Describing and Analyzing Distributed System Designs

George S. Avrunin*
Jack C. Wileden**

COINS Technical Report 82-2
January 1982

*Department of Mathematics and Statistics
University of Massachusetts
Amherst, Massachusetts 01003

**Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Note: Submitted for publication.
Please do not distribute in this form.

ABSTRACT

In this paper we outline an approach to describing and analyzing designs for distributed software systems. A descriptive notation is introduced and analysis techniques applicable to designs expressed in that notation are presented. The usefulness of the approach is illustrated by applying it to a realistic distributed software system design problem involving mutual exclusion in a computer network.

1. Introduction

As part of a project aimed at creating a sophisticated, integrated software development environment, we have been seeking tools and techniques to aid in the pre-implementation stages of distributed software system development. In particular, we have sought to produce a notation appropriate for describing the design of a distributed system along with a collection of analysis techniques applicable to designs expressed in that notation. Our goal is a notation and corresponding set of analysis techniques that can be understood and used by developers of distributed software systems and also can be automated so that they may provide a basis for distributed software design tools in an integrated software development environment.

To date our efforts have resulted in a framework for a notation and an approach to analyzing descriptions expressed in that notation. The notation describes systems as collections of sequential processes communicating entirely via message transmission, and hence is well-suited for use in developing a distributed system's design. The analysis techniques employ methods derived from basic algebra. Our experience has shown that these techniques provide valuable assistance in uncovering even very subtle flaws in designs expressed in the notation. Moreover, they can also be used to rigorously demonstrate that certain aspects of a design are correct.

In this paper we briefly describe both our notational framework and our approach to analysis, illustrating their use and their usefulness with a realistic example. The next section gives an outline of the notation and the analysis techniques, and compares our approach with some related work by other researchers. Following that, we discuss the distributed mutual exclusion problem that serves as the basis of our example. We then present the example, illustrating the use of our notation in developing a design for a distributed system and showing how our analysis techniques can be used both to uncover design errors and to demonstrate correctness of aspects of a design. We conclude the paper with an assessment of the applicability of our work and prospects for future progress.

2. Framework for Notation and Analysis

Our work on techniques for describing and analyzing distributed systems has been guided by our interest in contributing to the production of practical, automated tools applicable to the pre-implementation stages of distributed software system development. We believe that this goal imposes two basic constraints. First, it requires that we base our techniques on a descriptive formalism (with accompanying notation) that not only is precise enough to be unambiguous, but also is appropriate for use by distributed software developers who may have no special mathematical or theoretical training. Second, it requires that we provide practical analysis methods that can be applied to descriptions phrased in that formalism and that can answer the types of questions arising most crucially during the design of distributed software systems.

Our choice of a descriptive formalism reflects our view of the distributed software development process. We believe that the designer of distributed software needs tools that will support descriptions of a modularization for the system, identifying the component processes of the system and specifying the ways in which those components will interact. Such a description must be sufficiently abstract to allow the designer to focus on just the properties of interest, namely modularity and interaction, without being distracted by details concerning other properties that are irrelevant at this stage. At the same time, the description must be sufficiently rigorous that it can be analyzed. In addition to providing abstraction and rigor, we feel that a pre-implementation descriptive formalism must be relatively easy to understand and use. Specifically, it must be amenable to use by software designers who may have little or no training in advanced mathematics or theoretical computer science. Therefore, an appropriate formalism should bear a reasonable relationship to standard software specification and design techniques. Ideally, it should be possible to provide an automated version of the formalism to permit its use in an distributed software development environment [CLAR81]. Finally, a formalism can only be appropriate for general use during distributed software design if it is applicable to a wide range of distributed system organizations.

The descriptive formalism that we have chosen to use as a basis for our work is the Dynamic Process Modelling Scheme (DPMS) and its Dynamic Modelling Language (DYMOL). This formalism, described in detail in [WILE80], was originally developed for studying distributed systems with dynamic structure [WILE78]. It evolved from the PPML formalism [RIDD79] that served as the foundation for the DREAM software development system [RIDD78]. In DPMS a distributed system is viewed as a collection of sequential processes, which interact and coordinate their behavior via message transmission. The formalism provides a means for describing a system's configuration in terms of the processes that compose it and the interprocess communication linkages among those processes. An abstract description of the behavior of each process in a modelled system can be given by a DYMOL "program" for that process. DYMOL is an Algol-like language consisting of control constructs, message transmission instructions (SEND and RECEIVE), instructions for modifying interprocess communication pathways (ESTABLISH and CLOSE), and instructions for modifying the composition of the system by adding or deleting processes (CREATE and DESTROY). Since DYMOL provides almost no constructs for describing the algorithmic details of a process' internal computations, DPMS descriptions necessarily focus on a distributed system's modularization and the interaction among its components. A formal, automata-theoretic semantics, defined in [WILE78], underlies DPMS descriptions of distributed systems. We have also defined a closed form, non-procedural representation for distributed system behavior, called constrained expressions, and demonstrated the equivalence of DYMOL descriptions and constrained expression descriptions for a large subclass of distributed systems with dynamic structure [WILE78].

