User Guide CSP-Prover Ver.3.0

CSP-Prover Document
Version: DRAFT May 10, 2006

Yoshinao Isobe[1] and Markus Roggenbach[2]

[1] National Institute of Advanced Industrial Science and Technology, Japan,
y-isobe@aist.go.jp,
[2] University of Wales Swansea, United Kingdomm
M.Roggenbach@swan.ac.uk

Contents

1 Introduction

We describe a tool called Csp-Prover which is an interactive theorem prover dedicated to refinement proofs within the process algebra Csp. It aims specifically at proofs on infinite state systems, which may also involve infinite nondeterminism. For this reason, Csp-Prover currently focuses on the stable failures model $\mathcal{F}$ as the underlying denotational semantics of Csp.

Semantically, Csp-Prover offers both classical approaches to denotational semantics: the theory of complete partial orders (cpo) as well as the theory of complete metric spaces (cms). In this context the respective Fixed Point Theorems are used for two purposes: (1) to prove the existence of fixed points, and (2) to prove Csp refinement between two fixed points. Csp-Prover implements both these theories for infinite product spaces and thus is capable to deal with infinite systems of process equations.

Technically, Csp-Prover is based on the generic theorem prover Isabelle, using the logic HOL-Complex. Within this logic, the syntax as well as the semantics of Csp is encoded, i.e., Csp-Prover provides a deep encoding of Csp. The tool's architecture follows a generic approach which makes it easy to re-use large parts of the encoding for other Csp models. For instance, merely as a by-product, Csp-Prover includes also the Csp traces model $\mathcal{T}$. More importantly, Csp-Prover can easily be extended to the failure-divergence model $\mathcal{N}$ and the various infinite traces models of Csp.

Consequently, Csp-Prover contains fundamental theorems such as fixed point theorems on cpo and cms, the definitions of Csp syntax and semantics, and many Csp-laws and semi-automatic proof tactics for verification of refinement relation. Therefore, Csp-Prover can be used for

1. Verification of infinite state systems. For example, we applied Csp-Prover to verify a part of the specification of the EP2 system, which is a new industrial standard of electronic payment systems, in [IR05].

2. Establishing new theorems on Csp. For example, Csp-Prover assisted us very well in proving new theorems on a sound and complete axiom system for the stable failures equivalence over processes with unbounded nondeterminism over arbitrary alphabet. The result is included in the package `FNF_F` in `CSP-Prover-3-0.tar.gz`.

In Isabelle, theorems, together with definitions and proof-scripts needed for their proof, can be stored in *theory-files*. Currently, Csp-Prover consists of three packages of theory-files: `CSP`, `CSP_T`, and `CSP_F`. The package `CSP` is the reusable part independent of specific Csp models. For example, it contains fixed point theorems on cpo and cms, and the definition of Csp syntax. The packages `CSP_T` and `CSP_F` are instantiated parts for the traces model and the stable failures model. The packages have a hierarchical organisation as: `CSP_F` on `CSP_T` on `CSP` on Isabelle/HOL-Complex. The theorems for the sound and complete axiom system for the stable failures equivalence are stored in a new package `FNF_F` implemented on `CSP_F`.

In this document, we explain how to set up Csp-Prover and to use it.

2 Installing Isabelle2005

Csp-Prover is encoded in Isabelle2005/HOL-Complex. To install the interactive theorem prover Isabelle follow the instructions of the Isabelle Web page:

```
http://www.cl.cam.ac.uk/Research/HVG/Isabelle/index.html
```

For example, download the following files for Linux/x86 from the web page:

```
Isabelle2005.tar.gz
ProofGeneral.tar.gz
polyml_x86-linux.tar.gz
HOL_x86-linux.tar.gz
HOL-Complex_x86-linux.tar.gz
```

Then, uncompress and unpack them into e.g. the directory `/usr/local` as follows:

```
% tar -C /usr/local -xzf Isabelle2005.tar.gz
% tar -C /usr/local -xzf ProofGeneral.tar.gz
% tar -C /usr/local -xzf polyml_x86-linux.tar.gz
% tar -C /usr/local -xzf HOL_x86-linux.tar.gz
% tar -C /usr/local -xzf HOL-Complex_x86-linux.tar.gz
```

Isabelle/Isar/HOL is started by

```
% /usr/local/Isabelle/bin/isabelle -I HOL
```

Proof General is started by

```
% /usr/local/Isabelle/bin/Isabelle
```

For the rest of this document, we assume that `/usr/local/Isabelle/bin` is an executable path.

3 Setting up CSP-Prover

Download the file `CSP-Prover-3-0.tar.gz` from

`http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`

and unpack it e.g. in the directory

`/usr/local/CSP-Prover-3-0`

by an unpacking command (e.g. `tar zxvf CSP-Prover-3-0.tar.gz`).

`CSP-Prover-3-0` contains the 7 directories as follows:

- `CSP` : the reusable part of Csp-Prover,
- `CSP_T` : the instantiated part for the traces model $\mathcal{T}$,
- `CSP_F` : the instantiated part for the stable-failures model $\mathcal{F}$,
- `FNF_F` : the theory for full normalisation in the model $\mathcal{F}$,
- `DM` : an example to verify the Dining Mathematicians[CS01].
- `ep2` : an industrial case study on an electronic payment system **ep2**[ep202].
- `Test` : small examples for testing Csp-Prover.

It is recommended to make heap files: `CSP`, `CSP_T`, `CSP_F`, and `FNF_F`. If you make the heap files once, you do not have to prove them again before using them. You can make the four heap files by one command

```
% make_heaps
```

at the directory `/usr/local/CSP-Prover-3-0/`, where the environment variable "`ISABELLE_bin`" has to be set to the path containing the command `isatool` of Isabelle2005. The heap file will be made in your isabelle directory. If you did not specify the directory, it is probably

`~/isabelle/heaps/polyml-***` (which depends on your OS)

It may take time to make the four heap files. For example, about 13 minutes by Pentium M (1.5GHz).

In addition, if you like to comfortably read theory files of Csp-Prover by browsers (e.g. Netscape, mozilla, $\cdots$), you can make html files for them by a command

```
% make_html
```

at the directory `/usr/local/CSP-Prover-3-0/`. The theory files and theory dependency-graphs can be browsed by the web-browsers (e.g. mozilla):

```
% cd ~/isabelle/browser_info/HOL/HOL-Complex/CSP
% mozilla index.html
```

or the theory dependency-graphs can be browsed by `isatool`:

```
% cd ~/isabelle/browser_info/HOL/HOL-Complex/CSP
% isatool browser session.graph
```

where Java is needed for displaying graphs.

4 Starting CSP-Prover

You can start the logic `CSP_F` for the stable failures model $\mathcal{F}$ of Csp-Prover in a shell window by

```
% isabelle -I CSP_F
```

or start it in Proof General[Asp00] by

```
% Isabelle -l CSP_F
```

It is recommended to use Proof General, which is a superior interface for Isabelle. Proof General sometimes conflicts your `.emacs` and then fails. To avoid this, you may use an option "-u" as follows:

```
% Isabelle -u false -l CSP_F
```

This option disallows Proof General to use your `.emacs`.

In Proof General, you can also select a logic (e.g. `CSP`, `CSP_T`, `CSP_F`, `FNF_F`, `HOL`, `HOL-Complex`, $\cdots$) used in Isabelle from the menu bar. Click the button [Isabelle/Isar] $\rightarrow$ [Logics] $\rightarrow$ [CSP].

In addition, you can also activate X-symbols in Proof General from the menu bar. Click the button [Proof General] $\rightarrow$ [option] $\rightarrow$ [X-Symbol]. Csp-Prover also provides a more conventional syntax of processes based on X-symbols. For example, the external choice `P [+] Q` in ASCII mode is replaced with `P` $\Box$ `Q` in X-symbol mode.

5 Small demonstration

Let us prove small examples, for getting the overview how Csp-Prover works. If you use a shell window and an editor window, the proof is proceeding as follows:

1. Start Isabelle with the logic `CSP_F` in the shell window by

```
% isabelle -I CSP_F
```

2. Open the following example in the editor window:

   `/usr/local/CSP-Prover-3-0/Test/Test_infinite.thy`

3. Copy the commands from "`Test_infinite.thy`" and paste them to the Isabelle window line by line until the proof finishes.

If you can use Proof General, the proof is more elegant as follows:

1. Start Proof General with `CSP_F` by

   ```
   % Isabelle -l CSP_F
   ```

2. Open the following example in the Proof General window:

   `/usr/local/CSP-Prover-3-0/Test/Test_infinite.thy`

3. Click the button "Next" in the menu bar until the proof finishes.

Similarly, try to prove another example:

`/usr/local/CSP-Prover-3-0/Test/Test_finite.thy`

The examples `ep2` and `DM` are explained in the web-site of CSP-Prover:

`http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`

6 The CSP-dialect

This section summarises syntax and semantics of the input language of CSP-Prover, and then we show that it can deal with infinitely many mutual recursive processes.

6.1 Syntax

Figure 1 shows the syntax of CSP implemented in CSP-Prover: given an alphabet of communications Σ and the data type of natural numbers Nat, we form a set $Sel(\Sigma)$ of *selectors* to be explained below. $Proc_\Sigma$ denotes the set of the processes whose alphabet is Σ.

The set $Sel(\Sigma)$ of *selectors* used in the replicated internal choice is defined as the disjoint sum of the powerset over Σ and the set of natural numbers:

$$Sel(\Sigma) = \{(set)\, A \mid A \subseteq \Sigma\} \cup \{(nat)\, n \mid n \in Nat\}$$

Note that replicated internal choice takes a *subset* of $Sel(\Sigma)$ as its parameter.

One difference from conventional CSP is that we replace the generic internal choice $\sqcap \mathcal{P}$ by a replicated internal choice $!!\, c : C \bullet P(c)$, i.e., instead of having

$P ::=$	$\texttt{SKIP}$	%% successful terminating process
	$\mid \texttt{STOP}$	%% deadlock process
	$\mid \texttt{DIV}$	%% divergence
	$\mid a \rightarrow P$	%% action prefix
	$\mid ?\, x : A \rightarrow P(x)$	%% prefix choice
	$\mid P \,\Box\, P$	%% external choice
	$\mid P \sqcap P$	%% internal choice
	$\mid !!\, c : C \bullet P(c)$	%% replicated internal choice
	$\mid \texttt{IF}\ b\ \texttt{THEN}\ P\ \texttt{ELSE}\ P$	%% conditional
	$\mid P \,[\![X]\!]\, P$	%% generalized parallel
	$\mid P \setminus X$	%% hiding
	$\mid P[[r]]$	%% relational renaming
	$\mid P \fatsemi P$	%% sequential composition
	$\mid P \lfloor n$	%% depth restriction

where $A, X \subseteq \Sigma$, $C \subseteq \mathit{Sel}(\Sigma)$, b is a condition, $r \in \mathbb{P}(\Sigma \times \Sigma)$, and $n \in \mathit{Nat}$.

Figure 1: Syntax of basic CSP processes in CSP-Prover.

internal choice over an arbitrary class of processes $\mathcal{P} \subseteq \mathit{Proc}_\Sigma$, internal choice is restricted to run over an indexed set of processes $P(\cdot) : \mathit{Sel}(\Sigma) \Rightarrow \mathit{Proc}_\Sigma$ only, where the index set C is a subset of $\mathit{Sel}(\Sigma)$. The other difference is that we introduce depth-restriction $\lfloor$ as a basic operator [1]. Restriction plays an important role in full-normalisation. As [Ros98] shows, for the stable-failures model restriction cannot be defined in terms of the other basic operators.

The following shortcuts are also available in CSP-Prover:

- (Untimed) timeout:

$$P \triangleright Q := (P \sqcap \texttt{STOP}) \,\Box\, Q$$

- Replicated internal choices:

$$\begin{aligned}
!\texttt{set}\ A : \mathcal{A} \bullet P(A) &:= !!\, c : \{(\mathit{set})\, A \mid A \in \mathcal{A}\} \bullet P((\mathit{set})^{-1}(c)) \\
!\texttt{nat}\ n : N \bullet P(n) &:= !!\, c : \{(\mathit{nat})\, n \mid n \in N\} \bullet P((\mathit{nat})^{-1}(c)) \\
!\ x : A \bullet P(x) &:= !\texttt{set}\ X : \{\{x\} \mid x \in A\} \bullet P(\mathit{contents}(X)) \\
!\langle f \rangle\ z : Z \bullet P(z) &:= !\, x : \{f(z) \mid z \in Z\} \bullet P(f^{-1}(z))
\end{aligned}$$

  where $\mathcal{A} \subseteq \mathbb{P}(\Sigma)$, $N \subseteq \mathit{Nat}$, $A \subseteq \Sigma$, and $\mathit{contents}(\{x\}) = x$. The last one can be used for expressing the non-determinism over any type τ by a type converter $f : \tau \rightarrow \Sigma$. For example, if you want to use the non-determinism over real numbers, it can be expressed as follows:

$$!\langle(\mathit{real})\rangle\ r : R \bullet P(r)$$

[1] Although the restriction function is conventionally denoted by $\downarrow$, we have already used it as restriction function in semantic domains. Therefore, $\lfloor$ is used in process expressions in order to avoid syntactically ambiguous input in Isabelle.

where $R \subseteq \mathit{Real}$ and $\{(\mathit{real})\, r \mid r \in \mathit{Real}\} \subseteq \Sigma$.

- Internal prefix choice:

$$! \; x : A \to P(x) := ! \; x : A \bullet (x \to P(x))$$

- Sending '!', receiving '?', and non-deterministic sending '!?' prefixes:

$$\begin{aligned} a!v \to P &:= a(v) \to P \\ a?x : X \to P(x) &:= ? \; x : \{a(v) \mid v \in X\} \to P(a^{-1}(x)) \\ a!?x : X \to P(x) &:= ! \; x : \{a(v) \mid v \in X\} \to P(a^{-1}(x)) \end{aligned}$$

The prefix $a!?x : X \to P(x)$ nondeterministically sends a value $v \in X$, and then the value is retained in $P(v)$. The non-deterministic sending prefix may not be used in the implementations, but it can be used for expressing beginning (loose) specifications.

- If the range of selectors, receiving values, etc, is the universe, the universe can be omitted, for example we can write $!\texttt{nat}\ n \bullet P(n)$ instead of $!\texttt{nat}\ n : \mathit{Nat} \bullet P(n)$ and $a?x \to P(x)$ instead of $a?x : \mathit{Univ} \to P(x)$.

