VERIFIABLE COMMUNICATIONS PROCESSING
IN GYSY

Donald I. Good
Richard M. Cohen

June 1978  ICSCA-CMP-11

Printed in
Proceedings from COMPCON '78: Computer Communications Networks
Seventeenth IEEE Computer Society International Conference

CERTIFIABLE MINICOMPUTER PROJECT
Institute for Computing Science and Computer Applications
The University of Texas at Austin
Austin, Texas  78712
ABSTRACT

The Gypsy programming and specification languages have unique facilities for describing systems of concurrent processes. These facilities were developed particularly to support the development of formally verifiable and realistic communications processing programs. These facilities and their interaction with the Gypsy design constraint of full verifiability are described. The Gypsy specification and programming facilities for concurrency are illustrated by a simple multiplexor example.

---------
# TABLE OF CONTENTS

1.0 INTRODUCTION ............................................. 1

2.0 PROGRAMS .................................................. 1
   2.1 Synchronization ........................................ 2
   2.2 Concurrency ............................................ 3

3.0 SPECIFICATIONS ............................................ 5

4.0 MULTIPLEXOR EXAMPLE ...................................... 8

5.0 CONCLUSION ................................................ 12

6.0 REFERENCES ................................................ 13
1.0 INTRODUCTION

Gypsy is a fully verifiable language for describing systems programs and their formal specifications. The complete language consists of two strongly intersecting sublanguages: a high-level systems programming language and a formal specification language. The complete Gypsy language [Good, 78] was developed from Pascal [Jensen, 74]. The Gypsy programming language retains most of the generality of Pascal and includes several additional facilities for systems programming. The facilities for concurrency are particularly effective for communications processing applications. The Gypsy specification language integrates three major approaches to formal specification: the "verification" approach, the "state machine" approach, and the "algebraic" approach. These approaches can be used separately or collectively. Either the Gypsy programming language or the specification language can be used independently of the other or they can be used together to describe a formally verified program. Effective formal verification methods that demonstrate that a program is consistent with its specifications have been developed for the complete Gypsy language. In general, verifications can be done either by rigorous proof methods or by validating (evaluating) specifications at run-time.

The Gypsy programming and specification languages have unique facilities for describing systems of concurrent processes. These facilities were developed particularly to support the development of verifiable and realistic communications processing programs. The following sections describe the Gypsy mechanisms for expressing concurrent programs and their specifications as well as some of the interaction of these mechanisms with the Gypsy design constraint of full verifiability. The verification methods, however, are beyond the scope of this paper. These can be found in [Good, 77]. The programming and specification methods are illustrated using a simple multiplexor example.

2.0 PROGRAMS

The Gypsy programming language has been designed specifically to express verified programs for communications processing applications. The general design philosophy has been first to define a verifiable base language for general applications, then to extend the base language with additional capabilities to support general systems programming, and finally to give special consideration to communications processing applications. The base language is strongly based on a subset of Pascal and all extensions from this base have maintained a fully verifiable language. The mechanism for concurrency is one major extension and the only one that is oriented specifically toward communications processing.
At the time Gypsy development began in September, 1974, proof methods for systems of concurrent processes were just becoming a subject of active research. Much of this work centered on monitors [Hoare, 74; Howard, 76], and a version of monitors was the primary mechanism for concurrency in Concurrent Pascal [Brinch Hansen, 75]. Another significant approach to concurrency based on conditional critical regions also was in the early stages of development [Owicki, 75]. These approaches were studied and it was found that in communications processing applications, these more sophisticated approaches typically were used to build message buffers [Brinch Hansen, 73]. Because of their greater simplicity both in verification and specification, and their naturalness for communications processing, message buffers were chosen as the basis for process synchronization in Gypsy. The message buffers and the facilities for concurrency have been designed to provide a practical basis for proving systems of concurrent processes.

One of the essential prerequisites for effective verification of any sizeable system is a well-structured decomposition. It must be possible to decompose a system into subsystems and keep the verification of the system independent of the verification of its subsystems. This general approach is necessary for effective verification of sizable sequential programs; it is even more important for concurrent systems in order to control the increased complexity caused by the interactions of concurrent processes. This factorability has been attained in Gypsy through the design of a highly disciplined, yet effective set of mechanisms for process synchronization and concurrency.

