the `csp` package

Jim Davies

March 2001

1 Introduction

This package allows you to set CSP documents in LaTeX, using the Computer Modern, Lucida Bright, or Times font sets. It uses many features of the `zed` package, which is automatically loaded. This document describes the features that are needed for CSP: for a full account of the package, you should read the companion document—*the `zed` package*—available from the same source.

There are six options to consider:

- `cm`: use Computer Modern/ AMS symbol fonts—the default behaviour;
- `lucida`: use Lucida Bright/ Lucida New Math fonts—license required;
- `times`: use Times/ MathTime fonts—license required;
- `zed`: set displayed mathematics in the style of Z;
- `color`: set mathematical material in different colours;
- `nolines`: set displayed declarations without lines.

The first three are mutually exclusive.

The mathematical language of CSP includes that of Z, so all the Z symbols are automatically defined. However, the `zed` package can do more than this, providing display boxes for definitions, and redefining the standard `\[` and `\]` commands; the `zed` option makes similar provision here.

With the `color` option, the mathematical material will be set in colour:

- `CSPColor`: used for CSP text
- `CSPBoxColor`: used for lines in declaration boxes
- `MetaColor`: used for the fragments of metalanguage: process , channel , let , within , ...

In this document, we begin by explaining the language of mathematical expressions used with CSP; as far as possible, we have tried to make this consistent with the Z notation. In each case, we explain how the expression would be rendered in a machine-readable version of CSP, the language known as $\mathrm{CSP}_{\mathrm{M}}$.

2 Mathematical language

2.1 Propositional Logic

output:	*input:*	*machine:*
$true$	`true`	`true`
$false$	`false`	`false`
$p \land q$	`p \land q`	`p and q`
$p \Rightarrow q$	`p \implies q`	`(not p) or q`
$p \lor q$	`p \lor q`	`p or q`
$\neg\, p$	`\lnot p`	`not p`
$a = b$	`a = b`	`a == b`
$a \neq b$	`a \neq b`	`a != b`

2.2 Arithmetic

output:	*input:*	*machine:*
12	`12`	`12`
-42	`-42`	`-42`
$m + n$	`m + n`	`m + n`
$m - n$	`m - n`	`m - n`
$m * n$	`m * n`	`m * n`
$m \operatorname{div} n$	`m \div n`	`m / n`
$m \bmod n$	`m \mod n`	`m % n`
$m < n$	`m < n`	`m < n`
$m \leq n$	`m \leq n`	`m <= n`
$m > n$	`m > n`	`m > n`
$m \geq n$	`m \geq n`	`m >= n`

2.3 Basic set theory

output:	*input:*	*machine:*
$\emptyset$	`\emptyset`	`{}`
$\{1, 2\}$	`\{1, 2\}`	`{1,2}`
$a \cup b$	`a \cup b`	`union(a,b)`
$a \cap b$	`a \cap b`	`inter(a,b)`
$a \setminus b$	`a \setminus b`	`diff(a,b)`
$\bigcup A$	`\Bigcup A`	`Union(A)`
$\bigcap A$	`\Bigcap A`	`Inter(A)`
$\mathbb{P}\, a$	`\power a`	`Set(a)`
$a \times b$	`a \cross b`	`{(x,y) | x <- a, y <- b}`
$\{\, x : a \mid p \bullet t \,\}$	`\{~x : a | p @ t~\}`	`{t | x <- a, p}`

output:	*input:*	*machine:*
$(1, 2, 3)$	`(1,2,3)`	`(1,2,3)`
$x \in a$	`x \in a`	`member(x,a)`
$x \notin a$	`x \notin a`	`not(member(x,a))`
$\#a$	`\# a`	`card(a)`

Machine-readable CSP has a special operator for testing whether a set is empty: we define a macro for it.

output:	*input:*	*machine:*
$a = \emptyset$	`\Empty a`	`empty(a)`

The comprehension syntax used in machine-readable CSP is based upon the *generator* syntax for list comprehensions in Haskell. The mark-up that we have chosen is a compromise between this and Z, extending the form of comprehension by allowing a comma-separated list of terms in place of the expression term. To emphasise this extension, we use the delimiters `\lset` and `\rset` in place of the usual `\{` and `\}`.