- Interleaving, synchronous, and alphabetized parallels:

$$\begin{aligned} P \mid\mid\mid Q &:= P \; [\![\emptyset]\!] \; Q \\ P \mid\mid Q &:= P \; [\![\Sigma]\!] \; Q \\ P \; [\![X, Y]\!] \; Q &:= (P \; [\![\Sigma - X]\!] \; \texttt{SKIP}) \; [\![X \cap Y]\!] \; (Q \; [\![\Sigma - Y]\!] \; \texttt{SKIP}) \end{aligned}$$

- Inductive alphabetized parallel:

$$\begin{aligned} [|||] \; \langle\rangle &:= \texttt{SKIP} \\ [|||] \; (P, X) \frown PX_{\mathit{list}} &:= P \; [\![X, Y]\!] \; ([|||] \; PX_{\mathit{list}}) \end{aligned}$$

where $Y = \bigcup \{X \mid \exists P. \, (P, X) \in \mathit{set}(PX_{\mathit{list}})\}$ and $\mathit{set}(\mathit{list})$ is the set of all the elements in list, thus

$$\begin{aligned} \mathit{set}(\langle\rangle) &= \emptyset \\ \mathit{set}(\langle e \rangle \frown \mathit{tail}) &= \{e\} \cup \mathit{set}(\mathit{tail}) \end{aligned}$$

- Replicated alphabetized parallel:

$$[|||] \; i : I \bullet (P_i, X_i) := [|||] \; (\mathit{map} \; (\lambda \, i. \; (P_i, X_i)) \; I_{\mathit{list}})$$

where I is a finite index set and the list I_{list} is given from I as follows:

$$I_{\mathit{list}} := \varepsilon \; \mathit{list}. \; (I = \mathit{set}(\mathit{list}) \wedge \mid I \mid = \mid \mathit{list} \mid)$$

where $\mid I \mid$ is the size of the finite set I, $\mid \mathit{list} \mid$ is the length of list, ε is the Hilbert's ε-operator, thus $(\varepsilon \, x. \, \mathit{pred}(x))$ is an x such that $\mathit{pred}(x)$ is true, and map is defined as follows:

$$\begin{aligned} \mathit{map} \; f \; \langle\rangle &= \langle\rangle \\ \mathit{map} \; f \; (\langle e \rangle \frown \mathit{tail}) &= \langle f(e) \rangle \frown (\mathit{map} \; f \; \mathit{tail}) \end{aligned}$$

Note that I_{list} is not uniquely decided from I. However, the semantics of $[\!|||\!]\ i : I \bullet (P_i, X_i)$ is uniquely decided and it equals to the well known semantics of Replicated alphabetized parallel.

6.2 Semantics

Currently, CSP-Prover concentrates on the denotational stable-failures model $\mathcal{F}$ of CSP. Its domain $\mathcal{F}_\Sigma$ is given as the set of all pairs (T, F) that satisfy certain healthiness conditions.

Definition 1 *Given a set of communications Σ, the domain of the stable failures model $\mathcal{F}_\Sigma$ is a set of pairs (T, F) satisfying the following healthiness conditions, where $T \subseteq \Sigma^{*\checkmark}$ and $F \subseteq \Sigma^{*\checkmark} \times \mathbb{P}(\Sigma^{\checkmark})$* [2].

T1 *T is non-empty and prefix closed,*

T2 $(t, X) \in F \Longrightarrow t \in T$,

T3 $t \frown \langle\checkmark\rangle \in T \Longrightarrow (t \frown \langle\checkmark\rangle, X) \in F$,

F2 $(t, X) \in F \wedge Y \subseteq X \Longrightarrow (t, Y) \in F$,

F3 $(t, X) \in F \wedge (\forall\, a \in Y.\ t \frown \langle a\rangle \notin T) \Longrightarrow (t, X \cup Y) \in F$,

F4 $t \frown \langle\checkmark\rangle \in T \Longrightarrow (t, \Sigma) \in F$.

The labels **T1**, $\cdots$, **F4** *of the healthiness conditions are the same as ones used in [Ros98]. We denote the set of traces satisfying* **T1** *by $\mathcal{T}_\Sigma$, which is exactly the domain of the traces model.*

The semantics of a process P is defined by $[\![P]\!]_{\mathcal{F}}$, where $[\![\cdot]\!]_{\mathcal{F}} : Proc_\Sigma \to \mathcal{F}_\Sigma$ is a map expressed with the help of two functions: $[\![P]\!]_{\mathcal{F}} = (traces(P), failures(P))$. Then, the functions *traces* and *failures* are recursively defined by the semantic clauses given in Figure 2. Our definitions of *traces* and *failures* are identical to those given in [Ros98] except that the clauses of our two new operators, namely replicated internal choice[3] and depth restriction are added. The auxiliary notations $t_1 \,[\![\, X \,]\!]\, t_2$, $t \setminus X$, $[[r]]^*$, $[[r]]^{-1}$, $T \downarrow n$, and $F \downarrow n$ used in Figure 2 are defined as follows:

[2] $\Sigma^{\checkmark} := \Sigma \cup \{\checkmark\}$, $\Sigma^{*\checkmark} := \Sigma^* \cup \{t \frown \langle\checkmark\rangle \mid t \in \Sigma^*\}$.

[3] As we allow the empty set $\emptyset$ as a set C of selectors, we need to add $\{\langle\rangle\}$ to the set of traces.

$$
\begin{aligned}
traces(\texttt{SKIP}) &= \{\langle\rangle, \langle\checkmark\rangle\} \\
traces(\texttt{STOP}) &= \{\langle\rangle\} \\
traces(\texttt{DIV}) &= \{\langle\rangle\} \\
traces(a \to P) &= \{\langle\rangle\} \cup \{\langle a\rangle \frown t' \mid t' \in traces(P)\} \\
traces(?\ x : A \to P(x)) &= \{\langle\rangle\} \cup \{\langle x\rangle \frown t' \mid t' \in traces(P(x)), x \in A\} \\
traces(P \,\Box\, Q) &= traces(P) \cup traces(Q) \\
traces(P \sqcap Q) &= traces(P) \cup traces(Q) \\
traces(!!\ c : C \bullet P(c)) &= \bigcup\{traces(P(c)) \mid c \in C\} \cup \{\langle\rangle\} \\
traces(\texttt{IF}\ b\ \texttt{THEN}\ P\ \texttt{ELSE}\ Q) &= \text{if } b \text{ then } traces(P) \text{ else } traces(Q) \\
traces(P \,[\![\, X \,]\!]\, Q) &= \{t_1 \,[\![\, X \,]\!]\, t_2 \mid t_1 \in traces(P),\ t_2 \in traces(Q)\} \\
traces(P \setminus X) &= \{t \setminus X \mid t \in traces(P)\} \\
traces(P[[r]]) &= \{t \mid \exists\, t' \in traces(P).\ (t', t) \in [[r]]^*\} \\
traces(P \fatsemi Q) &= (traces(P) \cap \Sigma^*) \\
&\quad \cup \{t_1 \frown t_2 \mid t_1 \frown \langle\checkmark\rangle \in traces(P),\ t_2 \in traces(Q)\} \\
traces(P \lfloor n) &= traces(P) \downarrow n
\end{aligned}
$$

$$
\begin{aligned}
failures(\texttt{SKIP}) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma\} \cup \{(\langle\checkmark\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\} \\
failures(\texttt{STOP}) &= \{(\langle\rangle, X) \mid X \subseteq \Sigma^{\checkmark}\} \\
failures(\texttt{DIV}) &= \emptyset \\
failures(a \to P) &= \{(\langle\rangle, X) \mid a \notin X\} \\
&\quad \cup \{(\langle a\rangle \frown t', X) \mid (t', X) \in failures(P)\} \\
failures(?\ x : A \to P(x)) &= \{(\langle\rangle, X) \mid A \cap X = \emptyset\} \\
&\quad \cup \{(\langle x\rangle \frown t', X) \mid (t', X) \in failures(P(x)), x \in A\} \\
failures(P \,\Box\, Q) &= \{(\langle\rangle, X) \mid (\langle\rangle, X) \in failures(P) \cap failures(Q)\} \\
&\quad \cup \{(t, X) \mid (t, X) \in failures(P) \cup failures(Q), t \neq \langle\rangle\} \\
&\quad \cup \{(\langle\rangle, X) \mid X \subseteq \Sigma, \langle\checkmark\rangle \in traces(P) \cup traces(Q)\} \\
failures(P \sqcap Q) &= failures(P) \cup failures(Q) \\
failures(!!\ c : C \bullet P) &= \bigcup\{failures(P(c)) \mid c \in C\} \\
failures(\texttt{IF}\ b\ \texttt{THEN}\ P\ \texttt{ELSE}\ Q) &= \text{if } b \text{ then } failures(P) \text{ else } failures(Q) \\
failures(P \,[\![\, X \,]\!]\, Q) &= \{(u, Y \cup Z) \mid Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}), \\
&\qquad \exists\, t_1, t_2.\ (t_1, Y) \in failures(P), (t_2, Z) \in failures(Q), \\
&\qquad u \in t_1 \,[\![\, X \,]\!]\, t_2\} \\
failures(P \setminus X) &= \{(t \setminus X, Y) \mid (t, Y \cup X) \in failures(P)\} \\
failures(P[[r]]) &= \{(t, X) \mid \exists\, t'.\, (t', t) \in [[r]]^*, \\
&\qquad\qquad (t', [[r]]^{-1}(X)) \in failures(P)\} \\
failures(P \fatsemi Q) &= \{(t_1, X) \mid t_1 \in \Sigma^*,\ (t_1, X \cup \{\checkmark\}) \in failures(P)\} \\
&\quad \cup \{(t_1 \frown t_2, X) \mid t_1 \frown \langle\checkmark\rangle \in traces(P), \\
&\qquad\qquad (t_2, X) \in failures(Q)\} \\
failures(P \lfloor n) &= failures(P) \downarrow n
\end{aligned}
$$

Figure 2: Semantic clauses for the model $\mathcal{F}$ in our CSP dialect.

- $t_1 \,[\![X]\!]\, t_2$ is inductively defined by:

$$\begin{array}{rl}
\langle x\rangle \frown t_1 \,[\![X]\!]\, \langle x\rangle \frown t_2 &= \{\langle x\rangle \frown u \mid u \in t_1 \,[\![X]\!]\, t_2\} \\
\langle x\rangle \frown t_1 \,[\![X]\!]\, \langle x'\rangle \frown t_2 &= \emptyset \\
\langle x\rangle \frown t_1 \,[\![X]\!]\, \langle\rangle &= \emptyset \\
\langle\rangle \,[\![X]\!]\, \langle x\rangle \frown t_2 &= \emptyset \\
\langle\rangle \,[\![X]\!]\, \langle\rangle &= \{\langle\rangle\} \\
\langle y\rangle \frown t_1 \,[\![X]\!]\, \langle x\rangle \frown t_2 &= \{\langle y\rangle \frown u \mid u \in t_1 \,[\![X]\!]\, \langle x\rangle \frown t_2\} \\
\langle y\rangle \frown t_1 \,[\![X]\!]\, \langle\rangle &= \{\langle y\rangle \frown u \mid u \in t_1 \,[\![X]\!]\, \langle\rangle\} \\
\langle x\rangle \frown t_1 \,[\![X]\!]\, \langle y\rangle \frown t_2 &= \{\langle y\rangle \frown u \mid u \in \langle x\rangle \frown t_1 \,[\![X]\!]\, t_2\} \\
\langle\rangle \,[\![X]\!]\, \langle y\rangle \frown t_2 &= \{\langle y\rangle \frown u \mid u \in \langle\rangle \,[\![X]\!]\, t_2\} \\
\langle y\rangle \frown t_1 \,[\![X]\!]\, \langle y'\rangle \frown t_2 = & \{\langle y\rangle \frown u \mid u \in t_1 \,[\![X]\!]\, \langle y'\rangle \frown t_2\} \\
& \cup \{\langle y'\rangle \frown u \mid u \in \langle y\rangle \frown t_1 \,[\![X]\!]\, t_2\}
\end{array}$$

  where $x, x' \in X \cup \{\checkmark\}$, $y, y' \notin X \cup \{\checkmark\}$, and $x \neq x'$,

- $(t \setminus X)$ is inductively defined by:

$$\begin{array}{rll}
\langle\rangle \setminus X &= \langle\rangle & \\
(\langle x\rangle \frown t) \setminus X &= t \setminus X & (\text{if } x \in X) \\
(\langle y\rangle \frown t) \setminus X &= \langle y\rangle \frown (t \setminus X) & (\text{if } y \notin X)
\end{array}$$

- $[[r]]^*$ is the smallest set satisfying the following inference rules:

$$\begin{array}{c}
\mathit{True} \Rightarrow (\langle\rangle, \langle\rangle) \in [[r]]^* \\
\mathit{True} \Rightarrow (\langle\checkmark\rangle, \langle\checkmark\rangle) \in [[r]]^* \\
(a, b) \in r \wedge (t, t') \in [[r]]^* \Rightarrow (a \frown t,\ b \frown t') \in [[r]]^*
\end{array}$$

- $[[r]]^{-1}(X)$ is defined as:

$$[[r]]^{-1}(X) = \{a \mid \exists\, b \in X.\ (a,\ b) \in r \vee a = b = \checkmark\}$$

- Restriction functions $T \downarrow n$ and $F \downarrow n$ are defined as:

$$\begin{array}{l}
T \downarrow n = \{t \in T \mid |t| \leq n\} \\
F \downarrow n = \{(t, X) \in F \mid |t| < n \vee (\exists\, t'.\ t = t' \frown \langle\checkmark\rangle, |t| = n)\}
\end{array}$$

Process equivalence $=_{\mathcal{F}}$ and process refinement $\sqsubseteq_{\mathcal{F}}$ over the stable failures model are then defined as usual:

$$\begin{array}{l}
P =_{\mathcal{F}} Q \Leftrightarrow \mathit{traces}(P) = \mathit{traces}(Q) \wedge \mathit{failures}(P) = \mathit{failures}(Q), \\
P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow \mathit{traces}(P) \supseteq \mathit{traces}(Q) \wedge \mathit{failures}(P) \supseteq \mathit{failures}(Q).
\end{array}$$

Furthermore, we use the following extended relations over process functions:

$$\begin{array}{l}
Pf =_{\mathcal{F}'} Qf \Leftrightarrow \forall\, x.\ Pf(x) =_{\mathcal{F}} Qf(x) \\
Pf \sqsubseteq_{\mathcal{F}'} Qf \Leftrightarrow \forall\, x.\ Pf(x) \sqsubseteq_{\mathcal{F}} Qf(x)
\end{array}$$

At first glance, the above defined input language of CSP-Prover seems to be weaker then full CSP as the generic internal choice operator $\sqcap \mathcal{P}$ is missing. However, we proved that the map $[\![\cdot]\!]_{\mathcal{F}}$ is surjective, thus our language is expressive with respect to the semantic domain $\mathcal{F}_\Sigma$. The proof of the theorem `surj_domF` is given in the theory file `CSP_F_surj.thy` in the package `CSP_F`.

6.3 Recursive process

Recursive processes can be effectively expressed by fixed points. For example, a buffer *Buffer*, which iteratively receives a real number r from the channel *in* and sends it to a channel *out* together with an increasing natural number *id*, can be defined by using a solution f of the following system of equations:

$$\begin{array}{l} f\,(\mathit{Empty}\,(\mathit{id})) \;=_{\mathcal{F}}\; \mathit{in}\,?\,r \;\rightarrow (\,f\,(\mathit{Full}\,(r,\mathit{id}))) \\ f\,(\mathit{Full}\,(r,\mathit{id})) \;=_{\mathcal{F}}\; \mathit{out}\,(r,\mathit{id}) \;\rightarrow (\,f\,(\mathit{Empty}\,(\mathit{id}+1))) \end{array}$$

where *Empty* and *Full* are *names*, and f is a function whose domain (i.e. the set of *process-indexes*) is

$$\mathit{Dom}(f) = \{\mathit{Empty}\,(\mathit{id}) \mid \mathit{id} \in \mathit{Nat}\} \cup \{\mathit{Full}\,(r,\mathit{id}) \mid r \in \mathit{Real}, \mathit{id} \in \mathit{Nat}\}$$

and whose range is the set of all processes. Any solution f is a fixed point (`FIX` *fun*) of the function $\mathit{fun} : (\mathit{Dom}(f) \Rightarrow \mathit{Proc}_\Sigma) \Rightarrow (\mathit{Dom}(f) \Rightarrow \mathit{Proc}_\Sigma)$ given as:

$$\begin{array}{l} \mathit{fun}(f)(\mathit{Empty}\,(\mathit{id})) := \mathit{in}\,?\,r \;\rightarrow (\,f\,(\mathit{Full}\,(r,\mathit{id}))) \\ \mathit{fun}(f)(\mathit{Full}\;(r,\mathit{id})) := \mathit{out}\,(r,\mathit{id}) \;\rightarrow (\,f\,(\mathit{Empty}\,(\mathit{id}+1))) \end{array}$$

Therefore, the process *Buffer*, which initially has no data and whose initial *id* is zero, is given as (`FIX` *fun*)(*Empty* (0)).

