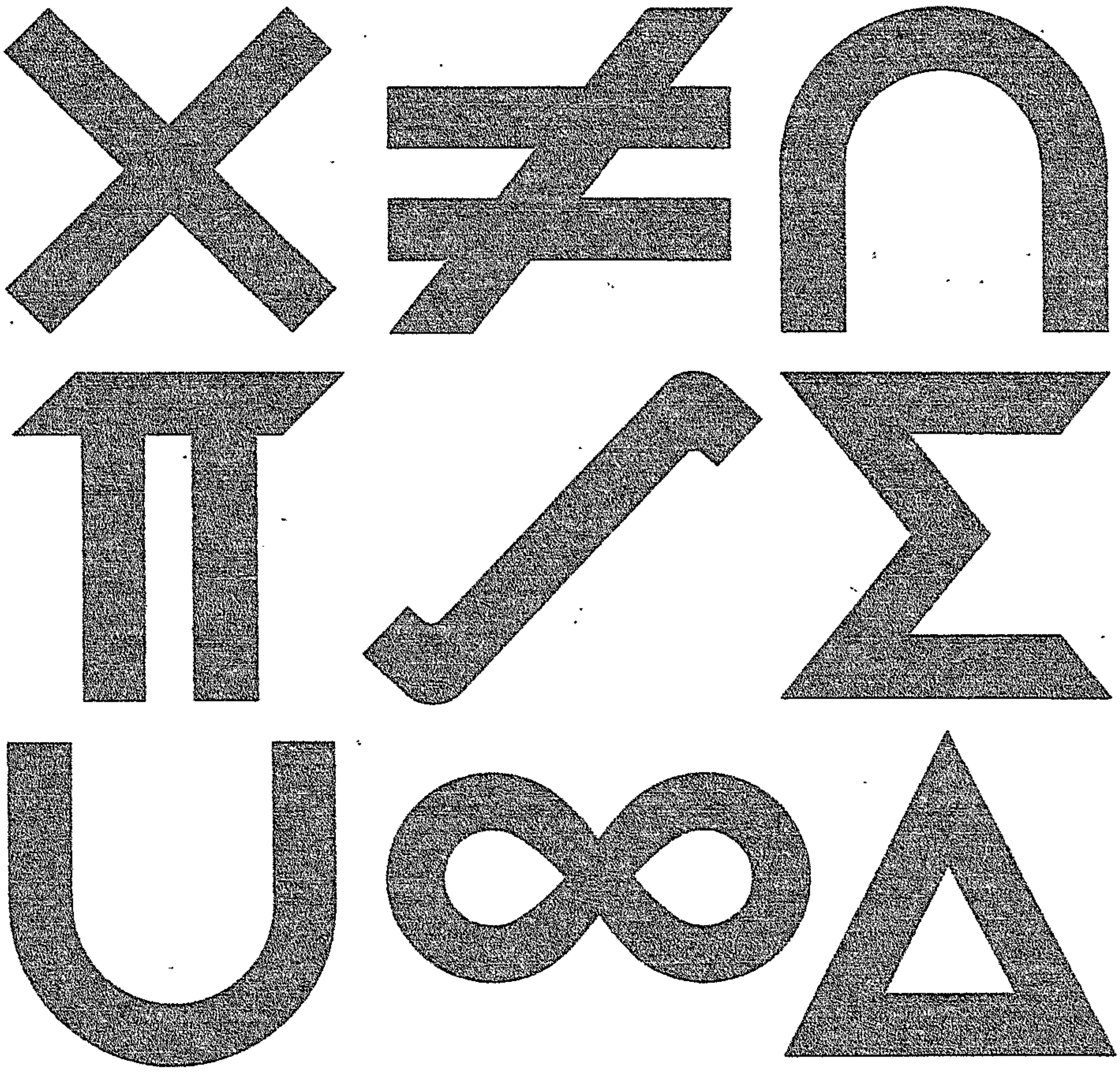
G320-2757, September 1984

AN APPROACH TO UNIFYING INTER-OBJECT COMMUNICATION

P. S. NEWMAN



IBM LOS ANGELES SCIENTIFIC CENTER REPORTS

FOR THE YEAR 1980

G320-2702 March 1980
K. EWUSI-MENSAH, Criteria for Decomposing an Information System into Its Subsystems for Business Systems Planning (26 p.)

G320-2703 March 1980
K. EWUSI-MENSAH, Computer-Aided Modeling and Analysis Techniques for Determining Management Information Systems Requirements (30 p.)

G320-2704 June 1980
P. S. NEWMAN, An Atomic Network Programming Language (29 p.)

G320-2705 July 1980
J. G. SAKAMOTO, Use of DB/DC Data Dictionary to Support Business Systems Planning Studies: An Approach (24 p.)

G320-2707 October 1980
P. S. NEWMAN, Towards an Integrated Development Environment (29 p.)

G320-2785-5 April 1980
Compiled by KATHERINE HANSON, Abstracts of Los Angeles Scientific Center Reports (104 p.)

FOR THE YEAR 1981

G320-2708 September 1981
M. M. PARKER, Enterprise Information Analysis: A Survey of Methodologies (32 p.)

