An Implementation of Communicating Reactive Processes

Basant Rajan and R.K. Shyamasundar
Computer Science Group
Tata Institute of Fundamental Research
Homi Bhabha Road, Bombay 400 005, India
e-mail: {basant,shyam}@tcs.tifr.res.in

April 1, 1997

Abstract

It has recently been shown that both *asynchrony* and *perfect synchrony* are needed for the specification of complex reactive systems and a paradigm called *Communicating Reactive Processes* (*CRP*) was proposed towards such a unification. The paradigm consists of a network of *perfectly synchronous* nodes communicating *asynchronously.* This paper addresses the implementation issues of such paradigms and discusses possible implementations that are consistent with the reactive and rendezvous preserving semantics. We present a *deadlock-free* protocol for the implementation of *CRP.* The implementation is shown to preserve the *reactive* nature of the synchronous nodes and also establishes techniques that will enable us to realize an implementation that preserves the rendezvous semantics if so desired. The paper discusses the properties satisfied by the implementation and shows that the implementation also provides another implementation for Hoare's CSP.

1 Introduction

In [4], a paradigm referred to as *Communicating Reactive Processes*(*CRP*), that unifies the capabilities of asynchronous and synchronous concurrent programming languages, has been proposed. The *CRP* paradigm consists of a network of ESTEREL nodes (representing perfectly synchronous nodes) communicating with one another through the *rendezvous* mechanism. One of the merits of the paradigm is that verification can be structured on the lines of the proof for ESTEREL programs.

In this paper, we discuss the issues and techniques related to the implementation of such paradigms. The implementation poses challenges as it is necessary to preserve the *reactive* nature of ESTEREL and/or the semantics of a CSP rendezvous. We shall now outline the intuitive semantics of a rendezvous in the context of CSP as well as *CRP.* It is assumed that communication is asynchronous, messages are never lost or corrupted and that message order is preserved.

For our purposes, a CSP style rendezvous succeeds *only* when *both* the participating processes are ready and there are no failures. The *rendezvous* semantics is preserved if and only if a communication primitive terminating implies *both* participating processes can assume that the other participated successfully too.

CRP style rendezvous differ from the CSP rendezvous in that the communicating process can be locally *preempted* at any stage. In particular, it is possible that one of the participating processes gets preempted while the other successfully completes the rendezvous. The *reactive* semantics is preserved if and only if the processes participating in a rendezvous can be *preempted* at *any* stage/phase of the rendezvous.

Although it would be nice to have an implementation that would preserve the semantics of the rendezvous in spite of the possibility of preemption, this *cannot* be achieved in practice. We present a protocol that provides the reactive semantics while preserving the rendezvous semantics as far as possible. We illustrate with a "banker-teller" example how we can indeed preserve both the reactive and rendezvous semantics under reasonable assumptions. In a general framework, the protocol enables us to adhere to the rendezvous semantics when it is essential.

Such a measure has the advantage of avoiding freezing which would deviate from the reactive semantics or resorting to some kind of atomic transaction for achieving the rendezvous semantics unless absolutely necessary.

The main contributions of the paper are:

- We present a protocol for *CRP* capable of preserving the reactivity semantics while providing the user the choice of preserving the CSP rendezvous semantics instead (at the cost of reactivity).
- The proposed protocol is shown to be *deadlock free* and the effects of non-determinism are also discussed.

The rest of the paper is organized as follows: in section 2, we provide an abstract description of the problem and discuss the requirements an implementation must meet. Section 3 explores a few options in implementing *CRP* and goes on to develop a reactive (freeze-free) implementation. In section 4, we describe the structure of our basic solution. Section 5 discusses problems that arise due to the need to handle *nondeterminism* and the basic principles we adopt for arriving at a deadlock-free protocol . In section 6, we analyze the properties of the solution followed by a concluding section.

2 Problem Specification

We first present an abstract model that captures the subset of properties one is interested in while dealing with a network of synchronous processes. We therefore abstract the *CRP* model as a network of processes with the following features:

- All commands except communication and timed-commands are instantaneous (such as those that can be specified in *reactive* languages like ESTEREL, LUSTRE *etc.*).
- All statements can be *preempted.*
- No process is necessarily sequential. That is, parallel[1] local actions or communications are possible.
- The communication between any two processes is through a *channel* between them. Further, a channel connects at most two processes.
- The communication medium is asynchronous. It is assumed that messages are never lost and message ordering is preserved.

2.1 Issues in Implementing a Rendezvous

Though several varieties of asynchronous communication primitives are possible, we choose the CSP *rendezvous* as the underlying communication mechanism as it is the most difficult from the point of view of implementation. The rendezvous is achieved through two primitive commands (similar to the CSP `send` and `receive` commands) given below with the usual interpretation.

- `send`(<*channelName*>,<*value*>);
  block until sending *value* on channel *channelName* succeeds.
- `recv`(<*channelName*>,<*variableName*>);
  block until a communication succeeds on channel *channelName*, at which point, assign the value received to the variable *variableName*.

Informally rendezvous enforces the following constraints:

- both *sender* and *receiver block* until completion of the rendezvous.

[1]Later we show that there is no need for an explicit *choice* (or non-deterministic) operator

- if the *sender* is *preempted* (after initiating and before or at completion) the *receiver* should *not* consider the rendezvous successful.
- the *receiver* may be *preempted* after acknowledging the receipt of data from the *sender* *without* affecting the *sender*. In this case the *preemption* of the *receiver* is treated as a *local* event occurring *after* the rendezvous.

The problem now is to design a protocol to implement the above requirements that satisfies the following properties.

1. The system of communicating processes must have a *consistent* view of the system in spite of local preemptions.
2. The implementation of the protocol should necessarily be distributed.
3. The time taken by a process to determine whether it can communicate with some other process should be bounded. Expressed differently, the system should not *deadlock*.
4. The number of messages exchanged for making a communication should also be bounded.
5. The communication protocol should not require/introduce a *freeze* state during the entire set of transactions involved in implementing the communication primitive. (This would ensure that the *reactive* nature of the interacting processes is preserved.)
6. When two processes have matching communication commands, then at least one of them should establish communication with a process (not necessarily the other one) in a finite time. This criterion is stronger than the notion of progress which only implies that the system of communicating processes will eventually evolve.

