Communication in eCSP*

Quentin Miller and Bernard Sufrin

Oxford University Computing Laboratory,
Wolfson Building, Parks Road, OXFORD OX1 3QD, U.K.
{Quentin.Miller,Bernard.Sufrin}@comlab.ox.ac.uk

Abstract. We introduce eCSP — a new experimental programming language which is being developed for specifying and implementing intelligent telephone systems. The langauge is based on the concurrent processes and synchronous channel communication of Occam, but this is extended with features such as ports, shared channels, dynamic process creation, and a richer type system. Channels, including those carrying values of free data types, allow communication between eCSP processes and Java threads. An example is given to illustrate how some of these features can be used to simplify the implementation of a telephone call chain.

1 Introduction

Modern telephone switching systems offer a rapidly growing range of intelligent telecommunication services — messaging, personal phone numbers, number translation, etc. These are often implemented in terms of interacting threads, each managing large finite-state machines. Such models can present a clear and simple view of a straightforward system; but as the sophistication of services increases these programs can quickly become unwieldy, despite the presence of elaborate graphical and other programming tools.

This field of application has motivated our development of a new language: Eclectic CSP (eCSP). It draws from a number of sources features that we believe are essential for managing the complexity of advanced switching services. Principal influences have been OCCAM [4], CSP [1], and Z [2].

The result is a very general-purpose concurrent programming language, but one that is particularly suited to process control. In this paper we introduce some key features of eCSP, and we use the example of a telephone call chain to show how they can be applied in this field. We also present a simple and high-level mechanism for using eCSP processes alongside Java threads. The resulting combination — eCSP's powerful process, communication, and datatype facilities; with Java's large range of well-supported libraries — should be very attractive for programming large and sophisticated systems in the telecommunications field and elsewhere.

* This work is supported by Marconi Communications Ltd.

2 The Language

The basis of eCSP is the concurrency and communication constructs of Occam. If P_1 and P_2 are processes, then

$$P_1 \| P_2$$

is their parallel composition — that is, their component statements are interleaved according to the needs of their communication, or they may be executed concurrently if the architecture supports this. Processes communicate with one another by writing values to ports, using the notation

$$\mathit{port}!\mathit{value}$$

and reading values from ports, using

$$\mathit{port}?\mathit{name} \rightarrow \mathit{process}$$

in which the value received is bound to *name*, which may then be used in the the process that follows the arrow.

A channel *ch* of type *t* provides an input port for reading (whose type is denoted $?t$) connected to an output port for writing (type $!t$). Both ports are named *ch* — the appropriate end is determined by context. (This channel/port distinction does not appear in Occam; it has been added to increase the power of static type-checking.) All channel communication is synchronous; the reader and the writer will both wait until the data has been transferred before continuing. However, input *alternative* may be presented — the following construct will perform whichever input is first ready:

$$\begin{array}{l} \textbf{if}\ \mathit{port}_1?x \ \rightarrow\ P_1 \\ [\!]\ \mathit{port}_2?x \ \rightarrow\ P_2 \\ \vdots \\ [\!]\ \mathit{port}_n?x \ \rightarrow\ P_n \\ \textbf{fi} \end{array}$$

If more than one port has a value waiting to be input, one of them is chosen non-deterministically (unless the alternatives are separated by **else** rather than $[\!]$, in which case those before the **else** are given priority).

Only one reader and one writer may attempt to communicate using a given channel at any time. However, channels may be declared as being *shared* in which case several processes may have access to either end; a process must acquire a lock on such a channel before using its ports. If *ch* is a shared channel, a process may use its ports only in the body of a **server** or **client** construct:

$$\begin{array}{l} \textbf{server}\ \mathit{ch}\ \textbf{in}\ P_1\ \textbf{end} \\ \textbf{client}\ \mathit{ch}\ \textbf{in}\ P_2\ \textbf{end} \end{array}$$

The **server** construct may also appear as an input alternative:

$$\begin{array}{l} \textbf{if}\ c_1?x \ \rightarrow\ \ldots \\ [\!]\ \textbf{server}\ c_2\ \textbf{in}\ \ldots\ \textbf{end} \\ \textbf{fi} \end{array}$$

Here the second alternative may be selected if a **client** claim is made on c_2 and this process is granted access ahead of other **server** claimants on c_2.

