Algebraic Specification Of A Communication Scheduler

Joseph Mathai*
Abha Moitra

TR 84-618
June 1984

*Computer Science Group
Tata Institute of Fundamental Research
Colaba, Bombay 400 005 INDIA

Department of Computer Science
Cornell University
Ithaca, New York 14853

Algebraic Specification Of A Communication Scheduler

Mathai Joseph

Computer Science Group
Tata Institute of Fundamental Research
Colaba, Bombay 400 005 INDIA

Abha Moitra

Department of Computer Science
Cornell University
Ithaca NY 14853

ABSTRACT

A distributed programming language normally incorporates one mechanism by which processes communicate with each other. This mechanism can be used to transfer information or to synchronize the flow of control in the program. Different communication mechanisms have been proposed for different languages. In this paper, we provide a common framework in which these mechanisms can be examined independently of the languages in which they may be embedded. Operationally, this framework is a communication scheduler : formally, it is specified algebraically as a data type. A number of different communication mechanisms, such as synchronous and asynchronous message passing, broadcasts and remote procedure calls, are modelled and, as an illustration of how global properties can be analysed, we consider the problem of deadlock detection.

1. INTRODUCTION

Modularity and abstraction are used in program construction so that a complex problem can be solved using a program with a number of small and relatively simple components. In sequential programs, these components are usually procedures, functions, or class-like encapsulations, while concurrent and distributed programs may also contain independently executing processes or tasks. Different kinds of components are distinguished by the form of control used to invoke their actions and by the means used to propagate information from one component to another. Thus, between the components of a sequential program, procedure (or function) calls and returns are normally used to transfer information and control, and block structure has been used to statically control access to data. Many schemes have been proposed for interprocess communication in concurrent and distributed programs and different languages have used, for example, synchronous message transfer [10], asynchronous message transfer [5], coroutine calls [15] and remote procedure calls [13]. In several of these schemes, sending data in a message is the common means of transferring information from one process to another, but the synchronization disciplines used for message transfer differ considerably and have important effects on global properties of the program, such as termination and deadlock. For example, the minor variation over standard CSP [10] needed to introduce output commands in guards adds to the symmetry and elegance of the language, makes it easier to avoid deadlock in certain cases, but certainly leads also to some implementational complexity [1].

Proof of the properties of a program written in a distributed programming language usually requires global reasoning about the whole program to be combined with local reasoning for each process. Such a proof can be established using a proof system which provides rules for each statement and for the particular communication mechanisms used in that language. Proof of a distributed program is usually more complex than that of a similar sequential program and the additional complexity is almost solely an outcome of the process interactions that take place through interprocess communication. Each proof system is specific to one programming language, and therefore to one set of communication mechanisms.

It is therefore of interest to model a multiplicity of communication mechanisms in a common framework so that their intrinsic properties can be examined independently of the operational details of the languages in which they may be embedded. We shall attempt to do so in this paper, using an algebraically specified communication environment in which the components of the distributed or concurrent program are defined as modules. The environment will be defined by a set of operations whose semantics are described by axioms; algebraic specification of this form has been used for defining abstract data types (e.g., Goguen, Thatcher and Wagner [8], and Guttag and Horning [9]), so we could consider this environment to be a 'communication data type' [3].

2. ALGEBRAIC SPECIFICATION

The algebraic specification of a data type consists of the definition of the set of operations, or functions, of the data type together with axioms which provide the semantics of the operations. We shall not describe the method of algebraic specification in detail (the interested reader is referred to [8], [9]) but we can illustrate it, and the syntax we shall be using, by defining a simple data type.

The operations of a data type will be defined using the following syntax :

[<mnemonic> =] <operation name> : <domain> -> <range>

Axioms will be numbered in sequence (for easy reference), sometimes given a mnemonic name, and written as algebraic equations with expressions on the left and right hand sides, the latter containing values of type range and logical expressions of the form *if..then..else..* .

