A FRAMEWORK FOR THE EVALUATION OF HIGH-LEVEL LANGUAGES FOR DISTRIBUTED COMPUTING

by

Michael L. Scott

Computer Sciences Technical Report #563

October 1984

A Framework for the Evaluation of High-Level Languages for Distributed Computing

Michael L. Scott

Department of Computer Sciences
University of Wisconsin - Madison
1210 W. Dayton
Madison, WI 53706

October 1984

ABSTRACT

The development of appropriate high-level languages for the expression of distributed algorithms is an active area of current research. This paper presents a framework within which such languages may be fruitfully contrasted and compared. Specifically, it attempts to organize and clarify the major design decisions involved in the creation of programming languages suitable for the writing of distributed programs for a multicomputer. It then examines several recent proposals, showing how they fit into the framework, and describing the novel features of each. The concluding section contains a limited number of suggestions for future research.

This work was supported in part by NSF grant number MCS-8105904, by Arpa contract number N0014/82/C/2087, and by a Bell Telephone Laboratories Doctoral Scholarship.

CONTENTS

1. Introduction

It has been recognized for some time that certain algorithms (operating systems in particular) are most elegantly expressed by **concurrent programs** in which there are several independent and, at least in theory, simultaneously active threads of control. On the assumption that the threads interact by accessing shared data, a whole body of research has evolved around methods for synchronizing that access [15, 16, 23, 26, 27, 38, 42, 43]. Even on a conventional uni-processor, effective synchronization is crucial in the face of context switches caused by interrupts.

With the development of multi-computers it has become practical to distribute computations across multiple machines. This prospect has lent a new urgency to the study of **distributed programs** — concurrent programs in which separate threads of control may run on separate physical machines. There are two reasons for the urgency:

(1) On a multi-computer, a distributed program may solve a problem substantially faster than its sequential counterpart

(2) The systems programs for a multi-computer must by their very nature be distributed

Unfortunately, there is no general consensus as to what language features are most appropriate for the expression of distributed algorithms. Shared data is no longer the obvious approach, since the underlying hardware supports message passing instead. The alternatives proposed to date show a remarkable degree of diversity. This paper attempts to deal with that diversity by developing a framework for the study of distributed programming languages. The framework allows us to compare existing languages for semantic (as opposed to purely cosmetic) differences, and it allows us to explore new and genuinely different possibilities.

Section 2 presents the framework. Section 3 uses that framework to evaluate a number of existing languages. No attempt is made to survey techniques for managing shared data (good surveys have appeared elsewhere [4]). The evaluations are intentionally biased towards languages that lend themselves to implementation on top of a distributed operating system, where message passing is the only means of process interaction.

2. The Framework

This section discusses what seem to me to be the major issues involved in distributed language design:

- processes and modules
- communication paths and naming
- synchronization
- implicit and explicit message receipt
- message screening and multiple rendezvous
- miscellany: shared data, asynchronous receipt, timeout, reliability

The list is incomplete. The intent is to focus on those issues that have the most profound effects on the flavor of a language or about which there is the most controversy in the current literature.

2.1. Processes and Modules

A **process** is a logical thread of control. It is the working of a processor, the execution of a block of code. A process is described by a state vector that specifies its position in its code, the values of its data, and the status of its interfaces to the rest of the world.

A **module** is a syntactic construct that encapsulates data and procedures. A module is a closed scope. It presents a limited interface to the outside world and hides the details of its internal operation.

In a sense, a module is a logical computer and a process is what that computer does. Several language designers have chosen to associate exactly one process with each module, confusing the difference between the two. It is possible to design languages in which there may be more than one process within a module, or in which a process may travel between modules. Such languages may pretend that the processes within a module execute concurrently, or they may acknowledge that the processes take turns. In the latter case, the language semantics must specify the circumstances under which execution switches from one process to another. In the former case, the language must provide some other mechanism for synchronizing access to shared data.

Modules are static objects in that they are defined when a program is written. Some languages permit them to be nested like Algol blocks; others insist they be disjoint. In some cases, it may be possible to create new **instances** of a module at run time. Separate instances have separate sets of data.

Some languages insist that the number of processes in a program be fixed at compile time. Others allow new ones to be created during execution. Some languages insist that a program's processes form a hierarchy, and impose special rules on the relationships between a process and its descendants. In other languages, all processes are independent equals. A process may be permitted to terminate itself, and perhaps to terminate others as well. It will usually terminate automatically if it reaches the end of its code.

2.2. Communication Paths

The most important questions about a distributed language revolve around the facilities it provides for exchanging messages. For want of a better term, I define a **communication path** to be something with one end into which senders may insert messages and another end from which receivers may extract them. This definition is intentionally vague. It is meant to encompass a wide variety of language designs.

Communication paths establish an equivalence relation on messages. Senders assign messages to classes by **naming** particular paths (see section 2.3). Receivers accept messages according to class by **selecting** particular paths (see section 2.6.1). Messages sent on a common path enjoy a special relationship. Most languages insert them in a queue, and guarantee receipt in the order they were sent. Some languages allow the queue to be reordered.

One important question is most easily explored in terms of the abstract notion of paths: how

many processes may be attached to each end? There are four principal options:[1]

(1) Many senders, one receiver — This is by far the most common approach. It mirrors the client/server relationship found in many useful algorithms: a server (receiver) is willing to handle requests from any client (sender). A single server caters to a whole community of clients. Of course, a server may provide more than one service; it may be on the receiving end of more than one path. Separate paths into a receiver are commonly called **entry points**. In theory, one could get by with a single entry point per server. The advantage of multiple entries is that they facilitate message screening (see section 2.6.1) and allow for strict type checking on each of several different message formats. From an implementor's point of view, multiple entry points into a single receiver are handled in much the same way as multiple senders on a single communication path

(2) One sender, many receivers — This approach is symmetric to that in (1). It is seldom used, however, because it does not reflect the structure of common algorithms.

(3) Many senders, many receivers — This is the most general approach. In its purest form it is very difficult to implement. The problem has to do with the maintenance of bookkeeping information for the path. In the one-receiver approach, information is conveniently stored at the receiving end. In the one-sender approach, it is kept at the sending end. With more than one process at each end of the path, there is no obvious location. If we store all information about the status of the path on a single processor, then all messages will end up going through that intermediary, doubling the total message traffic. If we attempt to distribute the information, we will be led to situations in which either a) a sender must (at least implicitly) query all possible receivers to see if they want its message, or b) a receiver must query all possible senders to see if they have any messages to send.

    Neither option is particularly desirable. Protocols exist whose communication requirements are linear in the number of possible pairs of processes [10, 21], but this is generally too costly. One way out is to restrict the model by insisting that multiple processes on one end of a path reside on a single physical machine. This approach is taken by several languages: messages are sent to *modules*, not processes, and any process within the module may handle a message when it arrives.

(4) One sender, one receiver — This approach is the easiest to implement, but is acceptable only in a language that allows programmers to refer conveniently to arbitrary sets of paths. In effect, such a language allows the programmer to "tie" a number of paths together, imitating one of the approaches above.

The preceeding descriptions are based on the assumption that each individual message has exactly one sender and exactly one receiver, no matter how many processes are attached to each end of the communication path. For some applications, it may be desirable to provide a **broadcast** facility that allows a sender to address a message to *all* the receivers on a path, with a single operation. Several modern network architectures support broadcast in hardware. Unfortunately, they do not all guarantee reliability. Broadcast will be complex and slow whenever acknowledgments must be

[1] These four options correspond, respectively, to the distributed operating system concepts of input ports, output ports, free ports, and bound ports. I have avoided this nomenclature because of the conflicting uses of the word "port" by various language designs.

returned by each individual receiver

Several language and operating system designers have attempted to implement *send* and *receive* as symmetric operations (see in particular sections 3.4 and 4.3.2). Despite their efforts, there remains an inherent *asymmetry* in the sender/receiver relationship: data flows one way and not the other. This asymmetry accounts for the relative uselessness of one-many paths as compared to many-one. It also accounts for the fact that no one even discusses the symmetric opposite of broadcast: a mechanism in which a receiver accepts identical copies of a message from all the senders on a path at once.

2.3. Naming

In order to communicate, processes need to be able to **name** each other, or at least to name the communication paths that connect them. Names may be established at compile time, or it may be necessary to create them dynamically. The issue of naming is closely related to the above discussions of processes, modules, and communication paths. Several comments should be made:

— In the typical case of many senders/one receiver, it is common for the sender to name the receiver explicitly, possibly naming a specific path (entry) into the receiver if there is more than one. Meanwhile the receiver specifies only the entry point, and accepts a message from anyone on the other end of the path.

— Compiled-in names can only distinguish among things that are distinct at compile time. Multiple instantiations of a single block of code will require dynamically-created names.

— In languages where messages are sent to modules, it may be possible for names (of module entry points) to be established at compile time, even when the processes that handle messages sent to the module are dynamically created. Processes within a module may be permitted to communicate with each other via shared data.

Several naming strategies appropriate for use among independent programs in a distributed operating system are not generally found in programming language proposals. Finkel [33] suggests that processes may refer to each other by capabilities, by reference to the facilities they provide, or by mention of names known to the operating system. These approaches allow for much looser coupling than one would normally expect among the pieces of a single program. They may be appropriate in languages designed to support communication among processes that have been developed independently [73].

