Principals of Proving Concurrent Programs in Gypsy

Donald I. Good
Richard M. Cohen
James Keeton-Williams

Technical Report 15 January, 1979

Institute for Computing Science
2100 Main Building
The University of Texas at Austin
Austin, Texas 78712
(512) 471-1901

ACKNOWLEDGEMENT

The mechanisms for specifying, programming, and verifying systems of concurrent processes in Gypsy have been developed over a four-year period. We gratefully acknowledge the significant contributions made during this period by Allen L. Ambler, John H. Howard, Robert H. Wells, Michael K. Smith, Charles K. Hoch, and James C. Browne.

ABSTRACT

Concurrency in Gypsy is based on a unique, formal approach to specifying and proving systems of concurrent processes. The specification and proof methods are designed so that proofs of individual processes are totally independent, even when operating concurrently. These methods can be applied both to terminating and non-terminating processes, and the proof methods are well suited to automated verification aids. The basic principals of these methods and their interaction with the design of Gypsy are described.

Keywords: program verification, program proving, concurrency, parallel programs, formal specifications, message buffers.

CR Categories: 4.20, 4.22, 5.24.

PRINCIPLES OF PROVING CONCURRENT PROGRAMS IN GYPSY

Donald I. Good
Richard M. Cohen
James Keeton-Williams

1. Introduction

The primary objective of Gypsy is to provide an effective, practical language for developing operational software systems of substantial size (1000 - 2000 lines of code) that are formally verified. This goal has been attained by developing Gypsy [6] from Pascal [11] in parallel with an integrated set of formal specification and verification methods. Both programs and their formal specifications are expressed directly in Gypsy, and methods for specifying, implementing and verifying systems of concurrent processes have been a major part of this development.

Gypsy has been used successfully in two major experimental applications involving significant amounts of concurrency. [16] describes a complete message switching network of 16 concurrent processes involving a total of approximately 1500 lines of specifications and 1000 lines of code. [9] gives a 900 line specification of a system of processes that will be used experimentally in conjunction with the ARPANET. In both applications, all parts of the sytems involving concurrency were formally verified. These verifications have contributed strongly to attaining high quality system design.

Methods for specifying, implementing, and verifying systems of concurrent processes in Gypsy have attained several important objectives that are necessary for verifying sizeable, real systems:

1. A general method for specifying systems of concurrent processes has been developed. Such formal specifications are an absolute prerequisite for formal verification.
2. Systems of concurrent processes can be refined into well-defined subsystems so that the proofs of the system and each of its subsystems are mutually independent. This mechanism for operational abstraction is necessary for decomposing the proof of a large system into tractable subproofs.
3. Nonterminating processes can be formally specified and proved. Nonterminating processes commonly occur in real systems.
4. Effective algorithms for mechanically constructing all of the necessary verification conditions have been defined. Automatic verification condition generation is a practical necessity for all except very small programs.

The first three objectives are significant advances toward proving systems of concurrent processes.

The methods used to attain these objectives in Gypsy have several distincive characteristics:

1. Process coordination is done strictly through message buffers.
2. Formal specifications are stated in terms of buffer transaction histories.
3. Buffer operation restrictions are used to simplify the specifications and proofs.
4. The cobegin statement that supports concurrent execution is highly disciplined and well structured.

The following sections describe the basic principles of proving systems of concurrent processes in Gypsy. Major concepts are introduced one at a time. We first describe the formal specification methods for concurrent processes, and then define the "Independence Principle" that has been maintained between specifications and implementations of routines throughout Gypsy to provide independent provability of routines. We then describe proof methods for the various Gypsy program statements involving message buffers and concurrency.

Next we turn our attention to the special problem of specifying and proving non-terminating processes. Finally, these methods are compared with other related work.

2. Specifications

Textually, Gypsy has the appearance of a Pascal-like programming language with embedded formal specification facilities. These specification facilities are incorporated into the language so that they provide the necessary basis for program proofs and so that Gypsy can be used strictly as a formal specification language if desired. We shall not atttempt to provide a complete description of the Gypsy specification language, but rather we shall concentrate primarily on specifications for routines (procedures and functions) because of their direct relation to concurrency.

2.1 External and Internal Specifications

The specifications for a routine may be either external or internal. External specifications are potentially visible to callers of the routine; internal ones are not.

External specifications of routines consist of two parts: a required interface specification and an optional functional specification. The interface specification defines the interface between the routine and its caller. It consists of the routine header which defines the name of the routine and its formal parameters. The only non-local data objects to which a Gypsy routine may refer are formal parameters and globally defined constants. Therefore, the formal parameter list gives a complete description of the interface between the routine and its calling environment. Functional specifications are stated as entry and exit assertions that are interpreted as weak pre- and post-conditions for the routine. Essentially, functional specifications are boolean expressions that may refer only to global constants or formal parameters in the interface specification.