Now, we give a formal definition of recursive processes. At first, define the set $\mathit{Procfun}_\Sigma$ of *process functions*, which take one argument $f : I_p \rightarrow \mathit{Proc}_\Sigma$, as the smallest set which contains the following expressions:

$$\begin{array}{ll} (\lambda f.\ f(p)), & (\lambda f.\ \texttt{SKIP}), \\ (\lambda f.\ \texttt{STOP}), & (\lambda f.\ \texttt{DIV}), \\ (\lambda f.\ a \rightarrow E(f)), & (\lambda f.\ ?x : A \rightarrow E'(x)(f)), \\ (\lambda f.\ E_1(f) \,\Box\, E_2(f)), & (\lambda f.\ E_1(f) \sqcap E_2(f)), \\ (\lambda f.\ !!c : C \bullet E''(c)(f)), & (\lambda f.\ \texttt{IF}\ b\ \texttt{THEN}\ E_1(f)\ \texttt{ELSE}\ E_2(f)), \\ (\lambda f.\ E_1(f) \,[\![\, X \,]\!]\, E_2(f)), & (\lambda f.\ E(f) \setminus X), \\ (\lambda f.\ E(f)[[r]]), & (\lambda f.\ E_1(f) \fatsemi E_2(f)), \\ (\lambda f.\ E(f) \lfloor n), & \end{array}$$

where E, E_i, $E'(x)$, and $E''(c)$ are already in $\mathit{Procfun}_\Sigma$ for all x and c.

The set of process functions is denoted by $Procfun_\Sigma$. Intuitively, each process-index $p \in I_p$ can be correspond to a *process constant* or a *process name* in general process algebra. Thus, f can be a map for assigning each process constant to a process.

Next, the notion of *guard* is defined. In general, the variable f is said to be guarded in a process function E (or E is guarded) if each occurrence of the variable f is within some subexpression $a \rightarrow E(f)$ or $?\,x : A \rightarrow E(f)(x)$. However, we should deal with sequential composition $E_1(f) \fatsemi E_2(f)$ more carefully, thus the question is if E_2 should be guarded or not? For example, the following process functions (1) and (2) are guarded, but (3) is not guarded.

$$\begin{array}{ll} (1) & (\lambda f.\ (a \rightarrow \texttt{SKIP}) \fatsemi f(p)) \\ (2) & (\lambda f.\ \texttt{SKIP} \fatsemi (a \rightarrow f(p))) \\ (3) & (\lambda f.\ \texttt{SKIP} \fatsemi f(p)) \end{array}$$

It means E_2 does not have to be guarded if each occurrence of `SKIP` in E_1 are guarded. Therefore, before defining the notion of guarded, define an auxiliary notion, called *guarded-SKIP* process functions as follows: the set $gSKIPfun_\Sigma$ of guarded-SKIP process functions is the smallest set which contains the following expressions:

$$\begin{array}{ll} (\lambda f.\ \texttt{STOP}), & (\lambda f.\ \texttt{DIV}), \\ (\lambda f.\ a \rightarrow E(f)), & (\lambda f.\ ?x : A \rightarrow E'(x)(f)), \\ (\lambda f.\ GS_1(f) \,\Box\, GS_2(f)), & (\lambda f.\ GS_1(f) \sqcap GS_2(f)), \\ (\lambda f.\ !!c : C \bullet GS''(c)(f)), & (\lambda f.\ \texttt{IF}\ b\ \texttt{THEN}\ GS_1(f)\ \texttt{ELSE}\ GS_2(f)), \\ (\lambda f.\ GS_1(f) \,[\![\, X \,]\!]\, E_2(f)), & (\lambda f.\ E_1(f) \,[\![\, X \,]\!]\, GS_2(f)), \\ (\lambda f.\ GS(f)[[r]]), & \\ (\lambda f.\ GS_1(f) \fatsemi E_2(f)), & (\lambda f.\ E_1(f) \fatsemi GS_2(f)), \\ (\lambda f.\ GS(f) \,\lfloor\, n), & (\lambda f.\ P \,\lfloor\, 0), \end{array}$$

where GS, GS_i, $GS'(x)$, and $GS''(c)$ are already in $gSKIPfun_\Sigma$ for all x and c, and E, E_i, and $E'(x)$ are already in $Procfun_\Sigma$ for all x. The following property, which can be proven by induction (see `lemma gSKIP_to_Tick_notin_traces` in the theory-file `CSP_T_contraction.thy` in the package `CSP_T`), shows what we need for defining guarded process functions.

$$GS \in gSKIPfun_\Sigma \text{ implies } \langle\checkmark\rangle \notin traces(GS(f)) \text{ for all } f.$$

Furthermore, we need another subset of $Procfun_\Sigma$, which is the set of process functions which do not contain the hiding operator (more exactly, if $E(f) \setminus X$ then E must be a constant). The set $nohidefun_\Sigma$ of such process functions is

the smallest set which contains the following expressions:

$$
\begin{array}{ll}
(\lambda f.\ f(p)), & (\lambda f.\ \texttt{SKIP}), \\
(\lambda f.\ \texttt{STOP}), & (\lambda f.\ \texttt{DIV}), \\
(\lambda f.\ a \rightarrow H(f)), & (\lambda f.\ ?x : A \rightarrow H'(x)(f)), \\
(\lambda f.\ H_1(f) \,\Box\, H_2(f)), & (\lambda f.\ H_1(f) \sqcap H_2(f)), \\
(\lambda f.\ !!c : C \bullet H''(c)(f)), & (\lambda f.\ \texttt{IF}\ b\ \texttt{THEN}\ H_1(f)\ \texttt{ELSE}\ H_2(f)), \\
(\lambda f.\ H_1(f) \,[\![X]\!]\, H_2(f)), & \\
(\lambda f.\ P \setminus X), & (\lambda f.\ H(f)[[r]]), \\
(\lambda f.\ H_1(f) \fatsemi H_2(f)), & (\lambda f.\ H(f) \lfloor n), \\
\end{array}
$$

where H, H_i, $H'(x)$, and $H''(c)$ are already in $\mathit{nohidefun}_\Sigma$ for all x and c.

Then, the set $\mathit{gProcfun}_\Sigma$ of guarded process functions is defined as the smallest set which contains the following expressions:

$$
\begin{array}{ll}
(\lambda f.\ \texttt{SKIP}), & \\
(\lambda f.\ \texttt{STOP}), & (\lambda f.\ \texttt{DIV}), \\
(\lambda f.\ a \rightarrow H(f)), & (\lambda f.\ ?x : A \rightarrow H'(x)(f)), \\
(\lambda f.\ G_1(f) \,\Box\, G_2(f)), & (\lambda f.\ G_1(f) \sqcap G_2(f)), \\
(\lambda f.\ !!c : C \bullet G''(c)(f)), & (\lambda f.\ \texttt{IF}\ b\ \texttt{THEN}\ G_1(f)\ \texttt{ELSE}\ G_2(f)), \\
(\lambda f.\ G_1(f) \,[\![X]\!]\, G_2(f)), & \\
(\lambda f.\ P \setminus X), & (\lambda f.\ G(f)[[r]]), \\
(\lambda f.\ GS_1(f) \fatsemi H_2(f)), & (\lambda f.\ G_1(f) \fatsemi G_2(f)), \\
(\lambda f.\ G(f) \lfloor n), & \\
\end{array}
$$

where G, G_i, $G'(x)$, and $G''(c)$ are already in $\mathit{gProcfun}_\Sigma$ for all x and c, GS_1 is already in $\mathit{gProcfun}_\Sigma \cap \mathit{gSKIPfun}_\Sigma$, and H and $H'(x)$ are in $\mathit{nohidefun}_\Sigma$ for all x.

Then, the sets $\mathit{ProcFun}_\Sigma$ and $\mathit{gProcFun}_\Sigma$ of functions such as *fun* used in the example *Buffer*, are defined as follows:

$$
\begin{array}{rl}
\mathit{ProcFun}_\Sigma & = \{\mathit{fun} \mid (\forall\, p.\ (\lambda f.\ \mathit{fun}(f)(p) \in \mathit{Procfun}_\Sigma\} \\
\mathit{gProcFun}_\Sigma & = \{\mathit{gfun} \mid (\forall\, p.\ (\lambda f.\ \mathit{gfun}(f)(p) \in \mathit{gProcfun}_\Sigma\}
\end{array}
$$

Now, we finished the preparation to express fixed-points in our Csp dialect. Csp offers two standard approaches to deal with fixed-points: complete partial orders (cpo) with Tarski's fixed point theorem or complete metric spaces (cms) with Banach's fixed point theorem. The limits (`FIX` *fun*) and (`FIX!` *fun*) of

the converging sequences in Tarski's and Banach's fixed point theorems can be defined in our Csp-dialect as follows:

$$(\texttt{FIX}\ \mathit{fun})(p) := \ !\texttt{nat}\ n \bullet ((\mathit{fun}^{(n)}(\lambda\, y.\, \texttt{DIV}))(p))$$
$$(\texttt{FIX!}\ \mathit{gfun})(p) := \ !\texttt{nat}\ n \bullet (((\mathit{gfun}^{(n)}(\lambda\, y.\, \mathit{Any}))(p))\, \lfloor\, n)$$

where $\mathit{fun} \in \mathit{ProcFun}_\Sigma$, $\mathit{gfun} \in \mathit{gProcFun}_\Sigma$, and DIV plays the role of the bottom element in the cpo approach and *Any* stands for any process, which corresponds to the arbitrary initial point of Banach's theorem. Then, as expected, the following properties hold:

1. Let $\mathit{fun} \in \mathit{ProcFun}_\Sigma$, then $(\texttt{FIX}\ \mathit{fun}) =_{\mathcal{F}'} \mathit{fun}\,(\texttt{FIX}\ \mathit{fun})$ and for any f, if $f =_{\mathcal{F}'} \mathit{fun}(f)$ then $f \sqsubseteq_{\mathcal{F}'} (\texttt{FIX}\ \mathit{fun})$. Thus, $(\texttt{FIX}\ \mathit{fun})$ is the greatest fixed point on $\sqsubseteq_{\mathcal{F}'}$, in other words, it is the least fixed point in the semantic domain.
2. Let $\mathit{gfun} \in \mathit{gProcFun}_\Sigma$, then $(\texttt{FIX!}\ \mathit{gfun}) =_{\mathcal{F}'} \mathit{gfun}\,(\texttt{FIX!}\ \mathit{gfun})$ and for any f, if $f =_{\mathcal{F}'} \mathit{gfun}(f)$ then $f =_{\mathcal{F}'} (\texttt{FIX!}\ \mathit{gfun})$. Thus, $(\texttt{FIX!}\ \mathit{gfun})$ is the unique fixed point on $=_{\mathcal{F}'}$.

Thus, both ways of Csp of dealing with systems of recursive equations, the cpo approach using Tarski's fixed point theorem as well as the cms approach using Banach's fixed point theorem, are expressible in the input language of Csp-Prover.

The fixed points $(\texttt{FIX}\ \mathit{fun})$ and $(\texttt{FIX!}\ \mathit{fun})$ are very powerful to express mutual recursive processes, while it is sometimes convenient to simply use single recursive processes such as $(\mu\, X \bullet\ R(X))$ which is the greatest fixed point of the single equation $X =_{\mathcal{F}} R(X)$. Such singe recursive processes can be defined as syntactic sugar:

$$(\mu\ X \bullet R(X)) := (\texttt{FIX}\ (\lambda f.\ \lambda p.\ R(f(\texttt{pmu}))))(\texttt{pmu})$$
$$(\mu!\, X \bullet R(X)) := (\texttt{FIX!}\ (\lambda f.\ \lambda x.\ R(f(\texttt{pmu}))))(\texttt{pmu})$$

where pmu is a dummy process-index. Furthermore, the sets ProcX_Σ and gProcX_Σ of such functions $R(\cdot) : \mathit{Proc}_\Sigma \Rightarrow \mathit{Proc}_\Sigma$ are defined from $\mathit{ProcFun}_\Sigma$ and $\mathit{gProcFun}_\Sigma$, respectively:

$$\mathit{ProcX}_\Sigma := \{R \mid (\lambda f.\ R(f(\texttt{pmu}))) \in \mathit{ProcFun}_\Sigma\}$$
$$\mathit{gProcX}_\Sigma := \{R \mid (\lambda f.\ R(f(\texttt{pmu}))) \in \mathit{gProcFun}_\Sigma\}$$

Then, as expected, the following properties hold:

1. Let $R \in \mathit{ProcX}_\Sigma$, then $(\mu\ X \bullet R(X)) =_{\mathcal{F}} R(\mu\ X \bullet R(X))$ and for any P, if $P =_{\mathcal{F}} R(P)$ then $P \sqsubseteq_{\mathcal{F}} (\mu\ X \bullet R(X))$. Thus, $(\mu\ X \bullet R(X))$ is the greatest fixed point of R on $\sqsubseteq_{\mathcal{F}}$.
2. Let $R \in \mathit{gProcX}_\Sigma$, then $(\mu!\, X \bullet R(X)) =_{\mathcal{F}} R(\mu\ X \bullet R(X))$ and for any P, if $P =_{\mathcal{F}} R(P)$ then $P =_{\mathcal{F}} (\mu!\, X \bullet R(X))$. Thus, $(\mu!\, X \bullet R(X))$ is the unique fixed point of R on $=_{\mathcal{F}}$.

7 Encoding of the CSP-dialect

This section shows how the Csp-dialect introduced in Section 6 is encoded in the generic theorem prover Isabelle.

7.1 Syntax

At first, we give ASCII style expressions of Csp processes in Figure 3 because the conventional operators use TeX symbols[4]. You will need Figure 3 when you use Csp-Prover in fact. However, we consistently continue to use the conventional symbols such as $\Box$ instead of `[+]` in this User-Guide because the conventional symbols allow this guide to be readable and they are almost available in the X-Symbol mode in the Proof-General which is an XEmacs-like interface of Isabelle. Also, we use conventional symbols on set, logic, etc, as used in the Isabelle-tutorial[NPW02], for example, $a \in X$ and $X \subseteq Y$ are used instead of a`:`X and X`<=`Y, respectively.

Now, the set of (basic) processes is given as a recursive type `'a proc` which is defined by the Isabelle command **datatype** as shown in Figure 4, where `'a` is the type of Σ. Since each selector is either a set of communications or a natural number, the type of selectors is defined with help of the type constructors `sel_set` and `sel_nat` in Figure 4. Furthermore, note the definitions of prefix choice and replicated internal choice. For example, the following process, which receives a value 0 or 1 and thereafter if the value is 0 then it successfully terminate else deadlocks,

$$? \, n : \{0, 1\} \rightarrow (\texttt{IF} \ (n = 0) \ \texttt{THEN SKIP ELSE STOP})$$

is defined by

$$? : \{0, 1\} \rightarrow (\lambda \, n. \ (\texttt{IF} \ (n = 0) \ \texttt{THEN SKIP ELSE STOP}))$$

because bound variables such as n are not used in the definition by **datatype**. But, this inconvenience is easily solved by the Isabelle commands **syntax** and **transform**, which make syntactic sugars, as shown in Figure 5.

Also, derived operators such as $\triangleright$ are defined by **syntax** and **transform**, with the help of **consts** to declare types and **defs** to define functions. We give some of them in Figure 6. You can find all the definitions of process expressions in the the theory-file `CSP_syntax.thy` in the package `CSP`.

[4] We had to use the ASCII symbols slightly different from the machine readable processes Csp-M used in FDR, in order to avoid overloading of symbols which Isabelle had already used.

