Eclectic CSP

Bernard Sufrin and Quentin Miller[*][†]

Revision 3.0
5 October 1999

Abstract

In this note we introduce the main features of Eclectic CSP — an experimental language designed to support the specification and implementation of intelligent network services.

Languages shape our thoughts — they are our fundamental tools for formulating ideas, communicating them and applying them to our advantage. Our languages must evolve to capture higher levels of abstraction as they are discovered and found to be useful, but they must remain as simple and precise as possible lest they generate more problems than they contribute solutions.

[*]Programming Research Group, Oxford University Computing Laboratory, OX1 3QD, UK.
[†]Research Sponsored by Marconi Communications Ltd.

Contents

Acknowledgements

We gratefully acknowledge the financial support of Marconi Communications Ltd., who sponsor the research programme towards which this work is directed.

Our intellectual debt to Tony Hoare and David May will be evident: the conceptual clarity offered by CSP, and the semantic coherence offered by the Occam programming language have been an inspiration to us and provided a secure base from which to explore. The essence of the design of the shared channel and remote procedure call features of eCSP are due to Geoff Barrett, (formerly of INMOS now of SGS-Thomson) with whom the first author was privileged to collaborate on aspects of the design of Occam 3 in 1989/90. Professor Jifeng He has been a patient sounding board for many of the novel features described here.

Warning: The language description herein is subject to change at any time.

1 Introduction

Support for intelligent services (such as messaging, personal phone numbers, and number translation) is an increasingly important part of the functionality offered by the vendors of telecommunications switching equipment. The complexity of the implementation of such services — even those that are functionally rather simple — seems set to increase as network clientele and equipment becomes more sophisticated. For example, as terminal equipment becomes more versatile, service implementations will have to take account, parametrically, of a multiplicity of capabilities. Subscribers will legitimately expect the user interface of a VizziPhone$^{(\mathrm{TM})}$ to be of higher quality than that of a Plain Old Telephone, and service implementations will be expected to negotiate the details of the parameterisation accordingly. After all, why shouldn't I have a Web-based interface to my phone-provider's messaging service if the phone I pick up can handle the necessary protocols?

As the complexity of services increases, the expressive power of current design notations for such services — often based on *ad-hoc* extensions of standard techniques for describing finite-state-machines — is being severely challenged. Although the graphical environments in which services are designed can look very attractive, the rationality of the design process and the reliability and predictability of the resulting services are fundamentally limited by the inadequate expressive power of the underlying notations. In particular it can be hard to predict the load that a particular service will impose on the INS infrastructure, it can be hard to reason about a design in advance of its deployment, and it can be hard to reason about possible interference between features of different services.

These considerations lead us to conjecture that the design and implementation of intelligent networking services will in future require no less than the power of a modern, general purpose, programming notation. Indeed we think it will require rather more. On the one hand it will be necessary to be able to reason about designs before they are implemented — and this suggests that the language be equipped with features that permit precise specification. On the other hand it ought to be possible to take account of the specificities of the setting in which the services are implemented — a network-control station or an intelligent switch — and that suggests certain situation-oriented language features.

In this note we introduce the main features of Eclectic CSP — an experimental specification and programming notation whose semantic features have been borrowed from many sources — including OCCAM [1, 2], CSP [2], Z [3], Polyá, and the Refinement Calculus [4]. Our purpose in devising the notation is speculative: we want to see whether a notation like this will facilitate the description of new and more imaginative services, whether service designers will find it a useful adjunct to (if not a replacement of) the notations they currently use, and whether the ideals of predictability and correctness are attainable with the support of this medium.

We do not expect the final form of the language to have all the features described in this document — but language design is a hard process, and the world of computing is littered with attempts to build application-specific languages by starting with an inadequate base and adding features *ad-hoc*, so we intend to start with a full-featured language, and seeing what can (or must) be removed.

Furthermore, even if the whole language as described here is eventually implemented, we do not necessarily expect all its features to be used in describing networking services. There is a well-established infrastructure within which service implementations presently run, and it would be impractical to propose replacing that infrastructure with one rewritten in eCSP. However, in order to reason about services and their interactions with the existing infrastructure it is necessary to have a model of the infrastructure in eCSP for the purposes of reasoning, and a useful by-product will be the ability to simulate the infrastructure when this is appropriate. In the long run the presence of the computational model may make it easier to reimplement the infrastructure — though this is not really part of our formal research brief, and we are not advocating it.

The presence of specification constructs in the language means that not every program or system described in it will be directly translateable into code. The process of transforming a pre-program (or specification) into a program is called refinement, and the language has features that make certain aspects of this transformation fairly easy.

All eCSP programs can be implemented by translating them into a language that is powerful enough to support concurrent threads — and this is the way we intend to implement the language at first. A compile-time detectable subset of eCSP programs are implementable by translation into a language that supports pseudo-concurrency (coroutines), and this may prove useful in certain contexts.

Note: In the body of this text we use two forms of the language: the *publication form* — which uses sophisticated typography including ordinary mathematical symbols and is intended to be easy on the eye, and the *machine-oriented form* — which is expressed entirely in the ISO-Latin-1 alphabet.

2 The Core Language

2.1 Types

2.1.1 Simple Types and Built-in Type Constructors

The built-in simple types of eCSP are the integers, reals, characters, and booleans — denoted respectively `INT`, `REAL`, `CHAR`, `BOOL`. The unit type `()` has a single element, written `()`.

Note that characters are delimited by single quotes (e.g. `'a'`). Escape sequences for denoting unprintable characters are the same as those used in C.

To assert that an expression E has type T we write $(E{:}T)$.

If $T_1, ..., T_n$ are types then $(T_1, ..., T_n)$ denotes the type of n-tuples, which are written $(E_1, ...E_n)$ (where each expression E_i has type T_i). If $v{:}(T_1, ..., T_n) = (v_1, ...v_n)$, and i is a constant in the range $1..n$, then $v.i$ denotes the i^{th} component of v.

If T is a type, then $[T]$ denotes the type of (finite, origin 0) vectors of T and $\{T\}$ denotes the type of (finite) sets of elements of T. If, in addition, E is a constant integer-valued expression, then $[T]\#E$ denotes the type of (finite, origin 0, size E) vectors of T: this is a subtype of $[T]$. Moreover $\{T\}\#E$ denotes the type of finite sets of elements of T with at most E elements: this is a subtype of $\{T\}$.

If $v{:}[T]$ then $\#v$ denotes its size, and $v(i)$ denotes the element with index i (if $i < \#v$). A variety of additional operations on vectors are provided, including catenation (written $+$).

If $v{:}\{T\}$ then $\#v$ denotes its size. The usual operations on sets are provided, including union (written $+$), intersection (written $*$), and difference (written $-$). The membership test and its negation are is written $\in$ and $\notin$ respectively (or in Latin-1, `MEM` and `NOTMEM`).

Expressions of the following forms denote sets and vectors respectively

$$\begin{array}{ll} \{E_1, ..., E_n\} & \text{(where all } E_i{:}T) \\ [E_1, ..., E_n] & \text{(where all } E_i{:}T) \end{array}$$

Notations for set and vector comprehension take the following forms

$$\begin{array}{l} \{\,(\textbf{for } v_1{:}T_1 \textbf{ in } E_1|B_1;\ \ ...;\ \ \textbf{for } v_n{:}T_n \textbf{ in } E_n|B_n)\ \ E\} \\ [\ (\textbf{for } v_1{:}T_1 \textbf{ in } E_1|B_1;\ \ ...;\ \ \textbf{for } v_n{:}T_n \textbf{ in } E_n|B_n)\ \ E] \end{array}$$

where the E are expressions representing each element, the v_i are variable names, the T_i are types, the E_i are expressions of type $\{T_i\}$ or $[T_i]$, and the B_i are boolean expressions which may be omitted if identically **true**.

Operator	Latin-1 Variant	Explanation
dom	`DOM`	Domain
ran	`RAN`	Range
$\oplus$	`++`	Relational overriding
$\cup$	`+`	Relational union
$\triangleright$	`|>`	Range Restriction
$\triangleleft$	`<|`	Domain Restriction

Table 1: Relational Operators

The (**for**...) construct in a comprehension is called a *generator*. If one of the binary operators $+$, $\times$, $\vee$, or $\wedge$ appears in parentheses in front of a generator and an expression, this operator will be used to combine the generated values into a single value. For example, if v is a vector of integers,

$$(+)\ (\mathbf{for}\ n\ \ \mathbf{in}\ \ v)\ (n \times n)$$

yields the sum of the squares of the values in v; and if s is a set of boolean values,

$$(\vee)\ (\mathbf{for}\ b\ \ \mathbf{in}\ \ s)\ b$$

calculates whether at least one of its elements is **true**.

Contiguous vectors of integers have a special notation

$m..n$ means the vector of numbers from m to n inclusive
$m\#n$ means the vector of numbers from m to $m+n-1$ inclusive

The type of (finite) mappings from elements of domain type D to elements of range type R is denoted $D \nrightarrow R$, and the type of (finite) binary relations between these types is denoted $D \leftrightarrow R$. The type $D \nrightarrow R$ is a subtype of $D \leftrightarrow R$, which is a synonym for $\{(D, R)\}$. The usual relational operators are available; they are listed in table 1.

2.1.2 Type Synonyms

The name nm may be given to type T by a declaration of the form

$$\mathbf{type}\ nm = T$$

For example,

```
type Coord   = (INT, INT)
type Polygon = [Coord]
```

2.1.3 Free Types

A free data type named nm with constructors $k_1, ...k_n$ over types $T_1, ..., T_n$ is defined by

$$\textbf{data}\ TypeName = k_1\ T_1\ +\ ...\ +\ k_n\ T_n$$

The constructors $k_1, ...$ are functions used to map values of the given types $T_1, ...$ into the data type $TypeName$ that is being defined.

