Compiling CSP

or having fun with a new occam-$\pi$ compiler and CSP

(and fringe presentation)

Fred Barnes, Systems Research Group
Computing Laboratory, University of Kent, UK
F.R.M.Barnes@kent.ac.uk
Contents

- Motivation
- The new occam-π compiler (fringe presentation)
- Compiling CSP
- Interleaving multiway synchronisations
- Generating code
- Conclusions and further work
**Motivation**

**CSP**, Hoare’s Communicating Sequential Processes, is a process algebra for describing **concurrent** processes and their interactions

- CSP itself primarily used for **formal modelling**
- e.g. with tools such as **FDR** and **ProBE**

Can describe some interesting and complex systems with CSP

- including some that we cannot yet implement directly
- e.g. with tools such as **KRoC/occam-π**, **JCSP**, **C++CSP**, **CTJ**, etc.

This work is concerned with the **compilation** of CSP to executable code

- so that we can experiment with interesting and complex systems  :-
- including the **TUNA** project’s models of platelet behaviour (investigating models of blood-clotting and, more generally, **nanite assemblers**)
The New occam-\(\pi\) Compiler

A new occam-\(\pi\) compiler to replace the existing compiler in KRoC

- the existing compiler is becoming increasingly difficult to maintain
- based on a fairly old (but industry proven) code base, mostly 1987
- designed to run in 2 MB of memory, so quite compact/optimal in places
- but was never really designed to handle the dynamics introduced by occam-\(\pi\)
- written in C, started off around 60,000 lines, now at around 120,000

Currently around 55,000 lines of C code, named NOCC

- maybe not the best language for implementing compilers ...
- and do we really need another compiler ?
- on the other hand, few compilers have low-level representations for parallelism (mostly in compilers for parallel hardware)
The CSP language implemented does not require a hugely complex compiler
- good test of NOCC’s ability to handle different source languages
- NOCC already generates ETC (virtual transputer byte-code), translated to
  native code and linked with the existing KRoC/occam-π run-time

An extensible monolithic multi-pass compiler:

(new passes can be added at run-time)
The New occam-π Compiler

When it starts up, the compiler is ‘empty’

- Parse tree structures and the parser built dynamically:

```c
tndef_t *tnd; ntdef_t *tag;
tnd = tnode_newnodetype("mcsp:scopenode", [2, 0, 0], TNFNONE);
tag = tnode_newnodetag("MCSPFIXPOINT", tnd, NTFNONE);
```

- Parser structures are described using a BNF or DFA notation:

```c
dynarray_add (transtbl, dfa_bnftotbl("mcsp:eventset ::= "
"( mcsp:event | @@{ { mcsp:event @@, 1 } @@} )"));
dynarray_add (transtbl, dfa_dfatotbl("mcsp:fixpoint ::= "
"[ 0 @@ 1 ] [ 1 mcsp:name 2 ] [ 2 @@. 3 ]"
"[ 3 mcsp:process 4 ] [ 4 {<mcsp:fixreduce>} -* ]"));
```

Fred Barnes, September 2006
The New occam-π Compiler

The compiler will typically need to process **several hundred** of these rules
- takes an insignificant amount of time — efficient implementation  :-(
- currently fed from constant strings in C function calls, will use a text file in the future — real-time compiler compiler

Rules already set can be augmented by language features (plug-in modules):

```c
dynarray_add (transtbl, dfa_dfatotbl ("occampi:process +:= "
  "[ 0 @SYNC 1 ] [ 1 occampi:operand 2 ] "
  "[ 2 { opi:syncreduce } -* ]"));
```

Compiler collects up DFA chunks, in **tables** and merges
- later resolution of **sub-parses** (branches out of the DFA)
The New occam-π Compiler

The reductions \{<mcsp:fixreduce>\} and \{<occampi:syncreduce>\} are pre-registered generic reductions:

```
parser_register_grule ("occampi:syncreduce",
   parser_decode_grule ("SN0N+C1R-", opi.tag_SYNC));
```

The ‘program’ is for a small stack machine which can manipulate the DFA state:
- can also make calls to C functions for more complex reductions