3 Towards an Implementation of CRP

First let us explore the implementation possibilities. The options available are :

1. Preserve the rendezvous semantics; that is, a rendezvous succeeds only when *both* sender and receiver successfully complete the rendezvous.
2. Preserve the reactive semantics; the sender/receiver must support preemption at *any* stage of the rendezvous.
3. Preserve both.

From a characterization of the first two possibilities in terms of the changes they bring about in the state of common knowledge, it becomes evident that only one of them can be implemented at a time. Obviously then, the third choice is ruled out.

In the following subsection, we discuss a reactive implementation that provides a framework that enables us to provide a user programmable choice between the two conflicting semantics as per requirement.

3.1 A Reactive Implementation

In this section, we discuss how we can arrive at an implementation that preserves the reactive nature of the underlying ESTEREL nodes

Consider the implementation of the `send` and `recv` primitives as shown below.

$$\texttt{send}_C, e; \quad \equiv \quad \begin{array}{l} \texttt{emit}\ \mathrm{Send}_C(e);\\ \texttt{do}\\ \quad \texttt{await}\ \mathrm{Rcpt}_C;\\ \quad \texttt{emit}\ \mathrm{Conf}_C;\\ \texttt{watching}\ \mathrm{Kill}_C;\\ \texttt{present}\ \mathrm{Kill}_C\ \texttt{emit}\ \mathrm{S\_Abort}_C; \end{array}$$

```
                      emit Recv_C;
                      trap done in
                        do
                        loop
                          trap redo in
                            await immediate Send_C(e);
                            emit Rcpt_C;
                            await
recv_C, e;     ≡                 case Conf_C : exit done;
                                 case S_Abort_C : exit redo;
                            end % await case
                          end % trap redo
                        end % loop
                        watching Kill_C;
                      end; % trap done
```

Note that in the implementation above, the intuitive semantics described earlier is preserved. The Kill_C signal used in the code segments above is merely a logical signal. It represents the activation of any `trap` or `watching` statement that would *preempt* the *rendezvous* in progress. In the sequel we show how such a logical signal can indeed be implemented. Further, it has been assumed that an external agent buffers the Send_C signal and delivers it to the esterel module *only after* the initiation of the corresponding receive. .

Can we implement the above protocol?
Though, we presented an "equivalent" Esterel implementation for the `send` and `recv` primitives, a little thought would make it plain that "in-place" rewriting of the `send` (or `recv`) primitive with the equivalent code would not suffice because of the semantics of nested `watching` statements. Thus, we employ the following mechanism to implement the `send` (or `recv`) primitive in Esterel ; for the sake of simplicity, we consider the mechanism for pure Esterel only.

For every (uniquely) labeled `send` (or `recv`) primitive on channel `C`, `L: send C` (or `L: recv C`), we do the following.

- replace *in-place* the `L: send C` with

```
emit Send_L_C; % initiate the send ...
await immediate Complete_Send_L_C; % rendezvous over!
```

- create a parallel module

```
                                     loop
loop                                   await
  await Send_L_C;                        case watch_1 : emit kill_Send_L_C;
  % place code for Send_C(e) here        case watch_2 : emit kill_Send_L_C;
  % with appropriate relabeling of  ||          ...
  % signals to incorporate labeling      case watch_n : emit kill_Send_L_C;
  emit Complete_Send_L_C;              end
end                                  end
```

where `watch_1` ... `watch_n` correspond to the watching statements enclosing the statement `L: send C`, in the original code segment.

Given the fact that an Esterel program has a static structure, a single pass over the given Esterel code would be sufficient to extract information pertaining to the `watching` and `trap` statements that enclose a particular communication primitive.

4 Structure of the Solution

The crux of the implementation lies in designing a protocol capable of ensuring *consistency of the rendezvous communication in spite of preemptions* in the system and at the same time preserving the *reactive* nature of the individual modules.

The process structure of the solution consists of the ESTEREL nodes interacting with the asynchronous medium through a communication *daemon* (process) local to each node. These *daemons* in turn communicate with each other (whenever necessary) using messages. The *daemons* exchange messages to determine a matching pair of processes waiting to communicate taking into account issues like *fairness* and ensuring *deadlock freedom*. In the sequel, the working of these *communication daemons*, which implement the protocol, is dealt with in greater detail and we also show that the assumptions are indeed feasible from a practical point of view.

Thus, the protocol can be thought of as involving four separate entities – two synchronous modules (the *sender* and the *receiver*), and two daemons (the *sender daemon* and the *receiver daemon*) which model the asynchronous medium. The *send* and *receive* parts of the rendezvous are not symmetric and are detailed separately below. The progress of a successful rendezvous is first outlined followed by the effect of preemption at different stages of the *rendezvous*. The rendezvous between two nodes consists of the following logical processes:

1. The *sender* – the *sender* initiates a rendezvous by emitting a DATA signal and blocking until it receives a RECEIPT signal. The *sender* then emits a CONFIRM signal and deems the rendezvous to be have been successfully completed.

2. The *sender daemon* – the *sender daemon* simply relays every signal the *sender* sends it to the *receiver daemon* and *vice-versa*. In addition, it sometimes suppresses signals from the *receiver daemon* when they are no longer relevant to the *sender*.

3. The *receiver daemon* – the *receiver daemon* idles until it receives an INIT signal from the *receiver and* a DATA signal from the *sender daemon*. At the receipt of the second of the the above two signals, the *receiver daemon* emits the DATA signal to the *receiver*. It also acts a relay between the *sender daemon* and the *receiver*.

4. The *receiver* – the *receiver* initiates a rendezvous by sending the INIT signal to the *receiver daemon* and blocking until it receives a DATA signal from the *receiver daemon*. Arrival of the DATA signal is acknowledged by instantaneously emitting a RECEIPT signal. It then waits for a CONFIRM signal which marks the successful completion of the rendezvous.