In eCSP, as in Occam, $(P_1 \| P_2)$ terminates when both P_1 and P_2 terminate. eCSP also provides dynamic non-terminating processes — the construct

$$\textbf{fork } P$$

causes P to be executed in parallel with the process that invoked it. Neither the original process nor its new parallel partner will wait for the other to terminate (although of course they may be made to do so by signalling their terminations to each other through channels).

As well as basic scalar types such as **int**, **char**, and **bool**, eCSP provides tuples, vectors, sets, and records. Values of non-scalar types are garbage-collected, and may be anonymous (i.e. they need not be bound to declared names).

Records are given as collections of *label*=*value* bindings, e.g.

$$\{|\ \mathit{name} = \text{"}\mathit{Fred}\text{"},\ \mathit{score} = 73,\ \mathit{completed} = \textbf{true}\ |\}$$

Vector and set expressions may be given by *enumeration*,

$\{0, 12, 4\}$	3-element set of type $\{\textbf{int}\}$
$[\textbf{true}, \textbf{false}, \textbf{true}, \textbf{true}]$	4-element vector of type $[\textbf{bool}]$

or by *replication*, as in

$$\{(\textbf{for } x \textbf{ in } S)\ x \times x\}$$

whose value is the set of squares of the elements of set S. Replication may also be used to create a number of parallel processes; the construct

$$\|(\textbf{for } x \textbf{ in } A)\ P(x)$$

yields

$$P(a_0) \parallel P(a_1) \parallel \ \ldots\ \parallel P(a_n)$$

where $a_0, a_1, \ldots, a_n$ are the elements of A.

Sets and vectors may be analysed by *pattern-matching* alternatives. This process assigns **true** or **false** to the variable *empty* depending on whether set s is empty or not.

$$\begin{array}{l} \textbf{if } s \textbf{ is} \\ \quad \{\} \ \rightarrow\ \mathit{empty} := \textbf{true} \\ \textbf{else} \\ \quad s \ \rightarrow\ \mathit{empty} := \textbf{false} \\ \textbf{fi} \end{array}$$

A pattern may be followed by a *filter*: this is a boolean-valued expression that must be true for a match to be made. A filter is introduced by a vertical bar:

$$\textbf{if } t \textbf{ is } (x, y) \mid x \leq y \ \rightarrow\ \ldots$$

The + operator creates the union of sets and the catenation of vectors — here it is used to convert vector v into set s by deconstructing the former and constructing the latter:

$$\begin{array}{l} s := \{\}; \\ \mathbf{do}\ v\ \mathbf{is} \\ \quad v' + [x] \ \rightarrow \ s := s + \{x\}; \quad v := v' \\ \mathbf{od} \end{array}$$

(**do**... **od** is like **if**... **fi** except that it repeats until no patterns match.)

Port input may also use pattern-matching alternatives; this process will block until it is able to read a value that matches the given pattern:

$$\begin{array}{l} \mathbf{if}\ \mathit{chan}_1\ ?\ \{x\} \quad \rightarrow \ \ldots\ x\ \ldots \\ [\!]\ \ \mathit{chan}_1\ ?\ \{\} \qquad \rightarrow \ \ldots \\ [\!]\ \ \mathit{chan}_2\ ?\ \mathbf{true} \rightarrow \ \ldots \\ [\!]\ \ \mathit{chan}_2\ ?\ \mathbf{false} \rightarrow \ \ldots \\ \mathbf{fi} \end{array}$$

(Note that sending a 0- or 1-element set — as on chan_1 — is a useful technique for handling error conditions where no meaningful value can be produced.)

