

# User Guide CSP-Prover Version 5.1

<p style="text-align: center;"><b>CSP-Prover Document</b> Version: DRAFT     January 6, 2010</p>
--

Yoshinao Isobe<sup>1</sup> and Markus Roggenbach<sup>2</sup>

<sup>1</sup> National Institute of Advanced Industrial Science and Technology, Japan,  
y-isobe@aist.go.jp,

<sup>2</sup> University of Wales Swansea, United Kingdom  
M.Roggenbach@swan.ac.uk

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Installing Isabelle2009-1</b>	<b>3</b>
<b>3</b>	<b>Setting up CSP-Prover</b>	<b>4</b>
<b>4</b>	<b>Starting CSP-Prover</b>	<b>5</b>
<b>5</b>	<b>Small demonstration</b>	<b>6</b>
<b>6</b>	<b>The CSP-dialect <math>\text{CSP}_{\text{TP}}</math></b>	<b>6</b>
6.1	Syntax . . . . .	6
6.2	Semantics . . . . .	10
<b>7</b>	<b>Encoding of the <math>\text{CSP}_{\text{TP}}</math></b>	<b>14</b>
7.1	Syntax . . . . .	15
7.2	Domain . . . . .	19
7.3	Semantics . . . . .	22
7.4	Recursive process . . . . .	26

<b>8 Verification</b>	<b>28</b>
8.1 Semantical proof . . . . .	28
8.2 Syntactical manual proof . . . . .	29
8.3 Syntactical semi-automatic proof . . . . .	42
<b>9 Conclusion</b>	<b>45</b>
<b>A Guarded function</b>	<b>46</b>

# 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 non-determinism. 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. 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 *methods* 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-5-1-2009-1.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. 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 Isabelle2009-1

CSP-Prover version 5.1 is encoded in Isabelle2009-1/HOL. To install the interactive theorem prover Isabelle follow the instructions of the Isabelle Web page:

`http://isabelle.in.tum.de/`

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

```
Isabelle2009-1.tar.gz
ProofGeneral-3.7.1.1.tar.gz
polym1-5.3.0.tar.gz
HOL_x86-linux.tar.gz
```

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

```
% tar -C /usr/local -xzf Isabelle2009-1.tar.gz
% tar -C /usr/local -xzf ProofGeneral-3.7.1.1.tar.gz
% tar -C /usr/local -xzf polym1-5.3.0.tar.gz
% tar -C /usr/local -xzf HOL_x86-linux.tar.gz
```

Isabelle can be started with plain tty interaction by

```
% /usr/local/Isabelle/bin/isabelle tty -l HOL
```

or with Proof General (Emacs interface wrapper) by

---

```
% /usr/local/Isabelle/bin/isabelle emacs -l HOL
```

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-5-1-2009-1.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-5-1-2009-1
```

by an unpacking command (e.g. `tar zxvf CSP-Prover-5-1-2009-1.tar.gz`).

`CSP-Prover-5-1-2009-1` contains the 9 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]),
- `NBuff` : linked  $n$  one-buffers,
- `UCD` : Uniform Candy Distribution Puzzle (see [IR08]).

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-5-1-2009-1/`, where the environment variable “`ISABELLE_bin`” has to be set to the path containing the command `isabelle` of Isabelle2009-1. The heap file will be made in your Isabelle directory. If you did not specify the directory, it is probably

---

```
~/isabelle/heap/Isabelle2009-1/polym1-*** (depends on your OS)
```

It may take time to make the five heap files. For example, about 8 minutes by Intel CPU (Core2, 2.5GHz).

In addition, if you like to comfortably read theory files of CSP-Prover by web-browsers (e.g. Firefox, ...), you can make html files for them by a command

```
% make_html
```

at the directory `/usr/local/CSP-Prover-5-1-2009-1/`. The theory files and theory dependency-graphs can be browsed by the web-browsers:

```
% cd ~/isabelle/browser_info/HOL/CSP
% firefox index.html
```

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

```
% cd ~/isabelle/browser_info/HOL/CSP/
% isabelle 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 tty -l CSP_F
```

or start it in Proof General[Asp00] by

```
% isabelle emacs -l CSP_F
```

It is recommended to use Proof General, which is a superior interface for Isabelle. In Proof General, you can also select a logic (e.g. `CSP`, `CSP_T`, `CSP_F`, `FN_F`, `HOL`, ...) used in Isabelle from the menu bar. Click the button [Isabelle]  $\rightarrow$  [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  $P \ \square \ Q$  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 tty -l CSP_F
```

2. Open the following example in the editor window:

```
/usr/local/CSP-Prover-5-1-2009-1/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 emacs -l CSP_F
```

