

**User-Server Communication in MSG:  
An Example of Abstraction and Refinement**  
*William R. Bush*  
Center for Research in Computing Technology  
Harvard University  
February 1982  
TR-27-81

## 1. Introduction

The construction of large software systems is recognized as an expensive task. At Harvard we have been investigating a programming methodology that substantially reduces the cost of the design, implementation, and maintenance of large systems. The basic technique involves specifying at an *abstract* level an accurate, clear *model* of the desired behavior, and defining *refinements* that are used to *transform* the model into an efficient, executable program. Such an approach is particularly economical when a *family* of programs is being produced, that is, when a single specification is used to produce a number of different running versions (for example, a compiler that will run on, and produce code for, several different machines). The model need only be defined once, and a new version only requires some new refinements. Our methodology is discussed in [Overview], and the tools for controlling and automating the refinement process, without which the methodology would be unmanageable, are discussed in [PDS].

To test our technique and to develop our tools, we have implemented a network interprocess communication facility called MSG. Designed for the National Software Works [NSW], MSG supports datagram<sup>1</sup> and virtual circuit<sup>2</sup> communication between processes on various hosts on the ARPA network. MSG is a serious test for us because it is moderately large and complex, and because it is a program family. We have written an abstract model and have investigated refining it to UNIX and TENEX. General descriptions of MSG, our model, and its refinement can be found in [Model] and [Refinement].

This report documents the modeling and refining of a part of our MSG system. It is a fairly lengthy and detailed example of an application of our methodology. Understanding the part of MSG being modeled and the host-dependent mechanisms used in the refinements is necessary to comprehending this example. Consequently, information about these subjects is also presented.

## 2. The Problem

An instance of MSG runs as a server process on each host offering its services. MSG routes messages between its client user processes and other user processes on other hosts on the network via MSG instances on those hosts. This document is not concerned with the ultimate user-process--user-process communication service provided by MSG, but rather with an aspect of the implementation of that service -- the user-process--server-MSG interprocess communication discipline -- the means by which user processes request service from MSG and by which MSG responds (see Figure 1).

All MSG activity is stimulated by user processes through the invocation of various MSG primitives. (Throughout this document user-server communication is described from the point of view of the user process.) These primitives all have the form of function calls (see [MSG]). A simple example is

```
WhoAmI(ThisProcess);
```

which gets from MSG its name for the user process. The implementation of this primitive call becomes a request sent from the user process to MSG for the "WhoAmI" service and the process name sent from MSG to the user process in response.

- 
1. A datagram is an isolated packet of data (unrelated, from the communication medium's point of view, to other packets) sent from one communicating entity to another, with no guarantee of delivery.
  2. A virtual circuit is a negotiated connection of indefinite duration established between two communicating entities over which a stream of data may be sent.

Not all primitives are so simple. In particular, there is an interesting and important class of primitives that involve interaction with remote user processes. Two such primitives are

```
SendSpecificMessage(SentMessage, DestProcess,  
                   SendSignal, SendDisposition,  
                   SendTimer, SendHandling);
```

and

```
ReceiveSpecificMessage(ReceivedMessage, SourceProcess,  
                      ReceiveSignal, ReceiveDisposition,  
                      ReceiveTimer, ReceivedHandling);
```

which, respectively, send a message to and receive a message from a remote process. The important characteristic of such primitives is that they can take an indefinite amount of time to complete. Requiring the initiating process to wait until contact is made with the remote process via MSG over the network is too restrictive -- the user process must be allowed to continue while the primitive completes. User-server communication is made considerably more complex by this requirement -- one MSG primitive call becomes two incidents of user-server interaction (one on invocation, one on completion). MSG primitives that cause two such incidents are said to create *pending events*.

The major problems that MSG user-server communication behavior raises are, for abstraction,

- modeling the interprocess communication mechanism in a simple, general way, and
- modeling the user-server argument passing protocol, particularly for primitives that create pending events,

and, for refinement,

- refining the interprocess communication mechanism straightforwardly and efficiently.

Solutions to these problems are presented in the remainder of this document.

### 3. The Abstract Model

Our MSG implementation has three basic levels (see Figure 2) -- a functional specification<sup>3</sup> (a high-level definition of MSG behavior), an abstract model (a host-independent program for an idealized machine, derived by transforming the functional specification), and concrete host-dependent implementations (executable programs, produced from the abstract model).

At the level of the functional specification, user-server communication (from the point of view of the user process) simply consists of function calls (the primitives described in the preceding section are examples). Through refinement these calls are expanded into an interprocess *dialog* with MSG. This section describes the abstract model of the user-server dialog, using an idealized interprocess communication mechanism. Further sections describe the refinements that transform the abstract dialog into concrete forms that use real interprocess communication mechanisms.

#### 3.1. The Abstract Interprocess Communication Mechanism

An interprocess communication mechanism is the basic requirement for all user-server interaction. Unlike the implementers of a concrete implementation, who use a predefined mechanism provided by a host operating system, we had to design an idealized abstract mechanism before we could proceed with the abstract model. (Thus it is in general with abstraction and refinement -- identify and design the necessary abstract primitives, design and implement the abstract system, and finally provide concrete realizations of the abstractions.) The design requirements of the abstract mechanism were generality and simplicity. An abstract mechanism that is too specific requires functionality not present in actual ones. A mechanism that is too complicated is difficult to comprehend and anticipates features not available in all actual targets.

Our abstract mechanism is both general and simple. It consists of a virtual circuit, called a *channel*, established by negotiation between entities wishing to communicate, and over which data may be sent and received. Once

---

3. The form of the functional specification is precisely that found in [MSG].

established, a channel can be used to transport arbitrary data. No assumptions are made about the transporting medium, so that in fact we use channels to model server-server network communication as well as user-server interprocess communication.

There are four channel operations: OPEN establishes a channel connection with another entity, SEND sends data, RECEIVE receives data, and CLOSE terminates the connection. Out of these basic operations we construct all user-server communication. The two operations that appear in the refinements discussed here are

```
SEND(data, channel)
```

and

```
RECEIVE(data, channel)
```

The data argument is unrestricted and can be a pointer (in which case the object pointed to is sent or received), or a list of objects (in which case the objects in the list are sent and received in order). The channel argument indicates which open channel is to be used. Both SEND and RECEIVE block until a matching RECEIVE or SEND occurs.

### 3.2. The Abstract Dialog for Simple Primitives

With these basic operations we can implement the user-server dialog. For simple MSG primitives, such as WhoAmI, the dialog is relatively straightforward. Over a channel to MSG, the user process must

- SEND the name of the operation to be performed,
- SEND a list of primitive-dependent arguments, and
- RECEIVE a list of primitive-dependent results.

