STAN-CS-77-607

SPECIFICATIONS AND PROOFS FOR ABSTRACT DATA TYPES IN CONCURRENT PROGRAMS

by

Susan S. Owicki

Technical Report No. 133

April 1977

This work was supported by the Air Force Office of Scientific Research under Contract No. F49620-77-C-0045.

STANFORD ELECTRONICS LABORATORIES

STANFORD UNIVERSITY · STANFORD, CALIFORNIA

STAN-CS-77-607

SPECIFICATIONS AND PROOFS FOR ABSTRACT DATA TYPES IN CONCURRENT PROGRAMS

by

Susan S. Owicki

Technical Report No. 133

April 1977

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

This work was supported by the Air Force Office of Scientific Research under Contract No. F49620-77-C-0045.

Digital Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Technical Report No. 133

April 1977

SPECIFICATIONS AND PROOFS FOR ABSTRACT DATA TYPES IN CONCURRENT PROGRAMS

by

Susan S. Owicki

ABSTRACT

Shared abstract data types, such as queues and buffers, are useful tools for building well-structured concurrent programs. This paper presents a method for specifying shared types in a way that simplifies concurrent program verification. The specifications describe the operations of the shared type in terms of their effect on variables of the process invoking the operation. This makes it possible to verify the processes independently, reducing the complexity of the proof. The key to defining such specifications is the concept of a private variable: a variable which is part of a shared object but belongs to just one process. Shared types can be implemented using an extended form of monitors; proof rules are given for verifying that a monitor correctly implements its specifications. Finally, it is shown how concurrent programs can be verified using the specifications of their shared types. The specification and proof techniques are illustrated with a number of examples involving a shared bounded buffer.

INDEX TERMS: Program verification, program proving, concurrency, parallel programs, abstract data types, shared types, and operating system design.

1. INTRODUCTION

An important development in structured programming is the use of data abstractions. An abstract data type defines a class of abstract objects and the set of operations on those objects. Considerable effort has been devoted to issues related to data abstraction: specification of the abstract type ([Guttag 75], [Guttag et al 76], [Liskov and Zilles 75], [Liskov and Berzins 76], [Parnas 72]), programming languages for expressing data abstractions (notable are CLU [Liskov 76) and Alphard [Wulf 76]), and proof methods for data abstractions ([Hoare 72], [Neumann 75], [Schorre, 75], [Shaw 76], [Spitzen 75], [Wulf 76]). In this paper these issues are considered as they arise in concurrent programs, where data abstractions are shared between parallel processes. The major focus will be on axiomatic proof techniques, in the style suggested by Hoare [69]. Verificationof both the implementation of an abstract data type and the processes that use it will be considered.

The only feasible way to verify a complex system is to compose the system proof from independent proofs of its modules. Abstract data types facilitate this approach. One can first specify and verify the type and its implementation, then use the specifications, rather than the detailed implementation, in verifying higher-level modules. It is also possible to verify each process in a concurrent system independently, provided that the processes access shared data in a disciplined manner (as with monitors or critical regions). This is accomplished by proving each process using only variables that can not be modified by other processes. This separation of processes greatly simplifies the proof (for comparision see [Lamport 75] and [Owicki and Gries 76b], where process proofs are not so completely separated).

To make such proofs possible, each operation of a shared type must be described in terms of its effect on variables of the process invoking the operation. Section 2 shows how a new concept, private variables, can be used to obtain such specifications; private variables are components of a shared object, but belong to just one process. Section 3 discusses the implementation of shared data types by an extended form of monitors, in which private and auxiliary variables are included for

the sake of proofs. Section 4 presents the rules for proving that a monitor satisfies its specifications, and sections 5 and 6 discuss the verification of concurrent processes that use shared data types.

Throughout the paper the abstract type "bounded buffer" will be used as an illustrative example. It consists of a buffer capable of holding N elements, and two operations:

append(a): wait until the buffer is not full, then
add a to the end of the buffer

remove(b): wait until the buffer is not empty, then
remove its first value and return it in b

More precise specifications are given in the next section.

Although the discussion of the bounded buffer here is primarily intended to illustrate the specification and proof techniques, it is also of interest in its own right. Buffers have many uses in concurrent systems, and other concepts, such as queues and message-passing operations, can be described in very similar terms. Thus the specification of the bounded buffer should be applicable to the verification of a number of concurrent systems.

2. SPECIFICATIONS

The specifications of an abstract data type form the interface between the program module which implements the type and the modules which use it. Program verification consists of proving that the implementation satisfies its specifications, and then employing the specifications to verify the modules that use the type. This separation simplifies verification; it also enhances modularity, since the method of implementation may be changed without affecting the correctness of the program, as long as the new implementation also satisfies the specifications.

