

INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the original text directly from the copy submitted. Thus, some dissertation copies are in typewriter face, while others may be from a computer printer.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyrighted material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is available as one exposure on a standard 35 mm slide or as a 17" × 23" black and white photographic print for an additional charge.

Photographs included in the original manuscript have been reproduced xerographically in this copy. 35 mm slides or 6" × 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.



Accessing the World's Information since 1938

300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA

Order Number 8820276

Responsive sequential processes

Costello, Roger Lee, Ph.D.

The Ohio State University, 1988

U·M·I

**300 N. Zeeb Rd.
Ann Arbor, MI 48106**

RESPONSIVE SEQUENTIAL PROCESSES

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor Of Philosophy in the Graduate

School of the Ohio State University

By

Roger Lee Costello, B.S., M.S.

The Ohio State University

1988

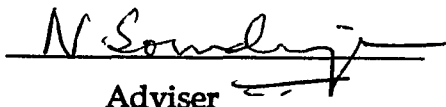
Dissertation Committee:

N. Soundararajan

P. Sadayappan

T. Lai

Approved by



Adviser

Department of Computer and
Information Science

ACKNOWLEDGEMENTS

I would like to express my sincerest thanks to my adviser, Prof. Neelam Soundararajan for his guidance, advice and friendship through the years. Thanks also go to the other members of my advisory committee, Drs. P. Sadayappan, and R. Parent, for their suggestions and comments. To my wife, Patricia, I offer sincere thanks for her support and encouragement.

VITA

November 6, 1957	Born - Davenport, Iowa
1980	B.S. (Chemistry), St. Ambrose College, Davenport, Iowa
1983	M.S. (Computer Science), The Ohio State University, Columbus, Ohio

PUBLICATIONS

"Modeling Switch-Level Simulation using Dataflow", Proceedings of the 22nd Design Automation Conference, June 1985, pp. 637-644.

"Distributed Discrete Event Simulation using Dataflow", Proceedings of the 1985 International Conference on Parallel Processing, pp. 503-510.

"Responsive Sequential Processes", SIGPLAN Notices, March, pp. 53-63.

"Responsive Sequential Processes", Workshop on Formal Techniques in Real-Time and Fault-Tolerant Systems.

FIELDS OF STUDY

Major Field: Programming Languages

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	ii
VITA	iii
LIST OF FIGURES	vi
CHAPTER	PAGE
I. INTRODUCTION	1
Motivation	1
Other Languages	5
RSP versus Real-Time	8
Notation	12
CSP	13
Organization of Thesis	15
II. RESPONSIVE SEQUENTIAL PROCESSES ...	16
III. SYSTEM MODEL	36
IV. EXAMPLES	43
Example 1. Prime Number Example	44
Example 2. Line Printer Spooler Example .	49
Example 3. Line Printer Spooler Example, version 2	55
Example 4. Alarm Clock	65
Example 5. Alarm Clock, version 2	67
V. DISCUSSION	68

VI. CONCLUSION	81
Modula vs RSP	81
Distributed Programming System vs RSP ...	83
CSP vs RSP	86
Motivating Concepts for RSP	88
Summary and Further Research	95
LIST OF REFERENCES	98

LIST OF FIGURES

FIGURES	PAGE
1. Achieving Efficient State Roll-back and Update	39
2. Communication Protocol State Diagram	42

CHAPTER I

INTRODUCTION

Motivation

Responsive Sequential Processes (RSP) is a programming language for distributed real-time systems. RSP provides the programmer with a natural view of distributed real-time problems and a powerful set of constructs for expressing his program.

This chapter is organized as follows: In this first section I describe RSP's intended application area and suggest the type of constructs that are needed in a programming language to satisfy the needs of the area. In the next two sections I consider other languages which attempt to satisfy the areas' requirements and explain why these languages fall short in their attempt. Before introducing RSP it will be useful to introduce the notation that will be used throughout the dissertation. This notation is presented in the next two sections. In the last section I describe the organization of the remaining chapters of the dissertation.

Consider a system that contains not only "computational" activities but also a number of "physical devices". Examples of such devices might

include robot arms, rocket engines, cameras, etc. In view of the complexity of such systems, the most natural way to organize such a system would be as a set of (nearly) independent processes interacting with each other by means of explicit communication. The communication mechanism in such systems must satisfy some special requirements. Consider an example: a process P requires a robot arm, that is under the control of process Q, to move a certain distance and return some data within, say, two seconds. Suppose P sends the request to Q, the two seconds elapse, and Q does not reply with the needed data. What can P do?

One approach that is often used is for P to activate some emergency routine when the two seconds expire. Typically, however, there is very little that can be done by the emergency routine. The data (from Q) was needed within two seconds and that time has passed.

I believe a better solution to the problem would be to have a communication mechanism that is 'responsive'; in the above situation, as soon as P sends (or tries to send) its request to Q, Q must either reject it (since it cannot meet the constraints specified in the request, perhaps because it is already committed to processing too many other requests), or accept it for later processing. If Q accepts the request, P can proceed with its other activities, confident that Q will perform the required service; if Q rejects the request, P will have plenty of time to try other alternatives such as trying to see if some other process, Q', would accept the request.

But what if Q is already busy with some other (internal) activities? How will it respond quickly to P's messages and accept it or reject it? I will postpone the answer to these questions to chapter II.

Also needed in this area of programming is the ability for a process to react quickly to certain messages it receives. To continue with the above example - suppose Q accepts P's request. Q will then try to perform the requested service within the specified time. But what if the robot arm that Q controls jams? Clearly Q cannot then meet its commitment to P (to move the arm the required distance and send back data within 2 seconds). In this situation, the very least Q should do is to inform P (as well as any other processes that it has accepted requests from) that it will not be able to honor its earlier commitment. It is, of course, quite simple to send such a message, but that is not enough. Recall that P is proceeding on the assumption that Q will indeed perform the required service. Hence the message from Q reniging upon its previous commitment might lead to serious consequences for P unless it reacts quickly and takes appropriate corrective action (possibly abandoning whatever computations it is currently engaged in). Note that the 'reaction' I am talking about here is very different from the 'response' I spoke of in the last paragraph. A quick response from the (intended) recipient of a message allows the sender to proceed appropriately depending upon whether the message was accepted or rejected by the recipient. On the other hand, a process needs to be able to react quickly to certain messages it might receive

and take appropriate corrective action since otherwise it (the receiving process) might get into serious difficulties. Quite possibly, as is the case in the example of the process controlling the robot arm, the sending process Q doesn't care whether the receiving process even accepts or rejects the message. Thus the ability on the part of the receiving process to react quickly to certain messages is needed not so much in order to allow the sender to proceed appropriately, as to allow the receiver to act quickly to handle the emergency that the message signifies. I will not explain here how a process will identify the set of messages that it wishes to consider as "emergencies" nor how it will react to the receipt of such a message, postponing these to chapter II.

These two ideas - that a recipient process must accept or reject messages as soon as they are sent and that it should be able to react quickly to certain messages when they arrive and take appropriate corrective action - will, I believe, be useful in building systems that include physical devices in addition to computational activities and are the basic ideas underlying the work reported in this proposal.

Other Languages

This section introduces other languages which have been designed for the same application area. A detailed comparison of RSP and these languages may be found in Chapter VI.

Systems that are comprised of physical as well as computational activities and have time constraints associated with the activities are commonly referred to as real-time systems. A real-time program is one in which the code is designed to meet as many of the time constraints as possible. Typically, in real-time systems there is an operating system which schedules the processes to run on the processors, and the operating system's scheduling algorithm tries to ensure that all time constraints are met.

The overwhelming focus (as may be seen by a cursory view of the literature) in real-time systems is on scheduling algorithms. That is, what algorithms will best ensure that time constraints of each process will be met?

I believe that this is not the appropriate focus. In the model for RSP it will be seen that there is no operating system. Instead, each process schedules its own activities. Hence there is no need for an operating system and its associated scheduling algorithms. Since RSP takes a view totally orthogonal to existing schemes, I do not present a complete survey of existing real-time languages. Instead, a brief overview of such languages is presented below. A complete survey may be found in the references [3, 11, 12, 13, 18, 19, 20, 21, 27, 29].

(a) Modula

Modula [27] was designed especially for applications such as embedded industrial and scientific real-time computer systems [29]. The language has no built-in high level I/O facilities but instead provides constructs for writing device handlers directly in the language. In [28] Wirth describes a *discipline* to be followed when writing real-time programs. The program, he states, is to be written first to ensure that it is logically correct and then another pass is made to check that any time constraints are met.

When reading a real-time Modula program it is not apparent that there are any time constraints. It is difficult, then, to determine what, if any, time constraints are being met. This inability to distinguish real-time programs from non-real-time programs is very typical of early real-time systems. Early languages for real-time systems were also responsible for introducing the notion of interrupts and processor sharing.

(b) Distributed Programming System

In 1985 Lee and Gehlot proposed a language that was designed especially for distributed, real-time systems [18]. Their language includes constructs (called temporal scope) for specifying time constraints of code execution and of interprocess communication. These constructs facilitated a run-time system (i.e. operating system) in scheduling processes for execution

on the processors. They have argued that this eliminates a lot of work and complexity for the programmer, since now the run-time system does the scheduling. In addition, the inclusion of time constraints makes it obvious to the reader of the program where and what the constraints are, which was missing from earlier languages such as Modula. Another feature missing in earlier languages but incorporated in DPS is exception handling. With DPS if a time constraint is violated then control is transferred to the end of the temporal scope, where an exception handler processes the error.

(c) CSP (and extensions)

CSP [28] is the first language to actually model distributed systems. Unlike other systems, each process has its own processor - thus eliminating the need for an operating system. All interactions between processes take place by means of explicit, synchronized communications. As a consequence, a process may wait at an input or output for an arbitrarily long time for the other process to synchronize. Obviously, for applications with time constraints this is unsuitable. OCCAM [24] proposed an extension to CSP whereby I/O statements and guards would have an associated time-out mechanism. Now a process remains at an I/O command until the other process executes a matching I/O, or, the time-out occurs. If the time-out occurs then some emergency action is attempted.

In summary, early real-time languages, such as Modula, were written without constructs to show what constraints were being met. Hence, a real-time program could not be distinguished from a non-real-time program. Later languages saw the inclusion of constructs to show what constraints are present in the program. Typically, processors were shared to save on the high cost of processors. This, of course, necessitated the use of an operating system to schedule the processes on the processors. Also, to achieve rapid response an interrupt mechanism was frequently employed. Processor sharing, scheduling, and interrupts are common to almost all real-time systems today. Extended CSP is the exception. Each process has its own processor. Processes interact by means of explicit communications. A time-out mechanism prevents a process from waiting at a communication point for an arbitrarily long time.

The next section contains a comparison between systems designed using RSP and other real-time systems.

RSP versus Real-Time

As described above, real-time systems are systems where "time" is important, and one cannot wait indefinitely for certain events to take place. We already saw one reason for this in the first section: if a system consists of physical devices (as well as computational activities), then the nature of these devices requires the programmer to ensure that certain actions are performed

within specified time limits and, if they cannot, the appropriate process should be informed quickly so it has time to pursue other alternatives. As I explained in the first section, RSP was designed for dealing with this type of constraint.

The system model underlying RSP is much like that underlying CSP: I assume that every "part" of a system has its own processor that performs all the computations required by that part of the system. Thus, for instance, in the example considered in the first section, all computations required to manipulate the robot arm are performed by a separate processor (executing the process Q). This is not true of many current real-time systems [22,23]; in these systems all computations required by the various parts of the system are carried out on a central pool of (one or more) processors, with an operating system responsible for scheduling the various computations. Many of these computations will have time constraints associated with them since, considering the example of the robot arm again, if the robot arm is to be moved within 2 seconds then the computations to decide the exact sequence of instructions to be sent to the robot arm to effect this movement should be completed long before then. The operating system tries to schedule the various computations such that their time constraints are met; in general it won't be able to meet all the constraints and when a constraint is violated then an appropriate "emergency handling routine" will have to be executed.

Obviously, because of such things as emergencies, an operating system cannot guarantee that a request's constraint will be met, and the same is true of a RSP process. So, both schemes must program for such events. Existing real-time systems, however, have an added level of complexity: A process in one of these systems must also program for the event that a computation scheduled by the operating system is not completed within the required time, not because of an emergency within the process, but because all of the computing resources are being used. Originally the operating system may have been able to schedule the request but later, perhaps due to emergencies in other processes, the computing resources are all being utilized to handle these emergencies and the computations of the first process cannot be completed now. This cannot happen in RSP since each process has its own processor. Thus, operating system used in existing real-time systems will force the programmer of each process to have a global view of the system.

Emergencies present other problems for existing real-time systems that are not found in RSP. With both schemes an emergency in one process may result in reniging on requests which result in generating an emergency for those processes, thus creating a "ripple effect". While this is undesirable, there is consolation in that the effect is localized to processes that are related (by interaction). The ripple effect in current real-time systems, however, may extend beyond, to totally unrelated processes. Consider a process P in a

real-time system which receives an emergency. P sends renig messages to those processes which made requests of it. Suppose Q is such a process and Q currently does not have a computing resource, and further, this renig message is an emergency for Q. If there are no computing resources free then the operating system must preempt another process R. However, this may cause an emergency for R. This, in turn, may create other emergencies, etc. The emergency induced in Q is common to both RSP and existing real-time systems. This is the *localized ripple effect*. The emergency induced in R is not found in RSP, and is an example of how, in current real-time systems, the ripple effect may extend over totally unrelated processes.

