line declarations.

Underlying this there is a sound theoretical framework, the denotational semantics [see G5,S5,S13 or T2 for details], which provides a framework on which a language may be constructed. This work has the double advantage that it provides both a methodology for designing a language and a means of rigorously defining it.

The design of Protocol, as a programming language, has been based upon this -kernel of guidelines. The design process itself is therefore not original, but the language produced by it is considered to be quite new in several respects. Protocol is a prototype for a highly parallel interactive command language. It differs from most command, or job-control languages, because it's structures are those of a full-grown programming language. It is usable at a command level because it is interactive. It is block structured, has quite a few data types, including composite types such as lists and sets, and is completely type-checked at run time. Protocol is really a complete programming environment, with

a means of dynamically defining, editing and running library files (definitions and instances thereof).

8.1 Names & Locations

Declarations introduce new names into the programming environment and request space in which to store some value. A name (as opposed to a literal, see the ontology in the frontispiece) denotes a location in the computer memory, and locations hold values. All declarations are initialised. Names for data objects may be associated with either constants or variables. The only difference is that a variable can be updated later in a program, whereas constants cannot. A constant is therefore regarded as a location which, once initialised, cannot be updated. Note that a constant is distinct from a literal, as it can be dynamically initialised from any appropriately typed statement, as demonstrated by Gunn and Morrison [G8]. The compiler reports attempts to update constants.

When discussing concrete syntax we shall employ the meta-syntax suggested by Wirth [W19], where {} implies zero or more occurences, [] implies optionality and () requires exactly one alternative. The standard BNF [N2] symbol for enumerating alternative syntactic units, ie the | meta-symbol, will also be used. The productions employ the = meta-symbol for definition.

The syntax of a declaration includes:

"let" exp init.symbol [type] name { "," [type] name }

where an init.symbol is of the form "->" for a variable, and "=>" for a constant. As we shall see later, in relation to 'chaining' messages across instances, there is another means of producing the initialising value, apart from the simple expression.

A name can be any combination of letters, dots and digits, so long as the first is a letter. By initialising all data objects as they are declared the common problem of uninitialised variables is banished.

In the syntax given above it is possible to have a number of names on the right hand side, each with some type restriction specification. Each will then be declared with that restriction, if any, together with the constancy specified by the initialising symbol. In such a case the value of the initialising expression must be a list, and that list must contain at least the desired number of components. The elements of the list are then 'stripped off' and assigned to consecutive names. This is really a form of 'syntactic sugar' for a series of head and tail operations on a diminishing list, initialising a series of declarations. Naturally there is also a similarly sugared form of assignment, as we shall see below.

8.2 Degrees Of Constancy.

The syntax for a declaration holds four variants, with differing degrees of constancy. Constancy is a natural form of value protection. It is often the case that most 'variables' are never changed. To guarantee that such a

value is never corrupted it can be declared to be constant, with "=>".

This polarisation, variability and constancy, provides a very useful facility, but it does not fully exploit the dynamic type checking of Protocol. It is quite reasonable to require that a certain location be limited to a certain type, yet remain a variable over the range of values of that type. It is also desirable that if an operation makes sense at run time then it should be permitted to take place. This requires that it be permissible to assign differently typed values to a location at different places in the program. Protocol readily permits a location to be initialised to different values, via messages, on different incarnations of a routine. It is natural therefore to permit the user to specify exactly what degree of protection or, conversely, degree of freedom, a location is to have [G9].

For this purpose it is possible to declare a name to denote a location which is to be completely general purpose:

```
let Ø -> x
```

which can take any type and value at any time and place, but is initially an integer with the value zero; or it can be a variable integer value

```
let Ø -> int x
```

initialised to zero. Alternatively, it could be a constant. There are two forms of constant, as with variability. The form:

```
let Ø => int x
```

would initialise x to be a fixed integer constant with the value zero. This can never be altered. It is a true constant.

The final form:

```
let Ø => x
```

is also a constant value, but because the type was not actually fixed it would be permissible to initialise it with a value of any type.

8.3 Sequences & Scope Rules.

A sequence is the unit of scope. It comprises a series of declarations and statements arranged in any order. The order of execution is sequential, top-to-bottom, according to the state of the various selective and repetitive statements making it up.

Brackets "{" and "}" may be used at any time, to form a new sequence, with its own scope, or to override the default precedence of the operators in an expression.

A sequence may, or may not, contain local declarations. Of the statements all but the last must be void (declarations are considered to be void statements). If the last statement is also void then the entire sequence is void, and returns no value, otherwise the sequence forms an expression with the type associated with the value produced by its final

statement.

In accordance with orthogonality, any statement, no matter how complex it is, can appear anywhere a value of the same type is required, so even a literal enclosed in a {}-pair is a sequence (not much of a sequence!), with that value, and can be employed anywhere that that value is valid.

As we stated earlier, all names for data objects must be introduced before they are used, by declarations, and all declarations must be initialised, so a sequence is made up of interspersed declarations and statements. This is intuitively more natural than the mass of name declarations found at the top of the more traditional algol-like blocks.

(declaration | statement) [separator sequence]

It is important to note that, since names for data objects must be initialised before use, the scope of a name starts immediately after its initialising statement, and so is not defined within it.

8.4 On Control Structures.

In order to express an algorithm we need some control structures. Since the store is present in the language we will exploit it to efficiently implement iterative constructs.

Most high level programming languages popular today provide a fairly orthogonal set of control structures. We provide several syntactically distinct forms of loop-control, rather

than a single low level loop or label and goto facility. Some have argued against redundant control structures, either on the grounds of compiler efficiency, eg Wirth's comments on the design of Modula [W16], or lack of understanding [S1]. We offer various, slighly different forms, because each expresses one form of algorithm more clearly than the others. For example the while-do loop and the repeat-while appear, together with the repeat-while-do form. These forms can, despite Wirth's claims for simple compilers, result in more efficient code if there is a re-initialisation phase in a loop which duplicates the initial entry code, because the repeat-while-do construct can be used to remove duplicate code.

8.5 Statements & Expressions.

The statement is the basic mechanism for coding an algorithm, and as this is what programming is all about, the fewer and more general the rules the better. There are several different statements. All may be considered in a frame of type rules, by giving a traditional statement a type void, and expression statements a type associated with the value that they produce. This slight extension of the notion of type allows us to discuss the various statements in a unified frame of reference. Type void is a convenience only, it has no tangible value, and does not appear explicitly.

8.5.1 Assignment.

Perhaps the most fundamental statement is the assignment. It certainly has the greatest impact on the underlying semantic model, because without it there would be no need for the notion of a location. The assignment statement, though unobtrusive, is of great significance. This takes the form of:

expression "->" name

where the value produced by the re-initialising expression must concur with any type restrictions introduced at its declaration. It is possible, following the simultaneous declaration, to assign the leading components of a list, to given names, by:

expression "->" name { "," name }

where the expression must yield a list. The appropriate number of components will be stripped off and assigned in order to the named variables. The list must, obviously be at least long enough, and is left unchanged by the operation. One could regard this construction as the pairing of the elements of the actual list value with the names enumerated in the assignment, and the updating of each in turn. The remaining tail of the list, if present, could be thought of as being 'assigned' to the null element at the end of the 'list' of names, which, as expected, means that its elements are inaccessible.

It is obvious therefore that this provides a natural

swapping construction:

y,x -> x,y

which does not require auxiliary variables. This is better illustrated in:

z,y,x -> x,z,y

which is terse but clear. The contents of "z" are placed in the location named "x", then those of "y" are placed in "z" and finally the contents of "x", as it originally was, are placed in "y". All these transfers are performed by a single operation which uses the stack for temporary storage.

Assignment has the effect of updating the contents of the location denoted by the name, replacing its previous contents, which are then lost.

8.5.2 Conditionals.

There are several selective and conditional statements. Firstly, a simple conditional, in which a boolean value is evaluated from any appropriately typed expression, and is used to decide whether to execute the attached statement, or to skip it, and go on to the next statement. This is quite traditional, and is considered to be void. That is, it can never return a value as this would cause problems, perhaps producing a value if the test was true, none if it was false. It takes the form:

"if" expression "do" statement

The second version is a two-armed selection, with the appropriate arm being decided by the boolean value evaluated in the initial test. This takes the form:

"if" expression "then" statement "else" statement

in which the first is the boolean expression, and the other two must either both be void or both produce a value of some type. If values are produced they need not be of the same type. It is illegal to have a void and a value-producing branch together. The compiler flags void and typed arm combinations.

If the branches are void then the entire statement is void, otherwise it forms an expression, the type being determined dynamically from the particular branch executed as a result of the test.

With these two forms there is no 'dangling else' problem, and there is no need to incorporate explicitly matching terminator symbols, as all compound statements are enclosed in {}-pairs, forming sequences.

8.5.3 Iteration.

There are various forms of looping statement, for repetitive algorithms. The for-statement, another old favourite of the algol family, takes the form:

"for" [name "=" expression "to"] expression
["by" expression] "do" statement

and is always void since it produces no value.

The newly named location is called the control constant. It is redeclared and re-initialised every time the body is executed. The scope of the control constant starts as soon as it is initialised, so it is available to the increment or the limit, and it extends to the end of the main statement. This lets us perform a statement a given number of times whatever its initial value. The control constant is made a constant because it is bad practice to adjust it within the main statement. If the control constant is named its initial value must be given, otherwise it is made equal to unity. The increment if given (unity is default), and the limit, are evaluated once before the body is entered. The limit must always be given. If the initial value of the control constant is outwith the limit then the body is never entered. The repeated execution of the body ceases as soon as the value of the control constant passes the limit. More precisely, when the control constant is greater than the limit (if the limit is positive) or less than the limit (if it is negative). Notice that the test is not for equality of constant and limit, as the step could cross its value without actually matching the limit.

There are three forms of while-statement, providing the test at the top, in the middle and at the end of the body. The simplest is:

"while" expression "do" statement

and it is void. It has the test at the top, so the loop can be executed zero or more times. The next form is:

"repeat" statement "while" expression

which is also void, and it has its test at the end. This is executed at least once. The final form is:

"repeat" statement "while" expression "do" statement

which is again void, here with the test in the middle. The first part is executed at least once, and the final part depends on the test, as do further loops around the initial statement. Note that in the latter two, the scopes of the repeat part and while part are distinct, and so it is necessary to declare common, but otherwise private, data objects outside of the entire statement. It would, perhaps, be more aesthetic to merge the same scope over both arms of the broken loop. Such a change, with the broken syntactic form, would require a complication of the otherwise extremely simple unit of scope (the sequence).

8.5.4 Expressions And Operators.

The other forms of construction are traditionally called expressions, in that they always produce values.

The simplest is the bracketed statement, or sequence, which as we have just seen can lead to a value being produced.

Next we might consider the general form:

value [binary.op expression]

The syntax for a value can be found in the Concrete Syntax elaborated below. The various operators are applied to

their operands as subexpressions according to their default precedence of application. A (partial) list of operators contains:

```
 unary:    +  -  ~  hd  tl  sizeof typeof  endof

binary:    +  -  *  div  rem  and  or
           =  ~=  <=  >=  <  >  ++  ,
           |  <>  ><  has  &  of  <<  >>
```

and their precedence is as follows:

```
    <<  >>
    *  div  rem  ++  has  <>  ><  &  of
    +  -
    ~
    =  ~=  <  >  <=  >=
    and
    or
    ,  |
```

The highest precedences are the unary operators, which bind closest inwards; of the binary operators, the top of the table is higher precedence. For any level in the table, the binary operators have equal precedence, and are applied left to right in expression evaluation, but the user can format his own subexpressions by bracketing with {}-pairs.

Note that string subscripting is of higher precedence than concatenation, by "++". The syntax for this is:

```
value "[" expression "by" expression "]"
```

such that on the left we have the string value being subscripted, and the specification for its substring to its right. The two expressions within the "[" and "]" brackets describe the substring required. These are integer values, the first being the starting character position, the second the number of characters in the substring, starting with that character. Strings start at character position one. The system checks that the substring is sensible, that is that the bounds are positive, and that they indicate a substring entirely within the base string. A substring length of zero results in a null string. It is not sensible to specify negative string lengths.

Boolean expression evaluation is non-strict, that is in the left to right evaluation the process stops once the final result is found, rather than continuing with, in this case, superfluous evaluation of following sub-expressions.

We can summarise the operators directly related to the instances and their message system as follows:

survey [DEFINITION] gives the set of instances currently known to the system, possibly restricted to those executing the specified definition value.

stateof INSTANCE gives the status value currently associated with the specified instance value.

holdup INSTANCE. If the specified instance has the waiting status this will give the instance value, or class of instances, from which it is expecting to receive a message.

If the specified instance is not waiting then this operation will cause the investigator to abort with the Delay.Err exception value.

faultof INSTANCE gives the exception value raised by the specified instance as it failed. If that instance has not in fact aborted then this operation will cause the investigator to abort with the Abort.Err exception value.

classof INSTANCE gives the definition value associated with the specified instance. This functions even if the specified instance has ceased to operate.

census [INSTANCE] gives the set of instances which form the sources of the messages currently in the queue of the specified instance value. If none is specified then the operation is performed on the current instance.

message [INSTANCE | DEFINITION] reports whether there are any messages in the current instance's queue from a particular source, or class of sources. If none is specified then it reports whether there are any messages at all.

Of the data structures, lists can be specified by separating values by commas "," whereas sets can be enumerated by bars "|". In the case of sets duplicates are stripped out and the ordering of the enumeration is not necessarily preserved, since sets are inherently unordered. All sets start off as the empty set value, and lists always start with an implicit tail of null.