Eventually all soaked up from a text file:

```plaintext
keyword "SYNC"
nodetype occampi:actionnode 3,0,0  # LHS, R
   nodetag occampi:sync "occampi:actionnode"
reduce opi:syncreduce "SN0N+00C[occampi:sync]3R-"
dfarule occampi:process {
   0: "@SYNC" -> 1
   1: "occampi:operand" -> 2
   2: "<opi:syncreduce>" "-*" -> return
}
```

# incase things weren’t getting silly yet:
keyword "bnfrule"
nodetype nocc:bnfrulenode 2,0,0
   nodetag nocc:bnfrule "nocc:bnfrulenode"
dfarule nocc:compilerdef {
   0: "@bnfrule" -> 1
   1: "+Name" -> 2
   2: "+String" -> 3
   3: "Newline" -> return
   3: cfunc ("noccparser_bnfreduce")
```
Once all the DFAs are set up (choice may depend on language being parsed), language-specific code will parse input with, e.g.:

```
parsetree = dfa_walk (lf, "occampi:declorprocstart");
```

(‘lf’ is a reference to a lexer which provides the tokens)

The **DFA engine** is what walks round the DFAs, using tokens from the lexer and maintaining a stack of **DFA states**

- tokens can be pushed back into the lexer — useful for occam-π, which requires up to 3 look-aheads to determine what is being parsed
Knowing a bit about how the DFA engine operates helps to make sense of the language definitions:

- push token onto the parser’s token stack
- reduce with the generic reduction ‘rfoo’
- push token back into the lexer
- match all initial matches from DFA rule ‘foo’

The frequently occurring ‘-*’ means match anything and push back into the lexer

- the ‘any’ match is special in that it is always tested last – default match

DFA edges (matched transitions) with no target pop the DFA state

Parser for a null language:

```
mylang ::= [ 0 * 0 ] [ 0 End ]
```
The New occam-$\pi$ Compiler

- The end result of the parser is a **parse tree**
  - this is then transformed by each pass of the compiler in turn
- Code that implements a language front-end can attach C functions to each pass
- Because the whole thing hangs together using structures containing function pointers, easy for code to intercept these and selectively override
  - not entirely unlike **aspect orientation**, albeit quite explicit
  - e.g. code for the occam-$\pi$ multiway synchronisation interferes with `occampi:actionnode`, handling 'SYNC' only and passing everything else along to whatever else was there before
- Last interesting pass for most of a language front-end is **name-map**, which inserts back-end specific nodes into the tree
The syntax used is not quite the same as that used by FDR, but this may be changed later on:

<table>
<thead>
<tr>
<th>CSP</th>
<th>MCSP</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>SKIP</td>
</tr>
<tr>
<td>stop</td>
<td>STOP</td>
</tr>
<tr>
<td>chaos</td>
<td>CHAOS</td>
</tr>
<tr>
<td>divergence</td>
<td>div</td>
</tr>
<tr>
<td>event prefix</td>
<td>e \rightarrow P</td>
</tr>
<tr>
<td>internal choice</td>
<td>(x \rightarrow P) \sqcap (y \rightarrow Q)</td>
</tr>
<tr>
<td>external choice</td>
<td>(x \rightarrow P) \sqcup (y \rightarrow Q)</td>
</tr>
<tr>
<td>sequence</td>
<td>P \circ Q</td>
</tr>
<tr>
<td>parallel</td>
<td>P \parallel Q</td>
</tr>
<tr>
<td>interleaving</td>
<td>P \parallel\parallel Q</td>
</tr>
<tr>
<td>hiding</td>
<td>P \setminus {a}</td>
</tr>
<tr>
<td>fixpoint</td>
<td>\mu X.P</td>
</tr>
</tbody>
</table>

Fred Barnes, September 2006
Using all the DFA paraphernalia, the compiler turns MCSP input into parse trees

- at the top-level, named process definitions are expected:

```
P ::= s -> SKIP
Q (e) ::= ((e -> P) [] (f -> Q(e))) \ {f}
FOO (x) ::= x -> x -> Q ; STOP
```

- As with occam-π, the last process definition is used for the ‘top-level’ process

- in parallel with this, the compiler generates an ‘environment’ process. Slightly artificial, but required to produce output:

```
ENVIRONMENT (out,x) ::= @z.(x -> out!"x*n" -> z)
SYSTEM (screen) ::= (FOO (k) || ENVIRONMENT (screen,k)) \ {k}
```

- Unbound events are captured by parameters, e.g.:

```
P (P_s) ::= P_s -> SKIP
Q (Q_P_s, e) ::= ((e -> P(Q_P_s)) [] (f -> Q(Q_P_s,e))) \ {f}
```
Most of the required run-time mechanisms for executing CSP programs are already present in the KRoC system

- no support for **interleaving** multiway synchronisations

```plaintext
BISCUIT (coin) ::= coin -> biscuit -> SKIP
CHOC (coin) ::= coin -> choc -> SKIP
MACHINE (coin) ::= @x.((BISCUIT(coin) ||| CHOC(coin)) -> x)
```

Any multiway synchronisation (or part thereof) falls into 1 of 3 categories:

- **1 of N**: the CSP model (strictly speaking \(\|\|\|\) is a binary operator, \(N = 2\), but NOCC will flatten nested interleaving)
- **M of N**: where \(1 < M < N\), useful for some implementations (aside later)
- **N of N**: full synchronisation (typical occam-\(\pi\) ‘BARrier’ type)

Only really need to support 1-of-N and N-of-N for MCSP, but would like the other cases for occam-\(\pi\) — same underlying implementation
Interleaving Multiway Synchronisations

- Support for these have now (just!) been added to the run-time
  - based on the non-interleaving version presented by Welch
  - only used by the occam-π front-end in NOCC, but expect MCSP to be using it before long — provided as a **language independent** feature

- Instead of having a single queue of blocked (**waiting-to-sync**) processes, groups processes into sets with **enroll**, **sync**, **down** counts and a queue
  - top-level barrier structure counts the number of enrolled sets and the number of sets left to synchronise
  - also a small structure associated with each synchronising process

- In most cases will have up to two levels of synchronisation
  - synchronisation completed in one of the **sets**
  - synchronisation completed at the top-level

Fred Barnes, September 2006
Compiler allocates structures in process workspaces in the **mwsynctrans** pass

- new set of instructions in the run-time to manage these

- **par-barrier**
  - procbar_t *parent
  - int sets_enrolled
  - int sets_downcount
  - parbar_t *set_fptr
  - parbar_t *set_bptr
  - int alt_count

- **proc-barrier**
  - procbar_t *q_next
  - procbar_t *q_prev
  - parbar_t *pbar_link
  - uint *wptr
  - int flags

- **interleave sub-barrier**
  - alt
  - alt-select

- sync-count specifies how many processes required to offer in this set

- down-count may drop below zero, indicating surplus; does sync as it goes past

Still uses a global lock around **ALTs** — to make sure that affected disabling sequences complete before more multiway synchronisations start

Certain cases of interleaving require nesting of these (or do they...)

- only when interleaving processes go sub-parallel or sub-interleave
Interleaving Multiway Synchronisations

- Slightly non-trivial implementation, but can be reasoned about in pictures:

\[ \text{BARRIER } b: \]
\[ \text{PAR} \]
\[ P \ (b) \]
\[ Q \ (b) \]

1, 0 → 2, 2, 0

P synchronises first
then Q synchronises, completing the local barrier
which then completes the top-level barrier

- Completed synchronisation resets top-level count, and in each synchronised barrier set, adds sync-count to down-count:

2, 2 → 2, 1, 1 → 3, 3, 3

- Sub-parallelism (or interleaving) creates a logically upside-down tree; if P goes parallel with 3 sub-processes, its own is resigned
Interleaving Multiway Synchronisations

If the other branch (Q) goes parallel simultaneously:

- nothing left in the original set (sync-count reached zero), so it is **resigned** from the top-level
- still 2 processes enrolled, so it doesn’t go away entirely

When one of the parallel sub-processes shuts down, **re-enrolled** in its parent set

- essentially the reverse process to setting up parallel processes, except that for occam-π (not MCSP) individual process ‘PB’s resign when the process terminates, not after the ‘PAR’ (can be overridden with a compiler flag)

Implementation currently leaves the disabled set attached to the linked-list of sets, could remove it if we wanted ...
Interleaving Multiway Synchronisations

» Straight-forward interleaving (1-of-N) is handled by fixing the sync-count at 1:

(P (b) ||| Q (b)) \ {b}

1, 0 ➽ 2, 1, 0

» First process to synchronise will complete the barrier

» This works fine, provided that the interleaving sub-processes (P and Q) do not themselves go parallel or interleave

• can have any amount of parallelism ‘above’ interleaving, e.g.:

3, 2 ➽ 3, 1, 1 ➽ 3, 1, -1 ➽ 2, 2, 1

» Two sets left to synchronise

» When complete, only one of the interleaving processes will be resumed (queue implementation provides fairness); set remains ready
This mechanism breaks down when interleaving processes go parallel, e.g.:

\[
Q(b) ::= R(b) \parallel S(b)
\]

\[
(P(b) \parallel Q(b) \parallel R(b)) \setminus \{b\}
\]

The existing route would see sync-count at zero and disable the set

- problem arises when one of the interleaving processes synchronises

Set is effectively inactive, so processes here won’t be rescheduled — and don’t know how to reset down-count (because sync-count now zero)

Also the top-level sync may never occur because down-count is already at zero

An early thought at a solution was to introduce a missing-count, separate to sync-count, but this breaks down in M-of-N interleaving
Interleaving Multiway Synchronisations

- One functional solution is to use nested barriers, as is almost supported:

- Problem with this solution is the run-time overhead — **proc-barrier** operations will need to test for sub-barriers and handle accordingly

- There may be a better solution, but haven’t found it yet..
The occam-π front-end in NOCC uses this mechanism to implement its BARRIER functionality.

Provides a nice solution to the classic santa-claus problem, still thinking about the language binding:

```
PROC santa (BARRIER elves, reindeer)
  WHILE TRUE
    PRI ALT
      elves
        ... meet with elves
      reindeer
        ... go deliver presents
```

Some outstanding issues relating to the sync-count when processes resign in-par — e.g. when there are only 2 elves left, are they allowed to meet with santa?
Once the parse tree has been built, the rest is mostly tree-transformations.

The compiler’s `target_t` structure defines various back-end specific nodes:

- inserted during the `name-map` pass

Starting with the code fragment `x -> y -> SKIP`:

```
then
  name
    sync
      nameref
        nparam
          "x"
then
  name
    sync
      nameref
        nparam
          "x"
ndecl
  "y"
```

```
then
  name
    sync
      nameref
        nparam
          "x"
then
  name
    sync
      nameref
        nparam
          "x"
ndecl
  "y"

workspace
ptr "x" +24
ptr "y" +20
reserved
Wptr (+0)
```
Generating Code

The compiler produces virtual transputer assembler as output, e.g.:

```
; PROCESS CONSUME = 44,28,12
.setws 44, 12
.setvs 0
.setms 0
.setnamedlabel "O_CONSUME"
.procentry "CONSUME"
.setlabel 7
 ajw -16
.setlabel 59
 ldc  1
 stl  8
 ldc 1000000
 adc  -1
 stl  12
 ...
```

(all offsets/sizes in bytes to avoid word-size confusions)

Eaten up by **tranx86**, assembled and linked with the **CCSP** runtime

Compiler also produces meta-data files (for use with separate compilation, etc.)
Performance

Have an MCSP version of the commstime benchmark:

- actually measuring the multiway synchronisation time

```
PREFIX (in,out) ::= out -> @x.(in -> out -> x)
SUCC (in,out) ::= @x.(in -> out -> x)
DELTA (in,out1,out2) ::= @x.(in -> out1 -> out2 -> x)
CONSUME (in,report) ::= @x.((;[i=1,1000000] in); report -> x)

COMMSTIME (report) ::= ((PREFIX (a,b) || DELTA (b,c,d)) ||
(SUCC (c,a) || CONSUME (d,report))) \ {a,b,c,d}
```

Because there are currently no timer facilities, have to rely on the time between ‘report’ outputs (every million cycles)

- on a 2.4 GHz P4, time for a complete synchronisation with 2 process is approximately 250 nanoseconds (syncs implemented as single-guard ALTs) (using Welch’s algorithm with dynamic wait-queue allocation)
Conclusions and Further Work

The compiler currently manages most **simple** MCSP programs
- some features still not implemented: replicated parallel/interleaving, alphabetised parallel, variables/expressions, interrupts, timers
- and some restrictions: **no** self/mutual recursion, **no** non-tail-call fixpoints

The MCSP specific part of NOCC weighs in at around **6,000** lines of C code
- not huge considering, and relatively easy to maintain

Items for future consideration:
- different **environments** — e.g. for graphical visualisations
- adjustment of the syntax for FDR compatibility