Recursive types can be specified this way, and the new type name may appear on the right hand side of the definition.

Example 1 Free Type definition: Binary Trees with integer leaves

```
DATA BT = Leaf Int + Branch (BT, BT)
```

A mutually recursive collection of free types $T_a, T_b, ...$ is defined by

$$\begin{array}{l} \textbf{data}\ \ T_a = k_{a_1}\ T_{a_1} + ... \\ \textbf{and}\ \ \ T_b = k_{b_1}\ T_{b_1} + ... \\ ... \end{array}$$

A constructor may be declared without an associated value field, as in the type of lists of integers declared in example 2.

Example 2 Free Type definition: Lists of integers

```
DATA IntList = Nil + Cons (INT, IntList)
```

Example 3 Mutually recursive Free Type Definition

```
DATA BT = Atom ST + Branch (BT, BT)
AND  ST = Empty   + Vec    [BT]
```

Note that free data types may not be anonymous — *i.e.* a type expression of the form

$$k_{a_1}\ T_{a_1}\ + ... + k_{a_n}\ T_{a_n}$$

may be used only in the declaration of a new named type.

2.1.4 Record Types

If $T_1, ...T_n$ are types, and $i_i, ...i_n$ are names, then the type expression

$$\{\!|i_1{:}T_1, ..., i_n{:}T_n|\!\}$$

denotes the type of records with fields named $i_1, ...$ which have types $T_1, ...$ respectively.

If $E = \{|i_1 = E_1, ..., i_n = E_n|\}$ then $E.i_1 = E_1, ..., E.i_n = E_n$.

Example 4 A record type: `Account` and an instance of that type

```
TYPE Account =
{|   balance:   INT,
     overdraft: INT
|}

{|   balance = - 50,  overdraft = 50 |}: Account
```

Example 5 Declaration of a record type: `Bank`

```
TYPE Bank =
{|   account: Person +> Account,
     balance: INT
|}
```

2.1.5 Free Type Extension and Subtyping

It can be convenient to define a free data type by extending an existing free type. If nm is the name of an existing free type, then the free type ext defined by

$$\textbf{data}\ ext = \textbf{data}\ nm + k_1\ T_1 + ... + k_n\ T_n$$

is an *extension* of the type nm which has all the constructors of nm together with the additional constructors $k_1, ...k_n$.

The type nm is said to be a *subtype* of the extended type, and a value of type nm may appear wherever a value of type of ext is required.

Example 6 A free type with two extensions

```
DATA SmallNum =                  Zero + One + Two
DATA MedNum   = DATA SmallNum + Three + Four + Five
DATA Num      = DATA MedNum   + Big INT
```

2.1.6 Record Type Extension and Subtyping

It can be convenient to define a record type by extending an existing record type. If T_1 and T_2 are two record types, with disjoint field names then the type $T_1 + T_2$ is a record type which has all the fields of T_1 and all the fields of T_2.

$T_1 + T_2$ is an extension of T_1 and an extension of T_2. It is a subtype of T_1, and a value of type $T_1 + T_2$ may appear whenever a value of type T_1 is expected. For reasons of implementation efficiency, $T_1 + T_2$ is *not* considered a subtype of T_2, although the gods of symmetry would no doubt be better pleased if it were.

2.1.7 Port Types

If T is a type, then $?T$ is the type of port from which values of type T can be input, and $!T$ is the type of port to which values of type T can be output. A port of type $?T$ and a port of type $!T$ may be joined by a *channel* of type T, through which data may be transferred in one direction. Ports are first-class values, in the sense that they can be passed as parameters, delivered as results, and communicated over channels.

A port of type $?T$ or $!T$ is always implemented by a channel of type T (see 2.2.16). In fact the only reason for distinguishing between ports and channels is that a port embodies an undertaking to use the channel that implements it in only one direction. Both these types are *simple port types*.

If PT is a port type, then the type $[PT]$ is a *port vector type* — the type of vectors of ports which all have the specified port type. Such types are *composite port types*, and their values are implemented by channel vectors (see 2.3.1).

If $PT_1, ..., PT_n$ are port types, then the type $\{\!|i_1{:}PT_1, i_n{:}PT_n|\!\}$ is a *port record type* — the type of records whose fields have the specified port types. Port record types are also composite port types, and their values are implemented by channel records (see 2.3.1).

If $PT_1, ..., PT_n$ are port types, then the type $(PT_1, ..., PT_n)$ is a *port tuple type* — the type of tuples whose components have the specified port types. Port tuple types are also composite port types, and their values are implemented by channel tuples (see 2.3.1).

2.1.8 Complementary Port Types

If a process communicates using a port of type PT, then it is sensible, indeed essential, to run it in parallel with a process that communicates using a port of the *complementary port type*.

If PT is a port type, then its complementary port type is written as $-PT$. Complementary port types are defined as follows:

$-?T$	$=$	$!T$	for any type T
$-!T$	$=$	$?T$	for any type T
$-[PT]$	$=$	$[-PT]$	for any port type PT
$-(PT_1, ..., PT_n)$	$=$	$(-PT_1, ..., -PT_n)$	for any port types $PT_1, ...$
$-\{\!|i_1{:}PT_1, ..., i_n{:}PT_n|\!\}$	$=$	$\{\!|i_1{:}-PT_1, ..., i_n{:}-PT_n|\!\}$	for any port types $PT_1, ...$

2.2 Processes

eCSP programs are built from processes, the simplest of which are called *actions*. An action may be a *skip*, an *exception*, an *assignment*, an *output*, or a *specified action*. It will shortly become clear why an *input* is not simply an action.

2.2.1 Skip

A skip, written **skip**, has no effect on the program. A skip is typically used to convey explicitly a programmer's intention that nothing be done at a particular point in the program. See Example 25 for a typical situation.

2.2.2 Assignment

An assignment takes the form

$$v_1, v_2, ..., v_n := e_1, e_2, ..., e_n$$

where the v_i are variables (or, more generally, expressions denoting locations), and the e_i are expressions. The expressions must be assignable to the variables, and this is so only if the type of each expression e_i is a subtype of the declared type of the corresponding variable v_i.

The list of expressions is evaluated in parallel, and the resulting values are assigned in parallel to the variables.

Example 7 Rotating three variables in a single assignment

```
a,b,c := b,c,a
```

2.2.3 Output

An *output* takes the form

$$p!e$$

where p is an *output port*, and e is an expression. The type of the expression must be a subtype of the type of the port.

The expression is evaluated, and its value transmitted via the port to another process; the action is not complete until that process has input the value.

2.2.4 Bracketed Processes

A process p may be enclosed in brackets **begin** p **end** without changing its meaning.

2.2.5 Composite Processes

The usual forms of sequential process composition are available:

Sequential composition	$a_1;\ a_2;\ ...$
Alternation	**if** $e_1 \to a_1$ **or** ... **or** $e_n \to a_n$ **fi**
Iteration	**do** $e_1 \to a_1$ **or** ... **or** $e_n \to a_n$ **od**

In these forms the e_i are boolean expressions, called guards, and the a_i are processes.

An alternation evaluates each of the guards. If none is true then the program aborts. If any are true, then one of the processes corresponding to a true guard is executed.

An iteration evaluates each of the guards. If none are true then the iteration terminates. If any are true, then one of the processes corresponding to a true guard is executed, and the iteration as a whole is repeated.

If I is the process

$$\textbf{do}\ \ e_1 \to a_1\ \ \textbf{or}\ \ ...\ \ \textbf{or}\ \ e_n \to a_n\ \ \textbf{od}$$

then I is equivalent to

$$\textbf{if}\ \ e_1 \to a_1;\ I\ \ \textbf{or}\ \ ...\ \ \textbf{or}\ \ e_n \to a_n;\ I\ \ \textbf{or}\ \ \neg e_1 \wedge ... \wedge \neg e_n \to \textbf{skip}\ \ \textbf{fi}$$

An alternation or iteration may use "**else**" (everywhere) to separate alternatives instead of "**or**". In this case the guards are evaluated in sequence, and the process corresponding to the first true guard is executed.

Example 8 Two ways to calculate the maximum of `m, n`

```
IF   m>n  -> max := m
ELSE TRUE -> max := n
FI

IF m>=n -> max := m
OR m<=n -> max := n
FI
```

Example 9 Calculate Greatest Common Divisor of `m` and `n`

```
DO m>n -> m:=m-n
OR m<n -> n:=n-m
OD
```

There are also two forms of composition that perform case-analysis by pattern-matching:

Case alternation	**if** e **is** $\pi_1 \to a_1$ **or** ... **or** $\pi_n \to a_n$ **fi**
Case iteration	**do** e **is** $\pi_1 \to a_1$ **or** ... **or** $\pi_n \to a_n$ **od**

In these forms the e is an expression, the π_i are patterns, and the a_i are processes.

A pattern is either a *variable name*[1], a *literal constant*, a *defined constant*, a record or tuple of patterns, a *free data constructor* applied to a pattern, or the wildcard pattern "_". A pattern that contains literal constants or free data constructors is called a *refutable* pattern.

A case alternation evaluates the expression e, and matches the value against the patterns π_i. If any patterns match, then one of them (π_m say) is chosen and the corresponding process a_m is executed. If no pattern is matched then the program aborts. Alternates may be separated by "**else**" rather than "**or**", in which case the patterns are matched in sequence.