For example, the external specifications of a simple producer/consumer procedure that moves objects from one sequence to another might be written as:

```
procedure PRO_CON(var R:  OBJ_SEQ;
                      S:  OBJ_SEQ) =
begin
   exit R=S;
end;

type OBJ_SEQ  = sequence(MAXSIZE) of object;
const MAXSIZE = 100;
type OBJECT   = . . . ;
```

The interface specification of procedure PRO_CON is the header, which precedes the = sign, and the functional specification is defined by the exit specification. The pre-condition true is assumed if no explicit entry specification is given. (Note: In Gypsy, unit declarations (procedures, functions, types, constants) may be given in any order.)

Internal specifications are specifications describing the internal operation of a routine. The most common form is the assert specification, typically used in inductive assertion proofs. Because these specifications may refer to local variables and, therefore, might reveal information about a particular implementation of a routine, Gypsy does not permit them to be visible outside the routine in which they appear. This is consistent with the conventional use of inductive assertions.

2.2 Message Buffers

Message buffers are the sole process coordination mechanism used in Gypsy. The address space of each active process in a Gypsy program is disjoint from the address space in every other active process. All inter-process communication is via message buffers. Buffers may be declared locally or passed as parameters. Any number of procedures may share access to a buffer created in a common ancestor and passed downward in the calling structure via parameter passage. A process can communicate with another concurrently active process only through a common buffer passed as an actual parameter when the processes were created.

Buffers are a predefined structure in Gypsy and closely resemble those defined in [1]. A typical buffer declaration is

```
type OBJ_BUF = buffer(MAXSIZE) of object;
```

Buffers are strictly first-in, first-out queues upon which language-defined send and receive operations are mutually excluded in time. A buffer may be declared to have a maximal number of elements (as was done with MAXSIZE in the preceding example). If a routine attempts to send to (receive from) a full (empty) buffer, it is blocked until some element is received from (sent to) the buffer.

2.3 Buffer Histories

Since all process communication must be done via message buffers, the complete history of process interactions can be analyzed by examining the histories of message traffic among processes. Every buffer B has two predefined history sequences. TimedInFrom(B,A), is a time stamped "local" input history that rcords the sequence of all objects received from buffer B by a procedure activation A: similarly, TimedOutTo(B,A) is a time stamped local output history that records all objects sent to buffer B by procedure activation A. A stamped history for a buffer of type OBJ_BUF, as defined above, is an object of

```
type timed_stamped_history = sequence of transaction;

type transaction = record (message: object; time: integer);
```

Whenever an object X is received from (sent to) a buffer B by procedure activation A, a new transaction is recorded on the TimedInFrom(B,A) (TimedOutTo(B,A)) history. The transaction that is recorded has X as its message field and T# as its time field, where T# is the time at which A acquires (releases) mutually exclusive access to B. Note that successive time stamps on a given history must be strictly increasing, because each individual procedure activation performs buffer operations strictly in sequence.

These time stamped local histories are the basis for defining several other important histories and functions on buffers. The local unstamped histories are defined implicitly by

```
InFrom (B,A)[i] = TimedInFrom(B,A)[i].message
OutTo (B,A) [i] = TimedOutTo(B,A)[i].message
```

for all elements in the timed histories. The unstamped histories record all messages in the same order as the stamped histories, but time stamps are not included.

If a buffer can be referred to by a procedure, the buffer must be either a formal parameter or a local variable. If buffer variable B is declared local to a procedure P, then a new buffer is created for each activation of P. Every procedure has an implicit formal parameter MYID of type activationid, which names each distinct activation of the procedure. Because B is local to P, only the activation of P and routines called by P can manipulate B.. Therefore, all transactions on B are recorded on the local histories with respect to procedure P. Therefore, we define stamped and unstamped "global" histories for B as

```
TimedAllFrom(B) = TimedInFrom(B,MYID)
TimedAllTo(B) = TimedOutTo(B,MYID)
AllFrom(B) = InFrom(B,MYID)
AllTo(B) = OutTo(B,MYID)
```

TimedAllFrom(B) records all receiving transactions on B by procedure P and any routines activated by P; similarly, TimedAllTo (B) records all sending transactions. Note that successive time stamps on the global histories also will be strictly increasing because only mutually exclusive access is provided for B.

The sequence of objects currently residing in a buffer B is denoted by content(B) and is defined implicitly by

AllTo(B) = AllFrom(B) @ Content(B)

(Gypsy notation: "@" is sequence append. Content(B) is the sequence of objects sent to but not yet received from B. The fullness or emptyness of a buffer is defined in terms of its content.)

empty(B) iff size (content(B)) = 0
full(B) iff size (content(B)) = N

where N is the declared maximal size of B. Gypsy requires N > O so that

not (full(B) and empty(B))

for all buffers B.

