Communicating Reactive Processes*

G. Berry
Ecole des Mines, Centre de Mathématiques Appliquées
B.P. 207, 06904 Sophia-Antipolis CDX, France

S. Ramesh
Dept. Of Computer Science and Engineering
Indian Institute of Technology
Powai, Bombay 400 076, India

R.K. Shyamasundar
Tata Institute of Fundamental Research
Homi Bhabha Road, Bombay 400 005, India

Abstract

We present a new programming paradigm called Communicating Reactive Processes or CRP that unifies the capabilities of asynchronous and synchronous concurrent programming languages. Asynchronous languages such as CSP, OCCAM, or ADA are well-suited for distributed algorithms; their processes are loosely coupled and communication takes time. The ESTEREL synchronous language is dedicated to reactive systems; its processes are tightly coupled and deterministic, communication being realized by instantaneous broadcasting. Complex applications such as process or robot control require to couple both forms of concurrency, which is the object of CRP. A CRP program consists of independent locally reactive ESTEREL nodes that communicate with each other by CSP rendezvous. CRP faithfully extends both ESTEREL and CSP and adds new possibilities such as precise local watchdogs on rendezvous. We present the design of CRP, its semantics, a translation into classical process calculi for program verification, an application example, and implementation issues.

*Work supported by the French Coordinated Research Projects C^3 and C2A and by IFCPAR (Indo-French Center for the Promotion of Advanced Research), New Delhi.

0

1 Introduction

Existing concurrent programming languages fall into two quite distinct classes that we shall call the asynchronous class and the synchronous class. The goal of this paper is to provide a unification between asynchronous and synchronous programming.

The class of asynchronous languages contains classical concurrent languages such as CSP [13], OCCAM [14], or ADA, [1]. Concurrent processes are viewed as being loosely coupled independent execution units, each process evolving at its own pace. Inter-process communication is done by mechanisms such as message passing or rendezvous. Communication as a whole is asynchronous in the sense that an arbitrary amount of time can pass between the desire of communication and its actual completion (rendezvous communication is sometimes called synchronous, but we prefer to call it synchronizing since the actual rendezvous is only the final act of a communication).

The class of synchronous languages contains mainly ESTEREL [5], LUSTRE [11], SIGNAL, [10], and STATECHARTS [12]. In synchronous languages, a program is thought of as reacting instantaneously to its inputs by producing the required outputs. Concurrent statements evolve in a tightly coupled input-driven way and communication is done by instantaneous broadcasting, the receiver receiving a message exactly at the time it is sent.

Asynchronous and synchronous languages are deeply different in nature, applications, and implementations. Asynchronous languages are relevant for distributed algorithms; they support non-determinism, which is appropriate for the framework. Implementing them on networks of processors is natural, since the linguistic asynchronous communication mechanisms are close to actual communication mechanisms in networks. Synchronous languages are specifically designed for reactive systems [2]; here logical concurrency is required for good programming style but determinism is a must — think of embedded controllers. Instantaneous reaction and instantaneous broadcast communication make it possible to marry concurrency and determinism. Direct implementation is feasible in hardware [3,4]. In software implementation, concurrency can be compiled away, for example by translating programs to straight-line code or to automata [5,11].

Presently, each language class is unable to handle the problems to which the other class is tailored. Asynchronous languages are inappropriate for truly reactive systems that require deterministic synchronous communication, as argued in [2]. On the other hand, existing synchronous languages lack support for asynchronous distributed algorithms. However, complex systems do require the abilities of the two classes of languages. For example, a robot driver must use a specific reactive program to control each articulation, but the global robot control may be necessarily asynchronous because of limitations of networking capabilities.

The paper develops a new unifying paradigm called *Communicating Reactive Processes*, where a set of individually reactive synchronous processes is linked by asynchronous communication channels. Technically, we unify the ESTEREL and CSP languages. This unification requires only a minor addition to ESTEREL. It preserves the spirit and semantics of both ESTEREL and CSP. At the same time, it also provides a rigorous semantics and implementation for constructs such as precise local watchdogs on asynchronous communications that are indispensable in practice but are not properly supported by existing languages.

Section 2 recalls the ESTEREL language and introduces a new `exec` asynchronous task execution primitive. In Section 3, we show how to augment ESTEREL with rendezvous based on asynchronous task execution. We show that all the constructs of CSP can be implemented by using constructs already existing in ESTEREL. We present new possibilities of fine synchronous control over asynchronous communication, and we give an application example. Section 4 presents the semantic model and shows that it conservatively extends those of ESTEREL and CSP. In Section 5, we give a translation of CRP into the MEIJE process calculus [6]; this translation gives an operational view of the semantics and makes it possible to automatically verify properties of CRP programs using a verification system such as AUTO [7]. Section 6 discusses implementation issues. An appendix presents the formal semantics of ESTEREL with `exec`.

For simplicity, we only deal formally with PURE ESTEREL, which is limited to pure synchronization and communication [3,4]. The full ESTEREL language [5] supports data handling and value passing. Extension of CRP to value passing is easy; it is done informally and is illustrated in the example.

2 The Esterel Language

We first briefly present the original PURE ESTEREL language of [3,4]. We then present the new `exec` asynchronous task execution primitive that is now supported by the ESTEREL compilers. We explain the language by giving a purely intuitive and non-systematic semantics based on examples.

2.1 Modules and Interfaces

The basic object of PURE ESTEREL is the *signal*. Signals are used for communication with the environment as well as for internal communication.

The programming unit is the module. A module has an interface that defines its input and output signals and a body that is an executable statement:

```
module M:
input I1, I2;
output O1, O2;
input relations
statement
end module
```

Input relations can be used to restrict input events [5]. We shall only use *exclusions*, written in the interface part as

```
relation I1 # I2;
```

Such a relation means that input events cannot contain `I1` and `I2` together. It is therefore an assertion on the behavior of the asynchronous environment.