G320-2709 September 1981
M. M. PARKER, Enterprise Information Analysis: An Application of Current Disciplines (86 p.)

G320-2710 September 1981
M. M. PARKER, Enterprise Information Analysis: A Proposal for Discipline Extension (24 p.)

G320-2711 July 1981
A. INSELBERG, N-Dimensional Graphics; Part 1 — Lines and Hyperplanes (142 p.)

G320-2712 September 1981
S. A. JUROVICS, Daylight, Glazing, and Building Energy Minimization (19 p.)

FOR THE YEAR 1981 (Continued)

G320-2713 October 1981
B. DIMSDALE, Conic Transformations (19 p.)

G320-2714 October 1981
S. H. LIN, Existential Dependencies in Relational Databases (80 p.)

FOR THE YEAR 1982

G320-2716 July 1982
M. M. PARKER, Enterprise Information Analysis: Cost-Benefit Analysis of Information Systems Using PSL/PSA and the Yourdon Methodology (62 p.)

G320-2717 September 1982
R. C. SUMMERS, An Overview of Computer Security (29 p.)

G320-2718 October 1982
R. J. HERRERA, Data Flow Analysis Aid (19 p.)

G320-2734 November 1982
R. C. SUMMERS, M. EBRAHIMI, J. MARBERG, K. J. PERRY, R. B. TALMADGE, U. ZERNIK, RM: A System of Personal Machines and Service Machines (65 p.)

G320-2736 December 1982
C. M. CUMMING, Editor, 1982 Annual Report of the Los Angeles Scientific Center (30 p.)

FOR THE YEAR 1983

G320-2721 June 1983
H. LEVY and D. W. LOW, A New Algorithm for Finding Small Cycle Cutsets (58 p.)

G320-2735 February 1983
G. SAKAMOTO, F. W. BALL, Document Generation with DCF and PSL/PSA: An Approach (11 p.)

FOR THE YEAR 1984

G320-2737 September 1984
P. S. NEWMAN, An Approach to Unifying Inter-Object Communication (15 p.)

The availability of reports is correct as of the printing date of this report.

IBM Los Angeles Scientific Center Report No. G320-2737

September 1984

An Approach to Unifying Inter-Object Communication

Paula S. Newman

IBM Corporation
Los Angeles Scientific Center
11601 Wilshire Boulevard
Los Angeles, California 90025

Abstract

Both data abstractions and asynchronous processes are useful types of programming units, as are traditional functions and subroutines. However, the incorporation of all four types in a single language poses problems of language size and consistency.

This paper sketches an approach to providing functional equivalents of these programming units, together with some aspects of co-routines, with limited syntactic and semantic variation. Two types of programming units, called processes and procs, and a set of closely-related inter-unit communication mechanisms are developed. Provisions for integrity are addressed briefly.

1. Introduction

Programming languages usually provide one or more kinds of programming-units, differing in definition, creation, behavior, and accessing. The earliest high level languages, (e.g., FORTRAN, ALGOL), generally provided three kinds of units - main programs, functions, and subroutines. Since then, many other kinds have evolved, perhaps the most important of which are the varied facilities for abstraction and concurrent programming.

There have been some (badly needed) recent efforts, such as [2, 14, 17, 18, 22], to understand and classify the dimensions along which programming unit variation occurs. One result of these studies, which focus primarily on approaches to concurrency, seems to be that much of the variation encountered is useful, from the point of view of convenience in particular application situations. For example:

- Fully asynchronous communication (send/receive), by a concurrent caller, maximizes potential parallelism, while synchonous communication (procedure call and remote procedure call) provides implicit synchronization, and is more succinct.

- Explicit message selection and response by a concurrent callee provides considerable flexibility, but procedure-like argument acceptance and return is more convenient (and expressive of intent) when requests are to be handled one-by-one, in order of arrival.

- Data abstractions have many virtues. However, traditional procedures can be more convenient, as they do not require separate, explicit create and destroy operations. Also procedures are useful in defining the individual operations of a data abstraction.

- Subroutine reference patterns are needed for stand-alone invocations, while functional reference patterns are needed within expressions.

The utility of all these variations represents a design problem, in that their incorporation into a single language seems to imply the use of four or five kinds of programming units (functions, subroutines, data abstractions, and one or more kinds of asynchronous processes) and associated communication forms.

In this paper, an approach is suggested for the subsumption of equivalents of the above variations using only two programming units, and a set of closely related communication methods.

The development is informal, and proceeds as follows. The basic programming unit assumed, called a process, is introduced. A process is an asynchronously operating abstraction with explicit message selection and response. Following this, a chain of increasingly succinct forms for referencing this unit, ranging from full exploitation of asynchronism to functional reference, is defined. The chain unifies, by abbreviation, asynchronous, synchronous, data abstraction, subroutine and function referencing patterns. The connection between asynchronous and synchronous referencing is made via a highly expressive request structure, useful in its own right. Finally, an alternate programming unit, the proc, which uses procedure-like request acceptance and response, is defined as a shorthand process form.

