at any point. In addition, it is possible to experiment with the efficacy of minor variations such as a deeper or shallower search for candidates, more frequent global iterations between phases 1 and 2, the use of coding schemes to prevent candidates from reentering candidate lists immediately after being rejected, etc. This type of manageability makes it easier to understand how the algorithm behaves and tailor the algorithm to particular types of problems.

Much remains to be done. The degree of nonoptimality caused by iterating between phases 1 and 2 rather than combining them should be explored. The handling of key orders in the current algorithm must be revised and made more general (a maximum of two keys are allowed for each dataset in the current implementation). Further refinement of methods for estimating execution costs would increase the applicability of the results.

This paper has presented an explanation and demonstration of an approach for producing answers to a puzzle that confronts system designers every day. The next step is to apply this approach in facilitating the design of real systems. Whether human designers really need help of this type is actually an empirical question that can be studied by comparing the efficiency of actual system designs with that of designs generated by algorithms such as the one presented here. Whether or not such algorithms outperform human designers, the need for their development is clear. To implement automatic programming capabilities in which people describe the substantive processing to be accomplished and machines translate such descriptions into executable code, design choices will have to be made automatically.

Received November 1977; revised January 1979

References

1. Alter, S. Optimizing the behavior of application systems. Proc. Sixth Annual Conf. of the Computer Measurement Group, San Francisco, Oct. 8-10, 1975, pp. 192-211.
2. Gerritsen, R. A preliminary system for the design of DBTG data structures. *Comm. ACM 18*, 10 (Oct. 1975), 551-557.
3. Hoffer, J. An integer programming formulation of computer data base problems. TR #1-74, Dept. of Management Studies, Case Western Reserve U., Cleveland, Ohio, Oct. 1974.
4. Kornfeld, W. Methodology for optimization in automatic programming systems. Unpub. B.S. Th., M.I.T., Cambridge, Mass., June 1975.
5. Low, J. Automatic coding-choice of data structures. Memo AIM-242, Stanford Artif. Intell. Lab., Stanford, Calif., Aug. 1974.
6. Mitoma, M. F., and Irani, K. B. Automatic database schema design and optimization. Proc. Int. Conf. on Very Large Databases, 1975, pp. 278-321 (available from ACM, New York).
7. Morgenstern, M. Automated design and optimization of management information systems software. Unpub. Ph.D. Th., M.I.T., Cambridge, Mass., 1976.
8. Nunamaker, J. F., Nylin, W. C., and Konsynski, B. Processing systems optimization through automatic design and reorganization of program modules. In *Information Systems*, J. T. Tou, Ed., Plenum, New York, 1974, pp. 311-336.
9. Ruth, G. Automatic design of data processing systems. Third ACM Symp. on Principles of Programming Languages, Atlanta, Georgia, Jan. 1976, pp. 50-57.
10. Yao, S. B., and Merten, A. G. Selection of file organization using an analytic model. Proc. Int. Conf. on Very Large Databases, 1975, pp. 255-267 (available from ACM, New York).

Programming Languages
J.J. Horning Editor

High Level Programming for Distributed Computing

Jerome A. Feldman
University of Rochester

Programming for distributed and other loosely coupled systems is a problem of growing interest. This paper describes an approach to distributed computing at the level of general purpose programming languages. Based on primitive notions of module, message, and transaction key, the methodology is shown to be independent of particular languages and machines. It appears to be useful for programming a wide range of tasks. This is part of an ambitious program of development in advanced programming languages, and relations with other aspects of the project are also discussed.

Key Words and Phrases: distributed computing, modules, messages, assertions
CR Categories: 4.22, 4.32

1. Introduction and Overview

When the University of Rochester Computer Science Department was started in 1974, our initial research goals included taking a really serious look at programming languages. There were two underlying assumptions: (1) that programming languages had changed little in the previous decade despite advances in many related areas and (2) that one could envision compilers as sophisticated as the best current artificial intelligence programs. We began by trying to isolate the most important concepts available in programming systems to see how they interacted with each other. The project was called PLITS (Programming Language in the Sky) and, al-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
Author's address: Dept. of Computer Science, Mathematical Sciences Building, University of Rochester, Rochester, N.Y. 14627.
© 1979 ACM 0001-0782/79/0600-0353 $00.75.

though it has come down a little closer to the ground, the name has stuck. The study is far from complete, but we claim to have an interesting preliminary result, namely that a judicious incorporation of three constructs: modules, messages, and assertions, can lead to programming language systems of considerable power and elegance.

This paper concentrates on the implications of the continuing advance of distributed computing on the design for high-level programming languages. Many problems of distributed computing (DC) do not arise in conventional programming. Solutions of these problems lead in a natural way to new programming language constructs. A distributed computation is spread among several computers which are assumed to be connected by some communication paths. For the foreseeable future these communication paths will be less reliable and have lower bandwidth than is available in the processors themselves. This leads us to expect that DC programs will be made up of largely self-contained *modules* which will share very little information directly. One would also want to have the communication between modules be some asynchronous *message* protocol, rather than subroutine or coroutine calls where one module would always have to wait for a response from the other. It appears to us that the module-message paradigm is inherently well suited to DC and is likely to appear in some form in any proposed high level language for DC. Even if we restrict consideration to a single machine, modules and messages seem to provide important advantages over existing languages and abstraction proposals such as Alphard [29], CLU [27] and Euclid [24].

The choice of the primitive high level language constructs for PLITS was not made a priori. Starting from a survey of the "powerful ideas" of programming systems, we attempted to see if there were inherent incompatibilities among them. There were only a few such situations, such as the incompatibility between separately compilable procedures and code optimization through procedure integration. A significant conclusion was that parallelism and data sharing are inherently difficult to combine effectively. Our interest in networking and our work on a message-based operating system [2] strongly suggested the idea of messages, but did not totally solve the problem of how to communicate without data sharing.

The decision to have symbolic names as the basis of communication seems obvious in retrospect, but was difficult to arrive at. By using *names* of message slots as the shared notion and having no name-valued constructs, we are able to have flexible communication among modules with no sharing of storage. It was clear from work in automatic programming and verification that more declarative information was needed—hence the general notion of assertions was included. Transaction keys were added later as a solution to several problems in selective reception, protection, flow control, and multiplexed servers. The bulk of the paper is concerned with the important questions that arose from attempting to implement this set of ideas and use them to solve hard problems. Although many difficult questions remain, enough clean solutions have been found to convince us that there is something fundamentally sound in the PLITS world view.

The use of the modules and messages can be illustrated by some simple examples. A module is a self-contained entity, like a Simula or Smalltalk *class*, a SAIL *process*, or a CLU *cluster*. It is not important which programming language is used to encode the body of a module; we will explicitly have to account for the case in which various modules are coded in different languages on a variety of machines. The presentation here is based on a specific choice of body language: PASCAL as defined in [22]. This should be easier to understand than definitions based on abstract syntax.

Modules communicate with one another solely through *messages*. In order to have communication, there must be something that is understood by both communicating modules. The shared element in PLITS is a *name* which may be thought of as an uninterpreted string of characters. A message is a *set* of (name ~ value) pairs called *slots*. The value portion of a slot will be an element of some primitive domain (e.g. integers) whose representation is also generally understood.

The modules of any PLITS system must compose, send, receive, and decompose messages. For this purpose, we add some data types and operations to PASCAL or any other body language. In the PASCAL case, the primitive data types are extended to include module and message. Each module explicitly declares (**public**) every slot name that it deals with along with the data type of that slot. As we will see later, there is a process analogous to linkage-editing that ensures that public slot names are used consistently. The word "public" refers to sharing within a job but not across jobs; the notion of job is extended to distributed job (DJOB) in Section 3.

PASCAL has four primitive types: integer, Boolean, char, and real, and three structure types: arrays, records, and sets. We will ignore sets and subrange types, but will use enumeration types. The PLITS constructs do not have a particular obvious encoding in PASCAL (if they did, we might not need them). We have chosen to represent modules as additional **type** constructors analogous to **array** or **record**. In keeping with PASCAL's fairly strong typing, each kind of module class will be a different type. Part of PLITS is deeply incompatible with strong typing (much more will be said about this below). In particular, we want to allow some slots to have as the data type the union of all module types, which we denote **module**. It is crucial that any module (written in PASCAL-PLITS or some other X-PLITS) be able to send a message to any other.