At execution time, a module is activated by repeatedly giving it an *input event* consisting of a possibly empty set of input signals assumed to be present and satisfying the input relations. The module reacts by executing its body and outputs the emitted output signals. We assume that the reaction is *instantaneous* or *perfectly synchronous* in the sense that the outputs are produced in *no time*. Hence, all necessary computations are also done in no time. In PURE ESTEREL, these computations are either signal emissions or control transmissions between statements; in full ESTEREL, they can be value computations and variable updates as well. The only statements that consume time are the ones *explicitly* requested to do so. The reaction is also required to be *deterministic*: for any state of the program and any input event, there is exactly one possible output event. Perfect synchrony is discussed at length in [5,2,11,10,12]. In perfectly synchronous languages, a reaction is also called an *instant*.

2.2 Statements

ESTEREL has two kinds of statements: the primitive or *kernel* statements, and the *derived* statements that can be expanded into primitive ones by macro-expansion and make the language more user-friendly. Derived statements are not semantically meaningful and will not be presented here. The list of kernel statements is:

```
nothing
halt
emit S
stat1; stat2
loop stat end
present S then stat1 else stat2 end
do stat watching S
stat1 || stat2
trap T in stat end
exit T
signal S in stat end
```

The kernel statements are imperative in nature, and most of them are classical in appearance. The `trap-exit` constructs form an exception mechanism fully compatible with parallelism. Traps are lexically scoped. The local signal declaration "`signal S in` *stat* `end`" declares a lexically scoped signal `S` that can be used for internal broadcast communication within *stat*. The `then` and `else` parts are optional in a `present` statement. If omitted, they are supposed to be `nothing`.

2.3 Intuitive Semantics

At each instant, each interface or local signal is consistently seen as present or absent by all statements, ensuring determinism. By default, signals are absent; a signal is present if and only if it is an input signal emitted by the environment or a signal internally broadcast by executing an `emit` statement.

To explain how control propagates, it is better to first give examples using the simplest derived statement that takes time: the waiting statement "`await S`", whose kernel expansion "`do halt watching S`" will be explained in a moment. When it starts executing, this statement simply retains the control up to the first *future* instant where `S` is present. If such an instant exists, the `await` statement terminates immediately; that is, the control is released instantaneously; If no such instant exists, then the `await` statements waits forever and never terminates. If two `await` statements are put in sequence, as in "`await S1; await S2`", one just waits for `S1` and `S2` in sequence: control transmission by the sequencing operator '`;`' takes no time by itself. In the parallel construct "`await S1 || await S2`", both `await` statements are started simultaneously right away when the parallel construct is started. The parallel statement terminates exactly when its two branches are terminated, i.e. when the last of `S1` and `S2` occurs. Again, the '`||`' operator takes no time by itself.

Instantaneous control transmission appears everywhere. The `nothing` statement is purely transparent: it terminates immediately when started. An "`emit S`" statement is instantaneous: it broadcasts `S` and terminates right away, making the emission of `S` transient. In "`emit S1; emit S2`", the signals `S1` and `S2` are emitted simultaneously. In a signal-presence test such as "`present S ...`", the presence of `S` is tested for right away and the `then` or `else` branch is immediately started accordingly. In a "`loop` *stat* `end`" statement, the body *stat* starts immediately when the `loop` statement starts, and whenever *stat* terminates it is instantaneously restarted afresh (to avoid infinite instantaneous looping, the body of a loop is required not to terminate instantaneously when started).

The `watching` and `trap-exit` statements deal with behavior *preemption*, which is the most important feature of ESTEREL.

In the watchdog statement "`do` *stat* `watching S`", the statement *stat* is executed normally up to proper termination or up to future occurrence of the signal `S`, which is called the *guard*. If *stat* terminates *strictly* before `S` occurs, so does the whole `watching` statement; then the guard has no action. Otherwise, the occurrence of `S` provokes immediate preemption of the body *stat* and immediate termination of the whole `watching` statement. Consider for example the statement

```
do
  do
    await I1; emit O1
  watching I2;
  emit O2
watching I3
```

If `I1` occurs strictly before `I2` and `I3`, then the internal `await` statement terminates normally, `O1` is emitted, the internal `watching` terminates since its body terminates, `O2` is emitted, and the external `watching` also terminates since its body does. If `I2` occurs before `I1` or at the same time as it, but strictly before `I3`, then the internal `watching` preempts the `await` statement that should otherwise terminate, `O1` is *not* emitted, `O2` is emitted, and the external `watching` instantaneously terminates. If `I3` occurs before `I1` and `I2` or at the same time as them, then the external `watching` preempts its body and terminates instantaneously, no signal being emitted. Notice how nesting watching statements provides for *priorities*.

We can now explain why "`await S`" is defined as "`do halt watching S`". The semantics of `halt` is simple: it keeps the control forever and never terminates. When `S` occurs, `halt` is preempted and the whole construct terminates just as expected. Notice that `halt` is the only kernel statement that takes time by itself.

The `trap-exit` construct is similar to an exception handling mechanism, but with purely static scoping and concurrency handling. In "`trap T in` *stat* `end`", the body *stat* is run normally until it executes an "`exit T`" statement. Then execution of *stat* is preempted and the whole `trap` construct terminates. The body of a `trap` statement can contain parallel components; the `trap` is exited as soon as one of the components executes an "`exit T`" statement, the other components being preempted. However, `exit` preemption is weaker than `watching` preemption, in the sense that concurrent components execute for a last time when exit occurs. Consider for example the statement

```
trap T in
    await I1; emit O1
  ||
    await I2; exit T
end
```

If `I1` occurs before `I2`, then `O1` is emitted and one waits for `I2` to terminate. If `I2` occurs before `I1`, then the first branch is preempted, the whole statement terminates instantaneously, and `O1` will never be emitted. If `I1` and `I2` occur simultaneously, then *both branches do execute* and `O1` is emitted. Preemption occurs only after execution at the concerned instant: by exiting a trap, a statement can preempt a concurrent statement, but it does leave it its "last wills".

Since we accept simultaneity, we must define what it means to exit several traps simultaneously, i.e. define priorities between traps. The rule is simple: *only the outermost trap matters, the other ones being discarded.* For example, in

```
trap T1 in
  trap T2 in
      exit T1
    ||
      exit T2
  end;
  emit O
end
```

the traps `T1` and `T2` are exited simultaneously, the internal trap `T2` is discarded and `O` is not emitted.

