CSP_M: A Reference Manual

Bryan Scattergood, Philip Armstrong

January 24, 2011

1 Introduction

The machine-readable dialect of CSP (CSP_M) is one result of a research effort [1] with the primary aim of encouraging the creation of tools for CSP. FDR was the first tool to utilise the dialect, and to some extent FDR and CSP_M continue to evolve in parallel, but the basic research results are publicly available (see later for more details). The language described here is that implemented by the 2.91 release of FDR.

CSP_M combines the CSP process algebra with an expression language which, while inspired by languages like Miranda/Orwell and Haskell/Gofer, has been modified to support the idioms of CSP. The fundamental features of those languages are, however, retained: the lack of any notion of assignment, the ability to treat functions as first-class objects, and a lazy reduction strategy.

Scripts

Programming languages are used to describe algorithms in a form which can be executed. CSP_M includes a functional programming language, but its primary purpose is different: it is there to support the description of parallel systems in a form which can be automatically manipulated. CSP_M scripts should, therefore, be regarded as defining a number of processes rather than a program in the usual sense.

[1] This Appendix was written by Bryan Scattergood, of Formal Systems (Europe) Ltd. He is the main developer and implementor of this version of CSP. It has been updated for the latest CSP_M release by Philip Armstrong. Comments and queries about the notation, and potential tool developers who wish to use these results, should contact him by email: `philip.armstrong@comlab.ox.ac.uk`.

2 Expressions

At a basic level, a CSP$_M$ script defines processes, along with supporting functions and expressions. CSP draws freely on mathematics for these supporting terms, so the CSP$_M$ expression-language is rich and includes direct support for sequences, sets, booleans, tuples, user-defined types, local definitions, pattern matching and lambda terms.

We will use the following variables to stand for expressions of various types.

m, n	numbers
s, t	sequences
a, A	sets (the latter a set of sets)
b	boolean
p, q	processes
e	events
c	channel
x	general expression

When writing out equivalences, `z` and `z'` are assumed to be fresh variables which do not introduce conflicts with the surrounding expressions.

Identifiers

Identifiers in CSP$_M$ begin with an alphabetic character and are followed by any number of alphanumeric characters or underscores optionally followed by any number of prime characters (`'`). There is no limit on the length of identifiers and case is significant. Identifiers with a trailing underscore (such as `fnargle_`) are reserved for machine-generated code such as that produced by Casper [**?**].

CSP$_M$ enforces no restrictions on the use of upper/lower-case letters in identifiers (unlike some functional languages where only data type constructors can have initial capital letters.) It is, however, common for users to adopt some convention on the use of identifiers. For example

- Processes all in capitals (`BUTTON`, `ELEVATOR_TWO`)
- Types and type constructors with initial capitals (`User`, `Dial`, `DropLine`)
- Functions and channels all in lower-case (`sum`, `reverse`, `in`, `out`, `open_door`)

Note that while it is reasonable to use single character identifiers (`P`, `c`, `T`) for small illustrative examples, real scripts should use longer and more descriptive names.

Numbers

Syntax

`12`	integer literal
m`+`n, m`-`n	sum and difference
`-`m	unary minus
m`*`n	product
m`/`n, m`%`n	quotient and remainder

Remarks

Integer arithmetic is defined to support values between -2147483647 and 2147483647 inclusive, that is those numbers representable by an underlying 32-bit representation (either signed or twos-complement.) The effect of overflow is not defined: it may produce an error, or it may silently wrap in unpredictable ways and so should not be relied upon.

The division and remainder operations are defined so that, for $n \neq 0$,

$$\begin{aligned} m &= n * (m/n) + m\%n \\ \mid m\%n \mid &< \mid n \mid \\ m\%n &\geqslant 0 \text{ (provided } n > 0) \end{aligned}$$

so that, for positive divisors, division rounds down and the remainder operation yields a positive result.

Floating point numbers (introduced experimentally for Pravda [**?**]) are not currently supported by FDR. Although the syntax for them is still enabled, it is not documented here.