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

```
/usr/local/CSP-Prover-5-1-2009-1/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-5-1-2009-1/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<sub>TP</sub>

This section summarises syntax and semantics of the CSP-dialect CSP<sub>TP</sub>, 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<sub>TP</sub> represents *Theorem Proving*.

### 6.1 Syntax

This section defines syntax of CSP<sub>TP</sub>: Given a type of process names  $\Pi$  and an alphabet of communications  $\Sigma$ . Figure 1 shows the syntax of processes in CSP<sub>TP</sub>, 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

$P ::=$	<b>SKIP</b>	%% successful terminating process
	<b>STOP</b>	%% deadlock process
	<b>DIV</b>	%% divergence
	$a \rightarrow P$	%% action prefix
	$? x : X \rightarrow P(x)$	%% prefix choice
	$P \square P$	%% external choice
	$P \sqcap P$	%% internal choice
	$!! c : C \bullet P(c)$	%% replicated internal choice
	<b>IF</b> $b$ <b>THEN</b> $P$ <b>ELSE</b> $P$	%% conditional
	$P \parallel X$	%% generalized parallel
	$P \setminus X$	%% hiding
	$P[[r]]$	%% relational renaming
	$P \circ P$	%% sequential composition
	$P \downarrow n$	%% depth restriction
	$\$p$	%% process name

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

Figure 1: Syntax of basic  $\text{CSP}_{\text{TP}}$  processes in CSP-Prover.

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

$$\begin{aligned} \text{!set } X : Xs \bullet P(X) &:= !! c : Xs \bullet P(c) \\ \text{!nat } n : N \bullet P(n) &:= !! c : N \bullet P(c) \end{aligned}$$

where  $Xs \subseteq \mathbb{P}(\Sigma)$  and  $N \subseteq \text{Nat}$ . This syntax is convenient in the theorem prover Isabelle because it is difficult to directly assign such two different types ( $\mathbb{P}(\Sigma)$  and  $\text{Nat}$ ) 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, generalised parallel, external choice, replicated internal choice, internal choice.

One difference from conventional CSP is that we replace the generic internal choice  $\sqcap P$  by a replicated internal choice  $!! c : C \bullet P(c)$ , i.e., instead of having internal choice over an arbitrary class of processes  $\mathcal{P} \subseteq \text{Proc}_{(\Pi, \Sigma)}$ , internal choice is restricted to run over an indexed set of processes  $P(\cdot) : \mathbb{P}(\Sigma) \uplus \text{Nat} \Rightarrow \text{Proc}_{(\Pi, \Sigma)}$  only, where  $C \in \text{Choice}(\Sigma)$ . The other difference is that we introduce depth-restriction  $\downarrow$  as a basic operator<sup>1</sup>. The restriction plays an important role in

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

full-normalisation. As [Ros98] shows, for the stable-failures model such restriction cannot be defined in terms of the other basic operators.

The following shortcuts are also available in CSP-Prover:

- (Untimed) timeout:

$$P \triangleright Q := (P \sqcap \text{STOP}) \sqcup Q$$

- Replicated internal choices:

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

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

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

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

- Internal prefix choice:

$$!x : A \rightarrow P(x) := !x : A \bullet (x \rightarrow P(x))$$

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

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

The prefix  $a!?x : X \rightarrow P(x)$  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 (loose) specifications in early design stages.

- If the index set in prefix choice, replicated internal choice, etc, is the universe, the universe can be omitted, for example we can write  $!\text{nat } n \bullet P(n)$  instead of  $!\text{nat } n : \text{Nat} \bullet P(n)$  and  $a?x \rightarrow P(x)$  instead of  $a?x : \text{Univ} \rightarrow P(x)$ .
- Interleaving, synchronous, and alphabetised parallels:

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



- Inductive alphabetised parallel:

$$\begin{aligned} [\llbracket \rrbracket] \langle \rangle &:= \text{SKIP} \\ [\llbracket \rrbracket] (P, X) \frown PX_{list} &:= P \llbracket X, Y \rrbracket ([\llbracket \rrbracket] PX_{list}) \end{aligned}$$

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

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

- Replicated alphabetised parallel:

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

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

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

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

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

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

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

$$p = P$$

where  $P \in \text{Proc}_{(\Pi, \Sigma)}$ . 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:

$$\begin{aligned} A &= a \rightarrow \$B \\ B &= (b \rightarrow \$A) \sqcap (c \rightarrow \text{SKIP}) \end{aligned}$$

$\text{CSP}_{\text{TP}}$  provides a special function  $\text{PNfun}_{\Pi} : \Pi \Rightarrow \text{Proc}_{(\Pi, \Sigma)}$ , 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  $\text{PNfun}_{\Pi}(p)$ . For example, the example  $A$  above is defined by the following function:

$$\begin{aligned} \text{PNfun}_{\Pi} A &= a \rightarrow \$B \\ \text{PNfun}_{\Pi} B &= (b \rightarrow \$A) \sqcap (c \rightarrow \text{SKIP}) \end{aligned}$$

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  $\text{PNfun}_\Pi$ :

$$\begin{aligned} \text{PNfun}_\Pi \quad & \text{Inc} = \text{start} \rightarrow \$(\text{Loop}(0)) \\ \text{PNfun}_\Pi \quad & (\text{Loop}(n)) = n \rightarrow \$(\text{Loop}(n+1)) \end{aligned}$$

where  $\Sigma = \{\text{start}\} \cup \text{Nat}$  and  $\Pi = \{\text{Int}\} \cup \{\text{Loop}(n) \mid n \in \text{Nat}\}$ . 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  $f : \Pi_1 \Rightarrow \text{Proc}_{(\Pi_2, \Sigma)}$ . Therefore, we define the operation

$$- \triangleleft - : \text{Proc}_{(\Pi_1, \Sigma)} \Rightarrow (\Pi_1 \Rightarrow \text{Proc}_{(\Pi_2, \Sigma)}) \Rightarrow \text{Proc}_{(\Pi_2, \Sigma)}$$

for the replacement. Thus,  $P \triangleleft f$  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:

$$\begin{aligned} (\text{Inc} \triangleleft \text{PNfun}_\Pi) \triangleleft \text{PNfun}_\Pi &= (\text{start} \rightarrow \text{Loop}(0)) \triangleleft \text{PNfun}_\Pi \\ &= \text{start} \rightarrow 0 \rightarrow \text{Loop}(1) \end{aligned}$$

This substitution operator is extended over functions as follows:

$$\begin{aligned} - \triangleleft - & : (\Pi_1 \Rightarrow \text{Proc}_{(\Pi_2, \Sigma)}) \Rightarrow (\Pi_2 \Rightarrow \text{Proc}_{(\Pi_3, \Sigma)}) \Rightarrow (\Pi_1 \Rightarrow \text{Proc}_{(\Pi_3, \Sigma)}) \\ f \triangleleft g &= (\lambda p. f(p) \triangleleft g) \end{aligned}$$

## 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  $\Sigma$ , 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  $F \subseteq \Sigma^{*\checkmark} \times \mathbb{P}(\Sigma^\checkmark)$ <sup>2</sup>.*

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

**T2**  $(t, X) \in F \implies t \in T$ ,

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

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

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

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

---

<sup>2</sup> $\Sigma^\checkmark := \Sigma \cup \{\checkmark\}$ ,  $\Sigma^{*\checkmark} := \Sigma^* \cup \{t \frown \langle \checkmark \rangle \mid t \in \Sigma^*\}$ .

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

When  $(\Pi, \Sigma)$ -model  $M$  is given, the semantics parameterised by  $M$  of a process  $P$  is defined by  $\llbracket P \rrbracket_{\mathcal{F}(M)}$ , 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  $\llbracket \cdot \rrbracket_{\mathcal{F}(M)}$  is a map  $(Proc_{(\Pi, \Sigma)} \Rightarrow \mathcal{F}_\Sigma)$  expressed with the help of two functions:

$$\llbracket P \rrbracket_{\mathcal{F}(M)} = (\text{traces}_{(\mathbf{fst} \circ M)}(P), \text{failures}_M(P)).$$

where  $\mathbf{fst}$  is a function for extracting the first component from a pair and  $\circ$  is the composition operator of two functions, thus  $(\mathbf{fst} \circ M)$  is the  $(\Pi, \Sigma)$ -model for giving a meaning to each process name in the traces model  $\mathcal{T}$  (i.e.  $(\mathbf{fst} \circ M) : \Pi \Rightarrow \mathcal{T}_\Sigma$ ), obtained from  $M$ . Then, the functions  $\text{traces}_M$  and  $\text{failures}_M$  are recursively defined by the semantic clauses given in Figure 2. Our definitions of  $\text{traces}_M$  and  $\text{failures}_M$  are identical to those given in [Ros98] except that the  $(\Pi, \Sigma)$ -models  $M$  are explicitly attached and the clauses of our two operators, namely replicated internal choice<sup>3</sup> and depth restriction are added. The auxiliary notations  $t_1 \llbracket X \rrbracket t_2$ ,  $t \setminus X$ ,  $\llbracket [r] \rrbracket^*$ ,  $\llbracket [r] \rrbracket^{-1}$ ,  $T \downarrow n$ , and  $F \downarrow n$  used in Figure 2 are defined as follows:

- $t_1 \llbracket X \rrbracket t_2$  is inductively defined by:

$$\begin{aligned} \langle x \rangle \cap t_1 \llbracket X \rrbracket \langle x \rangle \cap t_2 &= \{ \langle x \rangle \cap u \mid u \in t_1 \llbracket X \rrbracket t_2 \} \\ \langle x \rangle \cap t_1 \llbracket X \rrbracket \langle x' \rangle \cap t_2 &= \emptyset \\ \langle x \rangle \cap t_1 \llbracket X \rrbracket \langle \rangle &= \emptyset \\ \langle \rangle \llbracket X \rrbracket \langle x \rangle \cap t_2 &= \emptyset \\ \langle \rangle \llbracket X \rrbracket \langle \rangle &= \{ \langle \rangle \} \\ \langle y \rangle \cap t_1 \llbracket X \rrbracket \langle x \rangle \cap t_2 &= \{ \langle y \rangle \cap u \mid u \in t_1 \llbracket X \rrbracket \langle x \rangle \cap t_2 \} \\ \langle y \rangle \cap t_1 \llbracket X \rrbracket \langle \rangle &= \{ \langle y \rangle \cap u \mid u \in t_1 \llbracket X \rrbracket \langle \rangle \} \\ \langle x \rangle \cap t_1 \llbracket X \rrbracket \langle y \rangle \cap t_2 &= \{ \langle y \rangle \cap u \mid u \in \langle x \rangle \cap t_1 \llbracket X \rrbracket t_2 \} \\ \langle \rangle \llbracket X \rrbracket \langle y \rangle \cap t_2 &= \{ \langle y \rangle \cap u \mid u \in \langle \rangle \llbracket X \rrbracket t_2 \} \\ \langle y \rangle \cap t_1 \llbracket X \rrbracket \langle y' \rangle \cap t_2 &= \{ \langle y \rangle \cap u \mid u \in t_1 \llbracket X \rrbracket \langle y' \rangle \cap t_2 \} \\ &\quad \cup \{ \langle y' \rangle \cap u \mid u \in \langle y \rangle \cap t_1 \llbracket X \rrbracket t_2 \} \end{aligned}$$

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

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

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

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

$$\begin{aligned} \text{True} &\Rightarrow (\langle \rangle, \langle \rangle) \in \llbracket [r] \rrbracket^* \\ \text{True} &\Rightarrow (\langle \checkmark \rangle, \langle \checkmark \rangle) \in \llbracket [r] \rrbracket^* \\ (a, b) \in r \wedge (t, t') \in \llbracket [r] \rrbracket^* &\Rightarrow (a \cap t, b \cap t') \in \llbracket [r] \rrbracket^* \end{aligned}$$

<sup>3</sup>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} \text{traces}_M(\text{SKIP}) &= \{\langle \rangle, \langle \checkmark \rangle\} \\ \text{traces}_M(\text{STOP}) &= \{\langle \rangle\} \\ \text{traces}_M(\text{DIV}) &= \{\langle \rangle\} \\ \text{traces}_M(a \rightarrow P) &= \{\langle \rangle\} \cup \{\langle a \rangle \frown t' \mid t' \in \text{traces}_M(P)\} \\ \text{traces}_M(? x : A \rightarrow P(x)) &= \{\langle \rangle\} \cup \{\langle x \rangle \frown t' \mid t' \in \text{traces}_M(P(x)), x \in A\} \\ \text{traces}_M(P \sqcap Q) &= \text{traces}_M(P) \cup \text{traces}_M(Q) \\ \text{traces}_M(P \sqcap Q) &= \text{traces}_M(P) \cup \text{traces}_M(Q) \\ \text{traces}_M(!! c : C \bullet P(c)) &= \bigcup \{\text{traces}_M(P(c)) \mid c \in C\} \cup \{\langle \rangle\} \\ \text{traces}_M(\text{IF } b \text{ THEN } P \text{ ELSE } Q) &= \text{if } b \text{ then } \text{traces}_M(P) \text{ else } \text{traces}_M(Q) \\ \text{traces}_M(P \parallel X \parallel Q) &= \bigcup \{t_1 \parallel X \parallel t_2 \mid t_1 \in \text{traces}_M(P), t_2 \in \text{traces}_M(Q)\} \\ \text{traces}_M(P \setminus X) &= \{t \setminus X \mid t \in \text{traces}_M(P)\} \\ \text{traces}_M(P[[r]]) &= \{t \mid \exists t' \in \text{traces}_M(P). (t', t) \in [[r]]^*\} \\ \text{traces}_M(P \circledast Q) &= (\text{traces}_M(P) \cap \Sigma^*) \\ &\quad \cup \{t_1 \frown t_2 \mid t_1 \frown \langle \checkmark \rangle \in \text{traces}_M(P), t_2 \in \text{traces}_M(Q)\} \\ \text{traces}_M(P \downarrow n) &= \text{traces}_M(P) \downarrow n \\ \text{traces}_M(\$p) &= M(p) \end{aligned} $	
$ \begin{aligned} \text{failures}_M(\text{SKIP}) &= \{(\langle \rangle, X) \mid X \subseteq \Sigma\} \cup \{(\langle \checkmark \rangle, X) \mid X \subseteq \Sigma^\vee\} \\ \text{failures}_M(\text{STOP}) &= \{(\langle \rangle, X) \mid X \subseteq \Sigma^\vee\} \\ \text{failures}_M(\text{DIV}) &= \emptyset \\ \text{failures}_M(a \rightarrow P) &= \{(\langle \rangle, X) \mid a \notin X\} \\ &\quad \cup \{(\langle a \rangle \frown t', X) \mid (t', X) \in \text{failures}_M(P)\} \\ \text{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 \text{failures}_M(P(x)), x \in A\} \\ \text{failures}_M(P \sqcap Q) &= \{(\langle \rangle, X) \mid (\langle \rangle, X) \in \text{failures}_M(P) \cap \text{failures}_M(Q)\} \\ &\quad \cup \{(t, X) \mid t \neq \langle \rangle, \\ &\quad \quad (t, X) \in \text{failures}_M(P) \cup \text{failures}_M(Q)\} \\ &\quad \cup \{(\langle \rangle, X) \mid X \subseteq \Sigma, \\ &\quad \quad \langle \checkmark \rangle \in \text{traces}_{(\text{fst} \circ M)}(P) \cup \text{traces}_{(\text{fst} \circ M)}(Q)\} \\ \text{failures}_M(P \sqcap Q) &= \text{failures}_M(P) \cup \text{failures}_M(Q) \\ \text{failures}_M(!! c : C \bullet P) &= \bigcup \{\text{failures}_M(P(c)) \mid c \in C\} \\ \text{failures}_M(\text{IF } b \text{ THEN } P \text{ ELSE } Q) &= \text{if } b \text{ then } \text{failures}_M(P) \text{ else } \text{failures}_M(Q) \\ \text{failures}_M(P \parallel X \parallel Q) &= \{(u, Y \cup Z) \mid Y - (X \cup \{\checkmark\}) = Z - (X \cup \{\checkmark\}), \\ &\quad \exists t_1, t_2. u \in t_1 \parallel X \parallel t_2, \\ &\quad (t_1, Y) \in \text{failures}_M(P), (t_2, Z) \in \text{failures}_M(Q)\} \\ \text{failures}_M(P \setminus X) &= \{(t \setminus X, Y) \mid (t, Y \cup X) \in \text{failures}_M(P)\} \\ \text{failures}_M(P[[r]]) &= \{(t, X) \mid \exists t'. (t', t) \in [[r]]^*, \\ &\quad (t', [[r]]^{-1}(X)) \in \text{failures}_M(P)\} \\ \text{failures}_M(P \circledast Q) &= \{(t_1, X) \mid t_1 \in \Sigma^*, (t_1, X \cup \{\checkmark\}) \in \text{failures}_M(P)\} \\ &\quad \cup \{(t_1 \frown t_2, X) \mid t_1 \frown \langle \checkmark \rangle \in \text{traces}_{(\text{fst} \circ M)}(P), \\ &\quad (t_2, X) \in \text{failures}_M(Q)\} \\ \text{failures}_M(P \downarrow n) &= \text{failures}_M(P) \downarrow n \\ \text{failures}_M(\$p) &= \text{snd}(M(p)) \end{aligned} $	

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

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

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

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

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

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

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

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

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

Since the definition of process names depends on which approach is used,  $\text{CSP}_{\text{TP}}$  has a reserved word **FPmode**, which takes either **CMSmode** or **CP0mode** or **MIXmode**. If **FPmode** = **CMSmode**, then the unique fixed point of  $[\text{PNfun}_\Pi]_{\mathcal{F}}^{\text{fun}}$  is used, but it cannot deal with unguarded processes. If **FPmode** = **CP0mode**, then the least fixed point of  $[\text{PNfun}_\Pi]_{\mathcal{F}}^{\text{fun}}$  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  $[\text{PNfun}_\Pi]_{\mathcal{F}}^{\text{fun}}$  is used for all processes, but the uniqueness is guaranteed for guarded processes. Therefore, the ideal  $(\Pi, \Sigma)$ -model, written  $\text{MF}_\Pi$ , is given as follows:

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

where the function **FIX** is defined as follows:

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

<sup>4</sup>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  $\llbracket P \rrbracket_{\mathcal{F}}$  of each process  $P$  is defined as follows:

$$\llbracket P \rrbracket_{\mathcal{F}} = \llbracket P \rrbracket_{\mathcal{F}(\mathbf{MF}_{\Pi})}$$

Given two models  $M_1$  and  $M_2$  for two sets of process names  $\Pi_1$  and  $\Pi_2$  respectively, parameterised process equivalence  $=_{\mathcal{F}(M_1, M_2)}$  and parameterised process refinement  $\sqsubseteq_{\mathcal{F}(M_1, M_2)}$  over the stable failures model are then defined as follows:

$$\begin{aligned} P =_{\mathcal{F}(M_1, M_2)} Q &\Leftrightarrow \\ &\text{traces}_{(\mathbf{fst} \circ M_1)}(P) = \text{traces}_{(\mathbf{fst} \circ M_2)}(Q) \wedge \text{failures}_{M_1}(P) = \text{failures}_{M_2}(Q), \\ P \sqsubseteq_{\mathcal{F}(M_1, M_2)} Q &\Leftrightarrow \\ &\text{traces}_{(\mathbf{fst} \circ M_1)}(P) \supseteq \text{traces}_{(\mathbf{fst} \circ M_2)}(Q) \wedge \text{failures}_{M_1}(P) \supseteq \text{failures}_{M_2}(Q). \end{aligned}$$

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

$$\begin{aligned} P =_{\mathcal{F}} Q &\Leftrightarrow P =_{\mathcal{F}(\mathbf{MF}_{\Pi_1}, \mathbf{MF}_{\Pi_2})} Q, \\ P \sqsubseteq_{\mathcal{F}} Q &\Leftrightarrow P \sqsubseteq_{\mathcal{F}(\mathbf{MF}_{\Pi_1}, \mathbf{MF}_{\Pi_2})} Q. \end{aligned}$$

Then, as expected, the following properties hold:

1. Let  $\mathbf{FPmode} = \mathbf{CP0mode}$ . Then,

- $\forall p \in \Pi. \$p =_{\mathcal{F}} \mathbf{PNfun}_{\Pi}(p)$ ,
- $\forall f. ((\forall p \in \Pi. f(p) =_{\mathcal{F}} \mathbf{PNfun}_{\Pi}(p) \triangleleft f) \implies (\forall p \in \Pi. f(p) \sqsubseteq_{\mathcal{F}} \$p))$ .

2. Let  $\mathbf{FPmode} = \mathbf{CMSmode}$  and  $\mathbf{PNfun}_{\Pi}(p)$  be guarded for any  $p$ . Then,

- $\forall p \in \Pi. \$p =_{\mathcal{F}} \mathbf{PNfun}_{\Pi}(p)$ ,
- $\forall f. ((\forall p \in \Pi. f(p) =_{\mathcal{F}} \mathbf{PNfun}_{\Pi}(p) \triangleleft f) \implies (\forall p \in \Pi. f(p) =_{\mathcal{F}} \$p))$ .

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  $\mathbf{CSP}_{\mathbf{TP}}$ .

## 7 Encoding of the $\mathbf{CSP}_{\mathbf{TP}}$

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

Conventional symbol	ASCII symbol	Name
<b>SKIP</b>	<b>SKIP</b>	successful terminating process
<b>STOP</b>	<b>STOP</b>	deadlock process
<b>DIV</b>	<b>DIV</b>	divergence
$a \rightarrow P$	$a \rightarrow P$	action prefix
$? x : A \rightarrow P(x)$	$? x : A \rightarrow P(x)$	prefix choice
$P \square Q$	$P \text{ [ + ] } Q$	external choice
$P \sqcap Q$	$P \text{   } \sim \text{   } Q$	internal choice
$!! c : C \bullet P(c)$	$!! c : C \dots P(c)$	replicated internal choice
<b>IF</b> $b$ <b>THEN</b> $P$ <b>ELSE</b> $Q$	<b>IF</b> $b$ <b>THEN</b> $P$ <b>ELSE</b> $Q$	conditional
$P \parallel X \parallel Q$	$P \text{   } [X] \text{   } Q$	generalized parallel
$P \setminus X$	$P \text{ -- } X$	hiding
$P[[r]]$	$P \text{ [ [ } r \text{ ] ] }$	relational renaming
$P \circledast Q$	$P \text{ ; ; } Q$	sequential composition
$P \lfloor n$	$P \text{   } . n$	depth restriction
$\$p$	$\$p$	process name
$P \triangleright Q$	$P \text{ [ > } Q$	timeout
$!\text{set } X : Xs \bullet P(X)$	$!\text{set } X : \mathcal{X}f \dots P(X)$	replicated internal choice over $\mathbb{P}(\Sigma)$
$!\text{nat } n : N \bullet P(n)$	$!\text{nat } n : N \dots P(n)$	replicated internal choice over $Nat$
$! x : A \bullet P(x)$	$! x : A \dots P(x)$	replicated internal choice over $\Sigma$
$!\langle f \rangle z : Z \bullet P(z)$	$!\langle f \rangle z : Z \dots P(z)$	replicated internal choice with $f$
$! x : A \rightarrow P(x)$	$! x : A \rightarrow P(x)$	internal prefix choice
$a!v \rightarrow P$	$a!v \rightarrow P$	sending
$a?x : X \rightarrow P(x)$	$a?x : X \rightarrow P(x)$	receiving
$a!?x : X \rightarrow P(x)$	$a!?x : X \rightarrow P(x)$	non-deterministic sending
$P \parallel\!\!\!\parallel Q$	$P \text{       } Q$	interleaving
$P \parallel Q$	$P \text{     } Q$	synchronous
$P \parallel [X, Y] \parallel Q$	$P \text{   } [X, Y] \text{   } Q$	alphabetized parallel
$[[\text{ }]] i : I \bullet (P_i, X_i)$	$[[\text{ }]] i : I \dots (P_i, X_i)$	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<sup>5</sup>. 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  $\square$  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

<sup>5</sup>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"
    "('p, 'a) proc" ("_ \ _")
  | Renaming        "('p, 'a) proc" "('a * 'a) set"
    "('p, 'a) proc" ("_ [[ _ ] _]")
  | Seq_compo       "('p, 'a) proc" "('p, 'a) proc"
    "('p, 'a) proc" ("_ § _")
  | Depth_rest      "('p, 'a) proc" "nat"
    "('p, 'a) proc" ("_ ⊥ _")
  | Proc_name       "'p"
    ("$_")