eCSP also provides *free data types*. These can be simple enumerations of constants,

$$\mathbf{data}\ \mathit{Colour} = \mathit{red} + \mathit{green} + \mathit{blue}$$

or collections of constructed values; as in this type *Tree*, whose nodes are leaves containing integer values, or interior nodes containing a pair of subtrees:

$$\mathbf{data}\ \mathit{Tree} = \mathit{Leaf}\ \mathbf{int}\ +\ \mathit{Node}\ (\mathit{Tree}, \mathit{Tree})$$

Values of these types may also be pattern-matched — here is a process that assigns to r the integer at the rightmost leaf in tree t:

$$\begin{array}{l} \mathbf{do}\ t\ \mathbf{is} \\ \quad \mathit{Node}\ (\mathit{lt}, \mathit{rt}) \ \rightarrow \ t := \mathit{rt} \\ \mathbf{od}; \\ \mathbf{if}\ t\ \mathbf{is} \\ \quad \mathit{Leaf}\ n \ \rightarrow \ r := n \\ \mathbf{fi} \end{array}$$

Other important features of eCSP include

- subtyping, which allows existing record and free data types to be safely extended without altering existing programs;
- modules and module interfaces, both of which may be parameterised by types as well as values (thus allowing types and procedures to be polymorphic at the module level).

References for details of these and other aspects of the language can be found under 'Further Information' at the end of this paper.

3 Example: A Telephone Call Chain

Here we show how some of the features of eCSP may be used in a programming problem. A *telephone call chain* is an array of processes connected to a telephone at each end (Fig. 1). Each process is responsible for some aspect of the telephone

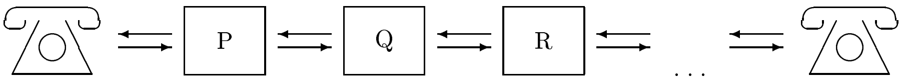

Fig. 1. a telephone call chain

connexion — collecting dialled digits, keeping track of whether one or both phones are still off the hook, etc. Messages are passed along the chain until they are received by a process for which they have some significance.

Two important aspects of this network of processes need to be considered. First, it is bidirectional — messages pass along the chain in both directions. This presents the possibility of *deadlock*; if process P tries to send a message to process Q while Q is trying to send a message to P, both outputs will block forever.

Secondly, it is dynamic, particularly in the presence of advanced services — a process for collecting dialled digits for example is needed until the full number has been dialled, then it may be dispensed with, then possibly brought back to collect dialled responses to questions. Thus when a process in the chain writes to a channel it cannot rely upon a particular process being the reader, as processes may be inserted into and removed from the chain as the call progresses.

To avoid deadlock in bidirectional communication we can make use of the synchronising aspect of shared channels. Recall that a **server** construct may be used in an alternation with port inputs. If the shared channel is used for output, then we know the output will succeed because the receiving process has made a **client** request on the same channel. And if it is used as an alternative with inputs, we know that it cannot block these inputs if they are ready to communicate and it is not.

Any input or **server** alternative may be prefixed with a boolean expression called a *guard*; that alternative will be considered only when the guard's value is **true**. A process may keep its pending outputs in a queue and not attempt to send them until it can do so safely (i.e. in an alternation with the next attempted inputs). Here, q is such a queue and v is the next value to be sent; *in* and *out* are the input and (shared) output ports, and $q \neq [\,]$ is the guard that detects the presence of pending output:

$$
\begin{array}{l}
q := [\,];\\
\textbf{do}\ \mathit{in}?x\ \ \rightarrow\ \ \ldots\ q := q + [v]\ \ldots\\
[\!]\quad q \neq [\,]\ \&\ \textbf{server}\ \mathit{out}\ \textbf{in}\\
\qquad\quad \textbf{if}\ q\ \textbf{is}\ [v]{+}q'\ \ \rightarrow\ \ q{:=}q';\ \ \mathit{out}!v\ \textbf{fi}\\
\quad\ \ \textbf{end}\\
\textbf{od}
\end{array}
$$

The problem of installing and removing processes from the call chain as the call progresses can be solved by the fact that channels can carry values of any type — even port types. Thus, a process may initially be using one port for output; it then receives a new output port (through its normal input channel) from another process, and henceforth it will use the new port for output.