The specifications for a shared data type are given in the form of assertions that can be incorporated into the proofs of concurrent processes. So that the proof of a process is independent of the actions of other processes, it must contain only safe assertions, i.e. assertions whose

free variables can not be modified by other processes. Thus the assertions that describe the effect of an abstract operation must also be safe. This is made possible by including private variables in the abstract type. A private variable t of type T is declared by var t: private T; this means that there is one instance of t for each process that uses the shared object. The instance of t belonging to process S can be changed only by execution of an operation invoked by S. Thus that instance of t may be used safely in the proof of S. We will use array notation for private variables; var t: private T is interpreted as var t: array process id of T, and t[S] denotes the instance of t for process S. In describing the effects of an operation, t[#] denotes the instance of t belonging to the process that invokes the operation.

The table below gives the format for specifications of a shared data type. Each clause gives the name of an assertion, with the free variables it may contain indicated in parentheses.

Specifications

typename($\bar{p}$): declaration of component variables

requires: Requires(p)

initially: **Init($\bar{a}$)**

invariant: I(U)

operations:

operation-name (var $\bar{x}$; $\bar{y}$)

entry: **entry($\bar{x}$, $\bar{y}$, $\bar{z}$[#])**

exit: **exit($\bar{x}$, $\bar{y}$, $\bar{z}$[#])**

where $\bar{a}$ = parameters and component variables of the type

$\bar{p}$ = parameters of the type ($\bar{p} \subseteq \bar{a}$)

$\bar{z}$ = private variables ($\bar{z} \subseteq \bar{a}$)

$\bar{z}$[#] = private variables of calling process

$\bar{x}$ = var parameters

$\bar{y}$ = value parameters

Let us consider each clause in turn. First, the name and parameters of the abstract type are given, followed by its components. Requires is a condition which must be satisfied when an instance of the type is created; for example, for the bounded buffer Requires assures that the buffer size is positive. **Init($\bar{a}$)** gives the initial value of a newly

created instance of the type. *I(U)*, the invariant, is a consistency assertion about the possible values that can be assumed by U. It is true for the initial value, and is preserved by each operation, although it may fail to hold temporarily during execution of an operation.

Each operation is defined by giving its name and the names and types of its formal parameters. Following Pascal, the formal parameter list contains var parameters, which may be modified by the operation, and value parameters, whose values are not changed. Two assertions describe the effect of the operation. The entry assertion gives the conditions required for correct performance; it is the programmer's responsibility to insure that the entry condition is satisfied each time the operation is invoked. The exit clause describes variable values upon completion. Note that entry and exit describe the operation in terms of private variables and parameters; they are safe assertions and may be used in the proof of a process which invokes the operation.

Specifications for the abstract type bounded buffer are given below; they are adapted from specifications proposed by Good and Ambler [1975] for concurrent programs synchronized with message buffers. The buffer stores values of type message, not defined here. The notation $\langle x_1, x_2, \ldots x_n \rangle$ denotes the sequence whose elements are $x_1, x_2, \ldots, x_n$. The empty sequence is written <>. X @ Y is the concatenation of the sequences X and Y. If X is nonempty, its first element is first(X) and X=<first(X)> @ tail(X); similarly, last(X) is the last element in X, and X=head(X) @ <last(X)>. The number of elements in X is length(X)*. If t is a private variable, ⌜t⌝ denotes the bag containing the values of all instances of t.

Specifications for the Bounded Buffer

bb(N:integer)

record buf : sequence of message
comment length (buf) ≤ N
instream: sequence of message
comment sequence of values appended to bb
outstream: sequence of message
comment sequence of values removed from bb

in: private sequence of message
comment values appended by each process
out: private sequence of message
comment values removed by each process

requires: N > 0

initially: buf = instream = outstream = in = out = <>

invariant: length(buf) ≤ N ∧
instream = outstream @ buf ∧
ismerge(instream, {in}) ∧
ismerge(outstream, {out})

operations:

append(a: message)
entry: in[#] = i' ∧ out[#] = o'
exit: in[#] = i' @ <a> ∧ out[#] = o'

remove(var b: message)
entry: in[#] = i' ∧ out[#] = o'
exit: in[#] = i' ∧ ∃c(b=c ∧ out[#] = o' @ <c>)

The bounded buffer has a single parameter N, the buffer size; because of the requires clause, N must be positive. The data for a bounded buffer is a record consisting of sequences buf (the actual buffer), instrearn, outstream, in, and out. Variables instream and outstream record the global history of buffer operations by storing the sequence of values appended to and removed from the buffer. The private variable in[S] contains the sequence of values appended by process S, while out[S] contains the values removed by S. We will see in section 3 that some of these variables are needed only for proofs, and do not have to be included in an implementation. Initially, all sequences are empty. The invariant states that only N items can be in the buffer (length(buf) ≤ N), that values appended to the buffer either have been removed or are still in the buffer (instream = outstream @ buf), that the global input history is some merge of the private input histories (ismerge(instream, {in})), and that the global output history is a merge of the private output histories (ismerge(outstream, {out})). The predicate ismerge(X,Y), where X is a sequence and $Y = \{y_1, y_2, \ldots, y_n\}$ is a bag of sequences, is defined by

ismerge(<>,Y) = true if y_i = <>, 1 ≤ i ≤ n

ismerge(X' @ <x>, Y) = true if

$y_k = y_k'$ @ <x> for some 1 ≤ k ≤ n

and ismerge(X', y_1, ... y_k', ... y_n)

ismerge(X,Y) = false otherwise

The behavior of append and remove is defined by their entry and exit assertions. For append, the value a is added to the private input history of the invoking process, while the private output history remains unchanged.