2.1. A Simple Data Type

Let us define *MNSEQ* a data type which models an abstract sequence of items of type ***mod-name***. *MNSEQ* is defined using seven operations and eleven axioms, and ***mod-name*** is just a sequence of characters. We shall assume that the data type *BOOLEAN* is predefined.

```
        MNSEQ

NS = NewSeq :                       -> mnseq
     Insert  : mnseq X mod-name -> mnseq
     In      : mnseq X mod-name -> boolean
     Empty   : mnseq                -> boolean
     Del1    : mnseq X mod-name -> mnseq
     DelAll  : mnseq X mod-name -> mnseq
     Concat  : mnseq X mnseq      -> mnseq

for all s,s1 in mnseq and m,m1 in mod-name

 1 In(NS,m) = false
 2 In(Insert(s,m),m1) = if Eq(m,m1) then true else In(s,m1)

 3 Empty(NS) = true
 4 Empty(Insert(s,m)) = false

 5 Del1(NS,m) = NS
 6 Del1(Insert(s,m),m1) = if Eq(m,m1) then s else Insert(Del1(s,m1),m)

 7 DelAll(NS,m) = NS
 8 DelAll(Insert(s,m),m1) = if Eq(m,m1) then DelAll(s,m1)
                            else Insert(DelAll(s,m1),m)

 9 Concat(NS,s) = s
10 Concat(s,NS) = s
11 Concat(s,Insert(s1,m)) = Insert(Concat(s,s1),m)
```

where we assume the existence of a operation *Eq* to determine equality between two objects of type ***mod-name***.

The operation *Del1* deletes the first occurrence and the operation *DelAll* deletes each occurrence of an element from the sequence : thus use of *DelAll* results in the sequence being treated more as a set than a sequence. The data type *MNSEQ* will be used later in the definition of other data types.

2.2. Design of Algebraic Specification

It is well known ([11], [14]) that it is not possible to specify every computable operation with a finite number of axioms. Majster [11] therefore proposed the use of hidden operations, where a hidden operation is one that may only be invoked from some operation of the data type, and which is not accessible from outside the data type; using hidden operations it is always possible to give a finite specification for any computable operation [14]. We make extensive use of hidden operations in this paper; a '#' symbol preceding a operation identifies it as being a hidden operation.

Though algebraic specifications have been formalized and used for a number of years, very few data types have actually been defined in this methodology. We believe this is probably due to the fact that writing algebraic specifications is still an art. In this paper we shall also try to motivate the development of the various data types introduced in this paper.

A particular data type can be algebraically described in a number of different ways. It is therefore important to know why a particular algebraic specification is chosen. So, we shall often indicate the choices possible as well as the consequences of each decision. Further, the requirements placed on a data type are typically in terms of the semantics that should be provided for the operations. But since hidden operations are not available to a user of the data type, there is no obvious guideline for deciding what hidden operations should be introduced but we shall attempt to explain the basis for our choices.

3. MODELLING COMMUNICATION

Communication between concurrent or distributed processes requires action through some mediating agency such as shared memory, a communication medium, a 'transport layer', or an operating system. This agency provides a name space in which processes are assigned unique identification, and a means of conveying messages from one named process to another : in other words, the agency is the environment in which interprocess communication takes place and we shall refer to it subsequently as the 'communication environment', or just as the 'environment'. To relate the syntax and semantics of the algebraic specification with the more familiar operational view, we shall 'annotate' the axioms with operational descriptions of their semantics.

Initially the environment is empty and each module that is to participate in communication must be defined in the environment using the operation *EnterModName*. A module then indicates its willingness to communicate using the operation *Request*, which requires as arguments the name of the module and two objects of type *mnseq* which contain the names of the modules to which it is ready to send messages and from which it will accept messages respectively. This corresponds to a non-deterministic construct in a programming language, where one of a set of input or output commands may be selected for execution (as is the case in extended CSP [1]). An additional constraint in a programming language would be that for communication to take place, the type of the message to be input in one process must be identical to that to be output by another process. For simplicity, we shall assume that separate syntactic checks ensure this and ignore message types in this analysis. In synchronous communications, which we shall be modelling initially, the first process to attempt a communication must wait until a matching request comes from some other process. Another operation that must be provided is to allow a module to be removed from the environment, *RemoveModName*.

The semantics of the operation *Request* can be informally described as follows.

```
  Request(e,m,s1,s2)
= if mod-name m is not defined in the environment e                          (1)
      then drop this request
  else if m is attempting to wait for itself then drop this request          (2)
  else if none of the mod-names in s1 and s2 is present in the environment e (3)
      then error
  else if m already has a request pending in the environment e               (4)
      then drop this request
  else if possible match this request with a pending request                 (5)
  else if by adding this request all modules will have a request pending     (6)
      then deadlock-error
  else keep this request pending (to be satisfied later)                     (7)
```

The test in (1) can be accomplished by introducing a new boolean operation *IsModNameInEnv*. This operation could be kept as a hidden operation but since there might be other uses for it we can allow it to be invoked by a user. The test in (2) can be handled by making use of the operation *In* defined in *MNSEQ*. The test in (3) can be done by introducing a new operation *Strip* which takes as arguments an environment and a *mnseq* and returns a *mnseq* which is the sequence of all mod-names in the input *mnseq* that are also present in the environment.