output:	*input:*	*machine:*

$\{x : a \mid p \bullet t, u\}$	`\lset`
	`  x : a | p @ t, u`
	`\rset`
		`{t, u | x <- a}`

The machine-readable expression

```
{t,u | x <- a, y <- b, p}
```

has the same semantics as

$$\bigcup \{ x : a;\ y : b \mid p \bullet \{t, u\} \}$$

and thus could be written as a distributed union:

```
\bigcup \{~x : a; y : b | p @ \{ t, u \}~\}
```

For this reason, we refer to these constructions as *distributed comprehensions*. Similar constructions will be used to define sequences, sets of compound events, and renaming functions.

There are also operators for constructing number ranges. These expressions may be thought as simple distributed comprehensions:

output:	*input:*	*machine:*
$\{m \,..\, n\}$	`\lset m \range n \rset`	`{m..n}`
$\{m..\}$	`\lset m \range \rset`	`{m..}`

Note that the first of these expressions has the same semantics as `m \upto n`. The overloaded syntax is, again, a compromise between the Z notation and machine-readable CSP.

2.4 Sequences

To delimit sequences in CSP, we will use `\lseq` and `\rseq` rather than `\langle` and `\rangle`. This will act as a reminder that the sequence notation used here is somewhat more flexible than that of standard Z.

output:	*input:*	*machine:*
$\mathrm{seq}\, a$	`\seq a`	`Seq(a)`
$\langle\rangle$	`\nil`	`< >`
$\langle 1, 2, 3\rangle$	`\langle 1, 2, 3 \rangle`	`<1,2,3>`

We have a special test for the empty sequence, and an abbreviation for the predicate 'is in the range of':

output:	*input:*	*machine:*
$s = \langle\rangle$	`\Nil s`	`Null(s)`
$s \frown t$	`s \cat t`	`s^t`
$\#s$	`\#~s`	`#s`
$\mathit{head}\ s$	`head~s`	`head(s)`
$\mathit{tail}\ s$	`tail~s`	`tail(s)`
$\frown / s$	`\dcat s`	`concat(s)`
$x \in \mathrm{ran}\ s$	`x \elem s`	`elem(x,s)`
$\mathrm{ran}\ s$	`\ran s`	`set(s)`

We will allow the definition of sequence comprehensions, assuming that each variable range is associated with an obvious ordering:

output:	*input:*	*machine:*

$\langle\, x : a \mid p \bullet t \,\rangle$	`\lseq`
	`x : a | p @ t`
	`\rseq`
		`< t | x <- a, p>`

Finally, we may define number ranges as sequences, rather than sets:

output:	*input:*	*machine:*
$\langle m \mathinner{..} n\rangle$	`\lseq m \range n \rseq`	`< m .. n >`
$\langle m \mathinner{..}\rangle$	`\lseq m \range \rseq`	`< m .. >`

For convenience, `\langle` and `\rangle` can be used in place of `\lseq` and `\rseq`; you might like to use `\langle` and `\rangle` for sequence expressions that would be valid in standard Z, and `\lseq` and `\rseq` for those that would not: that is, comprehensions and ranges.

Similarly, `\{` and `\}` will produce the same results as `\lset` and `\rset`. You may find it useful to distinguish between the two macros, or you may prefer to settle upon just one. The `csp` package treats them as the same command for typesetting purposes, and will continue to do so.

3 Process language

The three basic processes used in CSP do not have macros: a simple text string will suffice:

output:	*input:*	*machine:*
$Stop$	`Stop`	`STOP`
$Skip$	`Skip`	`SKIP`
$Chaos(a)$	`Chaos(a)`	`CHAOS(a)`

The basic operators are all defined as relation symbols:

output:	*input:*	*machine:*
$c \rightarrow p$	`c \then p`	`c -> p`
$c?x : a \rightarrow p$	`c?x:a \then p`	`c?x:a -> p`
$c!v \rightarrow p$	`c!v \then p`	`c!v -> p`
$p \fatsemi q$	`p \comp q`	`p ; q`
$p \triangle q$	`p \interrupt q`	`p /\ q`
$p \setminus a$	`p \hide a`	`p \ a`
$p \Box q$	`p \extchoice q`	`p [] q`
$p \sqcap q$	`p \intchoice q`	`p |~| q`
$p \triangleright q$	`p \timeout q`	`p [> q`
$b \mathbin{\&} q$	`p \guard q`	`p & q`
$p \interleave q$	`p \interleave q`	`p ||| q`

Notice that we don't write `\&` for the guarding construct: it would mean overwriting the behaviour of this standard LaTeX macro.

