User Guide CSP-Prover Ver.4.0

CSP-Prover Document
Version: DRAFT November 19, 2007

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 **2**

2 Installing Isabelle2005 **3**

3 Setting up CSP-Prover **4**

4 Starting CSP-Prover **5**

5 Small demonstration **6**

6 The CSP-dialect CSP_{TP} **6**

6.1 Syntax 6

6.2 Semantics 10

7 Encoding of the CSP_{TP} **14**

7.1 Syntax 15

7.2 Domain 18

7.3 Semantics 21

7.4 Recursive process 25

8 Verification **27**

8.1 Semantical proof 27

8.2 Syntactical manual sproof 29

8.3 Syntactical semi-automatic proof 41

9 Conclusion **43**

A Guarded function **44**

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-4-0.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 5 packages of theory-files: `CSP`, `CSP_T`, `CSP_F`, `DFP`, and `FNF_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 the package `FNF_F` (Full Normal Form for the model $\mathcal{F}$) implemented on `CSP_F`. The package `DFP` (Deadlock-Freedom Proof Package) provides some theorems used for proving deadlock freedom.

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-4-0.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-4-0`

by an unpacking command (e.g. `tar zxvf CSP-Prover-4-0.tar.gz`).

`CSP-Prover-4-0` 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 $\mathcal{F}$ (see [IR06]),
- `DFP` : the theory for proving deadlock freedom (see [IRG05]),
- `DM` : an example to verify the Dining Mathematicians[CS01] (see [IR05]),
- `ep2` : an industrial case study on an electronic payment system **ep2**[ep202] (see [IR05]),
- `Test` : small examples for testing Csp-Prover,
- `SA_Kung` : Kung's systolic array for the multiplication of $n \times n$ matrices (see [IRG05]),

It is recommended to make heap files: `CSP`, `CSP_T`, `CSP_F`, `FNF_F`, and `DFP`. After making the heap files once, you do not have to prove them again before using them. You can make the 5 heap files by one command

```
% make_heaps
```

at the directory `/usr/local/CSP-Prover-4-0/`, 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 Isabella 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 five heap files. For example, about 15 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-4-0/`. 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-4-0/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-4-0/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-4-0/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 CSP_{TP}

This section summarises syntax and semantics of the CSP-dialect Csp_{TP}, which is the input language of Csp-Prover, and then we show that it can deal with infinitely many mutual recursive processes. The subscript `TP` of Csp_{TP} represents *Theorem Proving*.

6.1 Syntax

This section defines syntax of Csp_{TP}: Given a type of process names Π and an alphabet of communications Σ. Figure 1 shows the syntax of processes in Csp_{TP}, where Nat is the set of natural numbers and $Choice(\Sigma) = \mathbb{P}(\mathbb{P}(\Sigma)) \uplus \mathbb{P}(Nat)$. The set of processes are denoted by $Proc_{(\Pi,\Sigma)}$. Note that replicated internal choice takes an index set $C \in Choice(\Sigma)$ as its parameter, thus $C \subseteq \mathbb{P}(\Sigma)$ or

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

where $X \subseteq \Sigma$, $C \in \mathit{Choice}(\Sigma)$, $b \in \mathit{Bool}$, $r \in \mathbb{P}(\Sigma \times \Sigma)$, $n \in \mathit{Nat}$, and $p \in \Pi$.

Figure 1: Syntax of basic CSP_{TP} processes in CSP-Prover.

$C \subseteq \mathit{Nat}$. To specify the type of the parameter, we often use the following syntax:

$$!\texttt{set } X : Xs \bullet P(X) := !! c : Xs \bullet P(c)$$
$$!\texttt{nat } n : N \bullet P(n) := !! c : N \bullet P(c)$$

where $Xs \subseteq \mathbb{P}(\Sigma)$ and $N \subseteq \mathit{Nat}$. This syntax is convenient in the theorem prover Isabelle because it is difficult to directly assign such two different types ($\mathbb{P}(\Sigma)$ and Nat) to a variable C. By a similar reason, the symbol \$ is attached to each process name, in order to convert the type from process names to processes. In Isabelle, they have to be explicitly distinguished.

To avoid too many parentheses, the operators have the decreasing binding power, in the following order: conditional, {hiding, renaming, depth restriction}, action prefix, prefix choice, sequential composition, generalized parallel, external choice, replicated internal choice, internal choice.

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 internal choice over an arbitrary class of processes $\mathcal{P} \subseteq \mathit{Proc}_{(\Pi,\Sigma)}$, internal choice is restricted to run over an indexed set of processes $P(\cdot) : \mathbb{P}(\Sigma) \uplus \mathit{Nat} \Rightarrow \mathit{Proc}_{(\Pi,\Sigma)}$ only, where $C \in \mathit{Choice}(\Sigma)$. 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

[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.

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:

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

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

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

  where $R \subseteq \mathit{Real}$, $\mathit{real} : \mathit{Real} \Rightarrow \Sigma$, $\{\mathit{real}(r) \mid r \in \mathit{Real}\} \subseteq \Sigma$, and Real is the set of real numbers.

- 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 index set in prefix choice, replicated internal choice, 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:

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

- Inductive alphabetized parallel:

$$[|||]\ \langle\rangle := \texttt{SKIP}$$
$$[|||]\ (P, X) \frown PX_{list} := P\ [\![X, Y]\!]\ ([|||]\ PX_{list})$$

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

$$set(\langle\rangle) = \emptyset$$
$$set(\langle e\rangle \frown tail) = \{e\} \cup set(tail)$$

- Replicated alphabetized parallel:

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

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

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

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

$$map\ f\ \langle\rangle = \langle\rangle$$
$$map\ f\ (\langle e\rangle \frown tail) = \langle f(e)\rangle \frown (map\ f\ tail)$$

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.

