TOWARDS THE CONSTRUCTION OF VERIFIABLE SOFTWARE SYSTEMS

L. Flon and A.N. Habermann

Carnegie-Mellon University

Abstract: Data types are an important design tool because they allow freedom of abstraction. Thus, they are useful for constructing large software systems, including operating systems. It is shown that when dealing with problems of concurrency, the use of path expressions, which are associated with data, makes the task of verification simpler than when the synchronization conditions are associated with programs.

Key Words and Phrases: verification, data types, path expressions

CR Categories: 4.0, 4.20, 4.22, 5.24

1. Introduction

Procedures and macros were originally invented to save the programmer the trouble of rewriting the same piece of code in several places in his program. Later on, programmers discovered that macros, and in particular procedures or functions, are an important abstraction tool. To the caller of a procedure, all that matters is the type of its parameters and its effect. It is not necessary to know the details of how it is implemented. Following Parnas' idea of information hiding [Par72] we adopt the rule that the call-site does not use any knowledge it might have about the implementation. In this way, a modification in the implementation does not affect the call-site, provided that the specification (the required parameters and the effect of the procedure call) is not changed.

Abstraction from implementation of executable code is achieved by the procedure concept. The equivalent abstraction tool for data objects is the abstract data type definition. It allows the programmer to distinguish between specification and implementation of a data object. A type definition describes to its users the possible states of an object of that type and the ways in which it can be manipulated. The type definition we have in mind is based on the class concept of SIMULA 67 [Dah68] and is close to the cluster concept in CLU [Lis74, Sch75] and the form concept in ALPHARD [Wul74, Wul76]. A slight difference from the SIMULA class is that the internal structure implementing the data objects is completely hidden from its users. Although it is possible to do this in SIMULA, the default rule gives the user complete access to the internal structure. For the purpose of this paper, we adopt a more Puritan view and do not allow any direct access at all to the internal structure of a typed object. Type definitions will be presented in an intuitive, informal syntax.

The ways in which a typed object can be manipulated are given by the type definition as a set of *operations*. An operation is a procedure or macro whose specification is made known to the users of the typed object. The set of operations is complete in the sense that a typed object cannot be accessed in any other way than through one of the defined operations.

The state resulting from the manipulation of a typed object is a function of the starting state and the applied operation. This implies that the state of an inactive object (i.e. one not currently being operated on) can be characterized by the initial state of the object and its execution history (i.e. the operations which have been applied and their application order). It is often useful to limit the set of histories that can be generated to a desirable subset of those possible. This can be achieved by incorporating a *path expression*[Hab75] in the type definition. The path expression defines the set of legal sequences in which the operations of a type can be applied to an individual object of that type.

We mentioned as the primary purpose of using type definitions the abstraction from implementation. Therefore, type definitions are an important *design* tool [Flo75]. The designer deliberately chooses to impose the rules and restrictions of type definitions on his design language in order to write more reliable programs. These restrictions enable the programmer to make verifiable assertions about his programs. Enforcing the rules does not necessarily result in much runtime overhead. The types of operands and parameters and the legality of the applied operations are statically checkable. Many operations can be written as macros so that runtime overhead due to parameter passing is avoided.

In the next section we discuss the formal basis for the significance of path expressions to the verification of data types. Our approach is based on a proof methodology provided by Hoare[Hoa72]. Path expressions provide a powerful additional verification tool. The difference between path expressions and Hoare's monitors is discussed briefly in the following section. The rest of the paper is devoted, via several examples, to a demonstration of the usefulness of data types and path expressions for verification purposes.

*This work was supported by the National Science Foundation under grant DCR74-24573.