If I is the case iteration

$$\mathbf{do}\ e\ \mathbf{is}\ \pi_1 \to a_1\ \ \mathbf{or}\ \ ...\ \ \mathbf{or}\ \ \pi_n \to a_n\ \ \mathbf{od}$$

then I is equivalent to

$$\mathbf{if}\ e\ \mathbf{is}\ \pi_1 \to a_1;\ I\ \ \mathbf{or}\ \ ...\ \ \mathbf{or}\ \ \pi_n \to a_n;\ I\ \ \mathbf{or}\ \ \_ \to \mathbf{skip}\ \mathbf{fi}$$

When a case pattern contains variables then a successful match will result in those variables being bound to parts of the structure (value) being matched. The scope of such a binding is the rest of the pattern and the corresponding process.

The pattern [ ] matches an empty vector, while patterns of the form

$$[\pi_1, ..., \pi_n] + \pi$$

match vectors whose first n elements match $\pi_1, ...\pi_n$, and whose remaining elements match π. Similarly,

$$\pi + [\pi_1, ..., \pi_n]$$

will match vectors whose last n elements match $\pi_1, ...\pi_n$.

The pattern $\{\}$ matches an empty set, while the patterns $\{v_1, ..., v_n\} + vs$ and $vs + \{v_1, ..., v_n\}$ (where the v_i, and vs are variable names) match a set with at least n elements, making an arbitrary choice of elements from the set to bind to the v_i and binding the remaining elements of the set to vs.

[1] Which should not, in this context, be the same as that of any nullary free data constructor or defined constant.

Example 10 Destructively make a set from a vector

```
set := {};
DO vec IS  [hd]+vec' -> set,vec := {hd}+set,vec' OD
```

Example 11 Destructively make a vector from a set

```
vec := [];
DO set IS  {hd}+set' -> vec,set := vec+[hd],set' OD
```

Patterns may be associated with *filter* expressions — boolean-valued expressions which may contain variables bound in the pattern. Identical patterns may guard different branches of a case composition, providing their filters are distinct.[2]

Example 12 Calculate the number of positive and negative elements in a list

```
ps,ns := 0,0;
DO list IS
   Cons(hd, tl) | hd>0 -> list,ps := tl,ps+1
OR Cons(hd, tl) | hd<0 -> list,ns := tl,ns+1
OR Cons(hd, tl) | hd=0 -> list := tl
OD
```

2.2.6 Guarded Case Iteration

It can sometimes be helpful to terminate a case iteration early. To this end, if G is a boolean expression the form

$$\textbf{do}\ G\ \&\ E\ \textbf{is}\ \ \pi_1 \rightarrow a_1\ \ \textbf{or}\ \ \ldots\ \ \textbf{or}\ \ \pi_n \rightarrow a_n\ \ \textbf{od}$$

is defined to be equivalent to the following (where *continue* is a fresh variable which does not appear in G, E, the π_i or the a_i):

$$\begin{array}{l}
\textbf{let}\ \ \textbf{var}\ \mathit{continue} = G\ \ \textbf{in} \\
\quad \textbf{do}\ \mathit{continue} \rightarrow \\
\quad\quad \textbf{if}\ \ e\ \textbf{is} \\
\quad\quad\quad \pi_1 \rightarrow a_1;\ \mathit{continue} := G \\
\quad\quad \textbf{or} \ldots \\
\quad\quad \textbf{or}\ \pi_n \rightarrow a_n;\ \mathit{continue} := G \\
\quad\quad\quad \textbf{else} \qquad\qquad\qquad\qquad _ \rightarrow \mathit{continue} := \textbf{false} \\
\quad\quad \textbf{fi} \\
\quad \textbf{od} \\
\textbf{end}
\end{array}$$

[2] Distinctness is not a statically decideable property, and a simplistic compiler may not be able to warn of overlapping filters.

Example 13 Calculate the number of positive and negative elements in a list

```
ps,ns := 0,0;
DO list IS
   Cons(hd, tl) ->
   IF hd>0 -> list,ps := tl,ps+1
   OR hd<0 -> list,ns := tl,ns+1
   OR hd=0 -> list := tl
   FI
OD
```

2.2.7 Subtype Matching

A case pattern may consist of a variable annotated with a type, in which case the pattern will only match values of that exact type and its subtypes. This is the means by which a *down cast* (*i.e.* a cast from a type to one of its subtypes) is implemented. For example, suppose that the variables `n, m, b` are respectively declared to be of types `Num, MedNum, INT` (see example 6), then the following case alternation is well-typed and never aborts.[3]

```
IF n IS (v:MedNum) -> m:=v
OR      Big i      -> b:=i
FI
```

2.2.8 Specified Actions

A specified action (henceforth *spec*) is a way of describing what one wishes to achieve without explaining how to achieve it. It takes the form:

$$\overline{w}{:}[P, Q]$$

where $\overline{w}$ is a list of variables, and P, Q are predicates over the program state. This action will change the state of the process so that the predicate Q becomes true, providing that the predicate P is true. If P is not true, then the action may do anything at all. The only variables that change are those named in $\overline{w}$. It is often convenient to be able to name the original values of the variables in the frame ($\overline{w}$). For any such variable v, the form ‘v (pronounced “original v”) denotes its original value. There is, therefore, an implicit conjunct in the precondition of each spec, namely $\overline{w} = {‘}\overline{w}$.

[3]The type of the variable `n` is still `Num`, even though its value has type `MedNum`, so replacing the assignment `m:=v` with `m:=n` would result in an ill-typed program.

Example 14 Increase the value of x if it is negative, otherwise do anything.

```
x:[x<0, x>‘x]
```

A spec is not executable, but it may be replaced by a spec that refines it,[4] or by executable code that refines it, without changing the meaning of the context in which it appears.

Example 15 Two processes — each of which refines the spec of Example 14

```
x:=0;
IF x>=0 -> x:=x-4 OR x<0 -> x:= 0 FI
```

The spec $\overline{w}$:$[P, Q]$ is *feasible* if $(P \wedge \overline{w} = ‘\overline{w}) \Rightarrow \exists \overline{w} @ Q$. Infeasible specs are, of course, unimplementable and no refinement of an infeasible spec can be feasible.

Example 16 A feasible spec: choose an element of a nonempty set

x:$[S \neq \emptyset, x \in S]$

Feasible because
$S \neq \emptyset \wedge x = ‘x \Rightarrow \exists x @ x \in S$

2.2.9 Input

An input action takes the form

$$p?\pi \to a$$

where p is an input port, π is a pattern (possibly type-annotated), and a is a process. This action waits until there is input available from the port, then executes the process a providing the input matches the pattern. If the input does not match the pattern, then the program aborts.

Ports are statically typed, and the purpose of a type-annotation in a pattern, if one appears, is to make explicit the type of value carried by the port.

When π is a refutable pattern, the input action $p?\pi \to a$ is equivalent to

$$p?v \to \textbf{if } v \textbf{ is } \pi \to a \textbf{ fi}$$

where v is a variable which appears neither in π nor a. So if a value that doesn't match the pattern is input from p then the program will abort.

[4]In the sense of the Morgan/Back refinement calculus [4].

Example 17 An infeasible spec: choose an element of a set

x:[**true**, $x \in S$]

Infeasible because
$x = {}`x \not\Rightarrow \exists x @ x \in S$

Example 18 Assign the next value input from the integer port `left` to `v`

```
left?(x:INT) -> v:=x
```

2.2.10 Input Alternation

A process may wait for input on more than one port at once. An *input alternation* process takes the form

$$\begin{array}{l} \textbf{if}\ \ e_1\ \&\ p_1?\pi_1 \rightarrow a_1 \\ \textbf{or}\ e_2\ \&\ p_2?\pi_2 \rightarrow a_2 \\ \quad \ldots \\ \textbf{or}\ e_n\ \&\ p_n?\pi_n \rightarrow a_n \\ \textbf{fi} \end{array}$$

where the e_i are boolean expressions, known as guards, (which may be omitted if identically **true**), the p_i are ports (not necessarily distinct), the π_i are patterns, and the a_i are processes. This process first evaluates all the boolean expressions, then waits until input is available on one of the ports corresponding to the true guards; an input from one such port is then matched against the pattern(s) corresponding to the port, and the processes corresponding to those it matches are considered eligible for execution. If one or more processes are eligible, then one of them is chosen for execution.

Example 20 Merge input from two ports

```
DO TRUE ->
   IF left?x  -> output!x
   OR right?x -> output!x
   FI
OD
```

Example 19 Abort unless `TRUE` is read from `ack`

```
ack?TRUE -> SKIP
```

Example 21 Interleave input from two ports

```
l,r := TRUE,TRUE;
DO l \/ r ->
  IF l & left?x  -> l,r := FALSE,TRUE; output!x
  OR r & right?x -> l,r := TRUE,FALSE; output!x
  FI
OD
```

A special port called **timer** has type ?**int**. This port may be read at any point in a program, and it is always ready to input an integer representing the current time in milliseconds. Such a port can be used with the **after** n pattern, which will allow input of any integer representing a time later than n. For example, we may use the following to limit the time that a process waits for input on channel `ichan` before going on to do something else.

Example 22 Wait 20 seconds for an input, then give up.

```
TIMER ? n ->
  IF TIMER ? AFTER n+20000 -> result:=GIVEUP
  OR ichan ? x             -> result:=x
  FI
```

A $p_i?\pi_i$ combination may be replaced by a wildcard ("_"). If this is the case, and the corresponding guard yields true, then the wildcard is treated as if it were a ready input port with a matching pattern and the corresponding process is eligible for execution. The process in Example 23 copies an input item to the output if there is input available when it starts, and *rushing* is false. If *rushing* is true and no input is available, then `NONE` is send to the output; if *rushing* is true and input is available, then one of the outcomes above is chosen.