Conventional symbol	ASCII symbol	Name
`SKIP`	`SKIP`	successful terminating process
`STOP`	`STOP`	deadlock process
`DIV`	`DIV`	divergence
$a \rightarrow P$	a `->` P	action prefix
$? \ x : A \rightarrow P(x)$	`?` x `:` A `->` $P(x)$	prefix choice
$P \ \Box \ Q$	P `[+]` Q	external choice
$P \sqcap Q$	P `|~|` Q	internal choice
$!! \ c : C \bullet P(c)$	`!!` c `:` C `..` $P(c)$	replicated internal choice
`IF` b `THEN` P `ELSE` Q	`IF` b `THEN` P `ELSE` Q	conditional
$P \ [\![X]\!] \ Q$	P `|[` X `]|` Q	generalized parallel
$P \setminus X$	P `--` X	hiding
$P[[r]]$	P `[[`r`]]`	relational renaming
$P \fatsemi Q$	P `;;` Q	sequential composition
$P \lfloor n$	P `|.` n	depth restriction
$P \triangleright Q$	P `[>` Q	timeout
`!set` $A : \mathcal{A} \bullet P(A)$	`!set` A `:` $\mathcal{A}$ `..` $P(A)$	replicated internal choice over $\mathbb{P}(\Sigma)$
`!nat` $n : N \bullet P(n)$	`!nat` n `:` N `..` $P(n)$	replicated internal choice over Nat
$! \ x : A \bullet P(x)$	$!$ x `:` A `..` $P(x)$	replicated internal choice over Σ
$!\langle f \rangle \ z : Z \bullet P(z)$	`!<`f`>` z `:` Z `..` $P(z)$	replicated internal choice with f
$! \ x : A \rightarrow P(x)$	$!$ x `:` A `->` $P(x)$	internal prefix choice
$a!v \rightarrow P$	a`!`v `->` P	sending
$a?x : X \rightarrow P(x)$	a`?`x`:`X `->` $P(x)$	receiving
$a!?x : X \rightarrow P(x)$	a`!?`x`:`X `->` $P(x)$	non-deterministic sending
$P \ \|\|\| \ Q$	P `|||` Q	interleaving
$P \ \| \ Q$	P `||` Q	synchronous
$P \ [\![X, Y]\!] \ Q$	P `|[X,Y]|` Q	alphabetized parallel
$[\|\|] \ i : I \bullet (P_i, X_i)$	`[||]` i`:`I `..` (P_i, X_i)	replicated alphabetized parallel
`FIX` *fun*	`FIX` *fun*	Tarski's fixed point
`FIX!` *fun*	`FIX!` *fun*	Banach's fixed point
$\mu \ X \bullet F(X)$	`MU` X `..` $F(X)$	Tarski's fixed point (single)
$\mu! \ X \bullet F(X)$	`MU!` X `..` $F(X)$	Banach's fixed point (single)

Figure 3: The ASCII expression of CSP processes.

7.2 Domain

In this subsection, we encode the domain for the stable-failures model $\mathcal{F}$. However, first of all, we briefly explain how to define a new type from an existing type by the Isabelle command **typedef**. It defines a new type as a non-empty subset of an existing type:

typedef `SubType = {`x`::SuperType. `*Pred*`(`x`)}`

Here, *Pred* is a predicate over the existing type `SuperType`, and `SubType` is the

```
datatype 'a selector = sel_set "'a set"  |  sel_nat "nat"

datatype
 'a proc
  = STOP
  | SKIP
  | DIV
  | Act_prefix     "'a" "'a proc"                (" _ → _")
  | Ext_pre_choice "'a set" "'a ⇒ 'a proc"       ("? : _ → _")
  | Ext_choice     "'a proc" "'a proc"           ("_ □ _")
  | Int_choice     "'a proc" "'a proc"           ("_ ⊓ _")
  | Rep_int_choice "'a selector set"
                   "'a selector ⇒ 'a proc"       ("!! : _ • _")
  | IF             "bool" "'a proc" "'a proc"    ("IF _ THEN _ ELSE _")
  | Parallel       "'a proc" "'a set" "'a proc"  ("_ ⟦_⟧ _")
  | Hiding         "'a proc" "'a set"            ("_ \ _")
  | Renaming       "'a proc" "('a * 'a) set"     ("_[[_]]")
  | Seq_compo      "'a proc" "'a proc"           ("_ ⨟ _")
  | Depth_rest     "'a proc" "nat"               ("_ ⌊ _")
```

Figure 4: The recursive definition of the process type.

```
syntax
 "@Ext_pre_choice" ::
  "pttrn ⇒ 'a set ⇒ 'a proc ⇒ 'a proc" ("?_ : _ → _")

 "@Rep_int_choice" ::
  "pttrn ⇒ ('a selector) set ⇒ 'a proc ⇒ 'a proc" ("!!_ : _ • _")

translations
 "? x : X → P" == "? : X → (λx. P)"
 "!! c : C • P"  == "!! : C • (λc. P)"
```

Figure 5: The expression of bound variables.

newly defined type by the subset. When a new type is defined by **typedef**, a set and two type-converters are automatically declared for relating the new type with the existing type.

```
    SubType :: SuperType set,
Rep_SubType :: SuperType ⇒ SubType,
Abs_SubType :: SubType ⇒ SuperType.
```

Then, the set `SubType` is defined as $\{x$`::SuperType.` $Pred$`(`x`)`$\}$, and the following properties are asserted:

```
%% timeout
syntax
 "_Timeout" :: "'a proc ⇒ 'a proc ⇒ 'a proc"      ("_ ▷ _")
translations
 "P ▷ Q" == "(P ⊓ STOP) □ Q"

%% replicated internal choice over sets
consts
 Rep_int_choice_fun ::
   "('b ⇒ ('a selector)) ⇒ 'b set
     ⇒ ('b ⇒ 'a proc) ⇒ 'a proc"           ("!!⟨_⟩ : _ • _")
defs
 Rep_int_choice_fun_def :
  "!!⟨f⟩ : X • P == !! s : (f ‘ X) • (P((inv f) s))"
syntax
 "@Rep_int_choice_set" ::
   "pttrn ⇒ ('a set) set
     ⇒ ('a set => 'a proc) ⇒ 'a proc"       ("!set : _ • _")
translations
 "!set : X • P" == "!!⟨sel_set⟩ : X • P"

%% replicated internal choice over communications
consts
 Rep_int_choice_com ::
   "('a set ⇒ ('a ⇒ 'a proc) ⇒ 'a proc"       ("! : _ • _")
defs
 Rep_int_choice_com_def :
  "! : A • P == !set X : {{a} | a. a ∈ A} • P(contents(X))"
```

Figure 6: The definitions of derived operators.

$$\begin{aligned} \texttt{Rep\_SubType}\ s \in \texttt{SubType}, \\ \texttt{Abs\_SubType (Rep\_SubType}\ s\texttt{)} = s, \\ s \in \texttt{SubType} \Rightarrow \texttt{Rep\_SubType (Abs\_SubType}\ s\texttt{)} = s, \end{aligned}$$

where the name `SubType` is used as both a type and a set.

Now, let us start defining the type of domain for the stable-failures model $\mathcal{F}$. At first, the type of events which consist of communications (whose type is `'a`) and the termination symbol ✓ (written `Tick` in ASCII) is defined as follows:

```
datatype 'a event = Ev 'a  |  ✓
```

Then, the type of traces which may have the successful termination symbol ✓ in the last place as follows:

```
typedef 'a trace = "{s::('a event list). ✓ ∉ set (butlast s)}"
```

where the function `butlast` removes the last element of the list s and the

function `set` transforms a list to a set of elements contained in the list. The basic operators over traces are defined from the corresponding operators over lists with help of type-converters `Rep_domT` and `Abs_domT`. For example, the concatenate operator $\frown$ (written `^^` in ASCII) over traces is defined from the concatenate operator `@` over lists as follows:

```
consts
  appt :: "'a trace ⇒ 'a trace ⇒ 'a trace" (infixr "⌢" 65)
defs
  appt_def : "s ⌢ t == Abs_trace (Rep_trace s @ Rep_trace t)"
```

See the theory-file `Trace.thy` in the package `CSP` for more details. Many useful lemmas on traces such as associativity are also given there.

Secondly, the type of domain for the traces model $\mathcal{T}$ is defined as the set of subsets of traces which satisfy the healthiness condition **T1** (i.e. non-empty and prefix closed) as follows:

```
typedef 'a domT = "{T::('a trace set). HC_T1(T)}"
```

where `HC_T1` is the encoded healthiness condition **T1**.

Isabelle has provided a *type class* of types together with a *partial order* $\leq$ (written `<=` in ASCII), and lemmas and theorems on such types have been proven. Such lemmas and theorems can be applied to newly defined types, provided such order $\leq$ over the types is defined and is proven to be a partial order. In the case of `domT`, such order $\leq$ over `domT` can be defined from the inclusion $\subseteq$ as follows:

```
defs (overloaded)
   subdomT_def : "T ≤ S == (Rep_domT T) ⊆ (Rep_domT S)"
```

where "overloaded" means $\leq$ (to be proven to be a partial order) is instantiated. See the theory-file `Domain_T.thy` in the package `CSP_T` for more details.

In the same way as `domT`, the set of failures satisfying the healthiness condition **F2** is given as a type as follows:

```
typedef 'a setF = "{F::('a failure set). HC_F2(F)}"
```

where `'a failure` is a synonym which can be defined by the Isabelle command **types**:

```
types 'a failure = "'a trace * 'a event set"
```

where `*` is a type constructor of pairs. And the partial order $\leq$ over `setF` is also overloaded as follows:

```
defs (overloaded)
   subsetF_def : "F ≤ E == (Rep_setF F) ⊆ (Rep_setF E)"
```

See the theory-file `Set_F.thy` in the package `CSP_F` for more details.

Then, the type of domain for the stable-failures model $\mathcal{F}$ is defined as the set of subsets of pairs of traces and failures which satisfy all the healthiness conditions:

```
typedef
  'a domF = "{TF::('a domT * 'a setF).
               HC_T2(TF) ∧ HC_T3(TF) ∧ HC_F3(TF) ∧ HC_F4(TF)}"
```

where `HC_T2`, `HC_T3`, `HC_F3`, and `HC_F4` are the encoded healthiness conditions **T2**, **T3**, **F3**, and **F4**. For example, **T3** is encoded as follows:

```
consts HC_T3 :: "('a domT * 'a setF) ⇒ bool"
defs
  HC_T3_def :
    "HC_T3 TF == ∀ t. (t ⌢ ⟨✓⟩ ∈t (fst TF) ∧ noTick t)
                      ⟶ (∀ X. (t ⌢ ⟨✓⟩ , X) ∈f (snd TF))"
```

where `fst` and `snd` are the functions for extracting the components of a pair: $\texttt{fst}(x, y) = x$ and $\texttt{snd}(x, y) = y$. The subscripts `t` and `f` are attached to operators on traces and failures, respectively, e.g. $\in_{\texttt{t}}$ and $\in_{\texttt{f}}$. The condition `noTick` t means that the trace t does not contain ✓. This condition is not explicitly written in the definition of **T3** shown in Subsection 6.2 because $t \frown \langle ✓ \rangle$ implicitly implies that t has no ✓. On the other hand, $\frown$ is a total function[5] because Isabelle does not allow us to define truly partial functions. Therefore, the condition `noTick` t is necessary. See the theory-file `Domain_F.thy` in the package `CSP_F` for more details.

The partial order over `domF` is defined as the combination of the partial orders $\leq$ over `domT` and `setF`:

```
defs (overloaded)
   subdomF_def : "SF1 ≤ SF2 == (Rep_domF SF1) ≤ (Rep_domF SF2)"
```

where `(Rep_domF` SF`)` has the type `('a domT * 'a setF)`, and the order over pairs is defined in the usual way (see `Infra_pair.thy` in `CSP`):

```
defs (overloaded)
  order_pair_def : "x ≤ y == (fst x) ≤ (fst y) ∧ (snd x) ≤ (snd y)"
```

In this case, it can be easily proven that $\leq$ over `domF` is a partial order indeed. That means (`domT`, $\leq$) can be proven to be an instance of the type class of partial ordered set as follows:

```
instance domF :: (type) order
```

[5] For example, $\langle ✓ \rangle \frown \langle ✓ \rangle$ is meaningless and cannot be interpreted to $\langle ✓, ✓ \rangle$, but such application $\langle ✓ \rangle \frown \langle ✓ \rangle$ of $\frown$ is not forbidden.

```
consts traces ::  "'a proc ⇒ 'a domT"

primrec
 "traces(STOP)       = {⟨⟩}ₜ"
 "traces(SKIP)       = {⟨⟩, ⟨✓⟩}ₜ"
 "traces(DIV)        = {⟨⟩}ₜ"
 "traces(a → P)      = {t. t = ⟨⟩ ∨
                        (∃s. t = ⟨Ev a⟩ ⌢ s ∧ s ∈ₜ traces(P))}ₜ"
 "traces(? : X → P)  = {t. t = ⟨⟩ ∨
                        (∃a s. t = ⟨Ev a⟩ ⌢ s ∧ s ∈ₜ traces(P a)∧ a ∈ X)}ₜ"
 "traces(P □ Q)      = traces(P) ∪ₜ traces(Q)"
 "traces(P ⊓ Q)      = traces(P) ∪ₜ traces(Q)"
 "traces(!! : C • P) = {t. t = ⟨⟩ ∨ (∃c ∈ C. t ∈ₜ traces(P c)) }ₜ"
 "traces(IF b THEN P ELSE Q) = (if b then traces(P) else traces(Q))"
 "traces(P ⟦ X ⟧ Q)  = {u. ∃s t. u ∈ s ⟦ X ⟧tr t ∧
                              s ∈ₜ traces(P)∧ t ∈ₜ traces(Q) }ₜ"
 "traces(P \ X)      = {t. ∃s. t = s \tr X ∧ s ∈ₜ traces(P) }ₜ"
 "traces(P[[r]])     = {t. ∃s. s [[r]]* t ∧ s ∈ₜ traces(P) }ₜ"
 "traces(P ⨟ Q)      = {u. (∃s. u = rmTick s ∧ s ∈ₜ traces(P)) ∨
                           (∃s t. u = s ⌢ t ∧ s ⌢ ⟨✓⟩ ∈ₜ traces(P) ∧
                                  t ∈ₜ traces(Q) ∧ noTick s) }ₜ"
 "traces(P ⌊ n)      = traces(P) ↓ n"
```

Figure 7: The encoding of the function *traces*

7.3 Semantics

The functions *traces* and *failures* for giving the meaning of processes are recursively defined by **primrec** which is an Isabelle command used for defining functions whose argument has a recursive type defined by **datatype** such as `proc`, see Figures 7 and 8. You will find the definitions of *traces* and *failures* in the theory-file `CSP_T_semantics.thy` in the package `CSP_T` and `CSP_F_semantics.thy` in `CSP_F`, respectively.

The encodings of the auxiliary functions $[\![X]\!]_{tr}$ and $\backslash_{tr}$ (see Subsection 6.2 for the definitions) over traces are given in `Trace_par.thy` and `Trace_hide.thy` respectively, and the encodings of $[[r]]^*$ and $[[r]]^{inv}$ are given in `Trace_ren.thy` in the package `CSP`. Also, (`rmTick` s) is the trace obtained by removing ✓ from s and it is encoded in `Trace_seq.thy`. Furthermore, $\downarrow$ (rewritten `.|.` in ASCII) is a restriction function which is given over both `domT` and `setF`, in `Domain_T_cms.thy` in the package `CSP_T` and `Set_F_cms.thy` in `CSP_F`, respectively.