2. Data Types and Verification

In [Hoa72], Hoare describes a method for the verification of abstract data types. In this method, the task is broken into two levels. At one level, an abstract object and the operations which manipulate that object are described mathematically, using such well defined concepts as sets and sequences. Verification of these operations may be considered to be verification of specification. At a lower level, the abstraction is implemented via suitable algorithms and data structures. The verification task then reduces to a proof that the chosen implementation models the specification. To accomplish this, it must be possible to map the concrete state of an object, c, into the abstract object, A, it represents. This mapping is denoted by $\mathscr{A}$. We write

$$A = \mathscr{A}(c)$$

if A is the interpretation of the concrete object c (we require that $\mathscr{A}$ be a function). Hoare indicates the need for an invariant condition, $I(c)$, which each of the concrete operations must preserve. In fact, I(c) is the characteristic function of the domain of $\mathscr{A}$:

$$\text{domain}(\mathscr{A}) = \{c \mid I(c)\}$$

Examples of such domain-restricting invariants include the demand that a stack pointer never overflow or go negative, and that a linear list remain linear.

One of the major effects that the goal of verifiability has had on the way we write software is the idea that the input state to a program segment be restricted and well-defined. This is, of course, one of the strongest arguments against indiscriminate use of the *goto*- statement, e.g.

```
x←0;
l1: x←x+1;
l2:
```

What is the value of x at l2? Because of the presence of l1 it is necessary to comb the entire program (not even just the scope of l1 if label parameters are allowed) and deduce the set of possible values of x before each "*goto* l1". It is now well understood that the use of *while*- and *repeat*- statements restricts possible program states in a useful way, thus making the task of verification simpler.

The use of abstract data types is another step in the direction of restricting the class of legal program states. In particular, we allow direct data structure access only to a small number of procedures and thus minimize the potential for creating inconsistent structures. Unfortunately, this is not quite enough to make some verification questions easily answerable. Consider the case of a data type implementing a queue as a linked list. We can verify relatively easily that the appropriate links are constructed upon insertion and deletion. In order to verify, however, that no attempt is made to delete an element from an empty queue, we must once again scan the entire programs of every user of type queue. The introduction of path expressions guarantees this, for a small runtime price. In operating systems, where concurrency is important, the path expression becomes a synchronization method whose cost is no more than that of the price of the P and V operations otherwise distributed throughout the system.

We will demonstrate that path expressions are suitable for stating certain program invariants and thus facilitate program verification. Without path expressions it is hard to prove such invariants from the programs. The parsing of path expressions was discussed earlier in [Cam74]. A detailed introduction to path expressions illustrated with several examples is found in [Hab75].

3. A Note on Monitors

Monitors[Hoa74] provide a mechanism for the control of concurrency which in some aspects is similar to that provided by data types with path expressions. We see at least two major differences in philosophy, however, which lead us to prefer the latter approach.

One inherent difficulty with monitors is the restriction which states that the operations of a monitor can never execute concurrently. This restriction is a very "coarse" one, because it dictates that the time spent in execution of one operation must be very short. In particular, it effectively prohibits one monitor from using (being implemented in terms of) another, since when the lower level monitor waits, it prevents the higher level monitor from any further execution. While this is also true for path expressions associated with the same object, operations on different objects of the same type can, at the discretion of the programmer, operate completely in parallel. This makes data types with path expressions appear to be a "simpler" concept, since it is not necessary to make the distinction between "monitor" and "class" and since data types may be hierarchically structured with minimal interference among different objects.

Another advantage of data types with path expressions, which is perhaps more relevant to this paper, is the fact that synchronization conditions are localized in a concise syntax and are not part of the individual operations as they are with monitors. While it is a major step to take the synchronization conditions out of user programs, it is of even further benefit to take them out of the implementation of the operations. The specification of an operation should be ignorant of the existence of other operations. Synchronization is a global property and should therefore be specified globally. The advantages to verification are discussed in succeeding sections.