```

Figure 4: The recursive definition of the process type.

$X \leq Y$ , respectively.

Now, the set of (basic) processes is given as a recursive type  $(\text{'p}, \text{'a}) \text{ proc}$  which is defined by the Isabelle command **datatype** as shown in Figure 4, where  $\text{'p}$  is the type of process names  $\Pi$  and  $\text{'a}$  is the type of communications  $\Sigma$ . Here, note that the types of index sets  $\text{'a sets\_nats}$  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  $\text{'a set set}$  and  $\text{nat set}$ . Therefore, it is defined by  $(\text{'a set set, nat set}) \text{ sum}$  as shown at the top of Figure 4, where **sum** is the *disjoint sum type* and is defined by

$$\text{datatype } (\text{'a}, \text{'b}) \text{ sum} = \text{type1 } \text{'a} \mid \text{type2 } \text{'b}$$

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,

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



```

syntax
"@Ext_pre_choice" ::
  "pttrn  $\Rightarrow$  'a set  $\Rightarrow$  'a proc  $\Rightarrow$  ('p,'a) proc" ("?_:_  $\rightarrow$  _")

"@Rep_int_choice" ::
  "pttrn  $\Rightarrow$  'a sets_nats  $\Rightarrow$  'a aset_anat  $\Rightarrow$  ('p,'a) proc" ("!!_:_  $\bullet$  _")

translations
"? x : X  $\rightarrow$  P" == "? : X  $\rightarrow$  ( $\lambda$  x. P)"
"!! c : C  $\bullet$  P" == "!! : C  $\bullet$  ( $\lambda$  c. P)"

```

Figure 5: The expression of bound variables.

is defined by

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

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 **translations**, which make syntactic sugars, as shown in Figure 5.