Example 23 Copy input to output, or send `NONE` to output

```
IF           input?x -> output!x
OR rushing & _       -> output!NONE
FI
```

An input alternation may separate its alternatives with "**else**" rather than "**or**". In this case, if more than one port is (or becomes) ready to communicate, then input is taken from the leftmost ready port. The pattern/filter combinations for that port are then matched against the input in the sequence in which they appear: if none succeeds then nothing more happens, otherwise the process corresponding to the first successful match is executed.[5]

Example 24 Input from `left` in preference to `right`

```
 IF   left?x  -> result:=x
 ELSE right?x -> result:=x
 FI
```

Example 25 Absorb all input until a control signal appears

```
DO absorb ->
 IF   control?_ -> absorb:=FALSE
 ELSE input?x   -> SKIP
 FI
OD
```

Example 26 Count the number of odd and even numbers on a port

```
DO TRUE ->
   IF input?x | odd  x ->  odds :=odds +1
   OR input?x | even x ->  evens:=evens+1
   FI
OD
```

2.2.11 Distributed Alternation

If $G(i)$ is an input guard in which the name i appears, $P(i)$ is a process in which the name i appears, and E is a set or vector containing elements $E_1, ...E_n$ then the form

$$\textbf{or } (\textbf{for } i \ \textbf{ in} E) \ G(i) \to P(i)$$

means

$$G(E_1) \to P(E_1) \ \textbf{ or } \ ... \ \textbf{ or } \ G(E_n) \to P(E_n)$$

when it appears within an input alternation. Similarly,

$$\textbf{else } (\textbf{for } i \ \textbf{ in} E) \ G(i) \to P(i)$$

can be used to mean

$$G(E_1) \to P(E_1) \ \textbf{ else } \ ... \ \textbf{ else } \ G(E_n) \to P(E_n)$$

[5] More discrimination can be exercised here by factoring a combined input/match/filter alternation into a nest of input, then match, then filter alternations.

Example 27 Same as example 26

```
DO TRUE ->
   IF input?x ->  IF odd  x ->  odds :=odds +1
                  OR even x ->  evens:=evens+1
                  FI
   FI
OD
```

Example 28 Same as example 26

```
DO TRUE ->
   IF input?x ->  IF   odd  x ->  odds :=odds +1
                  ELSE TRUE   ->  evens:=evens+1
                  FI
   FI
OD
```

2.2.12 Iterated Alternation

It is frequently desirable to execute an input alternation repeatedly whilst one of its guards is enabled. The form on the left below provides a convenient syntactic sugar for the common idiom on the right.

$$
\begin{array}{ll}
\begin{array}{ll}
\textbf{do} & e_1 \ \&\ p_1?\pi_1 \rightarrow a_1 \\
\textbf{or} & e_2 \ \&\ p_2?\pi_2 \rightarrow a_2 \\
 & \ldots \\
\textbf{or} & e_n \ \&\ p_n?\pi_n \rightarrow a_n \\
\textbf{od} &
\end{array}
&
\begin{array}{l}
\textbf{do}\ e_1 \vee e_2 \vee \ldots \vee e_n \rightarrow \\
\quad \textbf{if}\ \ e_1 \ \&\ p_1?\pi_1 \rightarrow a_1 \\
\quad \textbf{or}\ e_2 \ \&\ p_2?\pi_2 \rightarrow a_2 \\
\qquad \ldots \\
\quad \textbf{or}\ e_n \ \&\ p_n?\pi_n \rightarrow a_n \\
\quad \textbf{fi} \\
\textbf{od}
\end{array}
\end{array}
$$

Example 29 Merge `input(0), ..., input(9)`

```
DO OR (FOR i IN 0#10) input(i)?x -> out!x OD
```

2.2.13 Parallel Composition

The parallel composition of processes $p_1, ..., p_n$ takes the form

$$p_1 \ || \ p_2 \ || \ \ldots \ || \ p_n$$

The component processes are started at the same time, and the composite process terminates when all its components have terminated.

Example 30 Merge `input(0), ..., input(9)` until `control` signals `Stop`

```
DO
  go & control?Stop -> go:=FALSE
OR
  OR (FOR i IN 0#10) go & input(i)?x -> out!x
OD
```

Example 31 Parallel computation of three function values

```
a:=f(a) || b:=f(b) || c:=f(c)
```

A variable (or component of an array) that is assigned to in one of the components of a parallel composition must not appear in any of the other components.

Example 32 An improper use of variables in a parallel composition

```
x := 2;
BEGIN  x:=3 || y:=x END
--  At this point the value of y may be 2 or 3!
```

2.2.14 Distributed Parallel Composition

If $P(i)$ is a process in which the name i appears, and E is a set or vector containing elements $E_1, ...E_n$ then the form

$$||(\mathbf{for}\ i\ \ \mathbf{in}\, E)\ \ P(i)$$

means

$$P(E_1)\ ||\ ...\ ||\ P(E_n)$$

2.2.15 Variable and Constant Declarations

New variables and constants are introduced by declarations at the head of a block that takes the form

$$\mathbf{let}\quad \Delta_1\ \ \Delta_2\ \ ...\ \ \Delta_m\quad \mathbf{in}\quad p\quad \mathbf{end}$$

Where p is a process — which is known as the *body* of the block — and each Δ_i takes one of the forms

$$\begin{array}{ll}\mathbf{var} & \mathit{name}{:}\mathit{Type} = \mathit{Expression}\\ \mathbf{con} & \mathit{name}{:}\mathit{Type} = \mathit{Expression}\\ \mathbf{def} & \mathit{name}{:}\mathit{Type} = \mathit{Expression}\end{array}$$

Example 33 Parallel computation of n then of $2n$ function values

```
|| (FOR i IN 0#n) result(i):=f(i);

  || (FOR i IN 0#n)  result(i):=f(i)
||
  || (FOR i IN n#n)  result(i):=f(i)
```

The expression — whose type must be consistent with the declared type[6] — is evaluated, and in the **con** or **def** form is associated directly with the name. The :*Type* may be omitted, in which case it is taken to be the type of *Expression*. The expression may be omitted in a variable declaration, in which case the type must be given.

In the **var** form the value is stored in a memory location that is associated with the name, and subsequent assignments may change the value stored in that location. The special expression **any** may be used in this form (and this form only) — in which case an unspecified element of the appropriate type is used as the value.

The expression in the **con** form is evaluated at run-time, when the corresponding block of code is executed. Note that the name will *not* be treated as a constant in patterns; for example, in example 34 the pattern-matching on the input will always succeed and n will take the input value in $p(n)$. A name that is to be

Example 34 Run-time constant declaration

```
LET CON n = 0 IN
  ichan?n -> p(n)
END
```

used as a constant pattern must be declared as a compile-time constant, using the **def** form. In example 35, only a 0 may be input and hence $p(n)$ will always be $p(0)$. The scope of the association between the name and value (or location)

Example 35 Compile-time constant declaration

```
LET DEF n = 0 IN
  ichan?n -> p(n)
END
```

that is established by a declaration is the text of subsequent declarations in the block, and the text of the block's body. The usual scope rules for nested blocks apply: an association for a name i established in an outer block b is superseded by an association for i established in a block nested within b.

[6] *i.e.* of the same type or a subtype.

Example 36 Declarations, a legal and an illegal assignment

```
LET
  CON thirtyfour: INT = 34  --  Constant declaration
  VAR forty:      INT = 40  --  Variable declaration
IN
  forty := 34;              --  This assignment is legal
  thirtyfour := 34          --  This assignment is illegal
END
```

2.2.16 Channels

A channel implements point to point communication between two parallel processes by providing an output port for one of the processes and an input port for the other. A unidirectional channel *name* suitable for communicating values of type $Type$ is constructed by the declaration

chan *name*:$Type$

Such a channel is used to connect a process that writes to its !$Type$ port to a process that reads from its ?$Type$ port

The composite parallel process in Example 37 implements a one-place buffer, whose producer client writes to it using `left!data` and whose consumer client reads from it using `right?data`. The `middle` channel implements an output port for the left parallel component and an input port for the right component.

Example 37 A one-place buffer process

```
LET CHAN middle:INT IN
    DO left?x  -> middle!x OD || DO middle?y -> right!y OD
END
```

A *composite channel* may be used to connect two composite ports. Because the constituent ports in a composite might not all communicate in the same direction, the declaration of a composite port must indicate this. The declaration

channels *name*:$PType$

where $PType$ is a (possibly composite) port type produces a channel whose two ends have type $PType$ and $-PType$.

Example 38 shows a simple process farm built by parallel composition. The top process accepts raw data from its `left:?RAW` port if either of the worker processes are free, then it immediately transmits it to a free worker. It will accept cooked data back from a worker it believes is busy and transmit it to the `right:!CKD` port.

