Describing and Analyzing Distributed Software System Designs

GEORGE S. AVRUNIN and JACK C. WILEDEN
University of Massachusetts

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.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.4 [**Software Engineering**]: Program Verification; D.3.2 [**Programming Languages**]: Language Classifications; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms: Design, Languages, Theory, Verification

Additional Key Words and Phrases: Analysis of software design, design notation, distributed mutual exclusion, distributed software systems, software design tools.

1. INTRODUCTION

Motivated by the increasing demand for highly complex, yet highly reliable, distributed computing systems, we have been investigating tools and techniques to aid in the preimplementation stages of distributed software system development. In this paper we present a notation appropriate for describing designs of distributed software systems, and techniques for analyzing the behavior of systems whose designs are expressed in this notation. The notation describes systems as collections of sequential processes communicating entirely via message transmission, and hence is well suited for use in developing the design for distributed system software. 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, these techniques can also be used to rigorously demonstrate that certain aspects of a design are correct.

This work was supported in part by National Aeronautics and Space Administration grant NAG 1-115.

Authors' addresses: G. S. Avrunin, Dept. of Mathematics and Statistics, University of Massachusetts, Amherst, MA 01003; J. C. Wileden, Dept. of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.

ACM Transactions on Programming Languages and Systems, Vol. 7, No. 3, July 1985, Pages 380–403

The role that we envision for these techniques in preimplementation distributed software development is illustrated by the following scenario: A designer—early in the development of a large, complex, distributed software system—conceives a modularization for the system. The designer can then use our notation to describe this modularization, identifying the individual processes comprising the system and specifying how those processes will interact. Continued development of the system, eventually culminating in an implementation, will involve a great deal of time and effort, much of which would be wasted if any error were made at this early preimplementation stage. Therefore, before proceeding with the development, the designer employs our analysis techniques, expressly tailored for preimplementation use, to check for design flaws. Specifically, these techniques can be used to determine whether or not certain patterns of behavior occur, given the specified processes and process interactions. The patterns of interest may represent desirable properties of system behavior, such as mutually exclusive utilization of a shared resource, or graceful degradation and continued operation following the failure of one or more system components. Alternatively, the patterns might represent pathological behaviors such as deadlocks. Through use of these analysis techniques, the designer could gain confidence in the suitability of a design before proceeding to later stages of the software development process.

Although this paper focuses on the design stage of distributed software system development, it is worth noting that, with appropriate modifications, the analysis techniques presented here could be applied during later stages as well. Such generalized usage of these techniques would contribute to uniformity of analysis methods, and hence to increased integration of the development process for distributed software systems. After restricting our attention to the design stage in most of what follows, we return in the paper's concluding section to a consideration of the broader applicability of our approach to distributed system description and analysis.

We describe both our notational framework and our approach to analysis, illustrating their use and usefulness with a realistic example. The next section discusses our design notation framework, and compares our approach with some related work by other researchers. This is followed by a short section in which we discuss the distributed mutual exclusion problem that serves as the basis of our example. Section 4 presents the first part of the example in which we illustrate the use of our notation in developing a design for a distributed system. In Section 5 we describe the analysis techniques that we have developed, specifically showing how they can be applied to designs expressed in the notation introduced in Section 2. We continue our example in Section 6, showing how our analysis techniques can be used both to uncover design errors and to demonstrate the 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 A DESIGN NOTATION

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 preimplementation 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 is also appropriate for use by developers of distributed software 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 so that it can be analyzed.

In addition to providing abstraction and rigor, we feel that a preimplementation 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 a distributed software development environment [5]. Finally, a formalism can only be appropriate for general use in designing distributed software 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 Modeling Scheme (DPMS) and its Dynamic Modeling Language (DYMOL). This formalism, described in detail in [35], was originally developed for studying distributed systems with dynamic structure [33]. It evolved from the PPML formalism [31] that served as the foundation for the DREAM software development system [30, 34]. One component of DPMS is a modeling language, called DYMOL, that can be used to formulate precise, high-level, procedural descriptions of constituent processes in a distributed system [35]. A second component of the modeling scheme, called *constrained expressions* [6, 36], is a closed-form, nonprocedural, representation for all the possible behaviors that could be realized by some distributed system. For an important subset of dynamically structured distributed systems, these two components of DPMS are related by an effective procedure for deriving the constrained expressions describing the potential behavior of a given system described in DYMOL. In the remainder of this section we summarize the relevant features of the Dynamic Process Modeling Scheme. We first describe the computational model on which DPMS is based, then discuss the DYMOL language and relate it to other languages for describing distributed systems.

In DPMS, a dynamically structured distributed system is considered to be composed of individual sequential processes, communicating with one another by means of message transmission. Each individual process is an instance of a class of potential processes. Each class is described by a *template* (i.e., a generic program written in DYMOL). This DYMOL template precisely specifies the ways in which processes of the class may interact with other processes, through (asynchronous) message transmission or by creating or destroying processes, but only abstractly describes the local, internal activities of the process itself. Thus, DPMS descriptions focus on process organization and interaction, which is the appropriate orientation for design description and analysis, rather than on internal process activity.

Message transmission as modeled in DPMS is both a communication and a synchronization mechanism. A process may, by using a DYMOL instruction whose syntax is

SEND ⟨port_id⟩,

send a message through an outbound *port* into a *link* associated with that port. Each process template completely (and, in the present version of DYMOL, implicitly) specifies the set of distinct inbound and outbound ports through which processes of the given process class may communicate with other processes in the system. Associated (again implicitly) with each outbound port is a link, which is essentially an unbounded, unordered repository that is used to mediate the asynchronous message transmission activity of DPMS processes. Performing a SEND operation may be viewed as copying the current contents of the process' *buffer* (a distinguished memory location within the process) into the designated link, leaving the buffer's contents unchanged (a nondestructive copy operation). The buffer's contents may be modified by receipt of a message (via the RECEIVE instruction described below) or by using the buffer assignment instruction, whose syntax is

SET BUFFER := ⟨message_type⟩.

Having sent a message, the sending process may continue with subsequent activities as described by its DYMOL program.

Using a DYMOL instruction whose syntax is

RECEIVE ⟨port_id⟩

a process may request receipt of a message through one of its inbound ports. Such a request can be fulfilled whenever at least one link containing one or more messages is connected to the designated inbound port by an interprocess communication *channel.* When the request is fulfilled, the following steps are followed: First, one link is nondeterministically selected from among those that contain one or more messages and are connected to the designated port by channels. Then, one message is chosen, again nondeterministically, from those residing in the selected link. Finally, this message is removed from the link and placed into the buffer of the requesting process. If no messages are currently residing in any of the links currently connected to the designated inbound port when a receive request is lodged, the requesting process simply waits. The wait continues at least until a message becomes available in a link connected to the designated inbound port, or until a link containing a message is connected to the designated port by a newly established channel. (Both of these must obviously result from activities of processes other than the waiting process.) Neither the appearance of a message nor the opening of a channel will necessarily end a wait, however, since competing requests might be lodged in the interim and requests need not be serviced in the order in which they were made. Clearly, a process could wait for receipt of a message indefinitely.