A diagrammatic representation of the flow of signals that takes place among the processes mentioned above for a successful rendezvous is shown below.

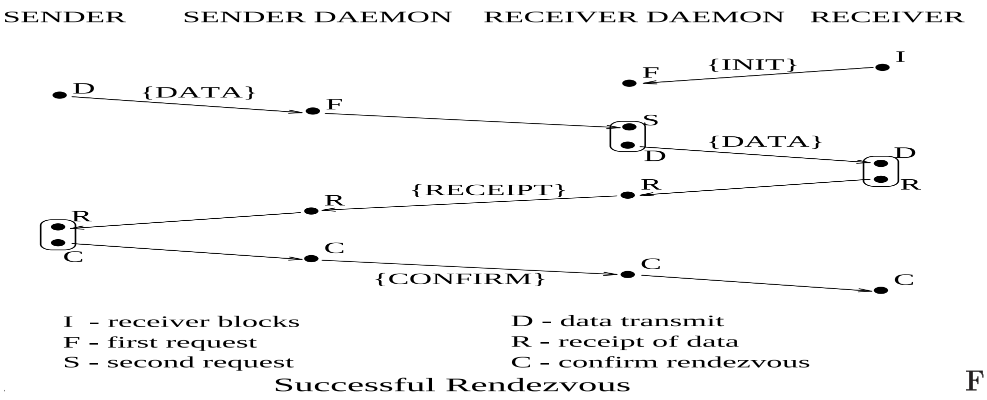


Figure 2: **Rendezvous Stages**

4.1 Consistency of the Rendezvous in spite of Preemptions

To show that the solution maintains the consistency for a rendezvous, let us consider the the effect of *preemption* on the above protocol. It is to be noted that *preemption* can occur at any time *after* the start of a rendezvous and *before* its completion. Let us consider the effect of preemption on the sender/receiver.

4.1.1 Preemption at the Sender end

Preemption on the part of the *sender* can only occur in the interval between emitting the DATA signal and receiving the RECEIPT signal (exclusive of the former and inclusive of the latter instant).

> *Preemption* in this context can be interpreted to mean that the data sent along with the DATA signal is being invalidated by the *sender*. *Preemption* is signaled by the *sender* emitting the KILL signal. This signal is relayed by the *sender daemon* and the *receiver daemon* to the *receiver* as the S-ABORT signal. At the *receiver* , the arrival of an S-ABORT signal is interpreted to mean abnormal termination of the current *rendezvous*. This causes the *receiver* to ignore the receipt of the DATA signal and the subsequent emitting of the RECEIPT signal.

Details related to the transitions involved are shown below.

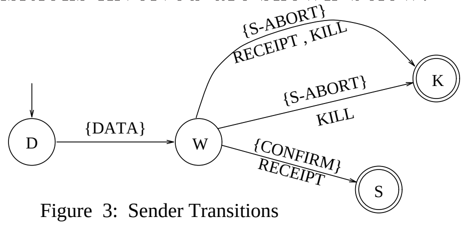


Figure 3: Sender Transitions

CRP Protocol (without freeze state)

4.1.2 Preemption at the Receiver end

Preemption on the part of the *receiver*, signalled by the receipt of the KILL signal, can occur in three disjoint intervals and their effects shall be discussed separately.

> *Preemption* occurring *strictly before* the *receiver* receives the DATA signal is interpreted as a cancellation of a previous INIT signal. The R-ABORT signal emitted by the *receiver* will be interpreted by the *receiver daemon* to mean the same.
>
> *Preemption* occurring *simultaneously* with the arrival of the DATA signal at the *receiver* will inhibit the emitting of the RECEIPT signal and again the *receiver daemon* will interpret the R-ABORT signal as canceling the previous INIT signal.
>
> *Preemption* occurring *strictly after* the arrival of the DATA signal is interpreted to mean that though the rendezvous is complete, the *receiver* chooses not to use the value received. The *receiver* would already have issued the RECEIPT signal by this time. The KILL signal associated with *preemption* occurring in this interval is *also* forwarded as an R-ABORT signal to the *receiver daemon* which from then on assumes that the *receiver* is no longer enabled. This interval is especially interesting because of the possibility of the arrival of an S-ABORT signal from the *receiver daemon* . It should however be clear that the data transferred will never be used unless *both sender* and *receiver* complete the rendezvous without either being preempted.

Details related to the transitions involved are shown below.

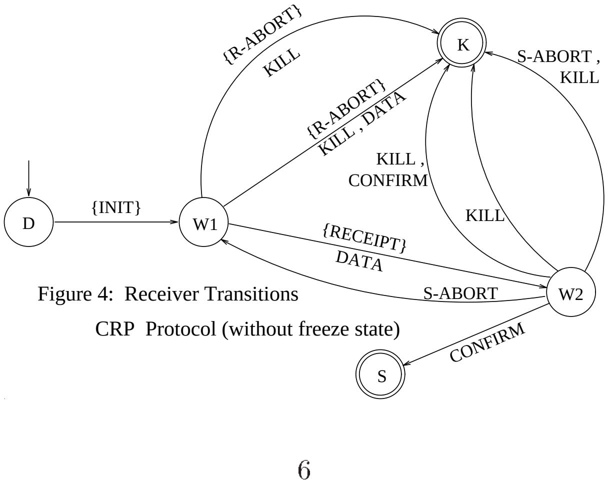
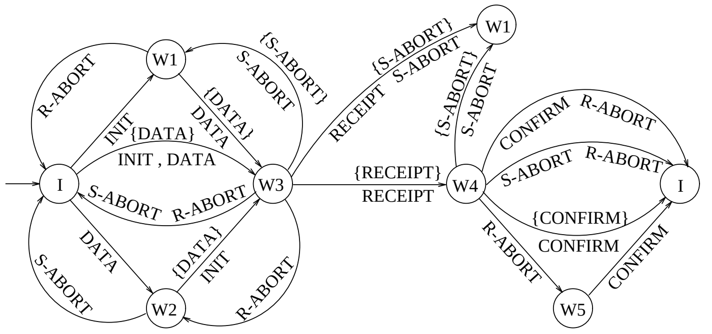


Figure 4: Receiver Transitions

CRP Protocol (without freeze state)

Preemption of either *sender* or *receiver* will cause the other to continue to block except in the case where the *receiver* is preempted after it emits the RECEIPT signal. In this case, the *sender* completes the rendezvous and the preemption is viewed as having occurred at the *receiver after* the rendezvous.