2.4. Synchronization

Since all inter-process interaction on a multi-computer is achieved by means of messages, it is neither necessary nor even desirable for a language to provide synchronization primitives other than those inherent in the facilities for communication. The whose question of synchronization can be treated as a sub-issue of the semantics of the *send* operation [24, 33, 58]. There are three principal possibilities:[2]

[2] In any particular implementation, the process of sending a message will require a large number of individual steps. Conceivably, the sender could be unblocked after any one of those steps. In terms of programming language semantics, however, the only steps that matter are the ones that are visible to the user-level program.

(1) No-wait send — In this approach the sender of a message continues execution immediately, even as the message is beginning the journey to wherever it is going. The operating system or run-time support package must buffer messages and apply back-pressure against processes that produce messages too quickly. If a communication error occurs (for example, the intended recipient has terminated), it may be quite difficult to return an error code to the sender, since execution may have proceeded an arbitrary distance beyond the point where the *send* was performed.

(2) Synchronization send — In this approach the sender of a message waits until that message has been received before continuing execution. Message traffic may increase, since the implementation must return confirmation of receipt to the sender of each message. Overall concurrency may decline. On the other hand, it is easy to return error codes in the event of failed transmission. Furthermore, there is no need for buffering or back-pressure (though messages from separate processes may still need to be queued on each communication path).

(3) Remote invocation send — In this approach the sender of a message waits until it receives an explicit reply from the message's recipient. The name "remote invocation" is meant to suggest an analogy to calling a procedure: the sender transmits a message (*in* parameters) to a remote process that performs some operation and returns a message (*out* parameters) to the sender, who may then continue execution. The period of time during which the sender is suspended is referred to as a **rendezvous**. For applications in which it mirrors the natural structure of the algorithm, remote invocation send is both clear and efficient. Both the original message and the (non-blocking) reply carry useful information; no unnecessary confirmations are involved. As Liskov [58] points out, however, many useful algorithms cannot be expressed in a natural way with remote invocation.

The choice of synchronization semantics is one of the principle areas of disagreement among recent language proposals. Section 3 includes examples of all three strategies.

2.5. Implicit and Explicit Message Receipt

Cashin [24] discusses a duality between "message-oriented" and "procedure-oriented" inter-process communication. Rather than semantic duals, I maintain that his approaches are merely varying syntax for the same underlying functionality. What is at issue is whether message receipt is an *explicit* or an *implicit* operation.

In the former case, an active process may deliberately receive a message, much as it might perform any other operation. In the latter case, a procedure-like body of code is activated automatically by the arrival of an appropriate message. Either approach may be paired with any of the three synchronization methods.

Implicit receipt is most appropriate when the functions of a module are externally driven. An incoming message triggers the creation of a new process to handle the message. After the necessary operations have been performed, the new process dies. Alternately, one may think of the message as awakening a sleeping process that performs its operations and then goes back to sleep, pending arrival of another message. There may be one such "sleeping process" for each of the module's entry procedures, or it may be more convenient to imagine a single sleeper capable of executing any of the entries. If remote invocation send is used, it may be intuitive to think of the "soul" of a

sender as traveling along with its message. This soul then animates the receiving block of code, eventually returning to its original location (along with the reply message), and leaving that code as lifeless as before. Each of these options suggests a different implementation.

Implicit receipt is a natural syntax for the client/server model. It is better suited than the explicit approach to situations in which requests may arrive at unpredictable times or in which there is no obvious way to tell when the last message has arrived. Explicit receipt, on the other hand, is more appropriate for situations that lack the client/server asymmetry. It is useful for expressing coroutines and similar applications in which communication is among active, cooperating peers. Typically both parties have useful work to do between conversations. An obvious example is a producer/consumer pair in which both the creation of new data and the consumption of old are time-consuming operations.

The choice of syntax for message receipt is a second major area of disagreement among recent language proposals (Synchronization was the first). Most languages employ one or the other, though StarMod (section 3.11) and NIL (section 3.16) provide both.

2.6. Details of the *Receive* Operation

As noted above, most languages permit multiple senders, but only one receiver on each communication path. In addition, they typically allow a process to be non-deterministic in choosing the entry point it wants to service next: instead of having to specify a particular path, a receiver is free to accept messages from any of a variety of paths on which they may be present. By comparison, *send* is relatively simple.[3] This section discusses two of the more important aspects of the *receive* operation.

2.6.1. Message Screening

Assume for the moment that a process may form the receiving end of several communication paths. Further, assume that each of these paths may carry a variety of messages from a variety of senders. In a completely non-deterministic situation, a receiver might be expected to cope with any message from any process on any path. This burden is usually unacceptable. A process needs to be able to exercise control over the sorts of messages it is willing to accept at any particular time. It needs to qualify its non-deterministic options with **guards** that specify which options are open and which are currently closed.

2.6.1.1. Semantics

There is a wide range of options for message screening semantics. Every language provides some means of deciding which message should be received next. The fundamental question is: what factors may be taken into account in reaching the decision? The simplest approach is to "hard-code" a list of open paths. In effect, this approach allows the decision to be made at compile time. Most languages, however, allow at least part of the decision to be made at run time. Usually, the

[3] Among the languages discussed in section 3, CSP/80 [48] is the only exception to this rule. Though it permits only a single sender and receiver on each communication path, the language allows both senders and receivers to choose among several alternative paths, depending on whether anyone is listening on the other end. This added flexibility entails implementation problems similar to those discussed in section 2.2 (3). For a more complete discussion of CSP, see section 3.4.

programmer will specify a Boolean condition that must evaluate to "true" before a particular message will be accepted. The question now becomes, on what may the Boolean condition depend? It is not difficult to implement guards involving only the local variables of the receiver. Complications arise when a process tries to base its choice on the contents of the incoming messages. In most languages, messages arriving on a particular communication path are ordered by a queue. In a few cases, it may be possible to reorder the queues. In any case, a simple implementation is still possible if path selection or queue ordering depends on some particular well-known **slot** of the incoming message. PLITS and ZENO for example, allow a process to screen messages by sender name (path) and **transaction** slot (see section 3.7).

In the most general case, a language may permit a receiver to insist on predicates involving arbitrary fields of an incoming message. The implementation then has no choice but to go ahead and receive a message sight unseen, then look at its contents to see if it really should have done so. Unless unwanted messages can be returned to their sender, the receiver may require an arbitrary amount of buffer space.

2.6.1.2. Syntax

The precise way in which guards are specified depends largely on the choice between implicit and explicit message receipt. With implicit receipt, there are two basic options:

(a) The language may allow an entry procedure to suspend itself while waiting for an arbitrary Boolean expression to become true.

(b) The language may provide **condition queues** or **semaphores** on which an entry procedure may suspend itself, assuming that some other procedure will release it when it is safe to proceed.

The first approach is the more general of the two. The second is easier to implement, and is generally more efficient. Brinch Hansen discusses the tradeoffs involved [19] (pp. 15-21). Both approaches assume that an entry procedure may suspend itself at any time, and is thus free to examine an incoming message before doing so. Since the messages will differ from one instance to the next, separate activation records will be required for each suspended entry. Campbell and Habermann [23] suggest the simpler (and more restrictive) approach of allowing guards to involve local data only, and of insisting they occur at the very beginning of their entry procedures. A language that took such an approach would be able to avoid the separate activation records. It would also be less expressive.

Guards are more straightforward with explicit receipt. Nonetheless, a fair amount of syntactic variety is possible. The most common approach looks something like a Pascal *case* statement, with separate clauses for each possible communication path. Each clause may be proceeded by a guard. The physical separation of clauses allows messages of different types to be received into different local variables. In a language with looser message typing (for example PLITS and ZENO, of section 3.7), there may be a statement that specifies receipt into a single variable from any of a set of open paths. An ordinary sequential *case* statement then branches on some field of the message just received.

mechanism for synchronizing access to the data those processes may share.

(3) Timeout and related issues — In most proposals employing synchronization or remote invocation send, the sender of a message may be suspended indefinitely if no one is willing to listen to it. Likewise a process that attempts to receive a message may have to wait forever if no one sends it anything. Such delays may be acceptable in a distributed program where communication patterns are carefully defined and each process is able to assume the correctness of the others. In certain real-time applications, however, and in language systems that attempt to provide for reliability under various sorts of hardware failure, it may be desirable to provide a mechanism whereby a process that waits "too long" times out and is able to take some sort of corrective action.

One particular sort of timeout is especially useful, and may be provided even in cases where the more general facility is not. By specifying a timeout of zero, a process can express its desire to send or receive a message only when such a request can be satisfied immediately, that is when some other process has already expressed its willingness to form the other end of the conversation.

(4) Robustness — Where hardware reliability is in doubt, or when a program is expected to respond in a reasonable fashion to unpredictable real-time events, the language may need to provide for error detection and recovery. Liskov's Extended CLU and Argus (section 3.8) are noteworthy examples. The problems involved in providing for reliability in distributed programs have not been adequately investigated. Like many others, I ignore them.

(5) Unreliable send — In certain applications, particularly in the processing of real-time data, speed may be more important than reliability. It may be more appropriate to send new data than to resend messages that fail. For such applications, a language may provide fast but unreliable messages. If aknowledgments are not required, it may be possible to provide a **broadcast** service at no more cost than point-to-point communication.

3. Several Languages

This section surveys more than two dozen distributed language proposals. For each, it describes how the language fits into the framework of section 2, and then mentions any features that are particularly worthy of note. Languages are considered in approximate order of their publication. For those without the patience of a saint, I particularly recommend the sections on monitor languages, CSP, Distributed Processes, Argus, and Ada.