The development is followed by a summary and discussion of related work. The last sections of the paper address some questions of integrity and performance raised by the language constructs proposed.

2. Processes

The basic programming unit assumed, the process, is defined so as to meet two objectives: (a) to allow the communication possibilities presented by concurrent processing to be fully exploited, and (b) to allow concurrent programs to be understood as data abstractions.

Exploiting the communication possibilities of concurrency implies not requiring synchronization for data transmission. That is, the transmission of data to a process should not require a sender wait, and the acceptance of data by a process should be performed at a process-specified point. This, in turn, implies that communication between concurrent processes is better modelled as the transmission of a message to a queue, followed by acceptance of the message into local variables of the receiver, than as the association of arguments with formal parameters.

Full communication flexibility also implies that a process should not have to accept messages in the order sent, or to respond to them in the order accepted, or to respond, for that matter, at all. It is also desirable that a (service-oriented) process should be able to respond to messages sent from previously unknown sources.

Based on these considerations, a process is defined as a programming unit which, once created, executes continuously, and has an associated queue. Basic communication is obtained by: (a) asynchronous (non-blocking) message sending facilities, and (b) message reception facilities which allow a process to accept messages from its queue when convenient, and on a selective basis.

The basic communication facilities are usable both by the provider of a service (receive request, then send response), and by the requestor of a service (send request, then receive response). Also, the message reception facilities permit a process to alternatively accept requests or responses (to requests which it initiates).

Before describing syntax, some additional assumptions are made about processes to allow them to be understood as data abstractions, to begin the process of programming unit amalgamation. A data abstraction (ultimately based on the SIMULA [5] class concept, and further developed in CLU [10]) is understood here as an object which provides a number of services or operations. It is persistent, that is, it remains in existence between invocations, and maintains its own local data.

The definition of a data abstraction is separate from its possibly many instances. The use of persistent multiple instances requires that instances be externally identified, e.g., by pointer, and that instance identifiers qualify operation requests (e.g., "StackPtr.Pop()"). For example, a stack abstraction might be referenced by "StackPtr.Pop()", where "StackPtr" identifies the instance of the abstraction, and "Pop" indicates the desired operation.

To allow processes to be understood as data abstractions they must be understood as instances of definitions, rather than as synonomous with definitions. To provide an analog of data abstraction operator definition, process definitions contain message type declarations. Messages are sequences of message elements, and message type declarations specify sequences of element types. As messages are used both for requests and for responses, process definitions contain declarations both for input message types and output message types.

Multiple process definitions may declare message types with the same simple name, and thus unique message-type references require both a process identifier and a simple name.

3. Basic Inter-Process Communication

Under the above assumptions, messages might be sent and received by the statements shown in Figure 1.

```
SENDF|SENDL Pointer.Msgtype (message element list)

RECEIVE rcv-option
  ON [defname1.]msgtype1 (rcv-var-list1) WHERE boolexp1 THEN stmt1
  ON [defname2.]msgtype2 (rcv-var-list2) WHERE boolexp2 THEN stmt2
  .......
  OTHERWISE stmt
  END
```

Figure 1. Send and Receive Statements

Both SENDF and SENDL send a message to the queue of the process identified by "Pointer". SENDF (send foreign) is used to send messages of type "Pointer.Msgtype", i.e., to send messages defined by the addressee. SENDL (send local) is used to send messages of type "*.Msgtype", i.e., to send messages declared by the sender. SENDF might be used to request services provided by the addressee, while SENDL might be used to send responses to requests.

The RECEIVE statement is a case structure ultimately deriving from the guarded commands of [6], as do communication structures in a number of languages (see [2] for an excellent survey). Messages on the receiving process queue are compared with the ON clauses, in an order determined by "rcv-option".[1] A message matches an ON clause if it is of the specified message type and the contained boolean expression (if present) is true. Message types are qualified by "defnamei" if they are defined by the sender (the defname involved is the definition of the sender). The first matching message found is copied, element by element, into the variables of the associated "rcv-var-list".

Built-in functions are provided for use within the boolean expressions to provide information about queued messages. For example, SENDER() would return a pointer to the sender of the message being tested, to allow the favoring of some senders over others. This approach has some resemblance to that of PLITS [8]. The messages of the latter are standardized, in that they contain only subsets of globally-known labelled fields, and can be queried prior to receipt. Here, the messages themselves cannot be queried, but information can be obtained about their senders, time of transmission, queue positions, and sending transactions (see discussion of integrity, further on).

Similar functions are needed to provide information about the last message received, to allow out-of-order responses. For example, LASTSENDER() might return a pointer to the sender of the last message received.

4. The Communication Abbreviation Chain

One interesting side effect of modelling abstractions as processes, with full message transmission for both requests and responses, is particularly important. This is the possibility of providing alternative responses to a particular request, leading to rather expressive code. For example, a stack abstraction might distinguish between two types of responses to a "Pop" operation: "Ok", accompanied by the popped element, and "Empty", accompanied by a null message. The stack might then be referenced by the sequence shown in Figure 2A, which clearly distinguishes between the actions to be taken in the two cases.