We also add a type **message** which is like the PASCAL type **record** except that it has all the **public** slot names of the module as potential field names. The global constructs in PASCAL-PLITS will be the public slot

Fig. 1.

```
 1  Const George = mod                      Const Fibonacci = mod
 2    Begin                                   Begin
 3    Public Recipient: module                Public Recipient: module
 4           Object: integer                         Object: integer
 5    var I, J, Next_Fib: integer             var This, Last, Previous: integer
 6        Mess1, Mess2: message                   Request: message
             .                                Last := 0
             .                                This := 1
             .                                While True Do
 7    Send message (Recipient ~ Me) To          Begin
                  Fibonacci                       Receive Request
 8    Receive Mess2 From Fibonacci                Previous := Last
 9    Next_Fib := Mess2·Object                    Last := This
             .                                    This := Last + Previous
             .                                    Send message (Object ~ This) To
             .                                                  Request·Recipient
10    End                                       End
                                              End
```

Fig. 2.

```
 0  Const Fibonacci = mod
 1  Begin
 2    Public Object: integer
 3    Public Recipient, Complaint_Dept, Complainer: module
 4    Public Problem: problem_type
 5    var Request, My_Complaint: message
 6        Complainee: module
 7        This, Last, Previous, Biggest: integer
 8    Last := 0; This := 1; Biggest := 2 ↑ 35 − 1
 9      My_Complaint := message (Problem ~ Overflow,
10                               Complainer ~ Me
11                              )
12    While True do
13      Begin
14        Receive Request
15        Previous := Last;
16        Last := This;
17        If Biggest − Last > Previous
18        Then Begin This := Last + Previous;
19               Send message (Object ~ This) To
                   Request·Recipient About Request·About
20             End
21        Else Begin
22               Put (Recipient ~ Request ·Recipient)
                   In My_Complaint;
23               Complainee :=
                   If Present Request·Complaint_Dept
                   Then Request·Complaint_Dept
                   Else City_Hall;
24               Send My_Complaint To Complainee About
                   Request·About
25             End
26      End While Loop
27  End
```

names, module type declarations (including **constant** modules) and some enumeration types. There are to be no global variables of any type.

For a first example (Figure 1), suppose there were a module, Fibonacci, which provided the service of supplying consecutive positive Fibonacci numbers, and a module, George, which wanted to make use of this service. (George and Fibonacci are actual module names, not module prototypes or class definitions.)

George is declared to be a constant of type **module** (using the **mod** construct) and to have two **public** slot names: "Recipient" of type **module** and "Object" of type **integer.** The type constructor **mod** is analogous to the PASCAL constructor **record**. The primitive type **module** denotes the union of all types declared with a **mod** constructor; PASCAL does not have an analogous union of all records. We will refer to types declared using **mod** as "module types." The variables Mess1 and Mess2 are declared to be of type **message**. The **Send** statement in line 7 uses the PASCAL constructor syntax to build a message and send it. After sending the message to Fibonacci, George is automatically suspended until a message from Fibonacci is received (line 8). This particularly simple control regime is equivalent to a subroutine call. Since Fibonacci has the same two public slot names as George, they can communicate. Fibonacci is a server module that waits for a request, computes the next value, and sends an answer message to the module named in the request. Although this is George in the example, it could be a general continuation.

One of the major goals of the PLITS effort is the development of techniques for programming reliable systems. Since each PLITS module has complete control over its internal state and the messages it accepts, one can program any module so that it never reaches an undesirable internal state. Of course, a module which contains internal protection against a wide variety of external errors can become quite bulky. There is a fundamental design tradeoff between hardening individual modules and guaranteeing at the system level that certain global conditions cannot arise. A detailed discussion of these issues is beyond the scope of this paper (cf. [16]). However, in Figure 2, we present a somewhat more protected Fibonacci module which will not be subject to integer overflow. Figure 2 also includes instances of several additional PLITS constructs.

The first new notion occurs on line 4 where a public slot name of type "*problem_type*" is declared. The type *problem_type* is a fixed collection of uninterpreted sym-

bols exactly like the PASCAL "enumeration" type. There will be several public enumeration types in a PLITS system. In lines 9–11, a prepackaged message is assembled and stored in the message variable, *My_Complaint.* In lines 19 and 24, the **Send** statements contain an extra component **"About"**; this specifies a particular **transaction** key in a way which is described below. The other new code is in lines 21–27; the Recipient slot of *My_Complaint* is filled in from the Request. If there is a *Complaint_Dept* (cf. [20]) slot in this request, the module which is its value will be sent the complaint. Otherwise some default complaint handler, *City_Hall,* will hear about it. The name of the Recipient module (which may have been awaiting an answer) is passed along to the *Complaint_Dept,* because there might be some appropriate response to the problem. For example, there could be some double precision Fibonacci module which would be able to return an appropriate value if George were prepared to accept it. This could require that George handle the double size integers; that is not very hard to arrange, for example, by an extra slot for the high order part.

There is a more interesting problem in the control discipline used in the coding of the module George given in Figure 1. The statement on line 8 is:

Receive Mess2 **from** Fibonacci

But we saw in the expanded Fibonacci module of Figure 2 that there might be an error recovery module that would supply the answer if Fibonacci could not. The coding style of line 8 requires that the answer be conveyed back to Fibonacci and then to George, but there is nothing to be gained by retracing our steps. To solve this and a number of other control problems, we will add one more construct, **transaction**, to PLITS. Intuitively, a transaction is a unique key which can be used in the regulation of message traffic. A transaction is required in the **About** slots of the **Send** statements of lines 19 and 24 of Figure 2. In this case, the outgoing transaction key is just the one accompanying the request message. If some *Complaint_Dept* were able to rectify the overflow problem, it could forward the correct answer to a slightly modified George. Our third example (Figure 3) shows a recoding of the module George which does selective reception on the transaction, Fibkey, which it originally provided. Figure 3 is an overview of a complete PASCAL-PLITS program containing the expanded version of Fibonacci, etc.

These introductory examples are intended to provide an intuitive notion of what is being proposed. In the next section a precise definition of PASCAL-PLITS is given, roughly following the style of [22]; this will be used for examples throughout the paper. Section 2 also shows how we capture key notions from several areas of computer science in a PLITS system. Section 3 considers the implementation problems arising in extending these ideas to distributed computing systems. In Section 4, we return to the consideration of a one-language PLITS system, more or less assumed to run on a single machine. Properties are introduced as a special kind of assertion which are an improvement on strong typing. Assertions are shown to be the key to verification and optimization of PLITS programs. Finally, we present a brief overview of debugging and exception handling in a PLITS environment. The material of Sections 3 and 4 summarizes a number of ongoing research efforts, for which references are provided.

A word about the current state of PLITS implementation seems to be called for. The basic concepts of PLITS were crystalized early in 1976 and some students attempted to carry out toy implementations. A serviceable SAIL-PLITS was implemented that summer by Jim Low using the SAIL macro facility; it has been used for running examples like those in this paper and for course problems and term projects. This version incorporates essentially the constructs described in Section 2. The underlying support system described in Section 3 has been implemented on our local network [2] under the direction of Paul Rovner, and is being used in current distributed systems [10] and artificial intelligence research [40]. Most of the ideas of Section 4 are being incorporated into Zeno [3], the base language for the Advanced Compiler project in our laboratory. A complete high level multilanguage multimachine PLITS system is under development, but has lower priority than the more basic efforts.

Fig. 3.

```
Program Everything
          Problem_type = (Overflow, Time Out,
                          Absent Slot, Illegal Module, ...)
  public Recipient, Complaint_Dept, Complainer: module
         Object: integer
         Problem: problem_type
         :
  const George = mod
         Begin
           Public Recipient: module
                  Object: integer
           var I, J, Next_Fib: integer
               Mess1, Mess2: message
               Fibkey: transaction
           Fibkey := New Transaction
           Send message (Recipient ~ Me)
             To Fibonacci About Fibkey
           Receive Mess2 About Fibkey
           Next_Fib := Mess2·Object
           :
         End
         :
  const Fibonacci = mod
                    : [cf. Figure 2]
                    End
  const City_Hall = mod
                    :
                    End
  :
  End
```

2. Definitions

2.1 A Precise Definition of Minimal PLITS

We now state more precisely what is proposed to be included in a PLITS system. The language described is somewhat more restrictive than one would want to use for the construction of large systems (cf. [2]), and the definition is being kept minimal so that the fundamental issues can be more easily addressed. For example, the current definition allows no block structure or other partitioning of the space of public slot names. A number of other straightforward extensions are mentioned in passing. Even so, we will omit all consideration of three important issues to be covered in Sections 3 and 4: computer networking, assertions and program transformation, errors, and other system issues.

We now proceed to a fairly precise definition of a PASCAL-PLITS. The purpose is to provide a fixed language in which to present subsequent PLITS examples, not to propose an extension to PASCAL. We will define the syntax of the additions locally rather than present modified syntax graphs. We will also (for now) be even less formal than [22].