The DPMS descriptive formalism on which we are basing our work is similar to several other approaches to describing distributed systems. In particular, viewing a system as a collection of communicating sequential processes is common to most description schemes. Hoare's Communicating Sequential Processes [HOAR78], Brinch Hansen's Distributed Processes [BRIN78], and the tasking facility in the Ada programming language [DOD80] are three of the better known examples of descriptive approaches that take this view. It is interesting to note, however, that all three of these approaches employ an interprocess communication protocol in which information is transferred only when both the sender and the receiver are simultaneously prepared to communicate. In DPMS we take the more general view that a message sender need not wait for a process to receive the message that it sends, or even know what process, if any, will receive the message. Thus, compared to DPMS, the Hoare, Brinch Hansen, and Ada approaches limit the possible concurrent activity and restrict the distributed system organizations that can be represented in a natural way. On the other hand, when it is desired to impose such restrictions this can be easily done using DPMS (c.f., [LISK79]). Thus, although we plan eventually to investigate the application of our analysis techniques to these well-known descriptive approaches, at present we prefer to base our work primarily on the more general DPMS formalism.

The Hoare, Brinch Hansen, and Ada descriptive formalisms are all programming languages, and thus were not explicitly intended for pre-implementation use. Lauer's COSY formalism [LAUE79] and the distributed system specification technique of Ramamritham and Keller [RAMA81] represent approaches that are intended for pre-implementation description of distributed systems. Both are based on formal semantic models, the former on the theory of nets and path expressions [LAUE75] and the latter on temporal logic [PNUE79]. Both also describe distributed systems as collections of communicating sequential processes. The two approaches both, however, distinguish two types of processes, sometimes called customers and resources, and limit process interaction to the use of resources by customers. Hence the specifications of process synchronization in the two approaches are restricted to statements regarding the acceptable usage of resources by customers, which limits the range of process interactions that can be described. While both of these approaches resemble ours, since they emphasize abstract pre-implementation descriptions and provide a formal basis for analysis, they differ from our work both in their limitations on the range of systems that can be described and in the style of the analysis that they support.

We conclude this discussion by outlining our approach to analysis, which is described in more detail in [AVRU81]. Our analysis techniques are based on applying algebraic methods to a DPMS description of a distributed system in order to determine whether a particular pattern of behaviors can occur in the described system. We view the behaviors of a distributed system as a set of sequences of event occurrences, implicitly defined by the DYMOL programs describing the system's processes, the configuration of communication paths interconnecting the processes, and the semantics of the DPMS formalism. An explicit, closed-form representation of the set of sequences of event occurrences is provided by the constrained expression corresponding to the DYMOL description [WILE78], in much the same way that a regular expression provides an explicit, closed-form representation for the set of behaviors implicitly defined by the corresponding finite automaton. Thus, to analyze a distributed system description we attempt to determine whether a particular event or (sub)sequence of events appears in the set of sequences of event occurrences constituting the system's behaviors. The event pattern in question may correspond to some desirable property of the system, whose absence would reflect an error in system design. Alternatively, the pattern may represent a pathological system behavior, such as a deadlock or process starvation, whose presence would represent an error in the distributed system as presently described. In any case, our analysis techniques proceed by deriving a collection of inequalities representing the number of occurrences of various events in the behaviors or partial behaviors of the system. This can be viewed as a generalization of the techniques employed by Habermann [HABE72] in analyzing a semaphore solution to a producer-consumer problem. The inequalities are based upon the system's description in the DPMS formalism (we have found that the constrained expression description facilitates this derivation procedure), the semantics of DPMS, and the event or event sequence whose existence as a system behavior is in question. An inconsistency among the derived inequalities implies that the behavior in question cannot occur in the system and indicates why that behavior is impossible. If the derived inequalities are consistent, they can be used in an attempt to produce a particular system behavior containing the specified event occurrence pattern. Examples illustrating both ways of using the analysis techniques appear in Section 4.

3. An Example Design Problem

To investigate the usefulness of our descriptive notation and associated analysis techniques, we have applied them to several distributed software design problems. In this paper we present the results of one such experiment in order to illustrate both the notation and the analysis techniques.

The distributed software design problem that we address in this example is mutual exclusion in a distributed system. The basic problem is to create a mechanism that will allow nodes in a distributed system to achieve mutual exclusion when they have no common shared memory, but can communicate only by message passing. This is a realistic problem that is of particular significance to designers of computer networks, since nodes in a network normally do not have access to a common shared memory, but can communicate only through messages.