Example 38 Simple process farm using ports `left:?RAW, right:!CKD`

```
LET
  VAR free = (TRUE, TRUE)
  CHANNELS to   : (!RAW, !RAW)
  CHANNELS from : (?CKD, ?CKD)
IN
      DO free.1\/free.2 & left?x ->                 -- Farmer
            IF free.1 -> free.1:=FALSE; to.1!x
            OR free.2 -> free.2:=FALSE; to.2!x
            FI
      OR ~free.1 & from.1?y -> free.1:=TRUE; right!y
      OR ~free.2 & from.2?y -> free.2:=TRUE; right!y
      OD
  ||  DO to.1?x -> from.1!cook(x) OD                -- Worker 1
  ||  DO to.2?x -> from.2!cook(x) OD                -- Worker 2
END
```

2.3 Procedures and Process Families

The two worker processes in Example 38 have the same structure, but use different input and output ports. This common structure can be made explicit by defining a named family of processes with the same structure, using a declaration of the form

$$\textbf{proc}\ name(\Phi_1, ..., \Phi_m)\ \textbf{is}\ \ p$$

where p is a process, and each Φ_i is a value parameter declaration of the form *name*:*Type*. The type of the resulting process is written

$$(T_1, ..., T_n) \rightarrow ()$$

where T_i is the type of the i^{th} parameter. (The () indicates that a call to this procedure yeilds no result value.)

An instance of the family is built (and started) using the notation

$$name(E_1, ..., E_m)$$

Each E_i must be a value of the specified type, or one that implements the specified type.[7]

In Example 39 we present a generalisation of the process farm, with the roles of farmer and workers being played by instances (invocations) of process families. Here `Farm`, `farmer`, and `worker` are families of processes.

Both the farmer and the worker processes defined in Example 39 can be used in other settings, and it is worthwhile considering whether the Farm process can itself be restructured so as to support re-use. Happily it can, and the result is shown in Example 40.

[7] Recall that a channel of type T implements ports of type $?T$ and $!T$.

Example 39 The process farm revisited

```
PROC worker(in:?RAW, out:!CKD) IS
  DO in?x -> out!cook(x) OD

PROC farmer(in:?RAW, to:(!RAW,!RAW), from:(?CKD,?CKD), out:!CKD) IS
  LET
    VAR free = (TRUE, TRUE)
  IN
    DO free.1\/free.2 & in?x -> IF free.1 -> free.1:=FALSE; to.1!x
                                OR free.2 -> free.2:=FALSE; to.2!x
                                FI
    OR ~free.1 &    from.1?y -> free.1:=TRUE; out!y
    OR ~free.2 &    from.2?y -> free.2:=TRUE; out!y
    OD
  END

PROC Farm(in:?RAW, out:!CKD) IS
  LET
    CHANNELS to   : (!RAW, !RAW)
    CHANNELS from : (?CKD, ?CKD)
  IN
       farmer(in, to, from, out)
    || worker(to.1, from.1)
    || worker(to.2, from.2)
  END
```

The type definitions in Example 40 associate names `Farmer` and `Worker` with process types. The declaration of `GenFarm` organizes its parameters systematically into two groups: the left hand group describes its *value* parameters (in this case they have process types), and the right hand group describes its *port* parameters. To create a member of such a process family, connect it to its environment and start it, all its parameters must be provided.

For example, a replica of the process farm of Example 38 could be created and started using the notation

```
GenFarm(Farmer, Worker, left, right)
```

To *refine* (or specialize) a process family it is only necessary to provide an initial subtuple of its parameters. Example 41 shows how to name specialized variants of a process family. Each family defined there takes its input from a port of type $?RAW$ and sends its output to a port of type $!CKD$ — the families have (respectively) up to twice, four times and eight times the capacity of a single worker.

When the remaining parameters of a refined (or specialized) process family are provided, a member of the family is created and started. For example, on a computer system with enough processors, a farm with (up to) four times the capacity of that in Example 38 can be created and started using the notation

Example 40 A generic process farm

```
TYPE Farmer = (?RAW, (!RAW,!RAW), (?CKD,?CKD), !CKD) -> ()
TYPE Worker = (?RAW, !CKD) -> ()

PROC GenFarm(farmer:Farmer, worker:Worker, in:?RAW, out:!CKD) IS
  LET
    CHANNELS to   : (!RAW, !RAW)
    CHANNELS from : (?CKD, ?CKD)
  IN   farmer(in, to, from, out)
    || worker(to.1, from.1)
    || worker(to.2, from.2)
  END
```

Example 41 Process families defined by specialization

```
CON TwoWorkers:   Worker = GenFarm(Farmer, Worker)
CON FourWorkers:  Worker = GenFarm(Farmer, TwoWorkers)
CON EightWorkers: Worker = GenFarm(Farmer, FourWorkers)
```

```
EightWorkers(left, right)
```

2.3.1 Composite Channel Structures

Examples 42 and 43 show two ways of building a variadic process farm that uses composite channel structures to implement port types.

In Example 42 we define the interface to a worker as a pair of ports. A farmer is given a vector of raw output ports to write to, and a vector of cooked input ports to read from. It is assumed that these vectors are of the same size.

In Example 43 we define the interface to a worker as a port type `Window` — a worker reads raw material from the `raw` port of a window, and writes cooked material to the `ckd` port of the same window. A farmer is given a vector of window complements: it writes raw material to their `raw` ports, and reads cooked material from their `ckd` ports.

2.3.2 Procedures with results

So far we have seen only families of processes that communicate with their environment by means of channels, and which never terminate.

Most ordinary sequential programming languages also provide support for families of sequential processes called *procedures*, and eCSP is no exception. A procedure is simply a process family whose members are intended to terminate.

Example 42 A variadic process farm

```
TYPE Farmer = (?RAW, [!RAW], [?CKD], !CKD) -> ()
TYPE Worker = (?RAW, !CKD) -> ()

PROC GenFarm (size:INT, farmer:Farmer, worker:Worker,
              in:?RAW, out:!CKD) IS
  LET
    CHANNELS raw : [RAW]#size
    CHANNELS ckd : [CKD]#size
  IN
       farmer(in, raw, ckd, out)
    ||
       ||(FOR i IN 0#size) worker(raw i, ckd i)
  END

PROC farmer(in:?RAW, raw:[!RAW], ckd:[?CKD], out:!CKD) IS
  LET VAR free = [(FOR i IN 0#size) TRUE]
  IN
    DO
      OR (FOR i IN 0#size)
         ~free(i) & ckd(i)?y -> free(i):=TRUE; out!y
    OR
      OR (FOR i IN 0#size)
          free(i) & in?x     -> free(i):=FALSE; raw(i)!x
    OD
  END
```

Example 44 shows a procedure that calculates the GCD of its two integer arguments, and assigns the result to the variable `gcdanswer`.

The effect of the process $gcd(e_1, e_2)$ is the same as the effect of the process

```
LET
  VAR m:INT=e1
  VAR n:INT=e2
IN
  DO m<n -> n:=n-m OR m>n -> m:=m-n OD;
  gcdanswer := m
END
```

In other words, the body of the procedure is executed in an environment in which its formal parameter names are associated with variables which are initialised to the values of its actual parameters.

The experienced reader will no doubt be grumbling that it is not very satisfactory for a procedure to communicate its result by assigning to a global variable in this way. In this she would be in line with our own thinking, for firstly, the relationship between the procedure name and the variable in which the answer

Example 43 The variadic process farm revisited

```
TYPE Window = {| raw:?RAW, ckd:!CKD |}
TYPE Farmer = (Window, [-Window]) -> ()
TYPE Worker = (Window) -> ()

PROC worker(env: Window) IS
  DO env.raw?x -> env.ckd!cook(x) OD

PROC GenFarm (size:INT, farmer:Farmer, worker:Worker, env:Window) IS
  LET CHANNELS chans : [ {|raw=CHAN RAW, ckd=CHAN CKD|} ]#size
  IN
       farmer(env, chans)
    ||
       ||(FOR i IN 0#size) worker(chans(i))
  END

PROC farmer(env: Window, chans:[-Window]) IS
  LET VAR free = [(FOR i IN 0#chans) TRUE] IN
    DO
      OR (FOR i IN 0##chans)
         ~free(i) & chans(i).ckd?y -> free(i):=TRUE; env.ckd!y
    OR
      OR (FOR i IN 0##chans)
          free(i) & env.raw?x      -> free(i):=FALSE; chans(i).raw!x
    OD
  END
```

is placed is not made explicit at the site of the call, and secondly the definition of recursive procedures is made more complicated.

The problem is resolved by a form of procedure definition and invocation in which the variable into which the result will be placed can be named explicitly. In example 45 we present a revised version of the procedure definition, in which a "result parameter" is named explicitly.

The GCD procedure is invoked using the notation $variable := gcd(e_1, e_2)$ and the effect of this is the same as the effect of the process

```
LET
  VAR m:INT=e1
  VAR n:INT=e2
  VAR gcdanswer:INT IN
    DO m<n -> n:=n-m OR m>n -> m:=m-n OD;
    gcdanswer := m;
  variable := gcdanswer
END
```

Those familiar with the academic literature on programming languages will

Example 44 A GCD Procedure

```
VAR gcdanswer: INT

PROC gcd(m:INT, n:INT) IS
  BEGIN
    DO m<n -> n:=n-m OR m>n -> m:=m-n OD;
    gcdanswer := m
  END
```

Example 45 A GCD Procedure with a result parameter

```
PROC gcdanswer:INT := gcd(m:INT, n:INT) IS
  BEGIN
    DO m<n -> n:=n-m OR m>n -> m:=m-n OD;
    gcdanswer := m
  END
```

recognise that the semantics of this form of parameter passing is that known as "call by result". The definition and invocation notations have the advantage that a simple inspection of the call site demonstrates that a variable can be altered by a procedure call, and a simple inspection of the signature of the procedure demonstrates that it has result parameters as well as value parameters. Furthermore there is no need to introduce words describing the modes in which parameters are passed.[8]

Procedures may have multiple result parameters, as in example 46

Example 46 A Procedure with two result parameters

```
PROC min:INT, max:INT := order(left:INT, right:INT) IS
  IF   left>=right -> max,min := left,right
  ELSE TRUE        -> max,min := right,left
  FI
```

The names of one or more value parameters may also appear as result parameters, as in example 47. In such cases, we have the mode of parameter passing known in the literature as "call by value-result".