Recall that a message is a set of name ~ value pairs, where the names and their associated types are publicly known. The first new construct is the public slot name definition, which is of the form:

Public S_1:T_1, S_2:T_2, ... S_m:T_m;

where the S_i are all public slot names and the T_i are their associated elementary types. An elementary type is either a PASCAL primitive type or **transaction** or a **module** type. We also add a type **message** which is similar to the PASCAL **record** type. The legal message constructors in module P are all of the form:

message ($S_j \sim X_j$, $S_k \sim X_k$, ...)

where each X_j is an element of type T_j and the slot name S_j:T_j has been declared **public** in P. This differs from the **record** constructor in that the field names are given explicitly in any order and need not all be present. Every message implicitly includes two extra slots: **Source** of type **module** and **About** of type **transaction.**

In analogy with PASCAL record field extractors, we define for a message M

$M \cdot S_i = X_i$

to hold, providing that the pair $S_i \sim X_i$ is an element of M. If there is no $S_i \sim X_i$ pair in M, an exception condition occurs; exceptions will be discussed in Section 4. We also extend the PASCAL **with** construct from implicitly naming the record to implicitly naming the message in the obvious way. This definition of **message** allows a module P to send and receive messages with slots not known to P but not to examine or alter these slots. There are some additional operations on a message:

Put $S_i \sim X_i$ **In** M
Remove S_i **From** M
$M \cdot S := X_i$

All of these require that X_i be of type T_i and that the slot S_i:T_i has been declared **public** in module P. The **put** statement adds the slot $S_i \sim X_i$ or replaces the existing value of S_i. The **remove** statement totally removes an $S_i \sim X_i$ if it is present; no error results if the slot is absent. The assignment statement updates the value of the slot S if it is present; otherwise an exception condition (see Section 4) is generated.

If M is a message, V is of a module type, and K is a transaction identifier, then there are valid statements of the form:

Sand M **To** V {About K}

In any PLITS messages are always sent by value (copied). There are also two predicates on message slots:

Absent S_i **In** M
Present S_i **In** M

The **Present** (**Absent**) predicate returns **True** if there is (is not) an $S_i \sim X_i$ pair present in M. The predicates of equality and inequality are extended to objects of **transaction, message,** and **module** type. This completes the discussion of message types from the internal PASCAL point of view. In addition to its importance for messages, the data structure consisting of a set of name ~ value pairs has many other good uses. We are currently calling this data structure an A-set (analogous to the LISP notion of A-list) and are exploring their use in a number of contexts.

An alternative PASCAL-PLITS syntax for the slots known to a module P would be achieved by declaring, in P, a local instance *Pmessage* of the type constructor **message**

```
type Pmessage = messageS1:T1
                       S2:T2
                        .
                        .
                        .
          end;
```

where the {S_i:T_i} are exactly the pairs that appeared in the **public** declaration in the definition used above. This definition of message is more PASCAL-like, allowing the compiler to carefully control what kind of messages a module could receive. What this definition gives up is the ability for a module to handle messages some of whose slots it knows nothing about. An important claim of this paper is that very strong typing is ineffective as a way to promote careful programming and that the PLITS methodology offers a way of saying more precisely what conditions a message must satisfy in order to be acceptable to a module. Similarly, one might want to have stronger restrictions on variables of type **module**. For example, a strongly typed language would have constructs like "fixed-length queue of real module." The proposed way of treating such issues with **properties** is discussed briefly in Section 4 and in detail in [16]. In general, PLITS does not currently support structured types in public slots (but see Section 3). In particular,

there can be no message slot of type **message**. If K is of type **transaction** and V is of **module** type, then

```
Receive <message exp.> {From V} {About K}
```

is the pattern for the four types of **Receive** statements. The internal axiomatic semantics of **Send** and **Receive** can be treated simply. A **Send** statement has no direct effect on the sending module. A **Receive** statement has the internal semantics of an assignment to a **message** variable. In many implementations, the failure of a message to arrive in a specified (perhaps by default) amount of time will result in an exception condition. The use of the axiomatic semantics for verification and the discussion of exception handling is presented in Section 4.

The sending and receiving of messages are central to any PLITS implementation and must be considered with some care; most of Section 3 is concerned with some issues arising in message-based systems. For our definition of PASCAL-PLITS, we need two semantic rules on message transfer. The first is that message queues are assumed to be unbounded, in the sense that the stack and the heap are considered unbounded in PASCAL and similar languages. Queue management, error detection, etc., are important system considerations but can be omitted at this stage of definition.

The second semantic rule concerns the preservation of order among messages. The rule is: for each triple (Sender, Transaction, Receiver) messages are guaranteed to arrive at the receiver in the same time order as they were sent from the sender. No other order properties can be relied upon. For all communication between a particular sender and receiver which does not have an ⟨**About** transaction⟩ clause, the system will assign a single fixed transaction identifier. Thus if no transactions at all are used between a sender and receiver, messages will all arrive in sequence.

A module type in PASCAL-PLITS is given by a declaration of the form:

```
type T = mod L; Begin S End
```

The module type notion does not have a direct counterpart in PASCAL. A module has some of the properties of a procedure and some of the properties of a record (like Simula classes, etc.). We have chosen to define it as a **type** constructor so we could easily have module classes and instances. The parameter list L is the same as that of a PASCAL procedure and is used to initialize new modules of the given type. All parameters are passed by value. The special case of a **constant** of module type was shown in Figure 3.

Module instances come into being in the standard PASCAL way. If T is a module type and V is of type T, we can write:

```
V := T(L)
```

where L is an actual parameter list. This creates a new instance of a module of type T and assigns a reference to this module to V. The next example (Figure 4) presents a fragment of the code for a fixed-length queue of **real** values. The style of coding of Figure 4 follows the Smalltalk [19] idea of having a data structure interpret messages with a command slot. The enumeration type *qactions* on line 2 defines the commands which are meaningful to qmod modules. The command *Generate* is intended to start the sequencing; the remaining qactions are obvious. The definition starting in line 4 of the prototype *qmod* has one formal parameter, *Length*, which is used in line 12 to fix the size of the array used for the queue instance. The **public** slot names, **vars**, and the **const** array are the minimal constructs needed for a simple queue module. Lines 14–18 lay out the standard PLITS style for a server module which waits for any message and then branches on the value of some action or command slot. In line 17, the PASCAL **With** construct is used to implicitly prefix all slot names within its scope with the message variable, *Mess*1. For example, line 18 is equivalent to:

```
Case Mess1·Command Of
```

Fig. 4.

```
 1  Program Everything
 2    type qactions = (Clear, Empty?, Size?, Remove, Append,
                       Generate, Next, Print)
 3    :
 4    type qmod = mod Length: integer
 5      Begin
 6        public Recipient:  module
 7               Command:  qactions
 8               Datum:    real
 9               Flag:     Boolean
10        var First, Next: integer
11            Mess1:     message
12        const Q = array [0: Length] Of real
13        First := 1; Next := 1
14        While True Do
15        Begin
16          Mess1 := Receive
17          With Mess1 Do
18            Case Command Of
19            Clear: Begin
20                     First := 1; Next := 1
21                     Send message (Flag ~ True) To Recipient
22                   End;
23            Empty?: Begin var Ans: Boolean
24                          Ans := (First = Next)
25                       Send message (Flag ~ Ans) To Recipient
26                     End
                       :
27        End of while loop
28      End of qmod definition
29      Const Fred = mod
30        Begin
31          public Recipient: module
                  :
32          var Diskq: module
                  :
33          Diskq := qmod (14)
34          Send {Recipient ~ Fred, Command ~ Append,
                  Datum ~ 3.14} To Diskq
              :
35        End of Fred
```

Starting in line 29, we have a constant module, *Fred*, which makes use of a private instance of qmod which it refers to through its local **module** variable *Diskq*. Line 33 gives the code for establishing an instance of *qmod* with the *Length* of the queue equal to 14. Line 34 shows how Fred would put an element on the queue, providing its own name as the *Recipient* for error messages.

In keeping with the general PLITS philosophy, a module cannot be killed from outside. There is a single statement:

Self destruct

which will allow a module to eliminate itself. The problem of messages to and from a destroyed module is a standard one in operating systems and in networking and is discussed in Sections 3 and 4.

There are five additional constructs which return Boolean values:

Pending {**From** P} {**About** K}
Extant P

where *P* is of **module** type and *K* is a **transaction**. The first four test if there is a message ready for the module executing them, and the fifth, if a given module instance is still active.

The final new construct to be defined is an additional simple type, **transaction**. An object of type **transaction** is specified to be unique across all "sites" in a PLITS environment. (The definition of a site and the implementation of **transaction** type objects is discussed in Section 3.) In PASCAL-PLITS, we declare **transaction** variables in the usual way:

T1, Key:**transaction**.

In addition to **transaction** variables and slots, there is one expression of type transaction, **New-transaction**; it is used in statements of the form:

Key := **New-transaction**.

As we discussed above, any **Send**, **Receive**, or **Pending** command can have an optional part

About K

where *K* is a transaction expression. Transaction objects are used in a variety of ways in the coordination of modules. For example, a *Fibonacci* module could assign separate transaction objects to each of a number of Fibonacci sequences which could be pulsed asynchronously.