3.1. Path Expressions

Path Expressions [23, 39] are more of a mechanism than a language. They were invented by Campbell and Habermann in the early 1970's to overcome the disadvantages of semaphores for the protection of shared data. Rather than trust programmers to insert **P** and **V** operations in their code whenever necessary, the designers of path expressions chose to make synchronization rules a part of the declaration of each shared object.

The path expression proposal makes no mention of modules, nor does it say much about the nature of processes. It specifies only that processes run asynchronously, and that they interact solely

2.6.2. Multiple Rendezvous

In a language using remote invocation send, it is often useful for a receiver to be in rendezvous with more than one sender at a time. One ingenious application involves a process scheduler [18, 64]. The scheduler has two entry points: *schedule-me* and *I'm-done*. Every process with work to do calls *schedule-me*. The scheduler remains in rendezvous with all of these callers but one. While that caller works, the scheduler figures out which process P has the next-highest priority. When the worker calls *I'm-done*, the scheduler ends its rendezvous with P.

In a language with both remote invocation send and implicit message receipt, a module may be in rendezvous with several senders at one time. If each entry procedure runs until it blocks, then the module is a monitor [43]. If the implementation time-slices among entries, or if it employs a multi-processor with common store, then the language must provide additional mechanisms for controlling access to the module's common data.

Multiple rendezvous is also possible with explicit message receipt. Several languages require the *receive* and *reply* statements to be paired syntactically, but allow the pairs to nest. In such languages the senders in rendezvous with a single receiver must be released in LIFO order. If senders are to be released in arbitrary order, then the *reply* (or *disconnect*) statement must be able to specify which rendezvous to end. Mutual exclusion among the senders is not an issue, since only one process is involved on the receiving end. Mao and Yeh [64] note that careful location of a *disconnect* statement can minimize the amount of time a sending process waits, leading to higher concurrency and better performance. Similar tuning is not generally possible with implicit receipt: senders are released implicitly at the end of entry procedures. It would be possible to provide an explicit *disconnect* with implicit receipt, but it would tend to violate the analogy to sequential procedure calls.

2.7. Side Issues

The issues discussed in this section are less fundamental than those addressed above. They fall into the category of convenient "extra features"— things that may or may not be added to a language after the basic core has been designed.

(1) Shared data — In order to permit reasonable implementations on a multi-computer, a distributed language must in general insist that all interaction among processes be acheived by means of messages. For the sake of efficiency, however, a language may provide for shared access to common variables by processes guaranteed to reside on the same physical machine. It may be necessary to provide additional machinery (semaphores, monitors, critical regions, etc.) to control "concurrent" access.

(2) Asynchronous receipt — Several communication schemes place no bound on the length of time that can pass before a message is noticed by the process to which it was sent. There is certainly no such bound for explicit receipt. There are times, however, when it is desirable to receive data as soon as it becomes available. One solution is to provide so-called **immediate** procedures [33] — special entry procedures that guarantee immediate execution. The existence of immediate procedures implies that multiple processes may be active in the same module. Moreover, it implies that execution may switch from one process to another at unpredictable times. As noted in section 2.1, the language will need to provide some special

by invoking the **operations** provided by shared objects. Like the monitors described below, path expressions can be forced into a distributed framework by considering a shared object to be a passive entity that accepts requests and returns replies. Under this model, the proposal uses remote invocation send with implicit message receipt. Communication paths are many-one. There may be several identical objects. Processes name both the object and the operation when making a request.

The declaration of a shared object specifies three things: the internal structure of the object, the operations that may be invoked from outside and that are permitted to access the internal structure, and the **path expressions** that govern the synchronization of invocations of those operations. There is no convenient way to specify an operation that works on more than one object at a time.

A path expression describes the set of legal sequences in which an object's operations may be executed. Syntactically, a path expression resembles a regular expression. "**(A, B); {C}; D**" for example, is a path expression that permits a single execution of either A or B (but not both), followed by one or more simultaneous executions of C, followed in turn by a single execution of D. There is no restriction on which executions may be performed on behalf of which processes.

The original path expression proposal [23] permitted Kleene closure only for expressions as a whole; internal pieces could not be repeated an arbitrary number of times. Operations could call each other, but no operation could appear on more than one path. Later modifications to the proposal [39] relaxed these restrictions and proposed additional syntax for bounding the difference in the number of executions of parts of a path. The original proposal included a proof that path expressions and semaphores are equally powerful: each can be used to implement the other.

Robert and Verjus [70] have suggested an alternative syntax for path expressions. Like Campbell and Habermann, they dislike scattering synchronization rules throughout the rest of the code. They prefer to group the rules together in a **control module** that authorizes the executions of a set of **operations**. Their synchronization rules are predicates on the number of executions of various operations that have been requested, authorized, and/or completed since the module was initialized. Their solutions to popular problems are both straightforward and highly intuitive.

3.2. Monitor languages

Monitors were suggested by Dijkstra [27], developed by Brinch Hansen [16], and formalized by Hoare [43] in the early 1970s. Like path expressions, monitors were intended to regularize the access to shared data structures by simultaneously active processes. The first languages to incorporate monitors were Concurrent Pascal [17], developed by Brinch Hansen, and SIMONE [51], designed by Hoare and his associates at Queen's University, Belfast. Others include SB-Mod [11], Concurrent SP/k [45, 46], Mesa [57], Extended BCPL [63], Pascal-Plus [78], and Modula [80]. Of the bunch, Concurrent Pascal, Modula, and Mesa have been by far the most influential. SIMONE and C-SP/k are strictly pedagogical languages. Pascal-Plus is a successor to SIMONE. SB-Mod is a dialect of Modula.

In all the languages, a **monitor** is a shared object with operations, internal state, and a number of **condition** queues. Only one operation of a given monitor may be active at a given point in time. A process that calls a busy monitor is delayed until the monitor is free. On behalf of its calling process, any operation may suspend itself by *wait*ing on a queue. An operation may also *signal* a queue, in which case one of the waiting processes is resumed, usually the one that waited first.

Several languages extend the mechanism by allowing condition queues to be ordered on the basis of **priorities** passed to the *wait* operation. Mesa has an even more elaborate priority scheme for the processes themselves.

Monitors were originally designed for implementation on a conventional uni-processor. They can, however, be worked into a distributed framework by considering processes as active entities capable of sending messages, and by considering monitors as passive entities capable of receiving messages, handling them, and returning a reply. This model agrees well with the semantics of Concurrent Pascal and SIMONE, where monitors provide the *only* form of shared data. It does not apply as well to other languages, where the use of monitors is optional. Distributed implementations would be complicated considerably by the need to provide for arbitrary data sharing.

Concurrent Pascal, SIMONE, E-BCPL, and C-SP/k have no modules. In the other four languages surveyed here, monitors are a special kind of module. Modules may nest. In Modula and SB-Mod, the number of modules is fixed at compile time. In Pascal-Plus and Mesa, new instances may be created dynamically. Pascal-Plus modules are called **envelopes**. They have an unusually powerful mechanism for initialization and finalization. Modules in SB-Mod are declared in hierarchical levels. Inter-module procedure calls are not permitted from higher to lower levels. SIMONE, C-SP/k, and Pascal-Plus provide built-in mechanisms for simulation and the manipulation of pseudo-time.

Concurrent Pascal and C-SP/k programs contain a fixed number of processes. Neither language allows process declarations to nest, but Concurrent Pascal requires a hierarchical ordering (a DAG) in which each parent process lists explicitly the monitors to which its children are permitted access. In the six other languages, new processes can be created at run time. Process declarations may be nested in Pascal-Plus, and the nesting defines an execution order: each parent process starts all its children at once and waits for them to finish before proceeding. In Mesa, process instances are created by *fork*ing procedures. Mesa compounds the problems of shared data by allowing arbitrary variables to be passed to a process by reference. Nothing prevents an inner procedure from passing a local variable and then returning immediately, deallocating the variable and turning the reference into a dangling pointer.

Under the distributed model described above, monitor languages use remote invocation send with implicit receipt. Communication paths are many-one. In languages that permit multiple monitors with identical entries (Concurrent Pascal, Pascal-Plus, and Mesa), the sender must name both the monitor and entry. It also names both in SIMONE, but only because the bare entry names are not visible under Pascal rules for lexical scope. In E-BCPL the sender calls the monitor as a procedure, passing it the name of the operation it wishes to invoke.

The precise semantics of mutual exclusion in monitors are the subject of considerable dispute [4, 40, 47, 52, 55, 62, 67, 79]. Hoare's original proposal [43] remains the clearest and most carefully described. It specifies two bookkeeping queues for each monitor: an **entry** queue, and an **urgent** queue. When a process executes a *signal* operation from within a monitor, it waits in the monitor's urgent queue and the first process on the appropriate condition queue obtains control of the monitor. When a process leaves a monitor it unblocks the first process on the urgent queue or, if the urgent queue is empty, it unblocks the first process on the entry queue instead.

These rules have two unfortunate consequences:

(1) A process that calls one monitor from within another and then waits on a condition leaves the outer monitor locked. If the necessary signal operation can only be reached by a similar nested call, then deadlock will result.

(2) Forcing the signaller to release control to some other waiting process may result in a prohibitive number of context switches. It may also lead to situations in which the signaller wakes up to find that its view of the world has been altered unacceptably.

One solution to the first problem is to release the locks on the outer monitors of a nested *wait*. This approach requires a means of restoring the locks when the waiting process is finally resumed. Since other processes may have entered the outer monitors in the intervening time, those locks might not be available. On a uni-processor, the problem can be solved by requiring all operations of all monitors to exclude one another in time. Outer monitors will thus be empty when an inner process is resumed. Most of the languages mentioned here use global monitor exclusion. The exceptions are Concurrent Pascal, Mesa, and SB-Mod.