Mutual exclusion in a distributed system has been studied by Lamport [LAMP78] and by Ricart and Agrawala [RICA81a,81b], who have presented algorithms for solving the problem. Our interest here is not in developing a new approach to solving the problem of mutual exclusion in a distributed system. Rather, our goal is to demonstrate the usefulness of our descriptive notation and analysis techniques for developing solutions to this and other distributed software design problems. We have, therefore, relied upon the approach developed by Ricart and Agrawala as a basis for our example solution to the problem of mutual exclusion in a distributed system. Hence, our example should not be construed as offering a novel solution to the distributed mutual exclusion problem, but as presenting an illustration of how a satisfactory solution to that problem might be developed.

Familiarity with the Ricart and Agrawala solution to the distributed mutual exclusion problem is not required for understanding and appreciating the example. A brief outline of their approach may, however, make the example easier to follow. In essence, their distributed mutual exclusion algorithm requires that a node wishing to obtain exclusive use of a shared resource send a request for such use to each of the other nodes in the distributed system and then wait until all of the other nodes have replied before proceeding to use the resource. Whenever a node receives a request message from another node, it decides whether to reply immediately, thereby granting its permission to use the resource, or to defer its reply until after it has used the resource itself. This decision is based upon the relative priority of the requesting node and the recipient of the request. Priorities are determined in part by a sequence number sent as one portion of the request message and in part by a fixed priority ordering on the nodes that is used in case two sequence numbers are equal. The sequence numbers are generated by the individual nodes and are similar to the numbers used in Lamport's "bakery algorithm" [LAMP74].

4. Example Design Development Process

Suppose that, at an early stage in designing a distributed software system, a designer recognizes that mutually exclusive use of some system resource by the nodes in the system would be necessary. Suppose further that the designer then chooses to focus temporarily on working out this aspect of the system's design, employing the notation and techniques outlined above. The remainder of this section describes a hypothetical design development process that this designer might then follow. As mentioned previously, the actual solution to the distributed mutual exclusion problem that results from this hypothetical design development process is based on an algorithm due to Ricart and Agrawala.

As a first step in the design development process, the designer chooses to decompose the distributed mutual exclusion aspect of a node's computation into three cooperating subparts. These subparts can be represented as processes, and might even be implemented on separate processors if the nodes of the overall distributed system were themselves networks of processors. One process in this decomposition would primarily be responsible for generating requests for use of the shared resource and then performing the critical section processing involving that resource once exclusive use of it had been granted. This process will be referred to as the _invoker_. A second process, designated the _reply handler_, would receive the replies from other nodes in the distributed system indicating that they had received the invoker's request for mutually exclusive use of the shared resource and were prepared to grant that request. Upon receiving such replies from all other nodes in the distributed system, the reply_handler process would inform the invoker process that it had been granted exclusive use of the shared resource and could proceed with its critical section processing. Finally, a set of processes would be responsible for receiving and responding to the requests for mutually exclusive use of the shared resource that will be generated by other nodes in the distributed system. Each such process, referred to as a request handler, would receive and respond to the requests of one of the distributed system's other nodes. Under certain circumstances a request_handler process might decide to defer a reply, in which case it would inform the invoker process of this decision so that the invoker could later send a reply. This modularization of the node's activity closely parallels the decomposition used in the Ricart and Agrawala distributed mutual exclusion algorithm ([RICA81a]).

Figures 1, 2, and 3 are DYMOL programs that the designer might use to describe the behavior of the invoker, reply_handler, and request_handler processes, respectively. Taken together, these three DYMOL programs describe one node in a distributed system consisting of three nodes. The designer must also specify how the processes are interconnected by communication linkages and indicate the communication linkages joining them with the other nodes in the distributed

```
WHILE INTERNAL TEST DO
   BEGIN
      RECEIVE cr_in;
      SET BUFFER := true;
      SEND cr_out;
      SEND listen;
      SET BUFFER := sequence_number;
      SEND req1_2;
      SEND req1_3;
      RECEIVE ok;
      SET BUFFER := critical;
      RECEIVE cr_in;
      SET BUFFER := false;
      SEND cr_out;
      RECEIVE rh2_in;
      IF BUFFER = def THEN
         BEGIN
            SEND reply1_2;
            SET BUFFER := no_def
         END
      SEND rh2_out;
      RECEIVE rh3_in;
      IF BUFFER = def THEN
         BEGIN
            SEND reply1_3;
            SET BUFFER := no_def
         END
      SEND rh3_out;
   END
```

Invoker DYMOL program

Figure 1

```
DO FOREVER
   BEGIN
      RECEIVE listen;
      RECEIVE reply2_1;
      RECEIVE reply 3_1;
      SEND ok
   END
```