However, this form is extremely longwinded, as compared with typical stack abstraction reference syntax such as "X = StackPtr.Pop()", and the verbosity interferes with readability. A chain of abbreviations for the above SEND/RECEIVE sequence is therefore developed which allows users to trade off concurrency for readability, and brevity.

The REQUEST statement shown in Figure 2B is equivalent to the SEND/RECEIVE sequence of 2A, but is more succinct and readable. It can be used when the requestor cannot continue until a response is obtained, and both the request and response messages are defined by the requestee. A similar form is described in [11].

[1] The "rcv-option"s are: (a) compare each message, in queue order, with all ON clauses, (b) compare each ON clause, in the specified order, with all messages, and (c) compare each ON clause, in random order, with all messages.

```
A. SEND/RECEIVE SEQUENCE

    SENDF StackPtr.Pop();
    RECEIVE
       ON StackDef.Ok(TopOfStack) WHERE SENDER() = StackPtr THEN ...
       ON StackDef.Empty () WHERE SENDER() = StackPtr THEN ...
       END;

B. REQUEST STATEMENT

   REQUEST StackPtr.Pop ()
       ON Ok (TopOfStack) THEN ....
       ON Empty () THEN ....
       ........

C. SINGLE-LINE REQUEST

   StackPtr.Pop (//TopOfStack)

D. FUNCTIONAL REQUEST

   X = StackPtr.Pop() + Y
```

Figure 2. Request Abbreviation Chain

Further abbreviation requires that processes not only declare message types, categorized as inputs and outputs, but also that they relate inputs to outputs, that is, that they indicate the outputs which may result from each input. Then, when only one response can be obtained from an input, a single-line, call-like request can be used:

```
Pointer.InputMsgType(input-msg//response-msg-rcv-list)
```

The response message is assumed to be of the single possible type. The applicability of this form can be broadened by distinguishing (declaratively) between normal and error responses, with the normal response assumed in single line requests. For example, if the "Ok" response to the stack "Pop" operation is classified as normal, and the "Empty" response as an error, then the single-line-request shown in Figure 2C is acceptable. "TopOfStack" is assumed to be a message reception list for a response of type "Ok". If a response of type "Empty" is received, it is treated as an exception.

The last abbreviation, shown in Figure 2D, unifies subroutine and function reference. If an input message has only one normal response, and that response contains exactly one element, then the request may be made as a functional reference. The single-element response is the value of the reference.

5. Subsuming Procedure-Like Referencing

When only one request is to be made of a programming unit, it would be desirable to allow procedure-like reference, in the sense of not requiring explicit create and destroy operations. For example, if cosine is the definition of a data abstraction, it should not be necessary to code the sequence:

```
P = NEW(cosine)
... P.(x)[2]
DESTROY(P)
```

if the instance is to be referenced only once.

To do this, the form "definition%msg-type" is used, in the request statement and its abbreviations. Thus the reference

```
cosine%(x)
```

specifies the sequence: (a) create an instance of cosine, (b) make the indicated request, (c) destroy the instance after accepting the response.[3]

6. Processes and Procs

The communication form abbreviations developed in the last section are desiged for object reference convenience, allowing asynchronous processes to be accessed by functional reference forms (when appropropriate). In this section, the perspective adopted is that of the referenced object, and forms are developed for more convenient acceptance of, and response to, messages.

The problem addressed can be stated as follows. The RECEIVE and SEND statements defined earlier provide considerable flexibility, allowing, for example, a process to accept and/or respond to messages not in order of arrival. However, if a process is to simply accept requests in order of arrival, and respond to those requests before accepting others, these statements are cumbersome and hide intent [14], especially as compared with the implicit argument acceptance and return structures of classic data abstractions and procedures.

[2] The form P.(x), with message type omitted, can be used to reference objects having only one declared input message type. It is assumed that instances of cosine fall into this class.

[3] Another alternative is to use the form "NEW(cosine).(x)" to simultaneously create and reference instances, and to rely on system mechanisms for the destruction of instances to which no pointers exist. The selected alternative seems somewhat more explicit of intent, and (correctly) causes any subsequent reference to the instance (possible if the instance distributes pointers to itself) to be understood as an error.

To provide for this case, another programming unit is introduced, called a "proc." The distinction between processes and procs is explained as follows: while processes have a single process-managed queue, procs have two queues, one system-managed and one proc-managed. Message types declared as inputs to the proc are placed on the system-managed queue, and activate the proc when quiescent. A proc must begin by receiving the activating message. While it is active, it may send and receive messages (from its general queue) using SEND/RECEIVE and request forms. The proc remains active until it responds to the activating message, at which point it is quiesced.

To receive an activating message; and send the associated response, a proc uses specialized syntax in place of a RECEIVE/SEND sequence. However, the specialized syntax, shown in Figure 3 is related to RECEIVE/SEND, and, at the same time, is more succinct.

The SRCV statement (Figure 3A) can be understood as a RECEIVE statement which draws messages from the proc's system-managed queue, in order of arrival. Since the queue may contain only messages whose type is defined locally, the local/foreign distinction is not needed. SRCV can be further abbreviated to ERCV (Figure 3B) if the proc declares only one input message type.