Here, it is noted that the meaning $[\![P]\!]_{\mathcal{F}}$ of each process P is not exactly (`traces(`P`), failures(`P`)`), but is `Abs_domF (traces(`P`), failures(`P`))` because the type `'a domF` is a subtype of `('a domT * 'a setF)`, defined by **typedef**. Therefore, it is convenient to define the following notations:

```
consts failures ::  "'a proc ⇒ 'a setF"

primrec
 "failures(STOP)       = {f . ∃X . f = (⟨⟩,X) }f"
 "failures(SKIP)       = {f . (∃X . f = (⟨⟩,X) ∧ X ⊆Evset) ∨
                              (∃X . f = (⟨✓⟩,X)) }f"
 "failures(DIV)        = {}f"
 "failures(a → P)      = {f . (∃X . f = (⟨⟩,X) ∧ Ev a ∉ X) ∨
                              (∃s X . f = (⟨Ev a⟩⌢s,X) ∧
                                    (s, X) ∈ failures(P)) }f"
 "failures(? : X → P)  = {f . (∃Y . f = (⟨⟩,Y) ∧ (Ev`X)∩Y = {}) ∨
                              (∃a s Y . f = (⟨Ev a⟩⌢s,X) ∧
                                    (s, X) ∈ failures(P a) ∧ a ∈ X) }f"
 "failures(P □ Q)      = {f . (∃X . f = (⟨⟩,X) ∧
                                   f ∈f failures(P) ∩f failures(Q)) ∨
                              (∃s X . f = (s,X) ∧ s ≠ ⟨⟩ ∧
                                   f ∈f failures(P) ∪f failures(Q)) ∨
                              (∃X . f = (⟨⟩,X) ∧ X ⊆ Evset ∧
                                   ⟨✓⟩ ∈t traces(P) ∪t traces(Q)) }f"
 "failures(P ⊓ Q)      = failures(P) ∪f failures(Q)"
 "failures(!! : C • P)  = {f . (∃c ∈ C . f ∈f failures(P c)) }f"
 "failures(IF b THEN P ELSE Q) = (if b then failures(P) else failures(Q))"
 "failures(P ⟦X⟧ Q)    = {f . ∃u Y Z . f = (u, Y ∪ Z) ∧
                              Y-((Ev`X) ∪ {✓}) = Z-((Ev`X) ∪ {✓}) ∧
                              (∃s t. u ∈ s ⟦X⟧tr t ∧ (s,Y)∈f failures(P)∧
                                                     (t,Z)∈f failures(Q))}f"
 "failures(P \ X)      = {f . ∃s Y . f = (s \tr X,Y) ∧
                              (s, (Ev`X)∪Y)∈f failures(P) }f"
 "failures(P[[r]])     = {f . ∃s t X . f = (t,X) ∧ s [[r]]* t ∧
                              (s, [[r]]inv X)∈f failures(P) }f"
 "failures(P ⨟ Q)      = {f . (∃t X . f = (t,X) ∧
                                 (t,X ∪ {✓})∈f failures(P) ∧ noTick t) ∨
                              (∃s t X . f = (t⌢t,X) ∧
                                 s⌢⟨✓⟩ ∈t traces(P) ∧
                                 (t,X)∈f failures(Q) ∧ noTick s) }f"
 "failures(P ⌊ n)      = failures(P) ↓ n"
```

Figure 8: The encoding of the function *failures*

```
consts
  pairF::  "'a domT ⇒ 'a setF ⇒ 'a domF" ("(_ ,, _)")
  fstF ::  "'a domF ⇒ 'a domT"
  sndF ::  "'a domF ⇒ 'a setF"

defs
  pairF_def:  "(T ,, F) == Abs_domF (T, F)"
  fstF_def :  "fstF == fst o Rep_domF"
  sndF_def :  "sndF == snd o Rep_domF"
```

Then, the semantics $[\![P]\!]_{\mathcal{F}}$, the process equivalence $=_{\mathcal{F}}$, and the process refinement $\sqsubseteq_{\mathcal{F}}$ are defined as follows (see `CSP_F_semantics.thy`):

```
consts
 semF :: "'a proc ⇒ 'a domF"          ("[![_]!]_F")
 refF :: "'a proc ⇒ 'a proc ⇒ bool"   ("_ ⊑_F _")
 eqF  :: "'a proc ⇒ 'a proc ⇒ bool"   ("_ =_F _")

defs
  semF_def: "[![P]!]_F == (traces(P) ,, failures(P))"
  refF_def: "P ⊑_F Q == [![Q]!]_F ≤ [![P]!]_F"
  eqF_def : "P =_F Q == [![P]!]_F = [![Q]!]_F"
```

Here, it is important to check that `(traces(`P`), failures(`P`))` is in `domF` indeed. It is proven in the following lemma (see `CSP_F_domain.thy`):

```
lemma proc_domF[simp]:  "(traces(P), failures(P)) ∈ domF"
```

This lemma allows us to prove the following expected properties:

```
lemma fstF_semF[simp]: "fstF [![P]!]_F = traces(P)"
lemma sndF_semF[simp]: "sndF [![P]!]_F = failures(P)"
lemma cspF_eqF_semantics:
   "(P =_F Q) = ((traces(P) = traces(Q))∧
                 (failures(P) = failures(Q)))"
lemma cspF_refF_semantics:
   "(P ⊑_F Q) = ((traces(Q) ≤ traces(P))∧
                 (failures(Q) ≤ failures(P)))"
```

At the end of this subsection, we would like to briefly tell the expressive power of our Csp dialect. At first glance, the input language of Csp-Prover seems to be weaker than full Csp as the generic internal choice operator $\sqcap \mathcal{P}$ [6] missing. However, we have proven the following theorem which shows that our language to be expressive with respect to the stable-failures domain.

```
theorem EX_proc_domF: "∀ SF. ∃ P. [![P]!]_F = SF"
```

[6] $\mathcal{P}$ is an non-emptyset of processes and the semantics is given as:

$$traces(\sqcap \mathcal{P}) = \bigcup\{traces(P) \mid P \in \mathcal{P}\}$$
$$failures(\sqcap \mathcal{P}) = \bigcup\{failures(P) \mid P \in \mathcal{P}\}$$

```
datatype Event = In real | Out "real * nat"
datatype BufferName = Empty nat | Full real nat

consts funDef :: "(BufferName, Event) procDef"

recdef funDef "{}"
 "funDef (f, (Empty n))  = In ? r -> (f (Full r n))"
 "funDef (f, (Full r n)) = Out (r,n) -> (f (Empty (Suc n)))"

syntax     "_fun" :: "(BufferName, Event) procFun" ("fun")
translations "fun" == "curry funDef"

consts Buffer :: "Event proc"
defs Buffer_def: "Buffer == (FIX fun)(Empty 0)"
```

Figure 9: An encoding of the example *Buffer* (also see Subsection 6.3)

This theorem and the proof are given in `CSP_F_surj.thy` in the package `CSP_F`.

7.4 Recursive process

The important task to describe recursive processes such as *Buffer* shown in Subsection 6.3 is to define functions such as *fun* whose fixed point is used. The most elegant way to define such functions is to use the Isabelle command **recdef**. For example, *Buffer* can be expressed in CSP-Prover as shown in Figure 9 [7]. Note that `funDef` is a non-recursive function, thus the measure is just the empty set {}. The reason why **recdef** is used here is that *pattern matching* on the first argument[8] is available in definition by **recdef**. The pattern matching allows us to define recursive processes in a readable way.

As mentioned in Subsection 6.3, `(FIX fun)` works well as the least fixed point of `fun` only if `fun` $\in$ `ProcFun`, and `(FIX! gfun)` works well as the unique fixed point only if `gfun` $\in$ `gProcFun`. In CSP-Prover, the sets of `ProcFun` and `gProcFun` are defined in the theory file `CSP_syntax.thy` in the package `CSP` as explained in Subsection 6.3, and the proofs `fun` $\in$ `ProcFun` and `gfun` $\in$ `ProcFun` are automatized well. For example, `fun` $\in$ `ProcFun` of the example *Buffer* can be proven by the following three steps (see `Test_Buffer.thy` in `Test`):

[7] The types `procFun` and `procDef` are synonyms defined as follows:

```
types ('p,'a) procFun = "('p => 'a proc) => ('p => 'a proc)"
types ('p,'a) procDef = "(('p => 'a proc) * 'p) => 'a proc"
```

[8] In definition of recursive processes, it is important to apply the pattern matching to the second argument (i.e. process indexes) such as `BufferName`, however the pattern matching by **recdef** is available *only* for the first argument. Therefore, the first argument `f` and the second argument of `funDef` are combined into a tuple, then it is separated by the function `curry`, see the definition of `fun`.

```
lemma ProcFun_fun[simp]: "fun ∈ ProcFun"   %% goal to be proven
apply (auto simp add: ProcFun_def)         %% unfolding ProcFun_def
apply (induct_tac p)                       %% induction on indexes
apply (auto)                               %% automatic proof
done
```

By applying the first command, the following subgoal is displayed.

```
goal (lemma (ProcFun_fun), 1 subgoal):
1. ⋀p. (λX. funDef (X, p)) ∈ Procfun
```

Next, in order to instantiate `p` whose type is `BufferName`, structural induction on `p` is applied by `(induct_tac p)` because `BufferName` is defined by **datatype** (note: it is not important whether `p` is recursively defined or not):

```
goal (lemma (ProcFun_fun), 2 subgoals):
1. ⋀p nat. (λX. funDef (X, Empty nat)) ∈ Procfun
2. ⋀p real nat. (λX. funDef (X, Full real nat)) ∈ Procfun
```

Then, the subgoals can be automatically proven. In general, this proof strategy is available for `fun ∈ ProcFun` as well as for `fun ∈ gProcFun`.

8 Verification

In order to verify the process equivalence $P =_{\mathcal{F}} Q$ and the process refinement $P \sqsubseteq_{\mathcal{F}} Q$ in CSP-Prover, CSP-Prover provides three kinds of strategies: (1) semantical proof by the definition of `traces` and `failures`, (2) manually syntactical proof by algebraic CSP laws, and (3) semi-automatically syntactical proof by tactics. It is recommended to take a look at the theory file `Test_proof.thy` in the package `Test`. The theory file gives three different proofs mentioned above of the following equality:

$$((a \rightarrow P) \,[\![\, \{a\} \,]\!]\, (a \rightarrow Q)) =_{\mathcal{F}} a \rightarrow (P \,[\![\, \{a\} \,]\!]\, Q)$$

8.1 Semantical proof

In this proof style, the important lemmas are `cspF_eqF_semantics`, `in_traces`, and `in_failures`. The lemma `cspF_eqF_semantics` shown in Subsection 7.3 translates the equality $=_{\mathcal{F}}$ into the equality over `traces` and `failures`, then lemmas `in_traces` and `in_failures` interpret `traces(`P`)` and `failures(`P`)` in accordance with the semantic clauses, see Figures 7 and 8. For example, when a subgoal contains the following form,

$$\cdots \wedge\ t \in_{\mathtt{t}} \mathtt{traces}(a \rightarrow P)\ \wedge \cdots$$

and if the command `(simp add: in_traces)` is applied, then it will be rewritten to the following subgoal (∗1):

$$\cdots \wedge \ (t = \langle\rangle \ \vee \ (\exists\, s.\ t = \langle \mathtt{Ev}\ \ a\rangle \frown s \wedge s \in_{\mathtt{t}} \mathtt{traces}(P))) \ \wedge \cdots$$

On the other hand, if the command `(simp add: traces.simps)`[9] was applied instead of `(simp add: in_traces)`, it would be rewritten to the following subgoal (∗2):

$$\cdots \wedge \ (t \in_{\mathtt{t}} \ \{t = \langle\rangle \ \vee \ (\exists\, s.\ t = \langle \mathtt{Ev}\ \ a\rangle \frown s \wedge s \in_{\mathtt{t}} \mathtt{traces}(P))\}_{\mathtt{t}}) \ \wedge \cdots$$

Here, note that it is not trivial to transform the subgoal (∗2) to (∗1) because $(t \in_{\mathtt{t}} \{t.\ \cdots\}_{\mathtt{t}})$ is an abbreviation of

$$t \in \mathrm{Rep\_domT}\ (\mathrm{Abs\_domT}\ \{t.\ \cdots\}),$$

thus the transform from (∗2) to (∗1) requires that $\{t.\ \cdots\} \in \mathtt{domT}$, in other words, $\{t.\ \cdots\}$ is a non-empty and prefix-closed set.

In Csp-Prover, the required property ($\{t.\ \cdots\} \in \mathtt{domT}$) for each operator has already been proven in the theory-file `CSP_T_traces.thy`, and then the lemma `in_traces` is given in order to reduce the proof obligation. Similarly, you will prefer `in_failures` to `failures.simps`. In summary, the semantical proof will proceed as follows:

1. $P =_{\mathcal{F}} Q$ is rewritten to

   $\mathtt{(traces}(P) = \mathtt{traces}(Q)) \wedge \mathtt{(failures}(P) = \mathtt{failures}(Q))$

   by `(simp add: cspF_eqF_semantics)`. For $P \sqsubseteq_{\mathcal{F}} Q$, you will apply the lemma `(cspF_refF_semantics)` instead.

2. $\mathtt{(traces}(P) = \mathtt{traces}(Q))$ is rewritten to two subgoals

   $\mathtt{(traces}(P) \leq \mathtt{traces}(Q))$ and $\mathtt{(traces}(Q) \leq \mathtt{traces}(P))$

   by `(rule order_antisym)`.

3. $\mathtt{(traces}(P) \leq \mathtt{traces}(Q))$ is rewritten to

   $\bigwedge t.\ \mathtt{(}t \in_{\mathtt{t}} \mathtt{traces}(P) \longrightarrow t \in_{\mathtt{t}} \mathtt{traces}(Q))$

   by `(rule subdomTI)`.

4. `in_traces` is applied to each $t \in_{\mathtt{t}} \mathtt{traces}(P)$.

5. Similarly, the lemmas `(rule subsetFI)` and `in_failures` will be used for $\mathtt{failures}(P)$.

Now, take a look at the proof script of the lemma `semantical_proof` in `Test_proof.thy` in `Test`. The proof proceeds in accordance with the above

[9] This rule `traces.simps` is automatically added to the simplification rules when `traces` is defined by **primrec**. However, we do not recommend to use `traces.simps` as explained later soon. Therefore, the rule is removed from the simplification rules by the command **declare** `traces.simps [simp del]`.

instruction [10]. During the proof, the subgoals are sometimes a little complex. It will be found that Csp-Prover assists the proof well.

8.2 Manually syntactical proof

Csp-Prover also provides a lot of algebraic Csp laws which have already been proven by the semantical way mentioned in the previous subsection. Such algebraic Csp laws allow us to prove the process equivalence and the process refinement by syntactically rewriting process expressions.

The Csp laws implemented in Csp-Prover are given in Figures 10, $\cdots$, 16. The Csp laws and their names such as ($\Box$-step) are *almost* the same as the laws and the names given in [Ros98]. The differences from [Ros98] are denoted by the superscripts $*$ and $+$ attached to names. The superscript $*$ means modified laws, and the superscript $+$ means added laws. All the laws in Figures 10, $\cdots$, 16 have already been proven by the semantical way mentioned, thus they are proven to be sound.

Here, you might have a question about completeness, thus for every $P, Q \in$`'a proc` such that $P =_{\mathcal{F}} Q$, is it possible to syntactically prove the equality by the Csp laws without using the semantics (i.e. $[\![P]\!]_{\mathcal{F}} = [\![Q]\!]_{\mathcal{F}}$) ? The answer is *yes*. We will discuss the completeness in the other paper (not yet published), but you can find the full proof in the theory-files in the package `FNF_F`.

In Figures 10, $\cdots$, 16, the labels written in ASCII such as `"cspF_reflex"` given for each block are the names of lemmas in Csp-Prover. Some lemmas such as `"cspF_reflex"` contains more than two laws. When such lemma is applied to a subgoal, a law matching to the subgoal is selected and is applied.