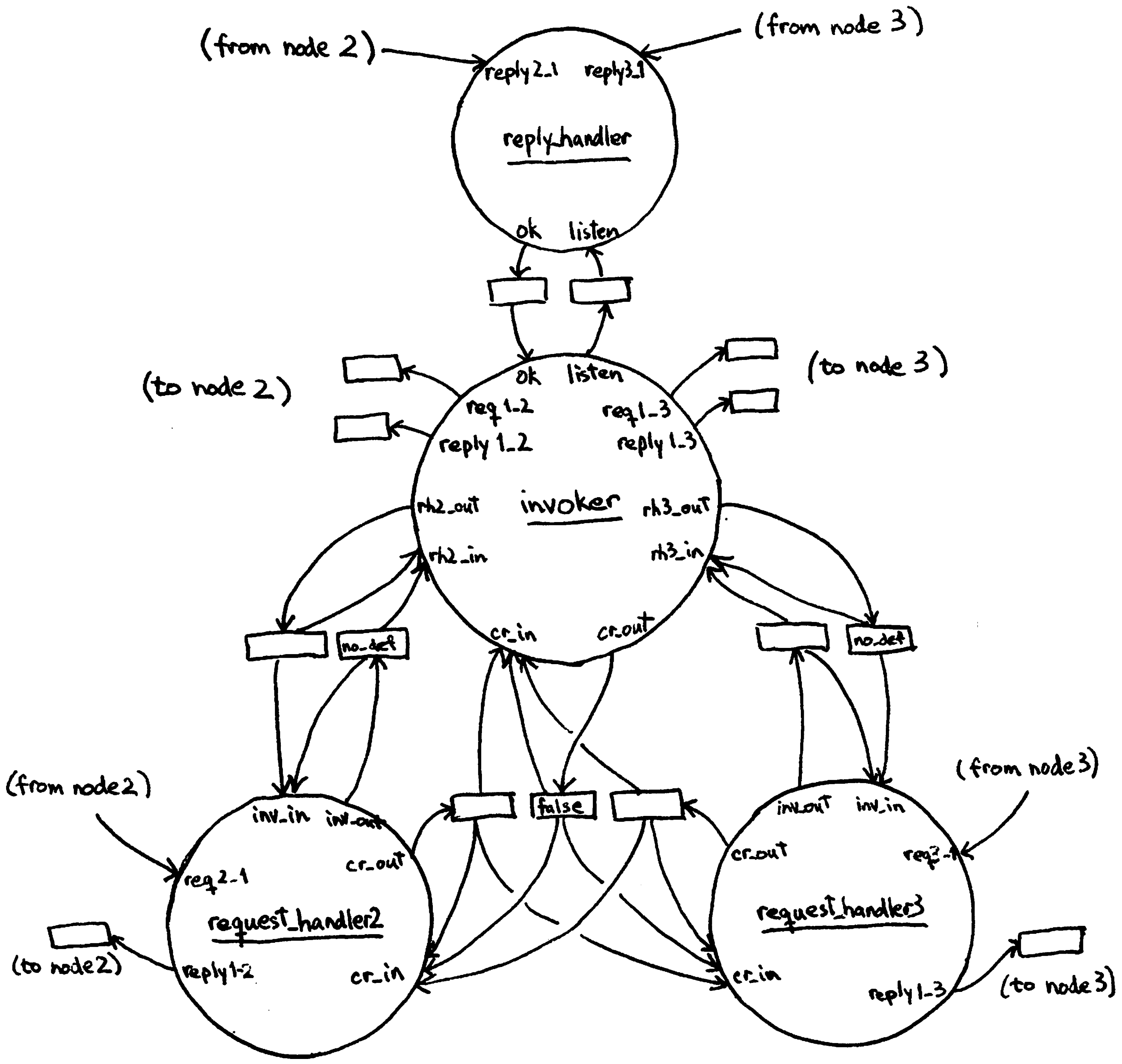
Reply_handler DYMOL program

Figure 2

```
DO FOREVER
   BEGIN
      RECEIVE req2_1;
      RECEIVE cr_in;
      SEND cr_out;
      IF BUFFER = true AND INTERNAL TEST THEN
         BEGIN
            RECEIVE inv_in;
            SET BUFFER := def
         END
      ELSE
         BEGIN
            SEND reply1_2;
            RECEIVE inv_in;
            SET BUFFER := no_def
         END
      SEND inv_out;
   END
```

Request_handler DYMOL program

Figure 3



Initial configuration of node 1

Figure 4

(Note: Figure will be redrawn for final manuscript.)

system. These linkages are shown in Figure 4. This figure also shows the messages that are assumed to be initially available through the communication linkages.

It must be emphasized that these DYMOL programs are not intended to be a complete description of all aspects of the node's activity. That is, although they have the form of programs they by no means represent an implementation of the processes that they describe. Instead, they should be viewed as a model offering only an incomplete and abstract description of the process' behavior. Here, in keeping with the designer's decision to concentrate on the distributed mutual exclusion aspect of the system, the DYMOL programs focus on just that aspect of the process' activity. Other aspects are represented in only the most abstract fashion or are omitted altogether. We feel that such selective description is both appropriate and necessary during early stages in the design of a complex, distributed software system.