The response to an activating message is sent by the RETURN statement (Figure 3C) which sends a message of the indicated local type to the sender of the last activating message received, and begins to wait for the next one. The "REENTER label" clause is optional, and is discussed further on.

Procs, while similar to processes in form, can be written with the clarity of classic abstraction definitions, in which each operation is defined by a separate procedure. To do this, the body of a proc is written as a single SRCV statement. All processing for a particular input message type is specified within a single ON clause of that statement:

```
SRCV
     ON msgtype THEN DO
                all processing for message
                RETURN
                END DO
      ....
END SRCV
```

The optional "REENTER label" clause of the RETURN statement specifies that the next activating message is to be received not at the beginning of the code, but rather at the indicated label (which must identify an SRCV or ERCV statement). This option allows procs to subsume the control-state retention capabilities of co-routines (which preserve their control states between activations [4]) and of path expressions (which constrain the sequencing of abstraction operation requests [1, 3]).

Processes inherently subsume these capabilities because they determine at what points they receive messages, and what messages are acceptable at

```
A. SRCV Statement

   SRCV
      ON msgtype1 (rcv-var-list1) THEN stmt1;
      ON msgtype2 (rcv-var-list2) THEN stmt2;
      ......
      END;

B. ERCV Statement

   ERCV (rcv-var-list)

C. RETURN Statement

   RETURN msgtype (msg-element-list) REENTER label
```

Figure 3. Procs: Activating Message Reception and Response

each point. The REENTER clause extends this characteristic to procs, without the coding overhead and verbosity of full-scale processes. More generally, it allows procs to be coded in the style of very clear finite state machine specifications, using SRCV statements to represent states, and REENTER clauses to represent transitions.

7. Discussion

The development to this point provides, to a large degree, the facilities of functions, subroutines, data abstractions, asynchronous processes, co-routines and path expressions with only two, related, programming units and a chain of reference form abbreviations.

The two programming units are processes and procs. Procs are similar to processes, with the basic difference being that a proc uses more succinct structures to accept its declared inputs, and send its declared responses.

The basic communication statements proposed are SEND/RECEIVE, with REQUEST, single-line request, and functional-form request, and their invoke (create, reference, destroy) equivalents, used as shorthand forms.

Most of the forms suggested are derived from forms in existing languages (as mentioned above, [2] is an excellent survey). Furthermore, the attempt to encompass different types of programming unit behaviors and communication possibilities without inordinate syntactic and semantic disjointedness is not new. The significance of the proposal lies, rather, in the number of variations subsumed without either disjointedness or functional limitations.

The unification of functions and subroutines is achieved as early as ALGOL68 [21], which substitutes a single programming unit - the procedure.

An ALGOL68 procedure always returns a result, which may be declared to be vacuous. Procedures with non-vacuous results correspond to functions, but may be referenced by call, as long as the result is irrelevant.[4]

However, more recent major languages do not preserve this unification. Also, while they do attempt to limit communication-form differentiation, this is often accomplished by limiting functionality. For example, ADA™[5] [20] provides asynchronously and synchronously executing programming units, but no special statements for communicating with asynchronous units are used. In the ADA rendezvous mechanism, request messages are sent as remote procedure calls, and responses are sent automatically, at the end of the rendezvous, to the last sender. This reduces the number of statements needed, but limits concurrency, and does not allow for out-of-order responses.

One language which embodies significant communication construct integration is NIL [19]. Like the processes discussed here, NIL processes are potentially asynchronous abstractions, which may be referenced both synchronously and asynchronously. However, the actual communication forms used differ, with those of NIL reflecting a rather unique, and interesting, approach to correctness and security.

Subsumption Extent. The facilities proposed subsume those of the larger set (subroutines, functions, data abstractions, processes, co-routines, and path expressions) only "to a large degree", not entirely. However, the subsumed facilities are generally those considered most important. For example, the "REENTER label" clause provides a co-routine-like capability, in that it allows a collection of procs to retain their control states between activations. It does not allow substitution of activations within a call stack, but use of this aspect of co-routines is probably inadvisable.

Similarly, the data abstraction equivalents proposed do not completely subsume the properties of procedures. Procedure-like reference succinctness is preserved, as is the ability to define generic operators, in that the same input message type may be declared by any number of process/proc definitions.[6] However, an exact analogy of procedure-based generic operator definition, in which all arguments participate in operator determi-

---

[4] The function/subroutine unification proposed in section 4 can be adapted to traditional argument passing to produce a more powerful capability than that of ALGOL68. To do this, parameters to a procedure must be characterized (by declaration) as "used", "updated", or "used and updated". Then, if the last parameter to a procedure is declared as updated (only), the procedure can be referenced as a function with the last argument omitted.

[5] ADA is a registered trade mark of the U. S. Government (Ada Joint Project Office).