The set operators are:

SET "has" T

which indicates whether the specified value, of any type, is in the given set value. Actually the "has" operator is more powerful than it seems, because it can be applied to either a set or a list, to find out whether a given value is within the data structure, and it can also be used on a queue of messages (by giving it an instance value) to see if a particular value (from whatever source) has been sent.

The following operator:

SET "<>" SET

T "<>" SET

SET "<>" T

T "<>" T

has four variations. If both operands are set values it forms the union of the two sets, if only one was a set then it augments it with the other value, and if neither was actually a set then it forms a set and adds these two values to it. Like operations on strings the operands are unaltered by such 'changes', as new sets are created whenever they are updated.

The intersection operator is:

SET "><" SET

and it produces the set of values which are in both given sets. This, together with the equality operator, which is defined for all types, permits a 'subset' operation to be

simulated, viz:

```
let { A >< B } = B  ->  B.is.subset.of.A
```

as expected.

Since it is not possible to access particular elements of a set, there is an iterative construct which permits a given statement to be applied to all its elements. This takes the form of:

"forall" name "in" SET "do" statement

This makes all elements available, but the order in which they are presented is unpredictable, as it depends upon the internal representation of the 'unordered' set.

There are the usual list operations for head "hd" and tail "tl", and they function for all lists except the null list, which has no internal structure. Like sets, new lists are formed whenever an 'update' operation is performed. This, as explained earlier, is essential if data structures are to be passed freely within a system of concurrent instances. The updating operations are:

LIST "&" LIST

T "&" LIST

LIST "&" T

T "&" T

and like the union "<>" for sets, this joins lists together, prefixes or postfixes them with a given value, or forms a list out of two new values. Where it is necessary to prefix

a list with a value, possibly another list, without actually joining such lists together, should they both be lists, the following operator suffices:

T ">>" LIST

Similarly:

LIST "<<" T

will append the value to the list. The difference between LIST "&" LIST and the form LIST "<<" LIST is that the case of "&" results in a new list made up of the elements of each component list joined together, whereas the "<<" or ">>" operations tack-on another element in which the extension forms a sublist. As with all "adjustments" to data structures the actual operands are copied before the "update" is performed.

Apart from a succession of hd and tl operations, the inner components of a list can be accessed randomly by:

INT "of" LIST

where the n'th component is extracted from the given list. Naturally the list must be at least that length!

The remaining operators, for the logicals, relational and arithmetic operations will not be elaborated upon here, as they are well known.

8.6 Abstraction.

Abstraction is the process of hiding details of

implementation and representation so that the essential purpose of a mechanism is laid bare. In programming abstraction is a major tool because it permits complex mechanisms to be broken down into more, individually simpler mechanisms.

Most programming languages popular today provide procedures and/or functions as the means of abstraction (over statements and expressions respectively). A similar abstraction, the definition, is available in Protocol. This enables a segment of code to be stored in the library. Each entry in the library is identified by a unique user-coined symbol - a literal of type definition.

Because the store has been constrained to individually named locations, and definitions cannot have free variables, they cannot share data, nor can instances (executing definitions) influence each other by 'side-effects'. The concept of the message supports all communication requirements, both internal and with the outside world.

There are no procedures, functions nor parameters as such, because the instances, passing each other messages, communicate at a more fundamental level. A procedure-call corresponds to invoking an instance of some definition (corresponding to the procedure body) and any parameters are equivalent to the sending of an initialising message. Similarly a function-call, taking parameters and returning a result, corresponds to an instance invoked with an initial message, followed by an immediate (from the viewpoint of the

caller) reply message.

We see therefore, that the distinct concepts of procedures and functions are merely special cases of what we term instances, with message passing as the mode of interaction. Furthermore, the concurrency of Protocol, together with the ability to send and read-in messages at any point in a definition, permit instances to interact with each other during their execution.

The abstraction mechanism of Protocol is thus both simpler and more powerful that the traditional procedural mechanisms. The concept of the message is more general than the parameter because, when combined with a queuing mechanism, it can support asynchronous interaction.

We wish to have as few concepts in out language as possible (although each is exploited thoroughly and uniformly) so we have not included procedures nor functions, the instance with its message system being sufficient to fulfil these roles. If we had included either, to give local abstractions, then, if we were to implement them in accordance with the principles of Correspondence and Completeness, they would have to be provided as values in their own right, and so they could be passed as messages, and executed by other instances within their creators local environment, thus re-introducing the problem of shared data which we have taken such pains to eradicate! It is significant, therefore, that Hansen's new language for concurrency, Edison [B26] which depends upon procedures for

its communication mechanism, includes them only as names, not as values, so complicating its ontology.

8.7 The Syntax Of Message Passing.

As outlined earlier the message system is dynamic because it evaluates the instances taking part in each transaction. To send a message we construct:

INSTANCE "_" T

where the message value, of any type, is passed to the instance evaluated on the left of the "_" operator. Similarly, to request that the next message to arrive from a particular source be read-in we write:

"(" INSTANCE ")"

so forming an expression, with the type of the acquired message value. Normally this value will then be assigned to a variable, or used to initialise a newly declared name. For example:

```
let ( input _ "Enter Maximum ?" ) -> Max.Value
```

will declare "Max.Value", make it a variable, and initialise it with the value acquired from the instance associated with "input" (here a literal of type instance - the standard inlet), having first sent it the prompt "Enter Maximum ?".

Consider the following "echo" definition:

```
let "Start'n" -> x
repeat world _ typeof x, x, "'n"
```

```
while x ~= -1
do ( world _ ">> " ) -> x
```

which loops until it receives the value -1, prompting with ">>" before reading in some value, the type and value of which it then prints out. Note in passing that 'n is the newline character and 't is the tab character. This is quite succinct, but as we shall now discover, there is also a 'sugared' mechanism for pipelining messages across members of the system, which directly expresses a complex series of interactions.

8.8 Chaining.

The concrete syntax chosen for the message sending and requesting operations is deliberately terse, and it lends itself well to the 'one-liner' syndrome. Each such construction is, however, only capable of expressing a single interaction, be it a simple send, a request for a message or a request associated with a prompt message.

By slightly extending the concrete syntax, as we do here with the ":" operator, it is possible to combine interactions into a pipelining construct. We take advantage of this syntactic simplicity to "chain" together several interactions. In complex cases there may be a whole series of 'filter' instances, down which values have to pass before being written out. Consider, for example, some data which is to be analysed, tabulated, built into columns and then paged before output, this might be processed:

```
analyse _ data : tab : col _ 2 : page _ "ABC",1 : world
```

where the ":" indicates that the activity on its immediate left will produce a message, which it waits for, and then it feeds it over to the next activity.

Notice that the messages 'flow' from left to right, eventually being output to the terminal, and that at each stage additional messages can be tagged on to the 'passing' message. The messages need not be simple values, they can be lists, as can be seen for "page", and any message tagged onto the flow is automatically bound into a list with that value before it enters the next stage. We call this chaining a message. In an interactive programming environment, as a 'shell' [R8] mechanism, this is extremely powerful, exploiting as it does the generality of the message passing system.

8.9 Concrete Syntax.

```
sequence =
 ( declaration | stat ) [ sep sequence ]

stat =
   "if" exp ( "do" stat | "then" stat "else" stat )
 | "for" [ name "=" exp "to" ] exp [ "by" exp ] "do" stat
 | "repeat" stat "while" exp [ "do" stat ]
 | "while" exp "do" stat
 | "forall" name "in" exp "do" stat
 | value "_" value
 | chain ":" ( value | "->" names )
 | exp [ "->" names ]

chain =
 interaction { ":" interaction }

interaction =
 [ "first" | "latest" ] value [ "_" exp ]

exp =
   "if" exp "then" exp "else" exp
 | value [ binary.op exp ]

value =
 { unary.op } ( name
```

```
              | literal
              | "{" [ sequence ] "}"
              | "(" interaction ")"
              | "var" name ) { "[" exp "by" exp "]" }

declaration =
 "let" ( exp | chain ":" )
          ( "->" | "=>" )
              [ type ] name { "," [ type ] name }

unary.op =
 "+" | "-" | "~" | "hd" | "tl" | "typeof" | "endof" | "new" |
 "sizeof" | "classof" | "stateof" | "holdup" | "faultof" |
 "message" | "survey"

binary.op =
 "+" | "-" | "*" | "div" | "rem" | "and" | "or" | "++" |
 "=" | "~=" | "<" | "<=" | ">" | ">=" | "is" | "," |
 "<>" | "><" | "|" | "has" | "&" | "of" | ">>" | "<<"

literal =
 digit { digit } | defined.symbol | string.literal

names =
 name [ "," names ]

name =
 letter { letter | digit | "." }
```

```
type =
 "int" | "bool" | "site" | "set" | "list" | "identity" |
 "library" | "string" | "type" | "status" | "exception"

sep =
 ";" | <newline>
```

8.10 Type Matching Rules.

```
"{" [ VOID ] "}" => VOID
"{" [ VOID sep ] T "}" => T
"if" BOOL "do" VOID => VOID
"if" BOOL "then" T "else" T => T
"for" [ name "=" INT "to" ] INT [ "by" INT ] "do" VOID => VOID
"repeat" VOID "while" BOOL [ "do" VOID ] => VOID
"while" BOOL "do" VOID => VOID
T "->" name => VOID
LIST "->" name [ "," names ] => VOID
( "first" | "latest" ) [ INSTANCE ] => INSTANCE
"message" [ INSTANCE | DEFINITION ]  => BOOL
"new" DEFINITION => INSTANCE
"let" T ( "->" | "=>" ) [ TYPE ] name-T { [ TYPE ] name-T } => VOID
"{" T "}" => T
STRING "[" INT "by" INT "]" => STRING
T-literal => T
INSTANCE "is" DEFINITION => BOOL
T "<>" T => SET
INT "of" LIST => T
T "&" T => LIST
LIST "<<" T => LIST
T ">>" LIST => LIST
SET "><" SET => SET
( SET | LIST | INSTANCE ) "has" T => BOOL
"forall" name "in" SET "do" VOID => VOID
INSTANCE "_" T  => VOID
"(" INSTANCE [ "_" T ] ")" => T
```

```
INSTANCE ["_" T ] { ":" INSTANCE ["_" T ] } ":" => T
INSTANCE ["_" T ] { ":" INSTANCE ["_" T ] } ":" INSTANCE => VOID
"endof" LIST => BOOL
"typeof" T => TYPE
"sizeof" ( LIST | SET | STRING ) => INT
( "+" | "-" ) INT => INT
"~" BOOL => BOOL
STRING "++" STRING => STRING
INT ( "+" | "-" | "*" | "div" | "rem" ) INT => INT
INT  ( "<" | "<=" | ">" | ">=" ) INT  => BOOL
BOOL ( "and" | "or" ) BOOL => BOOL
T ( "=" | "~=" ) T => BOOL
T "," T { "," T } => LIST
T "|" T { "|" T } => SET
"hd" LIST => T
"tl" LIST => LIST
"introduce" "_" name [ "," TYPE ] => VOID
"abort" "_" [ INSTANCE "," ] EXCEPTION => VOID
"define" "_" name stat => VOID
"stateof" INSTANCE => STATUS
"holdup" INSTANCE => [ INSTANCE | DEFINITION ]
"faultof" INSTANCE => EXCEPTION
"census" [ INSTANCE ] => SET (of INSTANCE)
"survey" [ DEFINITION ] => SET (of INSTANCE)
"classof" INSTANCE => DEFINITION
"var" name => BOOL
```

To be legal a program must satisfy the type matching rules

as well as be syntactically correct. The type rules are used to restrict the otherwise permissive syntax to a class of sensible operations. It might be run time before any particular type rule is matched.

The type T includes all types, except VOID which is merely a null-type used to provide a unified frame of reference in which to discuss type matching rules.

8.11 Literals.

Two instance literals:

```
input
output
```

form the default communications paths to the creator of an instance. These are initialised dynamically, on start up, and so can be different for separate incarnations of any given definition. The literal:

```
world
```

is fixed to the terminal, that is the on-line user, for direct communication with the external world. The original instance, the interactive shell-like instance, has its default paths linked to the terminal.

Each type has a literal representation, these are:

```
int
bool
string
list
status
exception
site
type
machine
set
instance
definition
```

The status literals are:

```
active
waiting
complete
aborted
```

The following literals are the predefined exception values:

```
Stack.O
Stack.U
Library.Err
Type.Match
Type.Wrong
Interaction
Null.List
Short.List
Camac.Err
World.Err
Delay.Err
Abort.Err
Digit.Err
Const.Err
Index.Err
Index.Neg
```

and of course new ones can be defined by the user at any time.

Further instance literals are:

```
introduce
define
abort
```

and these are explained in the text.

8.12 Further Examples.

The simple factorial evaluator, which sets up a series of instances, serves to illustrate the flavour of the message passing primitives.

```
define _ factorial
{
   let ( input ) => val

           output _ if val = 1 then 1
            else val * ( factorial _ val - 1 )
}
```

Note that "define" is a predefined constant holding an instance which, given a name, reads in a statement which it parses to form a definition which it then adds to the library.

Here we are simulating functions with individual instances. Although the instances of the factorial definition execute in parallel they do not actually interact, other than via the initial and final messages. This example, therefore, simply demonstrates the message system in a familiar situation, and shows how Protocol lends itself to recursive programming techniques.