The details of the transitions taken by the asynchronous layer have been lumped together in the following state diagram.



Asynchronous Layer Transitions
Protocol (without freeze state)

Figure 5

4.2 Preserving CSP Semantics

If indeed it is found desirable to preserve the *CSP* semantics for a rendezvous in *CRP*, one is forced to consider the possibility of *freezing* one or both of the communicating nodes during at least part of the communication period. What is then achieved is a limited form of freedom from failure, where a process *commits* to complete a communication after a certain predetermined amount of progress is initially achieved. It turns out that only the *receiver* needs to be *frozen* and that too only from the time at which it emits RECEIPT to the time it receives either a CONFIRM or a ABORT from the sender (Figure 2). However, during this interval, the receiver is rendered incapable of responding to *any* input and so loses its *reactive* characteristics.

It is however possible, in some cases, to achieve the CSP semantics for a rendezvous without resorting to freezing by taking advantage of the asymmetry between senders and receivers as outlined in the bank-teller example.

4.3 Implementation Structure

We will now discuss the implementation structure of the solution. The complete protocol in a guarded command like pseudo-code is given in the appendix.

The asynchronous layer is modeled by a collection of networked *CRP daemons*, one on each node of the physical network. Communication is implemented using TCP and the *transport layer interface* routines. The ESTEREL specification of each *CRP* node is parsed to extract network related information and communication constructs re-written to call pre-defined C routines and later linked in with library routines that interface the nodes to the communication daemons on the local machine. To aid monitoring of the state of a rendezvous a *CRP debugger* with an X-window interface is also being ported to Linux. The *debugger* is capable of monitoring and supplanting either or both modules involved in a rendezvous.

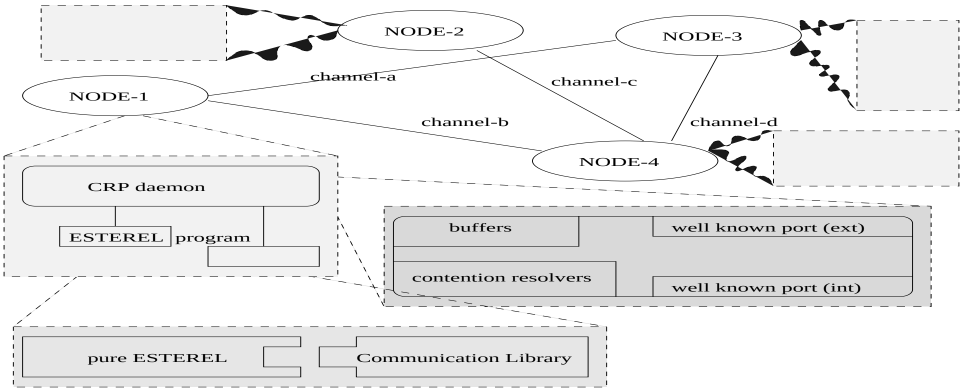


Figure 6: CRP Implementation

A network of CRP nodes is represented in Figure 6. Each node is capable of communicating with other nodes over designated channels. These channels are implemented using TCP connections between Unix nodes. Each node in the network runs of a *CRP daemon* and a set of ESTEREL programs. The daemon takes care of inter node communication on behalf of all ESTEREL programs on the local node. It communicates with other *CRP* daemons and local ESTEREL programs using a set of *well known ports* and TCP connections. Each *CRP* daemon takes care of buffering signals as well as *contention* resolution to effect deadlock freedom amongst competing processes. Each ESTEREL program in turn is linked with a communication library that encapsulates all the necessary communication functionality. The functions of this library are what the ESTEREL program will perceive as the asynchronous communication tasks it spawns.

For real time applications, the *CRP* programs will have to be translated into equivalent hardware implementations (say, using FPGAs) and embedded in the system of interest. The software implementation assumes importance because all initial testing [of desired functionality] can be done in software which more economical. Of course, these software version when run on *realtime* Unix platforms can be used for a number of real control applications.

Some preliminary work relating to implementation of the *CRP* protocol directly on hardware is also in progress.

5 Protocol Implementing the Rendezvous

Nondeterminism plays a crucial role in specifications and implementations. As highlighted already, in the context of reactive languages *determinism* is a must and the nondeterminism at the specification level would be replaced through input indeterminacy. Before we describe the actual protocol, we first look at the possibility of nondeterminism in *CRP* and the issues that arise in the design of deadlock-free protocols.

5.1 Effect of Nondeterminism

Let us consider the nondeterminism arising from the use of the communication primitives in conjunction with the parallel operator and the implementation of non-deterministic choice using rendezvous.

As the underlying ESTEREL language supports the parallel operator and we do not restrict communication primitives in distinct parallel threads we can simulate nondeterministic choice in *CRP* using the *exclusion relations* of ESTEREL [4]. Further, as there is no restriction in the use of the rendezvous primitive in parallel branches with reference to channel names (i.e., the same channel name can also be used), there is also a degree of nondeterminism at the level of matching `recv` and `send` statements during a rendezvous. In other words, we can simulate nondeterministic choice implicitly. Since the nondeterministic choice is useful from the point of view of the programmer, *CRP* has introduced an *explicit* guarded (i.e., rendezvous) choice primitive, similar to the one in CSP. We show that the protocol described thus far, with a few enhancements (that do not detract from anything discussed above), is sufficient to implement CSP style guarded commands executed in a context that supports *preemption*.