Each primitive invocation is transformed from a procedure call to such a dialog.

In our methodology transformations are rewriting rules, or *rewrites*, that replace one syntactic pattern with another, the patterns possibly containing pattern-matching variables. Transformation is just one aspect of refinement, which also involves augmenting and replacing abstract data and procedure definitions with concrete ones, but it is the technique we will primarily employ in these examples.

We could, for each MSG primitive, define a rewrite that would refine that primitive from its abstract procedure call form to a series of channel operations. Instead we transform each specific call into a generalized call and then refine all general calls, using a single rewrite, into channel dialogs, thus encapsulating the dialog in one rewrite, so that changes to the dialog involve only that one rewrite.

For example, the rewrite that takes the WhoAmI primitive from its specific to its general form is

```
WhoAmI ($pnam)
  <->
  MSGSimplePrimitive("WhoAmI", < >, < $pnam >);
```

Rewrites are indicated by the <-> operator, with the pattern to be matched preceding the operator, the replacement pattern following the operator, and match variables prefaced by a \$. The arguments to MSGSimplePrimitive in the replacement pattern are the name of the particular MSG primitive to be performed, the list of arguments to be sent to MSG (which is empty in the WhoAmI case), and the list of arguments to be received from MSG.<sup>4</sup> The result of transforming an instance of a WhoAmI call such as

```
WhoAmI(ThisProcess);
```

is thus

```
MSGSimplePrimitive("WhoAmI", < >, < ThisProcess >);
```

MSGSimplePrimitive is then transformed into the necessary channel dialog. The single general dialog rewrite is

---

4. The language of these examples is EL1 [ECL], in which < and > delimit lists of objects.

```

MSGSimplePrimitive($Operation, < ?SendList >, < ?ReceiveList >)
  <->
  BEGIN
    SEND(< $Operation, ?SendList >, MSGChannel);
    RECEIVE(< ?ReceiveList >, MSGChannel);
  END;

```

*Exemplar 1: Simple Dialog*

The ? is similar to \$ -- a \$ variable matches exactly one item, while a ? variable matches any number (including zero).

The result of applying the general dialog rewrite to the above instance of the generalized WhoAmI call is

```

BEGIN
  SEND(< "WhoAmI" >, MSGChannel);
  RECEIVE(< ThisProcess >, MSGChannel);
END;

```

*Exemplar 2: Abstract WhoAmI*

### 3.3. The Abstract Dialog for Pending Event Primitives

The user-server dialog for primitives that create pending events, involving two separate incidents of interaction, is considerably more complex than that for simple primitives. Such primitives cannot simply be refined into channel operations. There are four distinct aspects of pending event primitive refinement:

- the transformation of specific primitive calls into one general form (much the same as that for simple primitives);
- the transformation of the general call into the first interaction with MSG -- the starting of the primitive (this transformation is similar to the dialog rewrite for simple primitives);
- the specification of the general routine that handles the second interaction with MSG -- the completion of the primitive; and
- the specification of the data structure necessary to manage the pending event from start to completion.

The basic goal of these refinements is to define reasonably completely the pending event user-server communication discipline without obscuring it with implementation details and operating system peculiarities.

### 3.4. Specific Primitive Call Transformation

The transformation of specific primitive calls to a general form is very similar for simple and pending event primitives, and the motivation for doing so is the same -- isolate the complexity of the dialog (for pending event primitives, the start of the dialog) in one rewrite. The motivation is even stronger in the case of pending event primitives, where the dialog is more complicated.

Examples of the specific-to-general-form rewrites for a pair of primitives that send and receive messages are

```

SendSpecificMessage($msgarea, $pnam, $signal, $disp, $dt, $sphndl)
  <->
  MSGPendingEventPrimitive("SendSpecificMessage",
    < $msgarea, $pnam, $dt, $sphndl >,
    < >,
    $disp, $signal);

```

and

```

ReceiveSpecificMessage($msgarea, $srcnam, $signal, $disp, $dt, $sphndl)
  <->
  MSGPendingEventPrimitive("ReceiveSpecificMessage",
    < $dt >,
    < $msgarea, $srcnam, $sphndl >,
    $disp, $signal);

```

*Exemplar 3: Specific Receive*

The first three arguments to `MSGPendingEventPrimitive` are the same as the arguments to `MSGSimplePrimitive` (the primitive name, the list of arguments to be sent, and the list of arguments to be received). The two additional arguments are the disposition of the transaction, set during the dialog and indicating its success or failure, and the signal, used in managing the primitive. In the above examples, for `SendSpecificMessage` the send list consists of: `$msgarea`, the message to be sent; `$pnam`, the name of the destination process; `$dt`, the timeout interval after which, if the message has not been sent, the primitive should be aborted; and `$sphndl`, the special handling the message should be given, if any (used for message sequencing and error recovery). For `ReceiveSpecificMessage`, the send list consists of `$dt`, the timeout interval, and the receive list consists of `$msgarea`, the message, `$srcnam`, the name of the sending process, and `$sphndl`, the message's special handling.

The result of transforming the `ReceiveSpecificMessage` instance

```
ReceiveSpecificMessage(ReceivedMessage, SourceProcess,
                      ReceiveSignal, ReceiveDisposition,
                      ReceiveTimer, ReceivedHandling);
```

is thus

```
MSGPendingEventPrimitive("ReceiveSpecificMessage",
                        < ReceiveTimer >,
                        < ReceivedMessage, SourceProcess,
                          ReceivedHandling >,
                        ReceiveDisposition, ReceiveSignal);
```

*Exemplar 4: General Receive*

### 3.5. Pending Event Management

The complexity of the dialog for pending event primitives is due to the indefinite delay between the start of the primitive and its completion. An option available to the user process when it starts such a primitive is that it block while waiting for the primitive to complete. If this option is chosen the dialog becomes essentially the same as that for simple primitives. But if the user process elects to continue execution while the primitive is being processed by MSG, then information about the transaction must be kept by the user process, and the user process must be prepared to complete the primitive when MSG notifies it that results are at hand. Additional information must also be exchanged by the user process and MSG when the primitive is started, to allow them mutually to manage the transaction. One data structure is used to hold this information and is called a signal.

The signal data structure is defined to be

```
SignalType <-
  Structure(Block: Boolean,
           Notifier: Stimulus,
           ID: PendingEventID,
           Results: Pointer);5
```

The first two fields of `SignalType` are set by the invoker of a primitive. They consist of a boolean, `Block`, indicating whether the process should block until the primitive completes, and a user process stimulus<sup>6</sup>, `Notifier`, with which MSG can notify the user process that the primitive is ready to be completed.<sup>7</sup> The other two fields are filled in at the start of the dialog. They consist of a unique identifier for the transaction, `ID`, supplied by MSG (allowing the user process, for example, to identify and have MSG cancel a particular transaction from among a set of pending ones) and the list of results, `Results`, that are to be received when the primitive completes.