Interesting design issues arise in the choice of the **receive** and **pending** constructs. One would like a module to be able to do quite selective **receive**'s and not be bothered with messages that it was not ready to process. For example, one could allow **receive** to take an arbitrary predicate on the values of slots in the message. There are several difficulties. One cannot build into the system all the generality that might ever be required—for example, a module might want to receive that message which has the greatest value for some slot. Another problem is that having very selective **receive**'s puts a great burden on the system for storing, checking, and keeping track of messages. Finally, there are problems of defining the correct sequencing for messages which are being controlled by complex predicates. The definition we have chosen is a compromise. Clearly, having **receive** only specify the source is too restrictive. Many different kinds of selective **receive**'s can be coded into **transaction** keys. The proposed way of handling more complex **receive** specifications is through the use of a "front-end" module. The sender-transaction encoding has proved to be a convenient basis for the low level communication protocols required for reliable transmission and flow control (cf. Section 3 and [39]). A PLITS system should provide primitives, data structures, etc., which make this relatively easy. It probably is worth adding negation and sets of senders and transactions to the primitive **receive** and **pending**.

2.2 Discussion

One important feature of PLITS systems is the use of messages as the basic control primitive. Although there have been many proposals for synchronization and control disciplines, all of these are easily captured by the **message** construct often leading to clearer solutions to classic control problems. We also can deal with hardware or software interrupts and with timing signals as messages. All of this is well known, and has been incorporated in several systems, but has not been enunciated in high level constructs or languages ([20, 22] are partial exceptions). The notion of module-valued slots seems to provide easily a very flexible, but safe discipline for control transfers, incorporating continuations, complaint departments, etc.

The use of messages is also valuable in solving the problems of shared resources, particularly data structures. The general idea that a resource always be allocated by a single controlling module greatly simplifies all the common exclusion problems. The currently fashionable way [28] of manipulating data in an external module (class, form, etc.) is to execute a procedure in that module. The message paradigm has several advantages over subroutine calls. If the modules were in different languages, the subroutine call mechanisms would have to be made compatible. Any sophisticated lockout procedure would require the internal coding of queues equivalent to what the message switcher provides. In the subroutine discipline, a module which tries to execute a locked subroutine is unable to proceed with other computation. The total picture on the relative value of messages and calls is much more complex; Section 4 contains some additional discussion.

Another view of PLITS messages (*A*-sets) is as a generalization of parameter lists in subroutine or coroutine calls. The idea of explicitly naming parameters is common in assembly languages where the total number of parameters to a routine may be very large. More importantly, the set of slots presents a collection of

suggested parameters rather than filling in the values of parameters. This leads naturally to the use of semantic checks on the consistency of parameters and to the use of default values for unspecified ones. This is already stronger than strong typing and can be further strengthened by using *Assertions* (cf. Section 4). Three other advantages also fall out. The use of return messages frees us from the constraint of single-valued functions—there is no reason why an answer message should have only one slot. The use of *A*-sets of name~value pairs as the input and output of all modules provides the cleanest form we have seen for the composition of multiple-valued functions. For intermodule communication, we also solve the so-called "uniform reference problem"—one need not be concerned with whether an answer (say an array element) is computed by a procedure or a table. Mesa [18] also attempts to achieve these goals by a quite different method.

There is yet another useful view of messages. One can view a message as a partially specified relation (or pattern), with some slot values filled in and some unbound. This is common in relational databases [1] and artificial intelligence languages [5]. In this view, a message is a task specification with some Recipient and some Complaint Departments to talk to about it. Various modules can attempt to satisfy or contract out parts of the task of filling in the remaining slots. One nice feature of the current design is the ability for a module to handle messages containing slots unknown to it. This allows for several modules working together on a task while maintaining locality. For example, an executive module could route messages (on the basis of a few slots that it understood) to modules which deal with totally different public slot names. We can also view a message *A*-set (set of name~value pairs) as a collection of bindings of variables. This shares many of the properties of LISP *A*-lists and SAIL contexts and seems to be an excellent way to handle the problem of evaluation relative to an environment.

There is no apparent conflict among these alternative views of PLITS messages. It is too early in the development to be sure, but the combined power of these paradigms seems to provide a qualitative improvement in our ability to develop programs.

There are other interesting features that arise when messages are combined with the idea of modules. The most obvious feature of PLITS programming is the high degree of locality and protection it provides. Each PLITS module is totally self-contained and communicates solely through messages. This means that no local variables can even be examined from the outside, no procedures invoked, etc. A module can be asked to return or update a value, execute a function, etc. It now becomes quite natural to screen requests for validity (much more than type checking), to guard against conflicting demands on a data structure, etc. This does not solve all the problems attacked by structured programming strictures, but does make it clear what has to be done and where. For example, consider the problem of maintaining consistency in a multiply-accessed database.

Fig. 5.

```
const Alphonse = mod
 1  Begin
 2    public A, B: integer
 3           Action: actions
 4           Recipient: module
 5    var In_Mess, Out_Mess: message
 6           Key: transaction
 7    Key := New_Transaction
 8    Send message (Action ~ Lock, A ~ 0, B ~ 0) To Gaston
        About Key
 9    Send message (Action ~ Fetch, A ~ 0, B ~ 0,
        Recipient ~ Me)
10      To Gaston About Key
11    Receive In_Mess from Gaston About Key
12    Out_Mess := message (A ~ − In_Mess·B,
        B ~ − In_Mess·A, Action ~ Update,
13      Recipient ~ Me)
14    Send Out_Mess To Gaston About Key
15    Receive In_Mess from Gaston About Key
16    If In_Mess·Action ≠ Reject
17      Then Send message (Action ~ Unlock, A ~ 0, B ~ 0) To
          Gaston About Key
18      Else Comment whatever;
19  End
```

First let us consider a simple exclusion problem: Suppose a module *Alphonse* wants to swap and negate two integers *A* and *B* in another module *Gaston*, which we can think of as a global data structure. *Alphonse* must get the values of *A* and *B*, swap and negate them, and put them back. The problem is to do this without unduly locking Gaston and in such a way that no inconsistencies can arise. One rather elaborate PASCAL-PLITS solution to this problem is shown in the next example (Figure 5). The module *Alphonse* first sends a message to *Gaston* to lock *A* and *B* except for messages using transaction *Key*. It then fetches the two values, swaps and negates them, sends them back and waits for a response. When *Gaston* has completed the internal update, a message is returned to *Alphonse*. If all went well, *Alphonse* will send an unlock message to *Gaston* to complete the transaction. There are simpler ways to accomplish this, but this program above makes much of the discipline explicit.

The module *Gaston*, which we are viewing as a global data structure, must have a way of locking *A* and *B* during the critical period when their values are not stable. The easiest way is to delay response to any message involving *A* or *B* which does not have the magic key. Notice here that we are assuming that the public slot names *A* and *B* each correspond to a single "global variable" which we wish to regulate. This is a specialized use of public slot names, but an important one and one which naturally gives rise to exclusion problems.

As before, the key for this transaction can be passed from module to module. The pair of **send**'s on lines 8 and 9 could be made into a single fetch and lock statement, either by adding such a primitive action or adding

a **postaction** slot to the message. Similarly, one could simplify lines 14–17 by using an update and unlock construct. There is no difficulty making these constructs indivisible, because a module (e.g. *Gaston*) is never interrupted. There can be no deadlock (in this simple case) because *Alphonse* locks all the resources that it needs (*A* and *B*) before starting. The module *Gaston* can receive and process other messages while waiting for *Alphonse*—the entire module is not locked.

There is a potential problem of code in *Gaston* using the local variables associated with *A* and *B* but not mentioning them in messages. If one really wants a variable to be totally protected in PLITS, one must make it a module. A more plausible solution is that any module which allows locking of particular variables must maintain internal consistency by not modifying these variables itself while they are locked. The potential deadlock situations are internal to the module and should be easily avoidable.

More commonly, one is concerned with the exclusion problem in large complex data structures. One very clean analysis of this problem can be found in [9]. They develop a notion of a sufficient discipline for locking and extend it to "predicate" locks which are logical conditions set up by one module to guarantee the integrity of its modifications. The PLITS implementation of this algorithm is a straightforward extension of the ideas developed above [32].

The coding of the structure-monitoring module *Gaston* could be done in a number of different ways. We have suggested that *Gaston* could simply ignore messages that involve *A* or *B*. This could be done using a *Smart_Receive* function call of the form

Mess3 := Smart_Receive (Transaction = Key $\vee$
(**Absent**(A) $\wedge$ **Absent**(B)))

We have already discussed some of the alternative ways of coding a *Smart_Receive*, for example, using a filtering front end module. It is almost equivalent to have *Gaston* accept all messages and only process the appropriate ones, but there are advantages and disadvantages to each method. If *Gaston* deals with all messages, it must have an internal data structure which duplicates much of the system function of queuing messages. On the other hand, by looking at all messages, *Gaston* has the opportunity to detect high priority messages, time-critical situations, contradictions, etc. This discussion suggests the kind of issue that arises in the design of a system that is based on PLITS. We are attempting to provide a set of primitives that will support a variety of solutions to problems like the monitoring of global variables.