[6] And operations of a data abstraction can be referenced in infix form. For example, in SMALLTALK [24], "A+B" is interpreted as "A.+(B)", i.e., the first operand is understood as the name of the data abstraction providing the operation.

nation, is not proposed. While it could be provided by allowing message elements to further qualify message type names (if extremely strong typing is assumed), it is instead suggested that the use of procedure-based generic operators, and the latter adaptation, is questionable. The use of an unqualified generic operation can be misleading, as it tends to suggest a single meaning to the reader of a program. In contrast, the form "object.operation" makes explicit the dependence of the meaning of the operation on the object referenced.

The most important features of more conventional programming units not subsumed by the proposal, to this point, relate to synchronization and performance. None of the implicit synchronization facilities provided by synchronous calls or rendezvous mechanisms are included. What might be considered a performance-related omission is that of communication by parameter (as opposed to messages). Synchronization is addressed in connection with more general integrity issues, in the next section. Performance issues are discussed in the section following that.

8. Integrity Considerations

The specification of integrity facilities compatible with the programming-unit/communication structures proposed is currently incomplete. However, some aspects can be sketched, in overview form.

The integrity facilities under consideration are based on the data base management concept of a transaction [7], whose wider applicability has been the subject of extensive discussion and of concrete proposals for adaptation [9, 12, 13, 16, 23]. The reason for adoption of this direction is that transactions can be used for multiple aspects of integrity - atomicity, error handling, recovery, and synchronization - eliminating the need for separate facilities.

The adaptation being considered here consists of specifying transactions as qualified blocks, e.g.,

```
     Label: DO TRANS ONFAIL Label1
            .......

            END
```

where DO ... END is a block form, and the optional "ONFAIL Label" clause identifies another transaction block to which control is to be passed if the subject transaction fails. Transactions may be nested.

In general, when a transaction fails (because of program error, explicit request, or interlock condition), the effects of the transaction on its environment are backed out, control is passed to the indicated (ONFAIL) alternative, and built-in functions may be referenced (in the alternative block) to determine the reason for the failure.

Thus transactions, within individual program units, provide atomicity, error handling, and recovery. The handling of these aspects across program units, along with synchronization, requires the specification of the relationship of transactions to programming units.

The first question to be dealt with in this regard is what transaction is being executed when an object carries out a request. One approach is to classify objects as either executing their own transactions, or executing the transactions of others. Since procs (in contrast to processes) are always executing an identifiable request, it seems natural to specify that procs always (automatically) execute the transactions of others, while processes normally execute their own transactions.

Other questions relate to the treatment of multi-threaded transactions (transactions with contained asynchronism), which can be created by processes activating one or more procs while continuing to execute. Questions in this area have not yet been answered satisfactorily. Until they are, the generality of the proposal is limited by restricting requests made of procs to synchronous forms (REQUEST and REQUEST shorthand). (Note, however, that processes may be referenced by both synchronous and asynchronous forms.)

Thus only processes may define outermost (effectively non-nested) transactions. A request to a proc constitutes a kind of nested transaction, at least in the sense that:

- The request is either executed to completion, or its effects are (with some exceptions) backed out.

- During execution of a request by a proc A, any requests made by A are further nested in this sense.

Procs may define their own, more deeply nested, transactions to allow them to handle their own errors.

A proc may stand in one of two relationships to processes, established when the proc is created. It may be owned by a process, or it may be shared among processes. An owned proc is considered part of its owner, and is recovered together with the recovery of any transaction of its owner. Requests to owned procs are thus fully nested transactions, in that their effects are recovered even after return if the enclosing transaction fails.

A shared proc is, in general, recovered only if it fails during a particular request. However, like a file (which can be modelled by a proc), a shared proc may be reserved by a process for the duration of a transaction (by an extension to the transaction header clause). In this case it is considered part of the process during that transaction, and is recovered if the transaction fails.

Additional aspects of transactions, not discussed here, include provisions for processes to view copies of procs (to avoid the need for reservation), and provisions to allow processes to act if they were processing the transactions of other processes. This is needed, for example, to allow processes to perform the role of data base management systems.

In summary, the transaction facilities envisioned at this point seem to subsume many facilities in the areas of atomicity, synchronization, error handling, and recovery, and seem consistent with the concurrency and communication constructs described. It would be preferable, however, to define a transaction facility which allowed for intra-transaction concurrency, both to restore the consistency of procs and processes, and for other reasons not dealt with here.

9. Performance Questions

The primary device used to relate process and abstraction reference patterns is a message-based communication model, in which message elements are copied into local variables of the receiving object. This would seem to have serious implications for performance. If sending a message causes all message elements to be copied, then a single REQUEST statement, which involves the sending of two messages (the request and the response), would seem to require two copy steps. This contrasts badly with communication by reference, which does not require any copying at all.

The restriction of proc referencing to synchronous forms, made for purposes of defining a coherent integrity facility, also forms the basis for the alleviation of this problem. It ensures that local variables referenced in request message elements are not modified during request processing, and, thus, that they may be passed, and referenced, by internal pointer.[7]

In most cases the restriction to synchronous reference is sufficient to ensure that no copying is required on the input side, i.e., in the reception of a request. One appropriate method relates to those used for differential file [15] maintenance in data base management systems. All local variables in a proc which are used for request message reception are understood to be referenced by internal pointer. When a message element is received (by pointer) into a local variable, a directory of the blocks (some storage size used as a subdivision) for the variable is constructed. When a change is made in the proc to the variable, only the affected block is copied, and the directory updated accordingly.[8]

This can be done as long as the value of the local proc variable need not be saved between invocations. This would be true, for example, if it is assigned a new value every time a request is accepted. Otherwise copying into local storage is needed to avoid inadvertent sharing.