---

5. The abstract data type `Structure` (refined to an EL1 `STRUCT`) defines a record type with named components, and the abstract data type `Pointer` (refined to an EL1 `REF`) is an unrestricted pointer.

6. The type of stimulus is host-dependent. A typical example is a software interrupt channel.

7. According to [MSG] page 2-28 a signal must be: distinguishable, testable, and process-local. Examples given are: block/unblock, completion flag (possibly using the disposition field), software interrupt, and flag plus software interrupt. Our signals are distinguishable via the unique `PendingEventID` and are testable via the disposition field.

At any one time, a number of pending event primitives may be incomplete, requiring that a set of associated signals be maintained. Thus each user process has a

```
PendingEventSet <- Set(SignalType);
```

We need not concern ourselves now (at the level of the abstract model) with how Set is implemented. At some point PendingEventSet will be refined into some concrete representation such as an array or a linked list, but that will occur at a much lower level of refinement (see [Refinement] for a discussion of set refinement). Similarly, we can use the three abstract set operations

```
Insert E In S;  
Remove E From S;  
ForEach E In S Do ... End;
```

without having the details of their implementations intrude on our abstract program, making the set behavior of PendingEventSet explicit and clear.

### 3.6. The First Part of the Dialog

The first part of the actual user-server dialog consists of the starting of the primitive -- the request for service. This involves sending arguments to MSG and having their correctness verified, and performing pending event management (storing various items of information about the pending event). In detail (given a channel between the user process and MSG) the steps are:

- SEND to MSG the name of the operation to be performed;
- SEND to MSG the list of primitive-dependent arguments;
- SEND to MSG the pending event signal;
- RECEIVE from MSG a disposition code indicating whether or not the arguments sent are acceptable;
- if the arguments are unacceptable then exit;
- if the user wants the primitive to block until completion, then RECEIVE from MSG the list of primitive-dependent results and exit;
- save the list of primitive-dependent results to be received upon completion;
- add the pending event to the set of incomplete ones; and
- RECEIVE from MSG the unique identifier for the pending event.

These steps are implemented in the general rewrite of MSGPendingEventPrimitive. The purpose of the rewrite is to embody cleanly the above verbal specification while providing a good basis for later host-dependent refinement.

```

MSGPendingEventPrimitive($Operation,
                        < ?SendList >,
                        < ?ReceiveList >,
                        $Disposition, $Signal)
<->
BEGIN
  SEND(< $Operation, ?SendList, $Signal >, MSGChannel);
  RECEIVE(< $Disposition >, MSGChannel);
  IsArgumentError($Disposition) =>
    LogError($Operation, $Disposition);
  $Signal.Block =>
    BEGIN
      RECEIVE(< ?ReceiveList, $Disposition >, MSGChannel);
    END;
  $Signal.Results <-
    Allocate(Sequence(Pointer) OF
             ?ReceiveList, $Disposition);
  Insert $Signal In PendingEventSet;
  RECEIVE(< $Signal.ID >, MSGChannel);
END;8

```

#### *Exemplar 5: Start Pending Event*

The result of applying this rewrite to the above instance of the generalized ReceiveSpecificMessage call (see *Exemplar 4: General Receive*) is

```

BEGIN
  SEND(< "ReceiveSpecificMessage", ReceiveTimer, ReceiveSignal >,
       MSGChannel);
  RECEIVE(< ReceiveDisposition >, MSGChannel);
  IsArgumentError(ReceiveDisposition) =>
    LogError("ReceiveSpecificMessage", ReceiveDisposition);
  ReceiveSignal.Block =>
    BEGIN
      RECEIVE(< ReceivedMessage, SourceProcess,
              ReceivedHandling, ReceiveDisposition >,
              MSGChannel);
    END;
  ReceiveSignal.Results <-
    Allocate(Sequence(Pointer) OF
             ReceivedMessage, SourceProcess,
             ReceivedHandling, ReceiveDisposition);
  Insert ReceiveSignal In PendingEventSet;
  RECEIVE(< ReceiveSignal.ID >, MSGChannel);9
END;

```

#### *Exemplar 6: Abstract Receive*

---

8. In EL1 the construct  $P \Rightarrow Q$  means "if P is true then evaluate Q and exit the enclosing block". The construct  $\text{Allocate}(T \text{ OF } C1, C2)$  creates an object of type T with components initialized to C1 and C2 and returns a pointer to the object. The abstract data type Sequence (refined to an EL1 SEQ) defines a variable-length array type.

9. MSG will be blocked in the SEND corresponding to this RECEIVE until the RECEIVE is done, thereby keeping MSG from possibly finishing the pending event before the list of results is set up and the pending event has been inserted into the pending event set.

### 3.7. The Second Part of the Dialog

The second part of the user-server dialog is the completion of the pending primitive -- obtaining the results of the requested service. This consists of receiving data from MSG and performing pending event management. In detail the steps are:

- determine which pending event has completed;
- remove the pending event from the set of incomplete ones; and
- receive the arguments in the pending event's receive list.

These steps are embodied in a single completion routine to which control passes whenever MSG indicates that a primitive is ready to be completed.<sup>10</sup>

```
PendingEventCompletion <-  
  EXPR()  
  BEGIN  
    DECL E:SignalType  
      LIKE FindEventToComplete(PendingEventSet);  
    Remove E From PendingEventSet;  
    RECEIVE(E.Results, MSGChannel);11  
  END;
```

#### *Exemplar 7: Complete Pending Event*

The refinement goals are the same here as for the first part of the dialog -- to encode cleanly the verbal specification and to provide a useful foundation for host-dependent refinement. In fact, host-dependent refinement of PendingEventCompletion consists of defining FindEventToComplete and refining RECEIVE.

## 4. The UNIX Implementation

Given the abstract model, the only major refinement needed to produce a host-specific running system is the transformation of the abstract channel mechanism into the interprocess communication mechanism of the target host. Other refinements are necessary, such as those for sets, but they are relatively straightforward, are supplied by a global library, and are applied later in the refinement process.<sup>12</sup>

For UNIX, the channel refinement is straightforward, since the VAX UNIX virtual circuit mechanism [UNIX] is almost identical to our abstract channel mechanism (see Figure 3).<sup>13</sup>

### 4.1. Transforming the Dialog

The basic difference between channels and UNIX virtual circuits is that channel SENDs and RECEIVEs operate on lists of objects, while virtual circuit reads and writes operate on single objects.<sup>14</sup> This difference is resolved (for simple dialogs and the first part of pending event dialogs) by two recursive rewrites that transform SEND and RECEIVE lists into sequences of writes and reads,

---

10. The routine can be thought of as an interrupt handler, although its precise nature is a host-dependent issue.

11. Note that Results always includes the disposition, which indicates the ultimate success or failure of the transaction.