4. Data Invariants as Path Expressions

4.1. An Introductory Example

I/O devices are serial devices which can execute only one command at a time. The device must not start executing until the user of the device has placed a command in the device command buffer. When the device has executed the command in its command buffer, the user can inspect the device status to check for errors during the execution and send another command. The order of executing these device operations is stated by the path expression

path send; execute; inspect *end*

The operator ; is the sequencing operator. It requires that its operands are executed in left to right order. (This operator is associative, but not commutative.) The brackets *path* and *end* indicate cyclic repetition.

Assuming the existence of the type "command", a type definition for I/O devices is

```
type I/O device =
      var combuf = command;

      path send; execute; inspect end;

  let Dev = I/O device, com = command in

    op send(com, Dev) = Dev.combuf ← com;

    proc execute(Dev) =
          <defined by the hardware (Dev)>;

    op inspect(Dev) = return Dev.combuf

endtype
```

The states of an I/O device are represented by two fields (hidden from the user): combuf, and an implicit field "opstate" for the path expression. The combuf field holds the command that is executed by the device. The opstate field represents the operation state and determines which operation is currently executable. In this example, the opstate field can have three distinct values, indicating which of send, execute, or inspect can next be executed.

A short explanation of the implementation of path expressions is in order. A path expression is implemented by a state variable (opstate in this example), a simple lock, and a prologue and epilogue attached to every function named in the path expression. The prologue-epilogue pairs are generated by the compiler, not by the programmer. The function of the prologue is to lock the object, inspect the state, and determine whether or not the attempted operation can be executed. If so, the operation proceeds; if not, the program attempting the operation is put to sleep (with the object unlocked). The function of the epilogue is to change the execution state and unlock the object. Any sleeping programs which can now proceed are restarted at their prologues.

For each device in the above example, there is assumed to be an implicit process defined by the hardware which is equivalent to

repeat execute(I/O device) *end*

The *let* clause specifies the types of the formal parameters used in the following operation and procedure definitions. The difference between an operation and a procedure is that the former is available to all users of the type definition and the latter remains hidden from the user.

Operations and procedures are distinguished by the keywords *op* and *proc*.

The path expression prescribes the execution history of the operations performed on an I/O device. It assures that this history is described by the regular expression (xyz)*, where x = send, y = execute and z = inspect. Thus, if #(x) denotes the number of executions of x, the path expression guarantees that the relation

$$\#(\text{send}) \geq \#(\text{execute}) \geq \#(\text{inspect}) \geq \#(\text{send})-1$$

is always true. One can then easily prove by induction using this invariant that a user cannot overwrite a command nor inspect the device status before the device has finished executing a command. (The use of other path expression operators, such as '+' for exclusive selection and "*" for indefinite repetition, is described in [Cam74, Hab75].)

4.2. A Formal Example

As a formal example of the relationship of path expressions to verification, consider the definition of a type "bufferpool". We would like to think of a bufferpool as two distinct, non-intersecting sets of buffers whose union is fixed. One set of buffers designates the allocated or "in-use" ones, the other set the unallocated or "free" ones.

$$B = (B_{inuse}, B_{free})$$

where $B_{inuse} \cup B_{free} = \bigcup_{i=1}^{n} \{buffer_i\}$ and $B_{inuse} \cap B_{free} = \{\}$.

Initially, all buffers are free, so

(1) $B^0_{inuse} = \{\}$

(2) $B^0_{free} = \bigcup_{i=1}^{n} \{buffer_i\}$

The superscript 0 indicates the state of an object at initialization. Using Hoare's notation[Hoa69], we specify two operations on bufferpools (we will use uppercase letters when referring to abstract operations, and lowercase letters for their concrete counterparts):

(3) true { X←ACQUIRE(B) } $(\exists b \in B_{free})$

$$[B'_{free} = B_{free} - \{b\} \ \wedge \ B'_{inuse} = B_{inuse} \cup \{b\} \ \wedge \ X = b]$$