Now, take a look at the proof script of the lemma `syntactical_proof` in `Test_proof.thy` in `Test`. The key laws to prove the following main goal are *step-laws*.

```
goal (lemma (syntactical_proof), 1 subgoal):
1.  ((a → P) ⟦{a}⟧ (a → Q)) =F a → (P ⟦{a}⟧ Q)
```

However, you cannot directly apply `(simp add: cspF_step)` because the equality is not $=$ but is $=_{\mathcal{F}}$. We explain how to rewrite the expression by the Csp laws step by step.

In general, it is firstly stated by either `cspF_rw_left` or `cspF_rw_right` [11] which side of $=_{\mathcal{F}}$ is rewritten. For example, by applying `(rule cspF_rw_left)`, the main goal is rewritten to

[10] The lemmas whose name has the form `par_tr_`$\cdots$ relate to $[\![X]\!]_{\mathtt{tr}}$ over traces, and they are given in `Trace_par.thy` in `CSP`.

[11] The lemma `cspF_rw_right` includes $[|\ P_3 =_{\mathcal{F}} P_2;\ P_1 =_{\mathcal{F}} P_2\ |] \Longrightarrow P_1 =_{\mathcal{F}} P_3$, and it can be derived from `cspF_trans` and `cspF_sym`. `cspF_rw_left` includes `cspF_trans`

"cspF_reflex"

$P =_{\mathcal{F}} P$ (reflexivity)

"cspF_sym"

$P =_{\mathcal{F}} Q \Longrightarrow Q =_{\mathcal{F}} P$ (symmetry)

"cspF_trans"

$[\!|\ P_1 =_{\mathcal{F}} P_2;\ P_2 =_{\mathcal{F}} P_3\ |\!] \Longrightarrow P_1 =_{\mathcal{F}} P_3$ (transitivity)

"cspF_decompo"

$[\!|\ a = b;\ P =_{\mathcal{F}} Q\ |\!] \Longrightarrow a \rightarrow P =_{\mathcal{F}} b \rightarrow Q$ (prefix-cong)

$[\!|\ X = Y;\ \bigwedge x.\ x \in X \Longrightarrow P(x) =_{\mathcal{F}} Q(x)\ |\!]$
$\Longrightarrow\ ?\ x : X \rightarrow P(x) =_{\mathcal{F}} ?\ x : Y \rightarrow Q(x)$ (?-cong)

$[\!|\ P_1 =_{\mathcal{F}} Q_1;\ P_2 =_{\mathcal{F}} Q_2\ |\!] \Longrightarrow P_1 \,\Box\, P_2 =_{\mathcal{F}} Q_1 \,\Box\, Q_2$ ($\Box$-cong)

$[\!|\ P_1 =_{\mathcal{F}} Q_1;\ P_2 =_{\mathcal{F}} Q_2\ |\!] \Longrightarrow P_1 \sqcap P_2 =_{\mathcal{F}} Q_1 \sqcap Q_2$ ($\sqcap$-cong)

$[\!|\ C_1 = C_2;\ \bigwedge c.\ c \in C_1 \Longrightarrow P(c) =_{\mathcal{F}} Q(c)\ |\!]$
$\Longrightarrow\ !!\ c : C_1 \rightarrow P(c) =_{\mathcal{F}} !!\ c : C_2 \rightarrow Q(c)$ (!!-cong)

$[\!|\ P_1 =_{\mathcal{F}} Q_1;\ P_2 =_{\mathcal{F}} Q_2\ |\!] \Longrightarrow P_1 \,[\![X]\!]\, P_2 =_{\mathcal{F}} Q_1 \,[\![X]\!]\, Q_2$ ($[\![X]\!]$-cong)

$[\!|\ X = Y;\ P =_{\mathcal{F}} Q\ |\!] \Longrightarrow P \setminus X =_{\mathcal{F}} Q \setminus Y$ (hide-cong)

$[\!|\ r_1 = r_2;\ P =_{\mathcal{F}} Q\ |\!] \Longrightarrow P[[r_1]] =_{\mathcal{F}} Q[[r_2]]$ (ren-cong)

$[\!|\ P_1 =_{\mathcal{F}} Q_1;\ P_2 =_{\mathcal{F}} Q_2\ |\!] \Longrightarrow P_1 \fatsemi P_2 =_{\mathcal{F}} Q_1 \fatsemi Q_2$ ($\fatsemi$-cong)

$[\!|\ n_1 = n_2;\ P =_{\mathcal{F}} Q\ |\!] \Longrightarrow P \lfloor n_1 =_{\mathcal{F}} Q \lfloor n_2$ ($\lfloor$-cong)$^+$

Figure 10: CSP congruence laws

"cspF_IF"

IF True THEN P ELSE Q $=_{\mathcal{F}}$ P	(if-true)
IF False THEN P ELSE Q $=_{\mathcal{F}}$ Q	(if-false)

"cspF_idem"

$P \,\Box\, P =_{\mathcal{F}} P$	($\Box$-idem)
$P \sqcap P =_{\mathcal{F}} P$	($\sqcap$-idem)

"cspF_commut"

$P \,\Box\, Q =_{\mathcal{F}} Q \,\Box\, P$	($\Box$-sym)
$P \sqcap Q =_{\mathcal{F}} Q \sqcap P$	($\sqcap$-sym)
$P \,[\![X]\!]\, Q =_{\mathcal{F}} Q \,[\![X]\!]\, P$	($[\![X]\!]$-sym)

"cspF_assoc"

$P \,\Box\, (Q \,\Box\, R) =_{\mathcal{F}} (P \,\Box\, Q) \,\Box\, R$	($\Box$-assoc)
$P \sqcap (Q \sqcap R) =_{\mathcal{F}} (P \sqcap Q) \sqcap R$	($\sqcap$-assoc)

"cspF_unit"

$P \,\Box\, \mathit{Stop} =_{\mathcal{F}} P$	($\Box$-unit)
$P \sqcap \mathit{Div} =_{\mathcal{F}} P$	($\sqcap$-unit)

"cspF_Rep_int_choice_empty"

$!!\ c : \emptyset \bullet P(c) =_{\mathcal{F}}$ DIV	(!!-emptyset)$^{+}$

"cspF_Rep_int_choice_const"

$[\!]\ C \neq \emptyset;\ \forall\, c \in C.\, P(c) = Q\ [\!] \Longrightarrow\ !!\, c : C \bullet P(c) =_{\mathcal{F}} Q$	(!!-const)*

"cspF_Rep_int_choice_union_Int"

$!!\, c : (C_1 \cup C_2) \bullet P(c) =_{\mathcal{F}} (!!\, c : C_1 \bullet P(c)) \sqcap (!!\, c : C_2 \bullet P(c))$	(!!-union-$\sqcap$)*

Figure 11: CSP basic laws

"cspF_Dist"

$C \neq \emptyset \Longrightarrow (!!\, c : C \bullet P(c)) \,\Box\, Q =_{\mathcal{F}} !!\, c : C \bullet (P(c) \,\Box\, Q)$	($\Box$-Dist)
$C \neq \emptyset \Longrightarrow (!!\, c : C \bullet P(c)) \,[\![X]\!]\, Q =_{\mathcal{F}} !!\, c : C \bullet (P(c) \,[\![X]\!]\, Q)$	($[\![X]\!]$-Dist)
$(!!\, c : C \bullet P(c)) \setminus X =_{\mathcal{F}} !!\, c : C \bullet (P(c) \setminus X)$	(hide-Dist)
$(!!\, c : C \bullet P(c))[[r]] =_{\mathcal{F}} !!\, c : C \bullet (P(c)[[r]])$	($[[r]]$-Dist)
$(!!\, c : C \bullet P(c)) \fatsemi Q =_{\mathcal{F}} !!\, c : C \bullet (P(c) \fatsemi Q)$	($\fatsemi$-Dist)
$(!!\, c : C \bullet P(c)) \lfloor n =_{\mathcal{F}} !!\, c : C \bullet (P(c) \lfloor n)$	($\lfloor$-Dist)$^+$

"cspF_dist"

$(P_1 \sqcap P_2) \,\Box\, Q =_{\mathcal{F}} (P_1 \,\Box\, Q) \sqcap (P_2 \,\Box\, Q)$	($\Box$-dist)
$(P_1 \sqcap P_2) \,[\![X]\!]\, Q =_{\mathcal{F}} (P_1 \,[\![X]\!]\, Q) \sqcap (P_2 \,[\![X]\!]\, Q)$	($[\![X]\!]$-dist)
$(P_1 \sqcap P_2) \setminus X =_{\mathcal{F}} (P_1 \setminus X) \sqcap (P_2 \setminus X)$	(hide-dist)
$(P_1 \sqcap P_2)[[r]] =_{\mathcal{F}} (P_1[[r]]) \sqcap (P_2[[r]])$	($[[r]]$-dist)
$(P_1 \sqcap P_2) \fatsemi Q =_{\mathcal{F}} (P_1 \fatsemi Q) \sqcap (P_2 \fatsemi Q)$	($\fatsemi$-dist)
$(P_1 \sqcap P_2) \lfloor n =_{\mathcal{F}} (P_1 \lfloor n) \sqcap (P_2 \lfloor n)$	($\lfloor$-dist)$^+$
$!!c : C \bullet (P_1(c) \sqcap P_2(c)) =_{\mathcal{F}} (!!c : C \bullet P_1(c)) \sqcap (!!c : C \bullet P_2(c))$	(!!-dist)

"cspF_Ext_dist"

$(P_1 \,\Box\, P_2)[[r]] =_{\mathcal{F}} (P_1[[r]]) \,\Box\, (P_2[[r]])$	($[[r]]$-$\Box$-dist)
$(P_1 \,\Box\, P_2) \lfloor n =_{\mathcal{F}} (P_1 \lfloor n) \,\Box\, (P_2 \lfloor n)$	($\lfloor$-$\Box$-dist)$^+$

Figure 12: Csp distributive laws

"cspF_step"

$Stop =_{\mathcal{F}} \ ?\, x : \emptyset \to P(x)$ (stop-step)

$a \to P \ =_{\mathcal{F}} ?\, x : \{a\} \to P$ (prefix-step)

$$
\begin{array}{l}
(?\, x : A \to P(x)) \,\Box\, (?\, x : B \to Q(x)) \\
=_{\mathcal{F}} ?\, x : (A \cup B) \to (\mathtt{IF}\,(x \in A \cap B)\,\mathtt{THEN}\ P(x) \sqcap Q(x) \\
\qquad\qquad\qquad \mathtt{ELSE\ IF}\,(x \in A)\,\mathtt{THEN}\ P(x)\,\mathtt{ELSE}\ Q(x))
\end{array}
$$ ($\Box$-step)

$$
\begin{array}{l}
(?x : A \to P'(x)) \,[\![\, X \,]\!]\, (?x : B \to Q'(x)) \\
=_{\mathcal{F}} (?x : ((X \cap A \cap B) \cup (A - X) \cup (B - X)) \to \\
\quad \mathtt{IF}\,(x \in X) \\
\quad \mathtt{THEN}\ (P'(x) \,[\![\, X \,]\!]\, Q'(x)) \\
\quad \mathtt{ELSE\ IF}\,(x \in A \cap B) \\
\qquad \mathtt{THEN}\ ((P'(x) \,[\![\, X \,]\!]\, (?x : B \to Q'(x))) \sqcap \\
\qquad\quad ((?x : A \to P'(x)) \,[\![\, X \,]\!]\, Q'(x))) \\
\qquad \mathtt{ELSE\ IF}\,(x \in A) \\
\qquad\quad \mathtt{THEN}\ (P'(x) \,[\![\, X \,]\!]\, (?x : B \to Q'(x))) \\
\qquad\quad \mathtt{ELSE}\ ((?x : A \to P'(x)) \,[\![\, X \,]\!]\, Q'(x)))
\end{array}
$$ ($[\![X]\!]$-step)

$$
\begin{array}{l}
(?\, x : A \to P(x)) \setminus X \\
=_{\mathcal{F}} \mathtt{IF}\,(A \cap X =_{\mathcal{F}} \emptyset) \\
\quad \mathtt{THEN}\ ?\, x : A \to (P(x) \setminus X) \\
\quad \mathtt{ELSE}\,(?\, x : (A - X) \to (P(x) \setminus X)) \\
\qquad \triangleright\, (!\ x : (A \cap X) \bullet (P(x) \setminus X))
\end{array}
$$ (hide-step)

$$
\begin{array}{l}
(?\, x : A \to P(x))[[r]] \\
=_{\mathcal{F}} ?\, x : \{x \mid \exists\, a \in A.\, (a, x) \in r\} \to \\
\quad (!\ a : \{a \in A \mid (a, x) \in r\} \bullet (P(a)[[r]]))
\end{array}
$$ ([[r]]-step)

$(?\, x : A \to P(x)) \fatsemi Q \ =_{\mathcal{F}} ?\, x : A \to (P(x) \fatsemi Q)$ ($\fatsemi$-step)

$(?\, x : A \to P(x)) \lfloor (n+1) =_{\mathcal{F}} ?\, x : A \to (P(x) \lfloor n)$ ($\lfloor$-step)$^+$

Figure 13: CSP step laws

"cspF_step_ext"

$$
\begin{array}{l}
((?x : A \to P'(x)) \triangleright P'') \,[\![X]\!]\, ((?x : B \to Q'(x)) \triangleright Q'') \\
=_{\mathcal{F}} (?x : ((X \cap A \cap B) \cup (A - X) \cup (B - X)) \to \\
\quad \mathtt{IF}\,(x \in X) \\
\quad \mathtt{THEN}\,(P'(x) \,[\![X]\!]\, Q'(x)) \\
\quad \mathtt{ELSE\ IF}\,(x \in A \cap B) \\
\qquad \mathtt{THEN}\,((P'(x) \,[\![X]\!]\, ((?x : B \to Q'(x)) \triangleright Q'')) \sqcap \\
\qquad\quad (((?x : A \to P'(x)) \triangleright P'') \,[\![X]\!]\, Q'(x))) \\
\qquad \mathtt{ELSE\ IF}\,(x \in A) \\
\qquad\quad \mathtt{THEN}\,(P'(x) \,[\![X]\!]\, ((?x : B \to Q'(x)) \triangleright Q'')) \\
\qquad\quad \mathtt{ELSE}\,(((?x : A \to P'(x)) \triangleright P'') \,[\![X]\!]\, Q'(x))) \\
\quad \triangleright ((P'' \,[\![X]\!]\, ((?x : B \to Q'(x)) \triangleright Q'')) \sqcap \\
\quad\ (((?x : A \to P'(x)) \triangleright P'') \,[\![X]\!]\, Q'')) \qquad ([\![X]\!]\text{-}\triangleright\text{-split})^*
\end{array}
$$

$$
\begin{array}{l}
((?x : A \to P'(x)) \triangleright P'') \,[\![X]\!]\, (?x : B \to Q'(x)) \\
=_{\mathcal{F}} (?x : ((X \cap A \cap B) \cup (A - X) \cup (B - X)) \to \\
\quad \mathtt{IF}\,(x \in X) \\
\quad \mathtt{THEN}\,(P'(x) \,[\![X]\!]\, Q'(x)) \\
\quad \mathtt{ELSE\ IF}\,(x \in A \cap B) \\
\qquad \mathtt{THEN}\,((P'(x) \,[\![X]\!]\, (?x : B \to Q'(x))) \sqcap \\
\qquad\quad (((?x : A \to P'(x)) \triangleright P'') \,[\![X]\!]\, Q'(x))) \\
\qquad \mathtt{ELSE\ IF}\,(x \in A) \\
\qquad\quad \mathtt{THEN}\,(P'(x) \,[\![X]\!]\, (?x : B \to Q'(x))) \\
\qquad\quad \mathtt{ELSE}\,(((?x : A \to P'(x)) \triangleright P'') \,[\![X]\!]\, Q'(x))) \\
\quad \triangleright (P'' \,[\![X]\!]\, (?x : B \to Q'(x))) \qquad ([\![X]\!]\text{-}\triangleright\text{-input})^*
\end{array}
$$