In Csp, process names are defined by equations of the following form: for each process name $p \in \Pi$,

$$p = P$$

where $P \in Proc_{(\Pi,\Sigma)}$. Intuitively, it means that the process name p behaves like the process P. Since P can contain process names, it allows one to describe recursive processes. For example, a process A, which alternately performs events a and b iteratively, and can perform c just after b and then successfully terminates, is defined as follows:

$$A = a \rightarrow \$B$$
$$B = (b \rightarrow \$A) \,\Box\, (c \rightarrow \texttt{SKIP})$$

Csp_{tp} provides a special function $\texttt{PNfun}_\Pi : \Pi \Rightarrow Proc_{(\Pi,\Sigma)}$, which is called a *process-name function*, in order to describe the right hand sides of defining equations. Thus, it means that each process name $p \in \Pi$ behaves like the process $\texttt{PNfun}_\Pi(p)$. For example, the example A above is defined by the following function:

$$\texttt{PNfun}_\Pi\ \ A = a \rightarrow \$B$$
$$\texttt{PNfun}_\Pi\ \ B = (b \rightarrow \$A) \,\Box\, (c \rightarrow \texttt{SKIP})$$

where $\Sigma = \{a, b, c\}$ and $\Pi = \{A, B\}$.

Process names can include parameters. For example, process names *Inc*, which iteratively sends an increasing natural number n from 0 after *start*, can be defined by the following process-name function $\texttt{PNfun}_\Pi$:

$$\begin{array}{ll} \texttt{PNfun}_\Pi & Inc = start \rightarrow \$(Loop(0)) \\ \texttt{PNfun}_\Pi & (Loop(n)) = n \rightarrow \$(Loop(n+1)) \end{array}$$

where $\Sigma = \{start\} \cup Nat$ and $\Pi = \{Int\} \cup \{Loop(n) \mid n \in Nat\}$. Clearly, this process has infinite states.

We often need to replace a process name $p \in \Pi_1$ by a process $f(p)$, where f is a function such that $f : \Pi_1 \Rightarrow Proc_{(\Pi_2,\Sigma)}$. Therefore, we define the operation

$$\_ \lhd \_ : Proc_{(\Pi_1,\Sigma)} \Rightarrow (\Pi_1 \Rightarrow Proc_{(\Pi_2,\Sigma)}) \Rightarrow Proc_{(\Pi_2,\Sigma)}$$

for the replacement. Thus, $P \lhd f$ is the process obtained from P by replacing every process name p by $f(p)$. For the example *Inc* above, it can be unwound by the operator as follows:

$$\begin{array}{rl} (Inc \lhd \texttt{PNfun}_\Pi) \lhd \texttt{PNfun}_\Pi & = (start \rightarrow Loop(0)) \lhd \texttt{PNfun}_\Pi \\ & = start \rightarrow 0 \rightarrow Loop(1) \end{array}$$

This substitution operator is extended over functions as follows:

$$\begin{array}{l} \_ \lhd\lhd \_ \ : \ (\Pi_1 \Rightarrow Proc_{(\Pi_2,\Sigma)}) \Rightarrow (\Pi_2 \Rightarrow Proc_{(\Pi_3,\Sigma)}) \Rightarrow (\Pi_1 \Rightarrow Proc_{(\Pi_3,\Sigma)}) \\ f \lhd\lhd g = (\lambda\, p.\ f(p) \lhd g) \end{array}$$

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

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

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

When (Π, Σ)-*model* M is given, the semantics parameterized by M of a process P is defined by $[\![P]\!]_{\mathcal{F}(M)}$, where M is used for giving meanings to process names in the stable failures model $\mathcal{F}$, (i.e. $M : \Pi \Rightarrow \mathcal{F}_\Sigma$), and $[\![\cdot]\!]_{\mathcal{F}(M)}$ is a map $(Proc_{(\Pi,\Sigma)} \Rightarrow \mathcal{F}_\Sigma)$ expressed with the help of two functions:

$$[\![P]\!]_{\mathcal{F}(M)} = (traces_{(\mathtt{fst} \circ M)}(P), failures_M(P)).$$

where `fst` is a function for extracting the first component from a pair and $\circ$ is the composition operator of two functions, thus $(\mathtt{fst} \circ M)$ is the (Π, Σ)-model for giving a meaning to each process name in the traces model $\mathcal{T}$ (i.e. $(\mathtt{fst} \circ M) : \Pi \Rightarrow \mathcal{T}_\Sigma$), obtained from M. Then, the functions $traces_M$ and $failures_M$ are recursively defined by the semantic clauses given in Figure 2. Our definitions of $traces_M$ and $failures_M$ are identical to those given in [Ros98] except that the (Π, Σ)-models M are explicitly attached and the clauses of our two 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:

- $t_1 \,[\![\, X \,]\!]\, t_2$ 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:

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

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

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

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

Figure 2: Semantic clauses for the model $\mathcal{F}$ in our $\textsc{Csp}_{\texttt{TP}}$.

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

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

Now, we consider how to decide the (Π, Σ)-model M. As explained in Subsection 6.1, it is assumed that each process name p behaves like the process $\mathtt{PNfun}_\Pi(p)$. Therefore, the (Π, Σ)-model M has to be given so as to satisfy the following equation: for all $p \in \Pi$,

$$[\![\$p]\!]_{\mathcal{F}(M)} = [\![\mathtt{PNfun}_\Pi(p)]\!]_{\mathcal{F}(M)}$$

Since $[\![\$p]\!]_{\mathcal{F}(M)} = M(p)$, this can be rewritten to the following form:

$$M = [\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{F}}(M)$$

where $[\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{F}}(M) = (\lambda\, p.[\![\mathtt{PNfun}_\Pi(p)]\!]_{\mathcal{F}(M)})$. Consequently, the (Π, Σ)-model M is a fixed point of the function $[\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{F}}$.

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 cpo approach shows that the function $[\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{F}}$ has the least fixed point for any process-name function $\mathtt{PNfun}_\Pi$. On the other hand, the cms approach shows that the function $[\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{F}}$ has the unique fixed point if the process $\mathtt{PNfun}_\Pi(p)$ is *guarded* [4], for every process name $p \in \Pi$.