3. Implementation Considerations

The implementation of PLITS ideas has proceeded in parallel with the formulation of the general concepts described above. The development of an operating system for RIG (Rochester's Intelligent Gateway) [2] used the message-module paradigm, but no higher level language forms. A serviceable SAIL-PLITS with essentially the features described in Section 2 was written by Jim Low in the summer of 1976 and has been used for experimental and student work. One outgrowth of this effort is an "advanced compiler" project, which is discussed briefly in Section 4. The other major current effort is the implementation of a uniform framework for multimachine, multilanguage distributed user jobs ("DJOBs") in the PLITS style. This is being carried out in the context of our local network which currently contains four Altos, two Eclipses, and a DEC KL10 and is described in [12]. The discussion here is intended to point out the issues arising in the implementation of PLITS or any similar high level language for distributed computing. Once arrangements for starting a DJOB are made, the underlying system should be invisible to a PLITS user who will program as described in Section 2.

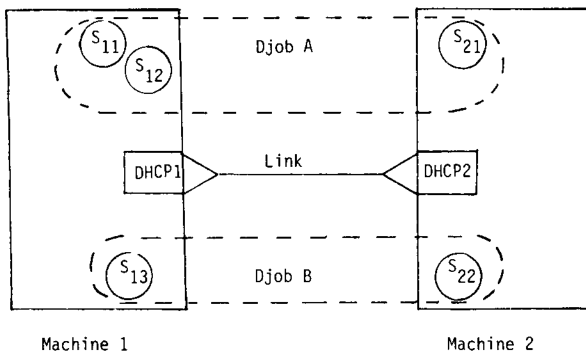
We first look more carefully at the process of sending and receiving messages. Even on a single machine, there will have to be some underlying programs which handle messages and schedule modules for execution. We will call such a collection of programs a *Kernel.* A Kernel is a conventional multiprogramming monitor which sequences through the modules on its "ready" queue. A Kernel also maintains data structures describing modules which are "suspended" waiting to **Receive** a message of a specified sort. These data structures, together with analogous ones for messages which result from **Send** statements, suffice to implement the PLITS message primitives.

With an underlying operating system like TOPS/10 or TENEX, it may be convenient to group modules into a single job if they communicate mainly with one another, or comprise a package, or share code. Such a group is called a *Site.* In general, each site will have its own kernel. A problem arises if the modules of a DJOB are written in different body languages. It may be the case that languages differ in their representation of primitive data types (e.g. **real**). We require that the representation of primitive data types be uniform within a site. This, as well as other considerations, may give rise to the situation where there is more than one site on a given machine involved in an individual distributed job.

A DJOB might consist of modules on several computers. For example, a distributed vision application might consist of an image processing site on the PDP-10, an interactive site on an Alto, a site on an Eclipse for managing a color display, and file servers on both the PDP-10 and on the Eclipse. One of the modules in each DJOB is designated the "controlling module" for the DJOB. In this example, the controlling module might be one on the Alto. The controlling module for a DJOB is responsible for initializing and terminating the DJOB and for taking appropriate action when one of the other modules of the DJOB fails.

Figure 6 is a graphic representation of the breakdown of functions and terminology which we have adopted. It is convenient to divide the PLITS support functions into

Fig. 6.



two subsets carried out by the site Kernel and by the *Host Control Program* (HCP) respectively. In Figure 6, there are two DJOBS, *A* and *B*, which have no connection but happen to be both distributed over Machines 1 and 2. DJOB A consists of three sites: *S*11 and *S*12 on Machine 1 and *S*21 on Machine 2. Each site has a Kernel assigned to it as described above. The Kernel performs the following functions:

(1) distributes messages to and from modules within the site;
(2) forwards messages to and from other sites;
(3) carries out needed representation shifts for inter-site messages;
(4) allocates resources within the site;
(5) generates unique (world-wide) module and transaction identifiers;
(6) checks for errors and assertion violations.

We have discussed the first three functions briefly and will give more details on them below. The fourth function, resource allocation within the site, is concerned with storage allocation and reclamation, scheduling of ready modules, etc. The fifth function is discussed later in this section. Error and assertion checking are discussed in Section 4.

Each HCP is an extension of its machine's operating system. It performs four main functions:

(1) distributes messages among sites local to this machine;
(2) forwards messages to and from other machines;
(3) starts and stops DJOBS, and provides access to other operating system services;
(4) checks for intersite errors and assertion violations.

When a PLITS message is sent by a module, its destination is checked. If it is within the sender's site, the site Kernel handles it; if not, it is given to the local HCP. If the destination is within another site on the same machine, it is given to the Kernel for that site; if not, the HCP has it forwarded to the appropriate machine—the job of HCP functions 1 and 2 above. To do this effectively requires quite a lot of mechanism beneath the surface. Problems faced include reliable transmission, flow control, error handling, and providing user services for distributed computations. This has led us to view the underlying support facilities as a distributed operating system (DSYS).

Each machine HCP has two parts: a "DSYS Job Manager" (for distributed jobs) and a "DSYS Communications Manager." This organization reflects the two separate facilities of DSYS: operating system support and services for PLITS DJOBS, and basic message communication in the PLITS style.

Each Job Manager:

(1) provides services for DJOBS (i.e. start, stop, access to services of the local operating system);
(2) remembers which local services are allocated to which DJOBs, and which module is the controlling module for each DJOB;
(3) arranges to recover resources used by such services when a DJOB finishes.

In addition, each Job Manager keeps track (for each DJOB whose controlling module is local) of the other computers that are involved in the DJOB. The Job Manager for the controlling module of a DJOB knows which other HCP's to notify when the DJOB finishes (or dies).

Consider the problem of setting up a DJOB. If there are two sites on the same machine with the same primitive data representations, the HCP must check that the use of public slot names is compatible—essentially the same process as binding the externals of two load modules. If two sites have incompatibility in representation of a primitive data type, then some conversion routines will be automatically invoked for inter-site messages when they are sent. The ARPA network voice protocol [7] presents a good model of a scheme in which a dialogue between machines is used to reconcile representation differences before messages containing data are sent. All of this is fairly messy, but should only be necessary when a new PLITS language processor is brought up on a machine. In the usual case, the standard conversions between sites will have been established and the negotiations between machines will be simple.

The DSYS Communications Manager (DCM) on each computer is responsible for forwarding messages to and from modules on other local sites and on other computers. The DCM accepts messages to be forwarded to remote modules from local ones, and passes messages to local modules that arrive from remote ones. In addition to dealing with communications I/O devices, the DCM controls the flow of messages from local senders based on the rate of acceptance by intended receivers and the availability of buffer space. The DCM also provides a "reliable transmission" service.

The DCM allocates buffer space for messages on a "destination" basis. Each (receiving module, transaction) pair is considered a "destination" for messages. A descriptor which includes a "destination queue" exists for each destination. Each such queue has its own allotment

of buffer space for messages. This space is not committed a priori, but is rather a (changeable) estimate of how much of a backlog of messages should be allowed for the destination. The basic flow control mechanism is simple: a sending module is kept suspended until space on the destination queue becomes available. If the destination is in the same site as the sender, the site Kernel controls message flow. If the destination is at a different site, the message is passed to the DCM, which arranges to forward it.

A destination descriptor is a distributed data structure. The destination's site Kernel has a portion, and each computer upon which there is at least one module sending messages to the destination has a portion (maintained by the sender's DCM). One can view the portions on remote computers as queue extensions. The primary job of each DCM is to maintain its part of this distributed data structure to support module-to-module communication across computer boundaries. The system is designed in such a way that a DCM can "forget" about its part of a remote destination descriptor if there is no message activity for a while. A scheme for "implicit connections" is the basis for this design: local knowledge about remote destinations is acquired when needed, automatically. There is purposely no requirement that state information about a remote module be maintained arbitrarily long.

There is a question of how to identify modules in a distributed system. If there were a central source of identifiers, it might take a long time to get one and the central source might be sometimes inaccessible. If each module created its own, there would either have to be a lot of handshaking or there would be a danger of duplications. Our solution is simple and quite general: an identifier (in the present design) is a 32-bit number composed of four fields: a computer number, an "incarnation number," a site number, and a "local module number." Such an identifier is a network "address" [33]. The computer number uniquely identifies one of the computers in our network. The incarnation number is used to distinguish old incarnations of the operating system on the indicated computer from the most recent one. DSYS uses this information to trap references to defunct operating system incarnations. The site number identifies a site on the indicated computer, and the local module number identifies a module at the site. Thus a module address uniquely identifies a module in the distributed system.

One consequence of this definition is that a given module always resides on the same machine, somewhat contrary to current fantasies about distributed computing. In our view, a module will be compiled to take full advantage of the hardware and software resources of its machine. There may be *equivalent* modules on various machines, and programs will be able to choose between them, but each will have a distinct unique address, hence machine of residence.