Now we consider the application of these methods to the particular problem. Recall that we could avoid deadlock by using only shared channels. However, a weaker restriction is still safe and leads to a simpler model: a process may output on an unshared channel only if the recipient does not also do so (i.e. the latter's outputs occur only as alternatives with its inputs). Thus we implement each link in the chain with two processes: one to input from left and right, and one to output to left and right (Fig. 2). The former process sends messages to the

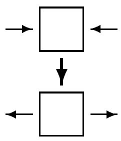

Fig. 2. a single link in the call chain

latter through a shared channel (although we never use its multiple reader/writer capabilities). These links are joined in a chain by criss-crossing their channels (Fig. 3).

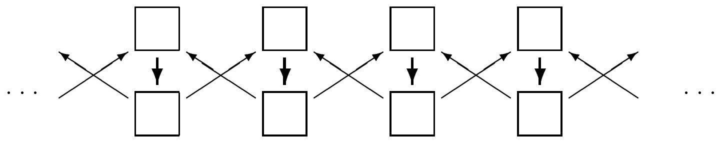

Fig. 3. links joined in a chain

The top component of a link receives messages from its *left* and *right* channels, and passes them to its partner through the shared *out* channel having tagged them (using free data type constructors) to tell the lower link to send them on or to use them itself:

```
done := false;
q := [ ];
do ¬done ∨ q≠[ ] →
  if left?Create l | createHere(l) →
    q := q + [(instructions for changing ports and creating new link l)]
  [] left?Destroy l | destroyHere(l) →
    q := q + [(instructions for changing ports and destroying this link)];
    done := true
  [] left?ChangePort p →
    left := p
⋮
  else left?x →
    q := q + [SendRight x]

  [] right? ...(as with left input)

  [] q≠[ ] & server out in
      if q is [v]+q' → q:=q'; out!v fi
    end
  fi
od
```

The bottom component of the link repeatedly claims **client** access on its input port and reads the message from its top link. It then either acts upon it or — if it has been tagged *SendLeft* or *SendRight* — writes it to its *left* or *right* output port respectively:

```
done := false;
do ¬done →
  client in in
    if in?SendLeft x →
      left!x
    [] in?SendRight x →
      right!x
    ⋮
  end
od
```

Note that what we have seen so far of these processes concerns only the propagation of messages along the chain (whilst avoiding deadlock) and the changing of the chain's structure; there has been no mention of actually processing telephone signals such as dialled digits or on/off hook messages, which is the call chain's purpose. This is intentional: the input and output processes in each link are identical throughout the chain — the functionality that distinguishes them can be relegated to a third process (called the *feature*) which communicates with the output process (Fig. 4).

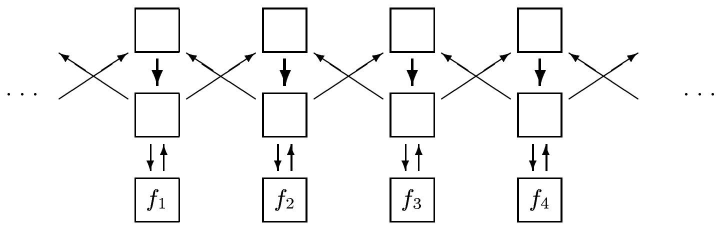

Fig. 4. feature processes ($f_1, f_2, \ldots$) attached to links

Feature processes repeatedly read a message from the bottom link component and respond with a vector of messages to be sent to the left or right. This simple request/response protocol with a single partner cannot introduce deadlock. For example, this feature process collects dialled digits from the left until it has whole telephone number, which it then sends to the right.

```
done := false;
completed := false;
num := [ ];
do ¬done →
  if ¬completed & in?FromLeft (Digit d) →
    num := num + [d];
    if completeNumber (num) →
      out![SendRight (Number num)];
      completed := true
    else true →
      out![ ]
    fi
  else in?Die →
    out![ ]
    done := true
  else in?FromLeft x → out![SendRight x]
  else in?FromRight x → out![SendLeft x]
  fi
od
```