[7] If it were not for the transaction problem, the adjustment would consist of classifying of some procs as "sprocs", and applying the synchronous reference restriction only to them.

[8] The differential-file-like approach is also a means of providing the transaction recovery described in the preceding section.

Avoiding copying on the response side is more complex, but still tractable. Suppose that proc definitions (optionally) included the specification of input/output combinations to be considered as possible update cases. For example, one such specification might group input message type A, element 1, with output message type B, element 2.

All references to those message types in the proc must use the same local variable in the specified positions. For example, if an SRCV statement receiving message type A receives element 1 into variable V1, then all SRCV statements receiving message type A must receive element 1 into V1, and all RETURN statements returning message type B must specify V1 as element 2 of the return message. Furthermore, as above, the proc must be written such that value of V1 need not be saved between invocations.

Given those specifications, no copying is needed for local variables of the requestor also used in those positions, e.g., X in

```
      REQUEST PTR.A (X, ....  );
              ON  B (Y, X, ...);
              .......
```

The value returned for X within message type B will be the directory pointer for B constructed earlier. The receipt of the returned value consists of copying only the modified blocks. If, however, the variables used in the two positions do not match, the response message element must be copied.

10. Concluding Remarks

The programming unit types and inter-object communication statements proposed here allow many types of object interactions, ranging from asynchronous message processing to traditional function calls, to be expressed in a highly consistent fashion. This is accomplished, for the most part, by the use of a message-passing model throughout, and by the introduction of a request structure, useful in its own right, to bridge the gap between synchronous and asynchronous forms.

The statements described form a part of a very-high-level language called PL/IDE (programming language for an integrated development environment), currently being defined. The integrity-related aspects of the language and execution environment are, at this point, only partially specified. The capabilities contemplated seem compatible with the described programming unit behaviors, but are overly constraining. More work is needed in this area.

11. Acknowledgements

I thank Dan Berry, Adolfo Di Mare, Nan Shu, John Sowa and Rita Summers for their valuable comments and suggestions.

12. References

1. S. Andler, "Predicate Path Expressions" Conference Record of the Sixth ACM Symp. on Principles of Prog. Lang., (January 1979) 226-236

2. G.R. Andrews, F.B. Schneider, "Concepts and Notations for Concurrent Programming", ACM Computing Surveys 15,1 (March 1983) 3-39

3. R.H. Campbell, A.N. Habermann, "The Specification of Process Synchronization by Path Expressions", Lecture Notes in Computer Science, 16, Springer Verlag, New York 1974 (89-102)

4. M.E. Conway, "Design of a Separable Transition-Diagram Compiler", Comm. of the ACM, 6 (1963) 396-408

5. O.-J. Dahl, K. Nygaard, "SIMULA - An ALGOL Based Simulation Language", Comm. of the ACM (September 1966)

6. E.W. Dijkstra, "Guarded Commands, Nondeterminism, and Formal Derivation of Programs", Comm. of the ACM, 18, 8 (August 1975) 453-457

7. K.P. Eswaran, J.N. Gray, R.A. Lorie, I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", Comm. of the ACM, 19,11 (November 1976) 624-633

8. J.A. Feldman, "High Level Programming for Distributed Computing", Comm. of the ACM 22,6 (June 1979(353-368

9. W.H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems", ACM Computing Surveys 13,2 (June 1981) 149-183

10. B. Liskov, "An Introduction to CLU", Computation Structures Group Memo No. 136, Laboratory for Computer Science, Mass. Institute of Technology, Cambridge, Mass. (1976)

11. B. Liskov, "Report on the Workshop on Fundamental Issues in Distributed Computing", ACM Operating Systems Review 15,3 (July 1981) 9-38

12. D.B. Lomet, "Process Structuring, Synchronization, and Recovery Using Atomic Actions", Proc. ACM Conf. on Language Design for Reliable Software, SIGPLAN Notices 12,3 (March 1977) 95-100

13. B. Randell, P.A.Lee, P.C. Treleaven, "Reliability Issues in Computing Systems Design", ACM Computing Surveys 10,2 (June 1978) 123-165

14. M.L. Scott, "Messages vs. Remote Procs is a False Dichotomy", SIGPLAN Notices 18,5 (May 1983) 57-62

15. D.G. Severance, G.M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases", ACM Trans. on Database Systems 1,3 (September 1976) 256-267

16. A.Z. Spector, P. M. Schwarz, "Transactions: A Construct for Reliable Distributed Systems", ACM Operating Systems Review 17,9 (April 1983) 18-35

17. J.A. Stankovic, "Software Communication Mechanisms: Proc Calls versus Messages", IEEE Computer 15,4 (April 1982) 19-25