(4) $b \in B_{inuse}$ { RELEASE(B,b) }

$$B'_{free} = B_{free} \cup \{b\} \ \wedge \ B'_{inuse} = B_{inuse} - \{b\}$$

In our implementation, we choose to represent a bufferpool as an array of n buffers (assuming the existence of type buffer), with a pointer to separate the in-use buffers from the free ones. The type definition for bufferpool uses a *numeric path element* which takes the form

path (acquire-release)^n *end*

and requires, by definition, that

(5) $0 \le$ #(acquire) - #(release) $\le n$

The following definition for type bufferpool is given (the *with* construct is borrowed from PASCAL[Wir72]):

```
type bufferpool(n:integer) =
     var A = array 1..n of buffer,
         ptr = integer(0);    comment initially 0;

   invariant 0≤ptr≤n;
   path (acquire-release)^n end;

   let bp=bufferpool, buf=buffer in
       op acquire(bp):buffer = with bp do
            begin
            ptr←ptr+1;
            return A[ptr]
            end;

       op release(bp,buf) = with bp do
            begin
            ptr←ptr-1;
            A[ptr+1]←buf
            end

endtype
```

In order to verify the correctness of this implementation, we must first define the mapping into our abstract model:

$$\mathcal{A}(A,ptr) = (B_{inuse}, B_{free})$$

where $B_{free} = \bigcup_{j=ptr+1}^{n} \{A[j]\}$ and $B_{inuse} = B^0_{free} - B_{free}$

Now we must verify the implementation by induction. We first show that the initial concrete object is mapped by $\mathcal{A}$ into the desired initial abstract object. Then, for each operation, we show that the resulting concrete object is both in the domain of $\mathcal{A}$ and mapped by $\mathcal{A}$ into the correct abstract object. Upon initialization,

$$\mathcal{A}(A,ptr) = \mathcal{A}(A,0) = (\{\}, \bigcup_{j=1}^{n} \{A[j]\})$$

is the correct initial abstract object (eqs. 1 and 2), and the invariant,

$$I(A,ptr) \ = \ 0 \le ptr \le n$$

is true for ptr=0. To complete the proof, we must verify equations (3) and (4).

Equation (3) states that any abstract bufferpool which results from an ACQUIRE operation must satisfy the given post-condition. Therefore, we must show that any concrete bufferpool which results from an acquire operation is mapped by $\mathcal{A}$ into an abstract bufferpool which satisfies the post-condition. First, however, we must show that the new concrete bufferpool is one to which $\mathcal{A}$ can be applied.

Because ptr is incremented by one in acquire and decremented by one in release, we can make the observation that

$$ptr = \#(acquire) - \#(release)$$

is invariant for inactive objects. We can use the additional information supplied by the path expression (equation 5) to assert that the relation

$$0 \le ptr \le n$$

is also invariant for inactive objects. And, since

$$domain(\mathcal{A}) = \{ (A,ptr) \mid 0 \le ptr \le n \}$$

we can conclude that all inactive objects are consistent, a fact which, without the path expression, would have been harder to derive.

Verifying the post-condition of equation (3) is now easy. In particular,

$$ptr' = ptr + 1$$

and A[ptr] is the returned value, so

$$\mathcal{A}'(A', ptr') = \mathcal{A}'(A, ptr+1) = (B'_{inuse}, B'_{free})$$

where $B'_{free} = \bigcup_{j=ptr+2}^{n} \{A[j]\} = \bigcup_{j=ptr+1}^{n} \{A[j]\} - \{A[ptr+1]\} =$

$$B_{free} - \{A[ptr+1]\}$$

and $B'_{inuse} = B^0_{free} - B'_{free} = B^0_{free} - B_{free} \cup \{A[ptr+1]\} =$

$$B_{inuse} \cup \{A[ptr+1]\}$$

Let b=A[ptr+1] and the post-condition falls out.

The verification of equation (4) is entirely similar, and hence the proof of the implementation is complete.