The buffers used to communicate among procedures running concurrently must be passed as parameters to those procedures. The functional external specifications for these procedures normally describe the effect of an activation of that procedure on the local histories of its buffer parameters. For example, a procedure that moves N objects from a sequence to a buffer can be specified as

```
Procedure GET(var B:  OBJ_BUF;
                  S:  OBJ_SEQ;
                  N:  integer) =
begin
   entry N in [0..MAXSIZE] and size(S) ge N;
   exit OutTo (B,MYID) = S[1..N];
end;

type OBJ_SEQ = sequence (MAXSIZE) of object;
```

(Gypsy notation: [0..MAXSIZE] is a set of integers from 0 to MAXSIZE. MYID is the implicit const parameter of type activationid. S[1..N] is subsequence S[1],...,S[N]. The elements of all sequences are numbered beginning with one.)

The exit specification states that what is sent to B by procedure GET is exactly the sequence S[1],...,S[N]. It says nothing about what other concurrently operating procedures might be sending to B. These other objects are not recorded on the local history of GET. They are, however, recorded on the global history AllTo(B). Nor does the exit specification say anything about what may be received from B by GET.

Should we want to show that nothing is received from (sent to) a buffer, we may state this in the exit specification. For example, we could extend the exit specification of GET to say

```
exit OutTo(B,MYID) = S[1..N]
 and InFrom(B,MYID) = null(OBJ_SEQ);
```

(Gypsy notation: Null(OBJ_SEQ) is an empty sequence whose type is OBJ_SEQ.)

This approach to stating that nothing was received (sent to) a buffer, however, quickly becomes cumbersome in both specification and proof. It requires extra uninteresting statements in the specifications and these extra statements require extra steps in the proofs. To eliminate these extra steps, Gypsy allows the programmer (or specifier) to declare operation restrictions on buffers. A buffer may be declared as an "input" buffer upon which send operations are not allowed, or as an "output" buffer upon which receive operations are not allowed. Consider the following specification for a GET and a companion PUT procedure.

```
procedure GET(var B: OBJ_BUF <output>;
                  S: OBJ_SEQ;
                  N: integer) =
begin
  entry N in [0..MAXSIZE]
    and size (S) ge N;
  exit OutTo(B,MYID) = S[1..N];
end;

procedure PUT(var R:  OBJ_SEQ;
              var B:  OBJ_BUF<input>;
                  N:  integer) =
begin
  entry N in [0..MAXSIZE];
  exit R = InFrom(B,MYID)
   and size (R) = N;
end;
```

Buffer B in GET is output restricted; no object may be received from it. This implies that InFrom(B,MYID) is empty upon exit. Similarly in PUT, OutTo(B,MYID) is empty by virtue of the input restriction.

Using operation restrictions eliminates a significant number of lines of specifications. These restrictions are designed to be checked statically. A routine may not use an input (output) buffer in a send (receive) operation. The parameter passage rules of Gypsy require a formal parameter to be at least as restricted as its corresponding actual. An unrestricted actual may correspond to a restricted formal, but not vice versa. This information can be used effectively to construct verification conditions in a simpler form than would otherwise be possible.

3. Independence Principle

The independence principle for routines has had a major influence on the design of Gypsy.

Independence Principle

The proof of a routine may only depend on its own specifications and implementation, and upon the external specifications of the routines to which it textually refers.

This principle is the basis upon which the proof of a sizeable program can be decomposed into manageable subproofs. Every routine can be proved independently of every other routine. This principle is not new; it has been generally adhered to in previous developments of proof methods for sequential programs such as the Pascal procedures call rule. However, it is important to recognize this principle as one of the keys to proving sizeable, complex programs. It is necessary to break a large program into pieces small enough and sufficiently simple to permit construction and understanding of the individual proofs. From a large program we can expect a large number of small pieces. If the proofs of N pieces are completely interdependent (each interacts with all the others), we have N*N possible interactions. This is intolerable for all but the smallest values of n. The independence principle insures that the implementation of one routine cannot interfere with the proof of any other routine.

Because concurrent systems are usually more complex than sequential systems, the use of the independence principle is even more important in concurrent systems; we know of no other proof methods for concurrent systems that adhere to this principle. As the following section will show, this principle has been rigidly adhered to in developing the mechanisms for specifying and implementing concurrent programs in Gypsy. Two major factors in this design are the local histories and the cobegin statement.

4. Programs

A Gypsy procedure is implemented by a sequence of statements that refer to its formal parameters and local data objects. The send and receive statements manipulate buffers directly. Buffers may also be manipulated as actual parameters in procedure calls. The proof methods for these are described in terms of the local histories. The fundamental characteristic of buffer parameters is that each reference to a particular buffer has a potentially different value, because the buffer may be manipulated by external processes. Since these external manipulations are not recorded on the local histories, complete descriptions of the local histories can be made independently for each routine.

We will now begin with the most fundamental concepts, then introduce one new concept at a time. These concepts are illustrated by an admittedly contrived set of examples leading up to a conventional producer/consumer system.