Derived operators such as  $\triangleright$  are defined by **abbreviation** for syntactic macros, **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

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

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:

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

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

As explained in Subsection 6.1,  $\text{CSP}_{\text{TP}}$  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  $\Rightarrow$  ('p,'a) proc"
datatype fpmode = CP0mode | 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
abbreviation
Timeout_abb :: "('p,'a) proc  $\Rightarrow$  ('p,'a) proc  $\Rightarrow$  ('p,'a) proc" ("_▷_")
where
"P▷Q" == "(P □ STOP) □ Q"

%% replicated internal choice over sets
consts
Rep_int_choice_set ::
  "'a set set  $\Rightarrow$  ('a set  $\Rightarrow$  ('p,'a) proc)  $\Rightarrow$  ('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  $\Rightarrow$  ('a  $\Rightarrow$  ('p,'a) proc)  $\Rightarrow$  ('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.

[**version 5**]: In CSP-Prover version 5, the following convenient short notations are introduced for renaming :

- $P[[a \longleftrightarrow b]]$  (written  $P[[a <--> b]]$  in ASCII) is the process obtained by swapping the events  $a$  and  $b$  in the process  $P$ .
- $P[[c \Longleftrightarrow d]]$  (written  $P[[c <==> d]]$  in ASCII) is the process obtained by swapping the channels  $c$  and  $d$  in the process  $P$ .
- $P[[a \leftarrow b]]$  (written  $P[[a <-- b]]$  in ASCII) is the process obtained by replacing the event  $a$  by  $b$  in the process  $P$ .
- $P[[c \Leftarrow d]]$  (written  $P[[c <== d]]$  in ASCII) is the process obtained by replacing the channel  $c$  by  $d$  in the process  $P$ .
- $P[[A \leftarrow b]]$  (written  $P[[A <<- b]]$  in ASCII) is the process obtained by replacing all the events in the set  $A$  by  $b$  in the process  $P$ .

For example, the following equations hold:

$$\begin{aligned}
(a \rightarrow b \rightarrow c \rightarrow \text{SKIP})[[a \longleftrightarrow b]] &=_{\mathcal{F}} b \rightarrow a \rightarrow c \rightarrow \text{SKIP} \\
(in?x \rightarrow out!x \rightarrow \text{SKIP})[[out \Longleftrightarrow mid]] &=_{\mathcal{F}} in?x \rightarrow mid!x \rightarrow \text{SKIP} \\
(a \rightarrow b \rightarrow c \rightarrow \text{SKIP})[[a \leftarrow b]] &=_{\mathcal{F}} b \rightarrow b \rightarrow c \rightarrow \text{SKIP} \\
(a \rightarrow b \rightarrow c \rightarrow d \rightarrow \text{SKIP})[[\{a, c\} \leftarrow b]] &=_{\mathcal{F}} b \rightarrow b \rightarrow b \rightarrow d \rightarrow \text{SKIP}
\end{aligned}$$

Auxiliary laws for the short notations are also proven in CSP-Prover version 5, and they are automatically applied by the CSP-methods `cspF_auto`, `cspF_hsf`, `cspF_renaming`, etc as explained in Subsection 8.3, later. It will make proofs easy to read. The short notations for renaming are used in the examples `NBuff` and `UCD` in `CSP-Prover-5-1-2009-1`.

## 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.

$$\begin{aligned} \text{SubType} &:: \text{SuperType set}, \\ \text{Rep\_SubType} &:: \text{SuperType} \Rightarrow \text{SubType}, \\ \text{Abs\_SubType} &:: \text{SubType} \Rightarrow \text{SuperType}. \end{aligned}$$

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

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

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

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

```
datatype 'a event = Ev 'a |  $\checkmark$ 
```

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

```
typedef 'a trace = "{s::('a event list).  $\checkmark \notin \text{set } (\text{butlast } s)}$ }"
```

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  $\hat{\phantom{x}}$  (written `^^` in ASCII)<sup>6</sup> over traces is defined from the concatenate operator  $@$  over lists as follows:

```
consts
  appt :: "'a trace  $\Rightarrow$  'a trace  $\Rightarrow$  'a trace" (infixr " $\hat{\phantom{x}}$ " 65)
defs
  appt_def : " $s \hat{\phantom{x}} t == \text{Abs\_trace } (\text{Rep\_trace } s @ \text{Rep\_trace } t)$ "
```

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 : " $T \leq S == (\text{Rep\_domT } T) \subseteq (\text{Rep\_domT } S)$ "
```

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 : " $F \leq E == (\text{Rep\_setF } F) \subseteq (\text{Rep\_setF } E)$ "
```

<sup>6</sup>**NOTE:** `^^` was used as the concatenate operator in `CSP-Prover` version 5.0, but it is newly used for expressing power functions in Isabelle 2009-1. Therefore, it is replaced by `^^^` in `CSP-Prover` version 5.1.

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: `fst(x, y) = x` and `snd(x, y) = y`. The subscripts `t` and `f` are attached to operators on traces and failures, respectively, e.g. `∈t` and `∈f`. The condition `noTick t` means that the trace  $t$  does not contain  $\checkmark$ . This condition is not explicitly written in the definition of **T3** shown in Subsection 6.2 because  $t \wedge \langle \checkmark \rangle$  implicitly means that  $t$  has no  $\checkmark$ . On the other hand,  $\wedge$  is a total function<sup>7</sup> because Isabelle does not allow us to define truly partial functions. Therefore, the condition `noTick t` is necessary. See the theory-file `Domain_F.thy` in the package `CSP_F` for more details.

The partial order over `domF` is defined as the combination of the partial orders `≤ 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 `≤ over domF` is a partial order indeed. That means `(domF, ≤)` can be proven to be an instance of the type class of partial ordered set as follows:

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

<sup>7</sup>For example,  $\langle \checkmark \rangle \wedge \langle \checkmark \rangle$  is meaningless and cannot be interpreted to  $\langle \checkmark, \checkmark \rangle$ , but such application  $\langle \checkmark \rangle \wedge \langle \checkmark \rangle$  of  $\wedge$  is not forbidden.

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  $\Rightarrow$  'a setF  $\Rightarrow$  'a domF" ("(_ , , _)")
  fstF  :: "'a domF  $\Rightarrow$  'a domT"
  sndF  :: "'a domF  $\Rightarrow$  '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  $\llbracket X \rrbracket_{\text{tr}}$  and  $\backslash_{\text{tr}}$  (see Subsection 6.2 for the definitions) over traces are given in `Trace_par.thy` and `Trace_hide.thy` respectively, and the encodings of  $\llbracket [r] \rrbracket^*$  and  $\llbracket [r] \rrbracket^{\text{inv}}$  are given in `Trace_ren.thy` in the package `CSP`. And,  $(\text{rmTick } s)$  is the trace obtained by removing  $\checkmark$  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 `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  $\Rightarrow$  ('a, 'b) sum set"

primrec
  "sumset (type1 X) = {type1 a | a. a  $\in$  X}"
  "sumset (type2 X) = {type2 a | a. a  $\in$  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 `CSPTP` does not allow it.

```

consts traces :: "('p,'a) proc  $\Rightarrow$  ('p  $\Rightarrow$  'a domT)  $\Rightarrow$  'a domT"

primrec
  "traces(STOP)      = ( $\lambda M. \{\langle \rangle\}_t$ )"
  "traces(SKIP)      = ( $\lambda M. \{\langle \rangle, \langle \checkmark \rangle\}_t$ )"
  "traces(DIV)       = ( $\lambda M. \{\langle \rangle\}_t$ )"
  "traces( $a \rightarrow P$ )   = ( $\lambda M. \{t. t = \langle \rangle \vee$ 
    ( $\exists s. t = \langle \text{Ev } a \rangle \wedge s \wedge s \in_t \text{traces}(P) M\}_t$ )"
  "traces( $? : X \rightarrow P$ ) = ( $\lambda M. \{t. t = \langle \rangle \vee$ 
    ( $\exists a s. t = \langle \text{Ev } a \rangle \wedge s \wedge s \in_t \text{traces}(P a) M \wedge$ 
     $a \in X\}_t$ )"
  "traces( $P \sqcap Q$ )    = ( $\lambda M. \text{traces}(P) M \cup_t \text{traces}(Q) M$ )"
  "traces( $P \sqcap Q$ )    = ( $\lambda M. \text{traces}(P) M \cup_t \text{traces}(Q) M$ )"
  "traces( $!! : C \bullet P$ ) = ( $\lambda M. \{t. t = \langle \rangle \vee$ 
    ( $\exists c \in \text{sumset}(C). t \in_t \text{traces}(P c) M\}_t$ )"
  "traces(IF  $b$  THEN  $P$  ELSE  $Q$ ) = ( $\lambda M. ($ 
    ( $\text{if } b \text{ then traces}(P) M \text{ else traces}(Q) M$ ))"
  "traces( $P \llbracket X \rrbracket Q$ ) = ( $\lambda M. \{u. \exists s t. u \in s \llbracket X \rrbracket_{\text{tr}} t \wedge$ 
     $s \in_t \text{traces}(P) M \wedge t \in_t \text{traces}(Q) M\}_t$ )"
  "traces( $P \setminus X$ )   = ( $\lambda M. \{t. \exists s. t = s \setminus_{\text{tr}} X \wedge s \in_t \text{traces}(P) M\}_t$ )"
  "traces( $P[[r]]$ )     = ( $\lambda M. \{t. \exists s. s[[r]]^* t \wedge s \in_t \text{traces}(P) M\}_t$ )"
  "traces( $P \circledast Q$ )   = ( $\lambda M. \{u. (\exists s. u = \text{rmTick } s \wedge s \in_t \text{traces}(P) M) \vee$ 
    ( $\exists s t. u = s \frown t \wedge s \frown \langle \checkmark \rangle \in_t \text{traces}(P) M \wedge$ 
     $t \in_t \text{traces}(Q) M \wedge \text{noTick } s\}_t$ )"
  "traces( $P \lfloor n$ )      = ( $\lambda M. \text{traces}(P) M \downarrow n$ )"
  "traces( $\$p$ )         = ( $\lambda M. M(p)$ )"