In the following, we show how non-deterministic choice can be simulated. For purposes of presentation, we use only two rendezvous choices. The following code segments give *equivalent* code written using the *rendezvous* primitive to implement the *guarded choice* construct.

$$
\begin{array}{lcl}
 & & \textsc{receipt}_A \oplus \textsc{receipt}_B; \\
 & & \texttt{trap}\ \text{done}\ \texttt{in} \\
\begin{array}{l} [\ \texttt{send}_A \ \to \mathit{skip} \\ [\!]\ \texttt{send}_B \ \to \mathit{skip}] \end{array} & \equiv & \qquad \begin{array}{l}\texttt{send}_A;\\ \texttt{exit}\ \text{done};\end{array} \ \| \ \begin{array}{l}\texttt{send}_B;\\ \texttt{exit}\ \text{done};\end{array} \\
 & & \texttt{end} \quad \%\ \text{trap}
\end{array}
$$

$$
\begin{array}{lcl}
 & & \textsc{data}_A \oplus \textsc{data}_B; \\
 & & \texttt{trap}\ \text{done}\ \texttt{in} \\
\begin{array}{l} [\ \texttt{recv}_A \ \to \mathit{skip}; \\ [\!]\ \texttt{recv}_B \ \to \mathit{skip}] \end{array} & \equiv & \qquad \begin{array}{l}\texttt{recv}_A;\\ \texttt{exit}\ \text{done};\end{array} \ \| \ \begin{array}{l}\texttt{recv}_B;\\ \texttt{exit}\ \text{done};\end{array} \\
 & & \texttt{end} \quad \%\ \text{trap}
\end{array}
$$

$$
\begin{array}{lcl}
 & & \textsc{receipt}_A \oplus \textsc{data}_B; \\
 & & \texttt{trap}\ \text{done}\ \texttt{in} \\
\begin{array}{l} [\ \texttt{send}_A \ \to \mathit{skip}; \\ [\!]\ \texttt{recv}_B \ \to \mathit{skip}] \end{array} & \equiv & \qquad \begin{array}{l}\texttt{send}_A;\\ \texttt{exit}\ \text{done};\end{array} \ \| \ \begin{array}{l}\texttt{recv}_B;\\ \texttt{exit}\ \text{done};\end{array} \\
 & & \texttt{end} \quad \%\ \text{trap}
\end{array}
$$

The use of the " *signal-1* $\oplus$ *signal-2* " statement is explained below. It serves as a directive to the *local daemon* to *serialize* the occurrences of the remote signals *signal-1* and *signal-2* in the following way:

- If both *signal-1* and *signal-2* are RECEIPT signals, then the *local daemon* is to forward the signal occurring first (or choose one at random, if they occur *simultaneously*) to the matching *sender* and generate an S-ABORT for the other (if any), to be passed on to the *receiver*. This has the desired effect of allowing exactly one of the guarded `send` commands to succeed. The other will be preempted by the enclosing `trap` statement.
- If both *signal-1* and *signal-2* are DATA signals, then the *local daemon* is to forward the signal occurring first (or choose one at random, if they occur *simultaneously*) to the matching *receiver* and store the other (if any).
  - If the selected `recv` statement completes successfully (*i.e.* the corresponding *sender* emits a CONFIRM) the enclosing `trap` statement will preempt the other `recv` statement.
  - On the other hand, if the selected `recv` statement receives an S-ABORT , the *local daemon* forwards the *stored* DATA signal to the other `recv` statement (thereby selecting it).
  - It should be noted, that should a DATA signal for one `recv` arrive while the other is selected, it is stored, to be dealt with as above.
  - It is easy to see that this behavior of the *local daemon* , in conjunction with the enclosing `trap`, has the effect of allowing exactly one of the guarded `recv` commands to succeed.
- If one of *signal-1* or *signal-2* is a RECEIPT signal and the other a DATA signal, the *local daemon* 's behaves as follows. One of the rendezvous statements is selected (depending on which signal arrived first or by choosing at random if they occurred *simultaneously*) and the other signal (if any) is stored. That is,
  - If the selected statement is a `send`, it is bound to succeed and therefore the enclosing `trap` will *preempt* the `recv`.
  - On the other hand, if the selected statement is a `recv`, the *local daemon* waits to see if the rendezvous completes successfully. If the rendezvous completes successfully the *local daemon* generates an S-ABORT signal for the RECEIPT signal it had stored (if at all) to be forwarded to the remote *receiver*. Otherwise, the *local daemon* forwards the stored RECEIPT signal (if any) to the `send` statement (thereby selecting it).

– If a RECEIPT signal arrives (from the remote *receiver*) while the `recv` statement is selected, it is stored to be dealt with as above.

This behavior of the *local daemon* together with the enclosing `trap` statement ensures that utmost one of the guarded commands succeeds.

5.2 Deadlock Prevention

An important aspect to be kept in mind while mixing `send` and `recv` in the choice is that the above behaviour of the *local daemon* can then lead to a deadlock even though the deadlock may not be there in the program semantically; in other words, the implementation could introduce deadlock. To overcome this problem the following modifications are introduced in the *local daemon* behaviour. Again, for simplicity, the description deals with deadlock occurring between two nodes.

Consider two nodes trying to execute the following guarded commands.

[$\mathtt{send}_A$; [] $\mathtt{recv}_B$]	|	[$\mathtt{send}_B$; [] $\mathtt{recv}_A$]
node-1		node-2

As would follow from the discussion presented above, execution would proceed to the state where DATA_A (the data sent from *node-1*) would have been delivered at *node-2* and RECEIPT $_A$ from the `recv` on *node-2* would be delivered to the *local daemon* at *node-1*. As both ends are symmetric, a situation where the `recv` statements at both ends are *pre-selected* arises. The *deadlock* that results can be broken by the techniques described described below.

5.2.1 A distributed implementation

We outline below a deadlock-free algorithm which, being truly distributed, does not depend on a global clock.