2.1 Synchronization

Message buffers are the sole means of process synchronization in Gypsy; they are the only objects that can be shared among processes running concurrently. Non-buffer objects may be referred to and manipulated by a process that is executing concurrently with other processes only if that process has exclusive access to those objects. These access requirements are enforced by the language.

A buffer is a primitive structure in Gypsy and is declared as

\[
\text{type packetbuf = buffer(bufsize) of packet;}
\]

A buffer of this type may contain from 0 to "bufsize" objects of type "packet". The buffer behaves strictly as a first-come, first-serve queue and all primitive buffer operations in Gypsy automatically provide mutually exclusive access to a buffer. A type declaration also may declare a buffer for use strictly as an input or an output buffer, for example

\[
\text{type packetinbuf = packetbuf<input>;}
\]

\[
\text{type packetoutbuf = packetbuf<output>;}
\]
If, for example, a procedure has a formal parameter of type "packetinbuf", that procedure may use the buffer only for input and not for output. These input and output restrictions are not essential in the language, but they provide a considerable convenience for formal specification and verification. Buffers are types and they may be used as components of other structured types such as arrays and records, for example

\[
\text{type inbufarray} = \\
\quad \text{array(integer[1..n]) of packetinbuf}
\]

This declaration defines an array of input-only packet buffers. The array is indexed from 1 to n.

The primitive buffer operations are invoked by special statements in the language,

\[
\begin{align*}
\text{send object}_x \text{ to buffer}_b, \\
\text{receive object}_x \text{ from buffer}_b, \text{ and} \\
\text{give object}_x \text{ to buffer}_b.
\end{align*}
\]

These operations automatically provide mutually exclusive access to their buffer operands. The "send" and "receive" statements have the expected interpretation. The effect of a "send" is to append a copy of object \(x\) as the newest element in the first-come, first-serve queue for buffer \(b\). If buffer \(b\) is full (contains its declared maximum number of elements), the sending process is blocked until some object is taken from buffer \(b\). Once object \(x\) is appended to the queue, the sending process proceeds. The effect of "receive" is to take the oldest element in buffer \(b\) from the queue and assign it to the object \(x\) in the receive statement. The receiving process is blocked if buffer \(b\) is empty (contains no elements). The "give" statement is similar to the "send" except that it provides a way of appending object \(x\) as the newest element of the queue for buffer \(b\) without making a copy of object \(x\). The "give" statement is part of a general Gypsy memory management facility whose description is beyond the scope of this paper. In brief the "give" statement hides pointer manipulation that permits large objects to be moved through buffers without the overhead of copying required by the "send" operation.

The MUXOUT procedure in the multiplexor example below is an example of a typical use of "send" and "receive" statements. The LPbuf parameter is an input buffer of labelled packets, and muxoutput is an array of n output buffers for unlabelled packets. MUXOUT is implemented as a non-terminating loop that receives a labelled packet LP from LPbuf, examines its label LP.LBL, and sends the unlabelled packet LP.PKT to the output buffer indicated by the label.

2.2 Concurrency

Operations in Gypsy are caused by activations of routines. These routines are similar to the procedures and functions of Pascal. The major differences are
1. "const" parameters in Gypsy are passed by reference rather than by value;
2. routines may not refer to non-local variables;
3. functions may not have "var" parameters;
4. functions may return any type of result rather than just a simple type.

This provides for explicitly defined interfaces between routines and does not allow a function to cause side effects. The primary mechanism for concurrency is the "cobegin" statement, which creates concurrent activations of procedures. An "await" statement provides a mechanism for polling. The "cobegin" and the "await" are the only sources of concurrency in Gypsy.