Similarly, the famous Towers of Hanoi game can be solved recursively by:

```
define _ hanoi
{
   let ( input ) => N, a, b, c
   if N < 1 then output _ "Done" else
   {
      hanoi _ N - 1, a, c, b :
      world _ "Disk ", a, " to ", b, "'n"
      hanoi _ N - 1, c, b, a :
   }
}
```

which invokes a tree of instances, none of which actually interact other than on entry and exit. In this case individual instances wait for their subtrees to complete before continuing, so as to guarantee the ordering of the messages sent directly to the terminal (the world instance value).

We see therefore, that although these classic examples of recursive programming techniques can be expressed within the framework of a message passing parallel system, they do not exploit it. Even so, the message system permits the various subtrees of the hanoi execution sequence to synchronise correctly, without resorting to low-level semaphores.

The following example provides a little more insight into the advantages of the asynchronous message system. The problem involves finding all the permutations of characters in a string 'p', appended for output to the base string 's', for permutations of length 'n' characters.

```
define _ perm
{
   let ( input ) -> s,p,r
   if r = 0 then world _ s,"'n"
   else
   {
      for i = 1 to sizeof p do
      {
         let s ++ p[ i by 1 ] -> ns
         let sizeof p -> length
         let { if i+1 > length then i else i+1 } -> break
         let p[ 1 by i-1 ] ++ p[ break by length - i ] -> np
         perm _ ns, np, r-1
      }
   }
}
```

Notice here, that although it sets up a tree-like system of instances, any instance which yields a permutation writes it

out and terminates immediately. All others set up more instances to find more permutations before themselves terminating. The order of evaluation is immaterial, all that is required is a series of permutations, as opposed to the hanoi problem which needs to order its outputs. It is obvious therefore, that in the evaluation of permute, complete instances can be garbage-collected, to release space for further instances. On a simple procedural stack implementation this would not be possible, as each level would have to remain in the stack until its subtree completed. We will return to the permutation problem in a moment, but first another example of a more traditional recursive solution illustrating the dynamic type checking built into Protocol. Here we wish to beautify a list for display to the user:

```
define _ print
{
   let ( input ) -> x
   if typeof x = list then
   if x = null then output _ "{}"
   else if typeof hd x = list then
      output
        "{", ( print _ hd x ), "},", ( print _ tl x )
      else
      output
        ( print _ hd x ), "," , ( print _ tl x )
   else output _ x
}
```

where again the output has to be ordered on the terminal and so the various instances printing sublists have to synchronise correctly.

All of these simple examples are directly expressible in sequential procedural programming languages allowing recursion. None really exploit the potential for interaction

which Protocol offers. Consider the next definition, which is another form of the above permutation mechanism.

```
define _ permute
{
   let identification => self
   let ( input ) -> s,p,r
   repeat
   {
      if r = ø then world _ s,"'n" else
      for i = 1 to sizeof p do
      {
         let s ++ p[ i by 1 ] -> ns
         let sizeof p -> length
         let { if i+1 > length then i else i+1 } -> break
         let p[ 1 by i-1 ] ++ p[ break by length - i ] -> np
         self _ ns, np, r-1
      }
   }
   while message self
   do self : -> s,p,r
}
```

This version, like the first, sends messages representing the permutations still to be found, but unlike the earlier case, where the messages were sent to new incarnations of permute, this one sends itself the messages, and reads them back in when it is free, ie after it has found a permutation. This instance of permute will therefore find all the permutations on its own, rather than calling on a family of similar instances to help it.

Another facet of the message system is the ability to send or read in messages at any point. This is seen in the permute definition, which picks up its own messages, but can be used in a slightly different way to produce 'memo'-functions - instances which remember previous results. Consider the simple fibonacci series evaluator:

```
define _ fibonacci
   let {1,ø},{2,1} -> list results
   while true do
```

```
   {   let input : -> int n
       if n > sizeof results do
       for i = hd { sizeof results of results } + 1 to n do
       {
          let hd tl { {i-1} of results } -> i1
          let hd tl { {i-2} of results } -> i2
          results << { i, i1+i2 } -> results
       }
       output _ hd tl { n of results }
   }
}
```

which holds the values known so far in a list, and loops back to pick up further requests. If the required value has already been calculated then it is written straight out, otherwise the known-value list is extended to include it. The more this instance is interrogated, the more it remembers, just as with memo-functions.

Note, however, that in this case the 'memory', in the form of a local list, is implemented by reading in new requests within the body of the definition, not by some special mechanism specifically for this purpose. In effect this provides the "own" property of Algol-60.

Several other examples will be given in the section dealing with CAMAC, since they provide administrative facilities for this application. The Protocol system, as explained earlier, permits instances to find out about any other member of the system. The following system state-display is quite useful.

```
define _ system.survey
{
   let survey -> sys
   world _ "'nSurvey Of System'n'n"
   forall id in sys do
   {
      world _ stateof id,"'t"
      world _ if stateof id = waiting
              then holdup id else "'t't"
      world _ "'t",id
```

```
      if stateof id = aborted
      do world _ "'t't", faultof id
      world _ "'n"
   }
}
```

Similarly, an instance can examine the queues of other instances, as well as its own; here we have a queue state-display:

```
define _ queue.survey
{
   let survey -> ids
   world _ "Survey Of Inputs'n'n"
   forall i in ids do
   {
      world _ i,"'t"
      world _ if stateof i=waiting then holdup i  else "'t"
      world _ "'t"
      world _ if {active|waiting} has stateof i
            then census i else "'t't"
      world _ "'n"
   }
}
```

The famous dining philosopher (spaghetti eater) problem can be written as follows:

```
let ( input ) => right
let ( input ) => left
let ( input ) -> forks
while true do
{
   while forks ~= 2 do forks + ( first THINKER ) -> forks
   world _ identification," is eating'n"
   right _ 1
   left _ 1
   Ø -> forks
}
```

where each instance of this definition is initialised by being passed the instance value of its immediate neighbours, left and right, and the number of forks with which it is to start. The various THINKER instances will therefore start off with either 2 or Ø forks, except the odd man out, which will have the remaining fork. All interactions are

automatically synchronised, and the philosophers settle down to a fair and uneventful life, eating and thinking!

9.Ø Protocol As A CAMAC Programming Aid.

In this section we will show how the general philosophy behind Protocol can be put to use, as an aid for programming CAMAC hardware. CAMAC [E2] is a hierarchical hardware system, and it has been adopted as a standard by the IEEE. It is mainly used in high data rate electrical monitor and control equipment, such as in nuclear physics, astronomy or medical data acquisition. CAMAC is basically a plug-in system for hardware 'modules'. The standard specifies physical and electrical characteristics for such equipment. The standard defines signal protocols for inter-module communications, but it does not specify any communication format between external processors and a CAMAC system.

9.1 A General Summary.

There are various schemes for such machine control already in existence, but it is felt that these are lacking in several respects. Most were developed from add-ons to existing languages, notably FORTRAN [H5,K1Ø] and Basic [E1,K9], or by standard macros into such low level languages as Forth [S1Ø]. Each has its own devoted following, and each has, as always, its attractive features. The FORTRAN facilities permitted substantial and easy data analysis to be carried out, by way of a similar approach to accessing the NAG Libraries. The Basic fraternity could point to the interactive nature of their base language, but could not perform such exhaustive data analysis so easily. The Forth people pointed out that the design of Forth permitted them

to programme intricate machine control sequences easily, and so CAMAC standards were soon grafted on here too. Those users though, soon realised too that the extreme power to fiddle with the individual bits of the machines was hindering them in the realm of data analysis, when fourier transformers etc were to be written!

It was decided herein that the add-on strategy so avidly expounded by the CAMAC IML [L8] Committees (the CAMAC standards people for software) was not really adequate, as it merely half-introduces CAMAC control to the host languages. To be done more effectively the CAMAC programming facilities must not only introduce a means of defining and operating on CAMAC names, but they must also harmonise with the language, not distort it, so that they exploit the natural power of the language to its fullest. It was therefore decided early on that CAMAC hardware should appear as a new data type in the language. Thus the type site was introduced (see below). This single data type is used for all items in CAMAC, the branch, crate, module or subaddress attributes of a particular item being part of the site value, automatically deduced during such a declaration. The value of a site is effectively its CAMAC address (in terms of BCNA codes). This however is not wholly new; some Basic systems, notable RT-Basic [E1,K9], have CAMAC items introduced as 'variables', but they are special process variables (to use their terminology), and as such are a special case (as indeed most things in the Basics tend to be). Here the site values are thoroughly normal, that is

they have the same 'civil rights' as values of any other type (in accordance with the principles of data type completeness and correspondence).

Thus it was decided that the CAMAC programming facilities, embedded in an otherwise self supporting language, should not only fit into that language naturally, but they should also be able to exploit it fully.

9.2 Sites.

The 'peculiar' type is called a site, and it represents a piece of hardware in a CAMAC system. There are four variants in such hardware, branches, crates, modules and substations, but all are of type site. CAMAC offers a powerful tree-structured architecture, of branches of crates, holding modules, which in turn house individual registers, or substations. Although it is bus-oriented it has a fully synchronous bus protocol, so bus operations can not be multiplexed, and transfers are quite slow (typically two microseconds). However there are many interlinked busses in a fully tree-structured system; each crate has a bus of its own, and the appropriate busses are linked together via branch highways (longer busses) whenever intercrate transfers need to take place. The entire system is therefore made up of largely independent crates of modules, working in parallel. At the root of the system lies the master Controller, and each crate is supervised by lesser crate controllers, each autonomous within its own crate. To communicate with any item in a CAMAC system it is only

necessary to specify the CAMAC address (in terms of numerical B,C,N and A codes) and the operation required (the F code) to the main controller. This usually looks like a peripheral to the computer.

In principle then, programming CAMAC is quite trivial. In practice it is rather tricky (if it is to be done well). In Protocol we have introduced CAMAC as a data type in the language, the power and flexibility stem directly from a strict adherence to the guiding principles of programming language design.

There are no literals of type site, but there is one predeclared site valued name "root", which is the top of the hardware tree. This is a constant. There are also four predefined library routines, already initiated, whose identity values are held in predeclared constants, which take various inputs and return a value of type site.

9.3 Our View Of CAMAC.

The CAMAC facilities implemented with Protocol are intended to exploit the interactive base, to provide a flexible tool for setting up, monitoring and if necessary, dynamically adjusting a CAMAC system. To this end names introduced for CAMAC sites build up a symbolic tree structure, directly representing the hardware architecture, and this tree is available to users to survey, traverse and adjust at will.

For instance, as one would expect in a file system, it is possible to display the current contents of any particular

item in the tree. This could be useful, say for general system housekeeping or statistics on usage etc. The user has a pointer into this tree marking his local objects. It is quite feasible to change the current container marker within the system. In fact each instance has a predeclared constant, "container", indicating its current position in the tree, so each instance can change its locality in the CAMAC system. All operations are relative to this, unless expressed by path-names from the "root". All this, though, is rather administrative, primarily concerned with observing the system.

It is further possible to move items about within the system, so long as it makes sense CAMAC-wise. One could, therefore, develop software for a given set of modules in a specific crate, then later (even while running) adjust the layout so that those modules are logically transposed to some other crate, simply by changing the position of their container. This would be the logical counterpart of unplugging a crate at place-A and reconnecting it at place-B, perhaps some distance away. The programs which use it need not be aware of this change, as they could find its place in the tree dynamically, wherever they are set up, or whenever it moves while 'on-line' to the system.

This is not as fanciful as might at first be thought, as say, if the pin connectors at one module station turned out to be faulty, that module could be slotted in and moved to that position logically, and all programs which use it will automatically find it there. There is no need to change

them.

9.4 The Tools Available To Us.

9.4.1 Creating CAMAC Sites.

The site creating instances "B", "C", "N" & "A" are predeclared, and so are immediately available. They take, as will be seen, several possible sets of inputs, and return a site value if no error occurs. The basic format is the path, in existing site values, to the desired container, either from the local position or from the root of the system. If none is given the current container is used. This can be followed by an integer value, and if present this is taken to be a request to make a new site at that position, if not no particular place is looked for. Finally a string can be taken. This forms the name by which the system will know the object in the CAMAC tree. If an integer code was given then an attempt will be made to add a site at that place and give it the associated name, returning its site value. If no code was given then the current container will be searched, by name, for the appropriate object and, if present, its site value returned, its actual position being immaterial here.

This flexibility of inputs is characteristic of Protocol, consider:

```
let B _ 2,"lab" : => a
```

which tries to make a new site at position 2 in the current container, initially the root crate of the system, and since we are trying to site a branch it has to be there, as opposed to:

```
let B _ "lab" : => b
```

which, because it hasn't got a numerical argument, doesn't try to make a new site, but searches the current container for an item with the name "lab" and returns this site value. Naturally it is illegal to either make a new site at an occupied position, or attach on to one that is not there.

This permits the dynamic binding of program names to existing sites, as well as the creation of new sites.

It is quite obvious that there are two naming conventions, those internal to a program, and those outside, for any CAMAC item. A site can, therefore, have a variety of names in different routines, either from being linked to as above, or by being passed as messages, but only one system wide name; that nominated when the site was initially made, "lab" in the above example, with its two internal, temporary names "a" & "b". This is the 'official' name of the item, but all represent the same object.

The user, able to inspect and alter the tree structure of the CAMAC system, is able to display the contents of any item, and access them via their stated names, if they were set up before the current session, or recently from another instance, or by his local name if they were created during his current session, or have been linked to.

Now, a library routine can be set up to use a given pattern of CAMAC items, not in any particular positions, and having had these items placed beforehand, it can automatically find them wherever they happen to be, so long as that part of the

tree pattern is correct.