There are two standard forms of communicating parallel combination in CSP: the first may introduce non-determinism, as events outside the shared set may occur independently, even if they have the same name; the second form identifies the domain of interest, or alphabet, of each process.

output:	*input:*	*machine:*
$p \llbracket a \rrbracket q$	`p \parallel[a] q`	`p [| a |] q`
$p \llbracket a \mid b \rrbracket q$	`p \parallel[a][b] q`	`p [ a || b ] q`

It is often convenient to break such expressions into their component parts: we may obtain the same results by writing

output:	*input:*	*machine:*
$p \llbracket a \rrbracket q$	`p \lpar a \rpar q`	`p [| a |] q`
$p \llbracket a \mid b \rrbracket q$	`p \lpar a \cpar b \rpar q`	`p [ a || b ] q`

A renaming expression is an enumeration or comprehension of renaming

terms: expressions of the form form $a \leftarrow b$, pronounced 'a becomes b',

output:	*input:*	*machine:*

$p[c \leftarrow d]$	`p \lren`
	`  c \becomes d`
	`\rren`
		`p[[c <- d]]`

$p[\, x : a \bullet c \leftarrow d \,]$	`p \lren~`
	`  x : a @`
	`  c \becomes d`
	`~\rren`
		`p [[`
		`  c <- d |`
		`  x <- a`
		`]]`

The linking operator is a combination of renaming, parallel composition, and hiding. Like the parallel operators, it can be written in component form.

output:	*input:*	*machine:*

$p \,\langle\!\langle\, a : A \bullet a \leftrightarrow b \,\rangle\!\rangle\, q$	`p \linking[`
	`  a : A @`
	`  a \linksto b`
	`  ]q`
		`p [a <-> b | a <- A] q`

$p \,\langle\!\langle\, a : A \bullet a \leftrightarrow b \,\rangle\!\rangle\, q$	`p \llink`
	`  a : A @`
	`  a \linksto b`
	`\rlink q`
		`p [a <-> b | a <- A] q`

Note that there should be no need to insert `~` commands to provide the correct spacing: `\llink` and `\rlink` are defined as binary operator symbols.

There are indexed forms of the sequential composition, choice, parallel, and linking operators. Note that the first and last of these take a sequence s, not an ordinary set a, as a range expression:

output:	*input:*	*machine:*
$\fatsemi\, x : s \bullet p$	`\Comp x : a @ p`	`; x : a @ p`
$\Box\, x : a \bullet p$	`\Extchoice x : a @ p`	`[] x : a @ p`
$\sqcap\, x : a \bullet p$	`\Intchoice x : a @ p`	`|~| x : a @ p`
$\|\|\|\, x : a \bullet p$	`\Interleave x : a @ p`	`||| x:a @ p`
$\|\, x : a \bullet b \circ p$	`\Parallel` `  x : a @ b \circ p`	`|| x:a @ [b] p`

$\langle\!\langle c \leftrightarrow d \,\rangle\!\rangle\, x : s \bullet p$	`\Linking`
	`  [c \linksto d]`
	`  x : s @ p`
		`[c <-> d] x:s @ p`

In the syntax for indexed alphabeticised parallel, the set expression that comes after $\circ$ describes the domain of interest of the corresponding indexed process.

The set of events shared by a parallel combination is often more easily described in terms of channels: sets of events with the same prefix. Since channels

are declared separately, these sets can be produced from the channel names alone; the function productions does this. The function extensions, on the other hand, generates the set of completions: the set of expressions that could be added to a channel name to produce a valid event.