12. Figure 2 is in fact simplified -- the UNIX and TENEX refinements consist of two refinement sets that are applied serially, the first specific (to specific parts of the MSG system, such as user-server channel refinements) and the second global (such as set refinements). Ultimately the system is refined to a restricted subset of EL1 (called SPECL) that can be translated into Ada or compiled directly (we are currently developing a family of SPECL compilers).

13. In this regard, because virtual circuits are fundamentally more general, they are a distinct improvement over Rand ports, our previous target mechanism.

```

SEND(< $Argument, ?Remainder >, MSGChannel)
  <->
  [ |
    write(MSGSocket, $Argument);
    SEND(< ?Remainder >, MSGChannel);
  ];

```

and

```

RECEIVE(< $Argument, ?Remainder >, MSGChannel)
  <->
  [ |
    read(MSGSocket, $Argument);
    RECEIVE(< ?Remainder >, MSGChannel);
  ];

```

and by two rewrites that terminate the recursive process (removing the residual SEND and RECEIVE),

```

SEND(< >, MSGChannel) <-> [ | ];

```

and

```

RECEIVE(< >, MSGChannel) <-> [ | ];

```

MSGSocket is the virtual circuit realization of MSGChannel. The special symbols [ | and ] are not part of the rewrite result, but instead enclose sequences of statements to be inserted in-line (surrounding such sequences with BEGIN-END pairs, in contrast, for example, would cause the BEGIN-END pairs to be inserted as part of the rewrite result).

The result of applying these rewrites to the abstract model level WhoAmI instance (see *Exemplar 2: Abstract WhoAmI*) is

```

BEGIN
  write(MSGSocket, "WhoAmI");
  read(MSGSocket, ThisProcess);
END;

```

and to the ReceiveSpecificMessage instance (see *Exemplar 6: Abstract Receive*) is

---

14. Channels and virtual circuits also differ in three other ways that pose low-level problems -- read and write 1) take a third (length) argument, 2) do not handle pointer objects, and 3) do not automatically handle variable-length objects. The first problem is solved by supplying the third argument, for example *read(MSGSocket, obj, LENGTH(obj))*. The second is solved by dereferencing pointer objects, for example *write(MSG, VAL(obj), LENGTH(VAL(obj)))*. The third, only a problem with read, is solved by using "recordmode" on the virtual circuit and implementing a variable-length-read function that constructs variable-length objects.

```

BEGIN
  write(MSGSocket, "ReceiveSpecificMessage");
  write(MSGSocket, ReceiveTimer);
  write(MSGSocket, ReceiveSignal);
  read(MSGSocket, ReceiveDisposition);
  IsArgumentError(ReceiveDisposition) =>
    LogError("ReceiveSpecificMessage", ReceiveDisposition);
  ReceiveSignal.Block =>
    BEGIN
      read(MSGSocket, ReceivedMessage);
      read(MSGSocket, SourceProcess);
      read(MSGSocket, ReceivedHandling);
      read(MSGSocket, ReceiveDisposition);
    END;
  ReceiveSignal.Results <-
    Allocate(Sequence(Pointer) OF
      ReceivedMessage, SourceProcess,
      ReceivedHandling, ReceiveDisposition);
  Insert ReceiveSignal In PendingEventSet;
  read(MSGSocket, ReceiveSignal.ID);
END;

```

## 4.2. Pending Event Completion

The refinement of pending event completion for UNIX is relatively simple (because of the nature of virtual circuits) and has three aspects.

First, the routine `PendingEventCompletion` (see *Exemplar 7: Complete Pending Event*) must be invoked whenever an event is to be completed. Since a virtual circuit can be made "asynchronous", so that activity on it causes an interrupt, an asynchronous virtual circuit is set up to effect completion (a distinct virtual circuit, since only completion activity should cause an interrupt), and `PendingEventCompletion` is made the interrupt handler for that virtual circuit.

Second, `PendingEventCompletion` must determine which pending event is ready to be completed (`FindEventToComplete` must be elaborated). This is done by having `MSG` send, as the first completion item, the `PendingEventID` of the event to be completed, and then having `FindEventToComplete` search `PendingEventSet` for the proper event.<sup>15</sup> In particular, `FindEventToComplete` is refined into

```

FindEventToComplete(PendingEventSet)
  <->
  BEGIN
    DECL CompleteID:PendingEventID;
    read(MSGCompletionSocket, CompleteID);
    ForEach E In PendingEventSet
      Do E.ID = CompleteID => E End;
  END;

```

which returns the event to be completed.

Third, the `RECEIVE`, in `PendingEventCompletion`, of the `Results` must be made to work on a virtual circuit. Here, as for the first part of the dialog, a `RECEIVE` of a list (or sequence) of objects must be converted into a sequence of reads. The `RECEIVE` thus becomes

---

15. An alternate approach would be to have one virtual circuit for completion per event. This approach has two disadvantages -- the overhead of opening and closing a virtual circuit for every transaction, and the problem of the restriction of one asynchronous socket per process (which could be circumvented by having one special permanent asynchronous socket in addition to the many transitory per-transaction synchronous virtual circuits)

```

RECEIVE(E.Results, MSGChannel)
  <->
  FOR I TO LENGTH(E.Results)
    REPEAT
      read(MSGCompletionSocket, E.Results[I]);16
    END;

```

## 5. The TENEX Implementation

Channel refinement for TENEX is considerably more complex than for UNIX, because TENEX does not have a close analog to the channel mechanism. Instead, TENEX processes communicate via shared memory [TENEX-System] (see Figure 4).

A possible refinement approach would be actually to implement, using shared memory, the abstract channel primitives SEND and RECEIVE, which would simply copy their arguments to and from a shared page. The abstract model would not be affected. The objection to this approach is inefficiency. Each matched SEND-RECEIVE pair would necessitate freezing the sending process and awakening the receiving one.

Fortunately, it happens that numerous SENDs and RECEIVEs can be collapsed into a single transfer of shared memory data. The cost of this approach, however, is substantial divergence from the abstract model dialog. We have chosen, despite the cost, to pursue this optimization (it follows closely the user-server discipline of the MACRO-10 MSG implementation [TENEX-MSG]). Currently, with our methodology such optimizations must be performed and verified by hand, but it is one of our goals to have tools that will aid in the optimization and verification process.