The way to change the current position in the tree, originally at its root, is to enter a given path. This leaves you at the end of the path if it exists, and where you were if the path was broken. This instance is also predeclared, and returns nothing.

There are a number of other predeclared instances which take a site value, or a path leading to one, the current container being assumed if none is given, and these may return lists of values.

9.4.2 Locate And Extract.

The locate facility produces a list of site names, tracing an object from the root, and extract does the same but with a list of position codes and BCN or A codes, indicating the structure of the hierarchy to that place. It is important to note that such routines do not print their results but pass them back to their caller. This can then be printed, if that is all that is required, or examined and used in whatever manner is deemed appropriate. This will be mentioned again below.

9.4.3 Display.

There is one other major administrative feature, the display. This also takes a site, or a path to one, or the current position by default, and produces a list of lists, each sublist being the pertinent information describing the items in that container. Thus it is perfectly feasible to

examine the distribution of modules in a crate, crates in a branch, or registers in a module, simply by sending its site value to display, and printing the list it produces. All values in Protocol can, of course, be printed out, but any list so produced will require 'beautifying' as lists are printed "flat" by default, but this is a minor point.

Now, as indicated, these are merely tools with which to construct more sophisticated routines. They are simply interrogative routines which report various information about the CAMAC layout. For example, display shows the contents and status of a given item, and it might be desirable to be able to view the entire hierarchy, sequentially, so rather than repeatedly request that specific items be displayed, manually, it is quite easy to write a library routine "show.system" say, which displays the root crate, takes its list of lists and recursively descends each of its constituent containers, in whatever order is required.

Consider, for example, the following definition "select":

```
define _ select
{
   let input : -> l,s
   let empty -> out
   introduce _ UNKNOWN
   let { if s=B then 1 else if s=C then 2 else if s=N then 3
         else if s=A then 4 else { world _ "??" ; -1 }
     } => S
   if S = -1 do abort _ UNKNOWN
   while ~ endof l do
   {
       if hd hd l = S do    out <> hd l  -> out
       tl l -> l
   }
   output _ out
}
```

which, given a list of sites and a 'key', will select those sites in the list which are of that form (ie B,C,N or A). Note that an instance of this definition will abort with the new exception valued literal "UNKNOWN" if the key does not match any of those catered for. Normally an instance of this definition would be sent the output from a call to display, that is:

```
display : select _ N : output
```

which will write out the (still flattened) set of lists holding module information.

Another useful facility, when writing out lists of information is a tabulator, such as is provided by the next definition:

```
define _ table
{
   let input : -> x
   if typeof x ~= set and typeof x ~= list then world _ "??"
   else
   {
      if typeof x = list then
      {
         while x ~= null and typeof hd x = list do
         {
            table _ hd x : output
            tl x -> x
         }
         if x ~= null do
         {
            let hd x -> M ;   tl x -> x
            world _ { if M=1 then "B" else
             if M=2 then "C" else
             if M=3 then "N" else
             if M=4 then "A" else "??" },"'t"
            world _ hd x,"'t" ;   tl x -> x
            world _ hd x,"'n"
         }
      } else forall i in x do table _ i : output
   }
   output _ ""
}
```

which would slot into the chain immediately before the information is written out:

display : select _ N : table : output

In this case the output from select will be arranged in the form of a three columned table giving various characteristics of the items selected from the display.

This demonstrates the power of the message passing system, since information is handled uniformly, being processed en-route as it is chained from its source to its destination.

9.4.4 Moving CAMAC Items.

The move instance, again predeclared, given a site (either from a simple site expression, or some arbitrarily complex statement of that type) and a (perhaps trivial) path to another place in the tree structure, and a numerical position code, will try to reconnect that site at that position in the container indicated by the second site value. That place is checked out first, to ensure that it is free and that the two items are compatible CAMAC-wise (modules in crates, crates in branches etc). The move only occurs if all the parameters are valid. This then makes the item appear to be in the new position in the CAMAC system, and so it should be physically moved to match its logical position.

In the implementation it is essential that this entry in the tree of entries, tracing out the system, remains fixed, as

the value of a site is essentially a pointer into a data structure containing these entries. Thus the pointers threading the tree are adjusted, the actual entries remaining fixed. This structure is discussed below.

9.5 Interacting With CAMAC.

Now, when a CAMAC function code (F-code) is to be initiated it requires that the appropriate F-value & CAMAC address (BCNA) be encoded into some CAMAC Controller dependent bit pattern. Naturally, in Protocol, the user need not be aware of how this is carried out, he does not need to know anything about the particular interface used, only his module positions and the F-codes needed to control them. Once he has defined names for these objects and functions he can forget that they are CAMAC items, and programme them as if they were normal data items and instances.

The definition of names to be employed as CAMAC F-codes is quite straight forward. There is a predeclared instance, called "F", obviously enough, and this takes a numerical F-code as its input, optionally followed by a value of type-type, type int is default. This type specification either tells the system to check inputs when applied, or produce output of that type (CAMAC functions are relatively simple, they either take or produce, not both). Since CAMAC is rather limited in the hardware data types it handles, it is likely that this will be either int or bool, for data or status signals. Such a call to F produces an instance as a result, which can subsequently be applied in its own right

to a site. For example:

```
let ( F _ 8,bool ) => state.of.module
```

would indicate that "state.of.module" would be an F=8 command, actually a test of LAM status, and the result is tagged as a boolean value. It could be applied by:

```
if ( state.of.module _ counter ) then ... else ...
```

to select actions by the current state of that piece of hardware. It is quite transparent, as the CAMAC is hidden from the user.

9.5.1 A Brief Example.

Here we have a preset (an interrupt on zero counter) and two scalers (pulse counters) in another module, we wish to load the preset, have it count down and interrupt, at which time we will read the counters. This is based on an example used by Stephens [S1Ø] in a comparative study of CAMAC oriented languages. That report considered various versions of Basic, FORTRAN and Forth, all of which were programming CAMAC via 'add-ons'.

```
let A _ preset,8 => timer
let A _ scaler,Ø => sky
let A _ scaler,1 => star

let F _ Ø => value.of

clear _ timer
if ~ accepted then output _ "Faulty Setup!'n"
else
{       world _ "Interval ? " : deposit _ timer
        while ~ lam _ timer : do {} !Busy Wait!
        let value.of _ star : -> v.star
        let value.of _ sky  : -> v.sky
        world _ "Star =", v.star," Sky =", v.sky
}
```

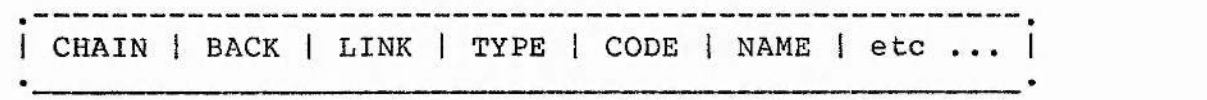
The text between the shrieks ("!") is taken to be a comment.

This assumes that "preset" and "scaler" have been sited as modules (N) and F-codes such as "clear", "lam" & "deposit". The "accepted" value merely checks that the last instruction to CAMAC was accepted - that is that the hardware exists (for those in the know it is the X-test).

9.6 Implementation Details.

Each site value initialised corresponds to an entry in the data structure holding all the information needed in order to programme CAMAC. This table is threaded, tracing out the current hierarchy of the CAMAC hardware. When dynamically altered the various pointers within the table are reset, the entries remaining fixed. The entire scheme depends on this fact.

As it is necessary to work down the tree, for new sites to be added, and also upwards, to dynamically extract the current position of an object in the tree, it is clear that the threading within this table will have to be bidirectional. The entries take the general form:



The BACK link points to the entry corresponding to the container in which it sits, for working back up the tree. The LINK connects items in a CHAIN hanging off a container. The CHAIN here is the base of a LINKed list of items to be

placed within this item. The TYPE is the B,C,N or A designation of the item. The CODE is its numerical position code within the container in which it is to be found. The NAME is that seen in display (not necessarily the same as that used locally). The rest is system housekeeping.

9.7 Driving CAMAC Hardware.

When an instance corresponding to a CAMAC F-code is involved in a transaction the system recognises this fact and sends the F-code together with the site value and any parameters to the CAMAC Table Manager. The Manager is another part of the system, distinct from the interpreter, and its job is to build and maintain the tables mapping out the CAMAC system.

In practice the Manager resides in an on-line microprocessor, along with the CAMAC device driver and assorted user programs. The actual i/o is controlled from here, so that CAMAC is invisible to the main processor.

Given a site value the Manager returns the current BCNA, as appropriate, to identify the path (CAMAC address) of the site in the system. These, together with the F-code associated with the operation required are then encoded, in the micro, and physically directed at the CAMAC Controller.

This hardware unit then strips apart the command and its target, carries out the operation and, if appropriate, sends an interrupt to the micro. On receiving such an interrupt the micro reads in the BCNA of the LAM source (supplied by a diagnostic module called a Grader working with the Controller within the CAMAC system). This is then sent on to the Manager, which traces the appropriate table entry and picks out the service routine required. In practice this will probably be an instance actually awaiting a response or it could be a routine instance linked to a LAM source, to be triggered off whenever that LAM is asserted. It is

considered to be an error to generate an unexpected LAM interrupt - this is taken to indicate that the hardware is malfunctioning.

9.8 Access To Status Registers.

The exact nature of the interface of Protocol to CAMAC depends upon the particular Controller employed. Using Controllers produced by different manufacturers requires that the software driver reflect the particular Controller design adopted. There are many such designs, some more suited than others to different applications. For example the General Electric Co. Executive Controller has a single control & status register (CSR) and (in the PDP-11 version) it looks like a device on the UNIBUS with thousands of registers. It is programmed by addressing a register at a particular address, that being an encoded form of the CAMAC operation required, and this is stripped down by the Controller. The NE [N5] Series-9000 Controllers, however, have a main CSR and a small cluster of addressing registers which are loaded with equivalent information. These hardware differences are, however, completely hidden from the user of Protocol, and are only apparent in the machine-specific part of the driver.

There are several status lines in CAMAC, and their state is usually available by way of bits in a CSR. However this is managed by the Controller employed, it is the responsibility of the driver to make them available to users. This though, is done such that these signal lines are readily available,

in a transparent yet unobtrusive manner.

There are several predeclared variables, corresponding to the signal lines. There is, for instance, a line (called 'X' in CAMAC parlance) which is asserted (made true) if the last command to CAMAC was accepted, so it is introduced as a boolean called "accepted". It can be used, for example to test if a module is actually in situ. It is available if needed, and is set automatically. If not needed it can be ignored.

In a similar vein there is the Q-line, here called "response", which is set or cleared (made true or false) in response to various commands. This is a commonly used means of testing signals in CAMAC. The state of X and Q are set into their associated variables as each CAMAC operation is undertaken. They are therefore there and set each time, if needed, yet inconspicuous if not required in any particular operation. This is considered to be better than the recommended IML practice [E3] of having to supply some user defined variable as a by-result parameter to every call to some extra-language standard subroutine for CAMAC operations (such as is the case in the more common add-ons to FORTRAN and BASIC).

Naturally, in view of the parallelism inherent in Protocol, these variables cannot be uniquely defined in the system itself, but must be private to each instance, so that each instance can operate on CAMAC in parallel with others (carrying out unrelated tasks) yet refer to the state of the

"last CAMAC operation" as if it were the sole source of CAMAC commands. Each instance has, therefore, an image of the state of the CSR status lines which it can examine at its leisure, so permitting other instances to operate on CAMAC items (via the CSR) immediately thereafter.

9.9 On Multi-Processors.

This separation of powers, the user and real time programming on-line in a main processor, and a micro dedicated to servicing CAMAC and the execution of tested, down-loaded routines, in production, is considered to be quite efficient. It permits each activity to run at its optimum speed. The micro need not be concerned with the high level user language, and the main processor need not be concerned with the peculiarities of CAMAC, indeed it is invisible to it.

Nowadays, with micros becoming more powerful this strategy is perfectly feasible. This is especially so because microprocessors are now being housed within CAMAC crates. Indeed the Controllers themselves are becoming intelligent, and in one case the same CAMAC unit contains a DEC LSI processor, a standard CAMAC Controller, a Grader and DEC terminal and Q-Bus driver, making it almost ideal for our purpose.

10.0 The Implementation Of Protocol.

Protocol is visible to the user as an interactive high-level language. He communicates with the compiler which analyses his commands and reports any syntactic errors or obvious type mismatches. The compiler then produces 'machine code' for an interpreter which then performs the desired actions. Any output or direct input is supervised by the interpreter. If the desired actions fall outwith the set of sensible operations then the interpreter presents some diagnostic to the user, resets itself and reports to the compiler that the last request failed. By cooperating in this way the compiler and the interpreter combine to implement Protocol.

For this project Protocol was implemented on a Digital Equipment PDP-11 computer running under the UNIX system [R8].

10.1 The Protocol Compiler.

A language and its compiler are closely related - they are best designed together, but neither should be compromised at the expense of the other. The actual concrete syntax to be compiled, although essentially independent of the underlying language, should be chosen so as to be readily parsed. If it is not then the compiler's job will be all the more difficult, and its size and complexity enlarged accordingly.

Because Protocol is interactive it's syntax is quite terse, rather than being elaborate; the amount of text being significantly reduced by the omission of formal parameter

declarations, together with type keywords in declarations.