4.1 Send And Receive Statements

Send statements enqueue objects to a buffer; receive statements dequeue objects from a buffer. Both statements also append the objects to the appropriate local history. The send statement

```
send X to B
```

is semantically equivalent to the assignment statement

```
TimedOutTo(B,MYID)  :=
        TimedOutTo(B,MYID) <: [time: T#; message: X]
```

(Gypsy notation: The expression "S <: E" denotes the sequence S extended by adding the element E at the end.)

The "[...]" notation is used to denote the corresponding buffer transaction.

Similarly, the receive statement

```
receive M from B
```

is semantically equivalent to

```
M := M#;
TimedInFrom(B;MYID)  :=
        TimedInFrom(B,MYID) <: [time: T#; message: M#]
```

where M# denotes some arbitrary value of the same type as M. (Informally, M# is the first element of content(B)).

When a procedures attempts a send operation, it first tests whether the buffer is full. If it is full, the procedure blocks until some other process performs a receive from the buffer. Similarly, a procedure attempting a receive operation on an empty buffer is blocked until some other process performs a send to that buffer.

Below is a full definition of the procedure GET introduced in Section 2.3.

```
procedure GET(var B: buf_obj<output>;
                  S: OBJ_SEQ;
                  N: integer) =
begin
  entry N in [0..MAXSIZE] and size (S) ge N;
  exit  OutTo(B,MYID) = S[1..N];

  var   K:  int  := 0;

  loop
     assert OutTo(B,MYID) = S[1..K]
          and K in [0..N];
     if K = N then leave end;
     K := K + 1;
     send S[K] to B;
     end;
   end;
```

(Gypsy notation: All compound statements in Gypsy end with the key word "end". A "leave" statement causes control to leave the innermost loop containing the leave statement.)

The proof rule for the send statement within the loop is that of the equivalent assignment statement defining the new local history. The induction theorem that results from the loop assertion is

```
       OutTo (B,MYID) = S[1..K]
  and  K in [0..size(S)-1]
->     OutTo (B,MYID) <: S[K+1] = S[1..K+1]
```

The theorem says, in essence, that equal sequences, each extended by appending the same element, remain equal. The procedure exit specification follows directly from the loop assertion. Note that the specifications and proof rely only upon the history of local buffer transactions.

4.2 Sequential Procedure Call

The procedure call allows operational abstraction. A code segment is encapsulated in a routine definition with its effects on its parameters described by its external specifications. Within a procedure, the effect of any send or receive operation on a buffer parameter (i.e., a non-local buffer) is recorded on the local buffer histories for that procedure. The semantics of the sequential procedure call in Gypsy would be similar to the procedure call in Pascal if we disallowed Pascal's global variables. There is an additional rule that describes how the local buffer histories of the called procedure relate to the local histories of the calling procedure.

Suppose procedure P (with activationid MYID) issues a procedure call "Q(...,B,...)" where B is an actual buffer parameter. This call is equivalent to

```
TIH  := TimedInFrom(B,MYID);
TOH  := TimedOutTo(B,MYID);
Q(...,B,...,Q#);
TimedInFrom(B,MYID)  :=
        TIH @ TimedInFrom(B,Q#);
TimedOutTo(B,MYID)  :=
        TOH @ TimedOutTo(B,Q#);
```

where TIH and TOH are fresh, uniquely named temporary vaiables and Q# is a unique activationid that is the actual parameter corresponds to the implicit formal parameter MYID of procedure Q. The effect of the procedure call is that local histories of the called procedure are appended to the corresponding local histories of

the calling procedure.

As an example of the use of this rule, let us write a simple object mover using the procedures GET and PUT of Section 2.3.

```
procedure PRO_CON1(var R:  OBJ_SEQ;
                       S:  OBJ_SEQ) =
begin
  entry size(S) le MAXSIZE;
  exit R = S;

  var B:  OBJ_BUF;

  GET (B,S,size(S));
  PUT (R,B,size(S));
end;
```

The procedure PRO_CON1 contains no send or receive statements. It invokes procedure GET, which copies the sequence S to the buffer B, and then invokes procedure PUT, which copies the contents of buffer B into sequence R. This "copying" action is described in the external specifications of GET and PUT. The procedure call rule requires us to verify the procedure entry specifications, and then allows us to assume the procedure exit specifications. Thus, for procedure GET, we must verify that

```
    size(S) in [0..MAXSIZE]
and size(S) ge size(S)
```

which follows directly from the entry condition for procedure PRO_CON1. The entry condition for procedure PUT follows from the entry of PRO_CON1 similarly.