Concurrent Pascal and Mesa provide a separate lock for each monitor. Nested calls leave the outer levels locked. SB-Mod provides a lock for each set of monitors whose data are disjoint. There are two forms of inter-monitor calls. One leaves the calling monitor locked, the other leaves it unlocked. Neither affects monitors higher up the chain. A process that returns across levels from a nested monitor call is delayed if the calling monitor is busy.

The second problem above can be addressed in several ways. Modula [81], SB-Mod, E-BCPL, and C-SP/k all reduce the number of context switches by eliminating the urgent queue(s). Careful scheduling of the uni-processor takes the place of mutual exclusion. In general, process switches occur only at *wait* and *signal* operations, and not at module exit.[4] When the current process signals, execution moves to the first process on the appropriate condition queue. When the current process waits, execution may move to any other process that is not also waiting.[5] A process that would have been on one of Hoare's entry queues may well be allowed to proceed before a process on the corresponding urgent queue.

Concurrent Pascal requires that signal operations occur *only* at the *end* of monitor routines. There is thus no need for an urgent queue. To simplify the implementation, Concurrent Pascal allows only one process at a time to wait on a given condition. Mesa relaxes these restrictions by saying that a signal is only a *hint*. The signaller does not relinquish control. Any process suspended on a condition queue must explicitly double-check its surroundings when it wakes up; it may find it cannot proceed after all, and has to wait again. Wettstein [79] notes that if signals are only hints then it is indeed feasible to release exclusion on all the monitors involved in a nested wait (though Mesa does not do so). Before continuing, a signalled process could re-join each of the entry queues, one by one. After regaining the locks it would check the condition again.

---

[4] E-BCPL timeslices among the runnable processes. Clock interrupts are disabled inside monitor routines. SB-Mod reschedules processes in response to hardware interrupts, but the interrupts are masked at all levels below that of the current process. Interrupted processes are resumed when the current process attempts to return to a lower interrupt level.

[5] The next process to run after a *wait* is always the next runnable process on a circular list. All processes stay on the list in Modula, SB-Mod, and E-BCPL. Their order is fixed. In C-SP/k, waiting processes are removed from the list, eventually to be re-inserted behind their signaller.

Kessels [55] suggests a different approach to the semantics of conditions. If every queue is associated with a pre-declared Boolean expression, then the *signal* operation can be dispensed with altogether. When a process leaves a monitor, the run-time support package can re-evaluate the Boolean expressions to determine which process to run next.

SB-Mod expands on Kessel's proposal. The Boolean expressions for condition queues are optional. *Wait* suspends the caller if the expression is false or was not provided. *Send* (*signal*) transfers control to the first process on the queue if the expression is true or was not provided. A new operation called "*mark*" sets a flag in the first process on the queue. When the current process leaves its monitor, the queue is re-examined. If the expression is true or was not provided, then the marked process is moved to the ready queue. No process switch occurs.

Of all the languages surveyed, SIMONE is truest to Hoare. It does not provide separate entry queues for every monitor, but it does provide an urgent *stack*, with processes resumed in LIFO order.

3.3. Extended POP-2

Kahn and MacQueen [50] have implemented a small but elegant language based on a generalization of **coroutines**. Their language has much in common with CSP (section 3.4, below) but was developed independently.

Process declarations in Extended POP-2 look very much like procedures. There are no modules. Processes share no data. They are instantiated with a *cobegin* construct called "*doco*." The *doco* statement uses a series of **channels** to connect input and output **ports** in the newly-created processes.

Once running, processes can communicate by means of *put* and *get* operations on ports. Given the binding to channels achieved by *doco*, communication paths are one-one. *Send* is non-blocking and buffered. *Receive* is explicit, and names a single port. There is no provision for non-deterministic or selective receipt. Processes with a single input and a single output port may be instantiated with a special functional syntax.

3.4. Communicating Sequential Processes

CSP [44] is not a full-scale language. Rather, it is an ingenious proposal by C. A. R. Hoare for the syntactic expression of non-determinism and inter-process communication. CSP/80 [48], Extended CSP [6], occam [65], and a nameless language by Roper and Barter [71] are all attempts to expand Hoare's syntax into a usable language. I will refer to Extended CSP as E-CSP and to Roper and Barter's language as RB-CSP.

Processes are the central entities in CSP. There are no modules. Regular CSP, E-CSP, occam, and RB-CSP all allow new processes to be created at run time with a modified *cobegin* construct. CSP/80 provides for a fixed number of independent processes, statically defined. Subprocesses in E-CSP and RB-CSP are not visible to their parent's peers. Messages from outside are addressed to the parent. The parent redirects them to the appropriate child. To avoid ambiguity, the E-CSP compiler guarantees that no two subprocesses ever communicate with the same outsider. RB-CSP performs the equivalent checks at run time. None of the CSP languages supports recursion.

Disjoint processes in CSP do not share data; all interaction is by means of a generalization of the traditional concepts of **input** and **output**. In regular CSP, and in CSP/80 and occam, the result is equivalent to explicit receipt and synchronization send. E-CSP provides both synchronization and no-wait send. RB-CSP uses only no-wait send.

Communication paths in CSP are one-one; both sender and receiver name the process at the other end. Forcing the receiver to name the sender prevents the modeling of common client/server algorithms. It also precludes the use of libraries. The four implementations mentioned here address the problem in different ways. CSP/80 lets processes send and receive through **ports**. Sender ports and receiver ports are bound together in a special linking stage. Occam processes send and receive messages through **channels**. Any process can use any channel that is visible under the rules of lexical scope. E-CSP and RB-CSP provide **processname** *variables*. An E-CSP receiver still specifies a sender, but the name it uses can be computed at run time. An RB-CSP receiver does not specify the sender at all. It specifies a message **type** and must be willing to receive from any sender with a matching type.

Communication is typeless in regular CSP and in occam. Types are associated with ports in CSP/80. They are associated with individual communication statements in E-CSP. Individual input and output commands match only if their types agree. RB-CSP provides a special type constructor called **message** with named **slots**, much like those of PLITS (section 3.7). A given process need only be aware of the slots it may actually use.

CSP incorporates Dijkstra's non-deterministic guarded commands [28]. A special kind of guard, called an **input guard**, evaluates to true only if a specified input command can proceed immediately. In regular CSP, and in E-CSP and RB-CSP, there is no corresponding *output* guard to test whether a process is waiting to receive. Hoare notes that the lack of output guards makes it impossible to translate certain parallel programs into equivalent, sequential versions. CSP with input guards alone can be implemented by the usual strategy for many-one communication paths (see section 2.2): information is stored at the receiving end. The provision of output guards as well leads to the usual problems of many-many paths (for a discussion, see the appendix of Mao and Yeh's paper on communication ports [64]). Moreover, as noted by the designers of CSP/80, the indiscriminate use of both types of guards can lead to implementation-dependent deadlock. Nonetheless, CSP/80 does provide both input and output guards. Special restrictions are imposed on the nature of paths to eliminate the deadlock problem.

3.5. Distributed Processes

In the design of Distributed Processes [18], Brinch Hansen has unified the concepts of processes and modules and has adapted the monitor concept for use on distributed hardware.

A Distributed Processes program consists of a fixed number of modules residing on separate logical machines. Each module contains a single process. Modules do not nest. Processes communicate by calling **common procedures** defined in other modules. Communication is thus by means of implicit receipt and remote invocation send. Data can be shared between common procedures, but not across module boundaries.

An entry procedure is free to block itself on an arbitrary Boolean condition. The main body of code for a process may do likewise. Each process alternates between the execution of its main

code and the servicing of external requests. It jumps from one body of code to another only when a blocking statement is encountered. The executions of entry procedures thus exclude each other in time, much as they do in a monitor. Nested calls block the outer modules: a process remains idle while waiting for its remote requests to complete. There is a certain amount of implementation cost in the repeated evaluation of blocking conditions. Brinch Hansen argues convincingly that the cost is acceptable, and well justified by the added flexibility.

3.6. Gypsy

Gypsy [37] was designed from the start with formal proofs in mind. Programs in Gypsy are meant to be verified routinely, with automatic tools.

Much of Gypsy, including its block structure, is borrowed from Pascal [49]. There is no notion of modules. New processes are started with a *cobegin* construct. The clauses of the *cobegin* are all procedure calls. The procedures execute concurrently. They communicate by means of **buffer** variables, passed to them by reference. Since buffers may be accessible to more than one process, communication paths are many-many. Sharing of anything other than buffers is strictly forbidden. There is no global data, and no objects other than buffers can be passed by reference to more than one process in a *cobegin*.

Buffers are bounded FIFO queues. Semantically, they are defined by **history** sequences that facilitate formal proofs. *Send* and *receive* are buffer operations. *Send* adds an object to a buffer. *Receive* removes an object from a buffer. *Send* blocks if the buffer is full. *Receive* blocks if the buffer is empty. In the nomenclature of section 2, Gypsy uses no-wait send and explicit receipt, with the exception that back-pressure against prolific senders is part of the language definition. Declared buffer lengths allow the synchronization semantics to be independent from implementation details.

A variation of Dijkstra's guarded commands [28] allows a process to execute exactly one of a number of *sends* or *receives*. The *await* statement contains a series of clauses, each of which is guarded by a *send* or *receive* command. If none of the commands can be executed immediately, then the *await* statement blocks until a buffer operation in some other process allows it to proceed. If more than one of the commands can be executed, a candidate is chosen at random. There is no general mechanism for guarding clauses with Boolean expressions.