Alternatively, we may use an expanding distributed comprehension. As with the other distributed comprehensions, this may have a list of terms rather than just a single term.

output:	*input:*	*machine:*
productions c	`\productions c`	`productions(c)`
extensions c	`\extensions c`	`extensions(c)`
$\{\!\|c, d\|\!\}$	`\lchan c, d \rchan`	`{| c, d |}`
$\{\!\|x : a \mid c, d\|\!\}$	`\lchan x : a | c, d \rchan`	`{| c, d | x <- a |}`

Finally, there is the conditional expression operator, usually applied to the definition of processes, but also found in the definition of other objects:

output:	*input:*	*machine:*
if B then E else F	`\If B \Then E \Else F`	`if B then E else F`

4 Definitions

4.1 Contexts

An important feature of this version of the CSP language is the introduction of explicit *contexts*. In constructing and explaining models, we may wish to declare several different versions of the same process. By making the context of each declaration explicit, we can avoid, or automate, a great deal of renaming.

One context, *Global*, is pre-defined: declarations without context arguments are added to this global context. Other contexts must be declared separately, and may *extend* or *use* other contexts, or even combinations of contexts.

We write 'B extends A' to declare a context B as an extension of A. If B is an extension of A, and the same object is declared in both contexts, then—within context B—only one declaration takes effect: that of the current context. The first declaration is overwritten.

We write 'D uses C' to declare a context D in which all of the objects declared in C are accessible in qualified form: by following the object name with the name of the context. For example, if P were defined in C, we would refer to it, within D, as $P[C]$. In general, we write

output:	*input:*	*machine:*
Name[*Context*]	`Name[Context]`	`Context_Name`

to indicate that this usage of *Name* corresponds to the declaration in *Context*.

A context declaration may be presented in vertical form,

```
\begin{context}
  A \extends Global \\
  B \extends A \\
  C \uses B
\end{context}
```

context
  A extends *Global*
  B extends A
  C uses B

The lined declaration syntax is explained below.

The right-hand argument to 'extends' and 'uses' may be a list of contexts. In the case of 'extends', this means that the new context is obtained by extending the contexts one-by-one, in the order given. In the case of 'uses', all of the declarations from the list of contexts are available, in qualified form.

For example, if O is defined in both A and B, and

context
  A extends *Global*
  B extends *Global*
  C uses A, B
  D extends A, B

then C may refer to $O[A]$ and $O[B]$, while D may refer only to O, whose value is taken from B.

4.2 Outer declarations

Each declaration that we make—of a process, a channel, or a function—will be associated with a context. There is one exception to this rule: *contexts* are not themselves declared within contexts. We can refer to the declared object only within the specified context and those contexts that extend it. If no context is specified, then the names declared will be added to the global context.

An outer declaration is an environment with an optional argument, the name of the corresponding context. There are several different types of declaration: one for each kind of declared object.

- `channel`
- `process`
- `function`
- `set`
- `datatype`
- `nametype`
- `subtype`
- `external`

– `transparent`

– `assertion`

For example, to declare a channel *input* in context *Example*, we would write

```
\begin{channel}[Example]
  input
\end{channel}
```

and obtain, as output,

Example

channel
  input

In machine-readable CSP, we might write

```
channel Example_input
```

to differentiate this channel from other *input* channels.

Several, separate outer declarations may be made for the same context. For example, suppose that we wished to declare some channels and a process within a single context *DiaryOne*. We could write

```
\begin{channel}[DiaryOne]
  invite, accept, reject, confirm, cancel,
  attend, notattend
\end{channel}
```

giving

DiaryOne

channel
  invite, *accept*, *reject*, *confirm*, *cancel*, *attend*, *notattend*

and then

```
\begin{process}[DiaryOne]
  Client = {} \\ \c1
  \begin{block}
    invite \then {} \\ \c1
    \begin{block}
      accept \then {} \\ \c1
      \begin{block}
        confirm \then attend \then STOP \\
        \extchoice \\
        cancel \then notattend \then STOP
      \end{block} \\
      \intchoice \\
      \begin{block}
        reject \then {} \\ \c1
        \begin{block}
```

```
            confirm \then notattend \then STOP \\
            \extchoice \\
            cancel \then notattend \then STOP
          \end{block}
        \end{block}
      \end{block}
    \end{block}
  \end{process}
```