```

Figure 7: The encoding of the function *traces*

As explained in Subsection 6.2, the parameterised semantics  $\llbracket P \rrbracket_{\mathcal{F}(M)}$  of each process  $P$  with respect to the  $(\Pi, \Sigma)$ -model  $M$  and  $\llbracket f \rrbracket_{\mathcal{F}}^{fun}$  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  $\Rightarrow$  ('p  $\Rightarrow$  'a domF)  $\Rightarrow$  'a domF"   ("⟦_⟧Ff")
  semFfun  :: "('p  $\Rightarrow$  ('p,'a) proc)  $\Rightarrow$  ('p  $\Rightarrow$  'a domF)  $\Rightarrow$  'a domF" ("⟦_⟧Ffun")

defs
  semFf_def  : "⟦P⟧Ff == ( $\lambda M. (\text{traces}(P) M, \text{failures}(P) M)$ )"
  semFfun_def: "⟦f⟧Ffun == ( $\lambda M. \lambda p. \llbracket f(p) \rrbracket_{\mathcal{F}} Ff M$ )"

```

Then, the ideal  $(\Pi, \Sigma)$ -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  $(\text{traces}(P) (\text{fstF} \circ M), \text{failures}(P) M)$  is in

```

consts failures :: "('p,'a) proc  $\Rightarrow$  ('p  $\Rightarrow$  'a domF)  $\Rightarrow$  'a setF"

primrec
"failures(STOP)      = ( $\lambda M. \{f. \exists X. f = (\langle \rangle, X) \}_{\mathfrak{f}}$ )"
"failures(SKIP)      = ( $\lambda M. \{f. (\exists X. f = (\langle \rangle, X) \wedge X \subseteq \text{Evset}) \vee$ 
    ( $\exists X. f = (\langle \checkmark \rangle, X) \}_{\mathfrak{f}}$ )"
"failures(DIV)       = ( $\lambda M. \{ \}_{\mathfrak{f}}$ )"
"failures( $a \rightarrow P$ )    = ( $\lambda M. \{f. (\exists X. f = (\langle \rangle, X) \wedge \text{Ev } a \notin X) \vee$ 
    ( $\exists s X. f = (\langle \text{Ev } a \rangle \frown s, X) \wedge$ 
    ( $s, X \in \text{failures}(P) M \}_{\mathfrak{f}}$ )"
"failures( $? : X \rightarrow P$ ) = ( $\lambda M. \{f. (\exists Y. f = (\langle \rangle, Y) \wedge (\text{Ev}' X) \cap Y = \{\}) \vee$ 
    ( $\exists a s Y. f = (\langle \text{Ev } a \rangle \frown s, X) \wedge$ 
    ( $s, X \in \text{failures}(P a) M \wedge a \in X \}_{\mathfrak{f}}$ )"
"failures( $P \sqcap Q$ )    = ( $\lambda M. \{f. (\exists X. f = (\langle \rangle, X) \wedge$ 
    ( $f \in_{\mathfrak{f}} \text{failures}(P) M \cap_{\mathfrak{f}} \text{failures}(Q) M \vee$ 
    ( $\exists s X. f = (s, X) \wedge s \neq \langle \rangle \wedge$ 
    ( $f \in_{\mathfrak{f}} \text{failures}(P) M \cup_{\mathfrak{f}} \text{failures}(Q) M \vee$ 
    ( $\exists X. f = (\langle \rangle, X) \wedge X \subseteq \text{Evset} \wedge$ 
    ( $\langle \checkmark \rangle \in_{\mathfrak{t}} \text{traces}(P) (\text{fstF} \circ M) \cup_{\mathfrak{t}}$ 
    ( $\text{traces}(Q) (\text{fstF} \circ M) \}_{\mathfrak{f}}$ )"
"failures( $P \sqcap Q$ )    = ( $\lambda M. \text{failures}(P) M \cup_{\mathfrak{f}} \text{failures}(Q) M$ )"
"failures( $!! : C \bullet P$ ) = ( $\lambda M. \{f. (\exists c \in \text{sumset}(C). f \in_{\mathfrak{f}} \text{failures}(P c) M \}_{\mathfrak{f}}$ )"
"failures(IF  $b$  THEN  $P$  ELSE  $Q$ ) = ( $\lambda M. (\text{if } b \text{ then failures}(P) M \text{ else failures}(Q) M)$ )"
"failures( $P \parallel X \parallel Q$ ) = ( $\lambda M. \{f. \exists u Y Z. f = (u, Y \cup Z) \wedge$ 
    ( $Y - ((\text{Ev}' X) \cup \{\checkmark\}) = Z - ((\text{Ev}' X) \cup \{\checkmark\}) \wedge$ 
    ( $\exists s t. u \in s \parallel X \parallel_{\text{tr}} t \wedge (s, Y) \in_{\mathfrak{f}} \text{failures}(P) M \wedge$ 
    ( $(t, Z) \in_{\mathfrak{f}} \text{failures}(Q) M \}_{\mathfrak{f}}$ )"
"failures( $P \setminus X$ )    = ( $\lambda M. \{f. \exists s Y. f = (s \setminus_{\text{tr}} X, Y) \wedge$ 
    ( $s, (\text{Ev}' X) \cup Y \in_{\mathfrak{f}} \text{failures}(P) M \}_{\mathfrak{f}}$ )"
"failures( $P \llbracket r \rrbracket$ )    = ( $\lambda M. \{f. \exists s t X. f = (t, X) \wedge s \llbracket r \rrbracket^* t \wedge$ 
    ( $s, \llbracket r \rrbracket^{\text{inv}} X \in_{\mathfrak{f}} \text{failures}(P) M \}_{\mathfrak{f}}$ )"
"failures( $P \circledcirc Q$ )    = ( $\lambda M. \{f. (\exists t X. f = (t, X) \wedge$ 
    ( $(t, X \cup \{\checkmark\}) \in_{\mathfrak{f}} \text{failures}(P) M \wedge \text{noTick } t \vee$ 
    ( $\exists s t X. f = (t \frown t, X) \wedge$ 
    ( $s \frown \langle \checkmark \rangle \in_{\mathfrak{t}} \text{traces}(P) (\text{fstF} \circ M) \wedge$ 
    ( $(t, X) \in_{\mathfrak{f}} \text{failures}(Q) M \wedge \text{noTick } s \}_{\mathfrak{f}}$ )"
"failures( $P \downarrow n$ )    = ( $\lambda M. \text{failures}(P) M \downarrow n$ )"
"failures( $\$p$ )          = ( $\lambda M. \text{sndF}(M(p))$ )"

```

Figure 8: The encoding of the function *failures*



```

(* 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

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  $\llbracket P \rrbracket F = \text{traces}(P)(\text{fstF} \circ MF)"$ "
lemma sndF_semF[simp]: "sndF  $\llbracket P \rrbracket F = \text{failures}(P) MF"$ "
lemma cspF_eqF_semantics: "( $P =_F [M_1, M_2] Q$ ) =
  ((traces( $P$ )(fstF  $\circ M_1$ ) = traces( $Q$ )(fstF  $\circ M_2$ ))  $\wedge$ 
   (failures( $P$ )( $M_1$ ) = failures( $Q$ )( $M_2$ )))"
lemma cspF_refF_semantics: "( $P \sqsubseteq_F [M_1, M_2] Q$ ) =
  ((traces( $Q$ )(fstF  $\circ M_2$ )  $\leq$  traces( $P$ )(fstF  $\circ M_1$ ))  $\wedge$ 
   (failures( $Q$ )( $M_2$ )  $\leq$  failures( $P$ )( $M_1$ )))"

```

At the end of this subsection, we would like to briefly tell the expressive power of our CSP dialect  $\text{CSP}_{\text{TP}}$ . At first glance, the input language of CSP-Prover seems to be weaker than full CSP as the generic internal choice operator  $\sqcap \mathcal{P}$ <sup>8</sup> 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: " $\forall SF. \exists P. \llbracket P \rrbracket F = SF$ "

```

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

## 7.4 Recursive process

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

```

PNFun $\Pi$  (Empty(id)) = left?r  $\rightarrow$  $(Full(r, id))
PNFun $\Pi$  (Full(r, id)) = right(r, id)  $\rightarrow$  $(Empty(id + 1))

```

where the set of process names  $\Pi$  and the alphabet  $\Sigma$  are given as follows:

```

 $\Pi = \{\text{Empty}(id) \mid id \in \text{Nat}\} \cup \{\text{Full}(r, id) \mid r \in \text{Real}, id \in \text{Nat}\}$ 
 $\Sigma = \{\text{left}(r) \mid r \in \text{Real}\} \cup \{\text{right}(r, id) \mid r \in \text{Real}, id \in \text{Nat}\}$ 

```

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

`Buffer` = \$(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

<sup>8</sup> $\mathcal{P}$  is a non-emptyset of processes and the semantics is given as:

$$\begin{aligned} \text{traces}(\sqcap \mathcal{P}) &= \bigcup \{\text{traces}(P) \mid P \in \mathcal{P}\} \\ \text{failures}(\sqcap \mathcal{P}) &= \bigcup \{\text{failures}(P) \mid P \in \mathcal{P}\} \end{aligned}$$

```

1  datatype Event = left real | right "real * nat"
2  datatype Name  = Empty nat | Full real nat
3
4  consts Bufferfun :: "Name  $\Rightarrow$  (Name, Event) proc"
5  primrec
6    "Bufferfun (Empty n) = left ?r  $\rightarrow$  $(Full r n)"
7    "Bufferfun (Full r n) = right (r,n)  $\rightarrow$  $(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

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 automatised 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.  $\bigwedge p$ . 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.  $\bigwedge p$  nat. guarded (Bufferfun (Empty nat))
2.  $\bigwedge p$  real nat. guarded (Bufferfun (Full real nat))

```

Then, the subgoals can be automatically proven by `simps`. 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 *methods*. 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:

$$((a \rightarrow P) \parallel \{a\} \parallel (a \rightarrow Q)) =_{\mathcal{F}} a \rightarrow (P \parallel \{a\} \parallel Q)$$

### 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,

$$\dots \wedge t \in_{\mathbf{t}} \mathbf{traces}(a \rightarrow P)M \wedge \dots$$

and if the command (**simp add: in\_traces**) is applied, then it will be rewritten to the following subgoal (\*1):

$$\dots \wedge (t = \langle \rangle \vee (\exists s. t = \langle \mathbf{Ev} \ a \rangle \wedge s \wedge s \in_{\mathbf{t}} \mathbf{traces}(P)M)) \wedge \dots$$

On the other hand, if the command (**simp add: traces.simps**)<sup>9</sup> was applied instead of (**simp add: in\_traces**), it would be rewritten to the following subgoal (\*2):

$$\dots \wedge (t \in_{\mathbf{t}} \{t = \langle \rangle \vee (\exists s. t = \langle \mathbf{Ev} \ a \rangle \wedge s \wedge s \in_{\mathbf{t}} \mathbf{traces}(P)M)\}_{\mathbf{t}}) \wedge \dots$$

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

$$t \in \mathbf{Rep\_domT} (\mathbf{Abs\_domT} \{t. \dots\}),$$

thus the transformation from (\*2) to (\*1) requires that  $\{t. \dots\} \in \mathbf{domT}$ , in other words,  $\{t. \dots\}$  is a non-empty and prefix-closed set.