The cobegin statement and procedure declarations are the mechanisms by which systems of processes can be decomposed into well-structured subsystems, as illustrated in the multiplexor example below. The main procedure of the multiplexor is procedure MUX which decomposes into three subsystems: MUXIN, PACKETMOVER, and MUXOUT. MUX has two parameters: an array of input buffers, muxinput, and a similar array of output buffers, muxoutput. MUX is implemented by MUXIN, PACKETMOVER, and MUXOUT which are invoked concurrently by the cobegin. Each of these is a nonterminating process, so once the cobegin is entered, it should never terminate. When a cobegin is entered, execution of the process containing the cobegin is suspended until all processes invoked by the cobegin terminate; then processing resumes with whatever follows the cobegin. The parent process does not run concurrently with its children. This maintains a simple single-entry, single-exit control structure. In MUX, process MUXIN continually receives packets from the input buffer array, labels them with their source buffer number, and puts the labelled packets into LPbuf1. PACKETMOVER moves labelled packets from LPbuf1 to LPbuf2, and MUXOUT moves labelled packets from LPbuf2 to the output buffer designated by the label.

In the multiplexor example, PACKETMOVER is implemented as a simple "receive" - "send" loop that moves labelled packets. In some cases, it might be necessary to decompose the PACKETMOVER into component processes. For example, it might be necessary to break packets into individual characters, move the characters over some channel, then reassemble them into packets. In such a case, PACKETMOVER might be written as follows:

```pascal
procedure PACKETMOVER
  (var LPin:LPacketbuf<input>;
   var LPout:LPacketbuf<output>)=
begin
  {The following "block" is a specification that is explained in the next section.}
  block
    not full(LPout) ->
```
empty(Lpin)
& infrom(LPin,myid) =
outto(LPout,myid);

var channel:buffer(bufsize) of character;

cobegin
  PACKETBREAKER(LPin,channel);
  PACKETMAKER(channel,LPout);
end;
end;

PACKETBREAKER breaks labelled packets into characters and puts them in the "channel" buffer; PACKETMAKER takes them from the "channel" and reassembles them into labelled packets.

The "await" statement provides a way of polling one or more buffers. It behaves much like a non-deterministic case statement as illustrated in MUXIN. The basic internal structure of the "await" is

on <buffer statement>
  then <statement list>

The "each" clause that precedes the "on-then" construct in effect defines a separate "on-then" construct for each value of "i" in the range 1 to n. (An "each" clause may also be used as a prefix before a procedure call in a cobegin to invoke a number of activations of the same procedure.) The <buffer statement> may be any "send", "receive", or "give" statement. The "await" may nondeterministically select any of the "on-then" clauses whose <buffer statement> is not blocked. It then performs the <buffer statement>, then the <statement list> following the "then", and then exits the "await". The nondeterministic selection must be fair. In MUXIN, the await will block until some input buffer "muxinput[i]" is not empty, receive a packet from that buffer, prepare a labelled packet, and exit the "await". MUXIN then sends the labelled packet to LPbuf.

3.0 SPECIFICATIONS

The Gypsy specification language is an integration of three major approaches to formal specification. Specifications can be stated using the "verification" approach, the "state machine" approach, the "algebraic" approach, or any combination of these. The "verification" approach refers to the placement of various kinds of assertions in the program text. These specifications typically are stated in terms of "entry", "exit" and "assert" expressions that are associated with functions and procedures. The "state machine" approach refers to the specification method proposed by [Parnas, 72], and its subsequent developments (e.g. [Roubine, 76]). The "algebraic" approach refers to the use of algebraic axioms developed, for example, by [Liskov, 73] and [Guttag, 76]. All of these approaches have been developed primarily for sequential programs and the Gypsy techniques for systems of concurrent processes are unique to Gypsy.
Specification of concurrent processes in Gypsy are based on the idea of "buffer histories". Each buffer history records all "send", "receive", and "give" transactions on a given buffer. This idea is common in proof methods for concurrency (see, for example [Howard, 76]), but the histories are normally installed in a program in an ad hoc fashion. In Gypsy these histories are built directly into the language definition and satisfy a given set of properties. The histories are intended primarily as a specification concept and implementation is not required. They do, however, have a rigorous definition that is a sound basis for formal proof.