In general Protocol is an LL language [F5,K8], making it particularly well suited to recursive descent parsing, without incurring back tracking overheads. The concrete syntax adopted is almost entirely LL(1). Most constructs actually start with a terminal symbol, such as 'if' or 'while', and so need no look ahead, but at worst, we need to examine the symbol after a starting non-terminal. In an incremental compiler it is highly desirable to be able to parse in a top down fashion, so this grammar is quite well suited to our purposes. Most other attempts to develop incremental compilers, for example [C2,G2,G3 and P3], have been based on LR(k) grammars, with elaborate tree-forms for holding parse trees. Parsing top down is, however, not without its problems, because it is not always possible to peep ahead into the source, at the end of a line for example, and this poses a slight difficulty.

The compiler is built along the lines of one suggested by Ammann [A1] some time ago, and owes much to the compilers written by Morrison [M6] who refined this simple but powerful version of recursive descent.

There are a few places where Protocol becomes rather difficult to parse. The most obvious case concerns the syntax of the generalised message interactions. The various forms and optional terms are parsed essentially bottom up, and are resolved via context-dependencies.

This, however, is justified on the grounds that it is a

highly localised irregularity, and is included because the resulting "chained" syntax is particularly attractive, especially interactively, as it allows for various optional, or default constructions. As message passing is fundamental to the system, and as it is interactive, it was felt that it was not prudent to clutter such constructions up with brackets and other punctuation, simply to make it parse more easily, especially as such constructions are susceptible to typing errors. Nor was it desirable to split operations into primitive actions, as this requires a lot of typing for even simple interactions.

The recursion is broken when parsing expressions, as it is felt that this is better done via iteration following an operator precedence grammar. This is quite acceptable. The recursive descent is employed mainly to parse the control structures and follow the block structure, for which it is very well suited.

As an example of tailoring the concrete syntax to aid the compiler, the assignment statement is reversed from its traditional form. The usual ":=" is now the "->" symbol, and the l-value is now on the right, with the r-value on the left! The 'becomes equal to' is now 'goes into', following the arrow. This is not really too confusing, so long as the terms are not taken literally. The flip was made in order to permit lists to be parsed properly. A list of identifiers (l-values) to be simultaneously updated from a list value cannot be readily distinguished from a list value enumerated as the end of a block expression. This then is uprooting

the traditional syntax to conform with the LL(1) grammar, as opposed to above where it was deliberately broken.

10.1.1 Error Detection.

There are three sources of error: context-free lexical errors, ill-formed syntactic clauses, and context-sensitive type errors. The simple lexical errors, unknown symbols etc, are detected by the input scanner and are reported as such. That which passes the lexical analyser is now in the form of 'basic symbols', rather than in the form of the concrete representation. Syntax analysis as such is undertaken by a set of mutually recursive procedures, one for each syntactic construction, and follows a recursive descent.

Being one-pass the compiler takes the raw input and feeds it thru the parser. During the descent the context sensitive syntactic errors are detected and reported. On the way back up from the recursion the compiler type matches the various expressions, reporting any type errors. This technique produces a small, fast and intelligible compiler, and its error diagnostics are made clearer because the nature of an error is clear as soon as it is detected. Because the compiler cannot employ a complete static analysis of expression types, it relies rather heavily on 'excess' symbols, such as the "_" and ":" of message interactions.

10.1.2 Error Recovery.

Having detected an error a question naturally arises - what

degree of recovery should be attempted? There are various levels of recovery which are employed by different compilers, exploiting different aspects of the parsing technique for their implementation. Of these the simplest of all is to simply abandon the compilation, skipping over the rest of the input. In an interactive environment this is actually quite an attractive option, and to some extent it has been adopted here.

Erroneous input will never be executed so if parsing continues the code generation can be disabled, relieving the burden of address calculation etc during the recovery. As the system is interactive only one error message per faulty statement is generated, so as to prevent the user being flooded with diagnostics from a single line of input (remember that an error detected deep in the recursive descent is likely to generate others on the return).

The strategy adopted for Protocol is therefore quite simplistic, there is after all little point having a sophisticated scheme. When parsing at most one error report is generated per statement, and if the input was a compound statement (a sequence) then one is allowed per component. This also applies to the on-line definition of library routines, as they are simply named statements. In the case of library definitions the listing, with any error messages embedded in it, is stored in the library for later viewing, and a report is delivered to the user informing him of the success or failure of the addition. If this was indeed faulty then it can be edited and recompiled.

10.1.3 Code Generation.

The one-pass recursive nature of the compilation process is exploited by embedding code to generate abstract code in line, at the base of each level of recursion, so having passed over the input text once, it is completely parsed and the appropriate abstract code is planted.

We do not, as often seems to be the case, produce a parse tree after the parse, and subsequently traverse this in order to generate code or interpret it. We plant the code as soon as it is available - at the base of each level of recursion. If the parse was completely successful, with no errors being detected then that code stream is sent to the interpreter for immediate execution. This is quite straight forward - there is no code to plant parse trees, none to read them in and none to evaluate them, and there is no need to store large tree structures, as we never explicitly manipulate the parse as a tree. A tree is present, conceptually at least, during the parsing, but it is implicit, being embodied in the stack, as it follows the recursive descent.

There are various reasons for choosing such a scheme. Firstly, since the compiler works incrementally when directly interacting with the user, it must generate code quickly for the interpreter to work from, so we should make as few passes of the input as possible. This code must be produced at the time, not later on, and it must be self-supporting, not be dependent upon code from the previous

input (now executed). Remember that when defining library routines, which are compiled and stored for later execution, such libraries must be compact, and be able to be called upon at any time. Secondly, going straight into abstract code is preferable, in an interactive situation, to holding and incrementally updating parse trees, especially in the case of library definitions.

In such 'hybrid' systems, where source is compiled into an abstract code which is then interpreted, although a program will run more slowly than one compiled into real code, the abstract code can be designed so as to be extremely compact. The degree to which this is possible depends upon the 'height' of the abstract machine above the real machine, in relation to the language being coded. In Protocol this is quite significant.

Traditionally incremental and interactive compilers have tended to employ tree representations for the compiler output. This is then interpreted, it being argued that this is better than interpreting the source directly, as was the case in the earliest interpreters. Interpreting from a parse tree is only marginally better, as much time can be spent 'walking' over the tree, and searching the environment for entries, especially if such tables contain type matching information.

The interactive nature of the applications being supported by such compilers nowadays has though, one major crippling effect on such tree based schemes, and this concerns the

patching up of trees after dynamic errors, or on-line changes of lines of already parsed input.

This problem is made even more acute if the system is enhanced to run faster by dynamically replacing the nodes of a tree with values once the corresponding subexpressions have been evaluated. Adjusting trees to follow changed input is even more difficult in interactive compilers working on block structured languages, as can be seen for example in [C2], where this is discussed at some length, for the case of LR(k) parsing.

The case against interactive systems working from tree-based program representations is quite strong. Bornat [B13] notes that because a program compiled into an abstract code is already linearised, the interpretation of such code will be faster than walking an equivalent tree structure.

One of the major reasons, it seems [B13] for the preference for tree-walking interpreters in interactive systems is the desire to be able to reconstitute the program from the tree, by using an attached symbol table, in order to produce more meaningful error diagnostics. That this is advocated surely says more about the inadequacy of the error detecting capabilities of such systems rather than the 'efficiency' of tree structures! In some such systems the symbol table is also used to hold type information, and this is searched dynamically in order to perform run time type checking.

Protocol is not implemented in this way. Firstly the tagged architecture of the abstract Protocol machine permits type

and constancy information to be part of a location along with the value itself. This makes dynamic type checking very simple and very fast. The space overhead needed for the tagging depends upon the nature of the underlying real machine. On a suitable machine this could be a non-existent overhead. Even so the tags are likely to be no more costly than the symbol tables, and are easier to use.

Secondly, despite the fact that Protocol throws away the symbol table during the parsing process, making it impossible to reconstruct a readable program from the machine readable form, the library keeps a listing of each routine, along with its abstract object code, and so it is an easy matter to recall this to digest an error message. Dynamic errors are reported in terms of routine name, line number and a brief diagnosis of the error. Any run time information, such as the actual as opposed to the expected types, if the error was a type mismatch, can also be provided, so it is an easy matter to determine the exact cause of an error. Once diagnosed the routine can be corrected and recompiled for later use.

10.2 The Library.

In an interactive system it is frequently necessary to either define new routines or adjust existing ones. The Protocol Library assists here too. Because routines are compiled straight into abstract code, and these tend to be short, there is little overhead in completely recompiling the body of a routine. This will most likely be no worse

than attempting to trace and patch up a parse tree, especially in a block structured language. The system therefore provides a simple editor for examining and changing the library routines.

The very existence of the Library offers a degree of abstraction not found in the Basic-like systems, and the block structure together with the universal message passing system make Protocol much more useful.

10.3 An Abstract Machine Architecture.

The Protocol system acts conversationally, permitting on-line development of programs. The user's text is read in by the compiler and this is compiled into instructions for an abstract machine. This code is then executed by the system's interpreter. In this way the interpreter effectively simulates the abstract machine on the underlying real computer. The notion of an abstract machine can be traced back to the 'beta machine' of Randell and Russell and to Landin's classic SECD machine [L5,R3,R4]. These formed the first real attempts to implement a software machine, and because this approach permits the design of 'clean' high-level machine structures it is now a widely accepted technique for enhancing language portability. A discussion on the motivations for such implementations can be found in [E5] by Elsworth.

All of Protocol is constructed around this abstract machine. The interpreter is its implementation on a given computer.

To move the system to a different computer one must reestablish the interpreter on that machine. We have examined the Protocol compiler, and the way it handles the source text, parsing it, checking it out, as far as it can, generating the code for the interpreter, we will now concentrate on the interpreter itself.

10.3.1 Overview.

The abstract machine at the heart of the Protocol system reflects the parallelism of the language. At any time the system comprises the interactive outer level's environment, and an arbitrary and dynamically variable number of instances of library routines. Each instance carries within it sufficient information to permit the interpreter, simulating the abstract machine, to switch between instances, as circumstances demand, without loss of continuity. This produces the illusion of concurrency.

Initially it executes in the outer interactive level, switching to other instances whenever necessary. The rules for such switching constitute a scheduling strategy. The current Protocol strategy generally permits an instance to run until it either waits for as yet unproduced input, or it exits (either normally or with an exception), at which time another instance, if one exists, is selected.

When an instance waits for input, its producer is reinstated; this occurs recursively, until the desired value is produced. This is the basic philosophy; it could be said

to be a "lazy" scheduler, as it doesn't actually run anything until it has to. Actually it is rather more responsible than that, as it monitors the build up of messages awaiting service and when things look as if they are getting out of hand such as too many at a particular instance the interpreter tries to reactivate that instance in order to to clear the backlog.

It is quite straight forward to schedule an instance which is blocked waiting for input (we start its companion), but whenever one terminates, the system is examined and the scheduler chooses one arbitrarily. In cases of abnormal termination the system clears away the remains of the ailing instance before reselection.

Each instance in the system has certain information representing its environment and state. This is called its frame. The system scheduler works on a linked list of such frames. Each frame includes, amongst other things, the instances identity, its title (definition name), various counters and machine registers, a pointer to its executable code (in the library) and a pointer to its initial stack segment, on which it operates when active. The list of frames is also interlinked, or threaded, with pointers to other frames, tracing out the master - slave hierarchy. The current status of an instance (waiting, active, completed and aborted) is held in the main list of frame pointers, for the use of the scheduler.

Each frame has its own stack. This is private, so

communicating instances send messages to one another (rather than take and leave items on each others stacks). This involves a sender placing a message on the input queue attached to the receiver instance's frame. The input queue is a linked list of messages, all of which are tagged with the identity of the sender and its type (for the dynamic type checker). Since it is a list, messages from a particular source need only arrive in relative sequence, they can be intermixed with messages from other sources. This asynchronism is important, as it is both in harmony with the parallelism in the system and is tolerant of variable transaction rates for any pair of communicating instances. The scheduler can therefore select the next instance without having to schedule according to as yet unfinished transfers. It is akin to having a pool of message buffers, commonly available, in which more can be sent even though the consumer hasn't yet received earlier messages. This tolerance could not be achieved if instances had a single (or some fixed number of) message buffer, as the sender would then have to be delayed until the receiver had cleared out his buffer! A discussion of the importance of such tolerance to temporary overproduction of data, and its associated asynchronism can be found in [C7].

The abstract Protocol machine is stack oriented, but rather than employ one large stack, accessible to all instances, each instance's frame has a separate, much smaller stack.

As the number of instances in the system is dynamically variable, with new ones being activated, and others

terminating or aborting, it is obvious that storage must be dynamically allocated. In principle, however, there is no reason to complicate the abstract machine; we could give it an infinite store, in which space is always available, and simply forget about now unwanted store, and similarly, we could regard the system as being composed of an infinite number of instances, most of which are awaiting a routine definition to execute.

The actual computer, on which this is being simulated (by the interpreter) has, of course, an all too limited store, and so a scheme must be devised for releasing and re-using no longer needed storage. The interpreter, then, takes as much store as it can get on the real computer, and starts carving off space for instance frames and their associated stack space. Whenever an instance terminates the bulk of its space is released, only a small epitaph (in the main list) remains to say how it passed away (ie did it terminate or did it abort) so that its companions may be informed as and when necessary.