As Figures 1, 2, and 3 suggest, DYMOL programs typically consist primarily of SEND and RECEIVE instructions. Each SEND has an outbound <u>port</u> (represented by an outbound arc in Figure 4) as its operand. Execution of a SEND by a process causes the contents of a distinguished memory location in that process, called the <u>buffer</u>, to be sent through the designated port to a <u>link</u> (represented by a box in Figure 4) associated with that port. The link is essentially an unbounded, unordered repository that is used to mediate the asynchronous message transmission activity of DPMS. Each RECEIVE has an inbound port as its operand. The specified inbound port may be connected to zero or more links by communication pathways called channels (represented by inbound arcs in Figure 4). Execution of a RECEIVE causes a message chosen (nondeterministically) from among the messages contained in the links to which the port is connected to be removed from the link and placed into the buffer of the process executing the RECEIVE. If no messages are currently available in any of the links to which its port is connected, the process executing the RECEIVE instruction is suspended until a message becomes available. The DYMOL SET instruction provides a means for placing a particular value, represented as the SET instruction's operand, into a buffer. Further details regarding the semantics of these and other DYMOL constructs may be found in [WILE80].

The DYMOL program representing the invoker process (Figure 1) consists of a nondeterministic (WHILE INTERNAL TEST) loop. This corresponds to the designer's view of this process' activity as it relates to mutual exclusion, namely that it will repeatedly attempt to enter its critical section, but may eventually decide to stop doing so. Each pass through the loop begins with the invoker's announcing its intention to enter the critical section. The invoker makes this announcement by replacing the currently available message in the links connected to cr_in inbound ports with the message "true", then sending a "true" message to the reply_handler via the listen port. (Since SEND does not destroy the contents of the buffer, no SET is needed after the SEND cr_out instruction.) After announcing its intention, the invoker process requests permission to use the shared resource by sending messages to each of the other nodes in the distributed system. The SET BUFFER := sequence_number instruction abstractly models the detailed internal processing that the invoker process uses in selecting the sequence number portion of its message. Such details are irrelevant at the current stage of the design development process, although they clearly must be addressed in later stages.

Having announced its intention to enter the critical section and having sent requests for use of the shared resource to the other nodes in the distributed system, the invoker awaits (at the RECEIVE ok instruction) a message from the reply_handler indicating that it can proceed. Upon receiving that message, the invoker performs its critical section processing, abstractly modelled in Figure 1 by the SET BUFFER := critical instruction. It then announces completion of its critical section processing by replacing the message currently available in the links connected to the cr_in inbound ports with a "false" message. Finally, it checks to see if any replies were deferred while it was performing critical section processing. If so, it dispatches the deferred replies and updates the appropriate link contents to indicate that no replies remain deferred. The invoker is then ready to repeat the instructions in its WHILE loop if it (nondeterministically) chooses to do so.

The DYMOL program representing the reply_handler process (Figure 2) consists of a non-terminating (DO FOREVER) loop. Upon being informed (via its listen port) that the invoker has requested use of the shared resource, the reply_handler awaits messages from the other nodes in the distributed system granting their permission for such use. When both other nodes have given their permission, the reply_handler so informs the invoker by sending a message through its ok port. Here, as in several other places in this design, the content of the message sent to the invoker is irrelevant, so a SET instruction replacing the present contents of the reply_handler's buffer is not used.

Figure 3 is the DYMOL program for one of the request_handler processes. The other request_handler's program is identical except for the replacement of port names req2_1 and reply1_2 by req3_1 and reply 1_3, respectively. The request_handler, upon receiving a request for use of the shared resource, checks the current status of the invoker process by obtaining the message currently available through its cr_in port. After returning this message so that it can be inspected by the other request_handler or updated by the invoker, the request_handler decides whether to send an immediate reply or to defer its reply. The decision depends in part upon the current status of the invoker (recall that SEND does not alter buffer contents) and in part upon a priority comparison, abstractly represented at this stage in the design as an INTERNAL TEST. If the invoker is attempting to enter its critical section and it has priority over the other requesting node, then the reply will be deferred. Otherwise, the reply is sent through the request_handler's reply1_2 port. In either case, an appropriate message is made available to the invoker indicating whether or not the reply was deferred.

The design description contained in Figures 1 through 4 represents a reasonable and realistic first step toward designing a distributed software system in which mutual exclusion plays an important role. In fact, this description is an accurate abstract version of the Ricart and Agrawala solution to the distributed mutual exclusion problem ([RICA81a]). Further iterative refinement steps would elaborate the design by detailing the priority determination used in the distributed mutual exclusion mechanism and gradually introducing other aspects of the overall function of the distributed software system.