When a procedure with value-result parameters is invoked, the actual parameters in the corresponding value and result positions must be identical variables. For example, the `reorder` procedure is invoked using the notation

$$v_1, v_2 := \texttt{reorder}(v_1, v_2)$$

[8]These notations for procedure definition and invocation were first used by Bernard Sufrin in his 1971 implementation of a dialect of John Reynolds's experimental language *Gedanken*.

Example 47 A Procedure with two value-result parameters

```
PROC left:INT, right:INT := reorder(left:INT, right:INT) IS
IF   left>right -> left,right := right,left
ELSE TRUE       -> SKIP
FI
```

and the effect of this is the same as the effect of the following process — in which $\underline{left}$, and $\underline{right}$ are variables distinct from v_1 and v_2.

```
LET
  VAR left: INT = v1
  VAR right:INT = v2
IN
  IF   left>right -> left,right := right,left
  ELSE TRUE       -> SKIP
  FI;
  v1,v2 := left,right
END
```

We denote the type of this process as

$$(\mathbf{int}, \mathbf{int}) \rightarrow (\mathbf{int}, \mathbf{int})$$

where the tuple on the left of the arrow represents the value parameters and the tuple on the right represents the result parameters.

A procedure with a result parameter may be invoked in a context where an expression is expected. In this case the effect is the same as if a new variable had been declared and used as an output parameter before being substituted for the procedure invocation. For example

```
.... print(gcd(39,45)) ....
```

is equivalent to

```
LET VAR v:INT IN
  v := gcd(39,45);
  .... print(v) ....
END
```

If more than one procedure invocation appears in a single expression, then the order in which the procedures are invoked is not specified. For example

```
.... print(gcd(39,45)+gcd(45,98)) ....
```

is equivalent to one of the following

```
LET
  VAR v:INT
  VAR w:INT
IN
  v := gcd(39,45);
  w := gcd(45,98);
  .... print(v+w) ....
END

LET
  VAR v:INT
  VAR w:INT
IN
  w := gcd(45,98);
  v := gcd(39,45);
  .... print(v+w) ....
END
```

Likewise, a procedure with a result parameter may be invoked as part of a variable declaration: for example

```
LET
  VAR v = gcd(39,45)
  VAR w = gcd(45,98)
IN
  .... print(v+w) ....
END
```

is equivalent to

```
LET VAR v:INT
IN
  v := gcd(39,45);
  LET VAR w:INT
  IN
    w := gcd(45,98);
    .... print(v+w) ....
  END
END
```

The use of such procedures in constant declarations is also permitted, in which case the effect is the same as if new variables were declared and initialised, then new constants declared with the same values. For example

```
LET
  CON v = gcd(39, 45)
  CON w = gcd(45, 98)
IN
  .... print(v+w) ....
END
```

is equivalent to

```
LET VAR v:INT IN
  v := gcd(39, 45);
  LET VAR w:INT IN
    LET CON v=v, w=wi IN
      w := gcd(45, 98);
      .... print(v+w) ....
    END
  END
END
```

2.4 Sharing Resources

2.4.1 Static Communication Structures

Many systems are organised along client/server lines — where a server offers a resource to several clients. Whilst it is possible to model this kind of system in the language introduced so far, one of the constraints on such models is that they are static.

For example, in Example 48 we present a system composed of a token-granting server and 42 token-requesting clients. A client can ask for as many tokens as it wishes, and the server hands out a token to a client whenever one is requested. Each client's interface to the token-granting service is described by the port type `TokenService`, defined as

```
{| ask:!(), answer:?INT |}
```

and the server's interface to each of its clients is described by the complementary port type `-TokenService`, defined as

```
{| ask:?(), answer:!INT |}
```

We describe the system as *static* because the server is equipped in advance with communication channels to and from all its potential clients, and the clients are all started at the same time as the server.

2.4.2 Dynamic Communication Structures

Although there are many inherent strengths in organising a system statically there are a couple of weaknesses. The first weakness is that there are as many channels as there are potential clients, *even though only one client can be serviced at a time.* The second weakness is that the server and all its clients must be started simultaneously — there is no way of starting a new client when it

Example 48 A Static Client/Server System

```
TYPE TokenService = {| ask:!(), answer:?INT |}

PROC Server(clients: [-TokenService]) IS
  LET VAR counter = 0 IN
    DO
      OR (FOR i IN 0##clients)
         clients(i).ask?() ->
            clients(i).answer!counter;
            counter:=counter+1
    OD
  END

PROC Client(server: TokenService) IS
  LET VAR token:INT  IN
    ...
    ...
    server.ask!();
    server.answer?n -> token := n;
    ...
    ...
  END

-- The system
LET
  CHANNELS chans : [TokenService]#42
IN
  Server(chans) || ||(FOR i IN 0##chans) Client(chans(i))
END
```

becomes necessary to do so, and terminating it when it has served its purpose. The third weakness is that there is an absolute upper bound on the number of clients that can *ever* be offered service.

We need something a little more flexible than this in order to describe the way in which a server that doesn't know about all its clients in advance can offer services to them. The mechanism that is behind our solution is the *shared channel structure*. The declaration

shared chan $name{:}Type$

creates a shared channel whose port ends have type **shared** $?Type$ and **shared** $!Type$. These ports may be used by any number of processes — though only one inputting and one outputting process may communicate through the channel at a time. Shared composite channels also may be declared; for example

```
SHARED CHANNELS c: {| ask:    SHARED ?(),
                      answer: SHARED !INT
                   |}
```

constructs a shared channel structure c with a unit channel called `ask` and an integer channel called `answer`.

Just as there are two ports associated with every channel (the input port and the output port), there are two interfaces to every shared channel:[9] the *server interface* and the *client interface*. Every time a client needs to use a shared channel for communication with a server it must ensure that no other process is using it as a client by waiting until it can become the owner of the client interface to the channel. Once the communication has finished the client must renounce ownership of the channel. This is accomplished by a process of the form

client n **in** communications with server **end**

where n is the name of the shared channel or one of its ports.

Likewise, whenever a server uses a shared channel for communicating with a client it must wait until it can become the exclusive owner of the server interface to the channel, then give up ownership when communication with the client has finished:

server n **in** communications with client **end**

For example, within the scope of the declarations

```
TYPE SineService = {| input:!REAL, output:?REAL |}
SHARED CHANNELS sine : SineService
```

the client interface to the shared channel is claimed, used and then renounced by the process

```
CLIENT shared IN
  sine.input!34; sine.output?n -> ...
END
```

It will be delayed until the shared channel structure is free, after which it communicates using the given ports before once again freeing the channel structure for re-use.

Within the scope of the same declarations, the server interface is claimed, used, and then renounced by a granting process

```
SERVER shared IN
  sine.input?n -> sine.output!SINE(n)
END
```

[9] Here and subsequently we abbreviate "shared channel structure" to "shared channel".

The granting process is synchronised with the claiming process at the point at which the bodies of the processes begin, and the termination of the bodies of the granting and claiming processes is also synchronised.

In example 49 we present a revised version of a server offering the same functionality as example 48, but which expects to offer its services to clients that it doesn't know about in advance. The body of a corresponding client is shown in example 50. The synchronization between `client.ask?()` and `server.ask!()` in example 48 is no longer necessary, since `SERVER client` in the server and `CLIENT server` in the client synchronise with each other.

Example 49 A Dynamic Server

```
TYPE TokenService = ?INT

PROC Server(client: SHARED -TokenService) IS
  LET VAR counter = 0 IN
    DO
      SERVER client IN
        client!counter;
        counter:=counter+1
      END
    OD
  END
```

Example 50 A Client for the Dynamic Server

```
PROC Client(server: SHARED TokenService) IS
  LET  VAR token: INT IN
    ...
    ...
    CLIENT server IN
      server?n -> token:=n
    END
    ...
    ...
  END
```

A token service is set up and offered by declaring a shared channel and running the server in parallel with the clients.

```
LET SHARED CHANNELS tokenservice : TokenService IN
     Server(tokenservice)
  ||
     ||(FOR i IN 1..42) Client(tokenservice)
END
```

Although one might usually expect to find many clients meeting a single server at a shared channel, in fact it can act as a meeting-place for any number of clients and any number of servers, as in example 51.

Example 51 Three clients using two servers

```
LET SHARED CHANNELS service : Interface IN
     ServerA(service)
  || ServerB(service)
  || ClientX(service)
  || ClientY(service)
  || ClientZ(service)
END
```

2.4.3 Alternation and Iteration of Granting Processes

If $c_1, ...c_n$ are **shared** port structures, then a process may use them as part of a **server** alternation or iteration

A **server** alternation takes the form

$$\begin{array}{l}
\textbf{if } e_1 \;\&\; \textbf{server } c_1 \textbf{ in } a_1 \textbf{ end} \\
\textbf{or } e_2 \;\&\; \textbf{server } c_2 \textbf{ in } a_2 \textbf{ end} \\
\textbf{or } \cdots \\
\\
\textbf{or } e_n \;\&\; \textbf{server } c_n \textbf{ in } a_n \textbf{ end} \\
\textbf{fi}
\end{array}$$

where the e_i are boolean expressions, known as guards, (which may be omitted if identically **true**), and the a_i are processes. Such a process first evaluates all the guards, then waits until a client claim is made for at least one of the shared channels corresponding to the true guards — at which point one of the claims is granted.