Sequences

Syntax

`<>`, `<1,2,3>`	sequence literals
`<`m`..`n`>`	closed range (from integer m to n inclusive)
`<`m`..>`	open range (from integer m upwards)
s`^`t	sequence catenation
`#`s, `length(`s`)`	length of a sequence
`null(`s`)`	test if a sequence is empty
`head(`s`)`	the first element of a non-empty sequence
`tail(`s`)`	all but the first element of a non-empty sequence
`concat(`s`)`	join together a sequence of sequences
`elem(`x`,`s`)`	test if an element occurs in a sequence
`<`x_1`,...,` x_n`|` x`<-`s`,` b`>`	comprehension

Equivalences

$$
\begin{aligned}
\texttt{null(}s\texttt{)} &\equiv s\texttt{==<>} \\
\texttt{<}m\texttt{..}n\texttt{>} &\equiv \texttt{if } m\texttt{<=}n\texttt{ then <}m\texttt{>\^{}<}m\texttt{+1..}n\texttt{> else <>} \\
\texttt{elem(}x\texttt{,}s\texttt{)} &\equiv \texttt{not null(< z | z<-}s\texttt{, z==}x\texttt{ >)} \\
\texttt{< }x\texttt{ | >} &\equiv \texttt{< }x\texttt{ >} \\
\texttt{< }x\texttt{ | }b\texttt{, ...>} &\equiv \texttt{if }b\texttt{ then < }x\texttt{ | ...> else <>} \\
\texttt{< }x\texttt{ | }x'\texttt{<-}s\texttt{, ...>} &\equiv \texttt{concat(< < }x\texttt{ | ...> | }x'\texttt{<-}s\texttt{ >)}
\end{aligned}
$$

Remarks

All the elements of a sequence must have the same type. `concat` and `elem` behave as if defined by

```
concat(s)      = if null(s) then <> else head(s)^concat(tail(s))
elem(_, <>)    = false
elem(e, <x>^s) = e==x or elem(e,s)
```

The following function tests if a sequence reads the same forwards and backwards

```
palindrome(<x>^s^<y>) = x==y and palindrome(s)
palindrome(_)         = true
```

Sets

Syntax

`{1,2,3}`	set literal
{m`..`n`}`	closed range (between integers m and n inclusive)
{m`..}`	open range (from integer m upwards)
`union(`a_1`,`a_2`)`	set union
`inter(`a_1`,`a_2`)`	set intersection
`diff(`a_1`,`a_2`)`	set difference
`Union(`A`)`	distributed union
`Inter(`A`)`	distributed intersection (A must be non-empty)
`member(`x`,`a`)`	membership test
`card(`a`)`	cardinality (count elements)
`empty(`a`)`	check for empty set
`set(`s`)`	convert a sequence to a set
`Set(`a`)`	all subsets of a (powerset construction)
`seq(`s`)`	convert a set to a sequence (in an arbitrary order)
`Seq(`a`)`	set of sequences over a (infinite if a is not empty)
{x_1`,...,` x_n`|` x`<-`a`,` b`}`	comprehension

Equivalences

`union(`a_1`,`a_2`)`	≡	`{ z,z'	z<-`$a_1$`, z'<-`$a_2$ `}`
`inter(`a_1`,`a_2`)`	≡	`{ z	z<-`$a_1$`, member(z,`$a_2$`) }`
`diff(`a_1`,`a_2`)`	≡	`{ z	z<-`$a_1$`, not member(z,`$a_2$`) }`
`Union(`A`)`	≡	`{ z	z'<-`$A$`, z<-z' }`
`member(`x`,`a`)`	≡	`not empty({ z | z<-`a`, z==`x `})`
`Seq(`a`)`	≡	`union({<>}, {<z>^z'	z<-`$a$`, z'<-Seq(`$a$`)})`
`{` x `|}`	≡	`{` x `}`
`{` x `|` b`, ...}`	≡	`if` b `then {` x `| ...} else {}`
`{` x `|` x'`<-`a`, ...}`	≡	`Union( { {` x `| ...} |` x'`<-`a `} )`

Remarks

In order to remove duplicates, sets need to compare their elements for equality, so only those types where equality is defined may be placed in sets. In particular, sets of processes are not permitted. See the section on pattern matching for an example of how to convert a set into a sequence by sorting.