Since the definition of process names depends on which approach is used, $\text{Csp}_{\mathtt{TP}}$ has a reserved word `FPmode`, which takes either `CMSmode` or `CPOmode` or `MIXmode`. If `FPmode` = `CMSmode`, then the unique fixed point of $[\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{F}}$ is used, but it cannot deal with unguarded processes. If `FPmode` = `CPOmode`, then the least fixed point of $[\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{F}}$ is used. It can deal with all processes, but the uniqueness of the fixed point is not guaranteed. If `FPmode` = `MIXmode`, then the advantages of the both approach are available. Thus, the least fixed point of $[\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{F}}$ is used for all processes, but the uniqueness is guaranteed for guarded processes. Therefore, the ideal (Π, Σ)-model, written $\mathtt{MF}_\Pi$, is given as follows:

$$\mathtt{MF}_\Pi = \mathtt{FIX}([\![\mathtt{PNfun}_\Pi]\!]^{fun}_{\mathcal{F}})$$

where the function `FIX` is defined as follows:

$$\mathtt{FIX}(fun) \ = \text{if } \mathtt{FPmode} = \mathtt{CMSmode} \text{ then } \mathtt{UFP}(fun) \text{ else } \mathtt{LFP}(fun)$$

[4] The definition of guardedness is given in Appendix A.

where UFP and LFP represent the unique fixed point and the least fixed point, respectively. Finally, the semantics $[\![P]\!]_{\mathcal{F}}$ of each process P is defined as follows:

$$[\![P]\!]_{\mathcal{F}} = [\![P]\!]_{\mathcal{F}(\mathtt{MF}_\Pi)}$$

Given two models M_1 and M_2 for two sets of process names Π_1 and Π_2 respectively, parameterized process equivalence $=_{\mathcal{F}(M_1,M_2)}$ and parameterized process refinement $\sqsubseteq_{\mathcal{F}(M_1,M_2)}$ over the stable failures model are then defined as follows:

$P =_{\mathcal{F}(M_1,M_2)} Q \Leftrightarrow$
$\quad traces_{(\mathtt{fst}\circ M_1)}(P) = traces_{(\mathtt{fst}\circ M_2)}(Q) \wedge failures_{M_1}(P) = failures_{M_2}(Q),$
$P \sqsubseteq_{\mathcal{F}(M_1,M_2)} Q \Leftrightarrow$
$\quad traces_{(\mathtt{fst}\circ M_1)}(P) \supseteq traces_{(\mathtt{fst}\circ M_2)}(Q) \wedge failures_{M_1}(P) \supseteq failures_{M_2}(Q).$

Then, since the ideal (Π, Σ)-model is $\mathtt{MF}_\Pi$, process equivalence $=_{\mathcal{F}}$: $Proc_{(\Pi_1,\Sigma)} \times Proc_{(\Pi_2,\Sigma)} \Rightarrow Bool$ and process refinement $\sqsubseteq_{\mathcal{F}}$: $Proc_{(\Pi_1,\Sigma)} \times Proc_{(\Pi_2,\Sigma)} \Rightarrow Bool$ are defined as follows:

$$P =_{\mathcal{F}} Q \Leftrightarrow P =_{\mathcal{F}(\mathtt{MF}_{\Pi_1},\mathtt{MF}_{\Pi_2})} Q,$$
$$P \sqsubseteq_{\mathcal{F}} Q \Leftrightarrow P \sqsubseteq_{\mathcal{F}(\mathtt{MF}_{\Pi_1},\mathtt{MF}_{\Pi_2})} Q.$$

Then, as expected, the following properties hold:

1. Let $\mathtt{FPmode} = \mathtt{CPOmode}$. Then,
   - $\forall\, p \in \Pi.\ \$p =_{\mathcal{F}} \mathtt{PNfun}_\Pi(p)$,
   - $\forall f.\ ((\forall\, p \in \Pi.\ f(p) =_{\mathcal{F}} \mathtt{PNfun}_\Pi(p) \triangleleft f) \Longrightarrow (\forall\, p \in \Pi.\ f(p) \sqsubseteq_{\mathcal{F}} \$p))$.
2. Let $\mathtt{FPmode} = \mathtt{CMSmode}$ and $\mathtt{PNfun}_\Pi(p)$ be guarded for any p. Then,
   - $\forall\, p \in \Pi.\ \$p =_{\mathcal{F}} \mathtt{PNfun}_\Pi(p)$,
   - $\forall f.\ ((\forall\, p \in \Pi.\ f(p) =_{\mathcal{F}} \mathtt{PNfun}_\Pi(p) \triangleleft f) \Longrightarrow (\forall\, p \in \Pi.\ f(p) =_{\mathcal{F}} \$p))$.

Thus, both ways of CSP for 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 available also in $\text{CSP}_{\mathtt{TP}}$.

7 Encoding of the $\text{CSP}_{\mathtt{TP}}$

This section shows how $\text{CSP}_{\mathtt{TP}}$ introduced in Section 6 is encoded in the generic theorem prover Isabelle.

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
$P \llbracket X \rrbracket Q$	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	$p	process name
$P \triangleright Q$	P [> Q	timeout
!set $X : Xs \bullet P(X)$	!set $X : \mathcal{X}f$ .. $P(X)$	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
$P \interleave Q$	P ||| Q	interleaving
$P \parallel Q$	P || Q	synchronous
$P \llbracket X, Y \rrbracket Q$	P |[X,Y]| Q	alphabetized parallel
$[\!\|\!\|] i : I \bullet (P_i, X_i)$	[||] i : I .. (P_i, X_i)	replicated alphabetized parallel

Figure 3: The ASCII expression of Csp processes.

7.1 Syntax

At first, we give ASCII style expressions of Csp processes in Figure 3 because the conventional operators use TeX symbols[5]. These are trivial translations from TeX symbols to ASCII symbols. 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. And, 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

[5]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.

```
types
   'a sets_nats = "('a set set, nat set) sum"
   'a aset_anat = "('a set, nat) sum"

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

Figure 4: The recursive definition of the process type.

X `<=` Y, respectively.

Now, the set of (basic) processes is given as a recursive type `('p,'a) proc` which is defined by the Isabelle command **datatype** as shown in Figure 4, where `'p` is the type of process names Π and `'a` is the type of communications Σ. Here, note that the types of index sets `'a sets_nats` in replicated internal choice. As explained in Section 6, an index-set of replicated internal choice is either a subset of subsets of communications or a subset of natural numbers, thus the type of index-sets is the disjoint of `"'a set set"` and `"nat set"`. Therefore, it is defined by `"('a set set, nat set) sum"` as shown at the top of Figure 4, where `sum` is the *disjoint sum type* and is defined by

datatype `('a,'b) sum = type1 "'a"  |  type2 "'b"`

and some lemmas on `sum` are proven in the theory file `CSP/Infra_type.thy`.

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,

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

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

 "@Rep_int_choice" ::
  "pttrn ⇒ 'a sets_nats ⇒ 'a aset_anat ⇒ ('p,'a) proc" ("!!_ : _ • _")

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

Figure 5: The expression of bound variables.

is defined by

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

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.

Derived operators such as $\triangleright$ are also 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. Here, note replicated internal choice over sets of communications in the middle of the figure. We explained that it is defined as

$$!\texttt{set}\ X : Xs \bullet P(X) := !!\, c : Xs \bullet P(c)$$

in Section 6, but more exactly, it has to be defined by explicitly considering the type conversion by `type1`, which is one of type-constructors of the disjoint-sum type `sum`, as follows:

$$!\texttt{set}\ X : Xs \bullet P(X) := !!\, c : (\texttt{type1}\ Xs) \bullet P(\texttt{type1}^{-1}(c))$$

in Isabelle. You can find all the definitions of process expressions in the the theory-file `CSP_syntax.thy` in the package `CSP`.

As explained in Subsection 6.1, CSP$_{\texttt{TP}}$ has two reserved words, `PNfun` and `FPmode`, for dealing with fixed points. The types of these words are declared in the theory file `CSP/CSP_syntax.thy` as follows, but they are not defined there.