A **server** iteration takes a similar form

$$\begin{array}{l}
\textbf{do } e_1 \;\&\; \textbf{server } c_1 \textbf{ in } a_1 \textbf{ end} \\
\textbf{or } e_2 \;\&\; \textbf{server } c_2 \textbf{ in } a_2 \textbf{ end} \\
\textbf{or } \cdots \\
\\
\textbf{or } e_n \;\&\; \textbf{server } c_n \textbf{ in } a_n \textbf{ end} \\
\textbf{od}
\end{array}$$

It is equivalent to

$$
\begin{array}{l}
\textbf{do}\ \ e_1 \vee e_2 \vee \cdots \vee e_n \rightarrow \\
\quad \textbf{if}\ \ e_1\ \&\ \textbf{server}\ \ c_1\ \ \textbf{in}\ \ a_1\ \textbf{end} \\
\quad \textbf{or}\ e_2\ \&\ \textbf{server}\ \ c_2\ \ \textbf{in}\ \ a_2\ \textbf{end} \\
\quad \textbf{or}\ \cdots \\
\\
\quad \textbf{or}\ e_n\ \&\ \textbf{server}\ \ c_n\ \ \textbf{in}\ \ a_n\ \textbf{end} \\
\quad \textbf{fi} \\
\textbf{od}
\end{array}
$$

2.4.4 Dynamic Processes

A shortcoming of the arrangement in example 51 is that the processes must be completely specified at compile time. An alternative scheme is to create processes dynamically as they are needed. This is done using the **fork** construct.

If p is a process of type $t \rightarrow ()$ and (v) has type t, then **fork** $p(v)$ invokes $p(v)$ as a separate process (we call this a *child* process) that runs concurrently with the process that invoked it (the *parent* process). Note that if the child process dies, the parent will continue to run; but if the parent process dies all its children will also stop running.

Example 52 shows a process `farmer` which creates a new `server` process each time a request comes on the `req` channel from one of the `client` processes.

2.5 Syntactic Sugar

In this section we outline some convenient notational features of eCSP whose meanings are defined in terms of the core language. The term "syntactic sugar" is due to Christopher Strachey and describes convenient high-level notations whose meanings are defineable in terms of core language features — the idea is that "syntactic sugar" can render "semantic spice" more palatable.

2.5.1 Strings

Strings are vectors of characters, and type `STRING` is a built-in synonym for `[CHAR]`. Literal strings can be denoted using the usual square bracket notation for vectors, or by delimiting the string with double quotes. The two forms are interchangeable:

```
"Hello, world!"  ≡  ['H','e','l','l','o',' ','w','o','r','l','d','!']
""               ≡  [ ]
```

Example 52 Dynamic server creation

```
PROC client (req: SHARED !(),  service: SHARED Interface) IS
  ...
END;

PROC server (service: SHARED -Interface) IS
  ...
END;

PROC farmer (req: SHARED ?(),  service: SHARED -Interface) IS
  DO
    SERVER req IN req?() -> FORK server(service) END
  OD

LET
  SHARED CHANNELS service: Interface;
  SHARED CHANNELS request: ?()
IN
     client (request, service)
  || client (request, service)
  || farmer (request, service)
END
```

2.5.2 Disjunctive Patterns

When the same action is associated with more than one guard/pattern in an alternation it is a tedious and error-prone task to write out the action repeatedly.

A multiple pattern of the form

$$G \;\&\; \pi_1 \textbf{ or } \pi_2 \textbf{ or } ... \textbf{ or } \pi_n \rightarrow a$$

may appear in an iteration or alternation. It is equivalent to

$$G \;\&\; \pi_1 \rightarrow a \;\textbf{ or }\; G \;\&\; \pi_2 \rightarrow a \;\textbf{ or }\; ... \;\textbf{ or }\; G \;\&\; \pi_n \rightarrow a$$

The patterns in such a construct must not contain variable names, hence they will not bind any new values. They may however contain wildcard (_) patterns.

Similarly,

$$G \;\&\; c?\pi_1 \textbf{ or } \pi_2 \textbf{ or } ... \textbf{ or } \pi_n \rightarrow a$$

is equivalent to

$$G \;\&\; c?\pi_1 \rightarrow a \;\textbf{ or }\; G \;\&\; c?\pi_2 \rightarrow a \;\textbf{ or }\; ... \;\textbf{ or }\; G \;\&\; c?\pi_n \rightarrow a$$

Example 53 A disjunctive pattern

```
IF
  input ? Yes OR Maybe -> GoAhead()
OR
  input ? No           -> Retreat()
FI
```

2.5.3 Bounded Iteration

If E:$[T]$ is a vector, G is a boolean expression (which may be omitted, if identically **true**, together with the &), and P is a process, then the form

$$\textbf{do}\ G\ \ \&\ \ (\textbf{for}\ i\ \ \textbf{in}\,E) \rightarrow P\ \textbf{od}$$

means "While G is true, execute P with i bound to successive elements of E." More formally, it can be translated into the following (where v', v'' are fresh variables which do not appear in G, i, P):

$$\textbf{let var}\ v'{:}[T] = E\ \ \textbf{in do}\ G\ \ \&\ \ v'\ \textbf{is}\ [i] + v'' \rightarrow P;\ v' := v''\ \textbf{od end}$$

Similarly if E:$\{T\}$ is a set then the form

$$\textbf{do}\ G\ \ \&\ \ (\textbf{for}\ i\ \ \textbf{in}\,E) \rightarrow P\ \textbf{od}$$

means "Choosing each element of E at most once, and while G is true, bind i to the chosen element and execute P." More formally, it can be translated into

$$\textbf{let var}\ v'{:}\{T\} = E\ \ \textbf{in do}\ G\ \ \&\ \ v'\ \textbf{is}\ \{i\} + v'' \rightarrow P;\ v' := v''\ \textbf{od end}$$

2.5.4 "Bidirectional" Ports and Channels

When a process communicates with a peer process via an input and an output port, it can be useful to give both ports the same name, rather than having to invent two names.

If T_1, T_2 are types, then $?T_1!T_2$ denotes the port tuple type $(?T_1, !T_2)$, and $!T_1?T_2$ denotes its complement tuple type $(!T_1, ?T_2)$. If $c{:}?T_1!T_2$ then the action $c.1?x \rightarrow a$ may be abbreviated to $c?x \rightarrow a$, and the action $c.2!v$ may be abbreviated to $c!v$; likewise if $c{:}!T_1?T_2$ then $c.1!v$ is abbreviated to $c!v$ and $c.2?x \rightarrow a$ to $c?x \rightarrow a$. The pair of channels that implement the requisite ports and their connections may be created by either of the delcarations **channels** $n{:}?T_1!T_2$, **channels** $n{:}!T_1?T_2$.

2.5.5 Remote Procedure Call

The shared channel machinery is a relatively low-level mechanism. Typically it will be used for providing a service to any of a number of client processes. With this in mind we introduce a slightly more memorable notation.

The shared channel is declared as a remote procedure — perhaps with some result parameters.

$$\textbf{shared proc } RT_1, ..., RT_n := name(VT_1, ..., VT_m)$$

This means

$$\textbf{shared channels } \ name : (?(RT_1, ..., RT_n), !(VT_1, ..., VT_m))$$

A server process grants the channel with a **server** process:

$$\begin{array}{l} \textbf{server } r_1{:}RT_1, ...r_n{:}RT_n := name(v_1{:}T_1, ...v_m{:}T_m) \ \textbf{ in} \\ \qquad body \\ \textbf{end} \end{array}$$

This means

$$\begin{array}{l} \textbf{server } name \ \textbf{ in} \\ \quad name.2?(v_1, ..., v_m) \rightarrow \\ \qquad \textbf{let } \ \textbf{var } r_1{:}RT_1, ..., r_m{:}RT_n \ \textbf{ in} \\ \qquad\quad body \\ \qquad\quad name.1!(r_1, ..., r_n) \\ \qquad \textbf{end} \\ \textbf{end} \end{array}$$

The channel is claimed using a syntax that is no different from that of an ordinary procedure call:

$$ar_1, ..., ar_n := name(e_1, ..., e_m)$$

In the given context, this means

$$\begin{array}{l} \textbf{client } name \ \textbf{ in} \\ \qquad name.2!(e_1, ..., e_m); \\ \qquad name.1?(v_1, ...v_n) \rightarrow ar_1 := v_1; \ ...; \ ar_n := v_n \\ \textbf{end} \end{array}$$

In example 58 (on page 42) we present part of an alternative implementation of the sine computation service presented earlier.

3 The Module Language

3.1 Modules and Interfaces

The simplest forms of module provide a way of packaging a data structure with a particular behaviour whilst hiding its implementation. The declaration in example 54 declares a *module type*, known as `Tokens`, which exports the type `Token`, the constant `limit`, and the procedures `newToken` and `freeToken`, while forbidding any process using these procedures to take advantage of the knowledge that `Token` is implemented as `INT`.

Example 54 A module that hides its implementation

```
MODULE Tokens IMPLEMENTS
  TYPE  Token
  CON   limit: INT
  PROC  Token := newToken()
  PROC  freeToken(Token)
WITH
  TYPE Token  = INT
  CON  limit  = 42
  VAR  unused : [BOOL] = [(FOR i IN 0#limit) TRUE]

  PROC tok:Token := newToken() IS
    LET VAR search = TRUE IN
      DO search & (FOR i IN 0#limit) ->
        IF unused(i) -> unused(i),tok,search := FALSE,i,FALSE FI
      OD
    END

  PROC freeToken(tok:Token) IS
    unused(tok) := TRUE
END
```

The set of exported types and values is called the module's *interface*. It can be declared independently of the module declaration, thus allowing more than one module to implement an interface. Example 55 shows the same interface as before, but with different implementations based on different Token types.