3.7. PLITS and ZENO

PLITS [30] is an acronym for "Programming Language in the Sky," an ambitious attempt at advanced language design. In the area of distributed computing, it envisions a framework in which a computation may involve processes written in *multiple languages*, executing on heterogeneous machines. ZENO [7] is a single language based heavily on the PLITS design. Its syntax is borrowed from Euclid [56].

A ZENO program consists of a collection of modules that may be instantiated to create processes. Processes are assigned names at the time of their creation. They are independent equals. A process dies when it reaches the end of its code. It may die earlier if it wishes. It cannot be killed from outside. There is no shared data. *Receive* is explicit. *Send* is non-blocking and buffered. There is only one path into each module (process), but each message includes a special **transaction**

slot to help in selective receipt. A sender names the receiver explicitly. The receiver lists the senders and transaction numbers of the messages it is willing to receive. There is no other means of message screening — no other form of guards. As in CSP (section 3.4), forcing receivers to name senders makes it difficult to write servers. A "pending" function allows a process to determine whether messages from a particular sender, about a particular transaction, are waiting to be received.

The most unusual feature of PLITS/ZENO lies in the structure of its messages. In contrast to most proposals, these languages do not insist on strong typing of inter-process communication. Messages are constructed much like the property lists of LISP [69]. They consist of name/value pairs. A process is free to examine the message slots that interest it. It is oblivious to the existence of others.

In keeping with its multi-language, multi-hardware approach, PLITS prohibits the transmission of all but simple types. ZENO is more flexible.

Recent extensions to PLITS [29] are designed to simplify the organization of large distributed systems and to increase their reliability. Cooperating processes are tagged as members of a single **activity**. A given process may belong to more that one activity. It enjoys a special relationship with its peers, and may respond automatically to changes in their status. Activities are supported by built-in atomic transactions, much like those of Argus (section 3.8).

3.8. Extended CLU and Argus

Extended CLU [58, 59] is designed to be suitable for use on a long-haul network. It includes extensive features for ensuring reliability in the face of hardware failures, and provides for the transmission of abstract data types between heterogeneous machines [41]. The language makes no assumptions about the integrity of communications or the order in which messages arrive.

The fundamental units of an Extended CLU program are called **guardians**. A guardian is a module; it resides on a single machine. A guardian may contain any number of processes. Guardians do not nest. Processes within the same guardian may share data. They may use monitors for synchronization. All interaction among processes in separate guardians is by means of message passing.

Receive is explicit. *Send* is non-blocking and buffered. Each guardian provides **ports** to which its peers may address messages. New instances of a guardian may be created at run time. New port names are created for each instance. The sender of a message specifies a port by name. It may also specify a **reply** port if it expects to receive a message in response. The reply port name is really just part of the message, but is singled out by special syntax to enhance the readability of programs. Within a guardian, any process may handle the messages off any port: processes are *anonymous* providers of services. A facility is provided for non-deterministic receipt, but there are no guards; a receiver simply lists the acceptable ports. In keeping with the support of reliability in the face of communication failure, a timeout facility is provided.

Argus [60, 61] is the successor to Extended CLU. Argus uses remote invocation send and implicit message receipt. Instead of ports, Argus guardians provide **handlers** their peers may invoke. Processes are no longer anonymous in the sense they were in Extended CLU. Each invocation of a handler causes the creation of a new process to handle the call. Additional processes may be created within a guardian with a *cobegin*-like construct.

Argus programs achieve robustness in the face of hardware failures with **stable storage** and an elaborate **action** mechanism. Actions are *atomic*: they either **commit** or **abort**. If they commit, all their effects appear to occur instantaneously. If they abort, they have no effect at all. Actions may nest. A remote procedure call is a nested action. Built-in **atomic objects** support low-level actions, and may be used within a guardian to synchronize its processes.

3.9. Communication Port

Like CSP and Distributed Processes, Communication Port [64] is less a full-scale language than an elegant concept on which a language might be based. A Communication Port program consists of a fixed collection of processes. There are no modules. There is no shared data. Processes communicate with remote invocation send and explicit message receipt.

Each process provides a variety of **ports** to which any other process may send messages. Ports provide strict type checking. Senders name both the receiver and its port. There may thus be several receivers with the same internal structure. The *receive* statement is non-deterministic, and guards may be placed on its options. The guards may refer to local data only. Receiving a message and resuming a sender are both explicit operations; it is possible for a receiver to be in rendezvous with several senders at one time. The senders may be released in any order. Careful placement of *release* statements is a useful tuning technique that can be used to minimize the length of rendezvous and increase concurrency.

3.10. Edison

Edison [19, 20] is a remarkable language in a number of ways. Based loosely on Pascal, Concurrent Pascal, and Modula, it is a considerably smaller language than any of the three. Its creation seems to have been an experiment in minimal language design.

Processes in Edison are created dynamically with *cobegin*. Modules are used for data hiding. Communication is by means of shared data, and mutual exclusion is achieved through critical regions. There are no separate classes of critical regions; the effect is the same as would be achieved by use of a single, system-wide semaphore. Entry to critical regions may be controlled by arbitrary Boolean guards. It is possible to follow a programming strategy in which all access to shared data is controlled by monitors created out of critical regions and modules. It is equally possible to avoid such rules.

Despite its title (''a multiprocessor language''), I question the suitability of Edison for use on multiple processors. The use of critical regions that *all* exclude each other will require the periodic halting of all processors save one. On a multi-computer, shared data is an additional problem. Unless a careful programming style is imposed above and beyond the rules of the language itself, Edison does not fit into the framework of section 2.

3.11. StarMod

StarMod [25] is an extension to Modula that attempts to incorporate some of the novel ideas of Distributed Processes. It provides additional features of its own. Modules and processes are distinct. Modules may nest. There may be arbitrarily many processes within a module. Processes

may be created dynamically: they are independent equals. Processes within the same **processor module** may share data. The programmer may influence their relative rates of progress by the assignment of **priorities**.

StarMod provides both explicit and implicit message receipt and both synchronization and remote invocation send. The four resulting types of communication employ a common syntax on the sending end. Communication paths are many-one. A sender names both the receiving module and its entry point. Entries may be called either as procedures or as functions. A *procedural send* allows the sender to continue as soon as its message is received. A *functional send* blocks the sender until its value is returned. Remote invocation send is thus limited to returning a single value

On the receiving end, a module may mix its two options, using explicit receipt on some of its communication paths and implicit receipt on the others. The sender has no way of knowing which is employed. As a matter of fact, it is possible to change a receiver from one approach to the other without any change to the sender. Libraries can be changed without invalidating the programs that use them. When a message arrives at a implicit entry point, a new process is created to handle the call. When a message arrives at a explicit entry point, it waits until some existing process in the module performs a *receive* on the corresponding **port**. There is no mutual exclusion among processes in a module: they proceed in (simulated) parallel. They may arrange their own synchronization by waiting on semaphores. The explicit *receive* is non-deterministic, but there are no guards on its options. A single receiver can be in rendezvous with more than one sender at a time, but it must release them in LIFO order. Separate calls to the same implicit port will create separate, possibly parallel, processes. Separate processes in a module may receive from the same explicit port.

StarMod was designed for dedicated real-time applications. The StarMod kernel behaves like a miniature operating system, highly efficient and tuned to the needs of a single type of user level program. Simplicity is gained at the expense of requiring every program to specify the interconnection topology of its network. Direct communication is permitted only between modules that are neighbors. The programmer is thus responsible for routing.

3.12. ITP

The Input Tool Process model [14] is an extension of van den Bos's Input Tool Method [13], an unconventional language for input-driven programs.

An ITP program consists of a collection of processes. There are no modules. Processes do not nest. They share no data. Each process consists of a hierarchical collection of **tools**. A tool looks something like a procedure. It is made available for activation by appearing in the **input rule** of a higher-level tool (The root tools are always available). It is actually activated by the completion of lower-level tools appearing in its own input rule. Leaf tools are activated in response to inputs from other processes, or from the user.

Input rules allow for message screening. Their syntax is reminiscent of path expressions (section 3.1). They specify the orders in which lower-level tools may be activated. Unwanted inputs can be disallowed at any layer of the hierarchy.

ITP uses synchronization send with implicit message receipt. Within a process, any tool can send data to any other process. The naming mechanism is extremely flexible. At their most general, the communication paths are many-many. A sender can specify the name of the receiving process,

the receiving tool, both, or neither. It can also specify **broadcast** to all the members of a process **set**. A receiver (leaf tool) can accept a message from anyone, or it can specify a particular sender or group of senders from which it wants to choose. A global communication **arbiter** coordinates the pairing of appropriate senders and receivers.

The current ITP implementation runs on multiple processors, but does not allow the most general many-many communication paths. Syntax for the sequential part of the language is borrowed from C [54].

3.13. Ada

The adoption of Ada [77] by the U. S. Department of Defense is likely to make it the standard against which concurrent languages are compared in future years.

Processes in Ada are known as **tasks**. Tasks may be statically declared, or may be created at run time. The code associated with a task is a special kind of module. Since modules may nest, it is possible for one task to be declared inside another. This nesting imposes a strict hierarchical structure on a program's tasks. No task is permitted to leave a lexical scope until all that scope's nested tasks have terminated. A task can be aborted from outside. Tasks may share data. They may also pass messages.