By assuming that there is a separate processor for each part of the system which will perform all of the computations required by that part of the system I eliminate all of these problems (including the need for a complex operating system), and gain a whole host of advantages. In particular, each process has flexibility in its scheduling, there is greater autonomy of processes and each process is less complex. Given the advantages of my approach and the current performance, size and price of microprocessors, my approach seems reasonable.

Notation

Before looking at RSP some notation needs to be introduced. The syntax for selection and alternative statements used in RSP are based upon CSP's notation [16]. The selection statement's syntax is:

$$\begin{array}{l} [G_1 \rightarrow S_1 \\ \quad \square G_2 \rightarrow S_2 \\ \dots \\ \quad \square G_n \rightarrow S_n \\] \end{array}$$

The semantics of this statement is that a statement S_i is executed whose guard G_i "succeeds", $1 \leq i \leq n$. If two or more guards succeed a choice is made nondeterministically. If none succeed, the statement aborts.

A guard may be either (1) a boolean expression (boolean guard), or (2) a boolean expression followed by an I/O statement (then the guard is known as an I/O guard). A boolean guard succeeds if the boolean expression evaluates to true. An I/O guard succeeds if its boolean portion is true and the I/O portion succeeds. The box, \square , denotes alternative.

The syntax for the loop statement is:

$$\begin{array}{l} * [G_1 \rightarrow S_1 \\ \quad \square G_2 \rightarrow S_2 \\ \dots \\ \quad \square G_n \rightarrow S_n \\] \end{array}$$

The $*$ symbol comes from the Kleene star in formal language theory and denotes repetition. This loop is repeated until all of the guards fail.

CSP

Responsive Sequential Processes (RSP) was designed with Communicating Sequential Processes (CSP) as a model and CSP is frequently referred to throughout this dissertation. It will be instructive, then, to describe CSP in a little greater depth than in the previous section.

A CSP program consists of a set of independently executing processes $[P_1 // \dots // P_n]$. Processes interact by means of explicit, synchronized communication: $P_i :: [P_j!x]$ means that process P_i outputs (shrieks) the value of x to process P_j . The communication does not actually take place until P_j executes a matching input statement. When P_j executes the matching input statement: $P_j :: [P_i?y]$ then the value is transferred and is assigned to y . The I/O statements may also be used as an I/O guard in selection and loop statements. When used in a selection statement: $P_i :: [P_j!x \rightarrow S_1 \sqcap P_k?y \rightarrow S_2]$ the system randomly selects one of the guards that succeeds, performs the I/O, and then executes the statement following the arrow. If both guards fail then the process aborts. An I/O guard fails when the matching process has

terminated. I/O guards may also be employed in loops:

$$*[P_j!x \rightarrow S_1 \sqcap P_k?y \rightarrow S_2].$$

The loop is repeated until both guards fail.

Here is an example of a CSP program which has three processes.

Process P_1 receives a value from each of processes P_2 and P_3 and then finds the maximum value of the two.

```
P1 :: [ cnt := 1;
      *[ cnt ≤ 2 → [P2?x → cnt := cnt + 1 ⊓ P3?y → cnt := cnt + 1];
      [ x ≤ y → max := y ⊓ y < x → max := x]
    ]
```

```
P2 :: [ value := 12; P1!value]
```

```
P3 :: [ value := 8; P1!value]
```

Process P_1 has a loop which it repeatedly executes until it has received two values. On each iteration a counter, cnt , is incremented. When the counter exceeds a value of two then the loop is terminated and it is determined whether x or y has the maximum value using a selection statement. In this example Max will be assigned the value of 12. Process P_2 simply assigns the variable "value" a value and then sends the value to P_1 . Note here that the output command is being used as an ordinary statement, whereas in P_1 the

I/O command is being used as an I/O guard. Control will remain at the output statement until P_1 executes a matching input statement. Similar comments may be made about process P_2 .

Organization of the Thesis

The rest of the dissertation is organized as follows. In Chapter II is an in-depth discussion of RSP, including both the semantics and syntax of the language. In Chapter III I show how to implement some of the key facets of RSP. Chapter IV demonstrates the usefulness of RSP with several examples. Chapter V is organized as a series of questions and answers about RSP. And lastly, in Chapter VI, I provide a comparison of RSP with other real-time languages, summarize the principles which underlie and motivate RSP, and discuss what further research needs to be done.

CHAPTER II

RESPONSIVE SEQUENTIAL PROCESSES

A RSP program consists of a set of n responsive sequential processes $[P_1 // \dots // P_n]$. Each process P_i is like a CSP process, independent of the others and interacts with them by I/O commands. The difference from CSP is in the nature of the I/O commands.

Consider first the output commands. Suppose in P_i , we wish to send a message x to P_j . (*What* a message means is entirely up to the programmer: it might be a request for a service, a piece of data, or anything else. In my discussion I will use both words 'message' and 'request' .) There are two commands that may be used to send x :

1. $P_i \uparrow x$: ("Try x on P_j ") This is typically used as a guard in selection statements and alternative statements; it will succeed if P_j accepts x and fail if it rejects x . Note: To "try" x on P_j does not mean that P_i communicates with P_j to see if it is willing to accept x and if it is willing to accept x then P_i actually sends x . Instead, it means that P_i sends x to P_j and if P_j accepts x then the communication is complete, i.e. x has been sent to and received

ed by P_j . If P_j does not accept x then x was refused by P_j and there is no trace of this communication in P_j .

2. $P_j \Downarrow x$: ("Dump x on P_j ") This will succeed irrespective of whether P_j accepts or rejects x . To execute $P_j \Downarrow x$, P_i sends x to P_j and the communication is complete, irrespective of whether P_j accepts or rejects it. P_i will not have any knowledge of whether P_j accepted or rejected x .

Here is a simple example:

$$P_j \Uparrow x; S_1$$

If P_j accepts x then S_1 will be executed. If P_j rejects x then this process will abort.

$$P_j \Downarrow x; S_1$$

This will send x to P_j and, irrespective of whether P_j accepts or rejects x , go on to do S_1 . P_i will not know whether P_j accepted or rejected x . P_i might use this for sending information to P_j about which P_i does not care whether it is accepted or rejected by P_j .

Here's an example of an output guard

$$[P_j \Uparrow x \rightarrow S_1]$$

This is equivalent to the first example. If P_j accepts x then S_1 is executed, otherwise the process aborts.

What if we want to do S_1 if P_j accepts x and S_2 if P_j rejects it? Note that

$$\begin{aligned} & [P_j \uparrow x \rightarrow S_1 \\ & \quad \square \text{true} \rightarrow S_2 \\ &] \end{aligned}$$

is not the answer; it will indeed do S_2 if P_j rejects x (since the first guard will then fail), but it might choose the second guard without even trying the first guard. To deal with this problem, I introduce a "not-done path" (in addition to the usual "done path") following an I/O guard. Thus in

$$\begin{aligned} & [P_j \uparrow x \rightarrow S_1 \\ & \quad \rightarrow S_2 \\ &] \end{aligned}$$

if P_j accepts x , the done path will be followed and S_1 executed; if P_j rejects x , the not-done path will be followed and S_2 executed.

Consider a somewhat more general example:

$$\begin{aligned} & [b; P_j \uparrow x \rightarrow S_1 \\ & \quad \rightarrow S_2 \\ & \quad \square b'; P_k \uparrow y \rightarrow S_3 \\ &] \end{aligned}$$

Suppose b' evaluates to true, we may pick the second guard and try y on P_k ; if P_k accepts y then S_3 will be executed, if P_k rejects y then we have to try the other guard since there is no not-done path. Suppose b evaluates to

true, we may pick the first guard and try x on P_j ; if P_j accepts x then the done path will be followed and S_1 executed, if P_j rejects x then the not-done path will be followed and S_2 executed. If both b and b' evaluate to false then P_i will abort. If b evaluates to false and b' evaluates to true then P_i will again abort if P_k rejects y . If b evaluates to true then P_i cannot abort.

Note that it doesn't make sense to use \rightarrow with \Downarrow ; thus in

$$\begin{array}{l} [P_j \Downarrow x \rightarrow S_1 \\ \quad \rightarrow S_2 \\] \end{array}$$

S_1 will necessarily be executed, irrespective of whether P_j accepts or rejects x , since in either case $P_j \Downarrow x$ is considered to have been done.

A final example:

$$\begin{array}{l} b := \text{true}; \\ * [b; P_j \Downarrow x \rightarrow b := \text{false} \\ \quad \rightarrow \text{Skip} \\] \end{array}$$

This will keep looping until P_j accepts x , and then b will be set to false and the loop terminates.

Note that the output commands will not cause any wait in P_i since the process P_j , to which P_i sends the message, responds quickly and either accepts or rejects the message. But how does P_j provide this quick response? The answer lies in the system model: in addition to the normal processor

$\mu_{j,1}$ that executes P_j , I assume that there is a second processor $\mu_{j,2}$ that responds to messages sent to P_j . If $\mu_{j,2}$ accepts a message then it adds the message to a buffer it maintains. If $\mu_{j,2}$ rejects the message then the contents of the buffer remain unchanged. In either case, the sender of the message is notified of the acceptance or rejection of the message. When P_j (or rather, $\mu_{j,1}$) executes the next input it reads in the entire contents of $\mu_{j,2}$'s buffer, i.e. the sequence of messages that $\mu_{j,2}$ has accepted since P_j 's last input, and $\mu_{j,2}$'s buffer is cleared.

Clearly then, an input command will not specify a source process, and will read in a sequence of messages rather than a single message. Now for the syntax of input commands:

1. $?h$: ("read non-empty sequence of messages" or "read non-empty h")
As a guard, this will be "done" if the sequence of messages in the buffer is non-empty.
2. $*h$: ("read any sequence of messages" or "read any h") This will always be "done".

Note: "h" is any user-defined sequence variable.

A simple example:

```

b := true;

* [ b; ?h → b := false
    → Skip
  ]

```

This will keep looping until there is some input, at which time the done path is followed and the contents of the buffer are read into h and the loop terminates. As long as there is no input, the not-done path is followed and the skip statement is executed.

This could be equivalently programmed using ? :

```

b := true;

* [ b; ?h → [ h = ε → Skip
              ∨ h ≠ ε → b := false
            ]
  ]

```

So, the ? command is redundant. However, it does add convenience to the notation and it does provide a symmetry between the input and output commands. Note that the \uparrow command is not redundant and cannot be replaced by the \Downarrow command, because if we use the \Downarrow to send a message the sender has no way of knowing whether the message was accepted or rejected.

How does $\mu_{j,2}$ decide whether to accept or reject a particular message? I require, as part of P_j , the specification of an 'acceptance' condition A_j - a Boolean function of the message. When $\mu_{j,2}$ receives a message m , it will accept m if m causes A_j to evaluate to true, and reject m if it causes A_j to evaluate to false. Thus it is $\mu_{j,2}$ that provides the quick response to requests from other processes, either accepting or rejecting them, and it is $\mu_{j,1}$ that actually processes the accepted requests as it executes the body of P_j . In general, the question of whether to accept or reject m will depend on m , on the messages that have already been accepted (and are in $\mu_{j,2}$'s buffer), and on the state of P_j . What I mean by the state of P_j is *not* its current state - that is constantly changing as $\mu_{j,1}$ performs P_j 's local computations and $\mu_{j,2}$ has no access to that state. Instead, the state that A_j refers to is the state of P_j immediately following P_j 's most recent input or output command. Thus, when P_j executes an I/O command, $\mu_{j,2}$ retains the state of P_j following the I/O command until P_j 's next I/O command, and uses it to decide whether to accept or reject messages from other processes (adding messages that it does accept to its buffer to be handed over to P_j at its next input). Underlying RSP is the assumption that a message can be accepted or rejected based upon the message itself, the messages accepted since the last input, and the state of the process at the last I/O command.

Consider a simple example: suppose we wish to program a 'Spooler' process, S, for a line printer to which other processes can send their requests for printing files. A request will be of the form (f, s, t), f being the file to be printed, s its size, and t the (absolute) time by which the output is required. The acceptance condition for the Spooler process is (informally):

$$\begin{aligned}
 A_S = [& \{ \text{time to process this request (i.e. time to print a file} \\
 & \text{of size s)} \\
 & + \text{time to process requests accepted since last input} \\
 & \text{(i.e. the requests in } \mu_{S,2} \text{'s buffer)} \\
 & + \text{time to process the requests that have already been read} \\
 & \text{in by } \mu_{S,1} \text{ but have not yet been processed (this} \\
 & \text{information will be available in the state of S at the time} \\
 & \text{of its most recent input/output)} \\
 & + \text{current time} \} < t \\
 &].
 \end{aligned}$$

This condition will be easy to evaluate since each request contains an indication of the size of the file to be printed. Thus the Spooler process will accept a request only if it can process this request in the specified time, given its earlier commitments and the state of the process at the last I/O.

Suppose S accepts some requests from some other processes. Suppose that before S is able to process the requests the printer jams and it takes a while for it to be fixed (suppose that there is a process dedicated to monitoring the printer for malfunction and when one is detected then it informs the Spooler process). Clearly some of the requests that were

previously accepted are not going to be processed on time. In this situation, the Spooler process will send a message to each affected process. Suppose that P_j is one of these processes. The message from the Spooler might well constitute an "emergency" for P_j , since P_j is proceeding on the assumption that the Spooler will process its request within the specified time, and the emergency might require immediate corrective action on P_j 's part. So, in addition to responding quickly to requests (messages in general), a process needs to be able to *react* quickly to those messages that constitute an emergency for the process. Note that the "reaction" I am talking about is much more than the response that P_j , or rather $\mu_{j,2}$ the second processor associated with P_j , provides when another process sends a message to it. To respond to a message we simply need to check whether or not the message satisfies P_j 's acceptance condition and accept or reject the message on the basis of that check. The quick response is needed to allow the sender of the message to continue with its own activities. On the other hand, a message from S (the Spooler) indicating that it is reniging on a commitment it previously made to P_j may have serious consequences for P_j unless it takes appropriate action immediately. Thus, P_j needs to react quickly to S 's message, not so much to let S continue, as to deal with the emergency that it (P_j) faces.