to give

DiaryOne

process
$Client =$
$invite \rightarrow$
$accept \rightarrow$
$confirm \rightarrow attend \rightarrow STOP$
$\Box$
$cancel \rightarrow notattend \rightarrow STOP$
$\sqcap$
$reject \rightarrow$
$confirm \rightarrow notattend \rightarrow STOP$
$\Box$
$cancel \rightarrow notattend \rightarrow STOP$

At this point, it is worth discussing the facilities provided for horizontal and vertical alignment. The `\c`n command produces n tabs' worth of indentation, where n is an integer between 1 and 9, inclusive; the `{}` after the binary symbols `=` and `\then` causes TEX to insert the correct spacing *before* each symbol.

The unit for indentation is given by `\csptab`: in the package, this is set to `1em`; you can change this by assigning a new length: e.g., `\csptab=2em`. If you are in the habit of typing `\t`n, the `zed.sty` indentation command, then you might prefer to type `\t`n instead of `\c`n, provided that you don't mind having the same indentation in both environments.

The `block` environment allows us to group several lines of text: it is simply a wrapper for the `array` environment, insisting upon the column specification `{@{}l@{}}` but allowing an optional argument to dictate vertical alignment, with `[t]` as the default. This alignment has semantic significance: `\begin{block}` and `\end{block}` act as parentheses for parsing purposes.

4.3 Inner declarations

Any of the declaration environments can be used within an existing declaration: the result is an *inner* declaration in which the context argument is ignored. This feature of the language can be exploited, in conjunction with the vanilla

– `declaration`

environment, to group a collection of declarations together, associating them all with the same context. For example,

```
\begin{declaration}[DiaryOne]
  \begin{channel}
    invite, accept, reject, confirm, cancel,
    attend, notattend
  \end{channel} \also
  \begin{process}
    Client = {} \\ \c1
    \begin{block}
      invite \then {} \\ \c1
      \begin{block}
        accept \\ \c1
        \begin{block}
          confirm \then attend \then STOP \\
          \extchoice \\
          cancel \then notattend \then STOP
        \end{block} \\
        \intchoice \\
        \begin{block}
          reject \then {} \\ \c1
          \begin{block}
            confirm \then notattend \then STOP \\
            \extchoice \\
            cancel \then notattend \then STOP
          \end{block}
        \end{block}
      \end{block}
    \end{block}
  \end{process}
\end{declaration}
```

which appears as

DiaryOne

channel

$invite, accept, reject, confirm, cancel, attend, notattend$

process

$$
\begin{array}{l}
Client = \\
\quad invite \rightarrow \\
\quad\quad accept \rightarrow \\
\quad\quad\quad confirm \rightarrow attend \rightarrow STOP \\
\quad\quad\quad \Box \\
\quad\quad\quad cancel \rightarrow notattend \rightarrow STOP \\
\quad\quad \sqcap \\
\quad\quad reject \rightarrow \\
\quad\quad\quad confirm \rightarrow notattend \rightarrow STOP \\
\quad\quad\quad \Box \\
\quad\quad\quad cancel \rightarrow notattend \rightarrow STOP
\end{array}
$$

has the same effect as the two outer declarations given previously.

Consecutive declarations will be set alongside each other unless an explicit `\\`, `\also`, or `\Also` command is included. These produce a new line with vertical skips of increasing magnitude: `\also` will usually suffice.

4.4 Local declarations

If the `declaration` appears inside another declaration, then the result is a local declaration: the names declared are accessible only within the enclosing declaration; they are not added to any named context. Any number of other declarations can be placed inside a local declaration environment, but every such environment must be followed immediately by a `within` environment, which identifies the scope of the declarations.