4.3. The Bounded Message Queue

The next example is primarily intended to illustrate the relative difficulty of proofs of programs using P and V operations[Dij65, Hab72] vs. programs which use path expressions. Thus, we will be a bit less formal than we were in the preceding example. Consider a message buffer which is used by senders to deposit messages and by receivers for removing and processing messages. Of course, no message can be taken from the buffer when it is empty. Therefore, a receiver finding the buffer empty should go to sleep until a message is deposited. If some receivers are asleep when a sender places a message in the buffer, one of the receivers should be awakened. The buffer also has a finite capacity for holding messages. Therefore, what has been said about receivers and empty buffers must also hold for senders and full buffers.

In order for messages to be processed in FCFS order, the buffer is implemented as a queue. The queue is stored in an array A[0:n-1], where n is the buffer size. The total number of messages sent is recorded in variable s, the total number removed in variable r. Both are initialized to zero. When a new message is sent, s is incremented by one and the message is placed in A[s mod n]. When a message is received and removed, r is incremented by one and the message is taken from A[r mod n]. Both sending and receiving use the elements of A[0:n-1] in a circular fashion.

The two operations on message queues are "send" and "remove". Assuming the existence of type ***message***, a type definition for message queues using semaphores is:

```
type message queue(n:integer) =
    var  mutex = semaphore(1),
         left = semaphore(n),
         filled = semaphore(0),
         r, s = integer(0),
         A = array 0..n-1 of message;

    let Q=message queue, m=message in

        op send(m,Q) = with Q do
            begin
            P(left);
                P(mutex);
                s←s+1;
                A[s mod n]←m;
                V(mutex);
            V(filled)
            end;

        op remove(Q) : message = with Q do
            begin
            local m=message;

            P(filled);
                P(mutex);
                r←r+1;
                m←A[r mod n];
                clear(A[r mod n]);
                V(mutex);
            V(left);
            return m
            end

endtype
```

The "mutex" semaphore is used to build a critical section around the actual depositing and removing of a message. (We will assume the standard mutual exclusion properties[Hab72] without explicit mention in the proof.) The "left" semaphore is initialized to n, the maximum queue size. The function of left is to insure that a process is blocked if it attempts to deposit a message in a full queue. Similarly, the "filled" semaphore is used to block any process which tries to do a remove when the queue is empty. The "clear" procedure used in remove (and which we have not defined) is assigned the task of cleaning up a message slot.

The crucial thing to prove about message queues is that, in fact, semaphores filled and left correctly perform their intended functions, i.e. we would like to be sure that once inside its critical section, the send operation is guaranteed to find an empty slot in which to place its message (and the similar statement for remove).

Let us consider relations on the length of the queue (qlength). Conceptually, qlength is incremented by one when s is incremented in send, and decremented when r is incremented in remove. Therefore,

$$(1) \quad \text{qlength} = s - r \geq \text{filled}$$

The inequality is true initially and, by induction, every increment of filled (i.e. every V(filled)) is preceded by an increment of qlength, and every decrement of qlength is preceded by a decrement of filled (i.e. P(filled)). By a similar argument,

$$(2) \quad \text{qlength} \leq n - \text{left}$$

Both equations (1) and (2) are always true. However, inside the critical sections of send and remove, we can make stronger statements. At the start of the critical section of send, we have successfully passed a P on left without having done the corresponding V on filled. Hence,

(3) qlength < n-left (when s is incremented in send)

Similarly,

(4) qlength > filled (when r is incremented in remove)

Since both left≥0 and filled≥0 (from the definition of semaphores), we obtain

(5) qlength < n (when s is incremented in send)

and

(6) qlength > 0 (when r is incremented in remove)

Equation (5) states that when the send operation enters its critical section it will find an empty slot in which to place its message. Equation (6) states that when remove enters its critical section, it will find a message to remove. The proof is therefore complete.