In CSP-Prover, the required property  $\{t. \dots\} \in \mathbf{domT}$  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

<sup>9</sup>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]**.

prefer `in_failures` to `failures.simps`. In summary, the semantical proof will proceed as follows:

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

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

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

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

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

by `(rule order_antisym)`.

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

$$\bigwedge t. (t \in_t \text{traces}(P)(M) \implies t \in_t \text{traces}(Q)(M))$$

by `(rule subdomTI)`.

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

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

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<sup>10</sup>. During the proof, the subgoals are sometimes complex. It will be found that CSP-Prover assists the proof well.

## 8.2 Syntactical manual proof

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

The CSP laws implemented in CSP-Prover are given in Figures 11, ..., 17. The CSP laws and their names such as  $(\square\text{-step})$  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.

<sup>10</sup>The lemmas whose name has the form `par_tr_...` relate to  $\llbracket X \rrbracket_{\text{tr}}$  over traces, and they are given in `Trace_par.thy` in CSP.

<b>"cspF_reflex"</b>	
$P =_{\mathcal{F}} P$	(reflexivity)
<b>"cspF_sym"</b>	
$P =_{\mathcal{F}} Q \implies Q =_{\mathcal{F}} P$	(symmetry)
<b>"cspF_trans"</b>	
$[P_1 =_{\mathcal{F}} P_2; P_2 =_{\mathcal{F}} P_3] \implies P_1 =_{\mathcal{F}} P_3$	(transitivity)
<b>"cspF_decompo"</b>	
$[a = b; P =_{\mathcal{F}} Q] \implies a \rightarrow P =_{\mathcal{F}} b \rightarrow Q$	(prefix-cong)
$[X = Y; \bigwedge x. x \in X \implies P(x) =_{\mathcal{F}} Q(x)] \implies ? x : X \rightarrow P(x) =_{\mathcal{F}} ? x : Y \rightarrow Q(x)$	(?-cong)
$[P_1 =_{\mathcal{F}} Q_1; P_2 =_{\mathcal{F}} Q_2] \implies P_1 \sqcap P_2 =_{\mathcal{F}} Q_1 \sqcap Q_2$	( $\sqcap$ -cong)
$[P_1 =_{\mathcal{F}} Q_1; P_2 =_{\mathcal{F}} Q_2] \implies P_1 \sqcup P_2 =_{\mathcal{F}} Q_1 \sqcup Q_2$	( $\sqcup$ -cong)
$[C_1 = C_2; \bigwedge c. c \in C_1 \implies P(c) =_{\mathcal{F}} Q(c)] \implies !! c : C_1 \rightarrow P(c) =_{\mathcal{F}} !! c : C_2 \rightarrow Q(c)$	(!!-cong)
$[P_1 =_{\mathcal{F}} Q_1; P_2 =_{\mathcal{F}} Q_2] \implies P_1 \llbracket X \rrbracket P_2 =_{\mathcal{F}} Q_1 \llbracket X \rrbracket Q_2$	( $\llbracket X \rrbracket$ -cong)
$[X = Y; P =_{\mathcal{F}} Q] \implies P \setminus X =_{\mathcal{F}} Q \setminus Y$	(hide-cong)
$[r_1 = r_2; P =_{\mathcal{F}} Q] \implies P[[r_1]] =_{\mathcal{F}} Q[[r_2]]$	(ren-cong)
$[P_1 =_{\mathcal{F}} Q_1; P_2 =_{\mathcal{F}} Q_2] \implies P_1 \circ P_2 =_{\mathcal{F}} Q_1 \circ Q_2$	( $\circ$ -cong)
$[n_1 = n_2; P =_{\mathcal{F}} Q] \implies P \downarrow n_1 =_{\mathcal{F}} Q \downarrow n_2$	( $\downarrow$ -cong) <sup>+</sup>

Figure 11: CSP congruence laws

<b>"cspF_IF"</b>	
$\text{IF True THEN } P \text{ ELSE } Q =_{\mathcal{F}} P$	(if-true)
$\text{IF False THEN } P \text{ ELSE } Q =_{\mathcal{F}} Q$	(if-false)
<b>"cspF_idem"</b>	
$P \sqcap P =_{\mathcal{F}} P$	( $\sqcap$ -idem)
$P \sqcap P =_{\mathcal{F}} P$	( $\sqcap$ -idem)
<b>"cspF_commut"</b>	
$P \sqcap Q =_{\mathcal{F}} Q \sqcap P$	( $\sqcap$ -sym)
$P \sqcap Q =_{\mathcal{F}} Q \sqcap P$	( $\sqcap$ -sym)
$P \llbracket X \rrbracket Q =_{\mathcal{F}} Q \llbracket X \rrbracket P$	( $\llbracket X \rrbracket$ -sym)
<b>"cspF_assoc"</b>	
$P \sqcap (Q \sqcap R) =_{\mathcal{F}} (P \sqcap Q) \sqcap R$	( $\sqcap$ -assoc)
$P \sqcap (Q \sqcap R) =_{\mathcal{F}} (P \sqcap Q) \sqcap R$	( $\sqcap$ -assoc)
<b>"cspF_unit"</b>	
$P \sqcap \text{Stop} =_{\mathcal{F}} P$	( $\sqcap$ -unit)
$P \sqcap \text{Div} =_{\mathcal{F}} P$	( $\sqcap$ -unit)
<b>"cspF_Rep_int_choice_empty"</b>	
$!! c : \emptyset \bullet P(c) =_{\mathcal{F}} \text{DIV}$	( $!!$ -emptyset) <sup>+</sup>
<b>"cspF_Rep_int_choice_const"</b>	
$\llbracket C \neq \emptyset; \forall c \in C. P(c) = Q \rrbracket \implies !! c : C \bullet P(c) =_{\mathcal{F}} Q$	( $!!$ -const) <sup>*</sup>
<b>"cspF_Rep_int_choice_union_Int"</b>	
$!! c : (C_1 \cup C_2) \bullet P(c) =_{\mathcal{F}} (!! c : C_1 \bullet P(c)) \sqcap (!! c : C_2 \bullet P(c))$	( $!!$ -union- $\sqcap$ ) <sup>*</sup>

Figure 12: CSP basic laws

<b>"cspF_Dist"</b>	
$C \neq \emptyset \implies (!!c : C \bullet P(c)) \sqcap Q =_{\mathcal{F}} !!c : C \bullet (P(c) \sqcap Q)$	( $\sqcap$ -Dist)
$C \neq \emptyset \implies (!!c : C \bullet P(c)) \llbracket X \rrbracket Q =_{\mathcal{F}} !!c : C \bullet (P(c) \llbracket X \rrbracket Q)$	( $\llbracket X \rrbracket$ -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)
$(!!c : C \bullet P(c)) \circledast Q =_{\mathcal{F}} !!c : C \bullet (P(c) \circledast Q)$	( $\circledast$ -Dist)
$(!!c : C \bullet P(c)) \sqcup n =_{\mathcal{F}} !!c : C \bullet (P(c) \sqcup n)$	( $\sqcup$ -Dist) <sup>+</sup>
<hr/>	
<b>"cspF_dist"</b>	
$(P_1 \sqcap P_2) \sqcap Q =_{\mathcal{F}} (P_1 \sqcap Q) \sqcap (P_2 \sqcap Q)$	( $\sqcap$ -dist)
$(P_1 \sqcap P_2) \llbracket X \rrbracket Q =_{\mathcal{F}} (P_1 \llbracket X \rrbracket Q) \sqcap (P_2 \llbracket X \rrbracket Q)$	( $\llbracket X \rrbracket$ -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)
$(P_1 \sqcap P_2) \circledast Q =_{\mathcal{F}} (P_1 \circledast Q) \sqcap (P_2 \circledast Q)$	( $\circledast$ -dist)
$(P_1 \sqcap P_2) \sqcup n =_{\mathcal{F}} (P_1 \sqcup n) \sqcap (P_2 \sqcup n)$	( $\sqcup$ -dist) <sup>+</sup>
$!!c : C \bullet (P_1(c) \sqcap P_2(c)) =_{\mathcal{F}} (!!c : C \bullet P_1(c)) \sqcap (!!c : C \bullet P_2(c))$	( $!!$ -dist)
<hr/>	
<b>"cspF_Ext_dist"</b>	
$(P_1 \sqcap P_2)[[r]] =_{\mathcal{F}} (P_1[[r]]) \sqcap (P_2[[r]])$	( $[[r]]$ - $\sqcap$ -dist)
$(P_1 \sqcap P_2) \sqcup n =_{\mathcal{F}} (P_1 \sqcup n) \sqcap (P_2 \sqcup n)$	( $\sqcup$ - $\sqcap$ -dist) <sup>+</sup>