Whenever the system finds that it has allocated all of the space initially available, it takes a survey of all the still active instances, marks their space, and collects up all of the remaining garbage (the space left by terminated instances and obsolete data structures) for reuse. The system scheduler, being all knowledgeable, can adjust its scheduling strategy to suit the space constraints. When space is plentiful it tends to create new instances and leave them as runable but as yet uninitiated, and continues

with the creator, until it has to wait. If the space begins to run low then it senses the frequency of garbage collections, and automatically changes gear, becoming more conservative, running old instances trying to coerce them to completion, and reading in messages, so as to release space. The performance of such a strategy is naturally a matter of fine tuning.

The interpreter acts, therefore, like a typical garbage collector for a heap based language implementation. There are obvious disadvantages: the heap might become hopelessly fragmented, so making reuse difficult, and any system which habitually grows too large on a small machine will have to take drastic action, but this is merely an indication that the underlying computer is not sufficiently endowed, not a fault in the Protocol system. The infinite store is only an illusion in the implementation .

The task of the garbage collector is simplified by the existence of the tagged memory. It only needs a pointer to the root of the main list of instance's frames, and from here it can recursively descend all heap pointers. Thus it follows the stacks, and the compound items residing in them, and traces the input queue of messages, marking all of the heap blocks currently in use. The remaining heap blocks are then chained together, into a free-space list, with adjacent free blocks being coalesced into a single bigger block. It can identify every object it finds simply from the tag field, so all that it need know about are those tags which denote block pointers. As all items in the heap are acquired

from the system via an allocate instruction, all items in the heap are blocks. As the entire store is arranged as a heap of variably sized blocks it is clear that the garbage collector is part of the fabric of the machine itself.

10.3.2 On Data Structures.

As far as the compiler is aware, all objects are placed on the stack, and are all of the same size. This is an important piece of deception, because as the the compiler cannot always deduce the type of an expression it would not be able to manipulate a multi-stack implementation. In the case of simple objects, such as integers, this is straight forward. In the case of strings (and other complex objects such as lists and sets) the actual value on the stack is a pointer to an object in the heap. This simple strategy permits dynamically variable length objects to be implemented easily. Thus arbitrarily long lists can be formed. It is this too which provides the enhanced extents needed to pass such compound objects out of scope and beyond the lifetime of their creators. It is clear, then, that all data objects are represented by some value on the stack, and each is necessarily the same size.

The interpreter, in implementing the abstract machine, is wholly responsible for management of the heap. The compiler knows only of the stack, and is primarily concerned with parsing and the generation of abstract code. It is essential that the real and virtual stacks remain in phase as they follow the block structure of the language. When a

block is left the stack is retracted to where it was prior to block entry, thus disposing of space allocated to local variables in that block instance.

Any block which ends with an expression produces a value. This is at the top of the stack at block exit, so it is copied down to the new stack top, ie to where the stack was before block entry. The compiler knows whether or not it wants the top of the stack to be copied down at this time, and it informs the interpreter via part of the return instruction.

The compiler is totally ignorant of the details of the implementation of lists. It sees a list simply as a value on the stack, and as we have seen all stack items are of the same size, so a list must appear to be a normal value on the stack. The interpreter, on finding a list on the stack (deduced from its type tag), is responsible for tracing its components in the heap. A list or a set is held as a bound mass of values corresponding to the individual components, each tagged with its type. Thus lists, sets and strings behave as normal objects on the stack. As Protocol is a clean language strings, sets and lists can be passed around and returned as block expressions. This presents no problem because we use the heap.

Because Protocol upholds the principle of data type completeness strings, lists or sets can be passed as messages between instances, so they might be accessible to many concurrent instances at once. It is therefore important

that operations on them leave them unchanged from the viewpoint of the other instances using them. Update operations on any string, list or set produce new values. Lists and sets are loops in the language's data space, not loops back into the denotation space; that is they are composed of values, not locations holding values.

10.3.3 The Instruction Set.

The abstract machine designed to support the Protocol system has its own instruction set. It is these, together with the storage management and i/o subsystem, which must be simulated on a real computer in order to implement the system.

There is a single instruction format. This is 16-bits long, with 7-bits holding the actual operation code, giving 128 in all, and some 9-bits of additional encoded information.

Some operations, mainly jump instructions, carry a code address (offset in the abstract code stream) in this information field. Such simple, one-word jump instructions are therefore limited to a 512 word abstract code jump. Although this sounds rather limited, the abstract code, being quite 'high-level' is quite compact, and so a 512 word jump range is actually sufficient.

As Protocol is an interactive system its routines are likely to be very small. If however, longer jumps are needed, then auxiliary two-word jump instructions can be used which carry the code address in the following word. Here though this is

not done.

Other instructions carry stack addresses, but most use this instruction field to carry type or other flags to the interpreter.

Many of the instructions are closely related to the state of the stack. There are various instructions which load items on to the stack. Most load values, of various types; one loads addresses, which are actually stack offsets to other locations.

There are several jump instructions, which branch either forward or backward depending on whether the stack top is true or false, and can do so either leaving the stack as it is, or popping the top element automatically, so making things a little simpler for oft encountered sequences. Similarly, there are two instructions specifically related to the for-statement, stepping and testing the control constant, which help speed up matters in such well defined iterative sequences. Most of these instructions, although commonly used, are derived directly from the work of Morrison [M6], and are to be found in the S-algol abstract stack machine [B2].

Although it would serve little purpose to describe in detail each instruction in the repertoire of the abstract Protocol machine, we will elaborate on those which relate to the creation, closedown and interaction of the concurrent instances which make up the system.

There are four such instructions:

startup

exit

receive

send

and we shall take each in turn.

Startup And Exit.

Loosely speaking, the startup instruction, the "new" operator, is executed by an instance in order to cause a thus-far dormant instance in the system to start executing some definition. An instance runs until it either aborts, is aborted or completes. It completes simply by 'dropping out' of the end of its definition. The last instruction in any definition is exit. This informs the machine that this instance is closing down so that its space can be reallocated. It also causes the system to recognise that it will not be able to accept any more messages.

Interactions.

Messages are sent asynchronously, directly between instances. This is called an interaction. The two message oriented instructions handle all of the various possible combinations of message passing, from simple sends or receives to the more complex chaining operations.

These interactions are syntactically of the form:

value "_" exp

for sending a message (of arbitrary complexity), and

```
"(" [ "first" | "latest" ) value [ "_" exp ] ")"
```

which requests the next message from the given source be read in (in this case the optional send will be performed first). The more complex interactions are:

```
chain ":" ( value | "->" names )
```

where

```
chain = interaction { ":" interaction }
```

and

```
interaction = [ "first" | "latest" ] value [ "_" exp ]
```

A few examples should suffice:

```
world _ "a message"

eval    world
eval    message
send( trailing )
```

```
input : world _ "was the value received"

eval    input
receive( first )
eval    world
eval    message
send( IOS, trailing )    ! for IOS see below !
```

```
input _ "prompt" : world _ ".. as above"

eval    input
eval    message
send( trailing then setup )
receive( first )
eval    world
eval    message
send( IOS, trailing )
```

```
display _ rootcrate : select _ N : table : world

eval    display
```

```
eval    message
send( trailing then setup )
receive( first )
eval select
eval    message
send( IOS and trailing then setup )
receive( first )
eval    table
send( IOS then setup )
receive( first )
eval    world
send( IOS only )
```

The send and receive instructions correspond to the isolated sends "_" (as in the first case), chained requests ":" (without sending some precursor message) as in the second case for "input", and in the 'combined operations' (of sending to and immediately requesting from a single instance) as in each of the "_" and ":" formations. As can be seen from the sample fragments of abstract code, in the complete ":" case the compiler actually plants pairs of instructions, first to send then to receive a message from the given target. In such cases this might well be optimised to a single, heavily encoded instruction for the entire interaction.

The receive instruction finds a value on the stack which evaluates the source of the message. Its lower field carries a flag indicating whether it is the first or the latest message from that source that is required. The value on the stack is replaced by the acquired value. A simple send instruction takes the message value itself above a target, and both are removed from the stack after the operation.

The send instruction also handles the 'merging' of the

message, travelling left to right across the chain, and any other message, 'trailing' behind the "_" of the interaction. For example, in:

input : world _ "was the value received"

the request (":") from "input" will produce a message, and this will be 'merged' into a list along with the 'trailing' "was the value received", before this is sent ("_") off to "world".

In this combined operation the following sequence of actions takes place. Firstly the value "input" is stacked. A call to request is made. The request is performed. The instance will pick up the first (default) value in its message queue which matches the desired source. If none is currently in the queue, then the instance will sleep until after such a message arrives. Having acquired the desired message, this will replace the instance value on the stack.

The next part of the chain will then be performed, in the knowledge that there is already an 'item on the stack', or IOS for short. This next operation loads the second occurrence of "world" onto the stack. It also loads the trailing message value, and then send is called. This is informed that the IOS condition is true and so it merges the value below the instance on the stack with the trailing value above the instance value. This new message is left on the top, replacing the trailing message as such. The send is then performed, dispatching the merged message to the instance value one down the stack. As it leaves the send

instruction tidies up the stack by removing however many items were on the stack. In this case it would be three. If there had just been a trailing message, or just a chained message then it would have been two. After this combined operation the IOS condition is false.

If the chain had been longer, as in the earlier parts of the later examples, then the send would have left its instance value on the top of the stack, for immediate use by the following receive operation. This would push the acquired value, leaving IOS true for the next interaction (or end-of-chain assignment) to manage.

The simple receive instruction corresponds to the form:

"(" ["first" | "latest"] value ["_" exp] ")"

which, after an optional initial send, receives a message from the given source. Since the send is performed via a trivial call to interact (with IOS false and trailing message) which clears the stack, we duplicate the instance value if there is a "_" following it. This preserves it for the subsequent receive operation. Since this "("...")" format produces a value to form an expression, this is not really part of the sugared chained syntax. Note that the "(" and ")" brackets form a (split) operator, and are not simply parsing brackets. Note that the following:

```
( input _ "??" ) : world

eval    input
duplicate
eval    message
send( trailing )
```

```
receive( first )
receive( first )
eval    world
send( IOS only )
```

is not the same as:

```
input _ "??"  :  world

eval    input
eval    message
send( trailing then setup )
receive( first )
eval    world
send( IOS only )
```

because whereas the second form prompts the "input" instance with "??" and then performs a request from this same instance before sending the acquired value to world, the first form prompts "input" with "??" and receives a message from it before using the received value itself as the source from which to acquire the value to be sent to world.

As a programming convenience the compiler permits the omission of the "new" operator whenever it is 'obvious' that a definition is to be coerced into a new instance value. The send instruction might well find itself given a definition, rather than an actual instance as its destination. In such cases the send calls on the startup instruction, after beautifying the stack to suit that expected by the startup, and then it uses the newly created instance for the transaction. This instance will therefore remain essentially anonymous to the user, who neglected to store it for later use. Such 'coerced' instances are only capable of single interactions with their creators, unless the creator takes the trouble to interrogate the system to determine its

companions.

A somewhat similar situation can arise in the case of the receive instruction. When it is given a definition value, rather than an instance value, it means that the user wishes (either the first or the last) input from any instance executing that particular definition. It will scan the queue looking for such a source, and if it does not find one then it will await such an arrival (actually it suspends itself).

Because the interactions are very high level, even in the abstract machine, the code is very compact. For example a chain with half a dozen sends and receives, involving a total of five instances, will only be of the order of 26 bytes of abstract code.

Conclusions.

The raw instructions are not, of course, actually available to the user, who need not know of the stack underlying the computation. They are generated by the compiler as it parses his high level text. This is considered to be better than presenting the user with a naked stack machine of the same power, as seems to have occurred in the very popular, but difficult to use, 'threaded' [see D2 or D7 for this technique] Forth [M4,J1] system. It is more important that the user concentrate on the logic of his program, than be needlessly concerned with the mapping of its computation on a stack. The same stack machine can, no doubt, be more effectively, and safely used, from the distance of a higher level input language.

This is all the more true when we consider the machine instructions which approach the level of the Protocol system itself, such as the startup & exit instructions for spawning and clearing away new instances in the system, which carry out all of the housekeeping and storage management associated with such activity. Likewise the send & receive instructions for building and transmitting messages within the system, are too complex for hand-coding.

In addition there are various instructions related to the building and stripping of lists, and others for sets and strings. There is a plethora of instructions which enable the instances in the system to interrogate their input queues, and each other's status. All these are relatively high-level instructions which one would not find on simpler sequential machines.

In the main, however, apart from the storage management, which is totally dynamic, the tagged architecture for dynamic type matching, and the more abstract instructions mirroring the Protocol system the abstract machine is essentially a simple stack machine, and is easily implemented.

11.0 References.

A1 Ammann U.
"The Method Of Structured Programming Applied To The Development Of A Compiler"
Proc. Int. Comp. Symp. pp93-99 (1973)
North-Holland Publ. 1973

A2 Andler S.
"Synchronisation Primitives And The Verification Of Concurrent Programs"
Carnegie-Mellon CS Report (1977)

A3 Andrews G.R. and McGraw J.R.
"Language Features For Process Interaction"
Proc. ACM Conf "Language Design For Reliable Software"
March 1977 in ACM SIGPLAN vol.12 pp114-127 (1977)

A4 Apt K.R., Francez N. and De Roever W.P.
"A Proof System For Communicating Sequential Processes"
ACM TOPLAS vol.2 pp359-385 (1980)

A5 Atkinson L.V. and McGregor J.J.
"CONA - A Conversational Algol System"
Software vol.8 pp699-708 (1978)

B1 Backus J.
"Can Programming Be Liberated From The von Neumann Style? A Functional Sytle And Its Algebra Of Programs"
CACM vol.21 pp613-641 (1978)

B2 Bailey P.J., Maritz P. and Morrison R.
"The Abstract S-algol Machine"
CS/80/- St.Andrews (1980)

B3 Balzer R.M.
"Dataless Programming"
AFIPS vol.31 pp535-544 (19??)