Alternatively, the seq() function will create a sequence from a set in some arbitrary order.

Sets of negative numbers (`{ -2}`) require a space between the opening bracket and minus sign to prevent it being confused with block comment.

Booleans

Syntax

`true`, `false`	boolean literals
b_1 `and` b_2	boolean and (shortcut)
b_1 `or` b_2	boolean or (shortcut)
`not` b	boolean not
x_1`==`x_2, x_1`!=`x_2	equality operations
x_1`<`x_2, x_1`>`x_2, x_1`<=`x_2, x_1`>=`x_2	ordering operations
`if` b `then` x_1 `else` x_2	conditional expression

Equivalences

$$
\begin{aligned}
b_1 \texttt{ and } b_2 &\equiv \texttt{if } b_1 \texttt{ then } b_2 \texttt{ else false} \\
b_1 \texttt{ or } b_2 &\equiv \texttt{if } b_1 \texttt{ then true else } b_2 \\
\texttt{not } b &\equiv \texttt{if } b \texttt{ then false else true}
\end{aligned}
$$

Remarks

Equality operations are defined on all types except those containing processes and functions (lambda terms).

Ordering operations are defined on sets, sequences and tuples as follows

$$
\begin{aligned}
x_1 \texttt{ >= } x_2 &\equiv x_2 \texttt{ <= } x_1 \\
x_1 \texttt{ < } x_2 &\equiv x_1 \texttt{ <= } x_2 \texttt{ and } x_1 \texttt{ != } x_2 \\
a_1 \texttt{ <= } a_2 &\equiv a_1 \text{ is a subset of } a_2 \\
s_1 \texttt{ <= } s_2 &\equiv s_1 \text{ is a prefix of } s_2 \\
\texttt{(}x_1\texttt{,}y_1\texttt{) <= (}x_2\texttt{,}y_2\texttt{)} &\equiv x_1 \texttt{ < } x_2 \texttt{ or (}x_1 \texttt{ == } x_2 \texttt{ and } y_1 \texttt{ <= } y_2\texttt{)}
\end{aligned}
$$

Ordering operations are not defined on booleans or user-defined types.

```
if b then {1} else <2>
```

is an error.

A standalone type checker for CSP$_M$is available which will catch this kind of error prior to compilation. It is not currently bundled with FDR2 as some CSP$_M$ scripts have infinite types that the type checker is unable to check in finite time. The type checker can still be very useful for debugging the majority of CSP$_M$ scripts and its use is strongly advised.

Tuples

Syntax

`(1,2), (4,<>,{7})` pair and triple

Remarks

Function application also uses parentheses, so functions which take a tuple as their argument need two sets of parentheses. For example the function which adds together the elements of a pair can be written either as

```
plus((x,y)) = x+y
```

or as

```
plus(p) = let (x,y) = p within x + y
```

The same notation is used in type definitions to denote the corresponding product type. For example, if we have

```
nametype T = ({0..2},{1,3})
```

then `T` is

```
{ (0,1), (0,3), (1,1), (1,3), (2,1), (2,3) }
```

Local definitions

Definitions can be made local to an expression by enclosing them in a '`let within`' clause.

```
primes =
  let
    factors(n)  = < m | m <- <2..n-1>, n%m == 0 >
    is_prime(n) = null(factors(n))
  within < n | n <- <2..>, is_prime(n) >
```

Local definitions are mutually recursive, just like top-level definitions. Not all definitions can be scoped in this way: channel and datatype definitions are only permitted at the top-level. Transparent definitions can be localized, and this can be used to import FDR's compression operations on a selective basis. For example,

```
my_compress(p) =
  let
    transparent normal, diamond
  within normal(diamond(p))
```

Lambda terms

Syntax

`\` x_1`,` `...`x_n `@` x lambda term (nameless function)

Equivalences

The definition

```
f(x,y,z) = x+y+z
```

is equivalent to the definition

```
f = \ x, y, z @ x+y+z
```

Remarks

There is no direct way of defining an anonymous function with multiple branches. The same effect can be achieved by using a local definition and the above equivalence. Functions can both take functions as arguments and return them as results.

```
map(f)(s) = < f(x) | x <- s >
twice(n)  = n*2
assert map(\ n @ n+1)(<3,7,2>) == <4,8,3>
assert map(map(twice))(< <9,2>, <1> >) == < <18,4>, <2> >
```