In DPMS, the structure of a dynamically structured distributed system can be altered either by adding or deleting processes or by adding or deleting interprocess communication channels. Since the dynamic structure aspects of DPMS are not used in this paper, we omit syntactic and other details of the corresponding DYMOL instructions, but include a brief treatment of them for completeness.

The DYMOL instruction CREATE causes a new process of a specified class, that is, a new instance of the process class described by a specified process template, to be added to the system and assigned a unique name. The DESTROY instruction is used to remove a particular process, specified by its unique name, from the system. The DYMOL instruction ESTABLISH causes two specified processes to become connected by a channel, which is simply a communication pathway. In fact, the ESTABLISH instruction designates the specific outbound port (and hence its associated link) of a specific process and the specific inbound port of another (possibly the same) specific process that are to be connected. Similarly, the CLOSE instruction is used to disconnect two specified ports by removing the channel between them. Since the SEND instruction only designates an outbound port through which a message is to be sent, and not a particular recipient, the establishing and closing of ports provides the only means of controlling message routing in DPMS. Finally, DPMS provides methods by which a designer can stipulate an initial configuration for a dynamically structured distributed system. In this paper, for simplicity, we specify the initial configuration of our DYMOL-described systems by using diagrams. Since our examples do not require dynamic structure, none of the instructions discussed in this paragraph are used in this paper, and the diagrams represent not just the initial configuration but also the permanent process structure and interprocess connectivity of the systems appearing in the examples.

The DYMOL language is a simple programming-like language whose syntax is based on Algol 60. In addition to the instructions for message transmission (SET, SEND, and RECEIVE), communication channel manipulation (ESTABLISH and CLOSE), and process creation and destruction (CREATE and DESTROY), discussed above, it provides a standard set of control-flow constructs. Branching within a DYMOL program can be based either on communications from other processes, represented by the current contents of the process' buffer, or on purely internal process computations. Branching decisions based upon internal process computation are modeled as nondeterministic choices (e.g., IF INTERNAL TEST . . . or WHILE INTERNAL TEST DO . . .). Examples of DYMOL descriptions appear in Section 4 of this paper, while further details on DYMOL can be found in [35].

Since DYMOL bears a strong resemblance to a programming language, DPMS models have a natural relationship to standard software specification and design techniques and are easy for system developers to understand. Because its primitives are message transmission and the creation and destruction of processes, DPMS is suitable for describing a wide range of distributed system organizations. DPMS focuses on process organization and interaction, and therefore addresses precisely those issues most crucial during specification and design. For these reasons we believe that the Dynamic Process Modeling Scheme is an appropriate basis for both describing and analyzing designs of distributed software systems.

The DPMS descriptive formalism on which we are basing our work is similar to several other approaches to describing distributed systems. It most closely resembles the DDN design language of the DREAM software design system [30, 34], which can be considered its predecessor. It also resembles the numerous other languages, such as PLITS [9], that use buffered (or asynchronous) message transmission as their principal interprocess communication and synchronization mechanism. DPMS differs from these other approaches primarily in its ability to describe dynamically structured distributed systems, a capability not illustrated in this paper,[1] and its close relationship to design-oriented analysis techniques.

Of course, viewing a system as a collection of communicating sequential processes is common to many description schemes. Hoare's Communicating Sequential Processes [13], Brinch Hansen's Distributed Processes [3], and the tasking facility in the Ada programming language [8] are three of the better known examples of descriptive approaches that take this view. Unlike DPMS, DDN, and PLITS, however, all three of these approaches (and many similar ones) employ an interprocess communication protocol in which information is transferred only when both the sender and the receiver are simultaneously prepared to communicate. We find buffered message transmission a more natural descriptive medium; it is easier to use, especially in formulating the high-level, abstract descriptions appropriate during the design stage of distributed system development. Thus, although it has been repeatedly pointed out that each style of communication can be used to describe the other with minimal difficulty (e.g., [13, 23]), we have chosen to base our initial development of analysis techniques on the DPMS descriptive scheme and its buffered communication constructs.

Alternative approaches that are explicitly intended for use in preimplementation development of distributed systems, and that provide for some analysis during that process, include the COSY formalism [22] and the distributed system specification technique of Ramamritham and Keller [27]. Both are based on formal semantic models, the former on the theory of nets and path expressions [21] and the latter on temporal logic [26]. Both also describe distributed systems as collections of communicating sequential processes. These two approaches both employ a descriptive style in which the behavior of individual processes is specified and then added constraints are imposed to limit process interaction. The resulting description seems further from a programmed solution, and hence less natural for the design stage of a distributed software system's development, than a DPMS description. Nevertheless, these two approaches are potentially very useful, especially if employed in conjunction with a language such as Path

---

[1] See [35] or [36] for illustrations of DPMS' application to dynamically structured systems.

Pascal [4] that supports the same style of description at the implementation level.

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 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 [19, 20] and by Ricart and Agrawala [28, 29], 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" [17].

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 outlined above. The remainder of this section describes the first stage in 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 is 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 ([28]).

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 (specifically node 1) 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 system. These linkages are shown in Figure 4, where processes are depicted as labeled circles, inbound and outbound ports are represented by inbound and outbound arcs, respectively, and links are represented by boxes. This figure also shows the messages assumed to be initially available through communication linkages. A character string such as "no_def", inside a link, represents an available message, while an empty box indicates that no message currently resides in that link.