We can rewrite the body of PRO_CON1 as the semantically equivalent form

```
TIH := TimedInFrom(B,MYID);
TOH := TimedOutTo(B,MYID);
GET (B,S,size(S),GET#);
PUT(R,B,size(S),PUT#);
TimedInFrom(B,MYID) := TIH
                @  TimedInFrom(B,GET#)
                @  TimedInFrom(B,PUT#)
TimedOutTo(B<MYID) := TOH
                @ TimedOutTo(B,GET#)
                @ TimedOutTo(B,PUT#);
```

Because B is output restricted in GET, the history TimedInFrom(B,GET) is null, reducing the second input history assignement to

```
TimedInFrom(B,MYID)  := TI @ TimedInFrom(B,PUT )
```

Similarly, because B is input restricted in procedure put, we get

```
TimedOutTo(B,MYID)  := TOH @ TimedOutTo(B,GET)
```

Further, since all local histories are initially null (before a procedure has begun execution), TIH and TOH are both null histories.

We now get the verification condition

```
H1.  size(S)  le MAXSIZE
H2.  TimedInFrom(B,MYID) = TimedInFrom(B,PUT#)
H3.  TimedOutTo (B,MYID) = TimedOutTo(B,GET#)
```

```
H4.  OutTo(B,GET#) = S
H5.  InFrom(B,PUT#) = R
H6.  size(R) = size(S)
->
C1.  R = S
```

(Hypothesis H1 is the entry condition of PRO_CON1. H2 and H3 describe the buffer histories local to PRO_CON1. H4, H5 and H6 are the exit specifications from procedures GET and PUT.)

Proof Outline. H2 and H 3 describe the time stamped histories, but equality of the time stamped histories implies equality of the unstamped histories. Since buffer B is local to PRO_CON1, we can use the global history axiom

```
OutTo(B,MYID) = InFrom(B,MYID) @ content(B)
```

to show that content(B) is null. By equalities H4, H5, H6, and properties of the append operator, we conclude that R = S, qed.

4.3 Concurrent Procedure Calls

The cobegin statement is a strict generalization of the sequential procedure call that may create several procedure activations which run concurrently. The only statement that may appear within a cobegin are sequential procedure calls. Thus,

```
cobegin
  Q1(...,B,...);
   .
    .
     .
  Qn(...,B,...);
end;
```

is equivalent to

```
TIH := TimedInFrom(B,MYID)
TOH := TimedOutTo(B,MYID)
Q1(...,B,...Q1#);
...
Qn(...,B,...Qn#);
TimedInFrom(B,MYID) := TIH(MYID,B) @ MIH;
TimedOutTo(B,MYID)  := TOH(MYID,B) @ MOH;
```

where

```
MIH = TimeMerge
        (TimedInFrom(B,Q1#),...
          ...,TimedInFrom(B,Qn#))

MOH = TimeMerge(TimedOutTo(B,Q1#),...
                ...,TimedOutTo(B,Qn#))
```

The TimeMerge function merges time stamped histories according to their time stamps. Each merge is done on the same buffer B. Each time stamp on each stamped history of the merge is unique because of mutual exclusion, and therefore, the merge function is completely deterministic. As in the sequential procedure call, the effect of a cobegin is to append a new history (MIH or MOH) to the local history. The new history is a time deterministic merge of histories produced by each of the called procedures.

The TimeMerge function also has the property

```
TimeMerge(null(trans_SEQ),H) = H
```

where

```
type trans_SEQ = sequence of transaction
```

and H is a time stamped history. Thus, if a cobegin calls only a single procedure, say Q, we get

```
MIH = TimedInFrom(B,Q#), and
MOH = TimedOutTo(B,Q#)
```

and the effect of the cobegin is identical to a sequential call of Q.

Gypsy does not allow dangerous aliasing in procedure calls. No actual var parameter in a procedure call may name an object that contains or is part of another actual parameter in the same procedure calls. This rule applies to concurrent procedure calls, as well. In a cobegin, the requirement is that no actual (non-buffer) var parameter overlaps any of the other actual parameters of any of the procedure calls under the cobegin. This, together with the fact that operations on buffer var parameters are exclusive in time, allows procedures with var parameters to execute concurrently without interference. Thus we can rewrite procedure PRO_CON using a concurrent procedure call.

```
procedure PRO_Con2(var R:  OBJ_SEQ;
                       S:  OBJ_SEQ) =
  begin
        entry size(S) le MAXSIZE;
        exit R = S;
        var B:  OBJ_BUF;

        cobegin
           GET(B,S,size(S));
           PUT(R,B,size(S))
        end;
  end
```

Changing the sequential procedure calls to concurrent procedure calls changes the assignment

```
TimedInFrom(B,MYID)  :=
              TIH @ TimedInFrom(B,GET)
                  @ TimedInFrom(B,PUT)
```

to

```
TimedInFrom(B,MYID)  :=
       TIH @ TimeMerge (TimedInFrom(B,GET),
                        TimedInFrom(B,PUT))
```

The change only affects how the two subhistories are merged. In the sequential case, we know that all buffer transactions in GET precede those in PUT, and so, the TimeMerge reduces to the append function.