3 Pattern matching

Many of the above examples made use of pattern matching to decompose values. For example, we can write

```
reverse(<>)    = <>
reverse(<x>^s) = reverse(s)^<x>
```

as well as

```
reverse(s) = if null(s) then <> else reverse(tail(s)) ^ <head(s)>
```

The branches of a function definition must be adjacent in the script, otherwise the function name will be reported as multiply defined.

Patterns can occur in many places within CSP$_M$ scripts

- Function definitions (`reverse` above)
- Direct definitions `(x,y) = (7,2)`
- Comprehensions `{ x+y | (x,y) <- {(1,2),(2,3)} }`
- Replicated operators `||| (x,y):{(1,2),(2,3)} @ c!x+y->STOP`
- Communications `d?(x,y)->c!x+y->STOP`

The patterns which are handled in these cases are the same, but the behaviour in the first two cases is different. During comprehensions, replicated operators and communications we can simply discard values which fail to match the pattern: we have a number of such values to consider so this is natural. When a function fails to match its argument (or a definition its value) silently ignoring it is not an option so an error is raised. On the other hand, functions can have multiple branches (as in the case of `reverse`) which are tried in top to bottom order while the other constructs only allow a single pattern. For example,

```
f(0,x) = x
f(1,x) = x+1
print f(1,2) -- gives 3
print f(2,1) -- gives an error
print { x+1 | (1,x) <- { (1,2), (2,7) } } -- gives {3}
```

The space of patterns is defined by

1. Integer literals match only the corresponding numeric value.
2. Underscore (`_`) always matches.
3. An identifier always matches, binding the identifier to the value.

4. A tuple of patterns is a pattern matching tuples of the same size. Attempting to match tuples of a different size is an error rather than a match failure.

5. A simple sequence of patterns is a pattern (`<x,y,z>`) matching sequences of that length.

6. The catenation of two patterns is a pattern matching a sequence which is long enough, provided at least one of the sub-patterns has a fixed length.

7. The empty set is a pattern matching only empty sets.

8. A singleton set of a pattern is a pattern matching sets with one element.

9. A data type tag (or channel name) is a pattern matching only that tag.

10. The dot of two patterns is a pattern. (`A.x`)

11. The combination of two patterns using `@@` is a pattern which matches a value only when both patterns do.

12. A pattern may not contain any identifier more than once.

For example, `{}`, `({x},{y})` and `<x,y>^_^<u,v>` are valid patterns. However, `{x,y}` and `<x>^s^t` are not valid patterns since the decomposition of the value matched is not uniquely defined. Also `(x,x)` is not a valid pattern by rule 12: the effect that this achieves in some functional languages requires an explicit equality check in CSP$_M$.

When a pattern matches a value, all of the (non-tag) identifiers in the pattern are bound to the corresponding part of the value.

The fact that tags are treated as patterns rather than identifiers can cause confusion if common identifiers are used as tags. For example, given

```
channel n : {0..9}
f(n) = n+1
```

attempting to evaluate the expression `f(3)` will report that the function `\ n @ n+1` does not accept the value 3. (It accepts *only* the tag `n`.)

Only names defined as tags are special when used for pattern matching. For example, given

```
datatype T = A | B
x = A
f(x) = 0
f(_) = 1
g(A) = 0
g(_) = 1
```

then `f` is not the same as `g` since `f(B)` is `0` while `g(B)` is `1`.

The singleton-set pattern allows us to define the function which picks the unique element from a set as

```
pick({x}) = x
```

This function is surprisingly powerful. For example, it allows us to define a `sort` function from sets to sequences.

```
sort(f,a) =
  let
    below(x)  = card( { y | y<-a, f(y,x) } )
    pairs     = { (x, below(x)) | x <- a }
    select(i) = pick({ x | (x,n)<-pairs, i==n })
  within < select(i) | i <-<1..card(a)> >
```

where the first argument represents a `<=` relation on the elements of the second. Because `pick` works only when presented with the singleton set, the `sort` function is defined only when the function `f` provides a total ordering on the set `a`.