```
consts PNfun :: "'p ⇒ ('p,'a) proc"
datatype fpmode = CPOmode | CMSmode | MIXmode
consts FPmode :: "fpmode"
```

They are defined by users, and especially, `PNfun` is defined for each set of process names by using the command `defs` with the option *overloaded* as explained in Subsection 7.4

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

%% replicated internal choice over sets
consts
 Rep_int_choice_set ::
   "'a set set ⇒ ('a set ⇒ ('p,'a) proc) ⇒ ('p,'a) proc"
                                                    ("!set :_ • _")
defs
 Rep_int_choice_set_def :
  "!set :Xs • P == !! c:(type1 Xs) • (P((inv type1) c))"

%% replicated internal choice over communications
consts
 Rep_int_choice_com ::
   "('a set ⇒ ('a ⇒ ('p,'a) proc) ⇒ ('p,'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.

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 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:

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

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[6] 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.

[6]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.

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 (domF, $\leq$) can be proven to be an instance of the type class of partial ordered set as follows:

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

As shown above, 'a domF is not ('a domT * 'a setF) but is its subtype defined by **typedef**. Therefore, it is convenient to define the following notations in order to directly express pairs in 'a domF and extract the first or the second component from them.

```
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"
```

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 $[\![X]\!]_{\mathtt{tr}}$ and $\backslash_{\mathtt{tr}}$ (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 $[[r]]^{\mathtt{inv}}$ are given in Trace_ren.thy in the package CSP. And, (rmTick *s*) is the trace obtained by removing ✓ from *s* and it is encoded in Trace_seq.thy. Furthermore, $\downarrow$ (written .|. in ASCII) is a restriction function which is given over both domT and setF, in

```
consts traces :: "('p,'a) proc ⇒ ('p ⇒ 'a domT) ⇒ 'a domT"

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

Figure 7: The encoding of the function *traces*

`Domain_T_cms.thy` in the package `CSP_T` and `Set_F_cms.thy` in `CSP_F`, respectively. The function `sumset` is used for mapping type-constructors, `type1` and `type2`, of disjoint-sum of two set types into elements in the sets and is defined in `CSP/Infra_type.thy` as follows:

```
consts
  sumset :: "('a set, 'b set) sum => ('a,'b) sum set"
primrec
  "sumset (type1 X) = {type1 a |a. a ∈ X}"
  "sumset (type2 X) = {type2 a |a. a ∈ X}"
```

Here, note that the type of index-sets used in replicated internal choice is not `"('a set, nat) sum set"`, but is `"('a set set, nat set) sum"`. The first `"('a set, nat) sum set"` allows one to mix sets of alphabets and natural numbers in an index-set like $\{0, 1, \{a\}, \{b, c\}\}$, but our CSP-dialect CSP_{TP} does not allow it.

As explained in Subsection 6.2, the parameterized semantics $[\![P]\!]_{\mathcal{F}(M)}$ of each

```
consts failures ::  "('p,'a) proc ⇒ ('p ⇒ 'a domF) ⇒ 'a setF"

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

Figure 8: The encoding of the function *failures*

process P with respect to the (Π, Σ)-model M and $[\![f]\!]^{fun}_{\mathcal{F}}$ of each process function f are defined with the help of the two functions `traces` and `failures` as follows (see `CSP_F_semantics.thy`):

```
consts
 semFf   :: "('p,'a) proc ⇒ ('p ⇒ 'a domF) ⇒ 'a domF"   ("[[_]]Ff")
 semFfun :: "('p ⇒ ('p,'a) proc) ⇒
             ('p ⇒ 'a domF) ⇒ ('p ⇒ 'a domF)"           ("[[_]]Ffun")

defs
  semFf_def  : "[[P]]Ff == (λ M. (traces(P) M ,, failures(P) M))"
  semFfun_def: "[[f]]Ffun == (λ M. λ p. [[f(p)]]Ff M)"
```

Then, the ideal (Π, Σ)-model $M_{\mathcal{F}}$ which give proper meanings to process names, the process equivalence $=_{\mathcal{F}}$ (written `=F` in ASCII), and the process refinement $\sqsubseteq_{\mathcal{F}}$ (written `<=F` in ASCII) are defined as shown in Figure 9. Here, it is important to check that `(traces(`P`) (fstF ∘` M`), failures(`P`) `M`)` is in `domF` indeed. It is proven in the following lemma (see `CSP_F_domain.thy`):

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

This lemma allows us to prove the following expected properties:

```
lemma fstF_semF[simp]: "fstF [[P]]F = traces(P)(fstF ∘ MF)""
lemma sndF_semF[simp]: "sndF [[P]]F = failures(P) MF"
lemma cspF_eqF_semantics: "(P =F[M1,M2] Q) =
   ((traces(P)(fstF ∘ M1) = traces(Q)(fstF ∘ M2))∧
   (failures(P)(M1) = failures(Q)(M2)))"
lemma cspF_refF_semantics: "(P ⊑F[M1,M2] Q) =
   ((traces(Q)(fstF ∘ M2) ≤ traces(P)(fstF ∘ M1))∧
   (failures(Q)(M2) ≤ failures(P)(M1)))"
```

At the end of this subsection, we would like to briefly tell the expressive power of our Csp dialect Csp_{TP}. At first glance, the input language of Csp-Prover seems to be weaker than full Csp as the generic internal choice operator $\sqcap \mathcal{P}$ [7] 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"
```