Although append(a) must also change the value of buf and instream, this fact is not explicitly included in the exit clause (it is implied by the exit clause and the invariant, however). This is because the exit assertion will be used in verifying the processes that invoke append, and in that context only the effect on private and local variables is relevant. For remove, the exit condition states that some (unknown) value is returned in b and appended to the process's private output history. One can deduce from the invariant that the value returned must be the first one in buf, but buf, as a shared variable, can not appear in the exit condition. This is an accurate reflection of the fact that, from the viewpoint of a process invoking remove, it is not generally possible to predict what value will be returned.

It is interesting to compare the bounded buffer specification given here to specifications suggested by Hoare [74]. Expressed in our notation, Hoare's specification is

bb2(N): record buf sequence of message

requires: N > 0

initially: buf = <>

invariant: length(buf) ≤ N

operations:

append(a:message)

entry: f = buf'

exit: buf = buf' @<a>

remove(var b:message)

entry: = buf'

exit: b = first(buf') ∧ buf = tail(buf')

Hoare's specification is shorter than ours, and it completely describes the effects of the bounded buffer operations. However, it is harder to use in proofs of concurrent programs because it does not provide any private variables. For example, although the effect of bb2.append is buf = buf' @ <x>, one cannot use

{true} bb2.append(x) {x = last(buf)}

in the proof of a process that invokes append. This is because other processes can also append and remove elements from the buffer; in fact, x may not even be in the buffer by the time append(x) returns control to the invoking process.

A valid use of append is

{true} bb2.append(x) {x ∈ buf or x has been removed by another process).

Our specifications give a convenient way of expressing this:

{true} bb.append(x) {x = last (in[#])}

and

(x ∈ in[#] ∧ bb.I) ⊃ (x ∈ buf v ∃S(x ∈ out[S])).

Howard [76] gives an informal specification of the bounded buffer. He uses variables like instream and outstream, and his specifications include the invariant instream = outstream @ buf. But he has nothing corresponding to the private variables in and out.

3. IMPLEMENTATION

An attractive means of implementing abstract data types in a parallel programming environment is the monitor, as proposed by Hoare [74] and Brinch Hansen [75]. A monitor is a collection of data and procedures shared by several processes in a concurrent program. The monitor data can be accessed only by invoking monitor procedures; thus the monitor presents in a single place a shared data object and all the code that has access to that object. Monitors also facilitate concurrent programming by ensuring that only one process at a time can operate on the shared data and by providing operations for process synchronization.

The general form of a monitor type definition is given below.

```
class classname: monitor(parameters)
    declaration of monitor data;
    declaration of monitor procedures;
    initialization of monitor data
    end
```

An instance of a monitor is created by the declaration **monitor** mname: classname(parameters). The notation for a call to a monitor procedure is mname.procedurename (var result parameters; value parameters). To simplify program verification the result parameters must be distinct -- see Hoare [71] for a discussion of parameters and program proofs. The value parameters are not modified by the procedure.

A monitor which implements the bounded buffer type is defined below. Some features of monitors which are important for this example (mutual exclusion, conditions, auxiliary variables, and private variables) will be discussed further. A more complete description of monitors is given in Hoare [74]. Auxiliary and private variables were not in the original definition of monitors; they have been added here because of their usefulness in verification.

```
class bb: monitor (N)
    begin
      BBvar: record m_buffer: array 0..N-1 of message;
                    last: 0..N-1;
                    count: 0..N;
                    m_instream, m_outstream:
                        auxiliary sequence of message;
                    m_in, m_out:
                        private auxiliary sequence of message end
      nonempty, nonfull: condition;
      procedure append(a:message);
        begin if count = N then nonfull.wait;
        last := last ⊕ 1; m_buffer[last] := a; count := count + 1;
        m_instream := m_instream @ <a>; m_in := m_in @ <a>;
        nonempty.signal
      end append;
```

```
    procedure rernove(var b:message);
        begin if count = 0 then nonempty.wait;
              count := count-1; b := m_buffer[last⊖count];
              m_outstream := m_outstream @<b>; m_out := m-out @ <b>;
              nonfull.signal
        end remove;
    begin count := 0; last := 0; m_instream := <>; m_outstream := <>;
          m_in := <>; m_out := <> end;
end bounded buffer
```

⊕ and ⊖ are computed modulo N

An instance of the monitor is BB:bb

In order to allow a number of processes to share the monitor data in a reliable fashion, execution of monitor procedures is mutually exclusive; i.e. only one procedure call at a time is executed. If a number of calls occur, all but the first are delayed until the monitor is finished with the first call. This prevents some of the obscure time-dependent coding errors that can occur with shared data.