Traps also provide a way of breaking loops, which would otherwise never terminate:

```
trap T in
  loop ... exit T ... end
end
```

2.4 Value Handling

Since full ESTEREL will be informally used in the sequel, we briefly describe the way in which values are handled.

Types can be either predefined like `integer` or be abstract like `Time`; abstract types are meant to be implemented in the host language in which a program is compiled, C or ADA for example.

A signal can carry a value of a type declared in the signal declaration. A valued signal has a *unique* value at each instant. A signal value may change only when the signal is received from the environment or locally emitted with a new value, by executing "`emit S(`*exp*`)`". The current value of a signal `S` is accessed at any time by the expression '`?S`'.

One can declare local variables by the statement

`var X in` *stat* `end`

Variables deeply differ from signals by the fact that they cannot be shared by concurrent statements. Variables are updated by instantaneous assignments "`X:=`*exp*" or by instantaneous side-effecting procedure calls "`call P(...)`", where a procedure `P` is an external host-language piece of code that receives both value and reference arguments.

Expressions may involve variables, signal values '`?S`', and external host-language function calls (external functions must not perform side effects). The computation of an expression is instantaneous. The "`if` *exp* `then` $stat_1$ `else` $stat_2$ `end`" statement instantaneously tests for the truth of *exp*.

Finally, occurrence counters can be added to preemption statements, as in "`do` *stat* `watching 5 S`".

2.5 The "exec" Statement

We mentioned that full ESTEREL external procedure calls are assumed to be instantaneous. This is not appropriate for long numerical computations or for dealing with external actions such as "move the robot arm to position (x, y)" [9]. The new `exec` statement [15] remedies this defect of the original ESTEREL language and is our gateway to asynchrony. It handles external *asynchronous tasks* homogeneous to procedures. In PURE ESTEREL, we only handle argumentless side-effecting tasks. An asynchronous task is declared by "`task P`", and task execution is controlled from ESTEREL by the statement

`exec P`

that starts `P` and waits for its completion to terminate. Since there can be several occurrences of "`exec P`" in a module for the same task `P`, several simultaneously active tasks having the same name can coexist. To avoid confusion among them, one can assign an explicit label to each `exec` statement:

`exec L : P`

The label name must be distinct from all other labels and input signal names. An implicit distinct label is given to unlabeled `exec` statements.

Given an `exec` statement labeled `L`, the asynchronous task execution is controled from ESTEREL by three implicit signals `sL`, `L`, and `kL`. The output *start signal* `sL` is sent to the environment when the `exec` statement starts. It requests to start an asynchronous incarnation of the task (passing values in full ESTEREL). The input *return signal* `L`, is sent by the environment when the task incarnation is terminated; it provokes instantaneous termination of the ESTEREL `exec` statement (and update of reference parameters in full ESTEREL). The output *kill signal* `kL` is emitted by ESTEREL if the `exec` statement is preempted before termination, either by an enclosing watching statement or by concurrent exit from an enclosing trap. For example, this is the case if `S` occurs before termination of `P` in

```
do
  exec L : P
watching S
```

It is assumed that the asynchronous environment cannot provide instantaneous feedback, so that an `exec` statement cannot terminate instantaneously when it is started. If one forgets about `kL`, "`exec L : P`" is simply

```
emit sL; await L
```

But the generation of `kL` is non-local since it depends on external preemption. This is why we must have `exec` as a primitive. In practice, the `kL` signal is essential to garbage-collect preempted computations or to stop external actions. In our implementation of CSP, it will play a central role to monitor communication.

Notice that a task may be restarted instantaneously at termination time, as in

```
loop
  exec L : P
end
```

or restarted instantaneously when killed, as in

```
loop
  do
    exec L : P
  watching S
end
```

In that case, the signals `kL` and `sL` are emitted simultaneously. Of course, `kL` kills the currently existing incarnation and `sL` starts a new incarnation.

There are also situations where a task should be started but killed immediately. For example, consider

```
trap T in
    exec L : P
  ||
    exit T
end
```

The `exec` statement starts in the first parallel branch, but it is instantaneously preempted by the `exit` statement. In this case neither the start signal `sL` nor the kill signal `kL` are emitted and no asynchronous task is started.

An `exec` return signal can be declared to incompatible with an input signal or with another return signal by writing an incompatibility input relation using the `exec` label to refer to the return signal name.

3 Communicating Reactive Processes

Let us now present the Communicating Reactive Processes or CRP model. It consists of a network $M_1//M_2//\ldots//M_n$ of ESTEREL reactive programs or *nodes*, each having its own input / output reactive signals and its own notion of an instant. The network is asynchronous and the nodes M_i communicate via asynchronous channels. Intuitively, each M_i is locally reactively driving a part of a complex process that is handled globally by the network.

To establish asynchronous communication between the nodes, the central idea is to extend the basic `exec` primitive into a communication primitive. The usual `send` and `receive` asynchronous operations can be represented by particular tasks that handle the communication. Several `send` / `receive` interactions are possible according to various types of asynchronous communication. For instance, `send` can be non-blocking for full asynchrony, or `send` and `receive` can synchronize for CSP-like rendezvous communication. All choices can be implemented through ESTEREL. We choose the latter as that is the most subtle.

In the sequel, we first introduce a `rendezvous` primitive in ESTEREL. We then show that this is enough to capture the full communication power of CSP. We show that synchronous and asynchronous constructs can be combined to realize fine communication control in a way that cannot be handled by conventional asynchronous languages, and we give an application example.

3.1 Rendezvous

CRP nodes are linked by *channels*. We start by describing the pure synchronization case, where channels are symmetric. Pure channels are declared in CRP nodes by the declaration

```
channel C;
```

A channel must be shared by exactly two nodes. Channels are handled in ESTEREL by the new statement

```
rendezvous L : C
```

The label `L` is optional, an implicit distinct label being created if it is absent. Except in the example below, we shall always use explicit channel names for more clarity.

The `rendezvous` statement is a particular instance of `exec` statement; as such, it defines three implicit reactive signals at a node : `sL`, `L`, and `kL`. The output signal `sL` requests for a rendezvous on `C`. Rendezvous completion is signaled to the node by the signal `L`. The signal `kL` signals abandoning the rendezvous request.