Figure 13: CSP distributive laws



<b>"cspF_step"</b>	
$\text{STOP} =_{\mathcal{F}} ?x : \emptyset \rightarrow P(x)$	(stop-step)
$a \rightarrow P =_{\mathcal{F}} ?x : \{a\} \rightarrow P$	(prefix-step)
$(?x : A \rightarrow P(x)) \sqcap (?x : B \rightarrow Q(x))$ $=_{\mathcal{F}} ?x : (A \cup B) \rightarrow (\text{IF } (x \in A \cap B) \text{ THEN } P(x) \sqcap Q(x)$ $\quad \text{ELSE IF } (x \in A) \text{ THEN } P(x) \text{ ELSE } Q(x))$	( $\sqcap$ -step)
$(?x : A \rightarrow P'(x)) \parallel [X] (?x : B \rightarrow Q'(x))$ $=_{\mathcal{F}} (?x : ((X \cap A \cap B) \cup (A - X) \cup (B - X)) \rightarrow$ $\quad \text{IF } (x \in X)$ $\quad \text{THEN } (P'(x) \parallel [X] Q'(x))$ $\quad \text{ELSE IF } (x \in A \cap B)$ $\quad \quad \text{THEN } ((P'(x) \parallel [X] (?x : B \rightarrow Q'(x))) \sqcap$ $\quad \quad \quad ((?x : A \rightarrow P'(x)) \parallel [X] Q'(x)))$ $\quad \quad \text{ELSE IF } (x \in A)$ $\quad \quad \quad \text{THEN } (P'(x) \parallel [X] (?x : B \rightarrow Q'(x)))$ $\quad \quad \quad \text{ELSE } ((?x : A \rightarrow P'(x)) \parallel [X] Q'(x)))$	( $\parallel [X]$ -step)
$(?x : A \rightarrow P(x)) \setminus X$ $=_{\mathcal{F}} \text{IF } (A \cap X =_{\mathcal{F}} \emptyset)$ $\quad \text{THEN } ?x : A \rightarrow (P(x) \setminus X)$ $\quad \text{ELSE } (?x : (A - X) \rightarrow (P(x) \setminus X))$ $\quad \triangleright (!x : (A \cap X) \bullet (P(x) \setminus X))$	(hide-step)
$(?x : A \rightarrow P(x))[[r]]$ $=_{\mathcal{F}} ?x : \{x \mid \exists a \in A. (a, x) \in r\} \rightarrow$ $\quad (!a : \{a \in A \mid (a, x) \in r\} \bullet (P(a)[[r]]))$	( $[[r]]$ -step)
$(?x : A \rightarrow P(x)) \circ Q =_{\mathcal{F}} ?x : A \rightarrow (P(x) \circ Q)$	( $\circ$ -step)
$(?x : A \rightarrow P(x)) \downarrow (n+1) =_{\mathcal{F}} ?x : A \rightarrow (P(x) \downarrow n)$	( $\downarrow$ -step) <sup>+</sup>

Figure 14: CSP step laws

<b>"cspF_step_ext"</b>	
$ \begin{aligned} & ((?x : A \rightarrow P'(x)) \triangleright P'') \llbracket X \rrbracket ((?x : B \rightarrow Q'(x)) \triangleright Q'') \\ & =_{\mathcal{F}} (?x : ((X \cap A \cap B) \cup (A - X) \cup (B - X)) \rightarrow \\ & \quad \text{IF } (x \in X) \\ & \quad \text{THEN } (P'(x) \llbracket X \rrbracket Q'(x)) \\ & \quad \text{ELSE IF } (x \in A \cap B) \\ & \quad \quad \text{THEN } ((P'(x) \llbracket X \rrbracket ((?x : B \rightarrow Q'(x)) \triangleright Q'')) \sqcap \\ & \quad \quad \quad ((?x : A \rightarrow P'(x)) \triangleright P'') \llbracket X \rrbracket Q'(x))) \\ & \quad \quad \text{ELSE IF } (x \in A) \\ & \quad \quad \quad \text{THEN } (P'(x) \llbracket X \rrbracket ((?x : B \rightarrow Q'(x)) \triangleright Q'')) \\ & \quad \quad \quad \text{ELSE } (((?x : A \rightarrow P'(x)) \triangleright P'') \llbracket X \rrbracket Q'(x))) \\ & \triangleright ((P'' \llbracket X \rrbracket ((?x : B \rightarrow Q'(x)) \triangleright Q'')) \sqcap \\ & \quad ((?x : A \rightarrow P'(x)) \triangleright P'') \llbracket X \rrbracket Q'') \end{aligned} $	$(\llbracket X \rrbracket \text{-}\triangleright\text{-split})^*$
$ \begin{aligned} & ((?x : A \rightarrow P'(x)) \triangleright P'') \llbracket X \rrbracket (?x : B \rightarrow Q'(x)) \\ & =_{\mathcal{F}} (?x : ((X \cap A \cap B) \cup (A - X) \cup (B - X)) \rightarrow \\ & \quad \text{IF } (x \in X) \\ & \quad \text{THEN } (P'(x) \llbracket X \rrbracket Q'(x)) \\ & \quad \text{ELSE IF } (x \in A \cap B) \\ & \quad \quad \text{THEN } ((P'(x) \llbracket X \rrbracket (?x : B \rightarrow Q'(x))) \sqcap \\ & \quad \quad \quad ((?x : A \rightarrow P'(x)) \triangleright P'') \llbracket X \rrbracket Q'(x))) \\ & \quad \quad \text{ELSE IF } (x \in A) \\ & \quad \quad \quad \text{THEN } (P'(x) \llbracket X \rrbracket (?x : B \rightarrow Q'(x))) \\ & \quad \quad \quad \text{ELSE } (((?x : A \rightarrow P'(x)) \triangleright P'') \llbracket X \rrbracket Q'(x))) \\ & \triangleright (P'' \llbracket X \rrbracket (?x : B \rightarrow Q'(x))) \end{aligned} $	$(\llbracket X \rrbracket \text{-}\triangleright\text{-input})^*$
<b>"cspF_Ext_choice_SKIP_DIV_resolve"</b>	
$P \sqcap \text{SKIP} = P \triangleright \text{SKIP}$	$(\sqcap\text{-skip-resolve})$
$P \sqcap \text{DIV} = P \triangleright \text{DIV}$	$(\sqcap\text{-div-resolve})$
<b>"cspF_Depth_rest_Zero"</b>	
$P \lfloor 0 =_{\mathcal{F}} \text{DIV}$	$(\lfloor \text{-zero})^+$
<b>"cspF_Depth_rest_min"</b>	
$(P \lfloor n) \lfloor m =_{\mathcal{F}} P \lfloor \min(n, m)$	$(\lfloor \text{-min})^+$

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

<b>"cspF_SKIP_DIV"</b>	
$\text{SKIP} \sqcap \text{DIV} =_{\mathcal{F}} \text{SKIP}$	(skip-div- $\sqcap$ )
$\text{SKIP} \llbracket X \rrbracket \text{SKIP} =_{\mathcal{F}} \text{SKIP}$	(skip- $\llbracket X \rrbracket$ )
$\text{DIV} \llbracket X \rrbracket \text{DIV} =_{\mathcal{F}} \text{DIV}$	(div- $\llbracket X \rrbracket$ )
$\text{SKIP} \llbracket X \rrbracket \text{DIV} =_{\mathcal{F}} \text{DIV}$	(skip-div- $\llbracket X \rrbracket$ )
$\text{SKIP} \llbracket X \rrbracket (?x : A \rightarrow P(x))$	
$=_{\mathcal{F}} ?x : (A - X) \rightarrow (\text{SKIP} \llbracket X \rrbracket P(x))$	( $\llbracket X \rrbracket$ -preterm)
$\text{DIV} \llbracket X \rrbracket (?x : A \rightarrow P(x))$	
$=_{\mathcal{F}} (?x : (A - X) \rightarrow (\text{DIV} \llbracket X \rrbracket P(x))) \sqcap \text{DIV}$	(div- $\llbracket X \rrbracket$ -step)
$\text{SKIP} \llbracket X \rrbracket ((?x : A \rightarrow P(x)) \sqcap \text{SKIP})$	
$=_{\mathcal{F}} (?x : (A - X) \rightarrow (\text{SKIP} \llbracket X \rrbracket P(x))) \sqcap \text{SKIP}$	(skip- $\llbracket X \rrbracket$ - $\sqcap$ -skip)
$\text{SKIP} \llbracket X \rrbracket ((?x : A \rightarrow P(x)) \sqcap \text{DIV})$	
$=_{\mathcal{F}} (?x : (A - X) \rightarrow (\text{SKIP} \llbracket X \rrbracket P(x))) \sqcap \text{DIV}$	(skip- $\llbracket X \rrbracket$ - $\sqcap$ -div)
$\text{DIV} \llbracket X \rrbracket (P \sqcap \text{SKIP}) =_{\mathcal{F}} \text{DIV} \llbracket X \rrbracket P$	(div- $\llbracket X \rrbracket$ - $\sqcap$ -skip)
$\text{DIV} \llbracket X \rrbracket (P \sqcap \text{DIV}) =_{\mathcal{F}} \text{DIV} \llbracket X \rrbracket P$	(div- $\llbracket X \rrbracket$ - $\sqcap$ -div)
$\text{SKIP} \setminus X =_{\mathcal{F}} \text{SKIP}$	(skip-hide)
$\text{DIV} \setminus X =_{\mathcal{F}} \text{DIV}$	(div-hide)
$((?x : A \rightarrow P(x)) \sqcap \text{SKIP}) \setminus X$	
$=_{\mathcal{F}} ((?x : (A - X) \rightarrow (P(x) \setminus X)) \sqcap \text{SKIP})$	
$\sqcap (!x : (A \cap X) \bullet (P(x) \setminus X))$	(skip-hide-step)
$((?x : A \rightarrow P(x)) \sqcap \text{DIV}) \setminus X$	
$=_{\mathcal{F}} ((?x : (A - X) \rightarrow (P(x) \setminus X)) \sqcap \text{DIV})$	
$\sqcap (!x : (A \cap X) \bullet (P(x) \setminus X))$	(div-hide-step)
$\text{SKIP}[[r]] =_{\mathcal{F}} \text{SKIP}$	(skip- $[[r]]$ -id)
$\text{DIV}[[r]] =_{\mathcal{F}} \text{DIV}$	(div- $[[r]]$ -id)
$\text{SKIP} \circledast P =_{\mathcal{F}} P$	( $\circledast$ -unit-l)
$\text{DIV} \circledast P =_{\mathcal{F}} \text{DIV}$	(div- $\circledast$ )
$((?x : A \rightarrow P(x)) \triangleright \text{SKIP}) \circledast R$	
$=_{\mathcal{F}} (?x : A \rightarrow (P(x) \circledast R)) \triangleright R$	(skip- $\circledast$ -step)
$((?x : A \rightarrow P(x)) \triangleright \text{DIV}) \circledast R$	
$=_{\mathcal{F}} (?x : A \rightarrow (P(x) \circledast R)) \triangleright \text{DIV}$	(div- $\circledast$ -step)
$\text{SKIP} \lfloor (n+1) =_{\mathcal{F}} \text{SKIP}$	(skip- $\lfloor$ ) <sup>+</sup>
$\text{DIV} \lfloor n =_{\mathcal{F}} \text{DIV}$	(div- $\lfloor$ ) <sup>+</sup>