The VC for PRO_CON2 is directly analogous to that for PRO_Con1, with the TimeMerge function substituted for the append operator. The proof remains basically the same. Instead of using the reduction

```
Timedh @ null(trans_SEQ) = Timedh
```

we use

```
TimeMerge(TimedH, null(trans_SEQ)) = Timedh.
```

5. Non-Termination

Both the traditional weak and strong interpretation of entry and exit specifications are ineffective for non-terminating processes, because a weak exit is to hold if a process terminates and a strong exit requires that the process must terminate. Specifications for non-terminating processes that have buffer parameters can be stated in Gypsy with a "block" specification. The block specification must hold whenever a process is fully blocked awaiting access to a buffer. This blockage point, in effect, defines a temporary halting point. The block specification is interpreted in a weak sense; it must hold if the process is blocked.

Potential blockage points in a procedure are sends, receives, (sequential and concurrent) procedure calls, and await statements. Send and receive statements are defined as calls to predefined send and receive procedures. These procedures have blockage specifications that their buffer parameter is full or empty, respectively. Thus the potential blockage points are basically procedure calls. A procedure call is blocked if all of its procedure activations are blocked. Ultimately, a procedure may block only because it invokes (perhaps indirectly) the send or receive procedures. Thus, only procedure calls passing buffer parameters may potentially block.

The await statement is a buffer polling mechanism. It is a non-deterministic case statement with send and receive operations as guards for the cases. The await is blocked if all of the buffer oprations are blocked. If the await is not blocked, it behaves as a non-deterministic case statement, selecting between those cases whose guards are not blocked. Thus the await blocks as multiple sends and receives would, and execute as a single send or receive.

The procedure blockage specification must be proven at each potential blockage point. It must be provable from the information available from the internal program assertions in the normal inductive assertion technique, together with any knowledge we may gain about the blockage. For example, blocking at a receive statement implies that the buffer is empty (due to a block specification of receive). Under a sequential procedure call, we know the single procedure activation created has blocked, and so may assume that procedure's block specification. Blockage at a concurrent procedure call, however, is more vague. At least one of the procedure activations must be blocked, but the rest may either be blocked or have already terminated. To express this, we define a predicate, ISBLOCKED(A), which is true if the procedure activation A is blocked. Thus, if a procedure is blocked at the cobegin statement

```
cobegin
   Q1(...);
    .
     .
      .
   Qn(...)
end
```

then we deduce that

```
if ISBLOCKED(Q1#) then Q1_BLOCKAGE
                  else Q1_EXIT fi
and .
     .
      .

and if ISBLOCKED(Qn#) then Qn_BLOCKAGE
                      else Qn_EXIT fi
and (ISBLOCKED (Q1#) or...
       ...or ISBLOCKED(Qn#)),
```

where Q1#,...,Qn# are the activationids corresponding to the procedure activations under the cobegin, and

Qi_BLOCKAGE and Qi_EXIT represent the blockage and exit specifications (respectively) for procedure Qi.

At the blockage point the local histories are updated as they are after a procedure call. (Intuitively, the local histories are extended by whatever transactions were performed by the called procedures before blockage.) Thus the updated local histories will usually appear in the conclusion of a blockage VC.

We can rewrite PRO_CON once again, this time as a system of two non-terminating concurrent processes (a producer and a consumer).

```
        Procedure PRO_CON3(var BIN: OBJ_BUF;
                           var BOUT:  OBJ_BUF) =
        begin
          block not full(BOUT) ->   empty(BIN)
                and OutTo(BOUT,MYID) = InFrom(BIN,MYID);
          exit false; {explicitly assert non-termination}

          var B:  OBJ_BUF;
          cobegin
           TRANSFER(BIN,B);
           TRANSFER(B,BOUT)
        end;
    end

        procedure TRANSFER (var X:  OBJ_BUF<input>;
                            var Y:  OBJ_BUF<output>) =
        begin
          block  not full (Y) -> empty (X)
                 and InFrom(X,MYID) = OutTo(Y,MYID);
        exit false;  {explicitly assert non-termination}

        var M:  OBJECT;

        loop
           assert OutTo*(Y,MYID) = InFrom(X,MYID);
           receive M from X;
           send M to Y;
           end;
        end;
```

Verification of procedure PRO_CON3 requires proof of two verification conditions: one corresponding to termination of the cobegin, the other corresponding to blockage at the cobegin. The termination VC for this example is

```
false -> false
```

which is trivially true.