The next mechanism in RSP is used to specify the 'emergency' messages that P_j must react immediately to, abandoning whatever local

actions it might be performing. As part of P_j , the programmer is required to specify an 'emergency condition' E_j . This condition, like the acceptance condition, is a function of the incoming message m , the state of P_j (at its most recent I/O), and the current contents of the buffer. If m causes E_j to evaluate to true, we have an 'emergency'; $\mu_{j,2}$ will add m to its buffer, and the sending process will see m as having been accepted by P_j . To see what happens next, consider a particular situation:

$?h; S_1; S_2; S_3; \dots$ (S_1, \dots are "local actions" and contain no I/O commands)

Suppose before the message m (that made E_j evaluate to true) arrived, P_j has executed $?h$, and then S_1 , and is in the middle of S_2 when m is received. $\mu_{j,1}$ stops execution, control in P_j is backed up to the input command, the state is restored to what it was at this input command and the input command is reexecuted. So, P_j receives, in h , a sequence that is the concatenation of the messages it received when the input was previously executed, and the messages that $\mu_{j,2}$ has since accepted, including m . $\mu_{j,1}$ now starts again from immediately after the input command. P_j 's state at this point is the same as it was when control was here previously, except that in h we have a longer sequence. To the programmer, it looks as if P_j is just starting to execute S_1 . How do we know that there is an emergency message in h ? Only by examining the individual elements (particularly the last element) of h .

Consider the Spooler process again. Its emergency condition E_S would essentially be $[m = \text{"printer-jammed"}]$. Thus, if the Spooler received the message "printer-jammed" from the printer (which itself is an RSP process but is implemented in hardware), we would have an emergency.

(Some useful functions on sequences:

$\#h$: number of elements in h .

$h[k]$: k^{th} element of h .

So, $h[\#h]$ is the last element in h .)

Suppose now that control is in a portion of S which has the form:

```

* [ ?h → [ h[#h] ≠ 'printer-jammed' → S1
      □ h[#h] = 'printer-jammed' → S2
    ]
  → Skip
]

```

$?h$ is first executed, resulting in the contents of the buffer being read into h . Let us assume that this reads a non-empty sequence of messages into h . Let us also assume that the printer-jammed message has not yet arrived. S_1 begins execution, presumably processing the requests in h . Suppose that as S_1 is being executed, and before the next I/O command is reached, a printer-jammed message arrives. Since this message will cause E_S to evaluate to true, the Spooler has an emergency. Control in S will be backed up to the point of $?h$, the state restored to what it was at that point, h now

will have a sequence which is the concatenation of the previous h and the sequence of messages (including the printer-jammed message) accepted since the earlier execution of $?h$, and then S_2 starts executing (since now $h[\#h] = \text{"printer-jammed"}$). No trace is left of the fact that a portion of S_1 was previously executed before the emergency message arrived.

What if another printer-jammed message arrives as S_2 is executing (and before the next I/O command)? Again control is backed up to $?h$, the state is restored, h now has an even longer sequence containing, in addition to what h had the last time, the messages (including the most recent printer-jammed message) that have been accepted since the last time control was here, and again S_2 is executed (since the last element in h will again be a printer-jammed message). Note that the earlier printer-jammed message will also be in h . But what is the point of doing that when we already know that the printer is jammed? How do we ignore the subsequent printer-jammed messages? The answer is that we simply need to rewrite E_S . If E_S is written as $[m = \text{"printer-jammed"}]$ then every printer-jammed message will indeed be treated as an emergency message even if S already knows that the printer is jammed. Thus to ignore further printer-jammed messages E_S should probably be written as

$[m = \text{"printer-jammed"} \wedge h \text{ does not contain a "printer-jammed" message}]$
 so that a printer-jammed message will be treated as an emergency only if a printer jammed message has not been input by S (i.e. $\mu_{S,1}$) as an element of

h. (We will see the precise form of E_S in the next chapter.) E_S can also be further modified to allow other emergencies while screening out printer-jammed messages; thus we can easily program "prioritized emergencies".

What does the Spooler do once the printer-jammed message is received? In other words, what is S_2 ? Essentially what S_2 should do is to inform, for every request in h , the process that sent the request that its request, although previously accepted by S , will not be processed. We will see the details of how to do this in the next chapter. The important point to note here, however, is that the language imposes no conditions on what S_2 should be. If the programmer so chooses, he could write S_2 to be just "Skip". That would be a rather unreliable Spooler, however, since it would not process some of the requests it accepts and not inform the sending process of the fact.

Let us assume that S does inform the affected processes that their requests, despite having been accepted earlier, are not going to be processed. Suppose that P_j is a process that is informed. What does P_j do when it receives the message from S ? That depends on how P_j has been coded. If the message causes E_j to evaluate to true then we have an emergency and P_j will be backed up to its previous I/O command; etc.. If the message causes E_j to evaluate to false, but causes A_j to evaluate to true, the message will be added to $\mu_{j,2}$'s buffer to be read in the next time P_j executes an input. If neither E_j nor A_j evaluates to true, the message will, of course, be rejected

by P_j (presumably P_j 's programmer felt that it does not really matter whether S processes P_j 's request or not).

One final note about the Spooler: how do we ensure that it does not accept further requests for printing files while the printer is still jammed? Essentially using the same approach we used for screening out printer-jammed messages once it is known that the printer has been jammed; we rewrite A_S , adding a clause to it saying that in addition to the time constraint being satisfied (as we specified earlier), it must be true that we do not have, in h , a printer-jammed message. In fact, A_S should be modified so that once there is a printer-jammed message in h , the only message that S accepts is a "printer-ok" message. A process sending a request to S will not know whether its request was rejected because S could not meet its constraints or because the printer is jammed.

So far I have assumed that when an emergency message arrived, the most recent I/O command was an input. What if it was instead an output command? To see the answer, I have to introduce the full form of the output commands (what we have seen earlier being a simplified version). Suppose P_j wants to 'dump' y on P_k and then proceed to do the local actions (i.e. no I/O commands) S_1 , S_2 , and S_3 . The syntax for this is

$$P_k \uparrow y(h); S_1; S_2; S_3$$

This will send y off to P_k , set h to ϵ , and proceed to do S_1, \dots . Suppose m (that causes E_j to evaluate to true) arrives when $\mu_{j,1}$ is executing S_2 . What

happens is that P_j is backed up to the point immediately following the output, the new value of h is the concatenation of its old value (ϵ) with the contents of $\mu_{j,2}$'s buffer, and then we start the execution of S_1 . To the programmer it looks like P_j is just starting to execute S_1 . How do we know there was an emergency? Again, only by examining h . But this time, if h is non-empty we can conclude that there was an emergency, whereas in the earlier case we had to actually look at the (last) element of h before reaching such a conclusion. Thus, the example above would typically be written as

$$\begin{aligned}
 &P_k \uparrow y(h); [h = \epsilon \rightarrow S_1; S_2; S_3 \\
 &\quad \square h \neq \epsilon \rightarrow \dots \text{deal with emergency} \\
 &\quad]
 \end{aligned}$$

Another example:

$$\begin{aligned}
 &[P_k \uparrow x(h) \rightarrow [h = \epsilon \rightarrow S_1 \square h \neq \epsilon \rightarrow S_2] \\
 &\quad \rightarrow [h = \epsilon \rightarrow S_3 \square h \neq \epsilon \rightarrow S_4] \\
 &\quad]
 \end{aligned}$$

If P_k accepts x , h will be set to ϵ (assuming that there is no emergency), and S_1 will start executing. If, before the next I/O command is reached, an emergency message arrives then control will be backed up to immediately after the output, the contents of the buffer, including the emergency message, will be transferred into h , and then S_2 will be executed (since $h \neq \epsilon$). Similarly, if P_k rejects x and an emergency message arrives when S_3 is

executing, control will be backed up, the buffer contents will be transferred into h and S_4 will begin execution.

In summary, an RSP process P_j consists of:

1. An emergency condition E_j that specifies the condition that a message sent to P_j must satisfy in order to be (accepted and) considered an emergency message by P_j .
2. An acceptance condition A_j that specifies the condition that a message sent to P_j has to satisfy in order to be accepted by P_j .
3. The body of P_j consists of standard local commands (such as Skip, assignment, etc.) and the following I/O:

a. Output:

- i. $P_k \Downarrow x(h)$: "Dump x on P_k ": Sends x to P_k and, irrespective of whether P_k accepts or rejects, P_j continues (and P_j has no knowledge of whether P_k accepted or rejected x); as a guard, $P_k \Downarrow x(h)$ can always be "done", and so we will take the done path following the guard. h will be set to ϵ . If after we do this output, and before we reach the next I/O command, P_j receives an emergency message then we back up to this point, concatenate the contents of the buffer to h and then start execution again.
- ii. $P_k \Uparrow x(h)$: "Try x on P_k ": Similar to (i) but can be "done" only if P_k accepts x . If P_k rejects x , we will take the "not-done"

path following the guard. If there is no not-done path, this guard fails and h will remain unchanged. If used as a command, $P_k \uparrow x(h)$ will cause P_j to abort if P_k rejects x .

b. Input:

- i. $?h$: "Read any h ": Transfers the contents of the buffer into h ; as a guard, this can always be done.
- ii. $?h$: "Read non-empty h ": Similar to (i) but can be done only if the buffer is non-empty. If the buffer is empty and $?h$ is used as a command, P_j will abort. If the buffer is empty, $?h$ is a guard, and there is a not-done path then h will be set to ϵ and we will take the not-done path. If there is no not-done path then h remains unchanged and the guard fails.

c. Guards: A guard is either a Boolean expression or a Boolean expression followed by an I/O. Following a guard we can either have a single path (the "done" path), or two paths (the "done" path and the "not-done" path). The guard fails if the Boolean portion of the guard evaluates to false, or if the I/O portion cannot be done and there is no not-done path. Otherwise (i.e. the Boolean portion evaluates to true and either the I/O can be done or there is a not-done path) the guard succeeds, and we take the done path if the I/O can be done, and the not-done path if the I/O cannot be done. (If the Boolean portion is false then the guard fails

irrespective of whether or not there is a not-done path.)

$\mu_{j,1}$ executes the body of P_j and $\mu_{j,2}$ responds to incoming messages.

Following every I/O command $\mu_{j,2}$ saves the state of P_j ; $\mu_{j,2}$ will use this state in evaluating E_j and A_j to decide what to do with an incoming message, and in backing up P_j when an emergency message arrives. If an incoming message m satisfies E_j then $\mu_{j,2}$ accepts it, adds it to its buffer, $\mu_{j,1}$ stops execution, P_j is backed up to the point of its last I/O command, the state is restored to what it was at that point, and the contents of $\mu_{j,2}$'s buffer are concatenated to the appropriate variable, say h , of P_j (as specified in that last I/O command) and then $\mu_{j,1}$ starts again immediately after the I/O command. If m does not satisfy E_j but does satisfy A_j then $\mu_{j,2}$ accepts it and adds it to its buffer (to be read in at P_j 's next input). If m satisfies neither E_j nor A_j then $\mu_{j,2}$ rejects it.

The programmer can completely ignore the 'backing up' that takes place when an emergency message is received. He can't do anything else, since, for all he can tell, the emergency message arrived just as P_j was executing the I/O command, resulting in the appropriate value appearing in h .

I/O commands are atomic. If P_i were to try to send a message to P_j just as P_j was about to send a message to P_k , the system will let one of them go before the other one.

Finally, the BNF for RSP:

```

<program> ::= [ <process> // ... // <process> ]
<process> ::= <emergency condition> <acceptance condition> <body>
<body> ::= <command>
<command> ::= <skip> | <assignment> | <command> ; <command> |
               <selection> | <repetition> | <input> | <output>
<selection> ::= [ <alternative> [] <alternative> [] ... [] <alternative> ]
<repetition> ::= * [ <alternative> [] <alternative> [] ... [] <alternative> ]
<alternative> ::=   <guard> → <command> |
                   <guard> → <command>
                   ↗→ <command>

```

{ An alternative that has two paths following the guard has the following meaning. If the Boolean portion of the guard is false then the guard fails. If the Boolean portion is true and the I/O portion can be done then the first path ("done path") is taken. If the Boolean portion is true and the I/O portion cannot be done then the second path ("not-done path") is taken. }

```

<guard> ::= <boolean expression> | <input guard> | <output guard>
<input guard> ::= <boolean expression> ; <input>
<output guard> ::= <boolean expression> ; <output>

```

{If in an <input guard> or <output guard> the
<boolean expression> is identically true then it may

be omitted. }

<input> ::= ? <sequence variable> | ? <sequence variable>

{When a process executes an input command it reads in the entire sequence of messages that have been accepted since the last input.}

**<output> ::= <process name> ↑ <message> (<sequence variable>) |
<process name> ↓ <message> (<sequence variable>)**

**<emergency condition> ::= [?E (<message> , <process name> ,
<sequence variable>) ≡ <boolean expression>]**

{ The <process name> is the name of the sending process. It is useful to be able to decide whether a message is an emergency based upon what process sent the message. }

**<acceptance condition> ::= [?A (< message> , <process name> ,
<sequence variable>) ≡ <boolean expression>]**

{ Again, <process name> is the name of the process that sent the message. Often it is desirable to accept or reject a message based upon which process sent the message. }

<sequence variable> ::= <variable>

CHAPTER III

SYSTEM MODEL

In the previous chapter I made several references to the system model. In this chapter I propose a possible implementation of two key facets of RSP: (1) whenever an emergency message is received the state is restored to that which existed following the last I/O. How can this be done efficiently and quickly? And, (2) how is the communication mechanism to be implemented? In the situation where two processes wish to communicate with each other at the same time how does the system ensure that one process communicates before another? More specifically, if a process P receives a message while it is waiting for a response from a message it sent out earlier, what should be done with the message it received?

