eCSP – a domain-adapted, higher order, concurrent programming language

Bernard Sufrin
(with Quentin Miller)

Work funded by GPT/Marconi Telecommunications/Marconi

OUCL Cakes Talk: May 4th 2000

◇ What's a value-added telephone service? Examples....

- ◦ Find-me (autoforwarding)
- ◦ Answerphone
- ◦ Voting
- ◦ (Multimedia) Conference call
- ◦ (Virtual) Callcentre (emergency) call
- ◦ Download a flick

◇ What happens when you dial a VA service access number? ◃

- ◦ Caller-side switch forwards the call to an IN centre via POTN ◃
- ◦ POTN does switching, message handling, digit-collecting, ...
- ◦ The INC instructs the network to do the right thing(s) ◃

◇ How are these kinds of service evolving?

- ◦ They're getting more complicated
- ◦ So is the business model (*viz.* independent service provision)
- ◦ Advent of broadband "edutainmunication" complicates context

◇ What techniques are used to implement them?

- ◦ (Yes, Virginia) Finite State Machines!

◇ What's wrong with that?

History of eCSP

◇ Marconi's original goals for the project ◁

- ○ A domain-adapted language with *adequate expressive power* ◁
- ○ High degree of service-wide and system-wide predictability ◁
- ○ A prototype design environment ◁

◇ Our original plan

- ○ Investigate existing service specs and implementations ◁
- ○ Ponder for a while
- ○ Prototype design of a "little" language to make programs which could sit within the existing IN architecture. ◁
- ○ Proof-of-concept to be by case studies of existing services.

◇ But not all went according to plan – in our ignorance we started designing services of our own and realized that

> Concurrency is unavoidable if you want to *simplify* the descriptions of useful and interesting services

◁

◇ But could we persuade the Marconistas of this? ◁

A Sequential eCSP Miscellany

ML style primitive data, linearly extensible free datatypes and records (with subtyping)

```
DATA DefiniteAnswer = Yes + No
DATA Answer         = DATA DefiniteAnswer + Maybe

TYPE Coord2D      = {| x:INT; y:INT |}
TYPE Coord3D      = Coord2D + {| z:INT |}
```

◁

Finite sets, sequences, relations, mappings, tuples

```
VAR forwardto: SUBSCR +>  [PHONENUMBER]
VAR owns:      SUBSCR <-> PHONENUMBER

VAR cache:     [INT]#100          = [(FOR i IN 0#100) fib(i) ]
VAR defaults:  [(STRING, STRING)] = []
```

Pattern guards

```
PROC result:DefiniteAnswer := improve(ans:Answer) IS
DO
   ans IS Maybe -> ans := PromptReadAnswer("Please be definite")
OD
IF ans
IS Yes -> result := Yes
OR No  -> result := No
FI
```

◁

Procedures, conventional guarded commands ◁

```
PROC a,b: T := sort(a,b: T) IS
IF a<=b -> SKIP
OR b<a  -> a, b := b, a
FI

...

m, n := sort(m, n)
```

eCSP generalises occam

◇ A T channel connects a T-input-port (?T) to a T-output-port (!T)

◇ Simple ports can be used only in one direction

```
PROC Buffer(left:?INT, right:!INT) IS
LET CHAN middle:INT IN
    DO left?x  -> middle!x OD || DO middle?y -> right!y OD
END

PROC Buffer2(left:?INT, right:!INT) IS
LET CHAN middle:INT IN
    Buffer(left, middle) || Buffer(middle, right)
END
```

◇ Ports can be "bundled" into interfaces.

```
TYPE Interface = {| raw:?RAW, ckd:!CKD |}
```

◇ The worker sees one end of the bundle – that of type `Interface`

```
PROC worker(client:Interface) IS
  DO client.raw?x -> client.ckd!cook(x) OD
```

◇ The complementary end of the bundle has type `-Interface`

```
PROC client(worker: -Interface) IS
     ... worker.raw!question  ...
     ... worker.ckd?answer ...
```

◇ Bundles can be built with a `CHANNELS` declaration

```
LET CHANNELS bundle: Interface
IN  client(bundle) || worker(bundle)
    ...
```

◇ Channels can carry ... ports.

```
fromBroker : ? Interface
...
fromBroker?client -> worker(client)
```

Example

◇ A process farm

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



PROC farmer(client: Interface, chans:[-Interface]) 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; client.ckd!y

     OR

      OR (FOR i IN 0##chans)
          free(i) & client.raw?x   -> free(i):=FALSE; chans(i).raw!x

     OD
END



PROC Farm (size:INT, farmer:Farmer, worker:Worker, client:Interface) IS
LET  CHANNELS chans : [ Interface ]#size
IN
     farmer(client, chans) || ||(FOR i IN 0#size) worker(chans(i))
END
```

◇ Process families can be specialized (by partial application)

```
CON twoworkers:    Worker = Farm(2, farmer,  worker)
CON fourworkers:   Worker = Farm(2, farmer,  twoworkers)
CON dozenworkers:  Worker = Farm(3, farmer,  fourworkers)

...