Our proof, while not particularly complicated, was nonetheless undesirably tedious. The corresponding proof for a type definition using path expressions is entirely trivial. Consider the new type definition:

```
type message queue(n:integer) =

    path (send - remove)^n end;

    let Q=message queue, m=message in

        op send(m,Q) = with Q do
            begin
            s←s+1;
            A[s mod n]←m
            end;

        op remove(Q) : message = with Q do
            begin
            local m=message;

            r←r+1;
            m←A[r mod n];
            clear(A[r mod n]);
            return m
            end

endtype
```

Suppose we wish to prove the same properties we proved about the P/V solution. We know from the definition of the path expression that

$$0 \leq \#(\text{send}) - \#(\text{remove}) \leq n$$

and that (because send and remove are entirely mutually excluded on the same queue)

$$\text{qlength} = s - r = \#(\text{send}) - \#(\text{remove})$$

Hence, send will not execute on a full queue or else qlength must increase beyond n. Also, remove will not execute on an empty queue or else qlength must decrease below 0. The reduction in proof effort between the two solutions is clear.

In our example, a higher degree of concurrency can be achieved if the receiving of messages is separated from the removal of processed messages. We now allow the existence of received but undeleted messages. If the total number of received messages is recorded in r and the total number of deleted messages is recorded in d, (with s as defined previously) we require

$$d \leq r \leq s \leq d+n$$

The remove operation of the earlier version is split in two. Operation "receive" increments r and returns the message in A[r mod n]. This assures that the elements of A are processed in a circular fashion. Operation "delete" increments d and clears A[d mod n].

To govern the execution of the (now) three operations, we will use a path expression which generalizes the numeric path element introduced earlier. The expression

path (send - receive - delete)^n *end*

will ensure that the relation

#(send) ≥ #(receive) ≥ #(delete) ≥ #(send)-n

is invariant. The modified type definition for message queues is

```
type message queue(n:integer) =
    var  A = array 0..n-1 of message,
         s,r,d = integer(0);

    path (send - receive - delete)^n end;

    let Q = message queue, m = message in

        op send(m,Q) = with Q do
            begin
            s←s+1;
            A[s mod n]←m
            end;

        op receive(Q) : message = with Q do
            begin
            local m=message;

            r←r+1;
            m←A[r mod n];
            return m
            end;

        op delete(Q) = with Q do
            begin
            d←d+1;
            clear(A[d mod n])
            end

endtype
```

The message processor is given by

repeat process(receive(Q)) *end*

and the message deleter by

repeat delete(Q) *end*

Achieving this higher degree of concurrency can be of importance if the messages contain commands which must be executed by a device with a critical time constraint. The message processor of this new version is immediately ready to process the next message (if any) so that it can keep the device busy. In the preceding version, the message processor had to delete a message from the queue before it could process it. Deleting a message is now a separate task. Moreover, the process which deletes the messages can be assigned additional duties (such as notifying a sender that his message has been processed) without affecting the processing of messages. We still must show that the messages are processed and deleted in the proper order. A message must not be deleted before it has been processed, the oldest unprocessed message must be the next to be received, and the oldest processed message must be the next to be deleted.

Our abstract model of the message queue consists of the sequence of messages Q_1, Q_2, Q_3, ... in the order sent, and the three integers s, r, and d. Sending a new message corresponds to extending the sequence with a new element. The length of the sequence is given by the value of s. The path expression guarantees the invariance of

$$d \le r \le s \le d+n$$

because d=#(delete), r=#(receive), and s=#(send). This means that the variables d and r point to existing messages in the sequence if their values are non-zero. The mapping from the concrete object to the abstract sequence is given by

$$@(A[(d+i) \bmod n]) = Q_{d+i} \quad \text{for } i=1,2,\ldots,s-d$$

The mapping is clearly one-to-one and preserves the order of the messages. Receiving a message corresponds to moving the pointer r ahead to the element after Q_r in the sequence. Likewise, deleting a message corresponds to moving the pointer d ahead to the element after Q_d in the sequence.

In order to show that the operations send, receive, and delete are meaningful, we must show that

1) there is room for a new message when sending is permitted

2) there is an unprocessed message when receiving is permitted

3) there is a processed and undeleted message when deleting is permitted.