Synchronization among concurrent processes is accomplished through condition variables in monitors. A condition is a queue for processes. There are two operations on conditions: condition_name.wait and condition_name.signal. A process which executes condition-name.wait is suspended and placed at the end of the condition queue. When a process executes condition_name.signal the first process waiting on the condition queue is reactivated. In order to insure that only one process at a time may execute a monitor procedure, the procedure executing the signal must be suspended while the reactivated procedure uses the monitor.

The bounded buffer monitor uses two conditions, nonempty and nonfull. If the append operation finds that there is no room in the buffer, it waits on condition nonfull. After a remove operation there must be room in the buffer, so remove ends with nonfull.signal. Condition nonempty is used in a similar way by processes trying to remove an element from the buffer.

The bounded buffer monitor il lustrates two added features of monitors: private and auxiliary variables. Auxiliary variables are included as aids

for verification; they are not necessary for the correct implementation of the monitor and may be ignored by a compiler. The importance of such auxiliary variables for proofs of parallel programs is discussed in Owicki [76].

In order to insure that the auxiliary variables are truly unnecessary for a correct implementation, they may appear only in assignment statements x := e, where x is an auxiliary variable and e does not contain any programmer-defined functions (which might have side effects). This guarantees that the presence of auxiliary variables does not affect the flow of program control or the values of non-auxiliary variables. Thus their presence or absence is invisible to a program which uses the monitor.

The auxiliary variables m_instream and m_outstream are history variables in the sense of Howard [76]. In fact, m_instream and m_outstream play the same role as the history variables A and R in Howard's verification of a bounded buffer monitor.

Private variables in a monitor are used to implement abstract private variables, and they have essentially the same meaning. The declaration t: private T creates one instance of the variable t for each process that uses the monitor; t[S] is the instance belonging to process S. A reference to t in a monitor procedure is treated as a reference to t[S], where S is the process which invoked the procedure. Thus it is syntactically impossible for a procedure to modify any private variables except those belonging to the process that invoked it. In this paper all private variables are auxiliary variables. Non-auxiliary private variables might be a useful extension of monitors, but their implementation is not discussed here.

In the bounded buffer monitor, m_in and m_out are private variables which implement the abstract private variables in and out. Private abstract variables must be implemented by private monitor variables, so that it is impossible for one process to modify the private abstract variables of another.

4. VERIFYING THE IMPLEMENTATION

The methodology for proving that a monitor correctly implements its specifications is derived from Hoare's method for abstract data objects in sequential programs [Hoare 72]; it is also closely related to generator induction [Spitzen 75]. The main difference is that the proof must take into account the sharing of the monitor among concurrent processes. One first defines the relation between the abstract object a and the monitor variables m by giving a representation function rep such that $a = rep(m)$. A monitor invariant must also be defined; it is called monitorname.I_M or simply IM and it gives a consistency condition on the monitor variables m just as I does for the abstract variables a. The verification of the monitor consists of proving the following conditions:-

1. $I_M(m) \supset I(rep(m))$
2. {Requires) monitor initialization $\{I_M(m) \wedge Init(rep(M))\}$
3. For each monitor procedure p(var $\bar{x}$; $\bar{y}$)
   $\{p.entry(\bar{x},\bar{y},rep(m)) \wedge I_M(m)\}$
   body of procedure p
   $\{p.exit(\bar{x},\bar{y},rep(m)) \wedge I_M(m)\}$

The proofs can be accomplished with the usual proof rules for sequential statements and the following axioms for wait and signal. Wth each condition variable bi associate an assertion Bi describing the circumstances under which a process waiting on bi should be resumed. Then the axioms for wait and signal are

$\{IM \wedge P\}\ b_i.wait\ \{I_M \wedge P \wedge B_i\}$

$\{IM \wedge P \wedge B_i\}$ bi.signal $(IM \wedge P\}$

where the free variables of P are private, local to the procedure, parameters, or constants. This is an extension of Hoare's original rules [Hoare74]. The assertion P was added to allow a proof to use the fact that the values of private and local variables can not change during wait or signal.

In the bounded buffer example, the relationship between the abstract buffer bb and the monitor data BBvar is given by

bb = (buf, instream, outstream, in, out)
= rep(BBvar)
= (seq(m_buffer, last, count), m_instream,
m_outstream, m_in, m_out)

where seq(b, ℓ, c) = <> if c=0
= seq(b, ℓ⊖1, c-1) @ <b[ℓ]> if c>0

In this case, the function rep is almost an identity function, because the abstract variables instream, outstream, in, and out are directly implemented by the corresponding monitor variables. The abstract sequence buf is implemented by the array m_buffer and variables last and count; function seq gives the value of the abstract buffer determined by the monitor variables.

The monitor invariant for the bounded buffer monitor BB is

$BB.I_M$: $0 \le count \le N \wedge 0 \le last \le N-1 \wedge$
m_instream = m_outstream @ seq(m_buffer, last, count)
∧ ismerge(m_instream, ⟨m_in⟩)
∧ ismerge(m_outstream, ⟨m_out⟩)

The conditions to be verified are

1. $BB.I_M \supset$ bb.I(rep(BBvar)) - obvious from the definition of rep
2. {bb.Requires} initialization $\{BB.I_M \wedge Init(rep(m))\}$

This expands to

{N > 0}
count := 0; last := 0;
m_instream := m_outstream := m_in := m_out := 0;
$\{I_M \wedge$ seq(m_buffer, last, count) = <> ∧
m_instream = m_outstream = m_in = m_out = <>$\}$

The proof is trivial.