The fundamental history concept is the "global" history. Two global histories "allto(b)" and "allfrom(b)", are defined for every buffer b. Whenever an object x is appended by a "send" or a "give" to the queue associated with buffer b, a copy of x also is appended to "allto(b)"; likewise, whenever an object x is dequeued from b (by a receive), a copy of x is appended to "allfrom(b)". These global histories, therefore, are sequences of objects (of the type of x) that satisfy the relation

allto(b) = allfrom(b) @ content(b)

where "content(b)" is the sequence of objects in the queue associated with buffer b and "@" is the Gypsy operator that appends sequences.

The Gypsy buffers also have two additional pre-defined boolean valued functions: "full(b)" and "empty(b)". These are defined in terms of "content(b)" as follows:

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

where N is the declared maximal size of buffer b. The language requires that N > 0 for all buffers so that the relation

not [full(b) and empty(b)]

is satisfied for all buffers. Another relation that is important in verification follows from the ones given above

empty(b) iff allto(b) = allfrom(b)

The global histories are the fundamental history concept in Gypsy; however, specifications associated with a particular system or subsystem are normally stated in terms of "local" histories. A local history records the transactions of one particular procedure activation on one particular buffer, whereas a global history records the transactions of all procedure activations on the buffer. The local histories are represented by the pre-defined functions "outto(b,p)" and "infrom(b,p)", where b is a buffer and p is an identifier of a particular activation of some procedure. These local histories are related to the global histories by

outto(b,p) sub allto(b) for all p,
infrom(b,p) sub allfrom(b) for all p
where "sub" is the Gypsy subsequence operator. The global "allto(b)"
history consists of a merge of the local "outto(b,p)" histories for
all procedure activations p that send objects to buffer b. Similarly,
allfrom is a merge of infrom histories.

Systems of concurrent processes often are intentionally
programmed not to terminate, as in the multiplexor example. This
renders either a weak or a strong "exit" specification useless because
a strong exit specification includes an implicit specification of
termination and a weak exit is to hold if the procedure terminates.
The Gypsy specification language contains a special "block" assertion
for stating specifications of nonterminating procedures that
manipulate buffers. The "block" specification is to hold whenever a
procedure is completely blocked waiting its turn for exclusive access
to a buffer. In effect, these blockage points define temporary
halting points at which a procedure is sufficiently stable to state
meaningful specifications and perform verifications.

The use of the block specification and the local histories are
illustrated by the PACKETMOVER procedure in the multiplexor.
PACKETMOVER has two parameters: an input buffer LPin and an output
buffer LOut. Because of the input and output restrictions, we know
that PACKETMOVER can be blocked only if "empty(LPin)" or
"full(LOut)". The "block" specification says that if PACKETMOVER
blocks and LOut is not full, then LPin is empty and everything that
was received from LPin has been sent to LOut. The symbol "myid" is a
Gypsy identifier that is pre-defined for every procedure and refers to
a fixed, but arbitrary, activation of the procedure. (Potentially
there may be several activations of a procedure running concurrently.)
"Infom(LPin, myid)" refers to the sequence of "Labelledpackets"
received from LPin by a particular activation of PACKETMOVER.

The potential blockage points of PACKETMOVER are its "send" and
"receive" statements. The block specification is easily proved at
these points from the internal "assert" specification. The assert
specification is proved in the usual way by inductive assertions. In
constructing these proofs, the semantic effects of the "receive" and
"send" statements are described in terms of their effects on the local
histories appending LP to "infom(LPin, myid)" and "outto(LOut,
myid)" , respectively. In addition to the normal inductive assertion
proof, the block specification is proved at each potential blockage
point. At the "receive" blockage, we know that "full(LOut)". This
contradicts the hypothesis of the block specification, and therefore,
the specification holds trivially. At the "send" blockage, we know
"empty(LPin)" and the rest of the block specification follows from the
"assert" specification.

The use of the local histories is what allows one procedure to be
specified and proved independently of any other procedure. Even
though a buffer may be shared among a number of procedure activations,
the effects on a local history are strictly local to each particular
activation and can be treated independently. When procedures are
activated concurrently under a cobegin, each activation performs its
local effects on its local history. The local histories, which
describe the local effects of each activation on a particular buffer,
are then merged to define the net effect of the "cobegin" on the
buffer. This merge is done in the proof of the "cobegin" and is completely independent of the proof of the called procedures. The verification of the "cobegin" depends solely on the specifications of the procedures it invokes concurrently in a manner that is completely analogous to sequential procedure calls. A procedure call in Gypsy is equivalent to a "cobegin" of a single procedure.