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

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

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

```
(* fixed point and semantics *)

consts
 semFfix :: "('p ⇒ ('p,'a) proc) ⇒ ('p ⇒ 'a domF)" ("⟦_⟧Ffix")
 MF      :: "('p ⇒ 'a domF)"
 semF    :: "('p,'a) proc ⇒ 'a domF"                ("⟦_⟧F")

defs
 semFfix_def : "⟦f⟧Ffix == (if (FPmode = CMSmode)
                            then (UFP (⟦f⟧Ffun))
                            else (LFP (⟦f⟧Ffun)))"
 MF_def      : "MF == ⟦PNfun⟧Ffix"
 semF_def    : "⟦P⟧F == ⟦P⟧Ff MF"

(* process equation and refinement *)

consts
 refF :: "('p,'a) proc ⇒ ('p ⇒ 'a domF) ⇒ ('q ⇒ 'a domF) ⇒
          ('q,'a) proc ⇒ bool" ("_ ⊑F[_,_] _")
 eqF  :: "('p,'a) proc ⇒ ('p ⇒ 'a domF) ⇒ ('q ⇒ 'a domF) ⇒
          ('q,'a) proc ⇒ bool" ("_ =F[_,_] _")

defs
 refF_def :  "P1 ⊑F[M1,M2] P2 == ⟦P2⟧Ff M2 ≤ ⟦P1⟧Ff M1"
 eqF_def  :  "P1 =F[M1,M2] P2 == ⟦P1⟧Ff M1 = ⟦P2⟧Ff M2"

syntax
 "_refFfix" :: "('p,'a) proc ⇒ ('q,'a) proc ⇒ bool" ("_ ⊑F _")
 "_eqFfix"  :: "('p,'a) proc ⇒ ('q,'a) proc ⇒ bool" ("_ =F _")

translations
 "P1 ⊑F P2" == "P1 ⊑F[MF,MF] P2"
 "P1 =F P2" == "P1 =F[MF,MF] P2"
```

Figure 9: The encoding of process equation and refinement

7.4 Recursive process

In this subsection, we show how to encode recursive processes into Csp-Prover by using the following example:

$$\begin{aligned} \mathtt{PNFun}_{\Pi} \quad (\mathtt{Empty(id)}) &= \mathtt{left?r} \rightarrow \$(\mathtt{Full(r,id)}) \\ \mathtt{PNFun}_{\Pi} \quad (\mathtt{Full(r,id)}) &= \mathtt{right(r,id)} \rightarrow \$(\mathtt{Empty(id+1)}) \end{aligned}$$

where the set of process names Π and the alphabet Σ are given as follows:

$$\begin{aligned} \Pi &= \{\mathtt{Empty(id)} \mid \mathtt{id} \in \mathtt{Nat}\} \cup \{\mathtt{Full(r,id)} \mid \mathtt{r} \in \mathtt{Real}, \mathtt{id} \in \mathtt{Nat}\} \\ \Sigma &= \{\mathtt{left(r)} \mid \mathtt{r} \in \mathtt{Real}\} \cup \{\mathtt{right(r,id)} \mid \mathtt{r} \in \mathtt{Real}, \mathtt{id} \in \mathtt{Nat}\} \end{aligned}$$

```
 1  datatype Event = left real | right "real * nat"
 2  datatype Name  = Empty nat | Full real nat
 3
 4  consts Bufferfun :: "Name ⇒ (Name, Event) proc"
 5  primrec
 6     "Bufferfun  (Empty n) = left?r → $(Full r n)"
 7     "Bufferfun (Full r n) = right(r,n) → $(Empty (Suc n))"
 8  defs (overloaded) Set_Bufferfun_def [simp]: "PNfun == Bufferfun"
 9
10  consts Buffer :: "(Name, Event) proc"
11  defs Buffer_def: "Buffer == $(Empty 0)"
```

Figure 10: An encoding of the example `Buffer`

where `Nat` and `Real` are the set of natural numbers and the set of real numbers. Now, let

$$\texttt{Buffer} = \$(\texttt{Empty}(0)).$$

Then, the process `Buffer` iteratively receives a real number `r` from the channel `left` and sends it to a channel `right` together with an increasing natural number `id` whose initial value is `0`.

The process `Buffer` can be encoded into Csp-Prover as shown in Figure 10. In the lines 1 and 2, the types of alphabets `Event` and process names `Name` are declared. Next, the function `Bufferfun` for defining each process name is defined by the Isabelle command **primrec** (line 5), which is useful for defining recursive processes in a conventional style because pattern matching on the first argument is available. Finally, the function `Bufferfun` is defined to be `PNfun` (line 8). It means that `Bufferfun` is automatically used for giving the meaning to process names in `Names`. There are a number of ways to define functions in Isabelle, but this is a good and simple way to define recursive processes in Csp-Prover.

As explained in Subsection 6.2, process-name functions have to be guarded when the cms approach is employed for dealing with unique fixed points. In Csp-Prover, it is almost automatized to prove the guardedness. For example, it can be easily proven that `Bufferfun` is guarded by the following three steps (see `Test_Buffer.thy` in `Test`):

```
lemma guardedfun_Bufferfun[simp]:
      "guardedfun Bufferfun"            %% goal to be proven
apply (simp add: guardedfun_def, rule)  %% unfolding guardedfun_def
apply (induct_tac p)                    %% instantiating names
apply (simp_all)                        %% automatic proof
done
```

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

```
goal (lemma (guardedfun_Bufferfun), 1 subgoal):
1. ⋀p. guarded (Bufferfun p)
```

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

```
goal (lemma (guardedfun_Bufferfun), 2 subgoals):
1. ⋀p nat. guarded (Bufferfun (Empty nat))
2. ⋀p real nat. guarded (Bufferfun (Full real nat))
```

Then, the subgoals can be automatically proven by `simp`s. This proof strategy is available to most of proofs for guardedness.