LET
   CHANNELS interface: Interface
IN
   client(interface) || dozenworkers(interface)
END
```

◇ Client-use and Server-use of shared channels are independent critical sections

```
SHARED CHANNELS bus: CalculationService

CLIENT bus IN
       ... bus.request!Pi; bus.reply?answer ...
END

||

SERVER bus IN
       bus.request?x -> bus.reply!SINE(x)
END
```

◇ Shared Channels are meeting-places for clients and servers

```
LET SHARED CHANNELS bus: CalculationService IN
    ServerA(bus) || ServerB(bus) || ClientX(bus) || ClientY(bus)
END
```

◇ A dynamic server: ready to be introduced to clients by a broker

```
PROC Service(system:?Config, frombroker: ? SHARED -CalculationService)
IS
DO frombroker?client ->
   LET VAR state:ONLINE|OFFLINE|TERMINATED = OFFLINE IN
   DO state ISNT TERMINATED ->
      IF system?TERMINATE     -> state := TERMINATED
      OR system?ON            -> state := ONLINE
      OR system?OFF           -> state := OFFLINE
      OR
         state = ONLINE & SERVER client
         IN
             client.request?x -> client.reply!SINE(x)
         END
      FI
   OD
OD
```

Contravariant Subtyping Issues

Suppose $T \sqsubseteq T'$, *i.e.* a T will do when a T' is expected.

- Output ports: $!T \not\sqsubseteq\ !T'$, to forestall nasty surprises at input port.

  Example:

```
TYPE pt2 = {| x:INT, y:INT |}
TYPE pt3 = pt2 + {| z: INT |}

PROC send2(out: !pt2, x: pt2) IS out!x

LET CON  x2: pt2 = {| x=3, y=4 |}
    CHAN c3: pt3
IN
    send2(c3, x2)                          --- is badly-typed
    ||
    c3?y3 -> .... y3:pt3 is a pt2....     --- NASTY SURPRISE
END
```

- Arrays: $[T] \sqsubseteq [T']$

  "Smuggling" counterexample

```
PROC mangle(f2:[pt2]) IS LET CON x2:pt2 = ... IN f2[0]:=x2 END -- prevent this

VAR  f3:[pt3] ... mangle(f3) ... f3[0]:pt3 is a pt2 ...        -- NASTY SURPRISE
```

  Elementwise assignment interpreted as whole-array assignment makes it impossible to "smuggle".

```
PROC f2:[pt2] := mangle(f2:[pt2]) IS LET CON x2:pt2 = ... IN f2[0]:=x2 END
VAR  f3:[pt3] ... f3 := mangle(f3) ... --- is badly typed
```

- Mappings to: $U \not\rightarrow T \ \sqsubseteq\ U \not\rightarrow T'$ (ditto)

- Mappings from: $T \not\rightarrow U \ ?\ T' \not\rightarrow U$ – guess!

Domain-specific features of eCSP

- Database liaison for services is via abstract table descriptions
  - Table accesses are translated to eCSP communications with DB
  - The translation schemes (in the compiler) are configurable
  - Simple interface *essential* for automatic load prediction

- A simple translation service with two kinds of access

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

  ASPECT Route (caller,callee: Number; network: NetworkAdapter) IS
    ... network!Connect(caller, translate(callee)) ....
  END;

  ASPECT Edit (caller,callee: Number; network: NetworkAdapter) IS
    ... translate(callee) := newtranslation ....
  END
END
```

  - The `translate` mapping is the sole interface to the database.
  - Explicit scoping of shared structure supports optimisation.
  - An `ASPECT` is a process invoked by the infrastructure when an IN call is made.
  - Its parameters are the sole interface to the network.

- ◇ Module Language permits late binding of parts of implementation, and thereby supports generic designs ("frameworks").
- ◇ IN-specific feature implementation needs some work from Marconi.
- ◇ Current prototype translates full eCSP into Java
- ◇ Channels are implemented in ways which depend on whether their endports reside in the same or in different address spaces.
- ◇ Subsets are translateable into occam 2, C^{++}+threads, *etc.*
- ◇ We are designing a family of lightweight eCSP virtual machines which are considerably simpler than the JVM

- ◇ Why Java first?
  - ○ We were attracted by the existing Java class library (esp: GUI and Networking)
  - ○ We were using Java in another domain and found the lack of a serious statically-enforceable discipline of concurrent Java programming alarming
  - ○ We wanted to explore some ideas about mixed-paradigm programming based on the strong duality between free datatype channels and object interfaces.