"cspF_Ext_choice_SKIP_DIV_resolve"

$$P \,\Box\, \mathtt{SKIP} = P \triangleright \mathtt{SKIP} \qquad (\Box\text{-skip-resolve})$$

$$P \,\Box\, \mathtt{DIV} = P \triangleright \mathtt{DIV} \qquad (\Box\text{-div-resolve})$$

"cspF_Depth_rest_Zero"

$$P \lfloor 0 =_{\mathcal{F}} \mathtt{DIV} \qquad (\lfloor\text{-zero})^+$$

"cspF_Depth_rest_min"

$$(P \lfloor n) \lfloor m =_{\mathcal{F}} P \lfloor \min(n, m) \qquad (\lfloor\text{-min})^+$$

Figure 14: CSP extended step laws and depth-restriction laws

"cspF_SKIP_DIV"

$\texttt{SKIP} \,\Box\, \texttt{DIV} =_{\mathcal{F}} \texttt{SKIP}$	(skip-div-$\Box$)
$\texttt{SKIP} \,[\![X]\!]\, \texttt{SKIP} =_{\mathcal{F}} \texttt{SKIP}$	(skip-$[\![X]\!]$)
$\texttt{DIV} \,[\![X]\!]\, \texttt{DIV} =_{\mathcal{F}} \texttt{DIV}$	(div-$[\![X]\!]$)
$\texttt{SKIP} \,[\![X]\!]\, \texttt{DIV} =_{\mathcal{F}} \texttt{DIV}$	(skip-div-$[\![X]\!]$)
$\texttt{SKIP} \,[\![X]\!]\, (?\, x : A \to P(x))$ $=_{\mathcal{F}} ?\, x : (A - X) \to (\texttt{SKIP} \,[\![X]\!]\, P(x))$	($[\![X]\!]$-preterm)
$\texttt{DIV} \,[\![X]\!]\, (?\, x : A \to P(x))$ $=_{\mathcal{F}} \; (?\, x : (A - X) \to (\texttt{DIV} \,[\![X]\!]\, P(x))) \,\Box\, \texttt{DIV}$	(div-$[\![X]\!]$-step)
$\texttt{SKIP} \,[\![X]\!]\, ((?\, x : A \to P(x)) \,\Box\, \texttt{SKIP})$ $=_{\mathcal{F}} \; (?\, x : (A - X) \to (\texttt{SKIP} \,[\![X]\!]\, P(x))) \,\Box\, \texttt{SKIP}$	(skip-$[\![X]\!]$-$\Box$-skip)
$\texttt{SKIP} \,[\![X]\!]\, ((?\, x : A \to P(x)) \,\Box\, \texttt{DIV})$ $=_{\mathcal{F}} \; (?\, x : (A - X) \to (\texttt{SKIP} \,[\![X]\!]\, P(x))) \,\Box\, \texttt{DIV}$	(skip-$[\![X]\!]$-$\Box$-div)
$\texttt{DIV} \,[\![X]\!]\, (P \,\Box\, \texttt{SKIP}) =_{\mathcal{F}} \texttt{DIV} \,[\![X]\!]\, P$	(div-$[\![X]\!]$-$\Box$-skip)
$\texttt{DIV} \,[\![X]\!]\, (P \,\Box\, \texttt{DIV}) =_{\mathcal{F}} \texttt{DIV} \,[\![X]\!]\, P$	(div-$[\![X]\!]$-$\Box$-div)
$\texttt{SKIP} \setminus X =_{\mathcal{F}} \texttt{SKIP}$	(skip-hide)
$\texttt{DIV} \setminus X =_{\mathcal{F}} \texttt{DIV}$	(div-hide)
$((?\; x : A \to P(x)) \,\Box\, \texttt{SKIP}) \setminus X$ $=_{\mathcal{F}} ((?\; x : (A - X) \to (P(x) \setminus X)) \,\Box\, \texttt{SKIP})$ $\quad \sqcap\, (!\; x : (A \cap X) \bullet (P(x) \setminus X))$	(skip-hide-step)
$((?\; x : A \to P(x)) \,\Box\, \texttt{DIV}) \setminus X$ $=_{\mathcal{F}} ((?\; x : (A - X) \to (P(x) \setminus X)) \,\Box\, \texttt{DIV})$ $\quad \sqcap\, (!\; x : (A \cap X) \bullet (P(x) \setminus X))$	(div-hide-step)
$\texttt{SKIP}[[r]] =_{\mathcal{F}} \texttt{SKIP}$	(skip-$[[r]]$-id)
$\texttt{DIV}[[r]] =_{\mathcal{F}} \texttt{DIV}$	(div-$[[r]]$-id)
$\texttt{SKIP} \fatsemi P =_{\mathcal{F}} P$	($\fatsemi$-unit-l)
$\texttt{DIV} \fatsemi P =_{\mathcal{F}} \texttt{DIV}$	(div-$\fatsemi$)
$((?\, x : A \to P(x)) \triangleright \texttt{SKIP}) \fatsemi R$ $=_{\mathcal{F}} (?\, x : A \to (P(x) \fatsemi R)) \triangleright R$	(skip-$\fatsemi$-step)
$((?\, x : A \to P(x)) \triangleright \texttt{DIV}) \fatsemi R$ $=_{\mathcal{F}} (?\, x : A \to (P(x) \fatsemi R)) \triangleright \texttt{DIV}$	(div-$\fatsemi$-step)
$\texttt{SKIP} \lfloor (n+1) =_{\mathcal{F}} \texttt{SKIP}$	(skip-$\lfloor\,)^+$
$\texttt{DIV} \lfloor n =_{\mathcal{F}} \texttt{DIV}$	(div-$\lfloor\,)^+$

Figure 15: CSP skip and div laws

"cspF_Rep_int_choice_input_set"

$$!!\, c : C \bullet (?\, x : A(c) \to P(c, x))$$
$$=_{\mathcal{F}} !\mathtt{set}\ X : \{A(c) \mid c \in C\} \bullet$$
$$(?\, x : X \to (!!\, c : \{c \in C \mid x \in A(c)\} \bullet P(c, x))) \qquad (!!\text{-input-}!\text{set})^+$$

"cspF_Rep_int_choice_Ext_Dist"

$$\forall\, c \in C.\, Q(c) \in \{\mathtt{SKIP}, \mathtt{DIV}\} \Longrightarrow$$
$$!!c : C \bullet (P(c) \,\Box\, Q(c))$$
$$=_{\mathcal{F}} (!!c : C \bullet P(c)) \,\Box\, (!!c : C \bullet Q(c)) \qquad (!!\text{-}\Box\text{-Dist})^+$$

"cspF_Rep_int_choice_input_Dist"

$$[|\ Q = \mathtt{SKIP} \vee Q = \mathtt{DIV}\ |] \Longrightarrow$$
$$(!\mathtt{set}\ X : \mathcal{X} \bullet (?x : X \to P(x))) \,\Box\, Q$$
$$=_{\mathcal{F}} (?x : \bigcup \mathcal{X} \to P(x)) \,\Box\, Q \qquad (!!\text{-input-Dist})^+$$

"cspF_input_DIV"

$$?\ x : A \to P(x)$$
$$=_{\mathcal{F}} ((?\, x : A \to P(x)) \,\Box\, \mathtt{DIV}) \sqcap (?\, x : A \to \mathtt{DIV}) \qquad (?\text{-div})^+$$

"cspF_Rep_int_choice_set_DIV"

$$!!\, c : C \bullet (!\mathtt{set}\ X : \mathcal{X}(c) \bullet (?\, x : X \to \mathtt{DIV}))$$
$$=_{\mathcal{F}} !\mathtt{set}\ X : \bigcup\{\mathcal{X}(c) \mid c \in C\} \bullet (?\, x : X \to \mathtt{DIV}) \qquad (!!\text{-}!\text{set-div})^+$$

"cspF_input_Rep_int_choice_set_subset"

if $\mathcal{X} \subseteq \mathcal{Y}$ and $(\forall\, Y \in \mathcal{Y}.\ \exists\, X \in \mathcal{X}.\, X \subseteq Y \subseteq A)$ then
$$((?\, x : A \to P(x)) \,\Box\, R) \sqcap (!\mathtt{set}\ X : \mathcal{X} \bullet (?\, x : X \to \mathtt{DIV}))$$
$$=_{\mathcal{F}} ((?\, x : A \to P(x)) \,\Box\, R) \sqcap (!\mathtt{set}\ X : \mathcal{Y} \bullet (?\, x : X \to \mathtt{DIV})) \quad (?\text{-}!\text{set-}\subseteq)^+$$

"cspF_nat_Depth_rest"

$$P =_{\mathcal{F}} !\mathtt{nat}\ n \bullet (P \lfloor n) \qquad (!\text{nat-}\lfloor)^+$$

Figure 16: Csp replicated internal choice laws and normalising laws

```
goal (lemma (syntactical_proof), 2 subgoals):
1. ((a → P) ⟦ {a} ⟧ (a → Q)) =_F ?P2.0
2. ?P2.0 =_F a → (P ⟦ {a} ⟧ Q)
```

where $?P2.0$ is a variable called *schematic variable* or *unknown* automatically generated by Isabelle. Such variable will be instantiated later.

Next, it may be expected to apply the ($[\![X]\!]$-step) law, but it is not available yet. Before applying ($[\![X]\!]$-step), $(a \rightarrow P)$ has to transformed to the form of $?\,a : Y \rightarrow P'(a)$. To do that, decompose the parallel operator in the first goal by `(rule cspF_decompo)`. It generates the following subgoals:

```
goal (lemma (syntactical_proof), 4 subgoals):
1. {a} = ?Y1
2. a → P =_F ?Q1.1
3. a → Q =_F ?Q2.1
4. ?Q1.1⟦?Y1⟧ ?Q2.1 =_F  a → (P ⟦ {a} ⟧ Q)
```

The first goal is trivial. By applying `(simp)`, the schematic variable $?Y1$ is instantiated to $\{a\}$ and the first goal disappears:

```
goal (lemma (syntactical_proof), 3 subgoals):
1. a → P =_F ?Q1.1
2. a → Q =_F ?Q2.1
3. ?Q1.1 ⟦{a}⟧ ?Q2.1 =_F  a → (P ⟦ {a} ⟧ Q)
```

Here, apply `(rule cspF_step)` to the first goal, then $?Q1.1$ is instantiated to $?\,x : \{a\} \rightarrow P$ as follows:

```
goal (lemma (syntactical_proof), 2 subgoals):
1. a → Q =_F ?Q2.1
2. (?x : {a} → P) ⟦ {a}⟧ ?Q2.1 =_F  a → (P ⟦ {a} ⟧ Q)
```

Similarly, by `(rule cspF_step)` again, you will get the following subgoal:

```
goal (lemma (syntactical_proof), 1 subgoal):
1. (?x : {a} → P) ⟦ {a} ⟧  (?x : {a} → Q) =_F  a → (P ⟦ {a} ⟧ Q)
```

Then, you can apply the ($[\![X]\!]$-step) law on the left side by `(rule cspF_rw_left)` and `(rule cspF_step)`. And continue to apply the commands until `done` in the proof script of the lemma `syntactical_proof` in `Test_proof.thy`. You will see the outline of the syntactical proof.

Hitherto, we have given the instruction for syntactical proof of $=_{\mathcal{F}}$. The process refinement $\sqsubseteq_{\mathcal{F}}$ is also proven by a similar way. The lemmas `(rule cspF_rw_left)`, `(rule cspF_rw_right)`, and `(rule cspF_decompo)` can be also applied to $\sqsubseteq_{\mathcal{F}}$ [12]. The additional laws for $\sqsubseteq_{\mathcal{F}}$ are shown Figure 17.

In summary, the syntactical proof will proceed as follows:

1. It is selected by either `(rule cspF_rw_left)` or `(rule cspF_rw_right)` which side of $P =_{\mathcal{F}} Q$ (or $P \sqsubseteq_{\mathcal{F}} Q$) is rewritten.

[12] For example, `(rule cspF_rw_right)` includes $[\![\ P_3 =_{\mathcal{F}} P_2;\ P_1 \sqsubseteq_{\mathcal{F}} P_2\ ]\!] \Longrightarrow P_1 \sqsubseteq_{\mathcal{F}} P_3$, and `(rule cspF_decompo)` includes $[\![\ a = b;\ P \sqsubseteq_{\mathcal{F}} Q\ ]\!] \Longrightarrow a \rightarrow P \sqsubseteq_{\mathcal{F}} b \rightarrow Q$.

"cspF_ref_eq"

$[|\ P \sqsubseteq_{\mathcal{F}} Q;\ Q \sqsubseteq_{\mathcal{F}} P\ |] \Longrightarrow P =_{\mathcal{F}} Q$ ($\sqsubseteq_{\mathcal{F}}$-$=_{\mathcal{F}}$)

"cspF_eq_ref"

$P =_{\mathcal{F}} Q \Longrightarrow P \sqsubseteq_{\mathcal{F}} Q$ ($=_{\mathcal{F}}$-$\sqsubseteq_{\mathcal{F}}$)

"cspF_Int_choice_left1"

$P_1 \sqsubseteq_{\mathcal{F}} Q \Longrightarrow P_1 \sqcap P_2 \sqsubseteq_{\mathcal{F}} Q$ ($\sqcap$-left-1)

"cspF_Int_choice_left2"

$P_2 \sqsubseteq_{\mathcal{F}} Q \Longrightarrow P_1 \sqcap P_2 \sqsubseteq_{\mathcal{F}} Q$ ($\sqcap$-left-2)

"cspF_Int_choice_right"

$[|\ P \sqsubseteq_{\mathcal{F}} Q_1;\ P \sqsubseteq_{\mathcal{F}} Q_1\ |] \Longrightarrow P \sqsubseteq_{\mathcal{F}} Q_1 \sqcap Q_2$ ($\sqcap$-right)

"cspF_Rep_int_choice_left"

$(\exists\, c.\ c \in C \wedge P(c) \sqsubseteq_{\mathcal{F}} Q) \Longrightarrow !!\ c : C \bullet P(c) \sqsubseteq_{\mathcal{F}} Q$ (!!-left)

"cspF_Rep_int_choice_right"

$(\bigwedge c.\ c \in C \Longrightarrow P \sqsubseteq_{\mathcal{F}} Q(c)) \Longrightarrow P \sqsubseteq_{\mathcal{F}} !!\ c : C \bullet Q(c)$ (!!-right)

"cspF_decompo_subset"

$[|\ C_2 \subseteq C_1;\ \bigwedge c.\ c \in C_2 \Longrightarrow P(c) \sqsubseteq_{\mathcal{F}} Q(c)\ |]$
$\Longrightarrow !!\ c : C_1 \bullet P(c)\ \sqsubseteq_{\mathcal{F}}\ !!\ c : C_2 \bullet Q(c)$ (!!-subset)

$[|\ Y \neq \{\};\ Y \subseteq X;\ \bigwedge x.\ x \in Y \Longrightarrow P(x) \sqsubseteq_{\mathcal{F}} Q(x)\ |]$
$\Longrightarrow !\ x : X \bullet (x \rightarrow P(x))\ \sqsubseteq_{\mathcal{F}}\ ?\ x : Y \rightarrow Q(x)$ (!!-?-subset)

"cspF_Ext_choice_right"

$[|\ P \sqsubseteq_{\mathcal{F}} Q_1;\ P \sqsubseteq_{\mathcal{F}} Q_2\ |] \Longrightarrow P \sqsubseteq_{\mathcal{F}} Q_1 \,\Box\, Q_2$ ($\Box$-right)

Figure 17: Csp refinement laws

2. Decompose the expression by `(rule cspF_decompo)` until the subexpression to be rewritten appears alone.
3. Apply the Csp rule by `(rule cspF_...)`.