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) syntactical manual proof by algebraic CSP laws, and (3) syntactical semi-automatic 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), according to the semantic clauses, see Figures 7 and 8. For example, when a subgoal contains the following form,

$$\cdots \wedge\ t \in_{\mathrm{t}} \texttt{traces}(a \rightarrow P)\, M\ \wedge \cdots$$

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

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

On the other hand, if the command `(simp add: traces.simps)`[8] was applied

[8] 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]`.

instead of `(simp add: in_traces)`, it would be rewritten to the following subgoal (∗2):

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

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

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

thus the transformation 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

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

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

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

   $$\mathtt{(traces}(P)\mathtt{(}M\mathtt{)} \leq \mathtt{traces}(Q)\mathtt{(}M\mathtt{))} \wedge$$
   $$\mathtt{(traces}(Q)\mathtt{(}M\mathtt{)} \leq \mathtt{traces}(P)\mathtt{(}M\mathtt{))}$$

   by `(rule order_antisym)`.

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

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

   by `(rule subdomTI)`.

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

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

Now, take a look at the proof script of the lemma `semantical_proof` in `Test_proof.thy` in `Test`. The proof proceeds according to the above instruction[9]. During the proof, the subgoals are sometimes complex. It will be found that CSP-Prover assists the proof well.

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

8.2 Syntactical manual sproof

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 11, ⋯, 17. The Csp laws and their names such as (□-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 11, ⋯, 17 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 process P, Q 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 discussed the completeness in [IR06], and you can find the whole proof in the theory-files in the package `FNF_F`.

In Figures 11, ⋯, 17, 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_decompo"` 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` [10] which side of $=_{\mathcal{F}}$ is rewritten. For example, by applying `(rule cspF_rw_left)`, the main goal is rewritten to

```
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* which is automatically generated by Isabelle. Such variable will be instantiated later.

[10] 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)

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

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

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

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

Figure 11: 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"

$[\!\mid\ C \neq \emptyset;\ \forall\, c \in C.\, P(c) = Q\ \mid\!] \Longrightarrow\ !!\, c : C \bullet P(c) =_{\mathcal{F}} Q$	(!!-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 12: CSP basic laws

"cspF_Dist"

$C \neq \emptyset \Longrightarrow (!!\, c : C \bullet P(c)) \,\Box\, Q =_{\mathcal{F}} !!\, c : C \bullet (P(c) \,\Box\, Q)$ (□-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)

$(!!\, c : C \bullet P(c)) \fatsemi Q =_{\mathcal{F}} !!\, c : C \bullet (P(c) \fatsemi Q)$ (⨟-Dist)

$(!!\, c : C \bullet P(c)) \lfloor n =_{\mathcal{F}} !!\, c : C \bullet (P(c) \lfloor n)$ (⌊ -Dist)$^+$

"cspF_dist"

$(P_1 \sqcap P_2) \,\Box\, Q =_{\mathcal{F}} (P_1 \,\Box\, Q) \sqcap (P_2 \,\Box\, Q)$ (□-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)

$(P_1 \sqcap P_2) \fatsemi Q =_{\mathcal{F}} (P_1 \fatsemi Q) \sqcap (P_2 \fatsemi Q)$ (⨟-dist)

$(P_1 \sqcap P_2) \lfloor n =_{\mathcal{F}} (P_1 \lfloor n) \sqcap (P_2 \lfloor n)$ (⌊ -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]]-□-dist)

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

Figure 13: Csp distributive laws

"cspF_step"

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

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

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

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

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

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

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

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

Figure 14: Csp step laws

"cspF_step_ext"

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

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

"cspF_Ext_choice_SKIP_DIV_resolve"

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

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

"cspF_Depth_rest_Zero"

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

"cspF_Depth_rest_min"

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

Figure 15: Csp extended step laws and depth-restriction laws

"cspF_SKIP_DIV"

$\texttt{SKIP} \,\square\, \texttt{DIV} =_{\mathcal{F}} \texttt{SKIP}$	(skip-div-$\square$)
$\texttt{SKIP} \,[\![X]\!]\, \texttt{SKIP} =_{\mathcal{F}} \texttt{SKIP}$	(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)
$\texttt{DIV} \,[\![X]\!]\, (?\, x : A \to P(x))$ $=_{\mathcal{F}}\ (?\, x : (A - X) \to (\texttt{DIV} \,[\![X]\!]\, P(x))) \,\square\, \texttt{DIV}$	(div-$[\![X]\!]$-step)
$\texttt{SKIP} \,[\![X]\!]\, ((?\, x : A \to P(x)) \,\square\, \texttt{SKIP})$ $=_{\mathcal{F}}\ (?\, x : (A - X) \to (\texttt{SKIP} \,[\![X]\!]\, P(x))) \,\square\, \texttt{SKIP}$	(skip-$[\![X]\!]$-$\square$-skip)
$\texttt{SKIP} \,[\![X]\!]\, ((?\, x : A \to P(x)) \,\square\, \texttt{DIV})$ $=_{\mathcal{F}}\ (?\, x : (A - X) \to (\texttt{SKIP} \,[\![X]\!]\, P(x))) \,\square\, \texttt{DIV}$	(skip-$[\![X]\!]$-$\square$-div)
$\texttt{DIV} \,[\![X]\!]\, (P \,\square\, \texttt{SKIP}) =_{\mathcal{F}} \texttt{DIV} \,[\![X]\!]\, P$	(div-$[\![X]\!]$-$\square$-skip)
$\texttt{DIV} \,[\![X]\!]\, (P \,\square\, \texttt{DIV}) =_{\mathcal{F}} \texttt{DIV} \,[\![X]\!]\, P$	(div-$[\![X]\!]$-$\square$-div)
$\texttt{SKIP} \setminus X =_{\mathcal{F}} \texttt{SKIP}$	(skip-hide)
$\texttt{DIV} \setminus X =_{\mathcal{F}} \texttt{DIV}$	(div-hide)
$((?\ x : A \to P(x)) \,\square\, \texttt{SKIP}) \setminus X$ $=_{\mathcal{F}}\ ((?\ x : (A - X) \to (P(x) \setminus X)) \,\square\, \texttt{SKIP})$ $\quad \sqcap\ (!\ x : (A \cap X) \bullet (P(x) \setminus X))$	(skip-hide-step)
$((?\ x : A \to P(x)) \,\square\, \texttt{DIV}) \setminus X$ $=_{\mathcal{F}}\ ((?\ x : (A - X) \to (P(x) \setminus X)) \,\square\, \texttt{DIV})$ $\quad \sqcap\ (!\ x : (A \cap X) \bullet (P(x) \setminus X))$	(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 16: CSP skip and div laws

"cspF_Rep_int_choice_input_set"

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

"cspF_Rep_int_choice_Ext_Dist"

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

"cspF_Rep_int_choice_input_Dist"

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

"cspF_norm"

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

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

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

"cspF_nat_Depth_rest"

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

Figure 17: Csp replicated internal choice laws and normalising laws

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 $\sqsubseteq_{\mathcal{F}}$ [11]. The additional laws for $\sqsubseteq_{\mathcal{F}}$ are shown Figure 18.

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

[11] 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_iff"

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

"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"

$[\!|\ P \sqsubseteq_{\mathcal{F}} Q_1;\ P \sqsubseteq_{\mathcal{F}} Q_2\ |\!] \Longrightarrow P \sqsubseteq_{\mathcal{F}} Q_1 \sqcap Q_2$ ($\sqcap$-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 18: Csp refinement laws

"cspF_FIX"

$[\!|\ \texttt{FPmode} = \texttt{CMSmode} \longrightarrow \texttt{guardedfun}\,\texttt{PNfun}\ |\!]$
$\Longrightarrow \$p =_{\mathcal{F}} (!\texttt{nat}\ n \bullet ((\texttt{PNfun}\triangleleft\!\triangleleft)^{(n)}(\lambda\, p'.\ \texttt{DIV}))(p)))$ (fix_!nat)

"cspF_unwind"

$[\!|\ \texttt{FPmode} = \texttt{CMSmode} \longrightarrow \texttt{guardedfun}\,\texttt{PNfun}\ |\!]$
$\Longrightarrow \$p =_{\mathcal{F}} \texttt{PNfun}(p)$ (unwind)

"cspF_fp_induct_right"

$[\!|\ \texttt{FPmode} = \texttt{CMSmode} \longrightarrow \texttt{guardedfun}\,\texttt{PNfun};$
$Q \sqsubseteq_{\mathcal{F}} f(p);\ \bigwedge p.\ f(p) \sqsubseteq_{\mathcal{F}} \texttt{PNfun}(p) \triangleleft f\ |\!]$
$\Longrightarrow Q \sqsubseteq_{\mathcal{F}} \$p$ (induct-right-ref)

$[\!|\ \texttt{FPmode} = \texttt{CMSmode};\ \texttt{guardedfun}\,\texttt{PNfun};$
$Q =_{\mathcal{F}} f(p);\ \bigwedge p.\ f(p) =_{\mathcal{F}} \texttt{PNfun}(p) \triangleleft f\ |\!]$
$\Longrightarrow Q =_{\mathcal{F}} \$p$ (induct-right-eq)

"cspF_fp_induct_left"

$[\!|\ \texttt{FPmode} = \texttt{CMSmode};\ \texttt{guardedfun}\,\texttt{PNfun};$
$f(p) \sqsubseteq_{\mathcal{F}} Q;\ \bigwedge p.\ \texttt{PNfun}(p) \triangleleft f \sqsubseteq_{\mathcal{F}} f(p)\ |\!]$
$\Longrightarrow \$p \sqsubseteq_{\mathcal{F}} Q$ (induct-left-ref)

$[\!|\ \texttt{FPmode} = \texttt{CMSmode};\ \texttt{guardedfun}\,\texttt{PNfun};$
$f(p) =_{\mathcal{F}} Q;\ \bigwedge p.\ \texttt{PNfun}(p) \triangleleft f =_{\mathcal{F}} f(p)\ |\!]$
$\Longrightarrow \$p =_{\mathcal{F}} Q$ (induct-left-eq)

Figure 19: Csp fixed-point laws

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. $(P \ \square\ Q) \ \square\ R =_{\mathcal{F}} P \ \square\ (Q \ \square\ R)$), 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 Figure 19. The law (fix-!nat) can replace fixed point by unbounded internal choice. It may be useful for theoretical work. On the other hand, the unwinding law and the fixed-point induction laws will be often used in practical verifications. The unwinding law is intuitively understandable, but the fixed-point induction