### 5.1. Transforming the Dialog

MSG primitive arguments are passed between a user process and MSG via argument blocks placed in shared memory. Each primitive call has its own type of argument block, a record data structure containing as its fields the individual arguments to the primitive. For example, the WhoAmI primitive uses the argument block type

```

WhoAmIArgumentBlockType <-
  Structure(pnam:ProcessName);

```

and the ReceiveSpecificMessage primitive (the arguments of which may be seen in *Exemplar 3: Specific Receive*) the argument block type<sup>17</sup>

```

ReceiveSpecificMessageArgumentBlockType <-
  Structure(dt:Interval,
           msgarea:MessagePointer,18
           srcnam:ProcessNamePointer,
           sphndl:HandlingCodePointer,
           signal:SignalPointer);

```

Each primitive call instance creates its own copy of the proper argument block and assigns to each field of the block the appropriate primitive argument.

After the user process has set up the argument block for a primitive, it must notify MSG that it wants the primitive performed. Due to the nature of TENEX shared memory and interprocess interrupts, MSG resides in the same process tree as its client user processes, running as a process superior to its clients. This enables the user process to interrupt MSG and vice versa, and allows MSG to freeze the user process and read and write its memory. Thus when the user process desires service it causes a software interrupt in MSG, passing a pointer to the relevant argument block. MSG then freezes the user process, uses the argument block pointer to get arguments from and put arguments into the argument block, and, when done, thaws the user process. (For a pending event primitive, if Signal is blocking then MSG will keep the process frozen until the pending event completes.)

16. See note 14.

17. The astute reader will notice that the disposition argument to ReceiveSpecificMessage is not in its argument block. It is passed in the Results field of ReceiveSignal.

18. The primitive call results (the ones that will be filled in when the primitive completes) must be pointer objects, so that they will be available outside the scope in which the argument block is declared.

For simple dialogs, the rewrite that sets up the proper argument block and causes an interrupt in MSG is

```
BEGIN ?Body END DerivedFrom
  MSGSimplePrimitive($Operation, < ?SendList >, < ?ReceiveList >);
  <->
BEGIN
  DECL Argument:Pointer LIKE
    Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
              OF ?SendList, ?ReceiveList);
  InvokeMSG(Argument);
END;
```

This rewrite effectively replaces the abstract model dialog (see *Exemplar 1: Simple Dialog*) with a TENEX-specific dialog. Central to this result is the DerivedFrom pattern qualifier, which causes the transformation history of relevant blocks to be examined. It serves two functions. It guarantees that the rewrite will only be applied to blocks with the correct ancestry (in this case, abstract model simple primitive dialog blocks), and it supplies the match variables from the earlier transformations (here, the transformations of the generalized calls).

The !! prefix operator is not part of the rewrite result but rather causes its argument to be evaluated at refinement time. Thus GetArgumentBlockTypeFromOpCode produces, at refinement time, based on Operation, the proper argument block type.

The result of applying the simple dialog rewrite to the abstract model level WhoAmI instance (see *Exemplar 2: Abstract WhoAmI*) is

```
BEGIN
  DECL Argument:Pointer LIKE
    Allocate(WhoAmIArgumentBlockType OF ThisProcess);
  InvokeMSG(Argument);
END;
```

The pending event dialog rewrite is more complex, involving pending event management. This rewrite, heavily optimized for TENEX, is derived from the abstract model dialog (see *Exemplar 5: Start Pending Event*). The steps in the derivation are presented in an appendix.

```

BEGIN ?Body END DerivedFrom
  MSGPendingEventPrimitive($Operation,
    < ?SendList >, < ?ReceiveList >,
    $Disposition, $Signal)
  <->
BEGIN
  DECL Argument:Pointer LIKE
    Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
      OF ?SendList, ?ReceiveList, $Signal);
  $Signal.Results <- $Disposition;
  $Signal.Block =>
    BEGIN
      InvokeMSG(Argument);
      IsArgumentError($Disposition) =>
        LogError($Operation, $Disposition);
    END;
  Insert $Signal In PendingEventSet;
  InvokeMSG(Argument);
  IsArgumentError($Disposition) =>
    BEGIN
      Remove $Signal From PendingEventSet;19
      LogError($Operation, $Disposition);
    END;
END;

```

The result of applying this rewrite to the ReceiveSpecificMessage instance (see *Exemplar 6: Abstract Receive*) is

```

BEGIN
  DECL Argument:Pointer LIKE
    Allocate(ReceiveSpecificMessageArgumentBlockType OF
      ReceiveTimer, ReceivedMessage, SourceProcess,
      ReceivedHandling, ReceiveSignal);
  ReceiveSignal.Results <- ReceiveDisposition;
  ReceiveSignal.Block =>
    BEGIN
      InvokeMSG(Argument);
      IsArgumentError(ReceiveDisposition) =>
        LogError("ReceiveSpecificMessage", ReceiveDisposition);
    END;
  Insert ReceiveSignal In PendingEventSet;
  InvokeMSG(Argument);
  IsArgumentError(ReceiveDisposition) =>
    BEGIN
      Remove ReceiveSignal From PendingEventSet;
      LogError("ReceiveSpecificMessage", ReceiveDisposition);
    END;
END;

```

## 5.2. Pending Event Completion

The nature of TENEX pending event completion refinement is largely determined by the asymmetric superior-inferior relationship between MSG and the user process, and by the memory sharing method of communication.

---

19. The Remove is necessary because of the Insert preceding the InvokeMSG. The Insert must precede the InvokeMSG so that the PendingEventCompletion interrupt routine will find the event in PendingEventSet in the case of immediate completion. If there is an argument error, PendingEventCompletion will not be activated.

MSG's natural control over completion (it always initiates completion, just as the user process always initiates requests for service) is manifest in the refinements.

In particular, MSG freezes the user process, puts results in the argument block, and causes a software interrupt in the user process. The PendingEventCompletion routine (see *Exemplar 7: Complete Pending Event*) is the interrupt handler on the software interrupt channel stimulated by MSG. The user process takes no part in data movement. The RECEIVE in PendingEventCompletion is meaningless and is therefore removed.

FindEventToComplete must be elaborated, but, not needing to do a RECEIVE, it is simple. It must only scan PendingEventSet for the event that MSG has completed -- for the one with a "Complete" disposition.<sup>20</sup> FindEventToComplete becomes

```
FindEventToComplete(PendingEventSet)
  <->
  ForEach E In PendingEventSet
    Do IsComplete(E.Results) => E End;
```

## 6. Observations<sup>21</sup>

The user part of the user-process--server-MSG interprocess communication discipline is only one small piece of the MSG system. The purpose of this document is to give a detailed example of our abstraction and refinement methodology, not to document our MSG implementation. Nor is its purpose to survey applications of our methodology. The level of abstraction of this example is basically that of host independence. Nothing in our methodology is any more or less suited to a host-independent level of abstraction than to any other level. Indeed, it is natural for a system to have many levels of abstraction. In MSG, for instance, set and queue refinement occur after host-dependent refinement; with more powerful tools, one can imagine an abstract model above the level of the abstract model presented here.