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 they describe. Instead, they should be viewed as a model offering only an incomplete and abstract description of the behavior of the processes.

```
INVOKER:
 IN1:   WHILE INTERNAL TEST DO
          BEGIN
 IN2:       RECEIVE get_status;     --announce intention to
                                    --enter critical section
 IN3:       SET BUFFER := true;
 IN4:       SEND put_status;
 IN5:       SEND listen;
 IN6:       SET BUFFER := sequence_number;
 IN7:       SEND ask_2;             --now send requests
 IN8:       SEND ask_3;             --. . . to other nodes
 IN9:       RECEIVE ok;             --permission to enter
                                    --critical section granted
 IN10:      SET BUFFER := critical;     --enter critical section
 IN11:      RECEIVE get_status;     --announce exit from
 IN12:      SET BUFFER := false;    -- . . . critical section
 IN13:      SEND put_status;
 IN14:      RECEIVE from_rq2;       --deal with any deferred
                                    --requests from other nodes
 IN15:      IF BUFFER = def THEN
              BEGIN
 IN16:          SET BUFFER := true;
 IN17:          SEND resp_2;
 IN18:          SET BUFFER := no_def
              END
 IN19:      SEND to_rq2;
 IN20:      RECEIVE from_rq3;
 IN21:      IF BUFFER = def THEN
              BEGIN
 IN22:          SET BUFFER := true;
 IN23:          SEND resp_3;
 IN24:          SET BUFFER := no_def
              END
 IN25:      SEND to_rq3
          END
```

Fig. 1. Invoker DYMOL program.

```
REPLY HANDLER:
 RP1:   DO FOREVER
          BEGIN
          RECEIVE get_reps;      --invoker has sent requests
                                 --to other nodes; get replies
 RP3:     RECEIVE reply_2;
 RP4:     RECEIVE reply_3;
 RP5:     SEND got_reps
        END
```

Fig. 2. Reply_handler DYMOL program.

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. 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.

```
REQUEST_HANDLER_1_2:
  RQ1:     DO FOREVER
             BEGIN
  RQ2:         RECEIVE req_2;            --receive request from node 2
  RQ3:         RECEIVE 2_status_in;      --check status of invoker
  RQ4:         SEND 2_status_out;
  RQ5:         IF BUFFER = true AND INTERNAL TEST THEN
                 BEGIN
  RQ6:             RECEIVE 2_from_inv;      --defer reply
  RQ7:             SET BUFFER := def
                 END
               ELSE
                 BEGIN
  RQ8:             SEND resp_to_2;      --send reply
  RQ9:             RECEIVE 2_from_inv;
  RQ10:            SET BUFFER := no_def
                 END
  RQ11:        SEND 2_to_inv
             END
```

Fig. 3. Request_handler DYMOL program.

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 (statements IN2 through IN5). The invoker makes this announcement by replacing the currently available message in the links connected to the get_status inbound port (which are also connected to the inbound ports 2_status_in and 3_status_in) 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 put_status" 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 (statements IN6 to IN8). The "SET BUFFER := sequence_number" instruction (IN6) 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 statement IN9) a message from the reply_handler indicating that it can proceed. Upon receiving that message, the invoker performs its critical section processing, abstractly modeled in Figure 1 by the "SET BUFFER := critical" instruction (IN10). It then announces completion of its critical section processing by replacing the message currently available in the links connected to the get_status (and 2_status_in and 3_status_in) inbound port with a "false" message (IN11 to IN13). Finally, the invoker checks to see if

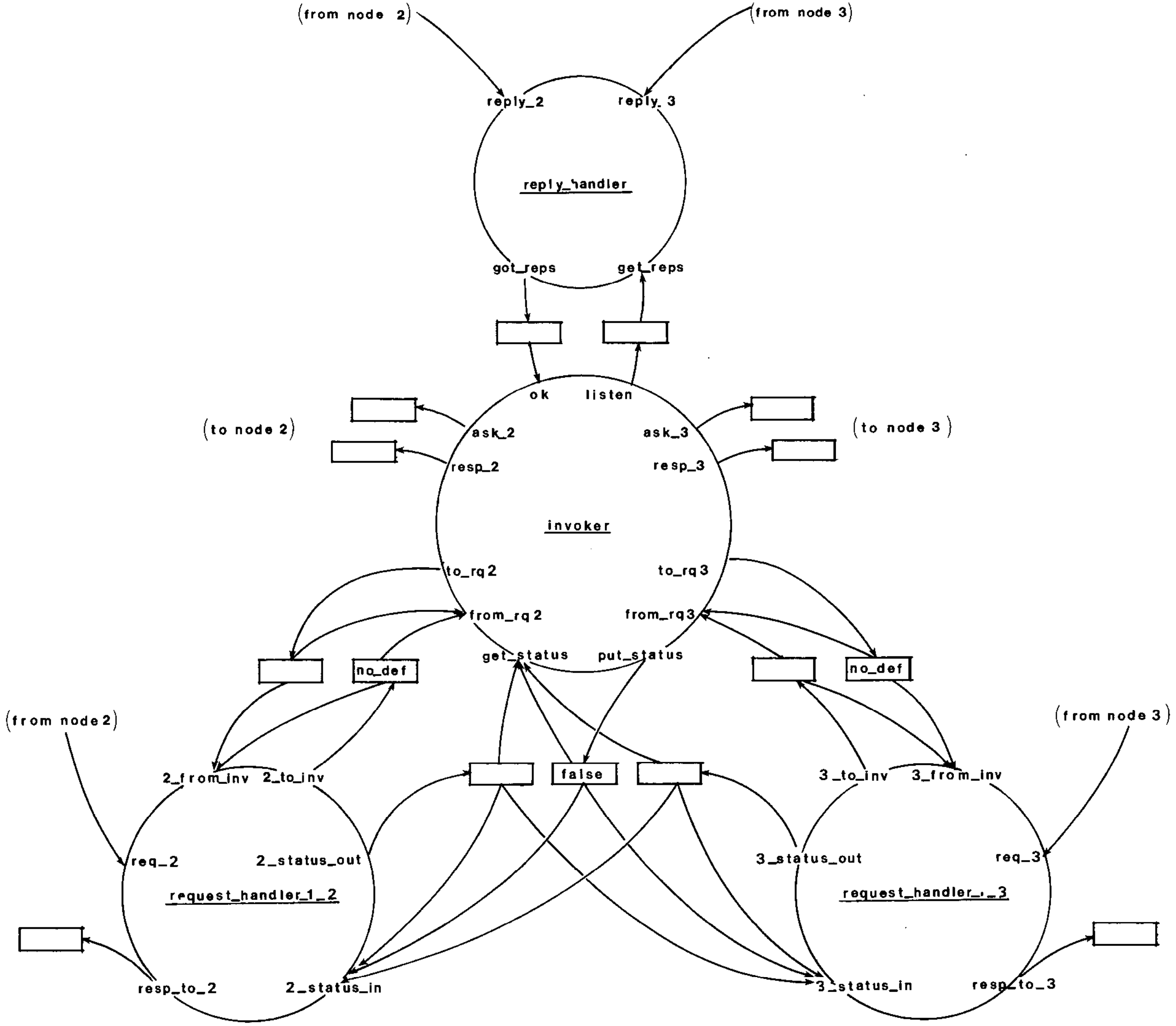

Fig. 4. Initial configuration of node 1.

any replies were deferred while it was performing critical section processing. The current contents of the links attached to ports from_rq2 and from _rq3 indicate whether request_handler_1_2 or request_handler_1_3, respectively, has deferred a request. The invoker inspects the contents of these links (at IN14 and IN15 and IN20 and IN21, respectively), returning the "no_def" message to the link immediately (IN19 and/or IN25) if it finds that no request is deferred. Should the invoker find a "def" message, indicating that a request has been deferred in either link, it dispatches the deferred replies (IN16 and IN17 and/or IN22 and IN23) and updates the appropriate link contents (IN18 and IN19 and/ or IN24 and IN25) to indicate that no replies remain deferred. The invoker is then ready to repeat the instructions in its WHILE loop if it chooses to do so.