A given channel can only perform one rendezvous at a time. Hence, in each node, the return signals of all `rendezvous` statements on the same channel are implicitly assumed to be incompatible.

In full CRP, one can pass values through unidirectional channels. For example, one can declare:

```
input channel C1 : type;
output channel C2 : type;
```

To send and receive values, one writes

```
rendezvous L1 : C1(exp)
rendezvous L2 : C2
```

The value sent in the first rendezvous is simply handled as the value of the start signal `sL1`. The value received in the second rendezvous is simply the current value '`?L2`' of the return signal `L2`.

As far as implementation goes, one can think of an implicit asynchronous layer that handles rendezvous by providing the link between asynchronous network events and node reactive events. This is also implicit in CSP. The ESTEREL program remains fully deterministic. If there are several active `rendezvous` statements for the same channel, it is the role of the asynchronous layer to choose which one is performed. If one `rendezvous` is chosen, the other ones remain active until completion or preemption. To serialize all rendezvous as in CSP, one simply declares all rendezvous labels to be incompatible; this is not necessary in the general CRP language.

3.2 Implementing CSP

We briefly show how to translate CSP nodes into CRP ones. We restrict ourselves to the communication part of CSP, forbidding pure boolean guards; such guards are usually used for non-deterministic sequential programming, which is definitely not a major issue in our application field. We also ignore distributed termination issues. We assume knowledge of the CSP syntax [13].

First, channels are declared, and labels are assigned to each CSP rendezvous command. All labels are declared incompatible to model CSP rendezvous serialization. The variables used in a CSP node are declared at toplevel in the translated module. The translation of statements is structural and mostly trivial. The `skip` CSP statement is translated into `nothing`. Assignments and expressions are kept unchanged. Boolean tests in guards are translated using `if` ESTEREL statements. Iteration is translated into a `loop` statement. The only other thing that is left is guard selection. We explain the translation on an example and leave its easy formal definition to the reader. Consider the CSP statement

```
[ C1?X →stat1
[] C2!exp →stat2 ]
```

If the respective labels are `L1` and `L2`, the CRP translation is:

```
trap T0 in
  trap T1 in
    trap T2 in
        rendezvous L1 : C1;
        exit T1
      ||
        rendezvous L2 : C2(exp);
        exit T2
    end;
    [[stat2]];
    exit T0
  end;
  X := ?L1;
  [[stat1]]
end
```

where $[\![stat_i]\!]$ is the translation of $stat_i$,

The two communications are requested simultaneously when the `rendezvous` statements are started. Assume `C1` is done first. Rendezvous provokes instantaneous termination of "`rendezvous C1`", instantaneous exit of `T1`, hence abortion of the other pending request "`rendezvous C2`" that immediately sends the `kL2` signal to cancel the other rendezvous, and selection of the appropriate continuation $[\![stat_1]\!]$ after assignment of the received value to `X`. Conversely, if `C1` is done first, then `T2` is exited, `kL1` is sent, and $[\![stat_2]\!]$ starts as expected. The `exit T0` statement is necessary to avoid executing $[\![stat_1]\!]$.

The reader familiar with ESTEREL will recognize similarity of this translation with the expansion of the ESTEREL "`await-case`" derived statement. This statement has the form

```
await
  case S1 do stat1
  case S2 do stat2
end
```

It waits for the first occurrence of `S1` or `S2` to start $stat_1$ or $stat_2$. One can use a similar concrete syntax for guard selection in CRP:

```
rendezvous
  case L1 : C1 do stat1
  case L2 : C2(exp) do stat2
end
```

Notice that there is no need for a variable to refer to the value received on `L1` since this value is simply '`?C1`'.

3.3 General CRP

CSP processes were translated into CRP modules that are interfaced only through incompatible channels and have no proper ESTEREL signals. In general CRP, signals and channels can be freely mixed with the only restriction that no more than one rendezvous can be performed at a time on a given channel. In particular, several concurrent rendezvous on distinct channels can be performed simultaneously.

Watchdogs can be imposed on asynchronous communication, which cannot be done properly in the usual CSP-like languages. For instance, if `S` is an input reactive signal of a node M_i (say "second" or "cancel key"), we can require a rendezvous request of M_i to be satisfied before `S` occurs, or else to be aborted. This is written

```
do
  rendezvous C
watching S
```

If `S` and the rendezvous return signal are simultaneous for the node, the rendezvous is considered to be completed by both parties, but its local effect is canceled by the `watching` statement. To avoid such tricky situations, it is often convenient to serialize rendezvous and reactive events, writing the input relation

```
relation S # L;
```

3.4 Example

To illustrate the CRP programming style, and in particular the respective use of local reactivity and asynchronous rendezvous, we present the example of a (simple-minded) banker teller machine in Figure 1. The machine is a CRP node assumed to be connected to another node, the bank. The machine reads a card, checks the code locally, and communicate by rendezvous with the bank to perform the actual transaction. During the whole process, the user can cancel the transaction by pressing a `Cancel` key.

The reactive interface is self-explanatory. The rendezvous interface consists of three valued channels. The output channels `CardToBank` and `AmountToBank` are used to send the card information and transaction amount. The input channel `Authorization` is used to receive a boolean telling whether the transaction is authorized. No rendezvous label is necessary here since there is exactly one `rendezvous` statement per channel.

The body repeatedly waits for a card and performs the transaction. The client-code checking section is purely local and reactive. When the valid code is read, one does two things in parallel: waiting locally for the amount typed by the client, and sending the card information to the bank using a rendezvous. When both these independent operations are completed, one sends the amount to the bank and waits for the authorization. If the return boolean is true, one instantaneously delivers the money and returns the card *exactly* when the authorization rendezvous is completed, to ensure transaction atomicity. During the full transaction, the user can press the `Cancel` key, in which case the transaction is gracefully abandoned, including the various rendezvous, and the card is returned. Since the bank considers the transaction to be performed when the `Authorization` rendezvous is completed, one must declare an exclusion relation between `Authorization` and `Cancel` to prevent these two events to happen simultaneously. The outer trap handles the abnormal cases of the transaction: the card is kept if the code is typed in incorrectly three times or if the transaction is not authorized. We use a full ESTEREL extension of the `trap` statement, in which one can write a trap handler to be activated when the trap is exited. This extension is easily derivable from kernel statements.