Our MSG implementation is not a trivial system. The abstract model consists of seven basic modules (one of global definitions) containing in total about 5000 lines of EL1 code. The refinements for our prototype system (which runs under the ECL interpreter) consist of twelve modules (three of global definitions) containing about 2000 lines of code. Our methodology provided us with a quite natural means of compartmentalizing, both vertically (with levels of abstraction) and horizontally (with modules), the development of the system. Crucial to the methodology's success are proper tools for supporting the refinement process -- tools for automatically producing a concrete system, for incrementally rederiving part of the system (allowing rapid debugging), and for aiding refinement development with appropriate analysis.

Our methodology does not automatically result in good programs. Since it is flexible and powerful, it can be used to produce obscure, inefficient, and erroneous programs. On the other hand, it makes it easier to write clear, efficient, and easily maintained programs.

## 7. References

- [ECL] *ECL Programmer's Manual*. Technical report TR-23-74, Center for Research in Computing Technology, Harvard University, Cambridge MA, December 1974.
- [Model] *Abstract Model of MSG*. Technical report TR-25-78, Center for Research in Computing Technology, Harvard University, Cambridge MA, October 1978.
- [MSG] 'MSG Design Specification', in *Third Semi-Annual Technical Report for the National Software Works*. Technical Report CADD-7702-2811, Massachusetts Computer Associates, Wakefield MA, February 1977
- [NSW] *Provide a National Software Works (NSW) Framework and Machine Connections*, Technical Report CADD-7602-1711, Massachusetts Computer Associates, Wakefield MA, February 1976.

---

20. Note that since no other results are received, Results need only contain the disposition, the only datum necessary for completion.

21. The Goal Of Refinement: Plus c,a change, plus c'est la me^me chose.

- [Overview] *An Overview of the Harvard PDS Project* Memorandum, Center for Research in Computing Technology, Harvard University, Cambridge MA, October 1981.
- [PDS] *PDS User's Manual*. Memorandum, Center for Research in Computing Technology, Harvard University, Cambridge MA, February 1982.
- [Refinement] *Refinement of an Abstract Model of MSG*. Technical report TR-06-80, Center for Research in Computing Technology, Harvard University, Cambridge MA, April 1980.
- [TENEX-MSG] *TENEX MSG User Manual*. Report No. 3540, Bolt Beranek and Newman, Cambridge MA, April 1977.
- [TENEX-System] *TENEX, a Paged Time Sharing System for the PDP-10*. Communications of the ACM, March 1972, page 135.
- [UNIX] *Proposals for enhancement of UNIX on the VAX*. Computer Systems Research Group TR/4, Department of Electrical Engineering and Computer Science, University of California, Berkeley CA, August 1981.

## Appendix: The TENEX Optimization

This appendix presents the sequence of transformations necessary to produce an optimized pending event dialog for TENEX (given the TENEX shared memory interprocess communication mechanism) from the abstract model dialog. Currently we must perform and verify such transformations by hand, but we are planning to produce tools that will aid this process. Note that the object we are transforming is itself a transformation. Note also that some of the steps are accompanied by corresponding changes in MSG.

### Step One

The pending event dialog rewrite for the abstract model (see *Exemplar 5: Start Pending Event*) is our starting point.

```
MSGPendingEventPrimitive($Operation,
                        < ?SendList >,
                        < ?ReceiveList >,
                        $Disposition, $Signal)
    <->
BEGIN
<1a>     SEND(< $Operation, ?SendList, $Signal >, MSGChannel);
<1b>     RECEIVE(< $Disposition >, MSGChannel);
<1c>     IsArgumentError($Disposition) =>
            LogError($Operation, $Disposition);
<1d>     $$Signal.Block =>
            BEGIN
                RECEIVE(< ?ReceiveList, $Disposition >, MSGChannel);
            END;
    $$Signal.Results <-
        Allocate(Sequence(Pointer) OF ?ReceiveList, $Disposition);
    Insert $$Signal In PendingEventSet;
    RECEIVE(< $$Signal.ID >, MSGChannel);
END;
```

## Step Two

Move <1d> to the beginning of the block, duplicating <1a-c> on both branches of the conditional.  
Preconditions: 1) <1d> must not depend on <1a-c>; 2) <1d>, when moved, must not affect <1a-c>. Met: 1) only <1b> could have any effect (<1a> has no relevant side effects), and it only modifies \$Disposition, which is independent of \$Signal; 2) manifestly true.

```
BEGIN
  $Signal.Block =>
    BEGIN
      SEND(< $Operation, ?SendList, $Signal >, MSGChannel);
      RECEIVE(< $Disposition >, MSGChannel);
      IsArgumentError($Disposition) =>
        LogError($Operation, $Disposition);
      RECEIVE(< ?ReceiveList, $Disposition >, MSGChannel);
    END;
  SEND(< $Operation, ?SendList, $Signal >, MSGChannel);
  RECEIVE(< $Disposition >, MSGChannel);
<2a>  IsArgumentError($Disposition) =>
      LogError($Operation, $Disposition);
<2b>  $Signal.Results <-
      Allocate(Sequence(Pointer) OF ?ReceiveList, $Disposition);
<2c>  Insert $Signal In PendingEventSet;
      RECEIVE(< $Signal.ID >, MSGChannel);
END;
```

## Step Three

Move <2b-c> above <2a>, inserting their inverses before LogError in <2a>. The inverse of <2b> can be ignored since the value of Results is irrelevant after <2b>. The inverse of <2c> is a Remove. Preconditions: 1) <2b-c> must not depend on <2a>; 2) <2b-c>, when moved, must not affect <2a>. Met: 1) <2a> has no relevant side effects; 2) manifestly true.

```
BEGIN
  $Signal.Block =>
    BEGIN
      SEND(< $Operation, ?SendList, $Signal >, MSGChannel);
      RECEIVE(< $Disposition >, MSGChannel);
      IsArgumentError($Disposition) =>
        LogError($Operation, $Disposition);
      RECEIVE(< ?ReceiveList, $Disposition >, MSGChannel);
    END;
  SEND(< $Operation, ?SendList, $Signal >, MSGChannel);
<3a>  RECEIVE(< $Disposition >, MSGChannel);
<3b>  $Signal.Results <-
      Allocate(Sequence(Pointer) OF ?ReceiveList, $Disposition);
<3c>  Insert $Signal In PendingEventSet;
      IsArgumentError($Disposition) =>
        BEGIN
          Remove $Signal From PendingEventSet;
          LogError($Operation, $Disposition);
        END;
  RECEIVE(< $Signal.ID >, MSGChannel);
END;
```