The DYMOL program representing the reply_handler process for node 1 (Figure 2) consists of a nonterminating (DO FOREVER) loop. Upon being informed (via its get_reps 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 got_reps port. Here, since the message ("true") received as a reply at RP4 will serve as an appropriate message for the invoker, 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 two request_handler processes in node 1. The program for the other request_handler in node 1 (i.e., request_handler_1_3) is identical except for the replacement of port names containing 2's with port names containing 3's. The request_handler shown in Figure 3 monitors port req_2 (at RQ2) awaiting a request from node 2 for use of the shared resource. Upon receiving such a request, the request_handler checks the current status of the invoker process by obtaining the message currently available through its 2_status_in port (at RQ3). After returning this message (at RQ4) 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 (made at statement RQ5) 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 (RQ6 and RQ7). Otherwise, the reply is sent through the request_handler's resp_to_2 port (RQ8). In either case, an appropriate message (composed at RQ6 and RQ7 or RQ9 and RQ10) is made available to the invoker indicating whether or not the reply was deferred (RQ11).

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 [28]. 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. The next section describes the analysis techniques that we have developed for use with our design notation. Section 6 illustrates these techniques by applying them to the example design that we have developed in this section.

5. AN APPROACH TO ANALYSIS

For the purpose of analysis, we regard the possible behaviors of a system modeled in DPMS as a set of strings of symbols representing events involving the internal computations of the component processes of the system and the transmission of messages between those processes. This view resembles the "trace" perspective used, for example, by Hoare in studying the semantics of CSP [14, 15]. In our setting, the events of interest include those involving the execution of a statement in the DYMOL program of some process in the system and the normal termination or starvation of such a process. To analyze a design for a distributed system expressed in DPMS, we determine whether a particular symbol, or pattern of symbols, appears in a string representing a possible behavior of the system. The symbols in question may correspond to some desirable property of the system, such as graceful degradation, or may represent a pathology, such as deadlock.

Our analysis techniques begin with a collection of rules which are used to iteratively generate inequalities involving the numbers of occurrences of particular symbols that can appear in various segments of a string representing an actual behavior of the system. These rules are based on the underlying semantics of DPMS, on the description of the given system in DPMS, and on the particular symbols in question. If the assumption that a certain pattern of symbols occurs in such a behavior string leads, at any stage of the iterative process, to an inconsistent system of inequalities, we have reached a contradiction. We may then conclude that our assumption is incorrect and that the given pattern does not occur in a behavior. Otherwise, we continue to generate inequalities until we have enough information to construct a behavior containing the given pattern.

Our approach can be viewed as a generalization of the technique employed by Habermann [11] in analyzing a semaphore solution to a producer–consumer problem. Other related approaches applied to different aspects of the problem of analyzing distributed software systems include Taylor's method for static analysis of Ada programs [32] and Holzmann's technique for protocol validation [16]. Numerous researchers have investigated the alternative of using proof techniques [10] and proof rules [12] for establishing properties of distributed software systems (e.g., [1, 18, 20, 24, 25]). While all these approaches have their strengths and weaknesses, we believe that the approach described in this paper is particularly promising as a practical tool for use in the design of realistic, full-scale, distributed software systems.

The rules we use to generate the systems of inequalities for analysis fall into three general classes. The first of these classes consists of rules that reflect the sequential nature of each of the component processes of the system. The rules in the second class are based on the message transmission protocol of DPMS. The third class reflects the dependence of branching on buffer contents, and so involves both the flow of control in the individual processes and the communication between processes. In the remainder of this section, we describe these rules and the way they lead to inequalities. We have chosen to present the rules in a somewhat informal fashion. A fully formal description would involve the introduction of a great deal of notation and the substitution of several mathematical statements for each of the rules stated here. Although the full formality is necessary to automate the analysis, the discussion given here more closely resembles the way that a human would use the techniques without automated assistance, and is therefore much easier to read and understand. Nevertheless, the discussion has been kept sufficiently formal to indicate the rigor of our approach.

The rules in the first class impose the requirements that, in an actual behavior of the system, statements from each individual process are executed in the correct order and that a process can halt only by terminating normally or by starving. We list the rules below, and indicate the types of inequalities they generate. We use the symbol $r(p, q, m)$ to represent the receipt of message m through inbound port q from the link associated with outbound port p, and the symbol $s(p, m)$ to represent the transmission of message m through outbound port p to its associated link.

Rule I.1. Statements in a given process are executed in order, as specified by DYMOL control constructs.

This rule implies, for example, that statement RQ3 in the request_handler_1_2 process is executed only after statement RQ2 in each pass through the loop. We state this conclusion in terms of the number of occurrences of symbols in a behavior string as

$$|r(*, \text{2\_status\_in}, *)| \le |r(*, \text{req\_2},*)| \le |r(*, \text{2\_status\_in}, *)| + 1$$

where we use | symbol | to denote the number of occurrences of "symbol" in the string under consideration, and the asterisks (*) are a shorthand indicating "don't care" with respect to links and message types. Equality on the left holds in any initial segment of a behavior containing an $r(*, \text{req\_2}, *)$ not followed by an $r(*, \text{2\_status\_in}, *)$, while equality holds on the right in any initial segment of a behavior containing an $r(*, \text{2\_status\_in}, *)$ not followed by an $r(*, \text{req\_2}, *)$.

Rule I.2. Once a process halts, either by normal termination or starvation, no further events from that process occur.

For a fixed behavior of the system, this rule implies that the number of occurrences of a symbol representing an event in a given process is the same for all initial segments of the behavior that contain a termination or starvation symbol for that process.

Rule I.3. A complete behavior includes exactly one termination or starvation symbol for each process in the system.

This rule implies that each process must continue to execute statements until it starves or terminates normally. This can be used to show that certain events must occur. For example, every execution of statement RQ3 in the request_handler_1_2 program must eventually be followed by an execution of statement RQ4. (Note that this uses Rule I.1 as well.) This can be stated as an inequality in a number of ways. Perhaps the simplest is

$|s(\text{2\_status\_out}, *)|_{\text{seg}} < |s(\text{2\_status\_out}, *)|$

where $|s(\text{2\_status\_out}, *)|_{\text{seg}}$ denotes the number of occurrences of the symbol $s(\text{2\_status\_out}, *)$ in any initial segment ending with an $r(*, \text{2\_status\_in}, *)$ and $|s(\text{2\_status\_out}, *)|$ denotes the number of occurrences of $s(\text{2\_status\_out}, *)$ in the whole behavior.