Before we consider the other tests that have to be performed, we must first decide on how a pending request will be kept (7). The simplest solution would be to rewrite (7) as

else Request(e,m,s1,s2) (7′)

but then if *Request(e,m,s1,s2)* is encountered there is no way of knowing whether this is a new request for which tests have not been performed or whether it is a pending request for which the tests have been done but for which there was no matching communication. Such a distinction has to be made to specify, among other things, the test in (4). So, we should introduce a new operation, say *New-Fn1*, and rewrite (7) as

else New-Fn1(e,m,s1,s2) (7′′)

Algebraic specifications are defined in a hierarchical fashion : a new type is defined using some predefined types. For example, here we are trying to define a new type *env* using the predefined types *mnseq*, *boolean*. For any data type, we can define a minimal set of operations that are sufficient to describe every possible element of the new type [9]. In the present case, the operations *EnterModName*, *NewFn1* and *CreateEnv* are the minimal operations that are sufficient to describe every element in the type *env*. So, any environment will be of the form

$\{\text{EnterModName, NewFn1}\}^{*}$ CreateEnv

This means that the various operations that test whether or not an environment satisfies a particular property must typically be defined using three axioms. Of course, if enough additional operations are introduced, each original operation can be defined using one axiom only but then this axiom can get quite complicated [12]. We can avoid this if we could get an arbitrary environment to be of the form

$\{\text{NewFn2}\}^*$ CreateEnv

where *NewFn2* is some new operation.

This can be accomplished in the following way :

EnterModName(e,m) = if the mod-name m is in the environment e then e
else AddModName(e,m,NS,NS)

and (7) is written as

else change AddModName(e′,m,NS,NS) in e to AddModName(e′,m,s1,s2)

In *AddModName(e,m,s1,s2)*, module *m* is not waiting for any communication if and only if both *s1* and *s2* are empty; otherwise it is waiting for a communication.

The operations and axioms of the communication data type *SYNCH-COMM* are given below. In this data type, communication between modules is synchronous, i.e. a module making a communication request is blocked until its message is transmitted and this takes place when the sender and the receiver are both ready. We assume that each domain is extended to include appropriate error elements [7].

SYNCH-COMM

CE	= CreateEnv	:	->	env
EMN	= EnterModName	: env X mod-name	->	env
#AMN	= AddModName	: env X mod-name X mnseq X mnseq	->	env
Req	= Request	: env X mod-name X mnseq X mnseq	->	env
#Add		: env X mod-name X mnseq X mnseq	->	env
CMatch	= CanBeMatched	: env X mod-name X mnseq X mnseq	->	boolean
#Match		: env X mod-name X mnseq X mnseq	->	env
RMN	= RemoveModName	: env X mod-name	->	env
#DMN	= DeleteModName	: env X mod-name	->	env
ISMNE	= IsModNameInEnv	: env X mod-name	->	boolean
Wait	= Waiting	: env X mod-name	->	boolean
AllWait	= AllWaiting	: env	->	boolean
Strip		: env X mnseq	->	mnseq

for all e in env, m,m1 in mod-name, s1,s2,s3,s4 in mnseq

1 EMN(e,m) = if ISMNE(e,m) then e else AMN(e,m,NS,NS)

Each module in the environment is unique; a new ***mod-name*** is added to the environment along with two ***mnseq*** objects (initially empty) that will be used to contain the names of the modules to which communications may be sent and from which communications may be received.

2 Req(e,m,s1,s2) = if $\neg$ISMNE(e,m) then e
else if In(s1,m) $\vee$ In(s2,m) then e
else if Empty(Strip(e,s1)) $\wedge$ Empty(Strip(e,s2)) then error
else if Wait(e,m) then e
else if CMatch(e,m,s1,s2) then Match(e,m,s1,s2)
else if AllWait(Add(e,m,s1,s2)) then deadlock-error
else Add(e,m,s1,s2)

Requests for communication may only be received from modules defined in the environment. The object *s1* contains the names of modules to any of which module *m* is prepared to send messages, and *s2* the names of modules from any of which *m* is ready to accept messages. *s1* and *s2* should not contain *m*, nor should they both be empty. To simulate synchronous communication, a request from a module that is already waiting for a previous communication is ignored. Each request is tested for a match with other pending requests; a request that can be matched is satisfied immediately. If a request from a module cannot be matched immediately, and all the other modules have either terminated or are also waiting for communication, then no progress by any module is possible and the system is deadlocked. (If only part of the system is deadlocked but progress can be made by some modules, the processing of further communication requests is carried on.) If a request cannot be matched, and the entire system is not deadlocked, then that request is added to the list of pending requests.