Let us start with the first problem. Recall that in process P_j , it is $\mu_{j,1}$ which executes the code in P_j , and it is $\mu_{j,2}$ which interfaces with the other processes - accepting or rejecting input messages and restoring the state to that which existed at the last I/O, if an input satisfies the emergency condition. How is the state to be saved? Is there an efficient method for

storing and later restoring the state?

In the following discussion I show how we can quickly and efficiently restore the state, at the expense of doubling the memory size. (Note: with the rapidly diminishing cost of memory, this does not seem to be an onerous expense.) Suppose that we have a dual memory, where each word of memory has a "partner word". Also, let me distinguish between the left side (of the dual memory) and the right side. The left side of memory will be the "active side". This is the side of the memory that $\mu_{j,1}$ has access to, and which it modifies. The right side of memory is the "passive side". It is used to store the state which existed at the last I/O. Recall that $\mu_{j,2}$ bases (in part) its decision to accept or reject an input based upon the state of the process, i.e. the state which existed at the last I/O. So, $\mu_{j,2}$ will use this right side in its decision of whether or not to accept an input.

Consider the following program segment: I/O; S1; S2; S3; Suppose that S1, ... are local statements. Suppose that control is in one of the local statements, when a message arrives which satisfies the process' emergency condition. The state needs to be quickly restored to that which existed at the I/O statement. With respect to the dual memory, this means that the left side must be set to equal to right side. We can quickly and efficiently accomplish this by copying the right side of memory into the left side. In Figure 1 (pg 39), I have shown the configuration of the dual memory. Each word in the left memory is connected to its partner word in the right

memory. When an emergency is detected, a signal is sent on the Regenerate State line, and this results in loading, in parallel, the words from the right side to the left side, i.e. we have, in one instruction, restored the state.

What happens when control finally reaches the next I/O statement? We need to get the right side to reflect the new state, which is in the left side. So, when control reaches the next I/O statement a signal is sent on the Update State line, which results in loading, in parallel, the new state (in the left side), into the right side.

It is important to note that the restoring and updating of the state as described above is to be done in parallel, independent of the size of the memory. It is not a block move, where a block of memory is moved to an arbitrary location. Each word in the left memory is moved to its partner word in the right memory, or vice versa. The time required for this operation is depends on the length of the memory word, not on the number of words in memory.

The communication mechanism may be implemented as follows. Consider some process P. If P is doing local computations when an input from process Q arrives then P (more specifically, $\mu_{p,2}$) can immediately respond. Of course, if the input satisfies the emergency condition then the state will be restored to the state which existed at the last I/O statement. Suppose P outputs to S. While P awaits a response from S, if $\mu_{p,2}$ receives an

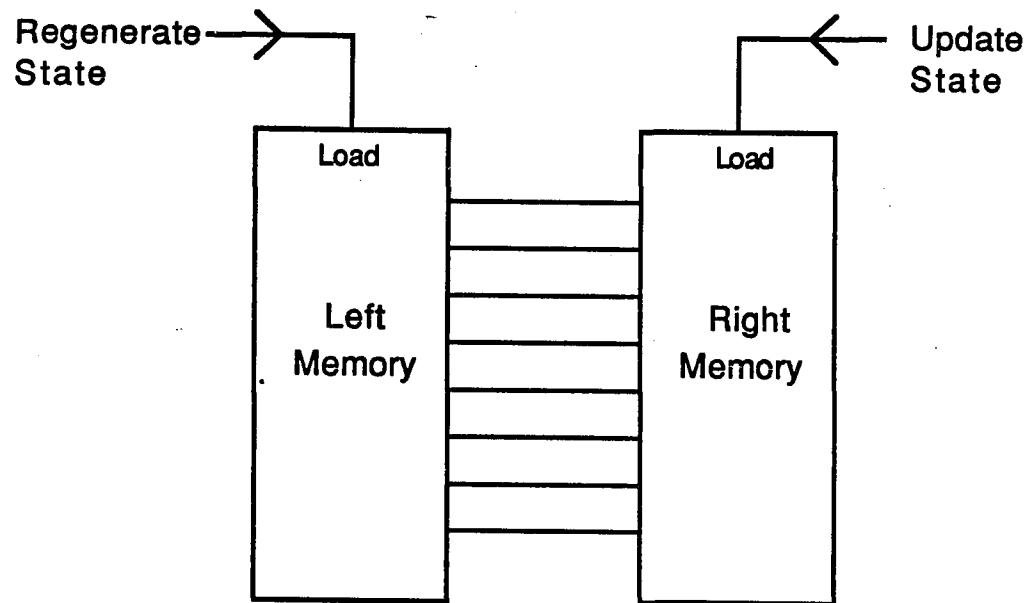


Figure 1
Achieving Efficient State Roll-back and Update

input from Q then $\mu_{p,2}$ replies immediately to Q with a "try-again-later" (TAL) message. This message means that P is in a transition state and is unable to accept or reject a Q's message since it is unclear whether the emergency and acceptance conditions should be evaluated based upon the state which existed at the previous I/O or, based upon the state which would exist if the output message is accepted. Let me refer to this state as the Outputting state, and the previous state as the Not Outputting state. If $\mu_{p,2}$ receives an accept or reject response from S then it goes back to the Not Outputting state. If it receives a TAL message from S then it moves to a Stalled state. While in the Stalled state we may consider control to be just prior to the output statement. After a time P will output to S again, thus taking $\mu_{p,2}$ back to the Outputting state. While in the Stalled state, if an input from Q is received and the input satisfies the emergency condition, based upon the state which existed at the last I/O statement, then P will accept the input, restore the state, and go back to the Not Outputting state. If the input does not satisfy the emergency condition then P will respond based upon the state which existed at the last I/O and remain in the Stalled state, until it outputs to S again.

In the above example, if P and Q wait the same amount of time in their Stalled state then they may oscillate back and forth between the Outputting state and the Stalled state. To prevent this, each process needs to spend a different amount of time in their respective Stalled states. This may

be accomplished by numbering each process and each process remains in its Stalled state for a time proportional to its process number. On the next page is a state diagram which shows pictorially what has been described above.

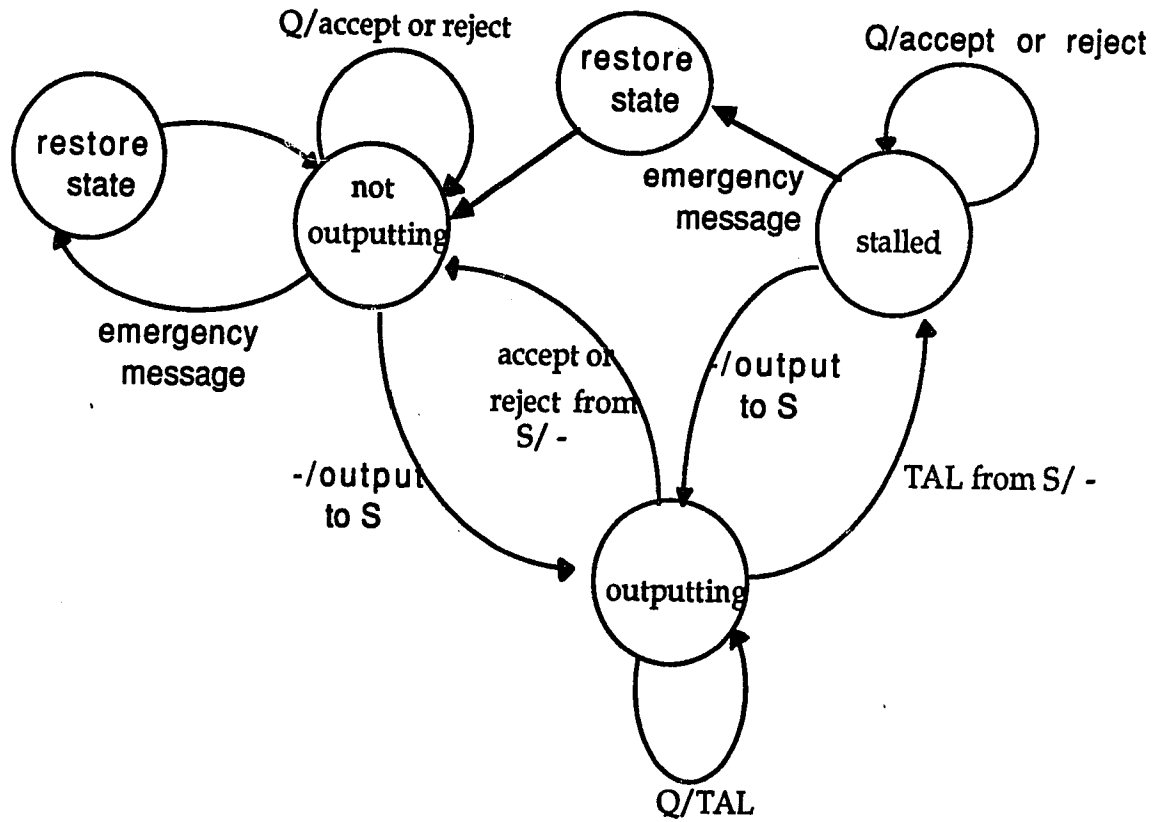


Figure 2
Communication Protocol State Diagram

Notation: input / output

CHAPTER IV

EXAMPLES

This chapter is comprised of five examples. The purpose of these examples is to illustrate how RSP may be used to program typical problems of the type mentioned in Chapter I. The first example, although not a real-time example, is especially appropriate for showing the usefulness of the emergency mechanism. The next two examples illustrates the response mechanism. And the last two examples nicely demonstrate some of the programming features of RSP programs.

Recall my notation: $\#h$ is the number of elements in h . $h[k]$ is the k^{th} element of h . Often, I have to refer to the last element of h ; I can, of course, write this as $h[\#h]$; for convenience I will allow this to be written as $h[-1]$. In general, if k is negative, $h[k]$ refers to the $|k|^{\text{th}}$ element from the right end of h .

I will use ' β ' to refer to the buffer of a process and π to refer to the process which sent the message. Thus the acceptance and emergency conditions of a process will refer to the message (usually denoted " m ") under consideration, β , π , and the variables (i.e. the state) of the process.

Example 1. Prime Number Example

This example shows how a set of RSP processes may be used to determine whether or not a number is prime. It demonstrates the general applicability of RSP's emergency mechanism. The emergency mechanism is used in stopping processes from performing useless computations, and these might arise in real-time as well as non-real-time programs.

Suppose that we wish to determine all of the prime numbers from three to infinity. Let us assume that there are $r+1$ processes, $P[0] \dots P[r]$. Given some number N we can decide if N is prime by seeing if any number between 2 and \sqrt{N} evenly divides N . Suppose we partition this job over the r processes, $P[1] \dots P[r]$. Process $P[i]$ will check all of the numbers from $((i-1)\sqrt{N}/r)+2$ to $(i\sqrt{N}/r)+1$ to see if any of them evenly divides N . Obviously, if any of the processes finds a number which evenly divides N then N is not prime. In addition, if one of the processes detects that N is not prime then it is pointless for any of the processes to continue with their computations. This is an instance where the emergency mechanism comes in very handy. Suppose that $P[0]$ is the controlling process and each process $P[i]$ ($i = 1..r$) receives from $P[0]$ the value N which is to be checked and returns to $P[0]$ a Boolean indication of whether or not it found, based upon the subset of numbers that it checked, that N is prime. If a process finds a number that evenly divides N then it will send a false to $P[0]$ to inform it that the number is not prime. Once $P[0]$ receives a false from any $P[i]$ then it knows that this is not a prime, and it does not have to wait for messages

from other $P[i]$. In fact, it can send these other $P[i]$ a 'stop' message so that they can also discontinue their check for primality of N . I will do this by treating the receipt of a 'false' message by $P[0]$ as an emergency for it. $P[0]$ will react by sending a 'stop' to the various $P[i]$. Each $P[i]$ will in turn treat a 'stop' message as an emergency and react by abandoning the check for primality of N and wait for the next N from $P[0]$.

Processes $P[i]$ ($i = 1 \dots r$) may be coded thus:

$P[i] :: [\{ ?E(m, \pi, \beta) \equiv (\pi = P[0] \wedge m = \text{'stop'}) \}$

$\{ ?A(m, \pi, \beta) \equiv \pi = P[0] \}$

$* [?h \rightarrow [h[-1] = \text{'stop'} \rightarrow \text{Skip}$

$\square h[-1] \neq \text{'stop'} \rightarrow N := h[-1];$

$k := \lfloor \sqrt{N} \rfloor;$

$[k \leq r \rightarrow lo := 2; hi := k$

$\square k > r \rightarrow lo := ((i-1) \cdot k \text{ div } r) + 2;$

$hi := (i \cdot k \text{ div } r) + 1$

$];$

$p := \text{true};$

$j := lo;$

$* [(j \leq hi) \wedge p \rightarrow p := (N \bmod j = 0); j := j + 1]$

$P[0] \Downarrow p(h)$

$]$

$\rightarrow \text{Skip} \quad]]$

The emergency condition states that a 'stop' message from $P[0]$ is an emergency message for $P[i]$; and the acceptance condition states that $P[i]$ will accept all messages from $P[0]$. The number to be checked for primality is read into h and then placed into a local variable, N . The range of values to be tested are computed and then for each value, j , in the range, test to see if j evenly divides N . If j evenly divides N then send a false (not prime) to $P[0]$. If none of the values evenly divides N then send $P[0]$ a true (as far as this process can tell N is prime). While the values are being tested by $P[i]$ a 'stop' message may arrive from $P[0]$, indicating that the number is not prime (one of the other processes detected this and informed $P[0]$, and $P[0]$ relays this fact to the processes). This message is an emergency for this process and will result in the state being backed up to the input guard and the process will start all over with the next N .