The algorithm is described below:

- Every distinct *daemon d* is assigned a unique constant C_d which remains invariant through sessions while $D_d = f(C_d, x_i)$ such that it is unique for pairs (C_d, x_i) in the system is computed for every active rendezvous primitive.

- Three new signals, RQST , PERM and VOID are introduced which are *local* to the *daemons*. The RQST signal is used to *request* permission to select a guarded command and PERM signifies that the request is *granted*. On the other hand, the VOID signal is used to signal the forfeiture of a PERM signal obtained.

- Every *local daemon* handling a set of guarded commands operates as follows:

  1. Each *local daemon d*, despatches signals DATA , RECEIPT , PERM , RQST and VOID tagged with the unique number D_d it generates.
  2. For every DATA signal received with a tag value *less than* D_d, a *local daemon* despatches a RQST signal (tagged with D_d).
  3. From the available set of RECEIPT signals (if any), one is chosen and the corresponding guarded command selected. Such a choice ensures progress as the CONFIRM signal that will be emitted in response would conclude the guarded rendezvous and will in turn preempt the other commands in the same statement. If no RECEIPT signals are available, the following rule is applied.
  4. From the set of available PERM signals, choose one and select the corresponding *receiver* command. This will result in the RECEIPT signal being issued and as the recipient has already expressed willingness to accept a RECEIPT signal, progress is assured.
  5. If no PERM signals were available, consider the RQST signals next. For every RQST signal send back a PERM signal.

6. If no outstanding PERM signals exist attend to the DATA signals by choosing one from among the set of untried ones and selecting the corresponding guarded command.

Again, the transition diagrams shown in section 5 do not show the transitions required to implement the guarded rendezvous.

6 Analysis of the Solution

In this section, we discuss the properties satisfied by the solution, its applicability as a guarded choice implementation in general and the realizability of the implementation assumptions.

6.1 Properties of the Solution

In this section, we analyze the properties of the solution informally rather than formally due to lack of space. In fact, we have used Auto/Autograph for the analysis of deadlock-freedom of the protocol.

The protocol proposed above ensures a correct implementation of the rendezvous primitive in networks of *preemptible* processes. The protocol guarantees a *consistent* view of the system to each networked node and is *distributed* in its implementation. It is easy to see that the number of messages exchanged by a process to find a matching pair is *bounded*. In the following, we show that the protocol is *deadlock free* and that it preserves the *reactive* (local) behavior of the synchronous modules.

6.1.1 The Solution Preserves Reactivity

Implementations that rely on the notion of a *freeze* state are easy to realize but do not capture the intuitive semantics of a *reactive* execution context that supports *preemption*. Further, the use of a freeze state would severely restrict the applicability of the method to *real time* environments subject to bounds on response times. The protocol described above does not use the notion of *freezing* either *sender* or *receiver* during any phase of the rendezvous. It can therefore be implemented using existing constructs in a synchronous language, preserving the intuitive meaning of *preemption* primitives and the *reactive* properties of the language.

6.1.2 Deadlock Freedom

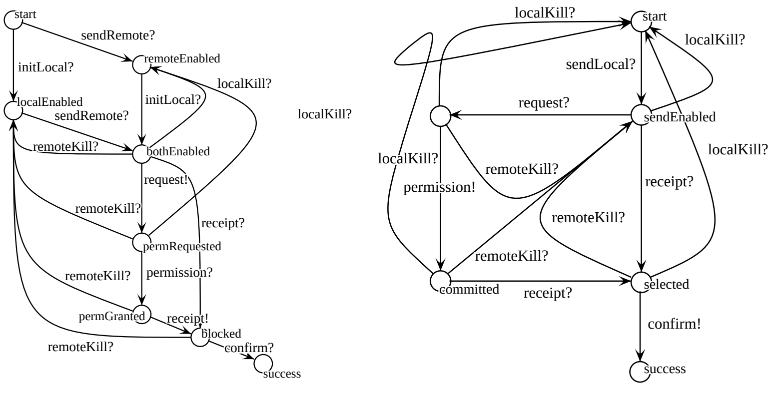


Figure 7: Transitions in a Rendezvous

In this section we will present an argument to show that the implementation cannot deadlock. The diagrams above show the transitions a SEND-DAEMON$_i$ and a RECV-DAEMON$_j$ would make (for a given `send` or `recv` primitive) during the establishment of a rendezvous .

1. Let the *daemons* participating in establishing a rendezvous at a given instant be numbered from 1 to n.

2. The implementation deadlocking would correspond to *all n daemons* getting into the state labelled *blocked* in the figure.

3. For a RECV-DAEMON$_i$ to block on a SEND-DAEMON$_j$ either $i < j$ or SEND-DAEMON$_j$ is in the state labelled *committed*.

   i) Utmost $n-1$ of the RECV-DAEMON$_i$s could block because $i < j$ for some j.

   ii) This would imply that at least one RECV-DAEMON$_i$ blocked after the corresponding SEND-DAEMON$_j$ entered the state labelled *committed*.

   iii) This in turn, precludes the possibility of *all n daemons* being in the blocked state simultaneously.

4. Therefore, we see that the implementation *cannot* deadlock.

We have been able establish the deadlock-freedom using tools such as Auto/Autograph.

6.1.3 CSP Style Guarded Commands

We have given an implementation of a CSP like *nondeterministic choice* primitive using the rendezvous construct and the *parallel* operator in section 5.1. Note that as *preemption* was not available in the CSP context, the semantics of our implementation of guarded choice will necessarily differ from that used in CSP. It should however be clear that modulo *preemption*, our semantics for the guarded choice is the same as the CSP semantics for it. As explained above, the implementation can handle *mixed* guards too. However, in some contexts, the CSP style rendezvous semantics assumes importance. Our implementation of CRP has therefore provided a user programmable option of preserving the rendezvous semantics. Using this feature, it is possible, at run time, to decide which semantics a communication primitive is to provide.

This feature is implemented by extending the daemons to cover the possibility that a client might be frozen for a part of the rendezvous.*i.e.* A daemon never selects a client that is *known* to be in a frozen state.