For the present, we take the simple and straightforward view that deadlock occurs only if no module can make any further progress. Later, we will also show how it is possible to detect a partial deadlock.

```
3  CMatch(CE,m1,s1,s2) = false
4  CMatch(AMN(e,m1,s3,s4),m,s1,s2) = if Eq(m,m1) then CMatch(e,m,s1,s2)
                                     else if In(s2,m1) ∧ In(s3,m) then true
                                     else if In(s1,m1) ∧ In(s4,m) then true
                                     else CMatch(e,m,s1,s2)
```

A request can be matched with a pending request if the names of the modules *m* and *m1* appear in complementary send and receive requests (i.e. in the objects *s2* and *s3*, or in the objects *s1* and *s4*).

```
5  Match(CE,m,s1,s2) = error
6  Match(AMN(e,m1,s3,s4),m,s1,s2)
       = if Eq(m,m1) then AMN(Match(e,m,s1,s2),m1,s3,s4)
         else if In(s2,m1) ∧ In(s3,m) then AMN(e,m1,NS,NS)
         else if In(s1,m1) ∧ In(s4,m) then AMN(e,m1,NS,NS)
         else AMN(Match(e,m,s1,s2),m1,s3,s4)
```

In axiom 6, checks are made for inclusion of *m1* and *m* in *s2* and *s3*, respectively, and then in *s1* and *s4*. Note that this order can be reversed. When two requests match, they are cancelled (this is done by setting the associated *mnseq* objects to the value *NS*).

```
7  Add(CE,m,s1,s2) = error
8  Add(AMN(e,m1,s3,s4),m,s1,s2) = if Eq(m,m1) then AMN(e,m,s1,s2)
                                  else AMN(Add(e,m,s1,s2),m1,s3,s4)
```

For a request that cannot immediately be matched, *Add* associates the request with the appropriate *AMN* operation for that module. Thus the objects *s1* and *s2* associated with an unmatched request will replace the empty *mnseq* objects associated with that module.

```
9  RMN(e,m) = if Wait(e,m) then e else DMN(e,m)

10 DMN(CE,m) = CE
11 DMN(AMN(e,m1,s1,s2),m)
       = if Eq(m,m1) then DMN(e,m)
         else if ¬(Empty(s1) ∧ Empty(s2)) ∧ Empty(DelAll(s1,m))
                 ∧ Empty(DelAll(s2,m)) then error
         else AMN(DMN(e,m),m1,DelAll(s1,m),DelAll(s2,m))
```

A module may be removed only if it is not awaiting any communication. If the removal of a module causes some other module to wait on empty objects *s1* and *s2* then an error is raised.

12 ISMNE(CE,m) = false
13 ISMNE(AMN(e,m,s1,s2),m1) = if Eq(m,m1) then true else ISMNE(e,m1)

14 Wait(CE,m) = false
15 Wait(AMN(e,m,s1,s2),m1) = if ¬Eq(m,m1) then Wait(e,m1)
else if Empty(s1) ∧ Empty(s2) then false
else true

A module which is not waiting for any communication will have empty objects *s1* and *s2*.

16 AllWait(CE) = true
17 AllWait(AMN(e,m,s1,s2)) = if Empty(s1) ∧ Empty(s2) then false else AllWait(e)

AllWait checks whether there are any modules in the environment that are not waiting for a communication : thus axiom 16 follows because in this case the environment is empty.

18 Strip(e,NS) = NS
19 Strip(e,Insert(s,m)) = if ISMNE(e,m) then Insert(Strip(e,s),m) else Strip(e,s)

Strip takes an *mnseq* object as an argument and deletes all *mod-name* in it that are not defined in the environment.

4. DEADLOCK

In the previous section we had assumed that it is necessary to detect a deadlock only if no module could make any further progress. While every deadlock situation will eventually lead to a situation in which no module can make any further progress, it would be preferable to detect even a partial deadlock as soon as it occurs. In this section we show that a deadlock of some processes can be detected even when there are other processes that can make further progress. The procedure for deadlock detection is formulated to minimize the amount of computation that is required when a new unmatched communication request is added to the environment. This procedure works as follows.