3. {m_in[#] = i' ∧ m_out[#] = 0' ∧ I_M}

code for append(a)

{m_in[#] = i' @ <a> ∧ m_out[#] = o' ∧ I_M}

and

{m_in[#] = i' ∧ m_out[#] = o' ∧ I_M}

code for remove(b)

{m_in[#] = i' ∧ ∃c(b = c ∧ m_out[#] = o' @ <c>) ∧ I_M}

A proof outline for remove(b) is given below; append(a) is similar.

Proof outline for BB.remove

Wait assertion for **nonfull**: count < N

for nonempty: count > 0

{I_M ∧ m_in[#] = i' ∧ m_out[#] = o'}.

begin

if count = 0 then

{I_M ∧ m_in[#] = i' ∧ m_out[#] = o'}

nonempty.wait;

{I_M ∧ count ≥ 0 ∧ m_in[#] = i' ∧ m_out[#] = o'}

{I_M ∧ count > 0 ∧ m_in[#] = i' ∧ m_out[#] = o'}

count := count - 1; b := m_buffer[last ⊖ count];

m_outstream := m_outstream @ <b>; m_out := m_out @ <b>;

{I_M ∧ 0 ≤ count < N ∧ m_in[#] = i' ∧

∃c(b = c ∧ m_out[#] = o' @ <c>)}

nonfull.signal

{I_M ∧ m_in[#] = i' ∧ ∃c(b = c ∧ m_out[#] = o' @ <c>}

end

{remove.exit ∧ I_M}

In addition to proving that a monitor satisfies its specifications, one may wish to show that it has other properties (probably related to performance). Howard [76] is an excellent source of techniques for verifying such properties.

5. PROGRAM PROOFS

In this section we show how to verify concurrent programs given the specifications of shared data types. Concurrent execution is initiated by a statement of the form

$$\underline{\text{monitor}}\ M_1:A_1,\ldots,M_m:A_m\ \underline{\text{cobegin}}\ L_1:S_1\ //\ldots//\ L_n:S_n\ \underline{\text{coend}}.$$

The S_i are statements to be executed concurrently, i.e. parallel processes, and Li is the name of process Si. The only variables that may appear in Si are those declared in S_i (its local variables) or constants declared in a block containing the cobegin statement. S_i also has indirect access, through procedure calls, to monitor variables. Thus all variables are protected from the danger of overlapping operations in different processes: they are constants (no modifications), local variables (accessible to only one process), or monitor variables (protected by the monitor mutual exclusion).

The specifications of type A_i are linked to monitor M_i by the convention that M.assertionname refers to the named assertion in the specifications of A_i, with the monitor name M prefixing each shared variable. Thus, given monitor BB: bb, **BB.Init** is the assertion **BB.buf** = BB.instream = BB.outstream = BB.in = BB.out = <>. Then the rule of inference for verifying cobegin statements is

$$\frac{\{P_i\}\ S_i\ \{Q_i\},\ (P_i, Q_i \text{ safe for } S_i,\ 1 \le i \le n)}{\{(\Lambda_i M_j.\mathrm{Init}) \supset (\Lambda_i P_i)\}\ \underline{\text{monitor}}..M_j:A_j..\underline{\text{cobegin}}..L_i:S_i..\underline{\text{coend}}\ \{(\Lambda_j M_j.I) \wedge (\Lambda_i Q_i)\}}$$

(The notation $\frac{P_1,\ldots,P_n}{Q}$ means that Q may be inferred if all Pi have been proved.) Recall that safe assertions can have no free variables

which can be changed by other processes, so P_i and Q_i may only refer to constants and local and private variables of S_i. The effect of the cobegin statement on private and local variables is obtained from independent proofs of the individual processes. For shared objects, the initial assertion can be assumed to hold at the beginning of concurrent execution, and the invariant holds at the end.

Monitor procedure calls in S_i are verified using the entry and exit assertions and the usual rules for procedure calls, as described in Hoare [1972]. The basic rule for a procedure call in process Si is

$$\{M.p.entry\ \bar{x}\ \bar{y}\ \#\ \bar{a}\ \bar{e}\ L_i\}\ M.p(\bar{a};\bar{e})\ \{M.p.exit\ \bar{x}\ \bar{y}\ \#\ \bar{a}\ \bar{e}\ L_i\}$$

where the actual var parameters $\bar{a}$ must be distinct from each other and from the actual value parameters $\bar{e}$. Mp.entry $\bar{x}\ \bar{y}\ \#\ \bar{a}\ \bar{e}\ L_i$ represents the result of substituting actual parameters $\bar{a}, \bar{e}$ for formal parameters $\bar{x}, \bar{y}$ and the name of the calling process Li for the symbol # in Mp.entry.