4 Semantics

The mathematical semantics is given in two steps. First, the semantics of each node is individually defined using the classical ESTEREL semantics. Then, cooperation between nodes is defined as in the standard CSP ready trace semantics.

4.1 Node Semantics

There are several formal semantics for ESTEREL [5]. The one we consider here is the *behavioral semantics*, extended to handle `exec` statements. It defines the reaction of a module M to an input event I satisfying the input relations as a transition $M \xrightarrow[I]{O} M'$ where O is the generated output event and M' is a new module whose body will perform the further reaction. The behavioral semantics rules are given in appendix.

An *history* is a sequence $I_0 \cdot O_0, \ldots, I_n \cdot O_n, \ldots$ of input-output event pairs where all input events I_i satisfy the declared exclusion relations and where there is no unexpected return signal: if $\texttt{L} \in I_n$, then there exists $i < n$ such that $\texttt{sL} \in O_i$ and $\texttt{kL} \notin O_j$ for $i < j < n$.

4.2 Cooperation Semantics

To define how nodes cooperate, we use the well-known CSP ready trace technique, with the slight complication that several concurrent communications can take place simultaneously at a node. We need to work with sets of channels rather than just channel names.

Given the set C of channels declared in a node, the projection I/C of an input event I on C is defined as the set of channels in C having a return signal in I (notice that there is at most one return signal for a given channel in I because of the implicit label exclusions).

```
module BankerTeller :

type CardInfo; % abstract type of card information
function CheckCode (CardInfo, integer) : boolean;

input Card : CardInfo;
input Code : integer, Amount : integer;
input Cancel;  % user cancel key

output GiveCode, EnterAmount;
output DeliverMoney : integer;
output KeepCard, ReturnCard;

output channel CardToBank : Cardinfo;
output channel AmountToBank : integer;
input channel Authorization : boolean;

relation Cancel # Authorization;

loop
   trap KeepCard in
      await Card;
      % read and check code, at most three times
      do % watching Cancel
         trap CodeOk in
            repeat 3 times
               emit GiveCode;
               await Code;
               if CorrectCode(?Card, ?Code) then exit CodeOk end
            end repeat;
            exit KeepCard % failed 3 times !
         end trap;

         [
            rendezvous CardToBank(?Card);    % send card to bank
         ||
            emit EnterAmount;                % local dialogue
            await Amount;
         ];
         rendezvous AmountToBank(?Amount);   % send amount to bank
         rendezvous Authorization;           % receive authorization boolean
         if ?Authorization then
            emit DeliverMoney (?Amount)
         else
            exit KeepCard
         end if

      watching Cancel;   % user explicit cancel at any time
      emit ReturnCard

   handle KeepCard do
      emit KeepCard
   end trap
end loop
```

Figure 1: The Banker Teller program

The projection on C of an history $I_0 \cdot O_0, \ldots, I_n \cdot O_n, \ldots$ is the sequence on nonempty I_i/C that represents the communication history over C; formally, using '$\bullet$' to add an element at the head of a list:

$$\begin{array}{rcl}(I \cdot O \bullet H)/C & = & I/C \bullet H/C \ \text{ if} I/C \neq \emptyset \\ & = & H/C \ \text{ otherwise}\end{array}$$

An *execution* of a set of nodes $\{M_i \mid i \in I\}$ is a set of valid ESTEREL histories H_i, one for each node M_i, satisfying the *consistency condition* that expresses that events must match for each channel between any two nodes. Formally, for any distinct i and j, if C_{ij} is the set of all common channels between M_i and M_j, then one must have $H_i/C_{ij} = H_j/C_{ij}$.

Value passing can be handled just as in CSP (not detailed here). Ready sets can also be extracted from the histories to detect deadlocks as in CSP.

CRP obviously extends ESTEREL since nodes can be arbitrary ESTEREL programs. That CRP extends CSP results from the following theorem:

Theorem 1 *Let P be a* CSP *program and P' be its translation into* CRP. *Then the* CSP *ready trace semantics of P coincides with the* CRP *semantics of P'.*

The proof will be given in an extended version of this paper.

5 Translation into the Meije Process Calculus

The cooperation semantics is not really constructive, in the sense that it does not tell how to execute programs. We now give an implementation of PURE CRP into Boudol's process calculus MEIJE [6,8]. We choose this calculus because it is able to handle together synchrony and asynchrony, which is not possible in less powerful calculi such as CCS. In addition to an implementation of CRP in a classical process calculus, the MEIJE translation provides us with an automatic program verification environment since MEIJE is accepted as input by verification systems such as AUTO [7,16]. For full CRP programs dealing with values, an approximate translation into MEIJE is feasible by ignoring value handling and only retaining the synchronization skeleton. This can still be useful for proving synchronization properties.

In the sequel, we assume that the reader is familiar with the definition of automata in process calculi using tail-recursive processes.

MEIJE actions consist of the free commutative group over a set of elementary signals `s`. Its elements are products of positive or negative elementary actions `s`! and `s`? with `s`! $\cdot$ `s`? = 1 (these notations point out the ESTEREL reactive input / output directionality better than the more usual s and $\overline{s}$). Prefixing in MEIJE is

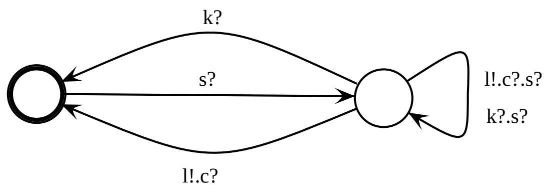

Figure 2: The label automaton L

written ':', the symbol '$\cdot$' being reserved for instantaneous action product. The only other things one should know about MEIJE is the behavior of the parallel and restriction operators. If $p \xrightarrow{a} q$ and $q \xrightarrow{b} q'$, then $p//q \xrightarrow{a} p'//q$, $p//q \xrightarrow{b} p//q'$, and $p//q \xrightarrow{a \cdot b} p'//q'$. If $p \xrightarrow{a} p'$, then $p \setminus s \xrightarrow{a} p' \setminus s$ provided that neither `s!` nor `s?` appear in (the reduced form of) a.