On the client side, the programmer choosing to preserve the rendezvous semantics at the cost of the reactive semantics arranges for the node to *suspend* after emitting the RECEIPT signal. This precludes the possibility of local preemption leading to inconsistency.

7 Conclusion

We have presented a protocol specification that implements a rendezvous primitive between a network of preemptible synchronous processes. We have also shown that there is no explicit need for introducing a nondeterministic operator as the same could be realized through the rendezvous and the *parallel* construct. Further, we have shown how the *derived* choice construct can in fact be implemented. Thus it offers another implementation for CSP as well. Further, the protocol is shown to be *deadlock free*. Unlike previous implementations, this protocol does not require/introduce a *freeze* state and allows for *preemption* at *any* stage of the rendezvous (unless the programmer chooses to preserve the rendezvous semantics at the cost of reactivity). The new implementation for *CRP* is based entirely on this protocol. In doing so, we have been able to preserve the desirable (local) properties of the individual nodes (like *reactivity*) while at the same time extending the synchronous paradigm to a distributed environment.

References

[1] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design semantics, Implementation. *SCP*, 19(2):87–152, Nov 92.

[2] C.A.R. Hoare. Communicating Sequential Processes. *CACM*, 21(8), Aug 78.

[3] R.K. Shyamasundar. Specification of Hybrid Systems in CRP. In *Proc. of AMAST 93*, Workshops in Computing Series, pages 227–238. Springer-Verlag, Dec 93.

[4] G. Berry, S. Ramesh, and R.K. Shyamasundar. Communicating Reactive Processes. *ACM POPL*, pages 85–99, Jan 93.

Appendix

Protocol Specification

In this section, we will describe the complete protocol using a guarded-command structure.

Data Structures used: The protocol uses the following data-structures:

- The *tuple* <SIG,*channel*> represents a signal SIG emitted by a local synchronous process to its communication *daemon* on channel *channel*.
- The <SIG,*channel*,*ident*> represents a signal SIG, emitted by daemon, *ident*, on channel *channel*.
- IN-SET and OUT-SET are two variables that store incoming and outgoing signals respectively. Information pertaining to the *state* of the communication *daemon* is encoded by the contents of these two variables.
- The predicate `present`(SIG) is `true` *if and only if* signal SIG is present in IN-SET .
- Predicates `commit` and `blocked` are used to test for the presence of PERMISSION and RECEIPT in OUT-SET respectively. To be more precise:
  - `commit` ≡ ∃h:<PERMISSION ,h,i> ∈ OUT-SET
  - `blocked` ≡ ∃h: <RECEIPT ,h,i> ∈ OUT-SET
  - `neither` ≡ ¬`commit`∧¬`blocked`
- To implement *CSP* semantics for a rendezvous it becomes necessary to refine our protocol further.
  - We associate a boolean state variable, $\texttt{freeze}_k$, with every client, k. Additionally we define `frozen` ≡ if ($\texttt{freeze}_k$ = `true`) then `true` else `false`
  - The *daemon* will use the signal FREEZE_k to instruct client k to freeze after acknowledging it.
  - Every client, k, will acknowledge a request from its associated *daemon* with a FROZE_k signal after which it will suspend all activity until the daemon *thaws* it.
  - The *daemon* will use the signal THAW_k to bring a client k out of a freeze state.