Note that I could have written $N := h[-1]$ as $N := h[1]$ since if the last element in h (i.e. $h[-1]$) is not 'stop', then there must be only one element in h , so I can refer to it as either $h[1]$ or $h[-1]$, and it must be the next number to be checked for primality. However, $h[-1] = \text{'stop'}$ cannot be replaced by $h[1] = \text{'stop'}$ since we might have two elements in h , the first message being N , the second being 'stop'.

Next consider $P[0]$:

$P[0] :: [[?E(m, \pi, \beta) \equiv (m = \underline{\text{false}})]$

$[?A(m, \pi, \beta) \equiv \underline{\text{true}}]$

$N := 3;$

$* [\underline{\text{true}} \rightarrow i := 1;$

$p := \underline{\text{true}};$

$* [i \leq r \wedge p; P[i] \uparrow N(h) \rightarrow [h = \epsilon \rightarrow i := i + 1 \sqcap h \neq \epsilon \rightarrow p := \underline{\text{false}}]$

$\rightarrow [h = \epsilon \rightarrow \underline{\text{Skip}} \sqcap h \neq \epsilon \rightarrow p := \underline{\text{false}}]$

$];$

$* [i > 1 \wedge p; ? h \rightarrow [h[-1] = \underline{\text{false}} \rightarrow p := \underline{\text{false}}$

$\sqcap h[-1] = \underline{\text{true}} \rightarrow i := i - \#h$

$]]$

$\rightarrow \underline{\text{Skip}}$

$];$

$[p \rightarrow \underline{\text{Skip}}$

$\sqcap \neg p \rightarrow i := 1; * [i \leq r \rightarrow P[i] \uparrow \text{'stop'}(h); i := i + 1; ? h$

$];$

$Q \uparrow (N, p)(h);$

$N := N + 2$

$]]$

$]]$

$P[0]$ sends N to $P[1], \dots, P[r]$ so that each can check for primality of N in the appropriate range. (It would have been simpler and correct to use the \Downarrow to send N to $P[i]$ since $P[i]$ accepts all messages from $P[0]$; in general, however, as a matter of programming methodology, a process that wants to be sure that the receiver really accepts the message, should use \Uparrow .) If, before this sending is complete, one of the processes finds that N is not prime then it will send back a false to $P[0]$ which will treat the message as an emergency, and set p to false; the sending loop will terminate and $P[0]$ will send the message (N, false) to Q , informing the process Q that N is not prime.

If $P[0]$ sends N to $P[1], \dots, P[r]$, it will then wait for a message from each of them. If $P[0]$ receives r true messages then it can be sure that N is prime since each of the $P[i]$ ($i = 1, \dots, r$) failed to find a factor for N in the range that it was responsible for. $P[0]$ keeps track of how many true messages it has received by counting i down to 1. (If $h[-1] = \text{true}$ then we can be sure that all the earlier messages, if any, must be true since if any of them were false then as soon as it was received $P[0]$ would have backed up and taken the $h[-1] = \text{false}$ path since it would have been an emergency message.) If $P[0]$ receives a false from any of them, it stops waiting for further messages, sends (N, false) to Q , and a 'stop' to $P[1], \dots, P[r]$ so that any of them that is still checking for primality of N can abandon the check. ($P[0]$ sends the 'stop' message to $P[1], \dots, P[r]$ also in the case that it receives a false during its sending loop.) The last input command flushes the buffer before I start consideration of the next N .

Example 2. Line Printer Spooler Example

In this example I write a process which serves in spooling files to a line printer. This example was discussed briefly in chapter II, when RSP was being introduced. It is a simple example with non-trivial acceptance and emergency conditions to show how such conditions might be written.

The Spooler process S receives requests of the form (f, s, t) , f being the file to be printed, s the size of the file, and t the time by which the file is required to be printed. S might also receive "p_j" (printer-jammed) and "p_f" (printer-fixed) messages from the line printer. S keeps track of the printer status in a Boolean variable p_up (printer-up), which is true if the printer is working and false otherwise.

The acceptance condition for S accepts a print request (f,s,t) if a file of size s can be printed within the required time t , given whatever other print requests S has already accepted (these requests will be in β ($\mu_{S,2}$'s buffer) and the variable h); of course if a "p_j" message has been received, or p_up is false, then S will not accept any print requests.

A "p_j" message is an emergency message unless such a message has already been received and is in h or h' , or unless p_up is already false. When the emergency message is received, p_up will be set to false (after which the only message that S will accept is "p_f"), and renig on its earlier commitments (i.e. it will return all of the print requests it previously accepted to the respective processes along with an indication that it is renig on the commitment to process that request that it had made

earlier). Note that S has to do this not only for the requests that it (i.e. $\mu_{S,1}$) has already input, i.e. the requests in h , but also those it has accepted since its last input; these will be in the buffer when the emergency is received and will be read into h or h' (along with the emergency message).

S :: [[?E (m, π , β) \equiv {m="p_j" \wedge "p_j" \notin h \circ h' \wedge p_up = true}] % see note 1

below

[?A (m, π , β) \equiv { [m="p_f" \wedge "p_f" \notin h \circ h' \wedge p_up = false]

\vee [m \neq "p_j" \wedge m \neq "p_f" \wedge p_up = true

\wedge "p_j" \notin h \circ h'

\wedge m.t > (τ + g(m) + f(h \circ β))] }] % see

note 2 below

h := ϵ ;

p_up := true;

h' := ϵ ;

* [p_up; ?h \rightarrow [h[-1] = "p_j" \rightarrow p_up := false

\square h[-1] \neq "p_j" \rightarrow ... process requests in h ... % see

below

];

[p_up \rightarrow Skip

\square \neg p_up \rightarrow ... renig on commitments in h \circ h' ...

%see below

]

\rightarrow Skip

\square \neg p_up; ?h \rightarrow p_up := true

\rightarrow Skip

]

Note 1: $h \circ h'$ is the concatenation of the sequences h and h' ; " p_j " $\notin h \circ h'$ means no element of $h \circ h'$ is " p_j "; thus S will accept a " p_j " message (as an emergency message) only if it has not already received a " p_j " and only if p_up is true.

Note 2: The first clause of $?A$ is similar to $?E$. A print request m will be accepted only if the printer is up, a " p_j " message has not been received, and the request can be processed within the specified time by which this output is required; $m.t$, the third component of m is this specified time; τ is the current time; $g(m)$ is the time required to print a file of size $m.s$ (recall that the s specifies the size of the file); g is a function that, given a request m , gives us the time for printing a file of size $m.s$. $f(h \circ \beta)$ is the time for processing the requests in $h \circ \beta$; f is a function that maps a sequence to the time required to process all the requests in the sequence by summing up the results obtained by applying g to each element in the sequence.

process requests in h:

```

* [ p_up ∧ h ≠ ε → r := h[1];
    i := 1;

    * [ p_up ∧ i ≤ r.s; LP↑r.f[i](h') → i := i + 1;

        [ h' = ε → Skip
          □ h' ≠ ε → p_up :=
                                false
        ]

    → [ h' = ε → Skip
        □ h' ≠ ε → p_up :=
                                false
        ]

    ];

[ p_up → h := h[2 .. ] □ ¬p_up → Skip ]
]

```

For each request r in h , S forwards the file $r.f$, line-by-line, to the line printer LP . Once $r.f$ is forwarded, we can remove r from h and do the same for the next request. Any time during this a "p_j" message might arrive in which case S will read in the entire contents of the buffer into h' and set p_up to false, thus terminating the loops. We will then renig on the

commitments in $h \circ h'$. $h[2 \dots]$ is the sequence obtained by deleting the first element of h .

reniging on commitments in $h \circ h'$:

Suppose that r is a request in $h \circ h'$. In order to renig. on S 's commitment to process r we need to know not only r but also the process that sent r . I will assume that given a sequence of messages h , $h^\pi[k]$ will be the process that sent the k^{th} message in h .

$h := h \circ h'$;

$h' := \epsilon$;

$r := h[1]$;

$p := h^\pi[1]$;

$* [r \neq "p_j" \rightarrow p \uparrow(r, "renig")(h'); h := h[2 \dots]; r := h[1]; p := h^\pi[1]]$

Note: I have assumed that the files to be printed are small. If they are even moderately large, it would be unreasonable to send the complete file in a single communication. In this case, a more reasonable solution would be for the process making the request to send only the name of the file, its size, and the time by which the output is required. If the request is accepted then S (or possibly another subordinate process) will have to communicate with the requestor process to obtain the actual file (line-by-line or block-by-block) (see Example 3).

Example 3. Line Printer Spooler Example, version 2

In this example the Spooler process has been revised to include a subordinate process. This version is geared for spooling large files. It is the first example where the acceptance condition bases its acceptance on the process which is sending the message.

In the first version I had assumed that the files to be printed are small. So, the entire file can be sent in a single communication. In this version I do not assume small files only. To avoid tying up the communication lines, and hence slowing down the responsiveness of the spooler, I introduce an additional process. Now the Spooler is composed of two processes, S[1] and S[2]. S[1] is similar to S in the previous version and schedules the print requests as they arrive. S[2] is subordinate to S[1] and is responsible for getting the actual file to be printed from the requestor and sending it, line-by-line, to the line-printer.

Let us first consider S[1]. Now when a process makes a request for a file to be printed it sends only two values: s, the size of the file to be printed and t, the time that the file is to be printed. I assume that the "p_j" and "p_f" messages are sent to both S[1] and S[2]. The code for S[1] is the same as S in the previous version except for processing requests in h.

$S[1] :: [[?E(m, \pi, \beta) \equiv \{m = "p_j" \wedge "p_j" \notin h \circ h' \wedge p_up = \underline{true}\}]$

$[?A(m, \pi, \beta) \equiv \{ [m = "p_f" \wedge "p_f" \notin h \circ h' \wedge p_up = \underline{false}]$

$\vee [m \neq "p_j" \wedge m \neq "p_f" \wedge p_up = \underline{true} \wedge$

$"p_j" \notin h \circ h'$

$\wedge m.t > (\tau + g(m) + f(h \circ \beta))]]]$

$h := \epsilon;$

$p_up := \underline{true};$

$h' := \epsilon;$

$* [p_up; ?h \rightarrow [h[-1] = "p_j" \rightarrow p_up := \underline{false}$

$\square h[-1] \neq "p_j" \rightarrow \dots \text{process requests in } h \dots \text{ \% see}$

next page

$];$

$[p_up \rightarrow \underline{Skip}$

$\square \neg p_up \rightarrow \dots \text{renig on commitments in } h \circ h' \dots$

%see next page

$];$

$\rightarrow \underline{Skip}$

$\square \neg p_up; ?h \rightarrow p_up := \underline{true}$

$\rightarrow \underline{Skip}$

$]]$

S[1] does only the scheduling of the requests. S[2] is actually responsible for processing each request. So, what it means for S[1] to process its requests is that it sends each request, one at a time, to S[2]. S[1] sends to S[2] a triple: p , the name of the requesting process, s , the size of the file to be printed and t , the time by which S[2] should finish processing this request. t is calculated as the current time plus the time required for fetching the file (assuming the process sends the file promptly when asked) and then sending it to the line-printer.

process requests in h:

```

* [ p_up  $\wedge$   $h \neq \epsilon \rightarrow p := h^\pi[1];$ 
     $s := h[1].s;$ 
     $t := \tau + g(h[1]);$ 
     $d := \underline{\text{false}};$ 
    * [  $\neg d \wedge p\_up; S[2] \uparrow(p,s,t)(h') \rightarrow h := h[2 \dots];$ 
        [  $h' = \epsilon \rightarrow d := \underline{\text{true}}$ 
          [  $h' \neq \epsilon \rightarrow p\_up := \underline{\text{false}}$ 
          ]
        ]
     $\rightarrow [ h' = \epsilon \rightarrow \underline{\text{Skip}}$ 
        [  $h' \neq \epsilon \rightarrow p\_up := \underline{\text{false}}$ 
        ]
    ]
]

```

Each request is repeatedly sent to S[2] until it is accepted. Note that when the request is accepted it is removed before checking for an emergency. This is because it will be S[2]'s responsibility for reniging on this request if a "p_j" message arrives. (Recall that S[2] will also receive the "p_j" message.) Reniging on commitments is unchanged:

reniging on commitments in hoh':

$h := h \circ h';$

$h' := \epsilon;$

$r := h[1];$

$p := h^\pi[1];$

$* [r \neq "p_j" \rightarrow p \Downarrow(r, "renig")(h'); h := h[2 \dots]; r := h[1]; p := h^\pi[1]]$

Now for S[2]: S[2] operates by first getting a request from S[1] and then processing the request, and then repeating. To process a request S[2] fetches the file from the requestor and then sends it, line-by-line, to the line-printer. Emergencies are handled in the same fashion as in S[1].