We remark here that, while an automated analysis would presumably rely entirely on the systems of inequalities, it is often convenient to formulate some steps in the analysis verbally, using the mathematics only when it is necessary. Thus, we usually simply say that an $s(\text{2\_status\_out}, *)$ must occur after each $r(*, \text{2\_status\_in}, *)$ in a behavior, without writing down an inequality.

The rules in the second class are based on the message transmission protocol of DPMS. The first of these is

Rule II.1. In order for a message from the link associated with outbound port p to be received at inbound port q, there must be a channel connecting the link and port q and there must be a message available in the link.

This rule leads to inequalities of the form

$$|r(p, q, *)| \leq |s(p, *)| + n_p - \sum_{q' \neq q} |r(p, q', *)|$$

where n_p is the number of messages in the link associated with p at the start of the behavior. This says that the number of messages from p received at q is less than or equal to the number of messages sent through p plus the number of messages initially in the link associated with p minus the number of messages received from p at ports other than q.

Rule II.2. In order for a process to starve while waiting to receive a message at port q, there must be no messages available in links connected to q at the time the process last reaches a "RECEIVE q" instruction and also at the end of the behavior.

We regard this last attempt to receive at q, which results in the process waiting forever, as the starvation event, and represent it by the symbol $w(q)$. Note that messages which could be received at port q might become available after the last attempt to receive at q. The rule asserts that if the process starves while waiting to receive at q, all these messages must be received by other processes before the end of the behavior. This rule gives equalities of the form

$$|r(*, q, *)| = \sum (|s(p, *)| + n_p) - \sum |r(p, q', *)|$$

where the first summation ranges over the outbound ports p connected to q, n_p is again the number of messages initially available in the link associated with p, and the second summation ranges over the pairs (p, q') with p an outbound port connected to q, and q' an inbound port connected to p. The rule asserts that such an equality must hold for any initial segment of a behavior ending with the symbol $w(q)$ as well as for any complete behavior containing a $w(q)$.

The third class of rules involves both the flow of control in the individual processes and the messages transmitted between those processes.

Rule III.1. Branching depends, correctly, on buffer contents.

This rule implies that certain events must be preceded by the placing of particular messages in a process's buffer. For example, if statement IN16 of the invoker DYMOL program is executed in a behavior of the system, the message received at the immediately preceding execution of statement IN14 must have been "def". Thus we know that the r(*, from_rq2, *) symbol representing that execution of IN14 must have been an r(*, from_rq2, def). This rule is used not to directly generate inequalities but to provide additional information to guide the process of producing the inequalities.

6. ANALYSIS OF AN EXAMPLE DESIGN

In this section we illustrate the analysis technique based on the rules and associated inequalities we have just described by applying it to the design of Section 4. This example shows how our analysis technique can be used both to detect errors in a design and to establish that a proposed system will function as intended.

The design described in Section 4 uses the message in the links connected to the port from_rq2 to inform the invoker process when the request_handler_1_2 process has deferred a reply. When the invoker completes its critical section processing, it can then send that reply. Thus it is essential that, while either the invoker or request_handler_1_2 is examining or updating the message in these links, the other process does not use either of the links. We begin our analysis by checking that these links are indeed used correctly.

Recall that in order to determine whether some property holds for the behaviors of a system, we first interpret that property in terms of the appearance of a pattern of event symbols in behavior strings. In this case we would like to show that, between the time one of the processes receives a message from a link connected to the port from_rq2 and the time it next sends a message to one of those links, the other process makes no use of those links. In terms of symbols in a behavior string, we would like to show that the next symbol representing a use of the links, following a symbol representing the receipt of a message from one of them, must represent the transmission of a message by the process which has just received.

We suppose to the contrary that the next symbol represents a use of the links by the other process. From Rule I.1 and statements IN14 and IN19 we know that

(1) $|s(\text{to\_rq2}, *)| \le |r(*, \text{from\_rq2}, *)| \le |s(\text{to\_rq2}, *)| + 1$

while Rule I.1 and statements RQ6 or RQ9 and RQ11 give us

(2) $|s(\text{2\_to\_inv}, *)| \le |r(*, \text{2\_from\_inv}, *)| \le |s(\text{2\_to\_inv}, *)| + 1$

with equality on the left for any prefix of a behavior containing the send symbol not followed by the appropriate receive symbol, and equality on the right for any prefix of a behavior containing the receive symbol not followed by the appropriate send symbol.

Rule II.1 implies that for any prefix of a behavior we have

$$|r(*, \text{from\_rq2}, *)| \le |s(\text{to\_rq2}, *)| + |s(\text{2\_to\_inv}, *)| - |r(*, \text{2\_from\_inv}, *)| + 1$$

(the 1 arising from the initial message in one of the links connected to 2_from_inv), from which it follows that

$$|r(*, \text{from\_rq2}, *)| + |r(*, \text{2\_from\_inv}, *)| \le |s(\text{to\_rq2}, *)| + |s(\text{2\_to\_inv}, *)| + 1$$

But adding (1) and (2), and combining with this last inequality, we have

(3) $|s(\text{to\_rq2}, *)| + |s(\text{2\_to\_inv}, *)| \le |r(*, \text{from\_rq2}, *)| + |r(*, \text{2\_from\_inv}, *)| \le |s(\text{to\_rq2}, *)| + |s(\text{2\_to\_inv}, *)| + 1$

In a prefix of a behavior containing one of the receive symbols, not followed by the corresponding send symbol, we have equality on the right in at least one of (1) and (2). But (3) implies that this cannot be true for both (1) and (2). Thus two receive symbols cannot occur in a behavior without an intervening send symbol.