There is one additional question that should be addressed at this time—sending structured data objects in messages. As defined, PLITS allows only single elements to be the value of a message slot. This is defensible in a one-site system where one can assume that access to arrays, for example, can be by message, with an advanced compiler (Section 4) making the simple cases efficient. This model simply breaks down in the case of remote sites and we are forced to consider sending blocks of information in messages. This also means that a PLITS user cannot, in practice, totally ignore the location of his modules, but one would hardly expect it to be otherwise.

There are several possible ways to add structured types to PLITS messages. The most general would be to allow for constructed modes (as in Algol 68) and the use of these modes as data types for public slot names. In this case, the initial connection dialogues would also have to come to agreement about all the publicly defined modes, but there does not seem to be any inherent difficulty about this (we already require checking enumeration types). One would not, of course, allow the use of references or pointers in the defined modes. General structured types are omitted from this version of PLITS for simplicity and because our ideas on a universal structure mechanism are just beginning to take shape [16].

We are using a somewhat simpler extension—a single additional public type: **bundle**. A **bundle** object is a self-describing collection of objects of primitive type. We will first present the syntax of **bundle** and then discuss its use.

⟨group⟩ ::= ⟨repetition⟩ ⟨elementary type⟩ ⟨repetition values of type⟩
⟨bundle⟩ ::= ⟨total length⟩ ⟨group⟩ | ⟨bundle⟩ ⟨group⟩

Thus a bundle is a collection of one level structures. For example, a bundle of two complex numbers followed by an **action** update would be of the form given in Figure 7 (where **action** is a public enumeration type). The idea is that bundles are relatively simple and are handled by relatively few modules at each site. If two sites wanted explicitly to include the information that a pair of reals was a complex number rather than a 2-vector, they would use another enumeration type to provide descriptors in bundles. Similarly, one could have bundle descriptors either in the bundle or in accompanying slots. Our current interest in bundles is primarily for passing very

Fig. 7.

12
2
real
Re1
Im1
2
real
Re2
IM2
1
action
Update

large collections of data such as images [40] and, to a lesser extent, for buffer management within a site. In connection with this, one probably wants to add a **Sendoff** construct which asserts that the sending module has no further need for the data in this message. It is too early to tell how much elaboration of the bundle mechanism will prove worthwhile.

4. Related Issues

The overall aims of the PLITS project go well beyond the distributed computing proposals presented above. The project originated as an attempt to look very carefully at programming languages and their use in a very broad context. There is a great deal that we do not yet understand, but a surprising number of questions do seem to yield to mutually compatible solutions, such as the module-message paradigm. The first technical report [10] on the PLITS project contains a loose overview of our ideas on a variety of topics. Some of these are being treated in detail in current reports [11, 16, 39], and others will follow. We include here just enough discussion of these issues to show how the material of the first three sections fits into the overall project.

A major focus of effort in PLITS is the use of declarative information in programming languages. Our concerns cover a broad range of issues, ranging from a careful study of type mechanisms [16] to general nonprocedural programming. We are attempting to develop a uniform solution, involving a general notion of *assertions* and very sophisticated compilers that will encompass conventional optimizations, language extension, verification and automatic programming within a single framework. We will outline the issues most closely related to distributed computing, starting with the issue of primitive data types.

In current programming languages, data types and the associated type machinery are used for a variety of purposes. In some cases alternatives to ordinary data types may serve better, providing language facilities which are more expressive and more extensible. At the simplest level, data types are used to indicate a particular hardware representation for variables. The compiler must have this information in order to generate correct code. **Real** and **integer** are common examples, directly related to the difference in code generated for real and integer variables. In a tagged architecture where the reals and integers were not distinguished, there might be a single type **number**, which combined both.

Given that the number of types which have a distinct, direct hardware representation is small and fixed, it seems appropriate to regard them as primitive. The present data type machinery is perfectly adequate for this purpose, if we accept that for a given site, each simple variable has a single particular hardware representation which we wish to specify. The compiler will ensure that we do not mix different representations in a meaningless way. It is central to the present conception of data type that the compiler checks type correctness.

This has led to the unfortunate notion that if the compiler is to check something, then it must be a data type. Data types are now being used to encode all those properties of a variable which the compiler checks. Strong typing (e.g. PASCAL) attempts to provide for assertional information and consistency checking, but uses much too weak an expressive mechanism. A strong type is essentially the logical AND of a set of properties, with the OR of these conjunctions expressible as a union type. This does not allow the programmer a convenient way of asserting the desired consistency checks. The use of union types also gives rise to some subtle aliasing effects [38].

As a simple case of the difficulty, suppose that a programmer would like to have three independent properties, such as

small(x) ≡ |x| < 15; **odd**(x) ≡ mod 2 = 1; **positive**(x) ≡ x > 0

In a strongly typed language, he can do this only by defining composite properties like **small odd** or **odd positive.** Unless a coercion is defined, one cannot, for example, use a **small odd** variable in a procedural call which requires a **small** argument. Composite types can be combined using the **union** construct, which has the effect of **or**ing properties together.

Even with three basic properties over a hundred different plausible types can be produced. For example:

union (**small, odd**)
union (**small positive, odd**)
union (**small positive, small odd**)

where presumably any property may also be negated.

This creates even greater problems with generic operators, which are operators associated with different function procedures for arguments of different type. For example:

small + **small**
odd + **odd**
positive + **positive**

might each invoke a different procedure. This might be adequate if three procedures were all that are needed, but what about:

small + **small odd**
small + **odd**

Perhaps the first can use the same procedure as

small + **small**

by invoking the coercion of **small odd** to **small**, but what about the second? The variable of type **small** might be odd, so that the **odd** + **odd** would be correct, or perhaps the variable of type **odd** is small, or perhaps they are both positive! A profusion of composite and union types could not possibly help; it would multiply the number of cases to be considered alarmingly.

The problem rests on two fundamental assumptions of the type mechanism: that a variable has just one type;

and that types match only if they are identical or can be coerced to be so. But neither assumption is necessary to do the job that property types are here being required to do. If a programmer wants to declare that a variable always satisfies some predicate P, or that the actual argument to a procedure must satisfy another, then it should be possible to say just that.

The solution to this and a number of related problems is to add a simple *property* mechanism to the primitive type facilities of a language. For problems involving only simple variables such as that described above, the use of separate properties for **small**, **odd**, and **positive** make it quite easy to specify the "type" requirements for variables and functions. For composite data structures, such as arrays or records, the property mechanism becomes more complex. Our current ideas on this are given in [16].

Even with the property mechanism, we are restricted to specifying only conditions involving one variable or argument. There is no way, for example, to specify that the arguments to a procedure sum to one (or approximately one). For this and a number of other reasons we propose to include a more general *assertion* facility in PLITS.

An assertion is a predicate which the compiler will guarantee to be true at run time. It will either prove it true (at compile time), or generate code to check the assertion. Assertions may be used to describe important properties of variables and data structures. A property is a special kind of assertion that applies to only one variable and holds throughout the lifetime of that variable. They help the programmer to write provably correct code, and can be used for error checking. Assertions may be used by the compiler to generate more efficient code. Since they have to be proved or checked, they function as "hints" as to what program properties the optimizer might use. There is no attempt, however, to coerce the programmer into providing enough assertions to allow formal verification of all programs.

We will present assertions and their uses more concretely, continuing the use of PASCAL-PLITS as a basis. Certainly the notion of ⟨assertion⟩ will include the existing ⟨Boolean⟩ expressions, but may be more extensive. As a first example, consider the reception of messages by a module. One would like to be able to **Assert** that a given **Receive** will acquire only certain kinds of messages. With our definitions, the obvious thing is to accompany the **Receive** statement with

Assert Slots Must Be ⟨set of **Public** slot names⟩

This assertion allows the compiler to assume that no other public slot names known to this module will appear in any message picked up at this point. One could also assert, e.g. that the *action* slot of incoming messages at this point could not have the value *Update*; the compiler should then be able to generate much better code for this read-only access. A well-written PLITS program will have every **Receive** statement accompanied by one or more assertions. Since modules are otherwise totally self-contained, the compiler can then analyze the receiving module for consistency, code-optimization, etc. The execution-time truth of an assertion might be provable at compile time, but otherwise it will require a compiled-in check. The idea here is that we can develop proofs of conditional correctness of individual modules given their assertions with errors being detected as run-time violation of specific assertions. The problem of verifying the correctness of a collection of modules is more complex and is addressed in [11]. The method proposed there is to characterize each module as a finite-state machine and to establish the properties of the system by reachability theorems in the vector space of states of the modules. The propagation of information among modules which communicate only by messages is addressed in [37].

The specifications of a module in PLITS should include the assertions on its incoming and outgoing messages. Assertions will enable us to capture easily constructs like the pre, post, invariants, and requires of Alphard [29]. For automatic programming, the specification of a module must also include other information, like the resource utilization [31] of the module.

More formally, there are two ⟨statement⟩ constructs involving assertions proposed for PASCAL-PLITS.