B4 Balzer R.M.
"PORTS - A Method For Dynamic Interprogram Communication And Job Control"
AFIPS vol.38 pp485-489 (1971)

B5 Barnes J.G.P.
"An Overwiew Of Ada"
Software vol.10 pp851-887 (1980)

B6 Barnett J.K.R.
"The Design Of An Inter-Task Communication Scheme"
Software vol.10 pp801-816 (1980)

B7 Barron D.W. et al
"The Main Features Of CPL"
Comp. J. pp134-143 (1963)

B8 Battarel G.J. and Chevance R.J.
"Design Of A High-Level Machine"
AFIP NCC 1979 pp649-655 (1979)

B9 Bell J.R.
"Threaded Code"
CACM vol.16 (6) pp370-372 (1973)

B10 Bernstein A.J.
"Output Guards And Nondeterminism In 'Communicating Sequential Processes' "
ACM TOPLAS vol.2 pp234-238 (1980)

B11 Berry D.M.
"Language Design Methods Based On Semantic Principles: - A remark of RD Tennants paper. Algol-68, A Language Designed Using Semantic Principles"
Acta.Inf vol.15 pp83-98 (1981)

B12 Bochmann G.V.
"Compile Time Memory Allocation For Parallel Processes"
IEEE. Trans.SE. vol.SE-4 pp517-520 (1978)

B13 Bornat R.
"Understanding and Writing Compilers"
Macmillan Press. Ltd. (1979)

B14 Bos J. van den
"Comments On ADA.Process Communication"
ACM SIGPLAN vol.15 (6) p77 (1980)

B15 Bourne S.R.
"The UNIX Shell"
Bell Systems Technical Journal vol.57 (6/2) pp1971-1990 (1978)

B16 Brinch Hansen P.
"The Architecture Of Concurrent Programs"
Prentice-Hall Publ. (1977)

B17 Brinch Hansen P.
"Universal Types In Concurrent Pascal"
Inf.Proc.Lett vol.3 p165 (1975)

B18 Brinch Hansen P.
"The Nucleus Of A Multiprogramming System"
CACM vol.13 pp238-242 (1970)

B19 Brinch Hansen P.
"The Purpose Of Concurrent Pascal"
ACM SIGPLAN vol.10 (6) pp305-309 (1975)

B20 Brinch Hansen P.
"A Programming Methodology For Operating System Design"

In "Information Processing 74", Rosenfeld (Ed.)
North Holland, Amsterdam, p394 (1974)

B21 Brinch Hansen P.
"Structured Multiprogramming"
CACM vol.15 pp574-578 (1972)

B22 Brinch Hansen P.
"The Programming Language Concurrent Pascal"
IEEE. Trans. SE. vol.SE-1 (2) pp199-207 (1975)

B23 Brinch Hansen P.
"Experience With Modular Concurrent Programming"
IEEE. Trans. SE. vol.SE-3 (2) pp156-159 (1977)

B24 Brinch Hansen P.
"Distributed Processes : A Concurrent Programming Concept"
CACM vol.21 (11) pp934-941 (1978)

B25 Brinch Hansen P.
"Concurrent Programming Concepts"
Comp.Surveys vol.5 (4) pp223-245 (1973)

B26 Brinch Hansen P.
"Edison : A Multiprocessor Language"
Software vol.11 pp325-361 (1981)

B27 Brinch Hansen P.
"The Design Of Edison"
Software vol.11 pp363-396 (1981)

C1 Campbell R.H. and Habermann A.N.
"Specification Of Process Synchronisation By Path Expressions"
In "Operating Systems" - Lecture Notes In Computer Science 16, 1974.

C2 Celentano A.
"Incremental LR Parsers"
Acta.Inf vol.10 pp307-321 (1978)

C3 Cheatham T.E.
"The Recent Evolution Of Programming Languages"
IFIP'71 pp118-134 (1971)

C4 Chu Y.
"High-Level Language Computer Architectures"
Academic Press Publ. (1975) - Editor

C5 Chu Y. and Cannon E.R.
"A Programming Language For High-Level Architectures"
AFIP NCC 1979 pp657-665 (1979)

C6 Coffman E.G., Elphick M.J. and Shoshani A.
"System Deadlocks"
Computing Surveys vol.3 pp67-78 (1971)

C7 Cohen D.
"Flow Control For Real-Time Communications"
ACM SIGCCR vol.10 pp41-47 (1980)

C8 Coleman D. et al
"An Assessment Of Concurrent Pascal"
Software vol.9 pp827-837 (1979)

D1 Dahl O., and Nygaard K.
"SIMULA - An Algol-Based Simulation Language"
CACM vol.9 pp671-678 (1966)

D2 Dakin R.J. and Poole P.C.
"A Mixed Code Approach"
Comp.J. vol.16 (3) pp219-222 (1972/3)

D3 Dawson J.L.
"Combining Interpretive Code With Machine Code"
Comp.J. vol.16 (3) pp216-219 (1972/3)

D4 Dembinski P. and Schwartz R.
"The Taming Of The Pointer"
ACM SIGPLAN vol.?? pp60-73 (1977)

D5 Denning P.J.
"Why Not Innovations In Computer Architecture?"
ACM SIGCAN vol.8 (2) p4-7 (1980)

D6 Denning P.
"Third Generation Computer Systems"
Comp. Surveys vol.3 pp175-216 (1971)

D7 Dewar R.B.K.
"Indirect Threaded Code"
CACM vol.18 (6) pp330-331 (1975)

D8 Dijkstra E.W.
"The Humble Programmer"
CACM vol.15 pp859-866 (1972)

D9 Dijkstra E.W.
"Hierarchical Ordering Of Sequential Processes"
In "Operating Systems Techniques" Academic Press (1972), and
Acta.Inf. vol.1 pp115-138 (1971)

D10 Dijkstra E.W.
"The Structure Of 'THE' Multiprogramming System"
CACM vol.11 pp341-346 (1968)

D11 Dijkstra E.W.
"Cooperating Sequential Processes"
In "Programming Languages" pp43-112 (1968)
Academic Press Publ.

D12 Donahue J.E..
"Locations Considered Unnecessary"
Acta.Inf. vol.8 pp221-242 (1977)

D13 Doran R.W
"Architecture Of Stack Machines"
In "High-Level Language Computer Architecture"
Edited By Chu pp63-109 (1975)

E1 ESONE
"Real-Time Basic for CAMAC"
ESONE/RTB/02 (1977)

E2 ESONE
"CAMAC - A Modular Instrumentation System For Data Handling"
EUR 4100e (march 1969), TID-25875.

E3 ESONE
"The Definition Of I.M.L. - A Language For Use In CAMAC Systems"
ESONE/IML/01, TID-26615.

E4 Ekanadham K. and Mahjoub A.
"Microcomputer Networks"
The Computer Journal vol.24 pp17-24 (1981)

E5 Elsworth E.F.
"Compilation Via An Intermediate Language"
Comp.J. vol.22 pp226-233 (1979)

E6 Eventoff W., Harvey D. and Price R.J.
"The Rendezvous And Monitor Concepts: Is There An Efficiency Difference?"
ACM SIGPLAN vol.15 pp156-165 (1980)

F1 Fabry R.S.
"Capability-Based Addressing"
CACM vol.17 (7) pp403-412 (1974)

F2 Feiler P.H. and Medina-Mora R
"An Incremental Programming Environment"
CMU-CS-80-126, Carnegie-Mellon Univ. (1980)

F3 Feustel E.A.
"On The Advantages Of Tagged Architecture"
IEEE. Trans. Computers. Vol.C.22 pp644-656 (1973)

F4 Feustel E.A.
"The Rice Research Computer - A Tagged Architecture"
AFIPS SJCC vol.40 pp369-377 (1972)

F5 Foster J.M.
"A Syntax Improving Device"
Comp. J. vol.11 pp31-34 (1968)

F6 Frailey D.J.

"Innovations In Microprocessor Architecture - Another View"
ACM SIGCAN vol.7 ppl1-13 (1979)

F7 Freeman M., Jacobs W.W. and Levy L.S.
"On The Construction Of Interactive Systems"
AFIPS NCC 1978 pp555-562 (1978)

G1 Gentleman W.M.
"Message Passing Between Sequential Processes"
Software vol.11 pp435-466 (1981)

G2 Ghezzi C. and Mandrioli D.
"Augmenting Parsers To Support Incrementality"
JACM vol.27 (3) pp 564-579 (1980)

G3 Ghezzi C and Mandrioli D.
"Incremental Parsing"
ACM TOPLAS vol.1 (1) pp58-70 (1979)

G4 Goodenough J.B.
"Exception Handling: Issues And A Proposed Notation"
CACM vol.18 pp683-696 (1975)

G5 Gordon M.J.C.
"The Denotational Description Of Programming Languages"
Springer-Verlag Publ. (1979)

G6 Gries D.
"Compiler Construction For Digital Computers"
Wiley Publ. (1971)

G7 Gunn H.I.E.
"hil Reference Manual"
CS/80/3 - St.Andrews (1980)

G8 Gunn H.I.E. and Morrison R.
"On The Implementation Of Constants"
Inf.Proc.Lett. vol.9 (1) p1-4 (1979)

G9 Gunn H.I.E. and Harland D.M.
"Constancy In Programming Languages"
Inf.Proc.Lett. In Press (1981)

H1 Habermann A.N.
"Synchronisation Of Communicating Processes"
CACM vol.15 (3) pp171-176 (1972)

H2 Haddon B.K.
"Nested Monitor Calls"
ACM SIGOSR vol.11 (10) (1977)

H3 Haridi S., Bauner J-O., and Svensson G.
"An Implementation And Empirical Evaluation Of

The Tasking Facilities In Ada"
ACM SIGPLAN vol.16 pp35-47 (1981)

H4 Harland D.M.
"Concurrency In A Langauge Employing Messages"
Inf.Proc.Lett. vol.12 pp59-63 (1981)

H5 Harland D.M.
"On Facilities For Interprocess Communication"
Inf.Proc.Lett. vol.12 pp221-226 (1981)

H6 Harland D.M.
"On Facilities For Handling Exceptions And Preventing Deadlock In A System Of Concurrent Processes"
Inf.Proc.Lett. (Submitted 1981)

H7 Harland D.M.
"The Role Of Machines In Distributed Systems"
Inf.Proc.Lett. (Submitted 1981)

H8 Hehner E.C.R.
"On Removing The Machine From The Language"
Acta.Inf. vol.10 pp229-243 (1978)

H9 Heller G., Kneis W., Rembold U. and Wiesner G.
"Standards And Proposals Of Industrial Real-Time FORTRAN"
Ann.Rev.Auto.Prog. vol.9 pp95-107 (1980)

H10 Henderson P.
"An Approach To Compile Time Type Checking"
IFIP'77 pp523-527 (1977)

H11 Hewitt C.
"Viewing Control Structures As Patterns Of Passing Messages"
AI vol.8 pp323-364 (1977)

H12 Higley
"Type Checking In A Typeless Language"
Comp. J p106 (1976)

H13 Hoare C.A.R.
"Monitors : An Operating System Structuring Concept"
CACM vol.17 p549 (1974)

H14 Hoare C.A.R.
"An Axiomatic Basis For Computer Programming"
CACM vol.12 pp576-583 (1969)

H15 Hoare C.A.R.
"Communicating Sequential Processes"
CACM vol.21 pp666-677 (1978)

H16 Holt R.C. et al
"Structured Concurrent Programming With Operating

System Applications"
Addison-Wesley Publ. (1978)

H17 Howard J.H.
"Proving Monitors"
CACM vol.19 pp273-279 (1976)

I1 I.B.M.
"PL/I Language Reference Manual"
(1969)

I2 Ichbiah J.D. et al
"Rationale for the design of the ADA programming language"
ACM SIGPLAN vol.14 (6) (1979)

J1 James J.S.
"FORTH For Microcomputers"
ACM SIGPLAN vol.13 pp33-39 (1978)

J2 Jamnel A.J. and Steigler H.G.
"Managers vs Monitors"
IP'77 p827 (1977)

J3 Jamnel A.J. and Steigler H.G.
"On Expected Costs Of Deadlock Detection"
Info.Proc.Lett vol.11 pp229-231 (1980)

J4 Jones D.W.
"Tasking and Parameters : A Problem Area In Ada"
ACM SIGPLAN vol.15 (5) p37 (1980)

K1 Kahrs M.
"Implementation Of An Interactive Programming System"
ACM SIGPLAN vol.14 pp76-82 (1979)
Symposium on "Compiler Construction".