Suppose a symbol representing a use of the links occurs between an $r(*, \text{from\_rq2}, *)$ and the next $s(\text{to\_rq2}, *)$ in some behavior. We have just seen that the first such symbol must be an $s(\text{2\_to\_inv}, *)$, since an $r(*, \text{2\_from\_inv}, *)$ cannot follow an $r(*, \text{from\_rq2}, *)$ without an intervening send symbol. Consider the prefix of the behavior ending with this $s(\text{2\_to\_inv}, *)$. For this prefix, we have

$$|r(*, \text{from\_rq2}, *)| = |s(\text{to\_rq2}, *)| + 1 \qquad \text{by} \quad (1)$$

and

$$|r(*, \text{2\_from\_inv}, *)| = |s(\text{2\_to\_inv}, *)| \qquad \text{by} \quad (2)$$

so

$$|r(*, \text{from\_rq2}, *)| + |r(*, \text{2\_from\_inv}, *)| = |s(\text{to\_rq2}, *)| + |s(\text{2\_to\_inv}, *)| + 1$$

Since the $s(\text{2\_to\_inv}, *)$ represents the first use of the links *following* the $r(*, \text{from\_rq2}, *)$, we have

$$|r(*, \text{from\_rq2}, *)| + |r(*, \text{2\_from\_inv}, *)| = |s(\text{to\_rq2}, *)| + |s(\text{2\_to\_inv}, *)| + 2$$

for the prefix ending with our $r(*, \text{from\_rq2}, *)$. This contradicts (3), so our system of inequalities is inconsistent and the assumption that a use of the links occurs between an $r(*, \text{from\_rq2}, *)$ and the next $s(\text{to\_rq2}, *)$ must be false. A similar argument shows that no use of the links occurs between an $r(*, \text{2\_from\_inv}, *)$ and the next $s(\text{2\_to\_inv}, *)$.

This demonstrates that the aspect of the design involving mutually exclusive use of the links connected to from_rq2 is indeed sound. Arguments of the same type, which we omit here, show that the links connected to the get_status port and the from_rq3 ports are also used in the proper mutually exclusive fashion.

Having shown that the messages in these links are used properly, we next consider whether any of the processes in the design starve unexpectedly. Since the reply_handler and request_handler DYMOL programs consist of nonterminating loops, we expect that these processes will eventually starve while waiting to receive at statements RP2 and RQ2 respectively. This is the intended behavior of the system after the invoker processes of the various nodes have terminated. But we would like to be sure that the processes of our system never starve under other circumstances. In the design described in Section 4 we have concentrated on a single node. We retain that perspective here, and for the moment we simply assume that every request from node 1 eventually receives replies from the other nodes of the system. We are then concerned with the possibility that one of the processes in node 1 suffers starvation while waiting for a message from another process in that node.

Suppose, for example, that request_handler_1_2 starves while waiting to receive at port 2_from_inv, that is, while waiting to execute statement RQ6 or RQ9. Clearly, we can interpret this simply as the appearance of the symbol $w(\text{2\_from\_inv})$ in a behavior. If a $w(\text{2\_from\_inv})$ appears, Rule II.2 implies that

$$|r(*, \text{from\_rq2}, *)| + |r(*, \text{2\_from\_inv}, *)| = |s(\text{to\_rq2}, *)| + |s(\text{2\_to\_inv}, *)| + 1$$

(the 1 arising from the initial message in one of the links connected to 2_from_inv), and Rule I.1 implies that

$$|r(*, \text{2\_from\_inv}, *)| = |s(\text{2\_to\_inv}, *)|,$$

both conditions applying to the complete behavior string. Combining these, we see that

$$|r(*, \text{from\_rq2}, *)| = |s(\text{to\_rq2}, *)| + 1$$

at the end of the behavior. This is only possible if the invoker process halts between statements IN14 and IN18. Rules I.2 and I.3 imply that a process can halt only at a RECEIVE statement (by starvation) or at a STOP statement. Since none of statements IN15, IN16, or IN17 is a RECEIVE or STOP statement, we see that the invoker cannot halt between IN14 and IN18, contradicting the last inequality. We may therefore conclude that no $w(\text{2\_from\_inv})$ appears in a behavior.

Again, arguments of a similar nature apply to the rest of the design, showing that none of the processes in node 1 starves while waiting for a message from another process in the node. We would now like to determine whether a node will in fact eventually send a reply for each request it receives. We begin by assuming that node 1 receives a reply for each request it sends and determining whether node 1 replies to each request from other nodes. We then consider the problems of interaction among nodes.

Suppose that node 1 eventually receives a reply for each request it sends, but that it permanently defers a reply destined for some other node, say node 2. We can interpret this as the appearance of an $r(*, \text{req\_2}, *)$ in a behavior which is not followed by an $s(\text{resp\_to\_2}, *)$ from request_handler_1_2 or an $s(\text{resp\_2}, *)$ from the invoker.

Since the request_handler does not starve while waiting for a message from inside node 1, the rules of class I imply that any $r(*, \text{req\_2}, *)$ is followed by an s(resp_to_2, *) and then an s(2_to_inv, no_def), or by an s(2_to_inv, def). Our assumption that the reply to node 2 is permanently deferred eliminates the first possibility, so a behavior in which a reply to node 2 is permanently deferred contains an r(*, req_2, *) followed by an s(2_to_inv, def), but not by an s(resp_to_2, *) or an s(resp_2, *).

Since node 2 waits for a reply to its request before initiating any additional requests, no further r(*, req_2, *) symbols occur in the behavior, and thus the rules of class I imply that request_handler_1_2 makes no further use of the links connected to port 2_from_inv after sending the "def" message. We saw earlier that the use of these links alternates between sends and receives, so the next use of the links after the s(2_to_inv, def) must be a receive. Since the request_handler does not use the links again, that receive must be an r(2_to_inv, from_rq2, def). But the rules of classes I and III imply that any r(*, from_rq2, def) will be followed by an s(resp_2, true), which would contradict our hypothesis. So our hypothesis can hold only if the "def" message sent by the request_handler is never received, and the s(2_to_inv, def) represents the last use of the links connected to 2_from_inv by any process in the system.

The fact that request_handler_1_2 sends a "def" message implies, by Rule III.1, that the last r(*, 2_status_in, *) in the behavior was an r(*, 2_status_in, true), indicating that the invoker process was in its critical section. Our assumption that node 1 eventually receives a reply for each request it sends, together with our earlier observation that none of the processes starves while waiting for a message from within node 1, implies that the invoker will exit the critical section after this last r(*, 2_status_in, *). The rules of class I and the fact that no process in node 1 starves while waiting for a message from within the node tell us that, after the invoker exits the critical section, it receives whatever message is then available in the links connected to the port 2_from_inv. We have seen that the last "def" message sent by the request_handler is never received, so we conclude that, if our hypothesis is true, the invoker must exit the critical section and take the message in one of those links before request_handler_1_2 sends the "def" message. That is, if there is a behavior fulfilling our hypothesis, then that behavior ends with a segment of the form:

s(put_status, true) . . r(*, 2_status_in, true) . . s(2_status_out, true) . .
r(*, get_status, true) . . s(put_status, false) . . r(*, from_rq2, no_def) . .
s(to_rq2, no_def) . . r(*, 2_from_inv, no_def) . . s(2_to_inv, def) . . .