### Step Four

Move <3c-d> (moved in Step Two) before <3a-b>. Preconditions: 1) <3c-d> must not depend on <3a-b>; 2) <3c-d>, when moved, must not affect <3a-b>. Met: 1) manifestly true; 2) \$Signal is sent in <3a> and modified in <3c>, but the Results field of \$Signal is used only locally, not by MSG (this observation is based on a knowledge of MSG).

```
BEGIN
    $Signal.Block =>
        BEGIN
            <4a>         SEND(< $Operation, ?SendList, $Signal >, MSGChannel);
            <4b>         RECEIVE(< $Disposition >, MSGChannel);
            <4c>         IsArgumentError($Disposition) =>
                LogError($Operation, $Disposition);
            <4d>         RECEIVE(< ?ReceiveList, $Disposition >, MSGChannel);
        END;
    $Signal.Results <-
        Allocate(Sequence(Pointer) OF ?ReceiveList, $Disposition);
    Insert $Signal In PendingEventSet;
    <4e>         SEND(< $Operation, ?SendList, $Signal >, MSGChannel);
    <4f>         RECEIVE(< $Disposition >, MSGChannel);
    <4g>         IsArgumentError($Disposition) =>
        BEGIN
            Remove $Signal From PendingEventSet;
            LogError($Operation, $Disposition);
        END;
    <4h>         RECEIVE(< $Signal.ID >, MSGChannel);
END;
```

## Step Five

Convert <4a-d> and <4e-h> into the common form

```
BEGIN
  SEND(...);
  RECEIVE(...);
  IsArgumentError(...) => NOTHING;
  RECEIVE(...);
END;
IsArgumentError(...) => <expression>;
```

Preconditions: 1) IsArgumentError must be repeatable; 2) <4d> and <4h>, in the common form, must not affect <4c> and <4g>. Met: 1) IsArgumentError has no relevant side effects; 2) the \$Disposition received in <4d> will not cause an argument error (this observation is based on a knowledge of MSG).

```
BEGIN
  $Signal.Block =>
    BEGIN
      BEGIN
        SEND(< $Operation, ?SendList, $Signal >, MSGChannel);
        RECEIVE(< $Disposition >, MSGChannel);
        IsArgumentError($Disposition) => NOTHING;
<5a>      RECEIVE(< ?ReceiveList, $Disposition >, MSGChannel);
        END;
        IsArgumentError($Disposition) =>
          LogError($Operation, $Disposition);
      END;
  $Signal.Results <-
    Allocate(Sequence(Pointer) OF ?ReceiveList, $Disposition);
    Insert $Signal In PendingEventSet;
    BEGIN
      SEND(< $Operation, ?SendList, $Signal >, MSGChannel);
      RECEIVE(< $Disposition >, MSGChannel);
      IsArgumentError($Disposition) => NOTHING;
<5b>      RECEIVE(< $Signal.ID >, MSGChannel);
    END;
    IsArgumentError($Disposition) =>
      BEGIN
        Remove $Signal From PendingEventSet;
        LogError($Operation, $Disposition);
      END;
END;
```

## Step Six

Convert <5a> and <5b> into the common form

```
$Signal.Block =>  
    RECEIVE(< ?ReceiveList, $Disposition >, MSGChannel);  
RECEIVE(< $Signal.ID >, MSGChannel);
```

Precondition: the preceding statements in the block must not affect the \$Signal.Block in the common form.  
Met: \$Signal is unmodified (SEND and Insert have no relevant side effects).

```
BEGIN  
    $Signal.Block =>  
        BEGIN  
<6a>            BEGIN  
                SEND(< $Operation, ?SendList, $Signal >, MSGChannel);  
                RECEIVE(< $Disposition >, MSGChannel);  
                IsArgumentError($Disposition) => NOTHING;  
                $Signal.Block =>  
                RECEIVE(< ?ReceiveList, $Disposition >, MSGChannel);  
                RECEIVE(< $Signal.ID >, MSGChannel);  
            END;  
            IsArgumentError($Disposition) =>  
                LogError($Operation, $Disposition);  
        END;  
    $Signal.Results <-  
        Allocate(Sequence(Pointer) OF ?ReceiveList, $Disposition);  
    Insert $Signal In PendingEventSet;  
<6b>    BEGIN  
        SEND(< $Operation, ?SendList, $Signal >, MSGChannel);  
        RECEIVE(< $Disposition >, MSGChannel);  
        IsArgumentError($Disposition) => NOTHING;  
        $Signal.Block =>  
            RECEIVE(< ?ReceiveList, $Disposition >, MSGChannel);  
            RECEIVE(< $Signal.ID >, MSGChannel);  
    END;  
    IsArgumentError($Disposition) =>  
        BEGIN  
            Remove $Signal From PendingEventSet;  
            LogError($Operation, $Disposition);  
        END;  
END;
```

## Step Seven

Replace <6a> and <6b> with TENEX argument block creation and MSG invocation.

```
BEGIN
  $Signal.Block =>
    BEGIN
      DECL Argument:Pointer LIKE
      Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
              OF ?SendList, ?ReceiveList, $Signal, $Disposition);
      InvokeMSG(Argument);
      IsArgumentError($Disposition) =>
        LogError($Operation, $Disposition);
    END;
<7a> $Signal.Results <-
      Allocate(Sequence(Pointer) OF ?ReceiveList, $Disposition);
<7b> Insert $Signal In PendingEventSet;
<7c> DECL Argument:Pointer LIKE
      Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
              OF ?SendList, ?ReceiveList, $Signal, $Disposition);
      InvokeMSG(Argument);
      IsArgumentError($Disposition) =>
        BEGIN
          Remove $Signal From PendingEventSet;
          LogError($Operation, $Disposition);
        END;
    END;
END;
```

## Step Eight

Move <7c> above <7a-b>. Preconditions: 1) <7c> must not depend on <7a-b>; 2) <7c>, when moved, must not affect <7a-b>. Met: 1) \$Signal is a pointer (automatically dereferenced on assignment to the record structure it points to) -- the free \$Signal and the \$Signal in Argument both point to the same object, so the assignment in <7a> is reflected by both \$Signals; 2) manifestly true.

```
BEGIN
  $Signal.Block =>
    BEGIN
      DECL Argument:Pointer LIKE
      Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
              OF ?SendList, ?ReceiveList, $Signal, $Disposition);
      InvokeMSG(Argument);
      IsArgumentError($Disposition) =>
        LogError($Operation, $Disposition);
    END;
  DECL Argument:Pointer LIKE
  Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
          OF ?SendList, ?ReceiveList, $Signal, $Disposition);
<8> $Signal.Results <-
  Allocate(Sequence(Pointer) OF ?ReceiveList, $Disposition);
  Insert $Signal In PendingEventSet;
  InvokeMSG(Argument);
  IsArgumentError($Disposition) =>
    BEGIN
      Remove $Signal From PendingEventSet;
      LogError($Operation, $Disposition);
    END;
END;
```