Before proceeding with further elaboration steps, however, our hypothetical designer decides to first analyze the design as it currently stands. One objective of such an analysis is to uncover any errors made to this point so that they can be corrected now rather than being incorporated into later, more detailed versions of the design. Alternatively, this analysis effort may serve to increase the designer's confidence in various portions of the current design by demonstrating that they will produce appropriate patterns of system behavior.

Our designer chooses to analyze this design using the analysis techniques that we briefly described in section 2. As we mentioned there, these techniques focus on the patterns of behavior that are possible in a system whose design is described in DYMOL. For the purpose of analysis, we regard each behavior of the system as a string of symbols representing occurrences of events in the system, and we formulate questions about the system in terms of the appearance of particular patterns of symbols in these strings. For example, our designer would like to know that the messages in the links connected to port rh2_in are used by the invoker and request_handler2 processes in a mutually exclusive fashion. In terms of the appearance of patterns of symbols in behaviors, this mutual exclusion can be interpreted as the requirement that, between the symbols representing a RECEIVE from one of these links and the next SEND to one of these links from the same process, no symbol representing a use of these links can occur.

When the properties of the computational model underlying DPMS are interpreted in terms of the appearance of event symbols in behaviors, they yield conditions on the numbers and types of symbols preceding and following the appearance of a given symbol in a behavior of the system. We use systems of inequalities derived from these conditions to determine whether a particular pattern of symbols occurs in the behaviors of the system under study. We now briefly discuss the sorts of conditions that arise. A more detailed discussion of such conditions and their use in analysis appears in [AVRU81].

There are three types of conditions arising from the fundamental properties of DPMS that are relevant to the analysis of the distributed system design described above. The first type consists of conditions reflecting the requirement that each sequential process in the system proceed through its program in the correct order, halting only when it completes the program or suffers starvation.

The second type consists of conditions related to interprocess communication. These conditions reflect the requirements that messages can be exchanged only over previously established channels, that no messages can be received from an empty link, and that no process can wait indefinitely if there is a message it can receive.

The third type of condition relevant to the analysis of our example design consists of conditions related to branching, and insures that branching depends correctly on the contents of the buffer. It is important to note that conditions of the last two types provide the only constraints on the ordering of events from different processes.

To illustrate our analysis techniques, we will outline below part of the analysis which the designer might perform on the design specified by Figures 1 through 4. In the interest of clarity and brevity, we omit many details and numerous inequalities, relying primarily on prose instead. It should be borne in mind, however, that all of the analysis outlined below can be expressed in terms of solving systems of inequalities.

As a first step, the designer decides to check that the messages in the links connected to port rh2_in are used by the invoker and request_handler2 processes in the proper mutually exclusive fashion. As mentioned above, this is equivalent to checking that, in every behavior of the system, no symbol representing a use of these links can occur between the symbol representing a RECEIVE from one of these links and the next SEND to one of these links from the same process. Examining the DYMOL programs and using conditions of the first type, the designer can see that events RECEIVE rh2_in and SEND rh2_out strictly alternate (with other events interleaved, of course), always beginning with a RECEIVE rh2_in. Similarly, it is evident that the events RECEIVE inv_in and SEND inv_out strictly alternate, always beginning with a RECEIVE inv_in.

Let r(inv_rh2) be the symbol representing either of the events RECEIVE rh2_in or RECEIVE inv_in. Similarly, let s(inv_rh2) be the symbol representing either of the events SEND rh2_out or SEND inv_out. The condition that no message can be received from an empty link implies that, in any initial segment of a behavior, $|r(inv\_rh2)| \leq |s(inv\_rh2)| + 1$, where we use $|symbol|$ to denote the number of occurrences of the symbol in the (sub)string under consideration and the 1 accounts for the message initially in the link connected to inv_out (see Figure 4). The strict alternation of r(inv_rh2) and s(inv_rh2) from each process implies, however, that $|r(inv\_rh2)| \geq |s(inv\_rh2)|$. Hence, the designer concludes that the symbols r(inv_rh2) and s(inv_rh2) alternate strictly in the entire behavior.

Suppose a use of one of these links does occur between a particular r(inv_rh2) and the next s(inv_rh2) from the same process. By the alternation noted above, the next reference to one of these links following the given r(inv_rh2) must be an s(inv_rh2) from some other process. Consider now the behavior segment preceding the given r(inv_rh2). For each process P, the designer knows that $|s(inv\_rh2)|_P \leq |r(inv\_rh2)|_P$, where we denote by $|symbol|_P$ the number of occurrences of the symbol which represent events in process P. For the process in which the next s(inv_rh2) occurs it must be true that $|s(inv\_rh2)|_P < |r(inv\_rh2)|_P$, so it follows that $|s(inv\_rh2)| \leq |r(inv\_rh2)| - 1$. Immediately following the given r(inv_rh2), it would have to be the case that $|s(inv\_rh2)| < |r(inv\_rh2)| - 1$, which contradicts the first inequality noted above. Thus, no reference to one of these links can occur between an r(inv_rh2) and the next s(inv_rh2) from the same process, and the messages in the inv_rh2 links are used in a mutually exclusive fashion. Having shown this, the designer could apply essentially the same analysis to the links connected to the cr_in ports and also to the links connected to the rh3_in port.