Hoare's rule of adaptation is also useful: it allows the entry and exit assertions to be adapted to the environment of the procedure call.

$$\frac{\{P\}\ M.p(\bar{a},\bar{e})\ \{Q\}}{\{\exists\bar{k}(P \wedge \forall\bar{a},\bar{z}[L_i](Q \supset R))\}\ M.p(\bar{a},\bar{e})\ \{R\}}$$

where $\bar{k}$ is a list of variables free in P and Q but not R, $\bar{a}$ or $\bar{e}$, and $\bar{z}[L_i]$ is a list of private variables of M belonging to L_i.

. For example, given

$$\{BB.in[L_i] = i' \wedge BB.out[L_i] = o'\}\ BB.append(x)\ \{BB.in[L_i] = i' \ @<x> \wedge BB.out[L_i] = o'\}$$

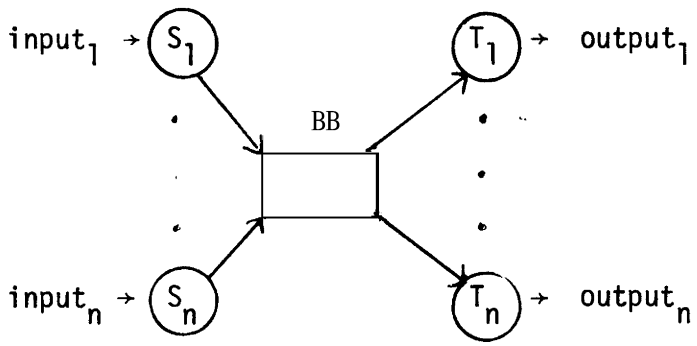
the rule of adaptation allows the inference of

$$\{true\}\ BB.append(x)\ \{x = last(BB.in\ [L_i]))\}$$

or

$$\{in[L_i] \ @ <x> = i_o \wedge out[L_i] = <>\}\ BB.append(x)\ \{in[L_i] = i_o \wedge out[L_i] = <>\}.$$

As an example of verifying a concurrent program, consider the system of processes illustrated below.



Process S_i reads an input stream, $input_i$, of m elements and feeds them into a bounded buffer BB. T_i removes m elements from the buffer (not necessarily the m elements appended by S_i) attd prints them on $output_i$. One can prove

$\{in[S_i] = out[S_i] = <>\}$ Si $\{in[S_i] = input_i$ A $out[S_i] = <>\}$

as outlined below. Let leading(j, X), where $X = <x_1, x,, \ldots . x_k>$ with $k \geq j$, be the initial segment $<x_1, x_2, \ldots . x_j>$ of X.

. Then

```
{BB.in[Si] = <> Λ BB.out[Si] = <>}
Si: begin
    j,x:integer;
    for j := 1 until m do
        {BB.in[Si] = leading(j-1,inputi) A BB.out[Si] = <>}
        read x from inputi;
        {BB.in[Si] @<x> = leading (j,inputi) A BB.out[Si] = <>}
        BB.append(x);
        {BB.in[Si] = leading(j,inputi) A BB.out[Si] = <>}
        od
    {BB.in[Si] = leading(m,inputi) A BB.out[Si] = <>}
    end
{BB.in[Si] = inputi A BB.out[Si] = <>}
```

Note that the assertions for BB.append are similar to the examples given earlier.

A similar proof shows

$\{BB.in[T_i] =$ BB.out $[T_i] = <>\}\, T_i\, \{BB.in[T_i] = <>$ A $BB.out[T_i] =$

$output_i$ A length($output_i$) = m}.

Now suppose these processes are initiated by the statement

L: monitor BB: bb cobegin $S_1//.../S_n//T_1//.../T_n$ coend.

The proof rule for cobegin gives

{BB.Init ⊃ $\bigwedge_i$ (BB.in[S_i] = BB.out[S_i] = BB.in[T_i] = BB.out[T_i] = <>
A $output_i$ = <> ∧ length($input_i$)=m)}
monitor BB: bb cobegin $S_1//.../T_n$ coend
{BB.I A ($\bigwedge_i$ BB.in[S_i] = $input_i$ A BB.out[T_i] = $output_i$ A BB.in[T_i] = <>
A BB.out[S_i] = <> A length($input_i$) = length($output_i$) = m)}

The pre-condition can be simplified to

$\bigwedge_i$ ($output_i$ = <> ∧ length($input_i$) = m)

The post-condition can be rewritten, expanding BB. I, to

ismerge(instream, ⟨$input_i$⟩) A ismerge(outstream, ⟨$output_i$⟩)
A length(instream) = n*m = length(outstream)
A instream = outstream @ buffer.