```
datatype DFName = DF

consts DFfun :: "DFName ⇒ (DFName, Event) proc"
primrec "DFfun (DF) = ! x → $DF"
defs (overloaded) Set_DFfun_def [simp]: "PNfun == DFfun"
```

Figure 20: The deadlock-free specification DF

laws might not be intuitive. We pick up the law (induct-right-ref) and explain it by using the example Buffer in Figure 10.

As a simple example, we verify Buffer is deadlock-free, thus $DF $\sqsubseteq_{\mathcal{F}}$ Buffer, where DF is the deadlock-free specification which always requires to perform an event at least, and is encoded into CSP-Prover as shown in Figure 20. The guardedness of DFfun can be proven by the same proof script used for Bufferfun. See the lemma manual_proof_Buffer in the theory file Test/Test_Buffer.thy. In this verification, we use the cms approach (i.e. FPmode=CMSmode). 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 $Empty 0
```

In general, the fixed point induction is applied at first. So, when you apply the fixed point induction to the goal by (rule cspF_fp_induct_right) and simp command twice, you will have the following two subgoals ($*$):

```
goal (lemma (manual_proof_Buffer), 2 subgoals):
1. $DF ⊑_F ?f(Empty 0)
2. ⋀p. ?f(p) ⊑_F (Bufferfun(p))◁?f
```

where the schematic variable $?f$ is a function used for relating each process name on the right hand side to a process on the left hand side. However, it is difficult to instantiate the variable $?f$ later. It would be better that such function $?f$ is given by users because it is hard to automatically find such functions, although CSP-Prover can *assist* them to find such functions. In this example, such function can be given as follows:

```
consts
  Buffer_to_DF :: "Name ⇒ (DFName, 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), 2 subgoals):
1. $DF ⊑_F Buffer_to_DF (Empty 0)
2. ⋀ p. Buffer_to_DF(p) ⊑_F (Bufferfun(p))◁Buffer_to_DF
```

The subgoal 1 is trivial because `Buffer_to_DF (Empty n) = $DF`. The variable p in the subgoal 2 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)`, the following two subgoals are obtained [12]:

```
goal (lemma (manual_proof_Buffer), 2 subgoals):
1. $DF ⊑_F ? a:(range left) → $DF
2. ⋀ real nat. $DF ⊑_F right (real, nat) → $DF
```

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

8.3 Syntactical semi-automatic 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 *tactics* to automatically 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 internal choice operators.

[12] In this proof, we added `csp_prefix_ss_def` into the simplification rules by

declare `csp_prefix_ss_def [simp]`

in order to automatically unfold the definition of syntactic sugar of sending and receiving because step-laws cannot directly be applied to the prefixes such as $a!x$ and $a?x$.