Let e be an environment with n modules m_1, m_2, .. m_n and let *Safe(e,*m_i*)* be the set of modules that must be unblocked for execution of m_i to be possible. Initially, when there is no communication request pending in an environment e, *Safe(e,*m_i*)* $=$ $\{m_i\}$ for all $1 \leq i \leq n$. When a new communication request of the form

m_j waiting on m_{j1}, m_{j2}, .., m_{jk}

that cannot be immediately satisfied is added to the environment e to give a new environment e', the operation *Safe* is redefined as follows :

Safe(e$'$,m_j) $=$ Safe(e, m_{j1}) $\cup$ Safe(e, m_{j2}) $\cup$ .. $\cup$ Safe(e, m_{jk}) - $\{m_j\}$

for $1 \leq i \leq k$ Safe(e$'$, m_{ji}) $=$ Safe(e, m_{ji}) - $\{m_j\}$

A deadlock occurs in an environment e if there is some $1 \leq j \leq n$ such that *Safe(e,*m_j*)* $= \emptyset$. This formulation of deadlock detection can be easily incorporated into the data type *SYNCH-COMM* by adding another *mnseq* argument to the operation *AMN* to keep track of the 'safe' set for that module. We do not present the new data type here as it involves a straightforward change to the data type *SYNCH-COMM*.

There are several ways in which deadlock may be detected. We have distinguished between partial and complete deadlocks, referring by the later term to the case where all the modules left in the environment are blocked awaiting communication. But a partial deadlock will also eventually become a complete deadlock. On the other hand, there is a specific communication request that completes the condition for a partial deadlock and it is naturally desirable that this be detected as soon as it occurs. In terms of our model, this condition is represented by the truth of the relation $Safe(e,\ m_j) = \emptyset$ for some j, $1 \leq j \leq n$.

Since this scheme detects partial deadlocks, it is closer in form to one described by Chandy, Misra and Haas [2] than, for example, the work on termination detection (e.g., [4], [6]) which tests for a global property. But it differs from all such work because, by its applicative nature, it does not rely on implementational details like the propagation of test messages such as probes [2] or signals [6] to detect deadlocks. This is a consequence of the fact that we are modelling the communication environment, rather than individual modules. For this reason also, it is not necessary to define a spanning tree or a ring along which to send deadlock detection signals.

As deadlock detection takes place before every unmatched communication request is added to e, the total associated cost is proportional to the number of such requests. Further, for each unmatched communication request, the cost of deadlock detection is proportional to the total number of modules in the environment e (one pass over the environment e is enough to update all 'safe' information).

5. MESSAGE QUEUES

Under the discipline of synchronous communication, there can be at most one request pending from a module so it follows that two successive messages sent from one module to another will reach in the order in which they were sent. Can it also be ensured that messages are accepted by modules according to the order in which they were received? This is easily done by replacing axioms 7 and 8 by the following axioms :

7a Add(CE,m,s1,s2) = AMN(CE,m,s1,s2)
8a Add(AMN(e,m1,s3,s4),m,s1,s2) = if Eq(m,m1) then Add(e,m,s1,s2)
else AMN(Add(e,m,s1,s2),m1,s3,s4)

which will result in message propagation having a first-in-first-out order. By suitably extending the operations *AMN* and *Add*, it can also be arranged for modules to be assigned priorities so that message transmission follows the order imposed by these priorities.

Note that in the last line of axiom 2 in *SYNCH-COMM* we could have used

else AMN(Del(e,m),m,s1,s2)

where *Del* would replace *AMN(e'*,m,NS,NS) in *e* by *e'*. But then such a choice would have made it more difficult to alter *SYNCH-COMM* to ensure that modules receive messages only in the order in which they were sent.

6. REMOTE PROCEDURE CALLS

An extended form of synchronous communication can be used to describe remote procedure calls from one module to another. Syntactically, a successful remote procedure call from module *m* to module *m1* can be simulated by four operations : a send request for the call from *m* to *m1*, acceptance of this request by *m1*, a send request for the reply from *m1* to *m*, and receipt of this by *m*. But defining this protocol literally in the axioms has several deficiencies : for example, the call by *m* and its acceptance by *m1* may be followed by other communication requests from *m* before waiting for a reply from *m1*, or the definition of the axioms may be such as to prevent nested remote procedure calls (i.e. calls from *m1* to other modules before a reply is sent to *m*).