4 Types

Simple types

Types are associated at a fundamental level with the set of elements that the type contains. Type expressions can occur only as part of the definition of channels or other types, but the name of a type can be used anywhere that a set is required. The fundamental types supported by the CSP$_M$ interpreter are `Int`, the type of integer values; `Bool`, the type of boolean values and `Proc`, the type of processes. So

```
{0..3} <= Int
{true, false} == Bool
```

Processes can be constituents of datatypes (see below) and sequences, but not sets since they are incomparable.

In type expressions the tuple syntax denotes a product type and the dot operation denotes a composite type so that

`({0,1},{2,3})`	denotes	`{(0,2),(0,3),(1,2),(1,3)}`
`{0,1}.{2,3}`	denotes	`{0.2, 0.3, 1.2, 1.3}`

The `Set` and `Seq` functions which return the powerset and sequence space of their arguments are also useful in type expressions.

Named types

Nametype definitions associate a name with a type expression, meaning that '`.`' and '`( , , )`' operate on it as type constructors rather than value expressions. For example,

```
nametype Values = {0..199}
nametype Ranges = Values . Values
```

has the same effect as

```
Values = {0..199}
Ranges = { x.y | x<-Values, y<-Values }
```

If, on the other hand, we had left `Values` as an ordinary set, `Values . Values` would have had the entirely different meaning of two copies of the set `Values` joined by the infix dot. Similarly the expression `(Values,Values)` means *either* the Cartesian product of `Values` with itself *or* a pair of two sets depending on the same distinction.

Data types

Syntax

`datatype T = A.{0..3} | B.Set({0,1}) | C`	definition of type
`A.0`, `B.{0}`, `B.{0,1}`, `C`	four uses of type

Remarks

Data types may not be parameterized (`T` may not have arguments).

The `datatype` corresponds to the variant-record construct of languages like Pascal. At the simplest level it can be used to define a number of atomic constants

```
datatype SimpleColour = Red | Green | Blue
```

but values can also be associated with the tags

```
Gun = {0..15}
datatype ComplexColour = RGB.Gun.Gun.Gun | Grey.Gun | Black | White
```

Values are combined with '.' and labelled using the appropriate tag, so that we could write

```
make_colour((r.g.b)@@x) =
  if r!=g or g!=b then RGB.x else
  if r==0 then Black else
  if r==15 then White else Grey.r
```

to encode a colour as briefly as possible.

Note that while it is possible to write

```
datatype SlowComplexCol = RGB.{r.g.b | r<-Gun, g<-Gun, b<-Gun} | ...
```

this is less efficient and the resulting type must still be rectangular, that is expressible as a simple product type. Hence it is *not* legal to write

```
datatype BrokenComplexColour = -- NOT RECTANGULAR
  RGB.{r.g.b | r<-Gun, g<-Gun, b<-Gun, r+g+b < 128 } | ...
```

Channels

Syntax

`channel flip, flop`	simple channels
`channel c, d :  {0..3}.LEVEL`	channels with more complex protocol
`Events`	the type of all defined events

Remarks

Channels are tags which form the basis for events. A channel becomes an event when enough values have been supplied to complete it (for example `flop` above is an event). In the same way, given

```
datatype T = A.{0..3} | ...
```

we know that `A.1` is a value of type `T`, given

```
channel c : {0..3}
```

we know that `c.1` is a value of type Event. Indeed, the channel definitions in a script can be regarded as a distributed definition for the built-in `Events` data type.

Channels must also be rectangular in the same sense as used for data types. It is common in FDR2 to make channels finite although it is possible to declare infinite channels and use only a finite proportion of them.

Channels interact naturally with data types to give the functionality provided by variant channels in occam2 (and channels of variants in occam3.) For example, given `ComplexColour` as above, we can write a process which strips out the redundant colour encodings (undoing the work performed by `make_colour`)

```
channel colour : ComplexColour
channel standard : Gun.Gun.Gun

Standardize =
    colour.RGB?x -> standard!x -> Standardize
  []
    colour.Grey?x -> standard!x.x.x -> Standardize
  []
    colour.Black -> standard!0.0.0 -> Standardize
  []
    colour.White -> standard!15.15.15 -> Standardize
```

It is not possible to communicate a value containing a Process over a channel.