4.0 MULTIPLEXOR EXAMPLE

The multiplexor example given below illustrates the main features of concurrent programming and specification in Gypsy. The example is an implementation of a software multiplexed communications channel. The channel connects two processors, and supports logical one-way connections between each of N processes on the source side with a single corresponding process on the destination side. The i-th process on the source side sends a stream of fixed length packets to the i-th process on the destination machine. The top level view is

```
-----|   |-----
   |
-----| MUX |-----
   |
-----|   |-----
```

MUX accepts packets from each of the input buffers, and delivers the packets to the corresponding output buffer. Each packet is a sequence of characters.

```plaintext
type packet =
    sequence (packetsize) of character;

const packetsize:integer = pending;
```

The types inbufarray and outbufarray have already been defined, so we may now specify the top level of MUX.

```plaintext
procedure MUX(var muxinput:inbufarray;
               var muxoutput:outbufarray) =

begin
    block
        nonefull(muxoutput)
    -> all i:[1..n],
        infrom(muxinput[i],myid) =
          outto(muxoutput[i],myid);
        pending;
    end;
```

Only the procedure header and external specifications are provided at this point. The keyword "pending" indicates that the body is to be filled in later.
Procedure MUX takes two parameters, an array of input lines and an array of output lines. These are the only external objects that may be affected by the procedure. The block specification basically asserts that the sequence of packets sent to each output line is the same as the sequence of packets received from the corresponding input line. "Infrom(MUXINPUT[i],myid)" is the sequence of packets received by this invocation of MUX and any subprocesses it invokes. The block assertion is phrased as an implication, a common technique for specifying blockage conditions. The implication forces the basic blockage specification (that "input" equals "output") to hold if we are not waiting to do output. That is, when we are again waiting for input, all received input packets have been properly handled. Stating the specification in this way avoids special handling of the many possible blockage points that may occur while internal processing is only partially complete.

The function nonefull is used only in non-validated specifications, so only its external specifications need be declared.

function nonefull.
   (bufs:outbufarray):boolean =
   begin
      exit (assume result = true iff
            all i:[1..n], not full(bufs[i]);
   end;

MUX decomposes in three subprocesses: MUXIN, which handles the multiple input lines, MUXOUT, which handles the multiple output lines, and PACKETMOVER, which handles the transmission of the serialized packet stream in between. MUXIN accepts packets from each of the input lines, and makes a labelled packet by affixing the source/destination address. These labelled packets are then passed by the PACKETMOVER to MUXOUT, which extracts the label and passes the packet on to the appropriate output line.

-----|-----------------|-----------------|
-----| MUX | PACKET | MUX |
-----| IN | MOVER | OUT |
-----|-----|------|-----|

We can now write the procedure MUX, the first level in a top-down development.

procedure MUX (var muxinput:inbufarray;
               var muxoutput:outbufarray)=
   begin
      block
         nonefull(muxoutput)
         -> all i:[1..n],
            infrom(muxinput[i],myid) =
            outto(muxoutput[i],myid);
VERIFIABLE COMMUNICATIONS PROCESSING IN GYPXY

Multiplexor Example

var LPbuf1, LPbuf2 : LPacketbuf;

cobegin
    MUXIN(muxinput, LPbuf1);
    PACKETMOVER(LPbuf1,LPbuf2);
    MUXOUT(LPbuf2, muxoutput);
end;
end;

type LabelledPacket =
    record ( lbl:[1..n];
              pkt:packet);

type LPacketbuf =
    buffer(bufsize) of LabelledPacket;

The body of MUX declares two local buffers to carry the labelled
packets between the internal subprocesses. It then starts up the
three subprocesses MUXIN, PACKETMOVER, and MUXOUT to run concurrently.
Logically, the packetmover procedure embodies the actual
communications channel being multiplexed, while MUXIN and MUXOUT
handle the multiplexing and demultiplexing on either end.