For example, we may declare several processes within the right-hand side of a process definition—common practice in machine-readable CSP—

```
\begin{process}[Example]
  P = {} \\ \c1
  \begin{block}
    \begin{declaration}
      \begin{process}
        PA = a \then PB \\
        PB = b \then PA
      \end{process}
    \end{declaration} \\
    \begin{within}
      PA
    \end{within}
  \end{block}
\end{process}
```

which appears as

Example

process
$P =$
let
process
$PA = a \rightarrow PB$
$PB = b \rightarrow PA$
within
PA

4.5 Horizontal form

For each of the declaration environments described above, there is a corresponding horizontal form. The commands

- `\Channel`
- `\Process`
- `\Function`

– `\Set`
– `\Datatype`
– `\Nametype`
– `\Subtype`
– `\External`
– `\Transparent`
– `\Assertion`

produce the declaration keyword followed by a horizontal space. The only way to specify a context is to enclose these declarations inside a vertical `declaration` environment. There is no horizontal form of `declaration`: instead, we write `\Let` and `\Within` for inner and local declarations.

output:	*input:*	*machine:*
let D within P	`\Let D ~~\Within P`	`let D within P`

If no context is to be specified, then the declarations can be made within the plain `csp` environment. In this case, no vertical line is added—although the `barcsp` environment will do this for you if you wish: the effect is the same as an outer `declaration` environment with no context specified.

Finally, there is also a `nolines` option to the package, which will produce declarations with neither horizontal nor vertical lines.

5 Semantics

5.1 Semantic equations

output:	*input:*	*machine:*
traces $[\![P]\!]$	`\traces[P]`	`---`
failures $[\![P]\!]$	`\failures[P]`	`---`
divergences $[\![P]\!]$	`\divergences[P]`	`---`

5.2 Assertions

output:	*input:*	*machine:*
deterministic$_m$ p	`\Deterministic[m] p`	`p [:deterministic [m]]`
deadlock free$_m$ p	`\DeadlockFree[m] p`	`p [:deadlock free [m]]`
divergence free p	`\DivergenceFree p`	`p [:divergence free]`
$p \sqsubseteq_m q$	`p \refinedby[m] q`	`p [m= q`
assert $p \sqsubseteq_m q$	`\begin{assertion}` `p \refinedby[m] q` `\end{assertion}`	`assert p [m= q`

5.3 Extensions

- You can add a new declaration environment using the `\newdeclaration` command. The first argument is the name of the vertical environment, the second is the name of the corresponding horizontal for; the third is the text to be inserted. For example, the `assertion` environment demonstrated above was defined by

  ```
  \newdeclaration{assertion}{Assertion}{assert}
  ```

- You can add a new semantic function using the `\newsemantics` command: the first argument is the new macro name; the second is the text to be inserted. Every semantic function takes an optional argument, to be set in semantic brackets. The standard definitions include:

  ```
  \newsemantics{traces}{traces}
  ```

  although you could redefine these with, for example, something like

  ```
  \newsemantics{traces}{\mathcal{T}}
  ```

  The commands `\ldbrack` and `\rdbrack` produce the double brackets on their own.

- `csp.sty` will load `zed.sty`, `color.sty` (for colour), `lucidabr.sty` (for Lucida fonts), and `mathtime.sty` (for Times fonts) only if they are not already loaded. If you need to load these packages with your own list of options, you have only to load them before you load `csp.sty`.

- In machine-readable CSP, the bars and brackets used to build parallel operators appear in a different order. If you would like the typeset versions to match the existing $\mathrm{CSP_M}$ form, simply insert the following code in a style file after loading `csp`:

  ```
  \def\lpar@sym{{[}\mkern -1mu{|}\mkern-1mu}
  \def\rpar@sym{\mkern-1mu{|}\mkern -1mu{]}}
  ```

  The existing typeset versions match the syntax chosen for parallel composition in the language LOTOS, an international standard for process description. Nevertheless, I am open to suggestions regarding the default behaviour of this package.

- If you would like a pair of optional arguments to the `\parallel` operator to be set as subscripts, then you need to insert

  ```
  \def\@@parallel[#1][#2]{%
    \mathrel{%
      {\vphantom\parallel@sym}_{#1}{\parallel@sym}_{#2}}}
  ```

  into your own style file, again after loading `csp`.