Closure operations

Syntax

`extensions(`x`)`	The set of values which will 'complete' x
`productions(`x`)`	The set of values which begin with x
`{|`x_1`,`x_2`|}`	The productions of x_1 and x_2

Equivalences

`productions(`x`)` $\equiv$ `{ `x`.z | z<-extensions(`x`) }`

`{\|`x `| ...\|}` $\equiv$ `Union( { productions(`x`) | ...} )`

Remarks

The main use for the `{| |}` syntax is in writing communication sets as part of the various parallel operators. For example, given

```
channel c : {0..9}
P = c!7->SKIP [| {| c |} |] c?x->Q(x)
```

we cannot use `{c}` as the synchronization set; it denotes the singleton set containing the channel `c`, not the set of events associated with that channel.

All of the closure operations can be used on data type values as well as channels. They are defined even when the supplied values are complete. (In that case `extensions` will supply the singleton set consisting of the identity value for the '`.`' operation.)

5 Processes

Syntax

`STOP`	no actions
`SKIP`	successful termination
c`->`p	simple prefix
c`?`x`?`x'`:`a`!`y`->`p	complex prefix
p`;`q	sequential composition
p`/\`q	interrupt
p`\`a	hiding
p`[]`q	external choice
p`|~|`q	internal choice
p`[>`q	untimed time-out
p`[|`a`|>`q	exception
b `&` p	boolean guard
p`[[`a`<-` b`]]`	renaming
p`|||`q	interleaving
p`[|`a`|]`q	sharing
p`[`a`||`a'`]`q	alphabetized parallel
p`[`c`<->`c'`]`q	linked parallel
`;`x`:`s`@`p	replicated sequential composition
`[]`x`:`a`@`p	replicated external choice
`|~|`x`:`a`@`p	replicated internal choice (a must be non-empty)
`|||`x`:`a`@`p	replicated interleave
`[|`a'`|]`x`:`a`@`p	replicated sharing
`||`x`:`a`@[`a'`]`p	replicated alphabetized parallel
`[`c`<->`c'`]`x`:`s`@`p	replicated linked parallel (s must be non-null)

Equivalences

As a consequence of the laws of CSP,

$$
\begin{aligned}
p\texttt{|||}q &\equiv p\texttt{[| \{\} |]}q \\
\texttt{;}x\texttt{:<>@}p &\equiv \texttt{SKIP} \\
\texttt{[]}x\texttt{:\{\}@}p &\equiv \texttt{STOP} \\
\texttt{|||}x\texttt{:\{\}@}p &\equiv \texttt{SKIP} \\
\texttt{[|}a\texttt{|]}x\texttt{:\{\}@}p &\equiv \texttt{SKIP} \\
\texttt{||}x\texttt{:\{\}[}a\texttt{]}p &\equiv \texttt{SKIP}
\end{aligned}
$$

Remarks

The general form of the prefix operator is $c f$`->`p where c is a communication channel, f a number of communication fields and p is the process which is the scope of the prefix. A communication field can be

`!`x	Output
`?`x`:`A	Constrained input
`?`x	Unconstrained input

Fields are processed left to right with the binding produced by any input fields available to any subsequent fields. For example, we can write

```
channel ints : Int.Int
P = ints?x?y:{x-1..x+1} -> SKIP
```

Output fields behave as suggested by the equivalence

$$c \texttt{ !}x\ f \texttt{ -> } p \quad \equiv \quad c.x\ f \texttt{ -> } p$$

The proportion of the channel matched by an input fields is based only on the input pattern. There is no lookahead, so if

```
channel c : {0..9}.{0..9}.Bool
P = c?x!true -> SKIP -- this will not work
Q = c?x.y!true -> SKIP -- but this will
```

then `P` is not correctly defined. The input pattern `x` will match the next complete value from the channel (`{0..9}`) and `true` will then fail to match the next copy of `{0..9}`. In the case of `@@` patterns, the decomposition is based on the left-hand side of the pattern.