It is necessary to introduce some new operations : let *RPC* be the remote procedure call and *Serve* the request to accept such a call :

```
RPC   : env X mod-name X mod-name -> env
Serve : env X mod-name            -> env
```

The corresponding axioms are :

```
20 Serve(e,m) = Req(e,m,NS,DelAll(NewFn2(e),m))

21 RPC(e,m,m1) = if ¬ISMNE(e,m) then e
                 else if Eq(m,m1) then e
                 else if Empty(Strip(e,{m1})) then error
                 else if Wait(e,m) then e
                 else if CMatch(e,m,{m1},NS)
                      then Req(Match(e,m,{m1},NS),m,NS,{m1})
                 else if AllWait(Add(e,m,{m1},NS)) then deadlock-error
                 else RP(Add(e,m,{m1},NS),m,m1)

22 RP(e,m,m1) = if Wait(e,m) then RP(e,m,m1) else Req(e,m,NS,{m1})
```

where *NewFn2(e)* returns the set of all the modules in the environment *e* ; *RP* is a constructor type operation for which new axioms have to be defined and $\{m1\}$ = *Insert(NS, m1)*.

This is a relatively simple solution, but it introduces an additional constructor type function *RP*. Another solution would be to 'tag' the existing constructor type operations so as to distinguish between *RPC* and ordinary *Req* operations. This type of solution will be used later in this paper for broadcast communication.

7. ASYNCHRONOUS MESSAGE PASSING AND BROADCAST

The addition of fully general asynchronous communication between modules requires unbounded buffering, because a module may send an unlimited number of asynchronous messages to one or more other modules. In any specific case, the number of asynchronous messages sent by one module and still to be received by another module would be limited only by module termination, or by the sender attempting a synchronous communication or a remote procedure call, both of which block the module's execution until completed. There are two constraints on asynchronous communication: messages sent from one module to another must be received in the order in which they were sent, and no more messages may be received than are sent. Permitting a module to send a message to a number of other modules, in a single operation, is equivalent to a multicast or a broadcast operation. Conversely, there is no essential difference between a broadcast operation with just one destination module and simple asynchronous communication. We shall therefore consider the problem of modelling broadcast communication.

The data type *SYNCH-COMM* has two constructor type operations [9], *CE* and *AMN*. One way of adding broadcasts to this data type would be to introduce another constructor type operation, e.g. *BroadCast(e,m,s1)*, which would allow more than one broadcast request to be pending for the same module. The addition of a new constructor type operation *BC* would require some of the existing axioms to be rewritten and the number of axioms required would also increase. (Typically, if an operation was originally defined for *CE* and *AMN*, it would then have to be defined for *CE*, *AMN* and *BC*.)

A simpler way of introducing the facility for broadcasts, and one we shall follow here, is to add an argument to the operation *AMN*.

```
  BC = BroadCast    : env X mod-name X mnseq                     -> env
AMN = AddModName : env X mod-name X mnseq X mnseq X mnseq -> env
```

Both *Req* and *BC* will be handled by the operation *AMN* (and the number of constructor type operations will therefore not increase.) The third argument for *AMN* is the set containing the names of modules to which messages are to be broadcast. In this set, the oldest broadcast requests will be at the front, thus guaranteeing a first-come-first-served order for broadcasts to the same module. The new data type, *ASYNCH-COMM*, is defined below.

ASYNCH-COMM

```
CE      = CreateEnv      :                                          -> env
EMN     = EnterModName   : env X mod-name                           -> env
#AMN    = AddModName     : env X mod-name X mnseq X mnseq X mnseq -> env
#Add                     : env X mod-name X mnseq X mnseq X mnseq -> env
```

```
Req      = Request          : env X mod-name X mnseq X mnseq -> env
BC       = BroadCast        : env X mod-name X mnseq         -> env
CMatch   = CanBeMatched     : env X mod-name X mnseq X mnseq -> boolean
#Match                      : env X mod-name X mnseq X mnseq -> env
RMN      = RemoveModName    : env X mod-name                 -> env
#DMN     = DeleteModName    : env X mod-name                 -> env
ISMNE    = IsModNameInEnv   : env X mod-name                 -> boolean
Wait     = Waiting          : env X mod-name                 -> boolean
AllWait  = AllWaiting       : env                            -> boolean
Strip                       : env X mnseq                    -> mnseq
```