The acceptance condition is modified from S[1] to allow S[2] to oscillate between getting a request from S[1] and processing the request. Within S[2] there are two variables, gr (get a request) and pr (process the request). gr is set to true when S[2] wants to get a request from S[1] and pr is

set to true when S[2] want to process the request. So, a request is accepted from S[1] only if $gr = \text{true}$ and we have finished processing the last request. If $pr = \text{true}$ then only messages from the requestor is accepted.

$S[2] :: [[?E (m, \pi, \beta) \equiv \{ m = "p_j" \wedge "p_j" \notin h \circ h' \wedge p_up = \underline{true} \}]$
 $[?A (m, \pi, \beta) \equiv \{ [m = "p_f" \wedge "p_f" \notin h \circ h' \wedge p_up = \underline{false}]$
 $\vee [m \neq "p_j" \wedge m \neq "p_f" \wedge p_up = \underline{true} \wedge$
 $"p_j" \notin h \circ h'$
 $\wedge \pi = S[1] \wedge gr \wedge h^\pi[1] \neq S[1]]$
 $\vee [m \neq "p_j" \wedge m \neq "p_f" \wedge p_up = \underline{true} \wedge$
 $"p_j" \notin h \circ h'$
 $\wedge \pi = p \wedge pr] \}]$
 $gr := \underline{true};$
 $pr := \underline{false};$
 $p_up := \underline{true};$
 $* [p_up; ?h \rightarrow [h[-1] = "p_j" \rightarrow p_up := \underline{false}$
 $\quad [h[-1] \neq "p_j" \rightarrow \dots \text{process request in } h \dots$
 $\quad];$
 $\quad [p_up \rightarrow \underline{Skip}$
 $\quad [\neg p_up \rightarrow \dots \text{renig on commitment in } h \dots$
 $\quad]$
 $\quad \rightarrow \underline{Skip}$
 $\quad [\neg p_up; ?h \rightarrow p_up := \underline{true}$
 $\quad \quad \rightarrow \underline{Skip}$
 $]]$

Processing a request consists of three steps: the first step is to send a message to the requestor, telling it to send its file. The second step is to receive the file from the requestor and store it in $S[2]$'s own buffer, f . The third step is now to transfer the file to the line-printer.

Throughout these three steps $S[2]$ must keep track of whether or not it is keeping within the time limit, t . If the requestor is delinquent in sending its file then there will not be enough time to receive the file and send it onto the line-printer. So, during these three steps we keep checking that the current time is less than t . If the current time exceeds t then a message is sent to the requestor informing it that we are unable to print its file since it was too slow in sending the file.

process request in h:

```

gr := false;
pr := true;
p := h[1].p;
t := h[1].t;
s := h[1].s;
p $\Downarrow$ ("send file")(h'); % step 1 (see above)
[h' =  $\epsilon$   $\rightarrow$  Skip
   $\square$  h'  $\neq \epsilon \rightarrow$  p_up := false
];
i := 1;
% step 2 (see above)
* [ ( $\tau < t$ )  $\wedge$  p_up  $\wedge$  (i < s); ?h'  $\rightarrow$  [ h'[-1] = "p_j"  $\rightarrow$  p_up := false
                                      $\square$  h'[-1]  $\neq$  "p_j"  $\rightarrow$  * [ ( h'  $\neq \epsilon$ )  $\wedge$  ( $\tau < t$ )  $\wedge$  (i < s)  $\rightarrow$ 
                                                                 f[i] := h'[1];
                                                                 h' := h'[2..];
                                                                 i := i + 1
                                     ]
                                     ]
                                      $\rightarrow$  Skip
];

```

```

i := 1;
% step 3 (see above)

* [ (τ < t) ∧ p_up ∧ (i < s); LP↑f[i](h') → [ h' = ε → i := i + 1
                                         □ h' ≠ ε → p_up := false
                                         ]
  ↪ [ h' = ε → Skip
    □ h' ≠ ε → p_up := false
    ]
];

[ (τ ≥ t) ∧ (i < s) → msg := "File transfer too slow. Could not finish request."
  p↓msg(h');
  [ h' = ε → Skip
    □ h' ≠ ε → p_up := false
  ];
  h := ε

□ i = s → h := ε

□ ¬p_up ∧ (i < s) → Skip
];
pr := false;
gr := true;
?h';      % flush β of any residual messages from p

```

```

[h'[-1]≠"p_j" → Skip
  □ h'[-1]="p_j" → p_up := true
];

```

There will be at most one request to renig on and it will be in h. In the event that a "p_j" message arrived after sending the file to the line-printer then there will be no messages in h.

renig on commitment in h:

```

[ h = ε → Skip
  □ h≠ε → r := h[1];
    [ r="p_j" → Skip □ r≠"p_j" → p:= h[1].p; p↓(r, "renig")(h') ]
]

```

Example 4. Alarm Clock

This process serves as an alarm clock for other processes. A process sends a wake-up time to this process and it sends back, at the designated time, a wake-up message. This is the first example where time is referred to directly in the program (of course, it has been used in prior examples in the acceptance and emergency conditions).

Alarm requests are kept in a buffer h . It is not possible at each point in time to see if a process needs to be waken-up, so I check to see, within an interval of time, what processes needs waking. t_{old} and t_{new} are used to mark the interval. Initially, t_{new} is set to infinity. This will prohibit acceptance of any requests the first time through the loop (see below), since the acceptance condition requires the request's time to be greater than t_{new} . If t_{new} had been initialized to τ (current time) then all requests accepted during the time interval τ and the time when $t_{new} := \tau$ is executed in the first iteration will not be considered in the second iteration. (If an input is received before a process executes its first I/O statement then the process will reject the message, i.e. the emergency and acceptance conditions evaluate to false no matter what messages are received. See Chapter V, Q10 for a more complete discussion of the problem of initialization.) Within the loop, h is concatenated with the new requests, t_{new} is updated to the current time and then I check each request in h to see if a request's time lies within the interval and if it does I issue a wake-up message to that process and delete the process from h . If it does not then I simply move onto the

next request in h . After this t_{old} is incremented to t_{new} , and I loop around again, reading in any new requests and checking for requests within the new interval.

Alarm_Clock ::

[?E (m, π, β) \equiv false]

[?A (m, π, β) \equiv $m.time > t_{new}$]

$t_{old} := \tau;$

$t_{new} := +\infty;$

$h := \varepsilon;$

$h' := \varepsilon;$

* [? $h' \rightarrow h := h \circ h';$

$t_{new} := \tau;$

$i := 1;$

* [$i \leq \#h \rightarrow [t_{old} \leq h[i].time \leq t_{new} \rightarrow p := h^\pi[i](h);$

$p \Downarrow ("wakeup")(h);$

% remove the i^{th}

message in h :

$h := h - h[i]$

$\square h[i] > t_{new} \rightarrow i := i + 1]$

]

$t_{old} := t_{new}$]

Example 5. Alarm Clock, version 2.

In this version of the alarm clock I have eliminated t_{old} and now we just have t_{new} , which I call t . Instead of checking an interval I now just check for any messages with requests before t . This example demonstrates how time may be easily programmed for in RSP.

Alarm_Clock ::

[?E (m, π , β) \equiv false]

[?A (m, π , β) \equiv m.time > t]

t := τ ;

h := ε ;

h' := ε ;

* [?h' \rightarrow h := h \circ h' ;

t := τ ;

i := 1;

* [i \leq #h \rightarrow [h[i].time \leq t \rightarrow p := h $^{\pi}$ [i](h);

p \Downarrow ("wakeup")(h);

h := h - h[i]

□ h[i] > t \rightarrow i := i + 1

]

]

]

CHAPTER V

DISCUSSION

This chapter is organized as a series of questions and answers about RSP and how it relates to CSP and other models for distributed real-time systems.

Q1: The problem with using CSP for real-time systems is synchronous communication: the sender may have to wait for a long time for the receiver to receive its request; why not solve this problem by going to an asynchronous model (or, for that matter, simulating asynchronous communication with a CSP process that acts as a buffer)?

Ans: The problem cannot be solved by using an asynchronous model. If a process makes a request of another process S, it is not enough for P to be able to send the request off. P also needs to know whether or not S will process the request within its (P's) time constraint, since what it should do next will, in general, depend very much on whether S can process its request; if S will process the request in the specified time then P can proceed with its other activities; if not, then P will have to make alternative

arrangements, such as sending the request to another process S' . It is this consideration that led to the idea of responsive processes.

Q2: So is the RSP communication model synchronous or asynchronous?

Ans: In a way it is both - the sender P sees the receiver S as responding quickly to messages, accepting some and rejecting others; so to P the communication seems quick and synchronous. In S , however, the communication seems asynchronous, each input command reading in a sequence of values from S 's buffer.

Q3: Since I don't have to actually refer directly to its buffer, can't we adopt a synchronous view in S as well, and imagine that all the messages read in at an input command were just received?

Ans: The problem is that the messages were accepted on the basis of the states at earlier I/O commands. If A_S does not depend very much on the state, or if the state does not change from one I/O command to the next then I may indeed imagine that all messages read in at the input were just received.

Q4: RSP requires the acceptance condition of a process to be based only on the message received, on the messages already accepted, and on the state of the process at its most recent I/O command. In general, can we write an

appropriate acceptance condition satisfying this requirement?

Ans: Let us consider the Spooler S again. Suppose a request to S was of the form (f, t) , f being the name of the file to be printed, and t the time by which the file is required to be printed. Now it would be impossible to write an acceptance condition since in S we don't have enough information to decide how much time it would take to process each request. Of course, S could try to obtain the size of the file by communicating with the requestor (say) P. But that would be against the idea of responsive processing since it might be a while before the $(\mu_S, 1)$ in the Spooler actually looks at this request, and communicates with P, and in the meantime P has no idea whether or not S will ultimately accept the request. The solution, at least in this example, was simple: P must include the size of f in its original request. In general I believe that it is always possible to program in such a way that a process can decide whether to accept or reject a message on the basis of the message, the identity of the sending process, the current contents of the buffer, and the state at the last I/O command.

Note that all of the above remarks apply to the emergency condition also.

Q5: In RSP how do we distinguish between various types of messages such as data, requests, emergencies, etc?

Ans: I don't. It is entirely up to the programmer to decide which messages are interpreted as data, which are interpreted as requests, etc. The same message that is treated as an emergency if control is in one part of the process might be treated as a "normal" message to be read in at the next input command if control is in a second part of the process, and even rejected if control is in a third part of the process. It all depends on the details of the acceptance condition and the emergency condition.

Q6: Aren't emergency messages really just interrupts? And why don't we have emergency service routines analogous to the interrupt service routines?

Ans: Interrupts are often used to provide quick response to the sending process P, allowing it to continue with its activities. In RSP this quick response is provided by $\mu_{S,2}$ and the acceptance condition A_S of the receiving process S. The purpose of the emergency mechanism is, however, quite different; there are certain messages that call for quick action by S since if such action is not taken then serious problems could arise for S, the receiving process. In all probability, the sending process does not even care what S does with the message.

One of the problems with interrupts is the interrupt service routine. At unpredictable points in a process the state changes - because an interrupt was received and the interrupt service routine was executed and this

routine changed the state. This makes it hard to understand the process. I tried to design the emergency mechanism to avoid this problem. Although control in a process is backed up to the last I/O command when an emergency message is received, the programmer can completely ignore this; he can just imagine that this message was received and input like all other messages. The state does not change unexpectedly since there is no emergency service routine which is implicitly invoked at any point in the process. Instead, following each I/O command we have to write code to deal with certain important messages - the emergency messages - if such were to be received at that I/O command. But doesn't that lead to considerable code repetition? It might, but most of that can be eliminated by using procedures in a fairly direct fashion. These procedures will, in effect, be the emergency service routines, but they will be invoked explicitly, thereby avoiding the problems of unexpected state changes that we would have if they were to be invoked implicitly.

Q7: One of the problems that proposals for models/languages for real-time systems try to solve is the following: one process P needs another process Q to do some computations for it (P) urgently (perhaps in a few "clock cycles"). To deal with this type of problem some models consider complex scheduling algorithms (in Q) to schedule the requests. How would we handle this problem in RSP?

Ans: We would not. And emergencies are not designed for this purpose. In general, I believe that a process should perform its own computational tasks, relying upon others only for "physical" (not computational) services. If indeed processes only request physical services then requiring such a service in a matter of a few clock cycles seems unreasonable.

There might be one exception to this. It seems possible that in order to deal with an emergency situation a process P might need some information from another process Q and, of course, P would need this information quickly since it is dealing with an emergency. I could deal with this situation by adding to an RSP process (Q, in particular) a function, F_Q , of the state of Q (at its last I/O command) and the messages in $\mu_{Q,2}$'s buffer; if any process sent to Q an urgent request for special information then $\mu_{Q,2}$ would simply send back the value of F_Q . It would be easy to include such a mechanism in RSP, but I didn't do so since, despite the intuitive appeal of this mechanism, I have been unable to come up with an example where it is actually needed. One other potential application, apart from dealing with emergencies, for such a mechanism would be for "monitoring" purposes. Thus one process could use this mechanism to track the progress of another process, and to watch for potential problems. (A possible notation for such a construct: $Q??x$, meaning that P is making an urgent request of Q.)

Q8: Where is the "clock" in my model?

Ans: I assume that each process has access to a clock and that these clocks are approximately synchronized. Since, as I said earlier, processes will typically request physical rather than computational services of other processes, the time constraints specified in those requests will typically be several milliseconds (or more), rather than a few clock cycles. Thus the various clocks don't have to be precisely synchronized. (Note: our Prime Number example is a rather atypical RSP program since the process $P[0]$ does rely upon other processes of the program for computational services.)