If an input occurs as the final communication field it will match any remaining values, as in

```
channel c : Bool.{0..9}.{0..9}
P = c!true?x -> SKIP -- this will work
Q = c!true?x.y -> SKIP -- this will also work
```

This special case allows for the construction of generic buffers.

```
BUFF(in,out) = in?x -> out!x -> BUFF(in, out)
```

is a one place buffer for any pair of channels.

Dots do not directly form part of a prefix: any which do occur are either part of the channel c, or the communication fields. (FDR1 took the approach that dots simply repeated the direction of the preceding communication field. This is a simplification which holds only in the absence of data type tags.)

The guard construct '`b & P`' is a convenient shorthand for

```
if b then P else STOP
```

and is commonly used with the external choice operator (`[]`), as

```
COUNT(lo,n,hi) =
  lo < n & down -> COUNT(lo,n-1,hi)
[]
  n < hi & up -> COUNT(lo,n+1, hi)
```

This exploits the CSP law that p `[]STOP` $= p$.

The linked parallel and renaming operations both use the comprehension syntax for expressing complex linkages and renamings. For example,

```
p [ right.i<->left.((i+1)%n), send<->recv | i<-{0..n-1}] q
p [[ left.i <- left.((i+1)%n), left.0<-send | i<-{0..n-1} ]]
```

Both the links (`c<->c'`) and the renaming pairs (`c<-c'`, read 'becomes') take channels of the same type on each side and extend these pointwise as required. For example

```
p [[ c <- d ]]
```

is defined when `extensions(c)` is the same as `extensions(d)` and is then the same as

```
p [[ c.x <- d.x | x<-extensions(c) ]]
```

The replicated operators allow multiple generators between the operator and the `@` sign in the same way as comprehensions. The terms are evaluated left to right, with the rightmost term varying most quickly. So

```
; x:<1..3>, y:<1..3>, x!=y @ c!x.y->SKIP
```

is the same as

```
c.1.2->c.1.3->c.2.1->c.2.3->c.3.1->c.3.2->SKIP
```

The linked parallel operator generalizes the chaining operator $\gg$. For example, if `COPY` implements a single place buffer,

```
COPY(in,out) =
  in?x -> out!x -> COPY(in,out)
```

then we can implement an n-place buffer by

```
BUFF(n,in,out) =
  [out<->in] i : <1..n> @ COPY(in, out)
```

The precedence rules for operators (both process and expression level) are set out in Table 1. The replicated versions of the process operators have the lowest precedence of all. The `@@` pattern operator has a precedence just below that of function application.

6 Special definitions

External

External definitions are used to enable additional 'magic' functions supported by a specific tool. Requiring a definition, rather than silently inserting names into the initial environment, has two advantages: any dependencies on such functions are made explicit and there is no possibility that users will introduce conflicting definitions without being aware of it. For example, to make use of an (imaginary) `frobnicate` external function, we might say

```
external frobnicate
P(s) = c!frobnicate(s^<0>, 7) -> STOP
```

Without the external definition, `frobnicate` would be reported as an undeclared identifier. Tools should report as an error any attempt to define an external name which they do not recognize.

Transparent

As described in Section **??**, FDR uses a number of operators that are used to reduce the state space or otherwise optimize the underlying representation of a process within the tool. While these could be defined using external definitions, they are required to be semantically neutral. It is thus safe for tools which do not understand the compression operations to ignore them. By defining them as transparent, tools are able to do so; unrecognized external operations would be treated as errors. As an example,

```
transparent diamond, normal
squidge(P) = normal(diamond(P))
```

Class	Operators	Description	Associativity
Application	`f(0)`	function application	
	`[[ <- ]]`	renaming	
Arithmetic	`-`	unary minus	
	`*`, `/`, `%`	multiplication	left
	`+`, `-`	addition	left
Sequence	`^`	catenation	
	`#`	length	
Comparison	`<`, `>`, `<=`, `>=`	ordering	none
	`==`, `!=`	equality	none
Boolean	`not`	negation	
	`and`	conjunction	
	`or`	disjunction	
Sequential	`->`	prefix	
	`&`	guard	
	`;`	sequence	
Choice	`[>`	untimed time-out	
	`/\`	interrupt	
	`[]`	external choice	
	`|~|`	internal choice	
	`[| |>`	exception	
Parallel	`[| |]`, `[ || ]`, `[ <-> ]`,	parallel	none
	`|||`	interleave	
Other	`if then else`	conditional	
	`let within`	local definitions	
	`\ @`	lambda term	