## Step Nine

Observe that the only Results item for TENEX is \$Disposition, and modify <8> accordingly. Precondition: ?ReceiveList must not be referenced through \$Signal. Met: it is not referenced either here or in PendingEventCompletion.

```
BEGIN
  $Signal.Block =>
  BEGIN
    <9a>      DECL Argument:Pointer LIKE
             Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
                       OF ?SendList, ?ReceiveList, $Signal, $Disposition);
             InvokeMSG(Argument);
             IsArgumentError($Disposition) =>
               LogError($Operation, $Disposition);
  END;
  DECL Argument:Pointer LIKE
  Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
           OF ?SendList, ?ReceiveList, $Signal, $Disposition);
  <9b>      $Signal.Results <- $Disposition;
  Insert $Signal In PendingEventSet;
  InvokeMSG(Argument);
  IsArgumentError($Disposition) =>
    BEGIN
      Remove $Signal From PendingEventSet;
      LogError($Operation, $Disposition);
    END;
  END;
END;
```

## Step Ten

Duplicate <9b> after <9a>. Precondition: Results must not be referenced in the remainder of the block. Met: Results is not used by MSG in InvokeMSG (this observation is based on a knowledge of MSG).

```
BEGIN
  $Signal.Block =>
  BEGIN
    <10a>     DECL Argument:Pointer LIKE
             Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
                       OF ?SendList, ?ReceiveList, $Signal, $Disposition);
    <10b>     $Signal.Results <- $Disposition;
             InvokeMSG(Argument);
             IsArgumentError($Disposition) =>
               LogError($Operation, $Disposition);
  END;
  <10c>     DECL Argument:Pointer LIKE
             Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
                       OF ?SendList, ?ReceiveList, $Signal, $Disposition);
  <10d>     $Signal.Results <- $Disposition;
  Insert $Signal In PendingEventSet;
  InvokeMSG(Argument);
  IsArgumentError($Disposition) =>
    BEGIN
      Remove $Signal From PendingEventSet;
      LogError($Operation, $Disposition);
    END;
  END;
END;
```

## Step Eleven

Move <10a-b> and <10c-d> to the head of the block, reducing them to one instance. Preconditions: 1) <10a-b> and <10c-d> must not depend on \$\$Signal.Block; 2) <10a-b> and <10c-d>, when moved, must not affect \$\$Signal.Block. Met: manifestly true.

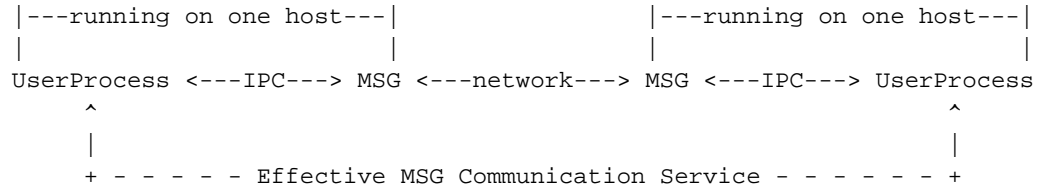
```
BEGIN
<11>   DECL Argument:Pointer LIKE
        Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
                OF ?SendList, ?ReceiveList, $Signal, $Disposition);
        $$Signal.Results <- $Disposition;
        $$Signal.Block =>
        BEGIN
            InvokeMSG(Argument);
            IsArgumentError($Disposition) =>
                LogError($Operation, $Disposition);
        END;
        Insert $Signal In PendingEventSet;
        InvokeMSG(Argument);
        IsArgumentError($Disposition) =>
        BEGIN
            Remove $Signal From PendingEventSet;
            LogError($Operation, $Disposition);
        END;
    END;
```

## Step Twelve

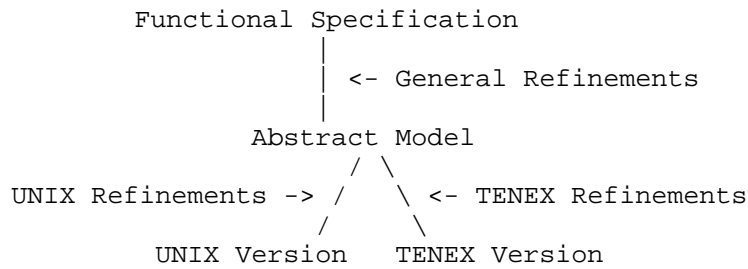
Remove \$Disposition from <11> (since it is also contained in \$\$Signal). Precondition: When MSG modifies \$\$Signal.Results such modifications must be reflected in \$Disposition. Met: \$\$Signal.Results and \$Disposition are pointers to the same object, and only the object pointed to is modified (this observation is based on a knowledge of MSG).

```
BEGIN
    DECL Argument:Pointer LIKE
        Allocate( !! GetArgumentBlockTypeFromOpCode($Operation)
                OF ?SendList, ?ReceiveList, $Signal);
        $$Signal.Results <- $Disposition;
        $$Signal.Block =>
        BEGIN
            InvokeMSG(Argument);
            IsArgumentError($Disposition) =>
                LogError($Operation, $Disposition);
        END;
        Insert $Signal In PendingEventSet;
        InvokeMSG(Argument);
        IsArgumentError($Disposition) =>
        BEGIN
            Remove $Signal From PendingEventSet;
            LogError($Operation, $Disposition);
        END;
    END;
```

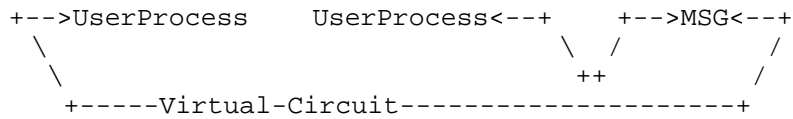
## Figures



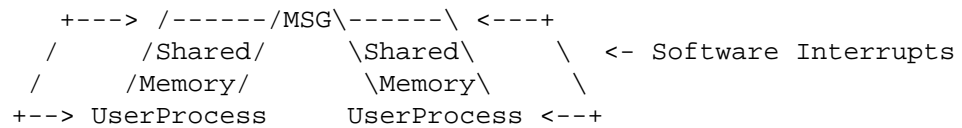
**Figure 1: Structure of User Process -- MSG Communication**



**Figure 2: Structure of The Abstract Model and Refinements**



**Figure 3: UNIX Virtual Circuits**



**Figure 4: TENEX Shared Memory**