6. The law `cspF_unwind` 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.

The second 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 *})`

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`. The tactic can be used for separating a large proof into some partial proofs.

The tactic `cspF_refine_with_tac` is the same as `cspF_simp_with_tac` except that it rewrites every subexpression by refinement relations already proven. For example, assume that the following law has already been proven:

[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`.

lemma `new_ref_law: "`$P \triangleright Q \sqsubseteq_{\mathcal{F}} P \,\Box\, Q$`"`

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

apply `(tactic {* cspF_refine_with_tac "new_ref_law" 1 *})`

By this tactic, $\triangleright$ in the left hand side is replaced by $\Box$, while $\Box$ in the right hand side is replaced by $\triangleright$. For example, by this tactic,

$$(P_1 \triangleright P_2) \,\Box\, P_3 \sqsubseteq_{\mathcal{F}} (Q_1 \triangleright Q_2) \,\Box\, Q_3 \qquad (*_1)$$

is rewritten to

$$(P_1 \,\Box\, P_2) \,\Box\, P_3 \sqsubseteq_{\mathcal{F}} (Q_1 \triangleright Q_2) \triangleright Q_3. \qquad (*_2)$$

Note that $(*_2)$ implies $(*_1)$.

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.

[IR06] Yoshinao Isobe and Markus Roggenbach. A complete axiomatic semantics for the csp stable failures model. In *CONCUR 2006*, LNCS. Springer, 2006.

[IRG05] Y. Isobe, M. Roggenbach, and S. Gruner. Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays. In *FOSE 2005*, Japanese Lecture Notes Series 31. Kindai-kagaku-sha, 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`.

A Guarded function

As explained in Subsection 6.2, process-name functions have to be *guarded* when the cms approach is employed. In this section, we explain the notion of guard and define the function `guardedfun` for checking the guardedness.

In general, the process name p is said to be guarded in a process P (or P is guarded) if each occurrence of p is within some subexpression $a \rightarrow P'$ or $?\,x : A \rightarrow P'(x)$. However, we should deal with sequential composition $P_1 \fatsemi P_2$ more carefully, thus the question is if P_2 should be guarded or not? For example, the following processes (1) and (2) are guarded, but (3) is not guarded.

(1) $(a \rightarrow \texttt{SKIP}) \fatsemi \$p$
(2) $\texttt{SKIP} \fatsemi (a \rightarrow \$p)$
(3) $\texttt{SKIP} \fatsemi \$p$

It means P_2 does not have to be guarded if each occurrence of `SKIP` in P_1 are guarded. Therefore, for defining guardedness, we need a predicate for checking whether `SKIP` is guarded or not. Figure 21 shows the encoded predicate `gSKIP` used for guaranteeing that `SKIP` is guarded. 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 processes.

$$\texttt{gSKIP}(P) \text{ implies } \langle\checkmark\rangle \notin \texttt{traces}(P)\, M.$$

By a similar way, the following two predicates over processes also are defined:

- `noPN`(P): It means that P has no process name.
- `noHide`(P): It means that if P has a subexpression of the form $Q \setminus X$, then Q has no process name (i.e. `noPN`(Q)).

```
consts gSKIP :: "('p,'a) proc ⇒ bool"

primrec
 "gSKIP(STOP)       = True"
 "gSKIP(SKIP)       = False"
 "gSKIP(DIV)        = True"
 "gSKIP(a → P)      = True"
 "gSKIP(? : X → P)  = True"
 "gSKIP(P □ Q)      = gSKIP(P) ∧ gSKIP(Q)"
 "gSKIP(P ⊓ Q)      = gSKIP(P) ∧ gSKIP(Q)"
 "gSKIP(!! : C • P)  = (∀ c ∈ C. gSKIP(P(c)))"
 "gSKIP(IF b THEN P ELSE Q)  = gSKIP(P) ∧ gSKIP(Q)"
 "gSKIP(P ⟦ X ⟧ Q)  = gSKIP(P) ∨ gSKIP(Q)"
 "gSKIP(P \ X)      = False"
 "gSKIP(P[[r]])     = gSKIP(P)"
 "gSKIP(P ⨟ Q)      = gSKIP(P) ∨ gSKIP(Q)"
 "gSKIP(P ⌊ n)      = gSKIP(P) ∨ n = 0"
 "gSKIP($p)         = False"
```

Figure 21: The predicate gSKIP for guaranteeing that SKIP is guarded.

```
consts guarded :: "('p,'a) proc ⇒ bool"

primrec
 "guarded(STOP)       = True"
 "guarded(SKIP)       = True"
 "guarded(DIV)        = True"
 "guarded(a → P)      = noHide(P)"
 "guarded(? : X → P)  = (∀ a ∈ X. noHide(P(a)))"
 "guarded(P □ Q)      = guarded(P) ∧ guarded(Q)"
 "guarded(P ⊓ Q)      = guarded(P) ∧ guarded(Q)"
 "guarded(!! : C • P)  = (∀ c ∈ C. guarded(P(c)))"
 "guarded(IF b THEN P ELSE Q)  = guarded(P) ∧ guarded(Q)"
 "guarded(P ⟦ X ⟧ Q)  = guarded(P) ∧ guarded(Q)"
 "guarded(P \ X)      = noPN(P)"
 "guarded(P[[r]])     = guarded(P)"
 "guarded(P ⨟ Q)      = (guarded(P) ∧ gSKIP(P) ∧ noPN(Q)) ∨
                         (guarded(P) ∧ guarded(Q))"
 "guarded(P ⌊ n)      = guarded(P) ∨ n = 0"
 "guarded($p)         = False"
```

Figure 22: The predicate guarded for checking guardedness of processes.

Then, the predicate guarded for checking the guardedness is defined as shown in Figure 22. Note that guarded(P ⨟ Q) is true even if Q is not guarded, if SKIP in P is guarded (i.e. gSKIP(P)).

Finally, the predicate `guarded` is extended over functions as follows because we have to check the guardedness of `PNfun`:

```
consts
  guardedfun :: "('p ⇒ ('q,'a) proc) ⇒ bool"
defs
  guardedfun_def: "guardedfun(f) == (∀ p. guarded(f(p)))"
```

All these predicates are encoded in the theory file `CSP/CSP_syntax.thy` because they are independent of semantic models.