Since the path expression guarantees that the relation among d, r, s, and n will hold after executing send or receive or delete, we have

(1) $(r \le s = s+1 \le d+n) \rightarrow (r \le s < d+n)$ when sending is permitted

(2) $(d \le r' = r+1 \le s) \rightarrow (d \le r < s)$ when receiving is permitted

(3) $(d' = d+1 \le r \le s) \rightarrow (d < r \le s)$ when deleting is permitted.

Equation (1) shows that there is room for a new message when sending is permitted, equation (2) that the message processor does not operate on an empty slot, and equation (3) that a message cannot be deleted before it is processed. This proves the most important aspect of the abstract model. The formal proof that the message queue is a consistent implementation of the model is omitted.

5. Conclusion

Data types are an important design tool both because of the freedom of abstraction they allow and because they contribute to the localization of design decisions. This localization contributes not only to program modifiability, but also to ease of verification. We assert that the goal of writing verifiable software is more easily attained when verification can be directed toward the definition of a concept rather than its usage.

Path expressions aid in the desired localization by expressing synchronization conditions in the form of invariant relations on data structures. These invariant relations, which have previously needed to be (often) laboriously derived from multiple concurrent programs, now become a separate part of the definitions of data types, thus easing the verification task. In addition, with regard to information distribution, the synchronization conditions required to maintain the integrity of a data structure are separable from the individual operations on that structure. This is related to the fact that most synchronization is a feature solely of the implementation, designed to ensure that the abstract state of the system is always well-defined. Path expressions provide a vehicle for describing synchronization without over-complicating the relationship between specification and implementation.

Acknowledgement

The authors would like to thank D. Jefferson for his suggestions which have helped to strengthen the paper.

REFERENCES

Cam74 Campbell, R. H. and Habermann, A. N., The Specification of Process Synchronization by Path Expressions. *Lecture Notes in Computer Science vol. 16*, Springer Verlag (1974).

Dah68 Dahl, O. J. et al, Simula 67 Common Base Language. Norwegian Computing Center, Oslo (May 1968).

Dij65 Dijkstra, E. W., Solution of a Problem in Concurrent Programming Control. *Comm. ACM 8,9* (Sept. 1965), 569.

Flo75 Flon, L., Program Design With Abstract Data Types. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (June 1975).

Hab72 Habermann, A. N., Synchronization of Communicating Processes. *Comm. ACM 15,3* (March 1972), 171-176.

Hab75 Habermann, A. N., Path Expressions. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (June 1975).

Hoa69 Hoare, C. A. R., An Axiomatic Basis for Computer Programming. *Comm. ACM 12,10* (Oct. 1969), 576-580.

Hoa72 Hoare, C. A. R., Proof of Correctness of Data Representations. *Acta Informatica 1* (1972).

Hoa74 Hoare, C. A. R., Monitors: An Operating System Structuring Concept. *Comm. ACM 17,10* (Oct. 1974), 549-557.

Lis74 Liskov, B. and Zilles, S., Programming With Abstract Data Types. *SIGPLAN Notices* (April 1974), 50-59.

Par72 Parnas, D. L., Information Distribution Aspects of Design Methodology. Proceedings of the IFIPS Congress 71, Vol. 1 (1972).

Sch75 Schaffert, C., Snyder, A., and Atkinson, R., The CLU Reference Manual. MAC-TR, MIT (June 1975).

Wir72 Wirth, N., The Programming Language PASCAL (Revised Report). Berichte der Fachgruppe Computer-Wissenschaften, EidgenossischeTechnische Hochschule, Zurich (1972).

Wul74 Wulf, W. A., ALPHARD: Towards a Language to Support Structured Programs. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (April 1974).

Wul76 Wulf, W. A., R. London, and M. Shaw, Verification and Abstraction in ALPHARD. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa. (to appear 1976).