18. P.D. Stotts Jr., "A Comparative Survey of Concurrent Programming Languages", SIGPLAN Notices 17, 10 (October 1982) 50-61

19. R.E. Strom, S. Yemini, "NIL: An Integrated Language and System for Distributed Programming", Proc. SIGPLAN '83 Symp. on Prog. Lang. Issues in Software Systems, SIGPLAN Notices 18, 6 (June 1983) 73-82

20. U.S. Department of Defense, "Reference Manual for the ADA Programming Language", MIL-STD 1815A (February 1983)

21. A. van Wijngaarden, B.J. Mallioux, J.E.L.Peck, C.H.A. Koster, "Report on the Algorithmic Language ALGOL 68", Numerical Mathematics 14 (1969) 79-218

22. P. Wegner, S.A. Smolka, "Processes, Tasks, and Monitors: A Comparable Study of Concurrent Programming Primitives", IEEE Trans. on Software Engineering SE-9,4 (July 1983) 446-462

23. W. Weihl, B. Liskov, "Specification and Implementation of Resilient, Atomic Data Types", Proc. SIGPLAN Symp. on Prog. Lang. Issues in Software Systems (June 1983) 53-64

24. Xerox Learning Research Group, "The Smalltalk-80 System", Byte 6,8 (August 1981) 36-48

SCIENTIFIC CENTER REPORT INDEXING INFORMATION

1. AUTHOR(S) :
Paula S. Newman

2. TITLE : An Approach to Unifying Inter-Object Communication

3. ORIGINATING DEPARTMENT
60G

4. REPORT NUMBER
G 320-2737

5a. NUMBER OF PAGES
15

5b. NUMBER OF REFERENCES
24

9. SUBJECT INDEX TERMS

Programming Language
Data Abstraction
Inter-process Communication
Distributed Processing,
Object-Oriented Language

6a. DATE COMPLETED
October 15, 1983

6b. DATE OF INITIAL PRINTING
August 1984

6c. DATE OF LAST PRINTING

7. ABSTRACT :

Both data abstractions and asynchronous processes are useful types of programming units, as are traditional functions and subroutines. However, the incorpoation of all four types in a single language poses problems of language size and consistency.

This paper sketches an approach to providing functional equivalents of these programming units, togather with some aspects of co-routines, with limited syntactic and semantic variation. Two types of programming units, called processes and procs, and a set of closely-related inter-unit communication mechanisms are developed. Provisions for integity are addressed briefly.

8. REMARKS :

IBM LOS ANGELES SCIENTIFIC CENTER OUTSIDE PUBLICATIONS

FOR THE YEAR 1982

F. W. BALL, J. G. SAKAMOTO, "Supporting Business Systems Planning Studies with the DB/DC Data Dictionary", IBM Systems Journal, Vol. 21, No. 1 (1982).

F. W. BALL, J. G. SAKAMOTO, "Supporting Business Systems Planning Studies with the DB/DC Data Dictionary", Database Management, November-December issue, Auerbach Information Management Series (1982).

S. A. JUROVICS, "Daylight, Glazing, and Building Energy Minimization", ASHRAE Transactions, Vol. 88, Part 1 (1982).

S. A. JUROVICS, "The Impact of Daylight Utilization on Thermal and Lighting Loads: A Case Study", presented at ASHRAE Annual Meeting, Toronto, June 1982.

J. M. MARSHALL, "Systems Architect Apprentice System (SARA)", Proceedings of Corporate Symposium on Structured Logic, Yorktown, November (1982).

P. S. NEWMAN, "Towards an Integrated Development Environment", IBM Systems Journal, Vol. 21, No. 1 (1982).

M. M. PARKER, "Enterprise Information Analysis: Cost-Benefit and the Data Managed System", IBM Systems Journal, Vol. 21, No. 1 (1982).

FOR THE YEAR 1982 (Continued)

M. M. PARKER, "Toward Cost-Benefit Analysis of Data Administration", SHARE Proceedings, Vol. 1, August 1982.

J. G. SAKAMOTO, "Use of DB/DC Data Dictionary to Support Business Systems Planning Studies: An Approach", The Economics of Information Processing, Vol. 1, Publisher: John Wiley & Sons Publishing Co. (1982).

J. G. SAKAMOTO, "Use of DB/DC Data Dictionary to Support Business Systems Planning Studies: An Approach", Symposium on The Economics of Information Processing, Sponsored by the IBM Systems Research Institute (1982).

N. C. SHU, "Specifications of Forms Processing and Business Procedures for Office Automation", IEEE Transactions on Software Engineering, Vol. SE-8, No. 5 (1982).

N. C. SHU, D. M. CHOY, V. Y. LUM, "CPAS: An Office Procedure Automation System", IBM Systems Journal, Vol. 21, No. 3 (1982).

R. C. SUMMERS, "Computer Security", chapter of Computer Handbook, Publisher: Van Naustrand Reinhold Publishing (1982).

FOR THE YEAR 1983

C. WOOD, E. B. FERNANDEZ, "Minimization of Demand Paging for the LRU Stack Model of Program Behavior", Information Processing Letters, Vol. 16, No. 2 (1983).