In some cases it is advantageous to parameterise modules or interfaces by types and/or values. In example 56 we define a parameterised interface *Atom* describing modules that administer collections of atoms, and a parameterised module which implements the interface by administering a set of atoms which are supplied in advance.

To create an instance of this module we use an `INSTANCE` declaration, *e.g.*

```
INSTANCE x:AtomInterface <INT> IS Atoms <INT> ({(FOR i IN 0#42) i})
```

Example 55 Two modules implementing the same interface

```
INTERFACE TokenInterface IS
  TYPE  Token
  CON   limit: INT
  PROC  Token := newToken()
  PROC  freeToken(Token)
END;

MODULE IntTokens IMPLEMENTS TokenInterface WITH
  TYPE Token = INT;
  ...
END;

MODULE StringTokens IMPLEMENTS TokenInterface WITH
  TYPE Token = STRING;
  ...
END
```

This instance of the module is called `x`, and it exports the type `x.Atom` (which in this case will be `INT`), the constant `x.limit` (42 in this instance), and the procedures `x.newAtom` and `x.freeAtom`.

We can create any number of instances of a parameterised module, and the specification of the interface in an instance declaration (e.g. "`:AtomInterface<INT>`") is optional.

Example 57 shows another implementation of `TokenInterface`. It introduces a parameterised module, called `RemoteTokens`, which can be instantiated by a declaration of the form

```
INSTANCE tokeninstance IS RemoteTokens(sharedchannel)
```

where `sharedchannel` is a shared channel to a remote token server. The `INITIALLY` process gets run when `tokeninstance` comes into existence — it is analogous to the initialising expression of an ordinary constant or variable. The `FINALLY` process gets run after termination of the body of the block in which `tokeninstance` is declared.

In example 58 we present another parameterised module which, when instantiated, offers a sine-computation service.

Example 56 Parameterised interface and module

```
INTERFACE AtomInterface <Atom> IS
  CON   limit: INT
  PROC  Atom := newAtom()
  PROC  freeAtom(Atom)
END

MODULE Atoms <Atom> (original: {Atom})
IMPLEMENTS AtomInterface <Atom>
WITH
  VAR  unused : {Atom} = original
  CON  limit = #original

  PROC tok:Atom := newAtom() IS
  IF
     unused IS {sometok}+rest -> tok,unused := sometok,rest
  FI

  PROC freeAtom(tok:Atom) IS
       unused := unused + {tok}
END
```

Example 57 An Implementation Description

```
TYPE TokenDatabasePort =
     {| begin:      ?INT;
        newtoken:   !()?INT;
        freetoken:  !INT;
        end:        !()
     |}

MODULE RemoteTokens(db: SHARED TokenDatabasePort)
IMPLEMENTS TokenInterface
WITH
  TYPE Token = INT
  CON  limit:INT = 42

  PROC tok: Token := newToken() IS
    BEGIN
      db.newtoken ! ();
      db.newtoken ? n -> tok:=n
    END

  PROC freeToken(tok: Token) IS
       db.freetoken!tok

  INITIALLY
    db.begin ? n -> limit:=n
  FINALLY
    db.end ! ()
END
```

Example 58 A sine server type with usage reporting

```
MODULE SineService(report: !INT) IMPLEMENTS
  SHARED PROC REAL := sine(REAL)
WITH
  VAR  usage: INT = 0

  PROC result:REAL = SINE(value: REAL) IS ....

  INITIALLY
    DO
      SERVER output:REAL := sine(input: REAL) IN
        output := SINE(input);
        usage := usage + 1
      END
    OD

  FINALLY
    report!usage
END
```

4 Services

4.1 Introduction

The conceptual architecture behind the provision of services is rather simple. A service (for example: number translation), will in general have several *aspects* (for example: client customisation, call handling, statistics reporting). Each invocation of every aspect of every service is handled in the INS by spawning an aspect-specific process in response to the initial call.

This *handler* process oversees the entire life of the invocation. It may call on facilities provided by other processes — either other evanescent handlers or long-lived utilities provided as part of the INS infrastructure such as databases and network adapters. Note that the distribution of messages from the network amongst currently-active processes, and the multiplexing of messages intended for the network will be implemented by a network adapter layer.

4.2 Example: A "Follow Me" Service

We begin with an example of a simple service in which calls to the subscriber's number are rerouted to another number. When the subscriber rings a dedicated number, he is prompted for information for a new routing. Thus, as the subscriber makes his way from office to customer to restaurant and back to office, he rings the special number upon his arrival at each place so that he can receive his calls on that phone. (Of course a full version of such a service would need more features to make it really useful, but this will suffice to show how a service is implemented.)

At the topmost level of description, we note that there are two aspects to this service: the routing of calls to the subscriber, and the changing of routing target. Although these aspects are processes that are run independently of each other, they may share some common resources — in particular they will need to refer to the same mapping from the subscriber's ordinary number to his new number.

The overall structure of the service is as follows:

```
SERVICE FollowMe IS
  VAR translate : Number +> Number

  ASPECT Route (caller:Number, callee:Number; sw:Switch) IS
    ...
  END

  ASPECT Edit (caller:Number, callee:Number; sw:Switch) IS
    ...
  END
END
```

The `translate` function is a finite mapping from `Number` (i.e. a telephone number) to `Number`, which implements the current routing for all subscribers to this service. Although it will be implemented using database access, the service description uses it only at this abstract level. This has important implications for efficiency which we discuss below.

`Route` and `Edit` are processes, instances of which are created dynamically when the appropriate trigger is detected. For `Route` this will be the dialling of a subscriber's number; for `Edit` it will be the dialling of the dedicated number for changing the routing. Triggers are set when these service aspects are installed; they are not part of the description of aspects.

The parameter list of a service aspect implements the interface between an instance of the aspect and the environment that created it and with which it interacts. In this example, each aspect is provided with the dialling number, the dialled number, and a port through which it communicates with the network. In practice this will be a process that provides the aspect with a simple and standardised view of network interaction.

First let us look at the body of the `Route` aspect:

```
ASPECT Route (caller:Number, callee:Number; sw:Switch) IS
  LET VAR callee':Number
  IN
    IF   callee MEM DOM translate -> callee' := translate(callee)
    ELSE TRUE                     -> callee' := callee
    FI;
    sw!Connect (caller,callee')
  END
```

If the called function is in the domain of `translate` — *i.e.* `translate` includes a mapping from that number — we use `translate` to find the new destination; otherwise we use the number originally dialled. An instruction is then sent to the switch asking for the connection to be made; this instruction will not require a reply, as there is no special action to be taken on a successful or unsuccesful connexion. There is nothing further for the aspect instance to do, so it terminates.

Now consider the `Edit` aspect:

```
ASPECT Edit (caller:Number, callee:Number; sw:Switch) IS
  sw!GetSubNumber;
  IF
    sw?Number sub
          -> sw!GetDestNumber;
             IF   sw?Number dest -> translate(sub) := dest
             ELSE sw?_           -> SKIP
             FI
  ELSE
    sw?_  -> SKIP
  FI
END
```

The switch is instructed to perform whatever actions are required to obtain the subscriber number and the destination number from the caller. (Note that the `caller` and `callee` numbers are not actually used in this process.) If numbers are yielded as a result of both these requests, the translate mapping is updated accordingly; otherwise no action is performed.

4.3 Database access optimisation

Consider a slightly more elaborate routing service than that described above, where routing can depend upon such things as the day of the week or the caller's area code, and may yield a sequence of alternative numbers to try. The abstract interface to the database used by such a service would, of course, be described by a more elaborate translation mapping – perhaps

$$translate{:}Number \nrightarrow (UserData \nrightarrow [Number])$$

It is important that the underlying database accesses are performed as efficiently as possible, so we almost certainly want to avoid generating multiple database accesses to retrieve the number sequence.

The problem may seem like one whose solution demands that the service designer deal with database access at a lower level. We wish to avoid this solution because not only does it make the programmer's task more difficult, it also makes the result less portable because it depends on the relative efficiencies of basic actions, and these may vary from machine to machine, from library to library, or from one version of a library to the next.

We choose instead to leave the optimisation to the compiler. While this would allow a programmer to inadvertently implement a service with unnecessarily expensive database access, the compiler would be able to detect this (it knows what patterns of access it can generate efficient code for) and warn the programmer at compile time. Should the set of desirable patterns change, perhaps owing to a library re-implementation, the compiler can identify any existing service programs whose database requirements will need to be restructured if they are to remain efficient.

It is, desirable, of course that the compiler be highly configurable with respect to the database access library, so a notation will have to be designed that can be used to describe that library succinctly. It seems unnecessary to burden service designers with the task of learning and using the configuration notation – we expect this to be the province of a database specialist.

References

[1] INMOS Ltd., *occam$^{\mathrm{TM}}$ 2 Reference Manual*, Series in Computer Science, Prentice-Hall International (1988).

[2] G. Jones & M. Goldsmith, *Programming in occam$^{\mathrm{R}}$ 2*, Series in Computer Science, Prentice-Hall International (1988).

[2] C.A.R. Hoare, *Communicating Sequential Processess*, Series in Computer Science, Prentice-Hall International (1985).

[3] J.M. Spivey, *The Z Notation: A reference manual*, Series in Computer Science, Prentice-Hall International (1992).

[4] C. Morgan, *Programming from Specifications (2nd edn.)*, Series in Computer Science, Prentice-Hall International (1994).