(a) **Assert** ⟨assertion⟩
(b) **Under** ⟨assertion⟩ **Do** ⟨statement⟩

The first of these is the basic form which generates an exception condition if the assertion fails to hold. Construct (b) specifies that the ⟨assertion⟩ is to hold throughout the ⟨statement⟩. This idea has also been called "invariants" or "continuously evaluating expressions" and is extremely powerful. Unfortunately, it is not easy to specify or to implement in a general and useful way. There are two difficulties: efficiency and the grain of evaluation. The efficiency question is easy to understand—how can we implement invariants without unduly slowing down the computation. It is our claim that standard flow analysis and value propagation techniques can make this feasible in the usual cases. This is a good example of what we would like from an advanced compiler.

The grain problem for invariants is much deeper. Should the semantics of **Under** specify that the ⟨assertion⟩ must hold at every individual machine cycle or at some coarser grain? Our current definition is that the ⟨assertion⟩ must hold at the end of each first level substatement of the ⟨statement⟩ which is the body of the **Under**.

In addition to their use in verification, assertions play a central role in our work on code optimization. One of the key problems in bringing PLITS-style programming into widespread use is the development of techniques for producing efficient code from the decoupled and protected constructs of PLITS. The idea here is to have the compiler understand those cases where certain checks are not needed at execution time.

One can get a feeling for this problem by considering

the implementation of standard variables and arrays in PLITS. It is certainly true that we could have **array** (even individual variable: cf. [20]) modules which took messages and returned values. Doing this in the obvious way would cause an unacceptable slowdown of about a hundred in the execution of simple programs. One solution to this "grain" problem is to make modules be rather large subsystems, coded in the usual way. The better solution is to use a more sophisticated compiler. In general, there are times when one would want the full PLITS paraphernalia for accessing a global array. For example, one might want to have critical sections or check the range of values or trace the updates or change the internal representation of the array or lots of other things. What we would like is to be able to **close** [45] a collection of PLITS modules and get very good code for the simple cases.

This module integration problem is the direct extension of the standard procedure integration task which is an important aspect of current optimization efforts. One encouraging initial result is that the narrow message-based interface among modules makes global flow analysis much less costly than in other proposed schemes for parallelism [37]. Similar module integration problems would arise in any of the proposed data abstraction languages like Alphard [29] or CLU [27] or EUCLID [24].

Debugging calls for some new techniques in a PLITS (or any Distributed Computing) environment. A direct ancestor of PLITS was the Stanford Hand-Eye System [15] in which some of these problems were addressed. The major additional debugging tools there were time-labeled selective message tracing and the ability to interact separately with individual modules. A major difficulty was that the user console handled everything sequentially, merging all communication streams. The use of multiple streams was developed by Swinehart [43] and is a central feature of the Rochester RIG system [2] which has a PLITS-like message basis.

Another important set of issues arises from the fact that message switching systems manifest errors in characteristic ways. Since modules are self-contained, careful coding can guarantee that a module never process a message that would force it into a bad state. Messages that violate assertions can be found and reported. The difficult error conditions come from situations like deadlock, flooding, starvation, etc.

A *deadlock* in PLITS arises when two (or more) modules are in a situation where each is attempting to receive a message which must be sent by another. *Flooding* occurs when a module generates too many messages and *starvation* where a module does not receive messages intended for it. There are a variety of other error conditions that can arise in a multiprocessing environment involving critical race conditions, inconsistent shared data, etc., but these will not normally be detectable as problems in queue management. The hope is that the message discipline and the use of assertions will make it easier and more natural to write correct programs and find errors.

We have developed two kinds of solutions to error conditions arising in queue management. The first kind of solution involves having the PLITS kernel use more sophistication in its management of message queues in order to minimize the number of avoidable deadlocks, etc. These techniques were described briefly in Section 3 and are treated more thoroughly in [39].

The second class of solution to error conditions involves the use of system-generated exception conditions. The system (DSYS) underlying a PLITS implementation will have a great deal of information about the message state of the various modules. It is straightforward to have the system detect a variety of illegal and dangerous situations. It is also possible (cf. [10]) to provide for the checking of user-provided assertions on message behavior. Errors or assertion violations can be treated as types of exception conditions.

In PLITS, exception conditions can be clearly divided into two classes: those that arise within a module and those that are external to the module. Since much PLITS programming is event driven, exception conditions will cover much more than the usual error conditions. Typical internal conditions include overflow, type violations, message arrival, and absent slots. External exception conditions include invalid messages, time-out notifications and the availability or demise of other modules. In the PLITS environment, the notification for an external exception condition will be through messages. Although it is not strictly necessary, it is preferable to have the notion of *priority* message for exception conditions (among other things). In our proposal, a priority message will be received (if present) immediately before the next **Send** or **Receive** statement executed by the module. This is equivalent to having a statement of the form:

```
1  While Pending About ⟨priority transaction⟩ Do
2    Begin Message M1;
3          Receive M1 About ⟨priority transaction⟩
4          Cause (priority_message, M1)
5    End
```

before each **Send** and **Receive** statement. The **Cause** statement in line 4 triggers an internal exception condition within the module. Thus, an external exception condition is announced as a priority message to the module; this could be explicitly checked for at any time, but will normally be converted to an internal exception condition of type *priority_message.*

The treatment of internal exception conditions will, of course, be different in different body languages. The following proposal for PASCAL-PLITS is a simplification of ideas of [26] made possible by the elimination of data sharing across modules. A PASCAL-PLITS program has a fixed set of named exception conditions and a (generally larger) set of procedures, called handlers. Following Levin, we assume that the handler to be invoked for a given condition can be *declared* as part of the declaration of a block by a statement of the form:

On ⟨condition⟩ **Invoke** ⟨handler⟩

The record of which handler is currently appropriate for each condition follows the PASCAL dynamic nesting structure. When invoked, a handler runs as a normal procedure called in the block where its condition was caused. This is all straightforward—the difficult problems are how to pass information to the handler and how the handler should complete. We expect the following simple solutions to suffice for PASCAL-PLITS.

There are two basic completion paths commonly needed for exception handlers—either the handler returns to the point of invocation or it must exit to some higher level. It might also need to pass on the situation to another handler. The following four statements:

Return {**Causing** (⟨condition⟩, ⟨arg⟩)}
Exit {**Causing** (⟨condition⟩, ⟨arg⟩)}

cover the possibilities. A handler might call other routines, change global data within its module, send messages, etc. while operating. When it is finished, it must either Return to the point of invocation or Exit from the block in which it was **declared**. Optionally, it can also cause another condition after it completes.

A thorough discussion of exception handling and the advantages of this design are beyond the scope of this paper. The astute reader will have observed that the ⟨arg⟩ accompanying a Cause was a ⟨message⟩ (*A*-set) in the previous example. The use of sets of name ~ value pairs as arguments to condition handlers seems particularly appropriate, because various handlers might deal with different slots. As was mentioned in Section 2, we are currently exploring the use of *A*-sets in a number of programming contexts.

Acknowledgments. The ideas presented here owe their present form to continual discussions and interactions. I would especially like to acknowledge the help of J. Low and P. Rovner, the students of CSC 400 and 520, and the former friends and captive seminar audiences who had to listen to various partial formulations.

We have had enough experience with PLITS to understand that it behaves as a mirror in which each observer sees a reflection of something different—this indicates the presence of many unacknowledged intellectual debts. The most direct ancestor of PLITS is the message procedure facility [15] which was put into SAIL for robotics work and which no one remembers.

Received March 1977; revised December 1978

References

(Note. References [4, 6, 8, 13, 14, 17, 21, 23, 25, 30, 34–36, 41, 42, and 44] are not cited in the text.)

1. Astrahan, M.M., et al. System R.: A relational approach to data base management. IBM Res. Lab., Feb. 1976.

2. Ball, E., Feldman, J., Low, J., Rashid, R., and Rovner, P. RIG, Rochester's intelligent gateway: System overview. *IEEE Trans. Software Eng. SE-2*, 4 (Dec. 1976), 321–328.

3. Ball, J., Williams, G., and Low, J. Preliminary ZENO language description. TR41, Comptr. Sci. Dept., U. of Rochester, Rochester, N.Y., Dec. 1978.

4. Birtwistle, G., et al. *Simula Begin.* Auerbach, Philadelphia, Pa., 1973.

5. Bobrow, D.G., and Raphael, B. New programming languages for artificial intelligence. *Computing Surv. 6*, 3 (Sept. 1974), 155–174.

6. Bolt, Beranek and Newman, Inc. Interface message processor: Specifications for the interconnection of a host and an IMP. Rep. 1822, BBN, Cambridge, Mass., 1978 (revised).

7. Cohen, D. Specifications for the network voice protocol. ISI/RR-75-39, Inform. Sci. Inst., U. of S. Calif., Los Angeles, March 1976.

8. Demers, A., Donahue, J., and Skinner, A. Data types as values: Polymorphism, type checking, encapsulation. Conf. Record Fifth Annual ACM Symp. on Principles of Programming Languages, 1978, pp. 23–30.