Procedure MUXIN accepts incoming packets from the N input
buffers, labels each packet according to its source/destination line,
and passes the labelled packets along in a single merged sequence.

procedure MUXIN (var muxinput:inbufarray;
                 var LPbuf:LPacketbuf)=
begin
    block
        not full(LPbuf)
    -> all i:[1..n],
        LabelledFor(outto(LPbuf,myid),i) = infrom(muxinput[i],myid);

    var p:packet;
    var LP:labelledpacket;

    loop
        assert
            all i:[1..n],
            LabelledFor(outto(LPbuf,myid),i) = infrom(muxinput[i],myid);
        await
            each i:[1..n],
            on receive p from muxinput[i]
            then LP.PKT := p;
            LP.LBL := i;
        end { of await }; 
        send LP to LPbuf;
    end { of loop } 
end { of MUXIN };

function Labelledfor
    (s:LPacketseq;
i:integer[1..n] : LPacketseq =
begin
  exit result = if size(s) then s
  else if first(s).lbl=i then
    first(s) => labelledfor(nonfirst(s),i)
    else labelledfor(nofirst(s),i)
  { This function is used for
    specifications only. }
  fi fi;
end;

type LPacketSeq =
  sequence of LabelledPacket;

The await statement is used to simultaneously wait for input from any of the N input lines. This await statement will block if all the specified input buffers are empty. The procedure may potentially block either at the await statement or the send statement. The blockage specification is again phrased as an implication so as to describe the program state when blocked at the await statement. It asserts that each message is properly labelled and the order of message from each line is preserved. The exact order of the interleaving of messages from different lines is not specified.

The function LabelledFor is used in specifications only. It extracts those labelledpackets destined for output line i. The notation "e :> S" adds the element "e" to the beginning of sequence "S".

Procedure MUXOUT is the inverse of MUXIN. It takes the labelledpackets, examines the label, and resends the unlabelled packet to the appropriate output buffer.

procedure MUXOUT
  (var LPbuf:LPacketbuf<input>;
   var muxoutput:outbufarray) =
begin
  block
    nonefull(muxoutput) -> empty(LPin)
    & all i:[1..n],
      LabelledFor(infrom(LPbuf,myid),i)
      = outto(muxoutput[i],myid);
  var LP:labelledpacket;
  loop
    assert all i:[1..n],
      LabelledFor(infrom(LPbuf,myid),i)
      = outto(muxoutput[i],myid);
    receive LP from LPbuf;
    send LP.PKT to muxoutput[LP.LBL];
  end { of loop };
end { of MUXOUT };
The single remaining piece of MUX is PACKETMOVER. If we already have a channel that logically moves labelled packets, then the realization of packetmover is quite simple.

procedure PACKETMOVER
  (var LPin:LPacketbuf<input>; var LOut:LPacketbuf<output>) =
begin
  block not full(LOut)
    -> empty(LPin)
      & from(LPin,myid) =
      outto(LOut,myid);
  var LP:labelledpacket;
  loop
    assert from(LPin,myid) =
    outto(LOut,myid);
    receive LP from LPin;
    send LP to LOut;
  end;
end;

If the hardware supports packet transmission, then PACKETMOVER can be replaced by a hardware device that satisfies the same external specifications. If the hardware only supports transmission of single characters, we could continue the decomposition of PACKETMOVER into procedures PACKETBREAKER and PACKETMAKER, as mentioned earlier. The external specifications are the same for both decompositions.

5.0 CONCLUSION

Gypsy has a unique set of methods for specifying and proving systems of concurrent processes. These methods support well-structured decomposition, specification, and verification of systems of concurrent processes and are particularly effective for communications processing. A realistic first example communications network resembling the IMP subnet of the ARPANET has been specified, coded, and verified in Gypsy [Wells, 76]. This example involved 1500 lines of specifications and 1000 lines of code, but was carried out only on paper. An incremental verification system for Gypsy is under development [Moriconi, 77] and has been used to verify some small but complete communications systems. The largest example verified by this system to date is a simple message switcher that is slightly over 200 lines of Gypsy text, including specifications. A cross compiler from the PDP-10 to a PDP-11/03 is also under development.
REFERENCES