Q9: CSP has input/output guards, but does not allow a not-done path following a guard. Why does RSP allow a not-done path in addition to the done path?

Ans: It is clear that it is useful to have a not-done path: what a process P does when it tries to send a request x to S will, in general, depend on whether or not S accepts the request, i.e. on whether or not $S \uparrow x$ can be done. The commands on the done path specify how P proceeds if x is accepted and the commands on the not-done path specify how it proceeds if x is rejected.

The reason that such a construct (the not-done path) does not belong in CSP is that if an output guard $S! \dots$ cannot be chosen (in a CSP process P) that only means that at that moment S is not trying to execute a matching

input command $P?$. . .; from that alone one cannot conclude that $S!$. . . has "failed" since in the very near future S might well try to execute $P?$ On the other hand, in RSP, if P sends a request to S , S will immediately respond and either accept or reject P 's request, and P knows S 's decision; so in RSP it is possible to have a not-done path (perhaps a more precise name would be "cannot-be-done path") whereas such an idea would be unsuitable for CSP.

Q10: When S receives a message that satisfies E_S then control in S is "backed-up" to its most recent I/O command. What if we have not yet executed an I/O command?

Ans: This is the problem of initialization. The right solution seems to be for S not to accept any messages until it reaches its first I/O command; i.e. before S reaches its first I/O command A_S and E_S will evaluate to false no matter what messages are received. This is reasonable since A_S and E_S will in general refer to the state of S and, until S has completed initialization, the state does not have a reasonable value.

Q11. Allowing output guards in CSP causes considerable problems in the implementation. What happens in RSP?

Ans. Consider a simple example: suppose a process P wishes to send the message x to the process Q , or the message y to process R . Using the notation of chapter II, I would write

$$[Q \uparrow x \rightarrow S \ \square \ R \uparrow y \rightarrow S']$$

To execute this P could send x to Q; if Q accepts it, the communication is complete and P will then execute S. If Q rejects x, P will send y to R; if R accepts y this communication is complete and P will execute S'. (If R also rejects y, P will abort.) P might try R \uparrow y first, and if R rejects y then try Q \uparrow x. But it does not poll Q and R to see whether they would be "willing" to accept x and y respectively and then choose one of the willing guards. Thus I do not have the implementation problems that output guards in CSP have.

Two further points should be made here: (i) If a process Q rapidly changes its criteria for acceptance/rejection of messages (i.e. its "acceptance condition" changes frequently), P might experience "race conditions" in its attempts at sending messages to Q since the same message that is rejected at one moment would have been accepted if tried a short while later. This seems to be a fault of the particular Q rather than the language. (ii) Our input guards/commands are, in a sense, "local" commands since they just result in the messages that have already been accepted to be "read in" by the process.

Q12. In CSP an output guard in P fails only if the receiving process Q has terminated, not if it is currently unwilling to input from P. Why do I define Q \uparrow x to fail even though Q might not have terminated?

Ans. Note first that the success/failure of $Q \uparrow x$ has nothing to do with whether or not Q is currently executing an input command. Instead $Q \uparrow x$ succeeds if Q accepts x , i.e. x satisfies Q 's acceptance condition, and fails if x does not satisfy Q 's acceptance condition. Recall also the purpose of the responsive communication mechanism: if Q does not accept x (since it doesn't expect to be able to process the request within the specified constraint), P would very likely want to try other alternatives such as sending x to Q' . I can express this easily by saying

$$\begin{aligned} & [Q \uparrow x \rightarrow \dots \\ & \quad \rightarrow [Q' \uparrow x \dots \\ &] \end{aligned}$$

which sends (or tries to send) x to Q' if Q rejects it. Of course, if P simply wants to keep trying Q until it accepts x , we can write

$$\begin{aligned} & b := \underline{\text{true}}; \\ & * [b; Q \uparrow x \rightarrow b := \underline{\text{false}} \\ & \quad \rightarrow \underline{\text{Skip}} \\ &]. \end{aligned}$$

Q13. CSP's selection/repetition commands are easy to understand. In understanding a particular path, we only have to worry about the guard corresponding to that path and the command following the guard; we do not have to worry about the other guards that might have been tried but

didn't succeed. Wouldn't this simplicity be lost if I define $Q \uparrow x$ to fail if Q rejects x?

Ans. No. Consider our earlier example: $[Q \uparrow x \rightarrow S \ \square \ R \uparrow y \rightarrow S']$. There will be no difference between i) the case when P sends x to Q, Q rejects it, P tries y on R which accepts it, and P goes on to execute S'; and ii) the case where P sends y to R which accepts it, and P goes on to execute S'. In other words, in understanding the path $R \uparrow y \rightarrow S'$, we do not have to worry about the fact that I might have tried $Q \uparrow x$ and Q rejected x. This would, of course, not be true if Q retained some memory of the fact that it rejected x. In fact, Q has access only to the messages it accepts, not to the messages it rejects (not even to the extent of knowing that it rejected anything!).

Q14. When a process Q receives an emergency message its state is supposed to be "backed up" to what it was immediately following its most recent I/O. Isn't this very difficult for the programmer to do?

Ans. The "system" (i.e. the implementation) does it. The programmer can completely ignore the backing up. To him the process seems "prescient": if there is going to be an emergency message before the process can read its next I/O command, it waits until the emergency message arrives. Thus the fact that the process started on the local computations but backed up when the emergency message arrived is completely transparent to the programmer.

Q15. Is a physical device that is part of a large system an RSP process in some sense? If so, how can it automatically back up the mechanical action it might have performed since the last I/O?

Ans. I do treat physical devices as (hardware) RSP proceses. But their emergency condition is identically 'false', so they never receive an emergency message, and never have to back up. Typically, if the device does need to be backed up, the (software) device driver associated with the device will send an explicit communication to the device instructing it to perform the opposing (backing up) action.

Q16. If I send a process a request that needs to be completed within, say, 15 nanoseconds, how can I be sure that the other process will respond with an accept or reject, let alone perform the request, within the 15 nanoseconds?

Ans. First of all, a RSP process is not designed to handle such requests. Requests are typically intended to be used for making requests for physical services, and time constraints for such services are normally in the range of tenths of seconds or greater. A request with a 15 ns time constraint is most likely a request for some computation. As I have stated repeatedly, due to the low cost of computing power, processes should perform their own computations.

Another view of this question might be: what if a process is besiged all at once by a number of requests. Won't the time for responding to requests become large? This could be a problem, especially if the emergency

and acceptance conditions are complicated. So, when designing the system the designer must take into account the complexity of the Boolean expressions and the maximum number of processes that could simultaneously output to each process, and see if the turnaround time would be bigger than the typical time constraint in the output messages.

CHAPTER VI

CONCLUSION

In the previous chapters a motivation for the language Responsive Sequential Processes was presented, the features of the language were fully characterized and illustrated with examples, a system model was presented, and a discussion was given regarding various aspects of the language. In this chapter I will compare RSP with some other languages which were designed with the same target application, and lastly I will summarize this research.

Modula vs RSP

Modula was designed especially for applications such as embedded industrial and scientific real-time computer systems. In [28] Wirth describes a discipline for writing real-time programs in Modula. Such programs, he explains, are to be written first to ensure that it is logically correct and then another pass is made to check that any constraints are met. Whether or not the constraints will be met, of course, will depend upon the machine on

which the program is being run. Processors are shared by the processes. This makes a static time analysis difficult at best. This difficulty is compounded by the use of interrupts. To account for interrupts and processor sharing Wirth introduces some formulas to approximate the execution time of the statements.

There are several problems with this approach. The first is that upon reading such a program a person is unaware of whether or not the program has to satisfy any timing constraints. Maintenance of such a program by someone other than the original programmer will be very difficult since it is unclear what constraints the program is attempting to meet.

Next, time analysis of Modula real-time programs are very difficult. Sharing processors makes it very hard to tell whether a piece of code will meet a time constraint. The code may meet its time constraint if it has a dedicated processor during the execution of the code. However, with processor sharing, it cannot be guaranteed that the processor will be dedicated to that process during that time span, since an emergency may arise in another process, which will result in process switching.

Interrupts makes a program both difficult to write and to understand. At any point in the program the state may change unexpectedly, because an interrupt has occurred. This forces the writer of the process to write the code under the assumption that the state which

exists when a statement is reached may be either the state which existed after the execution of the last statement *or* the state which exists after the last statement was executed and then an interrupt service routine was executed (possibly several interrupt routines may be executed if there are several interrupts). Clearly, this will make writing and understanding such processes very difficult. Actually the situation is even worse since an interrupt can occur in the middle of a statement.

None of these difficulties are present in RSP: Any constraints that are present in the program are explicitly programmed for in the program. In addition, the emergency condition and the acceptance condition clearly define what constraints a process is capable of meeting. Next, each process has its own dedicated processor, so the complexities of processor sharing are not present. Lastly, interrupts are not allowed in the RSP language - each process defines what messages that it receives constitutes an emergency for itself; it will react immediately to those messages it receives that are emergencies.

Distributed Programming System vs RSP

The Distributed Programming System (DPS) was designed for distributed real-time systems. The language allows the specification of timing constraints, which a system scheduler uses for scheduling. A "temporal scope" construct is provided for specifying the time constraints

of code execution and of interprocess communication. In addition, the language allows for the detection and handling of exceptions caused by timing constraint violations.

The temporal scope construct identifies a sequence of statements with timing constraints. The construct specifies when the statements are to be executed, in either absolute or relative time, and it specifies how soon the statements have to be completed. A communication temporal scope is used to specify the timing constraints of interprocess communication. It specifies when a message should be received and processed by a receiving process, how long a sending process is willing to wait for a reply, how long a receiving process is willing to wait for a message to arrive, and how long it takes a receiving process to process a message. The run-time support system detects any timing constraint violations and transfers control to the end of the temporal scope, where exception handlers are located.

Introducing constructs for specifying the time constraints facilitated a run-time support system in scheduling the processes on the processors. This alleviated the programmer from having to implement his own scheduling system. This is argued by the authors of the language as a strong advantage of DPS over such languages as Modula. I grant this. However, with the low-cost of processing units each process should be able to have its own processor, as is proposed in RSP. This will eliminate the need for any kind of external scheduler or run-time support system. Each process, then,

has greater flexibility in scheduling its own activities. In addition, if each process has its own dedicated processing unit then a process will be less complex and more autonomous since it no longer needs to program for the situation where there is no processor available when one is needed.

A second criticism of DPS is in its exception handling. With DPS an exception is raised by the run-time system if the job is not completed within the programmer specified time constraint. A group of non-communicating statements within a process may not be completed within the constraints specified in the temporal scope, thus creating an exception. This is inherent in any real-time language implementation where processors are shared. It obviously will not exist in RSP where each process has a dedicated processor. Another source of exception raising in DPS is when an interprocess communication fails. If the receiving process is overloaded with other requests, or if all of the processors are being utilized then the communication will fail. With the later, more processing power would solve the problem. With the former, if the sender was notified, when it sent its request, as to whether or not the receiver could process the request then it would see that the process was overloaded and would not be able to process its request and would then have time to try other alternatives.

A final objection against DPS is that it is unable to recognize emergencies, other than emergencies which occur when time constraints are violated. Clearly, in a distributed real-time system, it would be necessary

to have a mechanism for recognizing and reacting to messages that are received that will have grave consequences if not handled immediately. This mechanism is an integral part of RSP. Emergency messages are defined by the emergency condition. If a message arrives which satisfies the emergency condition then it is detected and reacted to immediately.

CSP vs RSP

A CSP program consists of a set of independently executing processes $[P_1 \text{ // } \dots \text{ // } P_n]$. Processes interact by means of explicit, synchronized communication: $P_i :: [P_j!x]$ means that process P_i outputs (shrieks) the value of x to process P_j . The communication does not actually take place until P_j executes the matching input statement. When P_j executes the matching input statement: $P_j :: [P_i?y]$ then the value is transferred and is assigned to y . The I/O statements may also be used as an I/O guard in selection statements and in loops. When used in a selection statement: $P_i :: [[P_j!x \rightarrow S_1 \sqcap P_k?y \rightarrow S_2]]$ the system randomly selects one of the guards that succeeds, performs the I/O, and then executes the statement following the arrow. If both guards fail then the process aborts. An I/O guard fails when the matching process has terminated. I/O guards may also

be employed in loops: $[*[P_j!x \rightarrow S_1 \sqcap P_k?y \rightarrow S_2]]$. The loop is repeated until both guards fail.

In [24] an extension has been proposed to CSP to make it more suitable for real-time applications. A time-out mechanism was added to the I/O constructs. Previously, a process could wait at an I/O construct for an indefinite period of time as it waited for the other process to execute a matching I/O command. Obviously, this is not suitable for time constrained processes. With the time-out mechanism control remains at the I/O construct until the I/O succeeds or until the time-out occurs, whichever comes first. This mechanism is denoted by Wait Now After alarm.time.

The problem with the time-out mechanism is twofold: suppose that a process P_i sets the time-out for two seconds, which is the time by which some job must be completed. Suppose that the receiving process does not complete the desired task within the 2 seconds. When the two seconds expire then it is typically too late for P_i to do anything. The job had to be done within two seconds and now that time has expired. Suppose we set the time-out for a lesser time and it expires. If we had only set the time-out for a longer time it may have finished. What is needed is a mechanism by which the sending process can immediately know whether or not its

requests' constraints can be met. If it has an immediate response then it can attempt to take some alternative action, if needed. Otherwise, if the receiver indicated that it can process the request, it may rest assured that the request will be processed within the constraints specified. The ability of a process to receive immediate response to a request lies at the very heart of RSP. As soon as a process makes a request of another process the requestee will respond with an accept or reject, indicating that it can or cannot process the request. If the requestee cannot process the request then the requestor will (presumably) try some other course of action. If the requestee accepts the request then the requestor may rest assured that its request will be carried out.