We start by translating each node M_i. Since ESTEREL programs are finite-state, the ESTEREL code at M_i can be readily translated into a MEIJE automaton A_i. This is actually done by the standard ESTEREL compiler. All exclusion relations are taken care of in the following way: only input events that satisfy the relations appear in the automaton. Since all rendezvous return signals for a given channel are exclusive, any MEIJE action appearing in A_i contains at most one return signal for each channel, even if there may be several simultaneously active rendezvous on this channel in the source code.

To compute the ready sets at M_i, we use an auxiliary automaton for each rendezvous label `L` of a channel `C`. Assume `C` links the considered node M_i with another node M_j, and call L_i the automaton defined as a copy of the automaton L of Figure 2 where the following renamings are performed:

$$\begin{array}{rcl}L_i & = & L[\texttt{sL/s}, \texttt{kL/k}, \texttt{L/l}, \texttt{C!/c}] \ \text{ if } i < j \\ L_i & = & L[\texttt{sL/s}, \texttt{kL/k}, \texttt{L/l}, \texttt{C?/c}] \ \text{ if } i > j\end{array}$$

To finish the node translation, we put all the L_i automata of channel labels used by M_i in parallel with the A_i automaton and hide all the `sL`, `kL`, and `L` signals. Let $[\![M_i]\!]$ denote the translation of the node M_i. The sort of $[\![M_i]\!]$ contains the ESTEREL reactive input / output signals of M_i and the channels. If a channel `C` links M_i and M_j, $i < j$, then the MEIJE signal `C` appears positively in $[\![M_i]\!]$ and negatively in $[\![M_j]\!]$.

To translate the full CRP network we put all the CRP nodes translations in parallel and hide all the channels. In the final translation, the nodes evolve asynchronously of each other except on rendezvous where they share an instant, in the sense that perform a single compound MEIJE transition.

As an example, consider a rendezvous between M_1 and M_2 on a channel `C`, with labels `L1` for M_1 and `L2` for M_2. The node automaton A_1 performs a compound action of the form $a_1 \cdot \texttt{L1?}$, where a_1 may itself be some compound action of M_1 involving its reactive signals or other rendezvous; the label automaton L_1 performs the action $\texttt{L1!} \cdot \texttt{C!}$; hence the node automaton $[\![M_1]\!]$ must perform $a_1 \cdot \texttt{C!} = a_1 \cdot \texttt{L1?} \cdot \texttt{L1!} \cdot \texttt{C!}$ since `L1` is hidden. Symmetrically, at node M_2, A_2 performs an action $a_2 \cdot \texttt{L2?}$, L_2 performs $\texttt{L2!} \cdot \texttt{C!}$, and $[\![M_2]\!]$ performs $a_2 \cdot \texttt{C?}$. Since `C` is hidden in the global network, M_1 and M_2 must perform their actions synchronously and the resulting action is the synchronous product $a_1 \cdot a_2 = a_1 \cdot \texttt{C!} \cdot a_2 \cdot \texttt{C?}$ of the local actions of M_1 and M_2.

Of course, such a rendezvous can occur only when both M_1 and M_2 are ready for it. This is just what is computed by the auxiliary label automata. In general CRP, several rendezvous can happen at the same time in the network, and even between the same two nodes on different channels; this is correctly modeled in MEIJE by instantaneous action products.

That the MEIJE translation implements correctly CRP according to the semantics of Section 4 will be precisely stated and proved in an extended version of this paper.

6 Implementation

Using the ESTEREL compiler, each CRP node can be independently compiled into a deterministic target code, written for example in C, that exactly realizes the reactive behavior. In the current ESTEREL compiling process, the target code has the form of a tabulated deterministic automaton where ESTEREL concurrency is compiled away[1], see [5].

Once the nodes are compiled, we are simply left with a set of sequential programs communicating by rendezvous, and we can use all the known implementation techniques for such classical objects: schedulers, network implementation, etc. Compared to CSP or OCCAM, there are two complications. First, the asynchronous layer at each node must take care of the declared exclusion relations by appropriate serialization of events. Second, a rendezvous request can be locally canceled by both parties: we need more elaborate rendezvous protocols. We developed a protocol that uses a pair of asynchronous fifo queues for each channel. It is itself written itself in ESTEREL and we proved its correctness using AUTO. Its description is outside the scope of this paper.

[1] Translating programs into automata has the advantage of excellent execution speed; however, there is a risk of size explosion. New compilation techniques based on the translation of ESTEREL into circuits [4,3] produce slightly slower run-time code but avoid size explosion.

From the software engineering point of view, the CRP system under development will realize a fully automatic implementation of CRP under Unix. In addition to the reactive nodes, the user will provide the system with a network configuration description in which he will describe where the nodes will be placed. Channels will be realized using sockets and the aforementioned protocol.

7 Conclusion

The CRP framework unifies ESTEREL and CSP. It should bring a new way of handling complex parallel systems, using synchronous and asynchronous approaches where they are most appropriate. We strongly believe that the CRP paradigm will prove useful in application domains such as process control, communication protocols, or robotics, but only actual real-size experimentation will confirm that view. We are currently in the process of implementing CRP on top of the existing ESTEREL system.

Since the semantical aspects of synchrony and asynchrony are kept independent, other asynchronous communication policies between synchronous nodes could be studied in the same way. They could also be translated into MEIJE since this calculus is universal among process calculi [8].

The CRP paradigm relies on a careful *separation* between the synchronous and asynchronous layers. Deeper unifications of synchrony and asynchrony should be investigated. For the time being, we have no idea of which programming concepts could be appropriate for that purpose.

References

[1] *The Programming Language ADA Reference Manual.* ANSI / MIL-STD-1815A, also Lecture Notes in Computer Science 155, Springer Verlag, 1983.

[2] G. Berry. Real-time programming: General purpose or special-purpose languages. In G. Ritter, editor, *Information Processing 89*, pages 11–17. Elsevier Science Publishers B.V. (North Holland), 1989.

[3] G. Berry. Esterel on hardware. *Philosophical Transaction Royal Society of London A*, 339:87–104, 1992.