The blockage VC for PRO_CON3 is

```
        H1.  if ISBLOCKED(T#1)  then
                not full(B) -> empty(BIN)
                and OutTo(B,T#1) = InFrom(BIN,T#1)
             else false fi
        H2.  if ISBLOCKED(T#2)  then
                not full(BOUT) -> empty(B)
                and OutTo(BOUT,T#2) = InFrom(B,T#2)
                else false fi
        H3.  ISBLOCKED(T#1) or ISBLOCKED(T#2)
        H4.  TimedAllTo(B) =
                  TimeMerge(TimedOutTo(B,T#1);
```

```
                        null(TRANS_SEQ),
      H5.  TimedAllFrom(B) = TimeMerge(
                             null(TRANS_SEQ),
                             (TimedInFrom(B,T#2))
      ->
      C1.  not full(BOUT) -> empty(BIN)
           and InFrom(BIN,T#2) = OutTo(BOUT,T#1)
```

where T#1 and T#2 are activationids corresponding to the two activations of TRANSFER. Hypotheses H1 and H2 are the instantiated block and exit specification of activations T#1 and T#2. Hypothesis H3 states that at least one of the two processes is blocked and not yet terminated. Hypotheses H4 and H5 describes how the histories of B in T#1 and T#2 merge to form the histories of B in PRO_CON3. The conclusion is the block specification for PRO_CON3 with InFrom(BIN,T#1) substituted for InFrom(BIN, MYID) and OutTo(BOUT, T#2) substituted for OutTo(BOUT, MYID). These substitutions are required because of the new values assigned to the buffer histories of the cobegin.

Proof Outline. The proof breaks into three cases, depending on the truth values of the ISBLOCKED predicates (H3). The two cases in which a procedure activation is assumed to have terminated (i.e., is not blocked) are trivial because hypothesis H1 or H2 reduces to false. The remaining case assumes both activations of TRANSFER are blocked. Thus we can immediately reduce H1 and H2 to simple implications. We can also eliminate the references to TimeMerge in H4 and H5 because merging with a null history is an identity function. The VC is of the form

```
      P -> (Q -> R)
```

which can be rewritten as

```
      P and Q -> R.
```

Thus "not full(BOUT)" becomes a new hypothesis. Using this new hypothesis and the reduced H2, we can apply modus ponens to deduce

```
      empty(B)
      and OutTo(BOUT,T#2) = InFrom(B,T#2).
```

By the axioms of buffers, empty(B) -> not full(B). Thus we can use modus ponens on H1 to deduce

```
      empty(BIN)
  and OutTo(B,T#1) = InFrom(BIN,T#1).
```

The VC now has the form

```
      H1.  empty(BIN)
      H2.  OutTo(B,T#1) = InFrom(BIN,T#1)
      H3.  empty(B)
      H4.  OutTo(BOUT<T#2) = InFrom(B,T#2)
      H5.  TimedAllTo(B) = TimedOutTo(B<T#1)
      H6.  TimedAllFrom(B) = TimedInFrom(B,T#2)
      H7.  not full(BOUT)
      ->
      C1.  empty(BIN)
      C2.  InFrom(BIN,T#1) = OutTo(BOUT,T#2)
```

Conclusion C1 matches H1, so only C2 remains to be proven. Using H2 and H4 we reduce C2 to

```
  C2.  OutTo(B,T#1) = InFrom(B,T#2).
```

Hypotheses H5 and H6 imply

```
       AllTo(B) = OutTo(B,T#1)
  and AllFrom(B) = InFrom(B,T#2).
```

From empty(B) we can deduce that content(B) is null and, using the definition of content, obtain AllTo(B) = AllFrom(B). C2 follows by transivity of equality. qed.

6. Related Work

The cobegin is a generalization of the sequential procedure call statement. It invokes several procedures concurrently and schedules these subprocesses nondeterministically and fairly. Only procedures can be invoked by a cobegin, and a cobegin of just one procedure is exactly equivalent to a sequential procedure call. Procedures (and also functions) in Gypsy may not refer to non-local variables, hence all variables that are used to communicate between procedures must appear explicitly as parameters. Buffers are the only variables that may be actual var parameters to several procedures operating concurrently. The cobegin structure is the basis for decomposing systems into independently verifiable subsystems. This decomposition is possible even in the presence of distributed processing.

The similarity of the cobegin to a procedure call is in direct contrast to Concurrent Pascal [2] and Modula [17]. In these languages, processes are initialized and then run forever. Restricting the cobegin only to call procedures also is substantially different from [15], which allows arbitrary statement lists as the subprocesses of a cobegin.

The await statement is similar to the guarded command of [5], where the guards are buffer operation statements (send, receive) and the commands are arbitrary Gypsy statement lists. The await waits until it can select (nondeterministically) some guard that is not blocked, does the buffer operations and the statement list, then exits the await.

Process coordination in Gypsy is done strictly by means of message buffers [1], as opposed to semaphores [4], monitors [8], or conditional critical regions [14]. The buffers are strictly FIFO queues upon which send and receive operations are excluded in time. Similar synchronization mechanisms have been used in the RC4000 operating system [1], and Hydra [18]; but as far as we know, message buffers have not been used previously as a basis for specification and proofs.

The Gypsy message buffers are predefined types with predefined axiomatic properties. This is in direct contrast to the monitors of [8], and [10] as well as the conditional critical regions of [14] in which axiomatic properties of each shared object must be specified and proved.