Ada uses remote invocation send. The sender names both the receiver and its entry point. Dynamically-created tasks are addressed through pointers. Communication paths are many-one. *Receive* is explicit. Guards (depending on both local and global variables) are permitted on each clause. The choice between open clauses is non-deterministic. A receiver may be in rendezvous with more than one sender at a time, but must release them in LIFO order. There is no special mechanism for asynchronous receipt; the same effect may be achieved through the use of shared data. Ada provides sophisticated facilities for timed pauses in execution and for communication timeout. Communication errors raise the TASKING-ERROR exception. A programmer may provide for error recovery by handling this exception.

Since data may be shared at all levels of lexical nesting, it may be necessary for separate tasks to share (logical) activation records. That may be quite a trick across machine boundaries. More subtle problems arise from the implicit relationships among relatives in the process tree. For example, it is possible for a task to enter a loop in which it repeatedly receives messages until all of its peers have terminated or are in similar loops. The implementation must detect this situation in order to provide for the normal termination of all the tasks involved.

3.14. Synchronizing Resources

SR [2, 3] is an attempt to generalize and unify a number of earlier proposals. It appears to have grown out of earlier work on extensions to monitors [1].

An SR program consists of a collection of modules called **resources**. A resource may contain one or more processes, and may export **operations** those processes define. Operations are similar to ports in Extended CLU and entries in Ada. The processes within a resource share data. Neither resources nor processes may nest. There is special syntax for declaring arrays of identical resources, processes, and operations. A procedure is abbreviated syntax for a process that sits in an

infinite loop with a *receive* statement at the top and a *send* at the bottom.

Receive is explicit. Its syntax is based on Dijkstra's guarded commands [28]. Input guards have complete access to the contents of potential messages. Moreover, messages need not be received in the order sent. A receiver may specify that the queue associated with an operation should be ordered on the basis of an arbitrarily complicated formula involving the contents of the messages themselves. It is possible for a process to be in rendezvous with more than one sender at a time. It must release them in LIFO order.

SR provides both no-wait and remote invocation send. Messages are sent to specific operations of specific resources. Thus each communication path has a single receiving resource and, potentially, multiple senders. Operations can be named explicitly, or they can be referenced through **capability** variables. A capability variable is similar to a record; it consists of several fields, each of which points to an operation of a specified type. Within a resource, a particular operation must be serviced by only one process.

There are no facilities for asynchronous receipt or timeout. Each operation, however, has an associated function that returns the current length of its queue. This function may be used to simulate a *receive* with timeout of zero: the receiver simply checks the queue length before waiting.

3.15. Linda

Linda [35, 36] provides the full generality of many-many communication paths. Processes interact in Linda by inserting and removing **tuples** from a distributed **global name space** called **structured memory** (**STM**). The name space functions as an associative memory; tuples are accessed by referring to the patterns they contain.

Published papers on Linda contain few details on the language syntax. Presumably, there is some sort of module concept, since processes that reside on the same machine are allowed to share data in addition to STM. There is also some sort of mutual exclusion mechanism that protects shared data.

Linda combines a no-wait send with explicit message receipt. Tuples are added to STM with the non-blocking *out()* command. They are removed with the *in()* command. A *read()* command (also called *in*()*) allows tuples to be read without removing them from STM. All three commands take an arbitrary list of arguments. The first is required to be an actual value of type **name**. The rest may be actuals or "formals." An *in()* command succeeds when it finds a tuple in STM that matches all its actuals and provides actuals for all its formals. In *out()* commands, formals serve as "don't care" flags; they match any actual. In *in()* commands, formals are slots for incoming data.

The matching of tuples according to arbitrary patterns of actuals provides a very powerful mechanism for message screening. It also leads to serious implementation problems. Much of the work on Linda involves finding tractable algorithms for managing STM. Current efforts are aimed at implementation on the Stony Brook microcomputer Network, a wrapped-around grid (torus) architecture.

3.16. NIL

NIL [22, 75] is a language under development at IBM's T.J. Watson Research Center. It is intended for use on a variety of distributed hardware. The current implementation runs on a single IBM 370. Processes are the fundamental program units; there is no separate module concept. There is no shared data; processes communicate only by message passing. The designers of NIL suggest that a compiler might divide a process into concurrent pieces if more than one CPU were available to execute it.

Communication paths are many-one. They are created dynamically by connecting **output ports** to an appropriate **input port**. Any process can use the *publish* command to create **capabilities** that point to its input ports. It may then pass the capabilities in messages to other processes that can use them in *connect* commands. All type checking on ports is performed at compile time.

NIL provides both no-wait and remote invocation send. Remote invocation sends may be **forwarded**. The process receiving a forwarded message is responsible for releasing the sender. No-wait sends are buffered and *destructive*: the sender loses access to the components of the message. From the programmer's point of view, destructive *send* eliminates the distinction between value and reference parameters. Even a parameter passed by reference cannot be modified by its sender after it is sent.

Receive in NIL is explicit. It has two varieties, one to correspond to each type of *send*. **Exceptions** are used to recover from communication errors. There are elaborate rules concerning the propagation of exceptions when a process terminates.

4. Related Notions

Each of the proposals described in section 3 has been described in the literature (at least in part) as a high-level language for distributed computing. For one reason or another, the proposals in this section have not. They all contain useful ideas, however, and are worth considering in any discussion of interprocess communication and concurrency.

The survey in section 3 is meant to be reasonably complete. No such claim is made for this section. I have used by own personal tastes in deciding what to include.

4.1. Concurrent Languages

Several early high-level languages, notably Algol-68 [76], PI/I [9], and SIMULA [12], provided some sort of support for "concurrent" processes, or at least coroutines. These languages relied on shared data for inter-process interaction. They were intended primarily for uni-processors, and may have been suitable for multi-processors as well, but they were certainly not designed for implementation on multi-computers. Recently, Modula-2 [82, 83] has re-awakened interest in coroutines as a practical programming tool. In designing Modula-2, Wirth has recognized that even on a uni-processor, and even in the absence of interrupts, there are still algorithms that are most elegantly expressed as a collection of cooperating threads of control.

Modula-2 is more closely related to Pascal [49] than to the Modula of section 3.2. For the purposes of this survey, the principal difference between the Modulas is that the newer language

incorporates a much simpler and more primitive form of concurrency. Processes in Modula-2 are actually **coroutines**; they execute one at a time. New processes are created by a built-in procedure that accepts a procedure name and an array to be used as stack space and returns the **id** of a newly created process. There is no pre-emption; a given process continues to run until it explicitly relinquishes control and names the process to be resumed in its stead.

One goal of Modula-2 is to permit a large variety of process-scheduling strategies to be implemented as library packages. By hiding all coroutine transfers within library routines, the programmer can imitate virtually any other concurrent language. The imitations can be straightforward and highly efficient. For a uni-processor, Modula-2 provides the richness of expression of multiple threads of control at very little cost.

4.2. Nelson's Remote Procedure Call

Nelson's thesis [66] is devoted to the development of a *transparent* mechanism for remote procedure calls. A remote procedure call combines remote invocation send with implicit message receipt. Transparency is defined to mean that remote and local procedure calls appear to be the same: they share the same

- atomicity semantics,
- naming and configuration,
- type checking,
- parameter passing, and
- exception handling.

Nelson describes a mechanism, called **Emissary**, for implementing remote procedure calls. Emissary attempts to satisfy all five of the "essential properties" listed above, together with one "pleasant property:" efficiency. The attempt at transparency is almost entirely successful, and the performance results are quite impressive.

Emissary falls short of true transparency in the area of parameter passing. Not all data types are meaningful when moved to a different address space. Unless one is willing to incur the cost of remote memory accesses, pointers and other machine-specific data cannot be passed to remote procedures. Moreover, *in/out* parameters must be passed by value/result, and not by reference. In the presence of aliasing and other side effects, remote procedures cannot behave the same as their local counterparts. So long as programmers insist on pointers and reference parameters, it is unrealistic to propose a *truly* transparent mechanism.

4.3. Distributed Operating Systems

The borderline between programming languages and operating systems is very fuzzy, especially in hypothetical systems. Interprocess communication lies very near the border. It is often difficult to tell whether a particular mechanism is really part of the language or part of the underlying system. Much depends on the degree to which the mechanism is integrated with other language features: type checking, variable names, scope rules, protection, exception handling, concurrency, and so forth. The mechanisms described in this section, at least in their current form, are fairly clearly on the operating system side of the line. Incorporating them into the language level is a job for future research.

4.3.1. Links

Links were introduced in the Demos [8] operating system. They have been adopted, in one form or another, by several descendant systems: Arachne (Roscoe) [34, 74], Charlotte [5, 31], and DEMOS/MP [68]. My work with Raphael Finkel on the LYNX project [73] is an attempt to embed the notion of links in a high-level programming language.

Links are a naming and protection mechanism. In Demos, and in Arachne and DEMOS/MP, a link is a capability to an input port. It connects an **owner** to an arbitrary number of **holders**. The owner can receive messages from the link. It owns the input port. A holder can send messages to the link. It holds the capability. A holder can create copies of its capability, and can send them in messages on other links. The owner can exercise control over the distribution of capabilities and the rights that they confer.

Where Demos links are many-one, Charlotte links are one-one. Their ends are symmetric. Each process can send and receive. There is no notion of owner and holder. Only one process can have access to a given end of a given link at a given point in time.

The protection properties of links make them useful for applications that are somewhat loosely coupled — applications in which processes are developed independently, and cannot assume that their partners are correct. Typically, a link is used to represent a **resource**. (In a timesharing system, a link might represent a file.) Since a single process may implement a whole collection of resources, and since a single resource may be supported by an arbitrary number of operations, links provide a *granularity* of naming somewhere in between process names and operation names.