for all e in env, m,m1 in mod-name, s1,s2,s3,s4,s5,s6 in mnseq

```
1 EMN(e,m) = if ISMNE(e,m) then e else AMN(e,m,NS,NS,NS)

2 Req(e,m,s1,s2) = if ¬ISMNE(e,m) then e
                   else if In(s1,m) ∨ In(s2,m) then e
                   else if Empty(Strip(e,s1)) ∧ Empty(Strip(e,s2)) then error
                   else if Wait(e,m) then e
                   else if CMatch(e,m,s1,s2) then Match(e,m,s1,s2)
                   else if AllWait(Add(e,m,s1,s2,NS)) then deadlock-error
                   else Add(e,m,s1,s2,NS)

3 BC(e,m,NS) = e
4 BC(e,m,Insert(s1,m1)) = if ¬ISMNE(e,m) then e
                          else if In(s1,m1) ∨ Eq(m,m1) then BC(e,m,s1)
                          else if ¬ISMNE(e,m1) then BC(e,m,s1)
                          else if Wait(e,m) then e
                          else if CMatch(e,m,Insert(NS,m1),NS)
                               then BC(Match(e,m,Insert(NS,m1),NS),m,s1)
                          else BC(Add(e,m,NS,NS,Insert(NS,m1)),m,s1)

5 CMatch(CE,m1,s1,s2) = false
6 CMatch(AMN(e,m1,s3,s4,s5),m,s1,s2) = if Eq(m,m1) then CMatch(e,m,s1,s2)
                                       else if In(s2,m1) ∧ In(s3,m) then true
                                       else if In(s1,m1) ∧ In(s4,m) then true
                                       else if In(s2,m1) ∧ In(s5,m) then true
                                       else CMatch(e,m,s1,s2)
7 Match(CE,m,s1,s2) = error
8 Match(AMN(e,m1,s3,s4,s5),m,s1,s2)
      = if Eq(m,m1) then AMN(Match(e,m,s1,s2),m1,s3,s4,s5)
        else if In(s2,m1) ∧ In(s3,m) then AMN(e,m1,NS,NS,s5)
        else if In(s1,m1) ∧ In(s4,m) then AMN(e,m1,NS,NS,s5)
        else if In(s2,m1) ∧ In(s5,m) then AMN(e,m1,s3,s4,Del1(s5,m))
        else AMN(Match(e,m,s1,s2),m1,s3,s4,s5)

9 Add(CE,m,s1,s2,s3) = error
```

```
10 Add(AMN(e,m1,s4,s5,s6),m,s1,s2,s3)
       = if ¬Eq(m,m1) then AMN(Add(e,m,s1,s2,s3),m1,s4,s5,s6)
         else if Empty(s3) then AMN(e,m,s1,s2,NS)
         else AMN(e,m,s4,s5,Concat(s3,s6))

11 RMN(e,m) = if Wait(e,m) then e else DMN(e,m)

12 DMN(CE,m) = CE
13 DMN(AMN(e,m1,s1,s2,s3),m)
       = if Eq(m1,m) then DMN(e,m)
         else if ¬(Empty(s1) ∧ Empty(s2)) ∧ Empty(DelAll(s1,m))
              ∧ Empty(DelAll(s2,m)) then error
         else AMN(DMN(e,m),m1,DelAll(s1,m),DelAll(s2,m),DelAll(s3,m))

14 ISMNE(CE,m) = false
15 ISMNE(AMN(e,m,s1,s2,s3),m1) = if Eq(m,m1) then true else ISMNE(e,m1)

16 Wait(CE,m) = false
17 Wait(AMN(e,m,s1,s2,s3),m1) = if ¬Eq(m,m1) then Wait(e,m1)
                                else if Empty(s1) ∧ Empty(s2) then false
                                else true

18 AllWait(CE) = true
19 AllWait(AMN(e,m,s1,s2,s3)) = if Empty(s1) ∧ Empty(s2) then false
                               else AllWait(e)

20 Strip(e,NS) = NS
21 Strip(e,Insert(s,m)) = if ISMNE(e,m) then Insert(Strip(e,s),m) else Strip(e,s)
```

The data type *ASYNCH-COMM* can be augmented in a straightforward way to provide operations for deadlock detection. In the term *AMN(e,m,s1,s2,s3)*, *s3* is not involved in deadlock detection and hence the extension for *ASYNCH-COMM* would be very similar to that suggested for *SYNCH-COMM.*