[4] G. Berry. A hardware implementation of pure Esterel. *Sadhana, Academy Proceedings in Engineering Sciences, Indian Academy of Sciences*, 17(1):95–130, 1992.

[5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. Rapport de Recherche 842,

INRIA, 1988. To appear in Science of Computer Programming.

[6] G. Boudol. Notes on algebraic calculi of processes. In K. Apt, editor, *Logic and Models of Concurrent Systems*. NATO ASI Series F13, 1985.

[7] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process calculi, from theory to practice: Verification tools. In *Automatic Verification Methods for Finite State Systems, LNCS 407*, pages 1–10. Springer-Verlag, 1990.

[8] R. de Simone. Higher-level synchronising devices in Meije-SCCS. *Theoretical Computer Science*, 37:347–360, 1985.

[9] B. Espiau and E. Coste-Manière. A synchronous approach for control sequencing in robotics applications. In *Proc. IEEE International Workshop on Intelligent Motion, Istambul*, pages 503–508, 1990.

[10] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with Signal. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[11] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous dataflow programming language Lustre. *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, Sept. 1991.

[12] D. Harel. Statecharts: a visual approach to complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[13] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[14] G. Jones and M. Goldsmith. *Programming in Occam 2*. C.A.R. Hoare Series in Computer Science. Prentice Hall International.

[15] J-P. Paris. Exécution de tâches asynchrones depuis Esterel. Thèse d'informatique, Université de Nice, 1992.

[16] V. Roy and R. de Simone. Auto and Autograph. In R. Kurshan, editor, *proceedings of Workshop on Computer Aided Verification*, New-Brunswick, June 1990.

Appendix

The Behavioral Semantics of Esterel

The *behavioral semantics* of ESTEREL is the reference semantical definition of the language [5,4]. We present the extension needed to handle the `exec` statement (or equivalently the `rendezvous` statement).

Given a module M and an input event I, the behavioral semantics determines a transition $M \xrightarrow[I]{O} M'$ where O is the generated output event and where the *derivative* M' is another module suited to perform further reactions. The derivative M' has the same interface as M and differs by its body. The reaction to a sequence of input events is computed by chaining such elementary transitions.

Inductive Rules

The relation $M \xrightarrow[I]{O} M'$ is defined using an auxiliary inductive relation $stat \xrightarrow[E]{E', L, k} stat'$ on statements, where E is the current event to which *stat* reacts, E' is the event emitted by *stat* in response to E, L is the set of labels of `exec` statements currently active in *stat*, and k is a termination level explained below. The start and kill signals of `exec`'s appear in E', the return signals appear in E. Since signals are broadcast, *stat* receives the signals it emits and E' will always be contained in E.

The relation between both transitions systems is as follows: one has $M \xrightarrow[I]{O} M'$ if and only if $stat \xrightarrow[I \cup O]{O, L, k} stat'$ where *stat* and *stat'* are the bodies of M and M'; we assume the harmless restriction that *stat* cannot internally emit input signals.

The integer *termination level* k determines how control is handled. In each reaction, any statement can behave in three ways: it can terminate and release the control, it can retain the control and wait for further events, or it can exit a trap. We set $k = 0$ for proper termination, $k = 1$ for waiting, and $k = l + 2$ for an exit `T`, where l is the number of traps declarations one must traverse from *stat* to reach the declaration of `T`. In the following example, the first "`exit T`" and the "`exit U`" statements has level 2 since they concern the closest `trap` statement, while the second "`exit T`" has level 3 since one must traverse the declaration of `U` to reach that of `T`:

```
trap T in
    exit T²
  ||
    trap U in
        exit U²
      ||
        exit T³
    end
end
```

The exit levels can be determined statically; we assume that all `exit` statements are labeled by their level. With this coding, the synchronization performed by a '`||`' statement amounts to compute the *max* of the levels returned by its branches: a parallel terminates only when all branches have terminated, an exit preempts waiting, and only the outermost exit matters if several exits are done concurrently.

With respect to the set L of active `exec` labels, we use the notation

$$\begin{aligned} E\#L &= E \cup \{\mathtt{kL} \mid \mathtt{L} \in L \text{ and } \mathtt{sL} \notin E\} \\ &\quad - \{\mathtt{sL} \mid \mathtt{L} \in L \text{ and } \mathtt{sL} \in E\} \end{aligned}$$

Then $E\#L$ differs from E by the facts that `exec`'s started before current instant E are killed and that `exec`'s started at the current instant are simply ignored if killed right away.

Comments on the Rules

Rules *(nothing)*, *(halt)*, *(emit)*, and *(exit)* are obvious. Rule *(execstart)* is used to start an `exec` statement. The label is put in the set of started `exec`'s and the statement is rewritten into an auxiliary `execwait` statement that does not exist in the language proper but is convenient in defining the semantics. Rule *(execwait1)* expresses `exec` termination upon task return. In rule *(execwait2)*, the exec label is put in L in case of non-termination to correctly handle `exec` preemption by enclosing statements. Rules *(seq1)* and *(seq2)* handle sequencing. In *(seq2)*, notice that both statements receive the *same* current event E because of broadcasting, and that there is a *single* result transition with a merge of E'_1 and E'_2 to model instantaneous control transmission and broadcast. Rule *(loop)* unfolds instantaneously a loop. In rule *(parallel)*, both branches evolve in the same current event E, the sets E' and L generated by both branches are merged because of perfect synchrony, and the termination level k is $max(k_1, k_2)$ as explained before. In rule *(watching)*, a `watching` statement is rewritten using a `present` statement that will behave as the required guard at future instants. To remember which `exec`'s should be killed if preemption occurs because of the presence of the signal `S` at some future instant, we decorate the generated `present` statement with the set L. This decoration is used in rule *(present1)* to generate the appropriate kill signals when the `then` branch is taken, i.e. at preemption time. Of course, a source `present` statement has an empty annotation. Rule *(trap1)* expresses that a trap terminates if its body terminates ($k = 0$) or exits the trap ($k = 2$). Rule *(trap2)* expresses that a trap has no effect if its body waits ($k = 1, k' = 1$) and that an exit of an enclosing trap must be propagated by subtracting 1 to k ($k > 2, k' = k - 1$)). Rules *(signal1)* and *(signal2)* express the coherence requirement on local signals: either a local signal is both received and emitted as in *(signal1)*, or it is not received and not emitted as in *(signal2)*; notice how lexical scoping is properly handled.