4.3.2. SODA

SODA [53] is an acronym for a "Simplified Operating system for Distributed Applications." It might better be described as a communications protocol for use on a broadcast medium with a very large number of heterogeneous nodes.

Each node on a SODA network consists of two processors: a **client processor**, and an associated **kernel processor**. The kernel processors are all alike. They are connected to the network and communicate with their client processors through shared memory and interrupts. Nodes are expected to be more numerous than processes, so client processors are not multi-programmed.

Communication paths in SODA are many-one, but there is a mechanism by which a process can broadcast a request for server names that **match** a certain pattern. All communication statements are non-blocking. Processes are informed of interesting events by means of software interrupts. Interrupts can be masked.

From the point of view of this survey, the most interesting aspect of the SODA protocol is the way in which it *decouples* control flow and data flow. In all the languages in section 3, message transfers are initiated by the sender. In SODA, the process that initiates an interaction can arrange to send data, receive data, both, or neither. The four options are termed, respectively, **put**, **get**, **exchange**, and **signal**. Synchronization in SODA falls outside the classification system described in section 2.4.

Every transaction between a pair of processes has a **requester** and a **server**. The server feels a software interrupt whenever a requester attempts to initiate a transaction. The interrupt handler is provided with a (short) description of the request. At its convenience, the server can **accept** a

request that triggered its handler at some point in the past. When it does so, the transaction actually occurs, and the requester is notified by an interrupt of its own. The user is responsible for writing handlers and for keeping track of outstanding requests in both the server and requester. In simple cases, the bookkeeping involved may be supported by library routines.

5. Conclusion

There is no doubt that the best way to evaluate a language is to use it. A certain amount of armchair philosophizing may be justified (this paper has certainly done its share!), but the real test of a language is how well it works in practice. It will be some time before most of the languages in section 3 have received enough use to make definitive judgments possible.

One very useful tool would be a representative sample of the world's more difficult distributed problems. To evaluate a language, one could make a very good start by coding up solutions to these problems and comparing the results to those achieved with various other methods. Much of the success of any language will depend on the elegance of its syntax — on whether it is pleasant and natural to use. But even the best of syntax cannot make up for a fundamentally unsound design.

Section 2 has discussed some major open questions. The two most important appear to be the choice of synchronization semantics for the *send* operation, and the choice between implicit and explicit message receipt. I have argued elsewhere [72] that a reasonable language needs to provide a variety of options. Just as a sequential language benefits from the presence of several similar loop constructs, so can a distributed language benefit from the presence of several similar constructs for inter-process communication.

The ubiquity of the client/server relationship makes remote invocation with implicit receipt an extremely attractive combination. In situations where a reply is expected, remote invocation send has cleaner syntax than either other synchronization method. It is also more efficient than the synchronization send. Clarity stems from the fact that the sender need not do anything special to receive a reply. Efficiency results from the lack of a separate message to acknowledge the original request.

Unfortunately, there are applications for which the client/server model is clearly inappropriate. Remote invocation may be overly slow in situations that lack a reply. Implicit receipt is both awkward and counter-intuitive in the case of cooperating peers. It is possible that implicit message receipt need only be paired with remote invocation send. Likewise, explicit receipt may best be paired with one of the other synchronization mechanisms. Yet there is something to be said for providing all the combinations, since they can be supported without syntactic complexity, and since all are occasionally useful.

There is a general consensus that concurrent programs are at least an order of magnitude more difficult to write than their sequential counterparts. Research to date has done distressingly little to reduce that level of complexity. The most important avenues of future research will be those that pull the task of concurrent programming down to the conceptual level of the average human being. Refinements of existing languages will help to some extent, much as modern control constructs have improved upon the old standbys of the '50's and '60's. Eventually, however, ways must be found to automate much of the programmer's task. For specific classes of problems, it may be possible to design application packages that hide the details of distributed implementations [32]. For

more general solutions, major advances are needed in error recovery, verification, synchronization, and even the specification of parallelism itself. It is worth noting that thirty years of effort have failed to produce an ideal sequential language. It is unlikely that the next thirty will see an ideal distributed language, either.

Acknowledgments

The original draft of this paper was prepared for an independent study course supervised by Raphael Finkel. Subsequent drafts have benefited from the written comments of Marvin Solomon and Prasun Dewan, and from informal discussions with various members of the Charlotte and Crystal research groups.

References

[1] Andrews, G. R. and J. R. McGraw, "Language Features for Process Interaction," *Proceedings of an ACM Conference on Language Design for Reliable Software* (28-30 March 1977), pp. 114-127. In *ACM SIGPLAN Notices* 12:3 (March 1977).

[2] Andrews, G. R., "Synchronizing Resources," *ACM TOPLAS* 3:4 (October 1981), pp. 405-430.

[3] Andrews, G. R., "The Distributed Programming Language SR - Mechanisms, Design and Implementation," *Software - Practice and Experience* 12 (1982), pp. 719-753.

[4] Andrews, G. R. and F. B. Schneider, "Concepts and Notations for Concurrent Programming," *ACM Computing Surveys* 15:1 (March 1983), pp. 3-44.

[5] Artsy, Y., H.-Y. Chang, and R. Finkel, "Charlotte: Design and Implementation of a Distributed Kernel," Computer Sciences Technical Report #554, University of Wisconsin - Madison, August 1984.

[6] Baiardi, F., L. Ricci, and M. Vanneschi, "Static Checking of Interprocess Communication in ECSP," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction* (17-22 June 1984), pp. 290-299. In *ACM SIGPLAN Notices* 19:6 (June 1984).

[7] Ball, J. E., G. J. Williams, and J. R. Low, "Preliminary ZENO Lnaguage Description," *ACM SIGPLAN Notices* 14:9 (September 1979), pp. 17-34.

[8] Baskett, F., J. H. Howard, and J. T. Montague, "Task Communication in Demos," *Proceedings of the Sixth ACM Symposium on Operating Systems Principles* (November 1977), pp. 23-31.

[9] Beech, D., "A Structural View of PL/I," *Computing Surveys* 2:1 (March 1970), pp. 33-64.

[10] Bernstein, A. J., "Output Guards and Nondeterminism in 'Communicating Sequential Processes'," *Transactions on Programming Languages and Systems* 2:2 (April 1980), pp. 234-238.

[11] Bernstein, A. J. and J. R. Ensor, "A Modula Based Language Supporting Hierarchical Development and Verification," *Software - Practice and Experience* 11 (1981), pp. 237-255.

[12] Birtwistle, G. M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA Begin*, Auerback Press, Philadelphia, 1973.

[13] Bos, J. van den, "Input Tools - A New Language for Input-Driven Programs," *Proceedings of the European Conference on Applied Information Technology*, IFIP (25-28 September 1979), pp. 273-279. Published as *EURO IFIP 79*, North-Holland, Amsterdam, 1979.

[14] Bos, J. van den, R. Plasmeijer, and J. Stroet, "Process Communication Based on Input Specifications," *ACM TOPLAS* 3:3 (July 1981), pp. 224-250.

[15] Brinch Hansen, P., "Structured Multi-programming," *CACM* 15:7 (July 1972), pp. 574-578.

[16] Brinch Hansen, P., *Operating System Principles*, Prentice-Hall, 1973.

[17] Brinch Hansen, P., "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering* SE-1:2 (June 1975), pp. 199-207.

[18] Brinch Hansen, P., "Distributed Processes: A Concurrent Programming Concept," *CACM* 21:11 (November 1978), pp. 934-941.

[19] Brinch Hansen, P., "The Design of Edison," Technical Report, University of Southern California Computer Science Department, September 1980.

[20] Brinch Hansen, P., "Edison: A Multiprocessor Language," Technical Report, University of Southern California Computer Science Department, September 1980.

[21] Buckley, G. N. and A. Silberschatz, "An Effective Implementation for the Generalized Input-Output Construct of CSP," *ACM TOPLAS* 5:2 (April 1983), pp. 223-235.

[22] Burger, W. F., N. Halim, J. A. Pershing, F. N. Parr, R. E. Strom, and S. Yemini, "Draft NIL Reference Manual," RC 9732 (#42993), I.B.M. T. J. Watson Research Center, December 1982.

[23] Campbell, R. H. and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," pp. 89-102 in *Operating Systems*, Lecture Notes in Computer Science #16, ed. C. Kaiser, Springer-Verlag, Berlin, 1974.

[24] Cashin, P., "Inter-Process Communication," Technical Report 8005014, Bell-Northern Research, 3 June 1980.

[25] Cook, R. P., "*Mod--A Language for Distributed Programming," *IEEE Transactions on Software Engineering* SE-6:6 (November 1980), pp. 563-571.

[26] Dijkstra, E. W., "Co-operating sequential processes," pp. 43-112 in *Programming Languages*, ed. F. Genuys, Academic Press, London, 1968.

[27] Dijkstra, E. W., "Hierarchical Ordering of Sequential Processes," pp. 72-93 in *Operating Systems Techniques*, A. P. I. C. Studies in Data Processing #9, ed. C. A. R. Hoare and R. H. Perrott, Academic Press, London, 1972. Also *Acta Informatica* 1 (1971), pp 115-138.

[28] Dijkstra, E. W., "Guarded Commands, Non-determinacy and Formal Derivation," *CACM* 18:8 (August 1975), pp. 453-457.

[29] Ellis, C. S., J. A. Feldman, and J. E. Heliotis, "Language Constructs and Support Systems for Distributed Computing," *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (18-20 August 1982), pp. 1-9.