- ◇ What can be thrown away?

But why didn't you use .....?

- ◇ occam 3
  - ∘ We agonized (BS was a consultant for occam 3)
  - ∘ High-level model of concurrent programming
    but ...
  - ∘ rather low-level model of data
- ◇ Java
  - ∘ Intrinsic concurrency features are very low-level
    ... and weren't soundly implemented when we started
  - ∘ Nonexistent discipline of concurrent Java programming
- ◇ Occam-in-Java
  - ∘ No safe implementation existed when we started.
  - ∘ Notational incoherence (collection of idioms).
  - ∘ Hard to identify efficiently-compileable subsets.
- ◇ Concurrent ML
  - ∘ Well thought-out module system
    but ...
  - ∘ Notational incoherence (collection of idioms).
  - ∘ No subtyping
  - ∘ Very hard to identify efficiently-compileable subsets.

and anyway, language design is fun!

Note 1 (p1)
Such a number isn't necessarily "special", it might simply be that of a friend who has enabled a particular feature on his line.

Note 2 (p1)
BT have two of these intelligent networking centres UK-wide.

Capacity is about 10m calls per minute! Performance is an issue.

Note 3 (p1)
It nearly always needs to consult a database. A lot of hard work has gone into designing these databases for speed.

Note 4 (p2)
Brief chronology

- ◇ Early 1996 Marconi "officially" recognises limitations of FSM
- ◇ Mid 1996: Marconi meet with Tony Hoare and Bernard Sufrin
- ◇ Later: BS meets with Marconi management to work out a project proposal for which OUCL will tender.

Note 5 (p2)
They were thinking of a special-purpose library embedded in C or C^{++} which are the languages in which the substrate IN code is written.

We realized that we had to investigate the semantics of existing and proposed services in order to help us understand what "adequate" meant. The result of this investigation surprised and alarmed us at first (see later)

Note 6 (p2)
Lack of this in the FSM

Note 7 (p2)
"But we realize that this is research and that you may want to change the goalposts".

Note 8 (p2)
Specs were highly *stylized* which lent them an air of formality, but they were so

prolix as to be virtually incomprehensible to folk other than the requirements teams (salespeople).

Implementations were “inside” an environment which meant that they had to be viewed through the relatively narrow (form-based, by and large) keyhole offered by the UI of the environment.

Only two real levels of abstraction possible in paper descriptions: “the whole thing” , or “the whole state machine”

Although this was a difficult phase of the work, in retrospect it was a stroke of good luck that for various reasons we didn’t have their environment available to us. It meant we had to start thinking up services off our own bat, unconstrained by what we knew of the limitations of the implementation machinery.

Note 9 (p2)
We were definitely thinking of a sequential language when we first wrote the plan.

Note 10 (p2)
Certainly nothing less than pseudo-concurrency is satisfactory.

Note 11 (p2)
The answer turned out (over the next three months) to be “yes”. But first we had to have a first cut a a design of a service that woould be convince them of this, and a first cut at the design of a useful design notation for it.

We started with an eclectic mixture of features from ML, Z, and Occam 3.[1] Not considering it formally a language at all but merely a notation in which to express our intentions.

Note 12 (p3)
$\mathit{DefiniteAnswer}$ is a subtype of Answer which means that a $\mathit{DefiniteAnswer}$ can appear wherever an Answer is required.

On the other hand, $\mathit{Coord3D}$ is a subtype of $\mathit{Coord2D}$ which means that a $\mathit{Coord3D}$ can appear wherever an $\mathit{Coord2D}$ is required. Finally, $\{|z{:}\mathit{INT}|\}$ is not a subtype of $\mathit{Coord3D}$.

Note 13 (p3)
equivalent to

[1] We used semantic concepts from Occam 3: the occam family has an abominable concrete syntax.

```
DO
    ans IS Maybe -> ans := PromptReadAnswer("Please be definite")
OD
result := ans
```

Note 14 (p3)
Here (as it happens) is a procedure with a pair of value/result parameters. The notation for the call is echoed in the procedure heading, and this means we need no explicit notation for parameter mode (value/result/reference/value-result).

Note 15 (p6)
Static communication topologies are safe, but require full resource-commitment up-front. In reality systems go through phases, and resources can be better utilized if they're committed as needed.

Most systems evolve dynamically: new components are downloaded and expect to use the existing infrastructure. We can't afford to rebuild the software for a switch just to add a single service or feature. Plug-and-play demands dynamic introduction of resources to each other.

Note 16 (p6)
The `SERVER` construct may be used as an input guard in an input alternation.

The architecture is fixed here. The system expects to have two fixed channels to this server, and to repeatedly send a client channel to the service then instruct the service about its performance.