Table 1: *Operator precedence*: the operators at the top of the table bind more tightly than those lower down.

enables the diamond and normal compression operators in FDR2, while other tools see definitions of the identity functions, as if we had written

```
diamond(P) = P
normal(P) = P
squidge(P) = normal(diamond(P))
```

Assert

Assertions are used to state properties which are believed to hold of the other definitions in a script. (FDR1 scripts adopted a convention of defining two processes `SPEC` and `SYSTEM`, with the understanding that the check `SPEC[=SYSTEM` should be performed. This has weaknesses: the correct model for the check is not always apparent, and some scripts require multiple checks.) The most basic form of the definition is

```
assert b
```

where `b` is a boolean expression. For example,

```
primes        = ...
take(0,_)     = <>
take(n,<x>^s) = <x> ^ take(n-1,s)
assert <2,3,5,7,11> == take(5, primes)
```

It is also possible to express refinement checks (typically for use by FDR)

```
assert p [m= q
```

where `p` and `q` are processes and `m` denotes the model (`T`, `F`, `R` pr `V` for traces, failures, refusal testing or revivals models respectvely. Add `D` to any of the models except traces to include divergences in the refinement check.) Note that refinement checks cannot be used in any other context. The (refinement) assertions in a script are used to initialize the list of checks in FDR2.

A refinement check under the tau priority model can be written

```
assert p [= q :[ tau priority ]: s
```

where s is the set of prioritised externally visible events. The tau priority syntax is only supported under the traces model in the current release of FDR2.

Similarly, we have

```
assert p :[ deterministic [FD] ]
assert p :[ deadlock free [F] ]
assert p :[ divergence free ]
```

for the other supported checks within FDR. Only the models `F` and `FD` may be used with the first two, with `FD` assumed if the model is omitted.

Note that process tests cannot be used in any other context. The process assertions in a script are used to initialize the list of checks in FDR2.

Print

Print definitions indicate expressions to be evaluated. The standard tools in the CSP$_M$ distribution include 'check' which evaluates all (non-refinement) assertions and print definitions in a script. This can be useful when debugging problems with scripts. FDR2 uses any print definitions to initialize the list of expressions for the evaluator panel.

7 Mechanics

CSP$_M$ scripts are expressible using the 7-bit ASCII character set (which forms part of all the ISO 8859-x character sets.) While this can make the representation of some operators ugly, it makes it possible to handle the scripts using many existing tools including editors, email systems and web-browsers.

Comments can be embedded within the script using either end-of-line comments preceded by '`--`' or by block comments enclosed inside '`{-`' and '`-}`'. The latter nest, so they can be safely used to comment out sections of a script.

If it is necessary to exploit an existing library of definitions, the '`include`' directive performs a simple textual inclusion of another script file. The directive must start at the beginning of a line and takes a filename enclosed in double quotes. Block comments may not straddle file boundaries (comments cannot be opened in one file and closed in another.)

Definitions within in a script are separated by newlines. Lines may be split before or after any binary token and before any unary token. (There are exceptions to this rule, but they do not occur in practice.)

The `attribute`, `embed`, `module`, `exports`, `endmodule`, `instance` and `subtype` keywords are currently reserved for experimental language features.

8 Availability

The research into tools for CSP has been sponsored by the US Office of Naval Research under N00014-87-J1242 and as such the basic results from that research are freely available on request. It is hoped that this will help encourage a common input syntax between CSP-based tools. The results include the machine-readable form of CSP complete with both denotational and operational semantics, a congruence proof between the two semantic models using a bridging semantics, an implementation of a parser for the language using flex and bison to produce a syntax-tree in C++ and methods defined over that tree which are sufficient to implement the operational semantics.