Motivating Concepts for RSP

In this section I summarize the software engineering principles which underly and motivate RSP, and propose a set of principles on what constructs are useful and what constructs are harmful in distributed programs.

1. Autonomous Processes

The first principle is that *distributed systems should be composed of independently executing processes, there should be no shared variables, and processes should interact by means of explicit communications.* The

justification for this principle is as follows: An autonomous process which cleanly interfaces with other processes via explicit communications will be easy to write and easy to understand. Shared variables destroy the autonomy of processes. Consider two processes P_1 and P_2 which share memory M . In order to understand P_1 one needs to understand how it changes M . But, P_2 also changes M . Thus you also need to understand P_2 and how it changes M . So instead of an autonomous view of each process we must have a global view of the entire program. This principle of autonomous processes, interacting by means of explicit communications, was, of course, first advocated by Hoare [17], and was the motivation behind the language CSP.

2. The Interrupt Mechanism

The use of an interrupt mechanism should be avoided in writing distributed programs. There are two primary reasons for this. Let us examine a distributed program which utilizes an interrupt mechanism. Consider a process P which is interrupted. How is P to be written? Suppose that statement S_i in P has just been executed when the interrupt occurs. Control is transferred to some interrupt service routine, which is executed (resulting, in general, in the state of P being changed), and finally control

returns to the next statement, S_{i+1} . From S_{i+1} 's viewpoint the state has somehow "magically" changed from the state which existed just after execution of S_i . So, for any two statements ... S_i ; S_{i+1} ... in P , S_{i+1} must be written under the assumption that the state which will exist when control reaches it, will be either the state which exists just after S_i is executed *or* the state which exists after S_i is executed and then the interrupt service routine is executed. Obviously reading and writing code with such arbitrary state changes will be very difficult. The situation is actually worse since multiple interrupts (i.e., more than one interrupt between S_i and S_{i+1}) would further exacerbate the situation!

A second argument against interrupts lies at the very heart of the philosophy of using interrupts. The underlying premise of interrupts is that an interrupting process Q knows how to best utilize the interrupted process' (P) time and resources. Thus, at any moment Q can stop P from executing its current task and get P to do some job for it. This is an unsound premise. Autonomous processes know best how to handle *their own* time and resources and not the time or resources of any other process.

Based upon these two arguments, my second principle states that *the use of an interrupt mechanism in distributed programs should be avoided*. However, consider the following problem: Some messages that a

process P might receive constitute an "emergency" for P , and require immediate corrective action. If we had an interrupt mechanism, we could deal with this by letting the message interrupt P . How do we deal with this problem if we do not have interrupts? The answer is that we need a mechanism that a process say, P , can use to identify the messages that constitute an emergency for it, and if such a message is received, P must be able to react quickly and take corrective action, abandoning whatever it is currently doing. Note that the sending process does not "interrupt" P ; it cannot; it is P that decides what messages constitute an emergency for it and how to react in the face of such an emergency. To the sender this message is like any other message it might send to P .

This brings me to my third principle: *a process should have the capability of defining what messages constitute an emergency for the process and be able to react immediately to an occurrence of such a message.*

3. Responsive Systems

In distributed programming there are numerous applications (i.e., real-time programs) where a process P_i makes a constraint-bound request of process P_j . A constraint-bound request is a request which has certain constraints, such as time constraints, associated with it (eg. perform some

service within x units of time). If P_j cannot process the request within the constraints then P_i would like to take some alternative action. Further, if P_i is to take the alternative action then it must have sufficiently advance knowledge that P_j will be unable to meet its request. It will be too late for P_i to take alternative action if it simply waits (for example) until the time specified in the constraints have expired. P_i needs to receive a quick *response* to its request. If the request is accepted then it (P_i) may rest confident that the request will be processed within the constraints specified in the request. If the request is rejected then P_i will have ample time to try other alternatives.

This brings me to my next principle: *processes should be responsive, i.e. a process should respond quickly to requests, indicating whether or not it will be able to carry-out the request within the specified constraints.* This does not mean that the requestor will interrupt the requestee, only that the requestee should quickly respond and either reject the request or accept the request for *later* processing. Note also that this is not the same as a CSP-like synchronized communication mechanism, as such a mechanism would indeed provide a response but not quickly - as is required.

4. A Processor for each Process

The notion of having a limited number of processing elements (P.E.s) to be shared among processes was a natural response to the high cost of hardware in the early seventies. Large, complex operating systems to manage and schedule the processes on the P.E.s soon ensued. Today, with the low cost of hardware and the skyrocketing costs of software such systems need to be rethought. Indeed, sharing processors does not create a system which promotes software productivity. My next principle of distributed software engineering is thus: *each process should have its own P.E.* There are several reasons for this. Obviously, it immediately rids us of the cumbersome operating system. Each process is now free to manage and schedule its own activities. Secondly, a system which is designed with one P.E. per process will be less complex. Consider a system which shares P.E.s. Suppose that a process P receives an emergency message and it presently does not have a P.E. allocated to it. Clearly a good operating system will attempt to quickly provide P with a P.E.. But suppose that all of the P.E.s are being used. Let's suppose that the operating system preempts process R, which has a P.E., and allocates the P.E. to P. This may create an emergency for R, which results in it preempting another process, etc. Thus, one emergency may have a "domino" effect by inducing emergencies in processes which are totally unrelated to the original process. The possibility of such a cascading of emergencies has to be accounted for in shared

processor systems and will naturally result in a much more complex system than in a system where such an occurrence is not possible.

5. Concise, Meaningful Communication Primitives

My final principle is that *communication primitives should be concise and meaningful*. Concise notations are easier to deal with, remember and, of course, type than long, wordy notations. A symbol is most easily remembered if the symbol has some association to something else we are already familiar with. Further, a language that has constructs which are easily remembered is more likely to be used than a language with equally powerful constructs but with wordy constructs which are difficult to remember. One reason for CSP's remarkable success seems to be the conciseness and simplicity of its notation. In RSP I have similarly tried to use a concise and simple notation for primitives that can be employed to build complex distributed real-time systems.

In this section I have identified a number of principles that will be useful in the construction of distributed programs. Also, I have identified constructs which will be dangerous if used in a distributed program.

Summary and Further Research

Over the last few years, a considerable amount of work has been done on what are called "real-time systems". Despite all the work, progress seems to have been limited. Most proposals seem to try to answer the question, "How can we express 'time constraints' in real-time systems, and how can we ensure that these constraints are met?". The situation is reminiscent of the status of "concurrent programming" in the early seventies. The question then was , "How do we express 'concurrent activities' in a program, and how do we ensure that these activities do not interfere with each other in unacceptable ways?". The fundamental contribution of CSP was to shift the focus from '*concurrent* activities' to '*distributed* processes'. The work presented in this dissertation is based upon my belief that a similar shift in focus is needed - from '*real-time* systems' to '*responsive* processes'.

The classic problem in real-time systems is to design a process S that will process a request from another process P within the time limit imposed by P (and specified as part of the request). In general, however, there is no way to ensure that S will indeed meet P's time constraint; S might, for instance, have already received a number of requests from other processes, thus preventing it from processing P's request within the desired time. Clearly then, P must be willing to take appropriate action if S is unable to meet its request. But in order to be able to do that P needs to know

whether or not S will process its request - and P needs this knowledge now, for only then will it have enough time to try other alternatives. In other words, S must be *responsive* - as soon as P makes a request, S must respond and either reject the request or accept it for processing at a later time (but within the time specified by P), and inform P of the acceptance or rejection. If S accepts the request, P can continue with its other activities confident that S will process the request in a timely manner. If S rejects the request, P will have plenty of time to try other alternatives such as making a similar request of another process S'.

Let us return to CSP for a moment. CSP did more than shift the focus to distributed processes. It also provided a powerful set of tools - in the form of I/O commands and guards - to build them. Responsive Sequential Processes (RSP) similarly tries to provide a set of tools for building responsive processes.

In summary, RSP is a language which is well-suited for systems that contain not only "computational" activities but also a number of "physical devices". The responsiveness of a process provides other requesting processes with a greater certainty over their request's destiny. The emergency condition and its associated semantics yields tremendous flexibility for a process in defining what constitutes an emergency for it. This is in contrast to the current emergency mechanisms, where emergencies are rather "cast in stone" and are not at all flexible. In example 1 (pg 43) a non real-time example was given to demonstrate the emergency

mechanism. This demonstrates RSP's emergency mechanism is useful not only in real-time systems but also in non real-time systems. The examples demonstrate that this language can elegantly handle a wide range of problems.

There are two problems that need further exploration: (1) How are heirarchical systems to be handled? RSP, proposed in this thesis, has only "flat" processes. Can we treat a group of RSP processes as a high-level process? What will its acceptance and emergency conditions be? How will it interface with other processes in the system? (2) How do we formally define the semantics for RSP?

The solution to these problems should prove to be very interesting and will provide further insights into the language and its ramifications in this area of programming.

LIST OF REFERENCES

1. Allchin, J.E. *Modula and a Question of Time*. IEEE Tran. on Soft. Eng. SE-6, 4 (July 1980), 390-391.
2. Berry, D.M., Ghezzi, C., Mandroili, D., and Tisato, F. *Language Constructs for Real-Time Distributed Systems*. Computer Languages 7, 1 (1982), 11-22.
3. Berry, G., Moisan, S., and Rigault, J.-P. *ESTERAL: Towards a Synchronous and Semantically Sound High Level Language for Real-Time Applications*. Proc. Real-Time Systems Symposium, IEEE, 1983, pp. 3-19.
4. Caspi, P. and Halwachs, N. *An Approach to Real-Time Systems Modelling*, Proc. of the Int. Conf on Distributed Systems, Miami Beach, Florida, 1982, IEEE Cat. No. 82 CH 1802-8.
5. Dasarathy, B. *Timing Constraints for Real-Time Systems: Constructs for Expressing Them, Methods for Validating Them*. IEEE Transactions on Software Engineering. January 1985. pp. 80-86.
6. Ghezzi, C., Jazayerri, M. *Programming Language Concepts*. pp. 137-8.

7. Glass, R., *Real-Time: The "Lost World" of Software Debugging and Testing*, Comm ACM, Vol. 23, No. 5, May 1980.
8. Gligor, V.D. and Luchenbaugh, G.L. *An Assessment of the Real-Time Requirements for Programming Environments and Languages*. Proc. Real-Time Systems Symposium, IEEE, 1983, pp. 3-19.
9. Gouda, M., *Analysis of Real-Time Control Systems by the Model of Packet Nets*, Proc. of the Nat. Comp. Conf., AFIPS Press, Vol. 48, Montrale, N. J. 1979.
10. Hansen, P. Brinch (1975), *A Real Time Scheduler*, Information Science Report, California Institute of Technology.
11. Hansen, P. Brinch (1978), *Distributed Processes: A concurrent programming concept*, Comm ACM, 21, 11, pp. 934-941.
12. Hansen, Per Brinch, *The Programming Language Concurrent Pascal*, IEEE TSE, Vol. SE-1, No. 2, June 1975, pp. 199-207.
13. Hansen, Per Brinch, *EDISON: A Multiprocessor Language*, USC Dept. of Comp. Sci., September 1980.
14. Hansen, Per Brinch, *The Design of EDISON*, USC Dept. of Comp. Sci., Sept. 1980.
15. Hasse, V., *Real-Time Behavior of Programs*, IEEE - Trans on Software Engineering, Vol. SE-7, No. 5, Sept. 1981.
16. Hoare, C.A.R. *Communicating Sequential Processes*. CACM 21, 8. August, 1978, pp. 666-677.

17. Holden, J., and Wand, I.C. (1980), *An assessment of Modula*, Software P and E, 10, 8, pp. 593-622.
18. Lee, I., and Gehlot, V. *Language Constructs for Distributed Real-Time Programming*. Technical Report. University of Pennsylvania. April 30, 1985.
19. Martin, T. *Real-Time Programming Language PEARL - Concept and Characteristics*. Proc. COMPSAC, Chacago, 1978, pp. 301-306.
20. Martin, T., *Real-Time Programming Language PEARL - Concept and Characteristics*, Proc. COMPSAC, Chicago, 1978, IEEE Cat. No. CH 1338-3/78/0000-0301.
21. Martin, T., *PEARL at the Age of Three*, Proc. of 4th Int. Conf. on Soft. Engin., Munich 1979, IEEE Cat. No. CH 1479-5/79/0000-0100.
22. Mok, A.K. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*. Ph.D. Th., MIT, May 1983. MIT/LCS/TR-297.
23. Mok, A.K. *The Design of Real-Time Programming Systems Based on Process Models*. Proc. Real-Time Systems Symposium, IEEE, Dec., 1984, pp. 5-17.
24. OCCAM Programming Manual. Prentice/Hall International.
25. Rao, Ram, *Design and Evaluation of Distributed Communication Primitives*. University of Washington, Dept. of Computer Science, TR 80-04-01, April 1980.
26. Ward, S.A., *An Approach to Real-Time Computation*, Proc. of Seventh Texas Conf. on Computing Systems, Oct. 1978.

27. Wirth, N. (1977), *Modula: a language for modula multiprogramming*, Software P and E, 7, pp. 3.
28. Wirth, N. *Toward a Discipline of Real-Time Programming*. Comm. of the ACM 20, 8 (Aug. 1977), 577-583.
29. Young, S.J. *Real Time Languages Design and Development*, Pub. Ellis Horwood Limited, ISBN 0-85312-460-4.