This implies that instream = outstream, yielding

ismerge(instream, ⟨$input_i$⟩) A ismerge(instream, ⟨$output_i$⟩)

The final theorem is

{(output$_i$ = <> A length($input_i$) = m, 1 ≤ i ≤ n)}
monitor BB: bb cobegin $S_1//.../T_n$ coend
{values printed on ⟨$output_i$⟩ = values read from ⟨$input_i$⟩}

A slight variation on this system has processes S and T, which use the bounded buffer in the same way as S_i and Ti above, plus processes $R_1...R_n$ whose actions are irrelevant except that they do not use the buffer. For these processes

{BB.in[S] = BB.out[S] = <> ∧ length(input) = m}
S
{BB.in[S] = input A BB.out[S] = <> ∧ length(input) = m}

and

{BB.in[T] = BB.out[T] = <> ∧ output = <>}
T
{BB.in[T] = <> A BB.out[T] = output A length(output) = m}

and

{BB.in[R_i] = BB.out[R_i] = <>} R_i {BB.in[R_i] = BB.out[R_i] = <>}

Using the rule for cobegin statements

{length(input) = m ∧ output = <>}
monitor BB: bb cobegin S//T//R_1//.../R_n coend
{BB.I ∧ BB.in[S] = input ∧ BB.out[T] = output ∧
length(input) = length(output) = m ∧ BB.out[S] = BB.in[T] = <>
∧(∧(BB.in[R_i] = BB.out[R_i] = <>))}
i

After expanding BB.I, this simplifies to

{length(input) = m ∧ output = <>}
monitor BB: bb cobegin S//T//R_1//.../R_n coend
{input = output}

6. SPECIFICATIONS FOR SPECIAL SYSTEMS

Often a set of processes use a shared data object in a special way, and a stricter set of specifications is appropriate. For example, if PBB.append(a) is only called with positive values of a, then PBB.remove(b) must return a positive value in b; a stronger entry condition for append implies a stronger invariant and a stronger exit condition for remove. It is always possible to deal with such systems by defining a new set of specifications for the shared object and re-verifying the implementation as described in section 4. In many cases, however, it is possible to derive the stronger specifications from the general ones, without examining the monitor implementation,

Suppose, then, we have already verified that monitor M satisfies a set of specifications, M.Init, M.I, and, for each procedure p, Mp.entry and Mp.exit. Then M must also satisfy the stricter specifications, M.I', M.p.entry', and M.p.exit', provided the following conditions hold:

1. M.Init ⊃ M.I'
2. for each procedure p
   a. {Mp.entry ∧ I} p($\bar{x}$;$\bar{y}$) {M.p.exit ∧ I}
      ⊢ {M.p.pre' ∧ I'} p($\bar{x}$;$\bar{y}$) {M.p.post' ∧ I'}

      where P ⊢ Q means Q can be proved using P as an assumption

   b. p has no wait or signal operations between the first and last modification of variables in M.I'

Condition 1 ensures that the stronger invariant M.I' holds initially. Condition 2a states that each procedure satisfies the stronger entry-exit conditions and preserves M.I'; the fact that it satisfies the original entry and exit and preserves M.I may be used as a hypothesis. The invariant M.I' must also hold at each wait and signal in M; condition 2b ensures that variables in M.I' have either their entry or exit values at a wait or signal, and in either case M.I' holds by rules 1 and 2a. Most monitor procedures seem to follow the pattern described in 2b.

Consider, as an example, the specifications for a positive-value bounded buffer PBB discussed earlier.

PBB.I' = PBB.I $\wedge \forall x(x \in instream \supset x > 0)$

PBB.append.entry' = PBB.append.entry $\wedge$ a > 0

PBB.append.exit' = PBB.append.exit

PBB.remove.entry' = PBB.remove.entry

PBB.remove.exit' = PBB.remove.exit $\wedge$ b > 0

Since the monitor PBB satisfies the restrictions in 2b, the new specifications can be verified by checking conditions 1 and 2a, which clearly hold.

As another example, consider a system in which a producer process adds an increasing sequence of values to a buffer ABB, and no other process executes append. In this system the sequence of values removed by any process must also be increasing. The specifications for ABB are

ABB.I' = ABB.I $\wedge \forall \ell(\ell \neq producer \supset in[\ell] = \langle\rangle)$
$\wedge$ increasing(in[producer])

ABB.append.entry' = ABB.append.entry $\wedge$ # = producer $\wedge$ (length(in[#]) = 0 $\vee$
a > last(in[#]))

ABB.append.exit' = ABB.append.exit

ABB.remove.entry' = ABB.remove.entry

ABB.remove.exit' = ABB.remove.exit $\wedge$ increasing(out[#])

The entry assertion of ABB.append requires that the calling process is the producer (# = producer), and that the value to be appended is greater than the last value appended. This is enough to imply the strengthened invariant. Note that ABB.I' $\supset$ increasing(in[producer]) $\wedge$
instream = in[producer] = outstream @ buffer $\wedge$
ismerge(outstream, ⟨out⟩),

which yields $\forall \ell(\text{increasing}(\text{out}[\ell]))$. Thus the stronger exit condition for ABB.remove can be derived from ABB.I'.