Remarks

The resulting statement *stat'* is unused and therefore immaterial for any rule returning $k > 1$; it is discarded by the exited `trap`. If a rule returns $k = 0$, then it also returns $L = \emptyset$ and its resulting term is worth `nothing`.

Because of the intrinsic circular character of the local signal rules that wind up E and E', our set of rules does not yield a straightforward algorithm to compute a transition, unlike in classical structural operational semantics. Given any input I one must *guess* the right current event E and use the rules to check that there is a correct transition. Moreover, solutions are not always unique. In the statement

```
signal S in
  present S then emit S end
end
```

one may consistently consider as present or absent, while in the statement

```
signal S in
  present S else emit S end
end
```

There is no way to consider consistently `S` as present or absent. We require a *correct* program to have a unique semantics. Correctness is studied in details in [5], where other more constructive but more intricate semantics are presented. The introduction of `exec` adds no particular complication with respect to correctness.

$$\texttt{nothing} \xrightarrow[E]{\emptyset,\emptyset,0} \texttt{nothing} \qquad (nothing)$$

$$\texttt{halt} \xrightarrow[E]{\emptyset,\emptyset,1} \texttt{halt} \qquad (halt)$$

$$\texttt{emit S} \xrightarrow[E]{\{\texttt{S}\},\emptyset,0} \texttt{nothing} \qquad (emit)$$

$$\texttt{exec L : P} \xrightarrow[E]{\{\texttt{sL}\},\{\texttt{L}\},1} \texttt{execwait L : P} \qquad (execstart)$$

$$\frac{\texttt{L} \in E}{\texttt{execwait L : P} \xrightarrow[E]{\emptyset,\emptyset,0} \texttt{nothing}} \qquad (execwait1)$$

$$\frac{\texttt{L} \notin E}{\texttt{execwait L: P} \xrightarrow[E]{\emptyset,\{\texttt{L}\},1} \texttt{execwait L: P}} \qquad (execwait2)$$

$$\frac{stat_1 \xrightarrow[E]{E_1',\emptyset,0} stat_1' \quad stat_2 \xrightarrow[E]{E_2',L_2,k_2} stat_2'}{stat_1\texttt{;}\ stat_2 \xrightarrow[E]{E_1' \cup E_2', L_2, k_2} stat_2'} \qquad (seq1)$$

$$\frac{stat_1 \xrightarrow[E]{E_1',L_1,k_1} stat_1' \quad k > 0}{stat_1\texttt{;}\ stat_2 \xrightarrow[E]{E_1',L_1,k_1} stat_1'\texttt{;}\ stat_2} \qquad (seq2)$$

$$\frac{stat \xrightarrow[E]{E',L,k} stat' \quad k > 0}{\texttt{loop}\ stat\ \texttt{end} \xrightarrow[E]{E',L,k} stat'\texttt{;}\ \texttt{loop}\ stat\ \texttt{end}} \qquad (loop)$$

$$\frac{stat_1 \xrightarrow[E]{E_1',L_1,k_1} stat_1' \quad stat_2 \xrightarrow[E]{E_2',L_2,k_2} stat_2'}{stat_1\ \texttt{||}\ stat_2 \xrightarrow[E]{E_1' \cup E_2', L_1 \cup L_2, max(k_1,k_2)} stat_1'\ \texttt{||}\ stat_2'} \qquad (parallel)$$

Figure 3: ESTEREL semantic rules

$$\frac{stat \xrightarrow[E]{E', L, k} stat'}{\texttt{do } stat \texttt{ watching S} \xrightarrow[E]{E', L, k} \texttt{present}_L \texttt{ S else do } stat' \texttt{ watching S}} \quad (watching)$$

$$\frac{\texttt{S} \in E \quad stat_1 \xrightarrow[E]{E'_1, L_1, k_1} stat'_1}{\texttt{present}_L \texttt{ S then } stat_1 \texttt{ else } stat_2 \texttt{ end} \xrightarrow[E]{E'_1 \# L, L_1, k_1} stat'_1} \quad (present1)$$

$$\frac{\texttt{S} \notin E \quad stat_2 \xrightarrow[E]{E'_2, L_2, k_2} stat'_2}{\texttt{present}_L \texttt{ S then } stat_1 \texttt{ else } stat_2 \texttt{ end} \xrightarrow[E]{E'_2, L_2, k_2} stat'_2} \quad (present2)$$

$$\frac{stat \xrightarrow[E]{E', L, k} stat' \quad k = 0 \text{ or } k = 2}{\texttt{trap T in } stat \texttt{ end} \xrightarrow[E]{E' \# L, \emptyset, 0} \texttt{nothing}} \quad (trap1)$$

$$\frac{stat \xrightarrow[E]{E', L, k} stat' \quad (k = 1 \text{ and } k' = 1) \text{ or } (k > 2 \text{ and } k' = k - 1)}{\texttt{trap T in } stat \texttt{ end} \xrightarrow[E]{E', L, k'} \texttt{trap T in } stat' \texttt{ end}} \quad (trap2)$$

$$\texttt{exit T}^k \xrightarrow[E]{\emptyset, \emptyset, k} \texttt{halt} \quad (exit)$$

$$\frac{stat \xrightarrow[E \cup \{\texttt{S}\}]{E' \cup \{\texttt{S}\}, L, k} stat' \quad \texttt{S} \notin E'}{\texttt{signal S in } stat \texttt{ end} \xrightarrow[E]{E', L, k} \texttt{signal S in } stat' \texttt{ end}} \quad (signal1)$$

$$\frac{stat \xrightarrow[E - \{\texttt{S}\}]{E', L, k} stat' \quad \texttt{S} \notin E'}{\texttt{signal S in } stat \texttt{ end} \xrightarrow[E]{E', L, k} \texttt{signal S in } stat' \texttt{ end}} \quad (signal2)$$

Figure 4: ESTEREL semantic rules (continued)