Relatively few changes were needed to convert *SYNCH-COMM* to *ASYNCH-COMM,* and the operations and axioms of the new data type show a high degree of similarity with those defined earlier. This was accomplished partly by treating *mnseq* objects both as sequences (using the operation *Del1*) and as sets (using the operation *DelAll*). Another reason for achieving this high similarity was that the number of constructor type operations in both data types was the same. It is also interesting to note that the blocking effect of synchronous communication can be preserved, despite the introduction of asynchronous communication, merely by choosing an appropriate order of checking in the axioms. Thus, by axiom 4, no broadcast requests (*BC*) are accepted from a module waiting for a synchronous communication request (*Req*) : when a new request is added, synchronous communication requests are matched before broadcast requests (axiom 8).

CONCLUSIONS

In this paper we have modelled a number of different interprocess communication schemes used in concurrent and distributed programming by specifying them algebraically as abstract data types.

Two criteria can be used to judge the usefulness of such specifications. First, how faithfully do they represent the commonly understood semantics of the communication mechanisms? Secondly, how well do the specifications for different communication mechanisms illustrate any inherent similarities between them?

The importance of achieving the first criterion lies in the fact that, in practice, each communication mechanism is defined either informally, or operationally as part of a programming language. Once a formally defined and operationally acceptable specification has been produced, there appears to be many operational similarities between the mechanisms.

The design of the specifications described in this paper proceeded with these criteria acting as constraints. The specification of the data type *ASYNCH-COMM* shows that mechanism as different as broadcasts and synchronous communication can be modelled in a common framework. Simple extensions to this data type permit more complex operations, such as remote procedure calls, to be specified. Global properties, like the presence of deadlocks, can also be quite easily considered.

REFERENCES

1. A.J. Bernstein, Output guards and nondeterminism in "Communicating Sequential Processes", *ACM Trans. Prog. Lang. and Sys.*, *2,* 2, April 1980, pp. 234-238.

2. K.M. Chandy, J. Misra, L.M. Haas, Distributed deadlock detection, *ACM Trans. on Comp. Sys.*, *1,* 2, May 1983, pp. 144-156.

3. P.R.F. Cunha, T.S.E. Maibaum, A communication data type for message oriented programming, *Proc. IV Intl. Symp. on Prog.*, Springer-Verlag, Lecture Notes in Computer Science, Vol. 83, 1980, pp. 79-91.

4. E.W. Dijkstra, C.S. Scholten, Termination detection for diffusing computations, *Inf. Proc. Lett.*, *11,* 1, August 1980, pp. 1-4.

5. J.A. Feldman, High level programming for distributed computing, *Comm. ACM*, *21,* 11, November 1978, pp. 934-941.

6. N. Francez, Distributed termination, *ACM Trans. on Prog. Lang. and Sys.*, *2*, 1, January 1980, pp. 42-55.

7. J.A. Goguen, Abstract errors for abstract data types, in *Formal Description of Programming Concepts,* E.J. Neuhold (Ed.), North Holland, 1978, pp. 491-525.

8. J.A. Goguen, J.W. Thatcher and E.G. Wagner, An initial algebra approach to the specification, correctness, and implementation of abstract data types, in *Current Trends in Programming Methodology Vol. IV : Data Structuring*, R.T. Yeh (Ed.), Prentice-Hall, Englewood Cliffs, 1978, pp. 80-149.

9. J.V. Guttag and J.J. Horning, The algebraic specification of abstract data types, *Acta Inform., 10,* 1, 1978, pp. 27-52.

10. C.A.R. Hoare, Communicating sequential processes, *Comm. ACM, 21,* 8, August 1978, pp. 666-677.

11. M.E. Majster, Limits of the 'algebraic' specification of abstract data types, *SIGPLAN Notices, 12,* 10, October 1977, pp. 37-42.

12. A. Moitra, Direct implementation of algebraic specification of abstract data types, *IEEE Trans. on Software Eng., SE-8,* 1, January 1982, pp. 12-20.

13. B.J. Nelson, Remote procedure call, Tech. Rep., Computer Science Department, Carnegie-Mellon University, May 1981.

14. J.W. Thatcher, E.G. Wagner and J.B. Wright, Data type specification : parameterization and the power of specification techniques, in *Proc. of the Tenth Annual ACM Symp. on Theory of Computing* (1978) 119-132.

15. N. Wirth, *Programming in Modula 2,* Springer-Verlag, 1982.