You will find a lot of syntactical proof technique in the theory-files (e.g. lemma `cspF_fsfF_Ext_choice_eqF` in `FNF_F_sf_ext.thy`) in the package `FNF_F`. For example, if you want to apply the law `cspF_assoc` to a subgoal in the opposite direction (i.e. $(P \ \Box \ Q) \ \Box \ R =_{\mathcal{F}} P \ \Box \ (Q \ \Box \ R)$), you can apply the command `(rule cspF_assoc[THEN cspF_sym])`. In this case, at first `cspF_sym` is applied to `cspF_assoc`, then the result is applied to the subgoal.

In the rest of this subsection, we give Csp laws for recursive processes, see Figures 18 and 19. The laws (Tarski-fix) and (Banach-fix) can replace fixed point by unbounded internal choice. They may be useful for theoretical work. On the other hand, the unwinding laws and the fixed-point induction laws will be often used in practical verifications. The unwinding laws are intuitively understandable, but the fixed-point induction laws might not be intuitive. We pick up the law (induct-cpo) and explain it by using the example *Buffer* in Figure 9.

As a simple example, we verify *Buffer* is deadlock-free, thus `DF` $\sqsubseteq_{\mathcal{F}}$ `Buffer`, where the deadlock-free specification `DF` is defined as: $\mu \ X \bullet (!\, x \rightarrow X)$. See the lemma `manual_proof_Buffer` in the theory file `Test_Buffer.thy` in the package `Test`. After setting `DF` $\sqsubseteq_{\mathcal{F}}$ `Buffer` as the main goal and unfolding the definition of `Buffer`, the following goal is displayed:

```
goal (lemma (manual_proof_Buffer), 1 subgoal):
1. DF ⊑F (FIX fun) (Empty(0))
```

In general, the fixed point induction is applied at first. So, if you apply the fixed point induction to the goal by `(rule cspF_fp_induct_right)`, you will have the following three subgoals ($*$):

```
goal (lemma (manual_proof_Buffer), 3 subgoals):
1. fun ∈ ProcFun
2. DF ⊑F ?f(Empty(0))
3. ⋀p. ?f(p) ⊑F fun(?f)(p)
```

Here, note that the fixed-point `(FIX fun)` on `fun` has been replaced by the inductive property `?f(`p`)` $\sqsubseteq_{\mathcal{F}}$ `fun(?f)(`p`)`. However, it is actually difficult to instantiate the schematic variable $?f$ later. The variable $?f$ is a function which intuitively relates the right process (i.e. `Buffer`) to the left process (i.e. `DF`). More exactly, it relates each process-index in `BufferName` to a reachable process from `DF`. It would be better that such function $?f$ is given by users because it is hard to automatically find such functions. However, Csp-Prover can assist to find such functions. In this example, such function can be given as follows:

"cspF_FIX_eq"

$(\texttt{FIX}\ fun)(p) =_{\mathcal{F}} (!\texttt{nat}\ n \bullet ((fun^{(n)}(\lambda\, p'.\ \texttt{DIV}))(p)))$ (Tarski-fix)

$(\texttt{FIX!}\ fun)(p) =_{\mathcal{F}} (!\texttt{nat}\ n \bullet (((fun^{(n)}(\lambda\, p'.\ Any))(p))\, \lfloor\, n))$ (Banach-fix)

"cspF_FIX_FIX1"

$fun \in \texttt{gProcFun} \Longrightarrow (\texttt{FIX}\ fun)(p) =_{\mathcal{F}}\ (\texttt{FIX!}\ fun)(p)$ (FIX-FIX!)

"cspF_unwind"

$fun \in \texttt{ProcFun} \Longrightarrow (\texttt{FIX}\ fun)(p) =_{\mathcal{F}}\ (fun(\texttt{FIX}\ fun))(p)$ (unwind-cpo)

$fun \in \texttt{gProcFun} \Longrightarrow (\texttt{FIX!}\ fun)(p) =_{\mathcal{F}}\ (fun(\texttt{FIX}\ fun))(p)$ (unwind-cms)

"cspF_fp_induct_right"

$[|\ fun \in \texttt{ProcFun};\ Q \sqsubseteq_{\mathcal{F}} f(p);$
$\ \bigwedge p.\ f(p) \sqsubseteq_{\mathcal{F}} fun(f)(p)\ |] \Longrightarrow Q \sqsubseteq_{\mathcal{F}} (\texttt{FIX}\ fun)(p)$ (induct-cpo)

$[|\ fun \in \texttt{gProcFun};\ Q \sqsubseteq_{\mathcal{F}} f(p);$
$\ \bigwedge p.\ f(p) \sqsubseteq_{\mathcal{F}} fun(f)(p)\ |] \Longrightarrow Q \sqsubseteq_{\mathcal{F}} (\texttt{FIX!}\ fun)(p)$ (induct-cms-ref)

$[|\ fun \in \texttt{gProcFun};\ Q =_{\mathcal{F}} f(p);$
$\ \bigwedge p.\ f(p) =_{\mathcal{F}} fun(f)(p)\ |] \Longrightarrow Q =_{\mathcal{F}} (\texttt{FIX!}\ fun)(p)$ (induct-cms-eq)

"cspF_fp_induct_left"

$[|\ fun \in \texttt{gProcFun};\ f(p) \sqsubseteq_{\mathcal{F}} Q;$
$\ \bigwedge p.\ fun(f)(p) \sqsubseteq_{\mathcal{F}} f(p)\ |] \Longrightarrow (\texttt{FIX!}\ fun)(p) \sqsubseteq_{\mathcal{F}} Q$ (induct-cms-ref)

$[|\ fun \in \texttt{gProcFun};\ f(p) =_{\mathcal{F}} Q;$
$\ \bigwedge p.\ fun(f)(p) =_{\mathcal{F}} f(p)\ |] \Longrightarrow (\texttt{FIX!}\ fun)(p) =_{\mathcal{F}} Q$ (induct-cms-eq)

Figure 18: Csp fixed-point laws

"cspF_MU_eq"

$R \in \texttt{ProcX}$
$\quad \Longrightarrow (\mu\ X \bullet R(X)) =_{\mathcal{F}} (!\texttt{nat}\ n \bullet R^{(n)}(\texttt{DIV}))$ (μ-eq)

$R \in \texttt{gProcX}$
$\quad \Longrightarrow (\mu!\ X \bullet R(X)) =_{\mathcal{F}} (!\texttt{nat}\ n \bullet ((R^{(n)}(\texttt{DIV}))\ \lfloor\ n))$ ($\mu!$-eq)

"cspF_MU_MU1"

$R \in \texttt{gProcX} \Longrightarrow (\mu\ X \bullet R(X)) =_{\mathcal{F}} (\mu!\ X \bullet R(X))$ (μ-$\mu!$)

"cspF_unwind_MU"

$R \in \texttt{ProcX} \Longrightarrow (\mu\ X \bullet R(X)) =_{\mathcal{F}} R(\mu\ X \bullet R(X))$ (unwind-μ)

$R \in \texttt{gProcX} \Longrightarrow (\mu!\ X \bullet R(X)) =_{\mathcal{F}} R(\mu!\ X \bullet R(X))$ (unwind-$\mu!$)

"cspF_fp_induct_MU_right"

$[|\ R \in \texttt{ProcX}\ ;\ Q \sqsubseteq_{\mathcal{F}} R(Q)\ |] \Longrightarrow Q \sqsubseteq_{\mathcal{F}} (\mu\ X \bullet R(X))$ (induct-μ)

$[|\ R \in \texttt{gProcX}\ ;\ Q \sqsubseteq_{\mathcal{F}} R(Q)\ |] \Longrightarrow Q \sqsubseteq_{\mathcal{F}} (\mu!\ X \bullet R(X))$ (induct-$\mu!$-ref)

$[|\ R \in \texttt{gProcX}\ ;\ Q =_{\mathcal{F}} R(Q)\ |] \Longrightarrow Q =_{\mathcal{F}} (\mu!\ X \bullet R(X))$ (induct-$\mu!$-eq)

"cspF_fp_induct_MU_left"

$[|\ R \in \texttt{gProcX}\ ;\ R(Q) \sqsubseteq_{\mathcal{F}} Q\ |] \Longrightarrow (\mu\ X \bullet R(X)) \sqsubseteq_{\mathcal{F}} Q$ (induct-$\mu!$-ref)

$[|\ R \in \texttt{gProcX}\ ;\ R(Q) =_{\mathcal{F}} Q\ |] \Longrightarrow (\mu\ X \bullet R(X)) =_{\mathcal{F}} Q$ (induct-$\mu!$-eq)

Figure 19: Csp fixed-point laws (single)

```
consts
  Buffer_to_DF :: "BufferName ⇒ Event proc"
primrec
  "Buffer_to_DF (Empty n)  = DF"
  "Buffer_to_DF (Full r n) = DF"
```

Then, it is possible to apply the fixed point induction whose $?f$ has been instantiated to `Buffer_to_DF` by

```
(rule cspF_fp_induct_right[of _ _ "Buffer_to_DF"])
```

The application generates the following subgoals instead of $(*)$:

```
goal (lemma (manual_proof_Buffer), 3 subgoals):
1. fun ∈ ProcFun
2. DF ⊑_F Buffer_to_DF(Empty(0))
3. ⋀p. Buffer_to_DF(p) ⊑_F fun(Buffer_to_DF)(p)
```

The subgoals 1 and 2 can be easily proven by `(simp_all add: DF_def)`. The variable p in the subgoal 3 can be instantiated to (`Empty` *nat*) and (`Full` *real nat*) by `(induct_tac p)` as explained at the end of Subsection 7.4. And thereafter, by applying `(simp_all add: DF_def)`, the following two subgoals are obtained:

```
goal (lemma (manual_proof_Buffer), 2 subgoals):
1. μ X • (! x • x → X) ⊑_F ? x : (range In) → (μ X • (! x • x → X))
2. ⋀real nat. μ X • (! x • x → X) ⊑_F Out(real, nat) → (μ X • (! x • x → X))
```

These two subgoals are easily proven by unwinding μ (i.e. `cspF_unwind_MU`) and decomposition (i.e. `cspF_decompo_subset`). See the proof script in the lemma `manual_proof_Buffer`.

8.3 Semi-automatically syntactical proof

In Subsection 8.2, we explained the syntactical proof. In this proof, you can completely control which subexpression is rewritten. It may be sometimes convenient for theoretical works, but may be redundant for practical verification. In this subsection, we give semi-automatic *tactics* to apply Csp laws as much as possible.

The most useful tactics are `cspF_hsf_left_tac` and `cspF_hsf_right_tac` which sequentialise processes in left-hand sides and in right-hand sides, respectively. The tactic `cspF_hsf_left_tac` repeatedly applies the following Csp laws to processes in left-hand sides with the following priority (i.e. the law `cspF_choice_IF` has the highest priority).

1. The law `cspF_choice_IF`, which consists of `(cspF_IF)`, `(cspF_idem)`, `(cspF_unit)`, etc to simplify processes.
2. The law `cspF_SKIP_DIV_sort`, which is derived from `(cspF_commut)` and `(cspF_assoc)` to sort processes over $\Box$ to the form $?\,x : X \to P(x) \Box$ `SKIP`

or $?\,x : X \to P(x) \,\Box$ `DIV` if unguarded `SKIP` or `DIV` exists.

3. The law `cspF_SKIP_DIV_resolve`, which is derived from `(cspF_SKIP_DIV)`, `(cspF_Ext_choice_SKIP_DIV_resolve)`, and `(cspF_step_ext)` to sequentialise processes together with `SKIP` or `DIV`.
4. The law `cspF_step`, to sequentialise processes.
5. The law `cspF_all_dist`, which consists of `cspF_dist`, `cspF_Dist`, and `cspF_Ext_dist` to distribute operators on choice operators.
6. The laws `cspF_unwind` and `cspF_unwind_MU` to unwind recursive processes.
7. The law `cspF_free_decompo` to decompose operators, which are not guarded by prefix or prefix choice or replicated internal choice, to avoid infinite unwinding.
8. The law `cspF_reflex`, applied to subexpressions which are not rewritten.

Symmetrically, the tactic `cspF_hsf_right_tac` applies the Csp laws to processes in right-hand sides. In addition, a tactic `cspF_hsf_tac` is given as the combination of `cspF_hsf_left_tac` and `cspF_hsf_right_tac`, thus you can apply a tactic `cspF_hsf_tac` to sequentialise processes in both-hand sides, by the following command:

apply `(tactic {* cspF_hsf_tac` n `*})`

where the tactic is applied to nth subgoal. Note that you may consecutively apply `cspF_hsf_tac` more than twice because an application of a Csp law can make the other Csp law applicable. If you want to automatically apply the tactic as repeatedly as possible, the Isabelle option $+$, which expresses one or more repetitions, is useful:

apply `(tactic {* cspF_hsf_tac` n `*})+`

Take a look at the proof-script of the lemma `tactical_proof` in `Test_proof.thy` in `Test`. By the tactic, the lemma is easily proven.

Another useful tactic is `cspF_simp_with_tac`[13], which tries to apply a law proven by users to every subexpression. Thus, it repeatedly decomposes a process, check whether the law can be applied to subexpressions, and then applies it if possible. For example, assume that the following law has already been proven:

lemma `new_law: "`$(P \,\Box\, Q) \,\Box\, P =_{\mathcal{F}} (P \,\Box\, Q)$`"`

Then, you can apply the law to every subexpression in the subgoal 1 by

apply `(tactic {* cspF_simp_with_tac "new_law" 1 *})`

[13]Similarly to the case of `cspF_hsf_tac`, `cspF_simp_with_tac` is the combination of `cspF_simp_with_left_tac` and `cspF_simp_with_right_tac`.

To simultaneously apply two or more proven laws, the command **lemmas** will be useful, for example to combine `law1` and `law2` to `laws`:

```
lemmas laws = law1 law2
```

An example in which the tactic `cspF_simp_with_tac` is often used is given in theory-files in the package `DM`. Also, the tactic can be used for separating a large proof into some partial proofs.

The other tactics are given as sub-tactics of the tactic `cspF_hsf_tac` for avoiding meaningless rewriting. For example the tactic `cspF_step_tac` focuses on applying step-laws, thus it applies `cspF_choice_IF`, `cspF_SKIP_DIV_resolve`, `cspF_step`, `cspF_free_decompo`, and `cspF_reflex`, while it does not apply distributive laws or unwinding laws. Similarly, the tactics `cspF_dist_tac`, `cspF_unwind_tac`, and `cspF_sort_tac` focus on distribution, unwinding, and sorting, respectively.

9 Conclusion

This User-Guide is a draft version, and will be updated near future. Please keep to check the Csp-Prover's web site:

`http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`

Also Csp-Prover is still being developed and improved. Your feedback would be very welcome!

References

[Asp00] D. Aspinall. Proof general: A generic tool for proof development. In *TACAS 2000*, LNCS 1785, pages 38–42. Springer, 2000.

[CS01] E. M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science, 2001.

[ep202] *eft/pos 2000 Specification, version 1.0.1.* EP2 Consortium, 2002.

[IR05] Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In *TACAS 2005*, LNCS 3440, pages 108–123. Springer, 2005.

[NPW02] T. Nipkow, L. C. Paulon, and M. Wenzel. *Isabelle/HOL*. LNCS 2283. Springer, 2002. `http://www4.in.tum.de/~nipkow/LNCS2283/`.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998. Or No.68 in `http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/pubs.html`.