The designer also wants to check that, under the assumption that each request from the given node eventually receives a reply, no request from another node is permanently deferred. He supposes to the contrary that a request from node 2 is permanently deferred. Examining the DYMOL programs and using conditions of the first type, he sees that this can occur only when the last message sent to a link connected to rh2_in is a "def" sent by request_handler2.

The conditions related to branching then imply that request_handler2's last RECEIVE cr_in must have received the message "true", so this RECEIVE cr_in must have occured between the events SET BUFFER := true, SEND cr_out and RECEIVE cr_in, SET BUFFER := false, SEND cr_out in the invoker. Thus, the conditions which insure that each sequential process proceeds through its program in the correct order imply that request_handler2's last RECEIVE cr_in is followed by at least one transmission of the message "no_def" into a link connected to the invoker's port rh2_in. Showing that this SEND event must occur <u>after</u> request_handler2's last SEND inv_out would contradict the designer's hypothesis and show that no request from node 2 can be permanently deferred.

In this situation, the designer examines the list of conditions of types 2 and 3, searching for a condition or chain of conditions constraining the ordering of these two SEND events. However, the only relevant conditions are those that imply the mutually exclusive use of the links connected to rh2_in, and these do not appear to prevent the invoker from receiving a message from one of these links and returning it between the time request_handler2 receives the "true" through port cr_in and receives the message through port inv_in. Unable to find constraints that would prevent this sequence of events, the designer looks for a behavior containing this sequence. In this case, it is not hard to find one.

In particular, it is easy to write down a behavior in which the request_handler executes its RECEIVE cr_in and SEND cr_out instructions between the execution of the SET BUFFER := critical instruction and the RECEIVE cr_in instruction by the invoker. If this behavior continues with the invoker executing its SET BUFFER := false, SEND cr_out, RECEIVE rh2_in, and (after skipping past the conditional because it received a "no_def" message though rh2_in) SEND rh2_out instructions before request_handler executes a RECEIVE inv_in instruction, the possibility of a permanently deferred reply will have arisen. Because the request_handler got a "true" message through port cr_in it can (assuming that the nondeterministic INTERNAL TEST evaluates to true) eventually execute its SET BUFFER := def and SEND inv_out instructions. If the invoker now decides to exit from its WHILE loop, that "def" message will never be received by the invoker and thus a reply will be permanently deferred.

The design error that has just been revealed by the designer's analysis is rather subtle. Indeed, essentially this same error appeared in the first published version of the Ricart and Agrawala algorithm ([RICA81a]), necessitating the publication of a revised version a few months later ([RICA81b]). The problem is that, although each message is used in a proper, mutually exclusive fashion (as the designer's previous analysis had demonstrated), it is possible for the request_handler to inspect one message and use that information in deciding what information to send in a subsequent message, but not manage to send that second message until the invoker has already invalidated the information used in making the decision and inspected an outdated, erroneous version of the message that the request_handler is about to replace. Our experience indicates that subtle errors of this kind, which are very difficult to discover by simply studying the programs for a distributed system, are generally uncovered with surprising ease using these analysis techniques.

After discovering this error, the designer modifies the system to eliminate it, and uses the same sort of analysis to check that the modification does indeed eliminate the error. A simple modification that appears to eliminate the problem is to change the request_handler's DYMOL program so that the request_handler removes the message available through its inv_in port as soon as it receives a request. The new DYMOL program for request_handler2 is given in Figure 5.

```
DO FOREVER
   RECEIVE req2_1;
   RECEIVE inv_in;
   RECEIVE cr_in;
   SEND cr_out;
   IF BUFFER = true AND INTERNAL TEST THEN
      SET BUFFER := def
   ELSE
      BEGIN
       SEND reply 1_2;
       SET BUFFER := no_def
      END
   SEND inv_out
END
```

Revised request_handler DYMOL program

Figure 5.

To check that this design modification eliminates the problem, the designer supposes again that a request from node 2 is permanently deferred. As before, it is easy to see that the last message sent to a link connected to rh2_in is a "def" sent by request_handler2. Therefore, the last s(inv_rh2) from the invoker must precede this last s(inv_rh2) from request_handler2. The analysis that showed mutually exclusive use of the two links connected to rh2_in still applies, so the last r(inv_rh2) - s(inv_rh2) pair must precede request_handler2's last r(inv_rh2) - s(inv_rh2) pair. But, as before, the branching conditions and the fact that request_handler2 sent the message "def" imply that request_handler2's last RECEIVE cr_in must precede the invoker's last SEND cr_out, and conditions of the first type imply that this SEND must precede the invoker's last r(inv_rh2). Finally, the same sort of conditions imply that request_handler2's last r(inv_rh2) precedes its last RECEIVE cr_in and therefore precedes the invoker's last r(inv_rh2). This is a contradiction, so the request from node 2 cannot be permanently deferred.