The rules of class I give no further information about these events, since the events within individual processes occur in the correct orders. We have used the rules of class I together with those of class II to show that the various links are used in a mutually exclusive fashion, and we have used rule III.1 to show that the behavior must end with a segment of the form given above. Seeing no way to generate further inequalities that would be inconsistent with this conclusion, and thus show that a reply cannot be permanently deferred, the designer might now attempt to construct an actual behavior in which a reply is indeed permanently deferred.

It is easy to write down a behavior in which request_handler_1_2 executes its "RECEIVE 2_status_in" and "SEND 2_status_out" instructions between the execution of the "SET BUFFER := critical" instruction and the "RECEIVE get_status" instruction by the invoker. If this behavior continues with the invoker executing its "SET BUFFER := false", "SEND put_status", "RECEIVE from_rq2", and (after skipping past the conditional because it received a "no_def" message through from_rq2) "SEND to_rq2" instructions before request-handler_1_2 executes a "RECEIVE 2_from_inv" instruction, a reply will be permanently deferred. Because request_handler_1_2 got a "true" message through port 2_status_in, it can (assuming that the nondeterministic INTERNAL TEST evaluates to true) eventually execute its "SET BUFFER := def" and "SEND 2_to_inv" 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 been revealed by this analysis is rather subtle. Indeed, essentially this same error appeared in the first published version of the Ricart and Agrawala algorithm ([28]), necessitating the publication of a revised version a few months later ([29]). 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 request_handler_1_2 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 request_handler_1_2 is about to replace. Our experience indicates that subtle errors like this one, 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.

At this point we have established that some aspects of the design are sound, but that it also contains a serious error. The next step in the development of the design would be to modify it so as to eliminate the error and then analyze the new design to assure that the modification does indeed correct the error and introduces no further errors. A modification that appears to eliminate the problem in our example is to change request_handler_1_2's DYMOL program so that request_handler_1_2 removes the message available through its 2_from_inv port as soon as it receives a request. The new DYMOL program is given in Figure 5.

Most of the analysis of the original system carries over to the modified one, and we do not describe it here. We do, however, show that node 1 of the modified system does not permanently defer requests, as long as its own requests receive replies.

Assume that a request from node 2 is permanently deferred. Proceeding exactly as before, we see that the behavior must have the form:

s(put_status, true) . . *r*(*, 2_status_in, true) . . *s*(2_status_out, true) . .
r(*, get_status, true) . . *s*(put_status, false) . . *r*(*, from_rq2, no_def) . .
s(to_rq2, no_def) . . *r*(*, 2_from_inv, no_def) . . *s*(2_to_inv, def) . . .

Rule I.1 implies that request_handler_1_2's last *r*(*, 2_from_inv, *) precedes its last *r*(*, 2_status_in, true) and invoker's last *r*(*, from_rq2, *) precedes its

```
REQUEST_HANDLER_1_2:
 R1:        DO FOREVER
                BEGIN
 RQ2:           RECEIVE req_2;          --receive request from node 2
 RQ3:           RECEIVE 2_from_inv;     --prevent invoker from
                                        --reading deferral message
 RQ4:           RECEIVE 2_status_in;    --check status of invoker
 RQ5:           SEND 2_status_out;
 RQ6:           IF BUFFER = true AND INTERNAL TEST THEN
 RQ7:             SET BUFFER := def      --defer reply
                ELSE
                  BEGIN
 RQ8:               SEND resp_to_2;     --send reply
 RQ9:               SET BUFFER := no_def
                  END
 RQ10:          SEND 2_to_inv     --make deferral message
                                  --available again
              END
```

Fig. 5. Revised request_handler DYMOL program.

last *s*(to_rq2, no_def). But then an *r*(*, from_rq2, *) appears between an *r*(*, 2_from_inv, *) and the succeeding *s*(2_to_inv, *). Since the argument showing mutually exclusive use of these links applies to the new system as well as the old one, this is a contradiction. Therefore, node 1 does not permanently defer a reply.

This discussion of the deferral of replies has assumed that all requests from node 1 eventually receive replies. This assumption is appropriate at this stage of the development process, when the designer is primarily concerned with the structure of a single node. Since the decision to defer a reply is described at this stage in the design as being based in part on the nondeterministic INTERNAL TEST, it is possible, according to this description, that node 1 does not receive a reply to each of its requests, leading to a deadlock with all replies being deferred. In the completed system this possible source of deadlock is avoided by the priority comparison used by the Ricart and Agrawala algorithm. Further elaboration of the design would introduce this priority comparison, and analysis at that later stage would then be able to confirm that no aspect of the design would allow deadlocks to occur.

7. 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.

Although we believe that the foregoing demonstrates that our approach can be a significant aid to designers of distributed software systems, several additional issues should be considered in assessing its potential as a practical software development tool. First, is the approach applicable to a wide range of problems or is its usefulness restricted to problems similar to the example considered in this paper? Second, how compatible is the approach with other stages of the software development process? Third, can the approach be successfully automated?

While our approach to describing and analyzing designs of distributed systems was originally developed to attack problems peculiar to concurrent systems, such as synchronization problems, it is potentially useful for a much wider class of problems. Since the fundamental analysis technique is determining whether some specific event or sequence of events can occur as part of a system's behavior, the approach applies to any problem that can be described in those terms. Thus, it is clearly applicable to almost any problem related to the *functionality* of a system. In fact, a related project has used the same fundamental event-based approach to observing system behavior as the basis for a general purpose debugging tool [2]. We also believe that with appropriate extensions, such as adding events that correspond to the elapsing of time intervals, the approach might be extended to problems related to system *performance*. To date we have only done preliminary work in this direction, however.

As presented in this paper, our approach is expressly tailored for use in the design phase of software development. In particular, the DYMOL notation is a modeling language meant for describing the intended behavior of a system, not a programming language meant for defining the declarative and imperative structure of an implementation that will achieve that behavior. (In fact, the DYMOL design notation presented in this paper is only a research vehicle. Improved syntax and additional constructs would be desirable in any design notation intended for practical use.) Nevertheless, the approach is certainly applicable to much more than just the design stage of development. One could, for example, conceive of extending DYMOL into an implementation language or, equivalently, of implementing its SEND, RECEIVE, CREATE, DESTROY, OPEN and CLOSE operations as services in an underlying operating system. This would permit fundamentally similar notations to be used in both design and implementation, and would also make our analysis techniques applicable to implementation as well as design descriptions of a system. Even if the design and implementation languages are quite dissimilar, however, the same basic analysis technique is still applicable at both stages. For instance, as part of her dissertation research [6], our student, Laura K. Dillon, demonstrated the applicability of event sequence-based behavior description to languages such as CSP [13] and Ada [8], whose interprocess communication primitives are fundamentally different from those of DYMOL. As a consequence, suitably modified versions of the analysis technique described here could be applied to programs implemented in those languages. In fact, Dillon has used such techniques to analyze the behavior of a small CSP system.