9 Eswaran, K.P., et al. The notions of consistency and predicate locks in a database system. *Comm. ACM 19*, 11 (Nov. 1976), 624–633.

10. Feldman, J.A. A programming methodology for distributed computing (among other things). TR9, Comptr. Sci. Dept., U. of Rochester, Rochester, N.Y., Jan. 1977.

11. Feldman, J.A. Synchronizing distant cooperating processes. TR26, Comptr. Sci. Dept., U. of Rochester, Rochester, N.Y., 1977.

12. Feldman, J.A., Low, J.R., and Rovner, P.D. Programming distributed systems. Proc. 1978 Annual ACM Conf., Washington, D.C., Dec. 1978, Vol. 1, pp. 310–316.

13. Feldman, J.A., Low, J.R., Swinehart, D., and Taylor, R. Recent developments in SAIL, an Algol-based language for artificial intelligence. Proc. AFIPS 1972 FJCC, AFIPS Press, Montvale, N.J., pp. 1193–1202.

14. Feldman, J.A., and Rovner, P.D. An Algol-based associative language. *Comm. ACM 12*, 8 (Aug. 1969), 439–449.

15. Feldman, J.A., and Sproull, R.F. System support for the Stanford hand-eye system. Proc. Second Int. Joint Conf. on Artif. Intell., London, Sept. 1971.

16. Feldman, J.A., and Williams, G.J. Some comments on data types. TR28, Comptr. Sci. Dept., University of Rochester, Rochester, N.Y., 1977.

17. Floyd, R.W. Assigning meanings to programs. Proc. Symp. Appl. Math., Vol. 19, 1967, pp. 19–32.

18. Geschke, C.M., and Mitchell, J.G. On the problem of uniform references to data structures. *IEEE Trans. on Software Eng. SE-3* (June 1975), 207–219.

19. Goldberg, A., and Kay, A., Eds. SMALLTALK-72 Instruction Manual. SSL 76-6, Xerox PARC, Palo Alto, Calif., 1976.

20. Hewitt, C.E., and Smith, B. Towards a programming apprentice. *IEEE Trans. Software Eng. SE-1*, 1 (March 1975), 26–45.

21. Hoare, C.A.R. Communicating sequential processes. *Comm. ACM 21*, 8 (Aug. 1978), 666–677.

22. Hoare, C.A.R., and Wirth, N. An axiomatic definition of the programming language Pascal. *Acta Informatica 2* (1973), 335–355.

23. Jones, A.K., and Liskov, B.H. A language extension for controlling access to shared data. *IEEE Trans. Software Eng. SE-2*, 4 (December 1976).

24. Lampson, B.W., et al. Report on the programming language Euclid. SIGPLAN Notices (ACM) *12*, 2 (Feb. 1977).

25. Lampson, B.W., and Sturgis, H. Crash recovery in a distributed data storage system. Unpublished paper, Xerox PARC, submitted to *Comm. ACM.*

26. Levin, R. Program structures for exceptional condition handling. Ph.D. Th., Carnegie-Mellon U., Pittsburgh, Pa., 1977.

27. Liskov, B., et al. Abstraction mechanisms in CLU. *Comm. ACM 20*, 8 (Aug. 1977), 564–576.

28. Liskov, B., and Zilles, S. Programming with abstract data types. Proc. Symp. Very High Level Languages, SIGPLAN Notices (ACM) *9*, 4 (April 1974), 50–59.

29. London, R.L., Shaw, M., and Wulf, W.A. An introduction to the construction and verification of Alphard programs. *IEEE Trans. Software Eng. SE-2*, 4 (Dec. 1976), 253–264.

30. Low, J.R. *Automatic Coding: Choice of Data Structures.* Birkhauser-Verlag, Basel und Stuttgart, 1976.

31. Low, J.R., and Rovner, P.D. Techniques for the automatic selection of data structures. Conf. Rec. Third ACM Symp. on Principles of Programming Languages, Atlanta, Ga., Jan. 1976, 58–67.

32. Mitchell, J.A., and Wegbreit, B. Schemes: A high level data structuring concept. CSL-77-1, Xerox PARC, Palo Alto, Calif., Jan. 1977.

33. McQuillan, J.M., and Walden, D.C. The ARPA network design decisions. *Computer Networks 1*, 5 (Aug. 1977), 243–290.
34. Owicki, S.S., and Gries, D. Verifying properties of parallel programs: An axiomatic approach. *Comm. ACM 19*, 5 (May 1976), 279–285.
35. Parnas, D.L., Shore, J.E., and Weiss, D.M. Abstract types defined as classes of variables. SIGPLAN Notices (ACM) *8*, 2 (1976), 149–154 (Vol. II, 1976 Special Issue).
36. Peterson, G.L., and Fischer, M.J. Economical solutions for the critical section problem in a distributed system. Proc. Ninth Annual ACM Symp. on Theory of Comptng., Boulder, Colo., May 1977, pp. 91–97.
37. Reif, J. Data flow analysis for communicating Modules. TR30, Comptr. Sci. Dept., U. of Rochester, Rochester, N.Y., 1978; also available as Data flow analysis of communicating processes, Conf. Rec. Sixth Annual ACM Symp. on Principles of Programming Languages, Jan. 1979, pp. 257–268.
38. Reynolds, J. On the syntactic control of interference. Conf. Rec. Fifth Annual ACM Symp. on Principles of Programming Languages, Jan. 1978, pp. 39–46.
39. Rovner, P.D. Message flow control in a local network. Comptr. Sci. Dept., U. of Rochester, Rochester, N.Y., 1978.
40. Selfridge, P.G. A flexible data structure for accessory image information. TR45, Comptr. Sci. Dept., U. of Rochester, Rochester, N.Y., Nov. 1978.
41. Sproull, R.F., and Thomas, E.L. A network graphics protocol. SIGGRAPH *8*, 3 (Aug. 1974).
42. Sturgis, H.E. A postmortem for a time-sharing system. Ph.D. Diss., U. of California, Berkeley, 1974.
43. Swinehart, D.C. Copilot: A multiple process approach to interactive programming systems. Ph.D. Diss., Stanford U., Stanford, Calif., 1974.
44. Thomas, R.H. A solution to the update problem for multiple copy data bases which uses distributed control. Rep. No. 3340, Bolt, Beranek and Newman, Cambridge, Mass., July 1976.
45. Wegbreit, B., and Spitzen, J.M. Proving properties of complex data structures. *J. ACM 23*, 2 (April 1976), 389–396.

Corrigendum. Programming Techniques

Robert Sedgewick, "Implementing Quicksort Programs," *Comm. ACM 21*, 10 (October 1978), 847–857.

On page 851, first column, line 30 and Program 2, line 8, change $l + 1$ to l.

On page 852, first column, lines 7-8, change **while** $A[j] < v$ **and** $j \leq N$ to **while** $j \leq N$ **and** $A[j] < v$. (Here **and** is a so-called conditional which does not evaluate the second argument if the first is false.)

The first of these errors was pointed out by Nelson H.F. Beebe. The published version has a running time proportional to N^2 for a file in reverse order. This problem has appeared in other Quicksort implementations (see Knuth, *Sorting and Searching*, p. 614).

The second error was pointed out by Burton L. Leathers, who suggests that the conditional **and** could be avoided by searching the rightmost subfile for the largest element in the array, and using it in $A[N]$ as a sentinel rather than ∞ in $A[N + 1]$.

Artificial Intelligence/ Language Processing
C. Montgomery Editor

The Cyclic Order Property of Vertices as an Aid in Scene Analysis

R. Shapira and H. Freeman
Rensselaer Polytechnic Institute
Troy, New York

A cyclic-order property is defined for bodies bounded by smooth-curved faces. The property is shown to be useful for analyzing pictures of such bodies, particularly when the line data extracted from the pictures are imperfect. This property augments previously known grammatical rules that determine the existence of three-dimensional bodies corresponding to given two-dimensional line-structure data.

Key Words and Phrases: Scene analysis, cyclic order, artificial intelligence, three-dimensional reconstruction, picture processing, computer graphics, pattern recognition

CR Categories: 3.2, 3.6, 8.2

Introduction

When we look at a good-quality picture of a scene containing a number of three-dimensional objects, we are usually able to "understand" the scene; that is, we are able to perceive the real physical nature of the objects. The reason for this is that we have seen similar scenes before and at those times were able, by a combination of touching and viewing, to develop the ability of relating pictures of three-dimensional objects to the objects themselves. The same applies—perhaps with the need for slightly more specialized learning—to the understanding of scenes depicted in terms of line drawings,

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

The work described here was supported by the Directorate of Mathematical and Information Sciences, Air Force Office of Scientific Research, AFSC, under grant AFOSR 76-2937.

Authors' addresses: R. Shapira, 23 Sweden Street, Dania, Haifa, Israel; H. Freeman, Rensselaer Polytechnic Institute, Troy, NY 12181.

© 1979 ACM 0001-0782/79/0600-0368 $00.75