7. CONCLUSIONS

There are two principles underlying the specification and proof methods presented in this paper. The first is that shared data abstractions provide a useful tool for building concurrent programs, and that their usefulness is much increased if they can be precisely specified. The second is that the proof of any program module should depend on assertions that cannot be affected by the concurrent actions of other modules. An easy way to insure that assertions have this property is to limit their use of variables. This not only reduces the complexity of formal verification, but also proves a helpful discipline for informal proofs. The techniques discussed here are suitable for automated verification and for human use. People cannot be expected to produce detailed formal proofs, so it is important that the methods can be used informally and still be (relatively) reliable. The use of safe assertions eliminates most of the complex interactions and the time-dependent error caused by concurrency. Note the importance of private variables in this methodology, both in specification and monitors. Without private variables in the specifications it would be impossible for safe assertions to describe an abstract operation adequately. Private variables in monitors make it easy to verify that a monitor satisfies its specifications.

Any verification technique is worthwhile only if it is general and powerful enough to handle a wide range of problems. The examples in this paper have shown that the proposed methods are adequate for verifying programs which use a bounded buffer in several different ways. The techniques have also been used to prove programs which communicate via message-passing monitors. With slight extensions to handle dynamic resource allocation, it was possible to verify several complex (though small) systems, including Hoare's structured paging system [Hoare 73). More experience is necessary, especially with larger systems, but it appears that these methods will be sufficient for many concurrent programs.

REFERENCES

[Brinch Hansen 73] P. Brinch Hansen. Operating Systems Principles. Prentice Hall, Englewood Cliffs, New Jersey, (1973).

[Brinch Hansen 75] P. Brinch Hansen. The programming language concurrent Pascal. IEEE Trans. on Software Eng., SE-1 No. 2, (June, 1975), pp. 199-207.

[Good and Ambler 75] D.I. Good and A.L. Ambler. Proving systems of concurrent processes synchronized with message buffers. Draft, (1975).

[Guttag 75] J.V. Guttag. The specification and application to programming of abstract data types. Ph.D. thesis, Computer Science, University of Toronto, (Sept. 1975).

[Guttag et al 76] J.V. Guttag, E. Horowitz, D.R. Musser. Abstract data types and software validation. Univ. of Southern California Information Sciences Institute report 76-48, (August, 1976).

[Hoare 69] C.A.R. Hoare. An axiomatic basis for computer programming. Comm. ACM 12, 10 (Oct. 1969), pp. 576-583.

[Hoare 71] C.A.R. Hoare. Procedures and parameters--an axiomatic approach. Symp. on the Semantics of Algorithmic Languages, Springer, Berlin-Heidelberg-New York, (1971), pp. 102-116.

[Hoare 72] C.A.R. Hoare. Proof of correctness of data representations. Acta Informatica I (1972), pp. 271-281.

[Hoare 73] C.A.R. Hoare. A structured paging system Computer J. 16, 3 (1973), pp. 209-215.

[Hoare 74] C.A.R. Hoare. Monitors: an operating system structuring concept. Comm. ACM 17, 10 (Oct. 1974), pp. 549-556.

[Howard 76] J.H. Howard. Proving monitors. Comm. ACM 19, 5 (May 1976), pp. 273-279.

[Lamport 75] L. Lamport. Formal correctness proofs for multiprocess algorithms. Proc. Second Int. Symp. on Programming, April 1976.

[Liskov and Zilles 75) B.H. Liskov and S. Zilles. Specification techniques for data abstractions. IEEE Trans. on Software Eng. SE-1, 1 (March 1975), pp. 7-19.

[Liskov and Berzins 76] B.H. Liskov and V. Berzins. An appraisal of program specifications. Computation Structures Group Memo 141, M.I.T. (July 1976).

[Manna 74] Z. Manna and A. Pnueli. Axiomatic approach to total correctness of programs. Acta Informatica 3 (1974) pp. 243-263.

[Neumann 75] P.G. Neumann, L. Robinson, K.N. Levitt, R.S. Boyer, A.R. Saxena. A provably secure operating system Stanford Research Institute, Menlo Park, California (June 1975).

[Owicki 76] S.S. Owicki. A consistent and complete deductive system for the verification of parallel programs. Proc. 8th ACM Symp. on Theory of Computing, (May 1976), pp. 73-86.

[Owicki and Gries 76a] S.S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. Comm. ACM 19, 5 (May 1976), pp. 280-285.

[Owicki and Gries 76b] S.S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I, Acta Informatica 6 (1976) pp. 319-340.

[Parnas 72) D.L. Parnas. A technique for the specification of software modules, with examples. Comm ACM 15, 5 (May 1972), pp. 330-336.

[Schorre 75] V. Schorre. A program verifier with assertions in terms of abstract data. Systems Development Corporation report SP 3841, Santa Monica, California.

[Shaw 76] M. Shaw. Abstraction and verification in Alphard: design and verification of a tree handler. Computer Science Department, Carnegie-Mellon University, (June 1976).

[Spitzen 75] J. Spitzen and B. Wegbreit. The verification and synthesis of data structures. Acta Informatica 4 (1975), pp. 127-144.

[Wulf 76] W.A. Wulf, R.L. London, and M Shaw. An introduction to the construction and verification of Alphard programs. IEEE Trans. on Software Eng., SE-Z, 4 (December, 1976), pp. 253-265.