In this paper, we have given a prose description of the analysis performed on our example distributed software design. The analysis that we describe, however, can be expressed entirely in terms of the consistency or inconsistency of systems of inequalities. We therefore believe that many aspects of this analysis can be automated. Such automation must, of course, confront the problem of combinatorial explosion. In this regard, Dillon's work on constrained expressions [6, 36] is especially promising. The constrained expression formalism gives a closed form description of all the possible behaviors of a distributed system, and the approach to analysis described here carries over naturally to the constrained expression setting. In that context, however, closed form descriptions allow large classes of behaviors to be handled simultaneously, rather than on a case-by-case basis, thereby greatly reducing the problem of combinatorial explosion.

In sum, 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 the early stages of the development process and is also compatible with various implementation languages. Moreover, it provides a means for rigorously analyzing the incomplete and abstract descriptions arising in early stages of development as well as the more complete and detailed descriptions that appear later in the process. 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 developers of distributed software systems.

ACKNOWLEDGMENTS

We are grateful to Laura Dillon, Alexander Wolf, and William Riddle for their comments on earlier versions of this paper. We also thank the referees for their useful advice.

REFERENCES

1. Apt, K., Francez, N., and DeRoever, W. A proof system for communicating sequential processes. *ACM Trans. Program. Lang. Syst. 2*, 3 (July 1980), 359–385.
2. Bates, P., and Wileden, J. High-level debugging of distributed systems. *J. Syst. Softw.* (Dec. 1983), 255–264.
3. Brinch Hansen, P. Distributed processes: A concurrent programming concept. *Commun. ACM* (Nov. 1978), 934–941.
4. Campbell, R., and Kolstad, R. Path expressions in Pascal. In *Proceedings of the 4th International Conference on Software Engineering* (Munich, Sept. 1979).
5. Clarke, L., Graham, R., and Wileden, J. Thoughts on the design phase of an integrated software development environment. In *Proceedings of the 14th Hawaii International Conference on Systems Science* (Honolulu, Jan. 1981).
6. Dillon, L. Analysis of distributed systems using constrained expressions. Ph.D. dissertation, Computer and Information Science Dept., Univ. of Massachusetts, Aug. 1984.
7. Dillon, L., Avrunin, G., and Wileden, J. Analyzing distributed systems using constrained expressions. SDLM 83-3, Univ. of Massachusetts, Feb. 1983.
8. DoD. United States Department of Defense. *Reference Manual for the Ada Programming Language.* ANSI/MIL-STD-1815A, Jan. 1983.
9. Feldman, J. High-level programming for distributed computing. *Commun. ACM* (June 1979), 353–368.
10. Floyd, R.W. Assigning meaning to programs. In *Proceedings of Symposia in Applied Mathematics, Mathematical Aspects of Computer Science* (1967), 19–32.
11. Habermann, A.N. Synchronization of communicating processes. *Commun. ACM* (Mar. 1972), 171–176.
12. Hoare, C.A.R. An axiomatic basis of computer programming. *Commun. ACM* (Oct. 1969), 576–580, 583.

13. HOARE, C.A.R. Communicating sequential processes. *Commun. ACM* (Aug. 1978), 666–677.
14. HOARE, C.A.R. Some properties of predicate transformers. *J. ACM* (July 1978), 461–480.
15. HOARE, C.A.R. A model for communicating sequential processes. In *On the Construction of Programs*, McKeag and McNaghton, Eds., Cambridge University Press, 1980, 229–243.
16. HOLZMANN, G.L. A theory for protocol validation. *IEEE Trans. Comput.* (Aug. 1982), 730–738.
17. LAMPORT, L. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM* (Aug. 1974), 453–455.
18. LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.* (Mar. 1977), 125–143.
19. LAMPORT, L. Time, clocks and the ordering of events in a distributed system. *Commun. ACM* (July 1978), 558–565.
20. LAMPORT, L. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst. 1*, 3 (July 1979), 84–97.
21. LAUER, P.E., AND CAMPBELL, R.H. Formal semantics for a class of high-level primitives for coordinating concurrent processes. Acta Inf. (1975), 247–332.
22. LAUER, P.E., TORRIGIANI, P.R., AND SHIELDS, M.W. COSY: A system specification language based on paths and processes. *Acta Inf.* (1979), 451–503.
23. LISKOV, B. Primitives for distributed computing. In *Proceedings of the 7th Symposium on Operating Systems Principles* (Dec. 1979), 33–42.
24. MISRA, J., AND CHANDY, K.M. Proofs of networks of processes. *IEEE Trans. Soft. Eng.* (July 1981), 417–426.
25. OWICKI, S., AND GRIES, D. Verifing properties of parallel programs: An axiomatic approach. *Commun. ACM* (May 1976), 279–285.
26. PNUELI, A. The temporal semantics of concurrent programs. In *Semantics of Concurrent Computation*, Kahn, Eds., Springer-Verlag, New York, 1979, 1–20.
27. RAMAMRITHAM, K., AND KELLER, R.M. Specifying and proving properties of sentinel processes. In *Proceedings of the 5th International Conference on Software Engineering* (1981), 374–382.
28. RICART, G., AND AGRAWALA, A. K. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM* (Jan. 1981), 9–17.
29. RICART, G., AND AGRAWALA, A.K. Corrigendum. *Commun. ACM* (Sept. 1981), 578.
30. RIDDLE, W., WILEDEN, J., SAYLER, J., SEGAL, A., AND STAVELY, A. Behavior modeling during software design. *IEEE Trans. Softw. Eng.* (July 1978), 283–292.
31. RIDDLE, W. An approach to software system modeling and analysis. *J. Comput. Lang.* (1979), 49–66.
32. TAYLOR, R.N. A general purpose algorithm for analyzing concurrent programs. *Commun. ACM* (May 1983), 362–376.
33. WILEDEN, J. Modeling parallel systems with dynamic structure. COINS Tech. Rep. 78-4, Univ. of Massachusetts, Jan. 1978.
34. WILEDEN, J. DREAM—an approach to the design of large-scale concurrent software systems. In *Proceedings of the 1979 National Conference of the ACM* (Oct. 1979), ACM, New York, 88–94.
35. WILEDEN, J. Techniques for modeling parallel systems with dynamic structure. *J. Digital Syst. 4*, 2 (Summer 1980), 177–197.
36. WILEDEN, J. Constrained expressions and the analysis of designs for dynamically structured distributed systems. In *Proceedings of the 1982 International Conference on Parallel Processing* (Aug. 1982), 340–344.

Received September 1983; revised October 1984; accepted April 1985