[30] Feldman, J. A., "High Level Programming for Distributed Computing," *CACM* 22:6 (June 1979), pp. 353-368.

[31] Finkel, R., M. Solomon, D. DeWitt, and L. Landweber, "The Charlotte Distributed Operating System: Part IV of the First Report on the Crystal Project," Computer Sciences Technical Report #502, University of Wisconsin - Madison, October 1983.

[32] Finkel, R. and U. Manber, "DIB: A Distributed Implementation of Backtracking," submitted to the *Fifth International Conference on Distributed Computing Systems*, September 1984.

[33] Finkel, R. A., "Tools for Parallel Programming," Appendix B of Second Report, Wisconsin Parallel Array Computer (WISPAC) Research Project, University of Wisconsin Electrical and Computer Engineering Report #80-27, August 1980.

[34] Finkel, R. A. and M. H. Solomon, "The Arachne Distributed Operating System," Computer Sciences Technical Report #439, University of Wisconsin - Madison, 1981.

[35] Gelernter, D. and A. Bernstein, "Distributed Communication via Global Buffer," *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (18-20 August 1982), pp. 10-18.

[36] Gelernter, D., "Dynamic Global Name Spaces on Network Computers," *Proceedings of the 1984 International Conference on Parallel Processing* (21-24 August, 1984), pp. 25-31.

[37] Good, D. I., R. M. Cohen, and J. Keeton-Williams, "Principles of Proving Concurrent Programs in Gypsy," *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages* (29-31 January 1979), pp. 42-52.

[38] Habermann, A. N., "Synchronization of Communicating Processes," *CACM* 15:3 (March 1972), pp. 171-176.

[39] Habermann, A. N., "On the Timing Restrictions of Concurrent Processes," *Fourth Annual Texas Conference on Computing Systems* (17-18 November 1975), pp. 1A.3.1-1A.3.6.

[40] Haddon, B. K., "Nested Monitor Calls," *ACM Operating Systems Review* 11:4 (October 1977), pp. 18-23.

[41] Herlihy, M. and B. Liskov, "Communicating Abstract Values in Messages," Computation Structures Group Memo 200, Laboratory for Computer Science, MIT, October 1980.

[42] Hoare, C. A. R., "Towards a Theory of Parallel Programming," pp. 61-71 in *Operating Systems Techniques*, A. P. I. C. Studies in Data Processing #9, ed. C. A. R. Hoare and R. H. Perrott, Academic Press, London, 1972.

[43] Hoare, C. A. R., "Monitors: An Operating Systems Structuring Concept," *CACM* 17:10 (October 1974), pp. 549-557.

[44] Hoare, C. A. R., "Communicating Sequential Processes," *CACM* 21:8 (August 1978), pp. 666-677.

[45] Holt, R. C., G. S. Graham, E. D. Lazowska, and M. A. Scott, "Announcing CONCURRENT SP/k," *ACM Operating Systems Review* 12:2 (April 1978), pp. 4-7.

[46] Holt, R. C., G. S. Graham, E. D. Lazowska, and M. A. Scott, *Structured Concurrent Programming with Operating Systems Applications*, Addison-Wesley, 1978.

[47] Howard, J. H., "Signaling in Monitors." *Proceedings of the Second International Conference on Software Engineering* (13-15 October 1976), pp. 47-52.

[48] Jazayeri, M., C. Ghezzi, D. Hoffman, D. Middleton, and M. Smotherman, "CSP/80: A Language for Communicating Sequential Processes," *IEEE COMPCON Fall 1980* (September 1980), pp. 736-740.

[49] Jensen, K. and N. Wirth, *Pascal User Manual and Report*, Lecture Notes in Computer Science #18, Springer-Verlag, Berlin, 1974.

[50] Kahn, G. and D. B. MacQueen, "Coroutines and Networks of Parallel Processes," pp. 993-998 in *Information Processing 77*, ed. B. Gilchrist, North-Holland, 1977. Proceedings of the 1977 IFIP Congress, Toronto (8-12 August 1977).

[51] Kaubisch, W. H., R. H. Perrott, and C. A. R. Hoare, "Quasiparallel Programming," *Software - Practice and Experience* 6 (1976), pp. 341-356.

[52] Keedy, J. L., "On Structuring Operating Systems with Monitors," *ACM Operating Systems Review* 13:1 (January 1979), pp. 5-9.

[53] Kepecs, J., "SODA: A Simplified Operating System for Distributed Applications," Ph. D. Thesis, University of Wisconsin - Madison, January 1984. Reprinted as Computer Sciences Technical Report #527, by J. Kepecs and M. Solomon.

[54] Kernighan, B. W. and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1978

[55] Kessels, J. L. W., "An Alternative to Event Queues for Synchronization in Monitors," *CACM* 20:7 (July 1977), pp. 500-503.

[56] Lampson, B. W., J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek, "Report On The Programming Language Euclid," *ACM SIGPLAN Notices* 12:2 (February 1977).

[57] Lampson, B. W. and D. D. Redell, "Experience with Processes and Monitors in Mesa," *CACM* 23:2 (February 1980), pp. 105-117.

[58] Liskov, B., "Primitives for Distributed Computing," *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (December 1979), pp. 33-42.

[59] Liskov, B., "Linguistic Support for Distributed Programs: A Status Report," Computation Structures Group Memo 201, Laboratory for Computer Science, MIT, October 1980.

[60] Liskov, B. and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM TOPLAS* 5:3 (July 1983), pp. 381-404.

[61] Liskov, B., "Overview of the Argus Language and System," Programming Methodology Group Memo 40, Laboratory for Computer Science, MIT, February 1984.

[62] Lister, A., "The Problem of Nested Monitor Calls," *ACM Operating Systems Review* 11:3 (July 1977), pp. 5-7. Relevant correspondence appears in Volume 12, numbers 1, 2, 3, and 4.

[63] Lister, A. M. and K. J. Maynard, "An Implementation of Monitors," *Software - Practice and Experience* 6 (1976), pp. 377-385.

[64] Mao, T. W. and R. T. Yeh, "Communication Port: A Language Concept for Concurrent Programming," *IEEE Trancactions on Software Engineering* SE-6:2 (March 1980), pp. 194-204.

[65] May, D., "OCCAM," *ACM SIGPLAN Notices* 18:4 (April 1983), pp. 69-79. Relevant correspondence appears in Volume 19, number 2 and Volume 18, number 11.

[66] Nelson, B. J., "Remote Procedure Call," Ph. D Thesis, Technical Report CMU-CS-81-119, Carnegie-Mellon University, 1981.

[67] Parnas, D. L., "The Non-Problem of Nested Monitor Calls," *ACM Operating Systems Review* 12:1 (January 1978), pp. 12-14. Appears with a response by A. Lister.

[68] Powell, M. L. and B. P. Miller, "Process Migration in DEMOS/MP," *Proceedings of the Ninth ACM Symposium on Operating Systems Principles* (10-13 October 1983), pp. 110-118. In *ACM Operating Systems Review* 17:5.

[69] Pratt, T., *Programming Languages: Design and Implementation*, Prentice-Hall, Englewood Cliffs, 1975.

[70] Robert, P. and J.-P. Verjus, "Toward Autonomous Descriptions of Synchronization Modules," pp. 981-986 in *Information Processing 77*, ed. B. Gilchrist, North-Holland, 1977. Proceedings of the 1977 IFIP Congress, Toronto (8-12 August 1977).

[71] Roper, T. J. and C. J. Barter, "A Communicating Sequential Process Language and Implementation," *Software - Practice and Experience* 11 (1981), pp. 1215-1234.

[72] Scott, M. L., "Messages vs. Remote Procedures is a False Dichotomy," *ACM SIGPLAN Notices* 18:5 (May 1983), pp. 57-62.

[73] Scott, M. L. and R. A. Finkel, "LYNX: A Dynamic Distributed Programming Language," *Proceedings of the 1984 International Conference on Parallel Processing* (21-24 August, 1984), pp. 395-401.

[74] Solomon, M. H. and R. A. Finkel, "The Roscoe Distributed Operating System," *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (December 1979), pp. 108-114.

[75] Strom, R. E. and S. Yemini, "NIL: An Integrated Language and System for Distributed Programming," *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems* (27-29 June 1983), pp. 73-82. In *ACM SIGPLAN Notices* 18:6 (June 1983).

[76] Tanenbaum, A. S., "A Tutorial on Algol 68," *ACM Computing Surveys* 8:2 (June 1976), p. 155.

[77] United States Department of Defense, "Reference Manual for the Ada Programming Language," (ANSI/MIL-STD-1815A-1983), 17 February 1983.

[78] Welsh, J. and D. W. Bustard, "Pascal-Plus - Another Language for Modular Multiprogramming," *Software-Practce and Experience* 9 (1979), pp. 947-957.

[79] Wettstein, H., "The Problem of Nested Monitor Calls Revisited," *ACM Operating Systems Review* 12:1 (January 1978), pp. 19-23.

[80] Wirth, N., "Modula: a Language for Modular Multiprogramming," *Software - Practice and Experience* 7 (1977), pp. 3-35.

[81] Wirth, N., "Design and Implementation of Modula." *Software - Practice and Experience* 7 (1977), pp. 67-84.

[82] Wirth, N., "Modula-2." Report 36, Institut fur Informatik, ETH Zurich, 1980.

[83] Wirth, N., *Programming in Modula-2*, Texts and Monographs in Computer Science, ed. D. Gries, Springer-Verlag, Berlin, Second Edition, 1983.