5. Conclusion

In this paper we have outlined an approach to describing and analyzing designs for distributed software systems. A descriptive notation has been introduced and analysis techniques applicable to designs expressed in that notation have been presented. We have given an example of the application of this approach to a realistic distributed software design problem. In the example, application of the analysis techniques to a design description makes it possible to uncover a subtle design error at a very early stage in the design development process. This permits the designer to repair the error, and subsequently to demonstrate that the repaired design is sound, before proceeding with refinement of the design.

We have given a prose description of the analysis performed on our example distributed software design. The analysis that we were describing, however, can all be expressed in terms of the consistency or inconsistency of systems of inequalities. We therefore believe that many aspects of this analysis can be automated. Although our preliminary efforts in this direction have encountered problems due to combinatorial explosion, we are currently investigating several approaches for turning our analysis techniques into tools suitable for inclusion in a software development environment. We note also that the DYMOL design notation presented in this paper is only a research vehicle at present. Improved syntax and additional constructs would be desirable in any design notation intended for practical use.

We believe that the approach outlined in this paper provides a basis for tools that will be extremely useful to distributed software system developers. In particular, this approach is well-suited for use in a systematic, iterative refinement style of distributed software system development. Our approach facilitates production of the incomplete and abstract descriptions that are appropriate during early stages of the development process. Moreover, it provides a means for rigorously analyzing these incomplete and abstract descriptions. Thus, it offers the prospect of a development process guided, from its earliest stages, by continual assessment of the evolving design. Such a carefully guided development process could dramatically increase the productivity of distributed software system developers.

References

[AVRU81] G. Avrunin and J. Wileden, "An Introduction to Algebraic Techniques for the Analysis of Concurrent Systems," COINS Technical Note, University of Massachusetts, (May 1981).

[BRIN78] P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," _Communications of the ACM_, (November 1978), pp.934-941.

[CLAR81] L. Clarke, R. Graham, and J. Wileden, "Thoughts on the Design Phase of an Integrated Software Development Environment," _Proceedings of the 14th Hawaii International Conference on Systems Science_, Honolulu (January 1981).

[DOD80] United States Department of Defense, _Reference Manual for the Ada Programming Language_, (July 1980).

[HABE72] A.N. Habermann, "Synchronization of Communicating Processes," _Communications of the ACM_ (March 1972), pp.171-176.

[HOAR69] C.A.R. Hoare, "An Axiomatic Basis of Computer Programming," _Communications of the ACM_ (October 1969), pp.576-580, 583.

[LAMP74] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," _Communications of the ACM_, (August 1974), pp.453-455.

[LAMP78] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," _Communications of the ACM_ (July 1978), pp.558-565.

[LAUE75] P.E. Lauer and R.H. Campbell, "Formal Semantics for a Class of High Level Primitives for Coordinating Concurrent Processes," _Acta Informatica_, (1975), pp.247-332.

[LAUE79] P.E. Lauer, P.R. Torrigiani, and M.W. Shields, "COSY: A System Specification Language Based on Paths and Processes," _Acta Informatica_, (1979), pp.451-503.

[LISK79] B. Liskov, "Primitives for Distributed Computing," _Proceedings of the Seventh Symposium on Operating Systems Principles_, (December 1979), pp.33-42.

[PNUE79] A. Pnueli, "The Temporal Semantics of Concurrent Programs," in Khan, (ed.) _Semantics of Concurrent Computation_, Springer-Verlag, (1979), pp.1-20.

[RICA81a] G. Ricart and A.K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," Communications of the ACM, (January 1981), pp.9-17.

[RICA81b] G. Ricart and A.K. Agrawala, "Corrigendum," Communications of the ACM, (September 1981), p.578.

[RIDD78] W. Riddle, J. Wileden, J. Sayler, A. Segal, and A. Stavely, "Behavior Modelling During Software Design," IEEE Transactions on Software Engineering, (July 1978), pp.283-292.

[RIDD79] W. Riddle, "An Approach to Software System Modelling and Analysis," Journal of Computer Languages (1979), pp.49-66.

[WILE78] J. Wileden, "Modelling Parallel Systems with Dynamic Structure," COINS Technical Report 78-4, University of Massachusetts, (January 1978).

[WILE80] J. Wileden, "Techniques for Modelling Parallel Systems with Dynamic Structure," Journal of Digital Systems, 4,2 (Summer 1980), pp.177-197.