Transaction histories have been used in most approaches to specifying and proving properties of systems of concurrent processes and shared objects ([7]; (Hoare74); (Owicki76); (Howard76). These approaches, however, have used the installation of "ghost" variables to record transaction histories in an ad hoc way. In Gypsy, the histories and the ways in which buffer transactions are recorded are predefined by the language, and the semantics of the buffer operation statements are defined in terms of these histories. An important and unique aspect of Gypsy is the distinction between global and local histories. Global histories record all transactions on a given buffer. The local histories record the effects of a given procedure activation of a given buffer. This is the basis for isolating the effects of a procedure so that it can be specified and proved independently of all other procedures. The time stamped buffer histories are being used as a basis for formal specifications of real-time constraints [3].

Specifications for procedures that manipulate buffers can be stated in terms of entry and exit assertions about the local buffer histories. The technique for nonterminating processes is similar to the intermittent assertions of [13]. A block specification can be stated that is to hold if a procedure is blocked awaiting its mutually exclusive access to a buffer. The block specifications also can be verified independently for each procedure.

Operation restrictions may be declared on a buffer. A buffer may be declared to be used strictly for input or for output. These restictions are in the spirit of [12] and are enforced statically. These restrictions reduce the size of specifications, and their static enforcement significantly simplifies proofs.

The proofs are done by an extension of the inductive assertion method. Assertions may refer to buffers and their respective transaction histories. Verification conditions are constructed and proved independently for each procedure. All consideration of the interactions of procedures running concurrently is isolated to the verification conditions for the cobegin. It is never necessary to consider simultaneously all processes running concurrently, as in [7], or to prove noninterference, as in [14].

7. Summary

This paper has described the basic principles for specifying and proving systems of concurrent processes in Gypsy, including automatable methods for both terminating and nonterminating processes. These methods are based on message buffers for process coordination and a cobegin statment that is a generalization of a sequential procedure call. Proofs of concurrent Gypsy processes are independent in that the implementation of one process cannot affect the proof of another.

The Gypsy language has several more advanced aspects of concurrency that have not been described. These include structures (e.g., arrays) of buffers and the ability to activate arrays of procedure activations within a cobegin. The basic principles described here, however, can be extended to cover these more complicated systems. Such advanced facilities have proven successful in the trial applications listed in the introduction.

References

[1] Per Brinch Hansen.
Operating Systems Principles.
Prentice-Hall, 1973.

[2] Per Brinch Hansen.
The Purpose of Concurrent Pascal.
In *Proceedings ICRS*. Per Brinch Hansen, 1975.

[3] R.M. Cohen.
Formal Specifications for Real-Time Systems.
In *Proceedings Seventh Texas Conference on Computing Systems*. R.M. Cohen, 1978.

[4] E.W. Dijkstra.
Programming Languages.
Academic Press, 1968, Chapter Cooperating Sequential Processes.

[5] E.W. Dijkstra.
Guarded Commands, Nondeterminacy, and Formal Derivation of Programs.
CACM 18-8, 1975.

[6] D.I. Good, R.M. Cohen, C.G. Hoch, L.W. Hunter, D.F. Hare.
Report on the Language Gypsy, Version 2.0.
Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, ICSCA, The University of Texas at Austin, September, 1978.

[7] A.N. Habermann.
Synchronizing of Communicating Processes.
CACM 15-3, 72.

[8] C.A.R. Hoare.
Monitors - an Operating System Structuring Concept.
CACM 17-10, October, 1974.

[9] Gary R. Horn.
Specifications for a Secure Computer Communications Newtork.
Master's thesis, The University of Texas at Austin, 1977.
ICSCA-Cmp-8, The University of Texas at Austin.

[10] J. Howard.
Proving Monitors Correct.
CACM 19(5), May, 1976.

[11] K. Jensen, N. Wirth.
Pascal User Manual and Report.
Springer Verlag, 1974.

[12] A.K. Jones, B.H. Liskov.
A Language Extension for Expressing Constraints on Data Access.
CACM 2l, 5, May, 1978.

[13] Z. Manna.
Is 'Sometimes' Better Than 'Always?'.
In *Proceedings Second International Conf. on Software Engineering*. Z. Manna, 1976.

[14] S.S. Owicki.
Axiomatic Proof Techniques for Parallel Programs.
PhD thesis, Cornell University, Ithaca, N.Y., August, 1975.

[15] S.S. Owicki, D. Gries.
An Axiomatic Proof Technique for Parallel Programs I.
Acta Informatica 6, 1976.

[16] R.E. Wells.
Specification and Implementation of a Verifiable Communications System.
Master's thesis, The University of Texas at Austin, December, 1976.
ICSCA-CMP-4.

[17] N. Wirth.
Modula: A Language for Modular Multiprogramming.
In *Software Practice and Experience*. N. Wirth, 1977.
Vol. 7, 3-35.

[18] W.A. Wulf, R. Levin, C. Pierson.
Overview of the Hydra Operating System Development.
In *Proceedings of the Fifth Symposium on Operating Systems Principles*. W.A. Wulf, 1975.

Table of Contents