Features inserted into the chain may also communicate with other processes — for example, a feature may offer a shared channel which features from other call chains may connect to in order to create conference calls. Alternatively, a feature may pass its messages through unmodified while sending a copy of each message seen to another process, which could collect accounting data, or present a graphical animation of the activity of the call chain. Such outside processes need not be written in eCSP in order to communicate with eCSP processes through channels — this is discussed next.

4 eCSP and Java

The current implementation of eCSP uses Java as the target language. Within eCSP programs, Java classes may be declared as types, and their methods may be called. Arguments to these methods are automatically converted to closely-corresponding Java types: scalars remain the same, vectors and sets become Vector objects, and records become objects whose variable names correspond to the labels.

eCSP ports may also be passed to Java methods; this allows Java threads to communicate synchronously with eCSP processes. Ports of type $?t$ for scalar type t are passed to the Java method as objects that implement

```
public t read();
```

while those of type $?t$ (again, for scalar type t) are passed as objects implementing

```
public void write( t value );
```

These methods effect blocking communication with the eCSP process at the other end of the channel.

When ports of free data types are passed to Java methods, the objects received at the Java end have read/write methods for each name in the type. For example, suppose we have a free data type that combines integer, floating-point, and rational numbers:

$$\begin{aligned}\textbf{data}\ \mathit{Num} = {} & \mathit{Int}\ \textbf{int} \\ & + \mathit{Float}\ \textbf{real} \\ & + \mathit{Rational}\ (\textbf{int}, \textbf{int})\end{aligned}$$

An output port of type $!\mathit{Num}$ would be seen by a java method as an object with the interface

```
interface Num_io {
  public void Int( int i );
  public void Float( float f );
  public void Rational( int n, int d );
}
```

Calling one of these methods in Java would cause the corresponding eCSP value to be written to the port.

An input port of type $?\mathit{Num}$ would be passed as an object with a method

```
public void read (Num_io handler);
```

Invoking the port's method read(h) would suspend the thread until a *Num* value became available on the channel. If, say, the value *Rational (x,y)* were written at the other end, the Java thread would become active again and the h.Rational (x,y) method would be invoked.

With this means of eCSP/Java interaction, we can embed Java in eCSP programs, or embed eCSP in Java programs, or write programs with varying proportions of the two languages. This is very useful in allowing eCSP programs straightforward access to the extensive GUI and other libraries of Java, and in allowing Java threads a more structured communication model and more abstract types. Hilderink *et al.* [3] have shown how CSP-style communication can be implemented in Java to provide a simpler and more abstract model of processes than Java's threads; eCSP builds on this by giving it a clear syntax.

5 Further Information

A fuller description of Eclectic CSP can be found in
ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Bernard.Sufrin/ecsp.ps

References

[1] C.A.R. Hoare. *Communicating Sequential Processess.* Series in Computer Science. Prentice-Hall International, Hemel Hempstead, U.K., 1985.

[2] A. Diller. *Z: An Introduction to Formal Methods.* John Wiley & Sons, Chichester, U.K., 1990.

[3] G. Hilderink, J. Groenink, W. Vervoort, and A. Bakkers. Communicating Java threads. In *Proceedings of the 20th World Occam and Transputer User Group Technical Meeting*, pages 48–76, Enschede, The Netherlands, Apr. 1997. IOS Press.

[4] G. Jones and M. Goldsmith. *Programming in occam*® *2.* Series in Computer Science. Prentice-Hall International, Hemel Hempstead, U.K., 1988.