K2 Kaubisch W.H., Perrott R.H. and Hoare C.A.R.
"Quasi-Parallel Programming"
Software vol.6 p341 (1976)

K3 Keedy J.J.
"On Structuring Operating Systems With Monitors"
ACM SIGOSR vol.13 (1) pp5-9 (1979)
and in Aust.Comp.J. vol.10 pp23-27 (1978)

K4 Kernighan B.W. and Plauger P.J.
"Software Tools"
Addison-Wesley Publ. (1976)

K5 Kernighan B.W. and Ritchie D.M
"The C Programming Language"
Prentice-Hall Publ. (1978)

K6 Kessels J.L.W.
"An Alternative To Event Queues For Synchronisation

In Monitors"
CACM vol.20 (7) pp500-503 (1977)

K7 Kieburtz R.B.
"Programming Without Pointer Variables"
ACM Conf on "Data Abstraction, Definition & Structure"
(1976)

K8 Knuth D.E.
"Top Down Syntax Analysis"
Acta.Inf. vol.1 pp79-110 (1971)

K9 Koblitz W.
"Real-Time BASIC: A Suitable Tool For
Process Control"
Ann.Rev.Auto.Prog. vol.9 pp61-66 (1980)

K10 Koblitz W. et al
"Industrial Real-Time FORTRAN"
EURO-IFIP pp21-25 North-Holland Publ. (1979)

K11 Kratzer G. and Schrott G.
"Interfacing Real-Time Operating Systems To
Process Control Languages"
Ann.Rev.Auto.Prog. vol.9 pp1-16 (1980)

K12 Krull F.N.
"Experience With ILIAD: A High-Level Process
Control Language"
CACM vol.24 pp66-72 (1981)

L1 Lagally K.
"Synchronisation In Layered Systems"
Lect.Notes In Op. Sys. - An Adv. Course p254

L2 Lampson B.W. and Redell D.D.
"Experience With Processes And Monitors In Mesa"
CACM vol.23 (2) pp105-117 (1980)

L3 Landin P.J.
"A Correspondence between Algol-60 and Church's
Lambda-Notation"
CACM vol.8 (2) and (3) pp89-101 and pp158-165 (1965)

L4 Landin P.J.
"The Next 700 Programming Languages"
CACM vol.9 (3) pp157-164 (1966)

L5 Landin P.J.
"The Mechanical Evaluation Of Expressions"
Comp.J. vol.6 (4) pp308-320 (1964)

L6 Ledgard H.F. and Marcotty M.
"A Genealogy Of Control Structures"
CACM vol.18 pp629-639 (1975)

L7 Levin R.

"Program Structures For Exceptional Condition Handling"
Ph.D. Thesis, Computer Science, Carnegie-Mellon (1977) [See summary in Wand,1979]

L8 Lewis A.
"Definition Of The CAMAC Intermediate Language"
ESONE SWG-20 (1973)

L9 Liskov B.
"Primitives For Distributed Computing"
ACM SIGOSR vol.?? pp33-42 (1979)

L10 Lister A.M.
"The Problem Of Nested Monitor Calls"
ACM SIGOSR vol.11 (2) (1977)

L11 Lister A.M. and Sayer P.J.
"Hierarchical Monitors"
Software vol.7 p613 (1977)

L12 Lister A.M. and Maynard K.J.
"An Implementation Of Monitors"
Software vol.6 pp377-386 (1976)

L13 Lohr K.
"Beyond Concurrent Pascal"
Proc. 6th ACM Symp. "Operating System Principles" in ACM SIGOSR vol.11 (5) pp173-180 (1977)

L14 Lomet D.B.
"Process Structuring, Synchronisation, And Recovery Using Atomic Actions"
ACM SIGPLAN vol.12 pp128-137 (1977)

L15 Lucas P.
"Formal Definition Of Programming Languages And Systems"
IFIP'71 pp110-116 (1971)

L16 Luckham D.C. and Polak W.
"Ada Exception Handling: An Axiomatic Approach"
ACM TOPLAS vol.2 pp225-233 (1980)

L17 Lycklama H. and Bayer
"The MERT Operating System"
Bell Systems Techinical J. [UNIX] vol.57 (6/2) pp2049-2086 (1978)

M1 Mahjoub A.
"Some Comments On Ada As A Real-Time Programming Language"
ACM SIGPLAN vol.16 pp89-95 (1981)

M2 McCarthy J. et al
"LISP 1.5 Programmer's Reference Manual"
MIT Press (1965)

M3 McMahan L.N. and Feustel E.A.
"Implementation Of A Tagged Architecture For Block Structured Languages"
ACM/IEEE Symposium "High-Level Language Computer Architectures" in ACM SIGARCH/SIGPLAN Issue pp91-100 Nov 1973

M4 Moore C.H.
"FORTH : A New Way To Program A Minicomputer"
Astronomy & Astrophysics Suppl. vol.15 pp497-511 (1974)

M5 Morrison R.
"S-algol Reference Manual"
CS/79/1 - St.Andrews (1979)

M6 Morrison R.
"On The Development Of Algol"
Ph.D Thesis St.Andrews (1980)

M7 Morrison R.
"A Method Of Implementing Procedure Entry And Exit In Block Structured High-Level Languages"
Software vol.?? pp537-539 (197?)

M8 Myers G.J.
"Advances In Computer Architecture"
McGraw-Hill Publ. (1978)

M9 Myers G.J. and Buckingham B.R.S.
"A Hardware Implementation Of Capability-Based Addressing"
ACM SIGCAN vol.8 (6) pp12-24 (1980)

N1 Narayana K.T., Prasad V.R and Joseph M.
"Some Aspects of Concurrent Programming In CCNPASCAL"
Software vol.9 p 749 (1979)

N2 Naur P. et al
"Revised Report On The Algorithmic Language Algol-60"
CACM vol.6 (1) pp1-17 (1963)

N3 Nehmer J.
"The Implementation Of Concurrency For A PL/I-like Language"
Software vol.9 pp1043-1057 (1979)

N4 Neumann J. von., Burks A.W. and Goldstein H.H.
"Preliminary Discussion Of An Electric Instrument"
Inst. For Advanced Studies. Princeton (1947)

N5 Nuclear Enterprises Ltd.
"CAMAC Product Range" - Sales and Technical Info

O1 Organick E.I.
"Computer System Organisation - B5700/6700 Series"

Publ. Academic Press (1973)

P1 Parnas D.L.
"The Non-Problem Of Nested Monitor Calls"
ACM SIGOSR vol.12 (1) pp12-14 (1978)

P2 Parnas D.L.
"On A 'Buzzword' - Hierarchical Structure"
IFIP'74 pp336-339 North-Holland Publ. (1974)

P3 Peccoud M. et al
"Incremental Interactive Compilation"
IFIP '68 ppB33-B37 (1968)

P4 Popplestone R.J.
"The Design Philosophy Of Pop-2"
In "Machine Intelligence 3" edited by Michie D.

P5 Prasad V.R.
"Variable Numbers of Parameters In Typed Languages"
Software vol.10 pp507-518 (1980)

P6 Pyle I.C.
"I/O In High Level Programming Languages"
Software vol.9 pp907-914 (1979)

R1 Radue J.E. and Mullins J.M.
"Solving Synchronisation Problems Using Semaphores"
Software vol.5 pp51-64 (1975)

R2 Ramsperger N.
"Concurrent Access To Data"
Acta.Inf. vol.8 pp324-334 (1977)

R3 Reynolds J.C.
"Definitional Interpreters for higher-order programming languages"
Proc 27th ACM Nat. Conf. pp717-740 (1972)

R4 Reynolds J.C.
"GEDANKEN : A Simple Typeless Language Based On The Principles Of Completeness And The Reference Concept"
CACM vol.13 (5) pp??-?? (1970)

R5 Richards M.
"BCPL : A Tool For Compiler Writing And System Programming"
AFIPS SJCC pp557-566 (1969)

R6 Richards M., Aylward A.R., Bond P., Evans R.D. and Knight B.J.
"TRIPOS - A Portable Operating System"
Software vol.9 pp513-526 (1979)

R7 Ritchie D.M.
"The UNIX I/O System"
UNIX Documentation (level-6). Bell Systems.

R8 Ritchie D.M. and Thompson K.
"The UNIX Time-sharing System"
CACM vol.17 pp365-475 (1974)

R9 Russell B.
"On An Equivalence Between Continuation And Stack Semantics"
Acta.Inf. vol.8 pp113-123 (1977)

S1 Sale A.H.J.
"Addition Of Repeat And Until As Identifiers"
ACM SIGPLAN vol.16 pp98-103 (1981)

S2 Schild R. and Lienhard H.
"Real-Time Programming In PORTAL"
ACM SIGPLAN vol.?? pp79-92 (1980)

S3 Schmid H.A.
"On The Efficient Implementation Of Conditional Critical Regions And The Construction Of Monitors"
Acta.Inf. vol.6 pp227-249 (1967)

S4 Schuller G.
"Conversation Among Processes"
?? pp460-469 (197?)

S5 Scott D. and Strachey C.
"Towards A Mathematical Semantics For Computer Languages"
In "Computers And Automata" pp19-46 (1972)
Wiley Publ.

S6 Silberschatz A.
"On The Access-Control Mechanism Of The Program Component Manager"
Software vol.11 pp159-166 (1981)

S7 Silberschatz A.
"On The Synchronisation Mechanism Of The Ada Language"
ACM SIGPLAN vol.16 pp96-103 (1981)

S8 Silberschatz A.
"Port Directed Communication"
The Computer Journal vol.24 pp78-82 (1981)

S9 Silberschatz A., Kieburtz R. and Bernstein A.
"Extending Concurrent Pascal To Allow Dynamic Resource Management"
IEEE. Trans. SE. vol.SE-3 (3) pp210-217 (1977)

S10 Stephens C.L.
"A Consumer's Guide To CAMAC Software"
ESO/SRC (UK) pp76-92 (1978) Geneva
"Applications Of CAMAC To Astronomy"

S11 Strachey C.
"Varieties Of Programming Languages"
In "High Level Languages" Infotech State Of The Art Report No.7

S12 Strachey C.
"Fundamental Concepts In Programming Languages"
OU PRG (197?)

S13 Strachey C.
"Towards A Formal Semantics"
In "Formal Language Description Languages"
North-Holland (1966)

S14 Strachey C. and Wadsworth C.P.
"Continuations - A Mathematical Semantics For Handling Full Jumps"
OU PRG-11 (1974) Oxford.

S15 Stroet J.
"An Alternative To The Communication Primitives In Ada"
ACM SIGPLAN vol.15 pp62-74 (1980)

T1 Tennent R.D.
"Language Design Methods Based On Semantic Principles"
Acta.Inf. vol.8 pp97-112 (1977)

T2 Tennent R.D.
"The Denotational Semantics Of Programming Languages"
CACM vol.19 (8) pp437-453 (1976)

T3 Thompson K.
"The UNIX Command Language"
In "Structured Programming" Infotech pp375-384 (1976)

T4 Turner D.A.
"Error Diagnosis & Recovery In One Pass Compilers"
Inf.Proc.Lett vol.6 (4) pp113-115 (1977)

T5 Turner D.A.
"Programs That Are Better - In Any Language"
Datalink 8/9/80

T6 Turner D.A. and Campbell R.W.
"SASL Language Reference Manual"
CS/-/79 (Revised)

W1 Wand I.C.
"Systems Implementation Languages and IRONMAN"
Software vol.9 pp853-878 (1979)

W2 Wand I.C.
"Dynamic Resource Allocation And Supervision With The Programming Language MODULA"
Computer J. vol.23 pp147-152 (1980)

W3 Ward S.A. and Halstead R.H.
"A Syntactic Theory Of Message Passing"
Journal ACM vol.27 pp365-383 (1980)

W4 Wegner P.
"Data Structure Models For Programming Languages"
ACM SIGPLAN vol.6 pp1-54 (1971)

W5 Weinberg G.M.
"The Psychology Of Computer Programming"
Van Nostrand Reinhold Publ. (1971)

W6 Welsh J. and Bustard D.W.
"Pascal-Plus = Another Language For Modular Multiprogramming"
Software vol.9 p947 (1979)

W7 Welsh J. and McKeag M.
"Structured System Programming"
Publ. Prentice-Hall International

W8 Welsh J. and Lister A. M.
"A Comparative Study Of Task Communication In Ada"
Software vol.11 pp257-290 (1981)

W9 Welsh J., Lister A.M. and Salzman E.J.
"A Comprison Of Two Notations For Process Communication"
In "Language Design And Programming Languages"
Lecture Notes in Comp.Sci No.79 (1980) Tobias (ed.)

W10 Wettstein H.
"The Implementation Of Synchronisation Operations In Various Environments"
Software vol.7 pp115-126 (1977)

W11 Wettstein H. and Merbeth G.
"The Concept Of Asynchronisation"
ACM SIGOSR vol.14 pp50-70 (1980)

W12 Whitby-Strevens C., May M.D., Taylor R. and Booth T.
"The Distributed Computing Research Project"
Collected Papers,
Dept Computer Science, Warwick (April 1978)

W13 Wijngaarden van et al
"Revised Report on the Algorithmic Language Algol-68"
Acta.Inf. vol.5 pp1-236 (1975)

W14 Wilner W.T.
"Design Of The B1700"
AFIPS vol.41 pp489-498 (1972)

W15 Wilner W.T.
"B1700 Memory Utilisation"

AFIPS vol.41 pp579-589 (1972)

W16 Wirth N.
"Modula : A Language For Modular Multiprogramming"
Software vol.7 p3 (1977)

W17 Wirth N.
"The Programming Language Pascal"
Acta.Inf. vol.1 pp35-63 (1971)

W18 Wirth N.
"On The Design Of Programming Languages"
IFIP-74 North-Holland Publ. pp386-393 (1974)

W19 Wirth N.
"What Can We Do About The Unnecessary Diversity Of Notation For Syntactic Definition?"
CACM (november) (1977)

W20 Wirth N.
"Toward A Discipline Of Real-Time Programming"
CACM vol.20 (8) pp577-583 (1977)

W21 Wulf W.A., Russell D.B. and Habermann A.N.
"BLISS : A Language For System Programming"
CACM vol.14 (12) pp780-790 (1971)

W22 Wulf W.A. et al
"Reflections On A System Programming Language"
ACM SIGPLAN vol.6 (9) pp42-49 (1971)