In the protocol given below, the guarded commands represent a possible action that the *daemon* can take. Every signal received by the communication *daemon* enables exactly one of these guards and the corresponding statement-sequence is executed. It should be noted that some guards when enabled merely *record* the occurrence of a signal for further processing at a later time. Code related to sequence control and other book-keeping activities are bunched together towards the end of the protocol and the the action CLEANUP is described at the end for purposes of ease of understanding. It may be noted that the algorithm is somewhat asymmetrical with respect to the sender and the receiver. The guarded transitions given below closely follows the descriptions that are given earlier through Figures 2-5 and the technique discussed towards a deadlock-free solution in section 5.2.2. For lack of space, we avoid describing it again here.

```
IN-SET  := φ;                                    /* set of incoming signals */
OUT-SET  := φ;                                   /* set of outgoing signals */
i  := local-daemon-id;                           /* unique for each daemon */
present(X) ≡ X ∈ IN-SET ;                        /* checks presence of a signal */
do
```

```
[] <DATA ,h>  →
          /* send initialized; update state ; notify receiver daemon */
          IN-SET := IN-SET ∪ {<DATA ,h>};
          emit <DATA ,h,i>;

[] <INIT ,h>  →
          /* receiver initialized; update state */
          IN-SET := IN-SET ∪ {<INIT ,h>};

[] <DATA ,h,j>  →
          /* remote sender ready; update state */
          IN-SET := IN-SET ∪ {<DATA ,h,j>};

[] <RECEIPT ,h,j>  →
          /* selected by remote receiver; process later */
          IN-SET := IN-SET ∪ {<RECEIPT ,h,j>};

[] <REQUEST ,h,j>  →
          /* request to select received; update state */
          IN-SET := IN-SET ∪ {<REQUEST ,h,j>};

[] <PERMISSION ,h,j>  →
          /* perm to select received; forget req to select */
          /* store perm received */
          IN-SET := IN-SET ∪ {<PERMISSION ,h,j>};
          OUT-SET := OUT-SET − {<REQUEST ,h,i>};

[] ¬blocked∧present(<INIT ,h>)∧¬present(<PERMISSION ,h,j>)∧
   present(<DATA ,h,j>)∧(j < i)  →
          /* sender, receiver ready; req permission to select */
          /* remember req sent */
          OUT-SET := OUT-SET ∪ {<REQUEST ,h,i>};
          emit <REQUEST ,h,i>;

[] ¬frozen∧neither∧present(<INIT ,h>)∧present(<DATA ,h,j>)∧(j ≥ i)  →
          /* sender,receiver enabled; neither committed */
          /* nor blocked, so select */
          OUT-SET := OUT-SET ∪ {<RECEIPT ,h,i>};
          set frozen := true;
          emit <FREEZE_k ,h,i>;

[] ¬frozen∧¬blocked∧present(<REQUEST ,h,j>)  →
          /* received req to select and not blocked */
          /* grant permission to select */
          OUT-SET := OUT-SET ∪ {<PERMISSION ,h,i>};
          emit <PERMISSION ,h,i>;

[] ¬frozen∧neither∧present(<PERMISSION ,h,j>)  →
          /* neither blocked nor committed; have perm */
          /* to select; so select remote send */
          set frozen := true;
          emit <FREEZE_k ,h,i>;

[] ¬frozen∧¬blocked∧present(<RECEIPT ,h,j>)  →
          /* selected by remote receiver; free to accept */
          /* so send confirmation; complete book-keeping */
          emit <CONFIRM ,h>;
          emit <CONFIRM ,h,i>;
          CLEANUP

[] frozen∧present(<PERMISSION ,h,j>)∧<FROZE_k ,h,i>  →
          /* the receiver has agreed to freeze */
          IN-SET := IN-SET − {<PERMISSION ,h,j>};
          OUT-SET := OUT-SET ∪ {<RECEIPT ,h,i>};
          emit <RECEIPT ,h,i>;
```

```
[] frozen∧present(<DATA ,h,j>)∧<FROZE_k,h,i>∧(j ≥ i)  →
                          /* free to select sender, and receiver agrees to freeze */
                          OUT-SET := OUT-SET ∪ {<RECEIPT ,h,i>};
                          emit <RECEIPT ,h,i>;

[] frozen∧<ABORT ,h>∧present(<init,h>)  →
                          /* request to freeze denied */
                          set frozen := false;
                          CLEANUP

[] blocked∧<CONFIRM ,h,j>  →
                          /* rendezvous confirm by sender; inform  */
                          /* receiver, then complete book-keeping */
                          if (frozen) then
                          begin
                          set frozen := false;
                          emit <THAW_k>;
                          end
                          emit <CONFIRM ,h>;
                          CLEANUP

/* code related to book-keeping, preemption, non-deterministic choice */

[] <ABORT ,h>∧present(<DATA ,h>)  →
                          /* sender preempted; inform remote receiver */
                          emit <S-ABORT ,h,i>;
                          CLEANUP

[] <ABORT ,h>∧present(<INIT ,h>)  →
                          /* receiver preempted */
                          CLEANUP

[] <S-ABORT ,h,j>∧present(<PERMISSION ,h,j>)  →
                          /* remote sender preempted */
                          /* perm,data received is no longer valid */
                          IN-SET := IN-SET − {<PERMISSION ,h,j>};
                          IN-SET := IN-SET − {<DATA ,h,j>};

[] <S-ABORT ,h,j>∧(<RECEIPT ,h,i> ∈ OUT-SET )  →
                          /* abort received instead of conf */
                          OUT-SET := OUT-SET − {<RECEIPT ,h,i>};
                          IN-SET := IN-SET − {<DATA ,h,j>};
                          if (frozen) then begin
                          set frozen := false;
                          emit <THAW_k>;
                          end

[] <S-ABORT ,h,j>∧present(<DATA ,h,j>)∧¬present(<init,h>)  →
                          /* abort received before local */
                          /* receiver was enabled */
                          IN-SET := IN-SET − {<DATA ,h,j>};

[] present(<VOID ,h,j>)∧present(<REQUEST ,h,j>)  →
                          /* disregard request for received earlier */
                          IN-SET := IN-SET − {<REQUEST ,h,j>};

[] present(<VOID ,h,j>)∧(<REQUEST ,h,j> ∈ OUT-SET )  →
                          /* disregard permission granted earlier */
                          IN-SET := IN-SET − {<PERMISSION ,h,j>};
od
/* code for the CLEANUP procedure follows*/
CLEANUP:

    ∀h : <PERMISSION ,h,i> ∈ OUT-SET  →
                          emit <S-ABORT ,h,i>;
                          IN-SET := IN-SET − {<data,h> <init,h>};
                          OUT-SET := OUT-SET − {<perm,h,i>};
    ∀h : <REQUEST ,h,i> ∈ OUT-SET  →
                          emit <VOID ,h,i>;
                          IN-SET := IN-SET − {<data,h> <init,h,i>};
                          OUT-SET := OUT-SET − {<rqst,h,i>};
```

The banker-teller problem

In this section we describe a typical situation that cannot be modeled using either ESTEREL or CSP alone. We however can use CRP to address the problem (thus showing CRP to be strictly more powerful that either ESTEREL or CSP).

The banker-teller problem can be informally described by the following steps:

- the teller machine reads a card
- authenticates the card and interacts with the customer locally
- communicates with the bank to perform the actual transaction
- returns the card and money
- the user can cancel the transaction at any time (by pressing say a CANCEL key).

Due to the possibility of preemption, CSP cannot be used to specify the problem. On the other hand, the asynchronous communication between the bank and teller machine makes it impossible to specify the problem wholly in ESTEREL either. We can however use CRP to model the situation as it supports both locally reactive nodes as well as asynchronous communication between such nodes.

It being essential that consistency be maintained regarding transactions within the system, it seems necessary to require that the rendezvous between the teller and the bank be done preserving the rendezvous semantics as in CSP. In particular, we would like to avoid cases where a debit/credit operation was *canceled* by the user while the bank actually journals it.

A careful examination of the CRP protocol shows that due to the asymmetry inherent in the sender and receiver only a preemption at the *receiver* end can lead to inconsistent views of the rendezvous status. Therefore, if we can ensure that the teller will always be the *sender* in a rendezvous, we can preserve consistency *and* maintain the *reactivity* semantics for the rendezvous. This argument of course assumes that the bank is *never* preempted.

On the other hand, if we choose to implement the communication using the rendezvous semantics (as in CSP), we would need to constrain the process modeling both the bank and the teller to be free from preemption at least during certain phases of the communication. This in turn would detract from the original specification of the problem where the user was allowed to preempt at any point. Instead, the user will have to at some point confirm his intentions and *wait* for the transaction to be journalled at the bank.