Figure 16: CSP skip and div laws

<b>"cspF_Rep_int_choice_input_set"</b>	
$!!c : C \bullet (?x : A(c) \rightarrow P(c, x))$ $=_{\mathcal{F}} !\text{set } X : \{A(c) \mid c \in C\} \bullet$ $(?x : X \rightarrow (!!c : \{c \in C \mid x \in A(c)\} \bullet P(c, x)))$	$(!!\text{-input-!set})^+$
<b>"cspF_Rep_int_choice_Ext_Dist"</b>	
$\forall c \in C. Q(c) \in \{\text{SKIP}, \text{DIV}\} \implies$ $!!c : C \bullet (P(c) \sqcap Q(c))$ $=_{\mathcal{F}} (!!c : C \bullet P(c)) \sqcap (!!c : C \bullet Q(c))$	$(!!\text{-}\sqcap\text{-Dist})^+$
<b>"cspF_Rep_int_choice_input_Dist"</b>	
$[ [ Q = \text{SKIP} \vee Q = \text{DIV} ] ] \implies$ $(!\text{set } X : \mathcal{X} \bullet (?x : X \rightarrow P(x))) \sqcap Q$ $=_{\mathcal{F}} (?x : \bigcup \mathcal{X} \rightarrow P(x)) \sqcap Q$	$(!!\text{-input-Dist})^+$
<b>"cspF_norm"</b>	
$?x : A \rightarrow P(x)$ $=_{\mathcal{F}} ((?x : A \rightarrow P(x)) \sqcap \text{DIV}) \sqcap (?x : A \rightarrow \text{DIV})$	$(?\text{-div})^+$
$!!c : C \bullet (!\text{set } X : \mathcal{X}(c) \bullet (?x : X \rightarrow \text{DIV}))$ $=_{\mathcal{F}} !\text{set } X : \bigcup \{\mathcal{X}(c) \mid c \in C\} \bullet (?x : X \rightarrow \text{DIV})$	$(!!\text{-!set-div})^+$
$\text{if } \mathcal{X} \subseteq \mathcal{Y} \text{ and } (\forall Y \in \mathcal{Y}. \exists X \in \mathcal{X}. X \subseteq Y \subseteq A) \text{ then}$ $((?x : A \rightarrow P(x)) \sqcap R) \sqcap (!\text{set } X : \mathcal{X} \bullet (?x : X \rightarrow \text{DIV}))$ $=_{\mathcal{F}} ((?x : A \rightarrow P(x)) \sqcap R) \sqcap (!\text{set } X : \mathcal{Y} \bullet (?x : X \rightarrow \text{DIV}))$	$(?\text{-!set-}\subseteq)^+$
<b>"cspF_nat_Depth_rest"</b>	
$P =_{\mathcal{F}} !\text{nat } n \bullet (P \lfloor n)$	$(!\text{nat-}\lfloor)^+$

Figure 17: CSP replicated internal choice laws and normalising laws

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.  $\llbracket P \rrbracket_{\mathcal{F}} = \llbracket Q \rrbracket_{\mathcal{F}}$ ) ? 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`<sup>11</sup> 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.

Next, it may be expected to apply the ( $\llbracket X \rrbracket$ -step) law, but it is not available yet. Before applying ( $\llbracket X \rrbracket$ -step),  $(a \rightarrow P)$  has to be 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:

<sup>11</sup>The lemma `cspF_rw_right` includes  $\llbracket P_3 =_{\mathcal{F}} P_2; P_1 =_{\mathcal{F}} P_2 \rrbracket \implies P_1 =_{\mathcal{F}} P_3$ , and it can be derived from `cspF_trans` and `cspF_sym`. `cspF_rw_left` includes `cspF_trans`.

```
goal (lemma (syntactical_proof), 3 subgoals):
1.  $a \rightarrow P =_{\mathcal{F}} ?Q1.1$ 
2.  $a \rightarrow Q =_{\mathcal{F}} ?Q2.1$ 
3.  $?Q1.1 \parallel \{a\} \parallel ?Q2.1 =_{\mathcal{F}} a \rightarrow (P \parallel \{a\} \parallel Q)$ 
```

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

```
goal (lemma (syntactical_proof), 2 subgoals):
1.  $a \rightarrow Q =_{\mathcal{F}} ?Q2.1$ 
2.  $(?x : \{a\} \rightarrow P) \parallel \{a\} \parallel ?Q2.1 =_{\mathcal{F}} a \rightarrow (P \parallel \{a\} \parallel Q)$ 
```

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

```
goal (lemma (syntactical_proof), 1 subgoal):
1.  $(?x : \{a\} \rightarrow P) \parallel \{a\} \parallel (?x : \{a\} \rightarrow Q) =_{\mathcal{F}} a \rightarrow (P \parallel \{a\} \parallel Q)$ 
```

Then, you can apply the ( $\parallel X$ )-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  $\sqsubseteq_{\mathcal{F}}$ <sup>12</sup>. 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_...`).

You will find a lot of syntactical proof technique in the theory-files (e.g. lemma `cspF_fsF_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.  $(P \sqcap Q) \sqcap R =_{\mathcal{F}} P \sqcap (Q \sqcap R)$ ), 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

<sup>12</sup>For example, (rule `cspF_rw_right`) includes  $\parallel P_3 =_{\mathcal{F}} P_2; P_1 \sqsubseteq_{\mathcal{F}} P_2 \parallel \implies P_1 \sqsubseteq_{\mathcal{F}} P_3$ , and (rule `cspF_decompo`) includes  $\parallel a = b; P \sqsubseteq_{\mathcal{F}} Q \parallel \implies a \rightarrow P \sqsubseteq_{\mathcal{F}} b \rightarrow Q$ .

"cspF_ref_eq_iff"	
$(P \sqsubseteq_{\mathcal{F}} Q) = (P =_{\mathcal{F}} Q \sqcap P)$	$(\sqsubseteq_{\mathcal{F}} =_{\mathcal{F}} \text{-iff})$
"cspF_ref_eq"	
$[  P \sqsubseteq_{\mathcal{F}} Q; Q \sqsubseteq_{\mathcal{F}} P  ] \implies P =_{\mathcal{F}} Q$	$(\sqsubseteq_{\mathcal{F}} =_{\mathcal{F}})$
"cspF_eq_ref"	
$P =_{\mathcal{F}} Q \implies P \sqsubseteq_{\mathcal{F}} Q$	$(=_{\mathcal{F}} \sqsubseteq_{\mathcal{F}})$
"cspF_Int_choice_left1"	
$P_1 \sqsubseteq_{\mathcal{F}} Q \implies P_1 \sqcap P_2 \sqsubseteq_{\mathcal{F}} Q$	$(\sqcap \text{-left-1})$
"cspF_Int_choice_left2"	
$P_2 \sqsubseteq_{\mathcal{F}} Q \implies P_1 \sqcap P_2 \sqsubseteq_{\mathcal{F}} Q$	$(\sqcap \text{-left-2})$
"cspF_Int_choice_right"	
$[  P \sqsubseteq_{\mathcal{F}} Q_1; P \sqsubseteq_{\mathcal{F}} Q_2  ] \implies P \sqsubseteq_{\mathcal{F}} Q_1 \sqcap Q_2$	$(\sqcap \text{-right})$
"cspF_Rep_int_choice_left"	
$(\exists c. c \in C \wedge P(c) \sqsubseteq_{\mathcal{F}} Q) \implies !! c : C \bullet P(c) \sqsubseteq_{\mathcal{F}} Q$	$(!! \text{-left})$
"cspF_Rep_int_choice_right"	
$(\bigwedge c. c \in C \implies P \sqsubseteq_{\mathcal{F}} Q(c)) \implies P \sqsubseteq_{\mathcal{F}} !! c : C \bullet Q(c)$	$(!! \text{-right})$
"cspF_decompo_subset"	
$[  C_2 \subseteq C_1; \bigwedge c. c \in C_2 \implies P(c) \sqsubseteq_{\mathcal{F}} Q(c)  ]$ $\implies !! c : C_1 \bullet P(c) \sqsubseteq_{\mathcal{F}} !! c : C_2 \bullet Q(c)$	$(!! \text{-subset})$
$[  Y \neq \{\}; Y \subseteq X; \bigwedge x. x \in Y \implies P(x) \sqsubseteq_{\mathcal{F}} Q(x)  ]$ $\implies ! x : X \bullet (x \rightarrow P(x)) \sqsubseteq_{\mathcal{F}} ? x : Y \rightarrow Q(x)$	$(!! \text{-?subset})$
"cspF_Ext_choice_right"	
$[  P \sqsubseteq_{\mathcal{F}} Q_1; P \sqsubseteq_{\mathcal{F}} Q_2  ] \implies P \sqsubseteq_{\mathcal{F}} Q_1 \sqcup Q_2$	$(\sqcup \text{-right})$

Figure 18: CSP refinement laws

<b>"cspF_FIX"</b>	
$\begin{aligned} & \llbracket \text{FPmode} = \text{CMSmode} \longrightarrow \text{guardedfun PNfun} \rrbracket \\ & \implies \$p =_{\mathcal{F}} (\text{!nat } n \bullet ((\text{PNfun} \triangleleft) ^{(n)} (\lambda p'. \text{DIV}))(p))) \quad (\text{fix\_!nat}) \end{aligned}$	
<b>"cspF_unwind"</b>	
$\begin{aligned} & \llbracket \text{FPmode} = \text{CMSmode} \longrightarrow \text{guardedfun PNfun} \rrbracket \\ & \implies \$p =_{\mathcal{F}} \text{PNfun}(p) \end{aligned}$	(unwind)
<b>"cspF_fp_induct_right"</b>	
$\begin{aligned} & \llbracket \text{FPmode} = \text{CMSmode} \longrightarrow \text{guardedfun PNfun}; \\ & \quad Q \sqsubseteq_{\mathcal{F}} f(p); \bigwedge p. f(p) \sqsubseteq_{\mathcal{F}} \text{PNfun}(p) \triangleleft f \rrbracket \\ & \implies Q \sqsubseteq_{\mathcal{F}} \$p \end{aligned}$	(induct-right-ref)
$\begin{aligned} & \llbracket \text{FPmode} = \text{CMSmode}; \text{guardedfun PNfun}; \\ & \quad Q =_{\mathcal{F}} f(p); \bigwedge p. f(p) =_{\mathcal{F}} \text{PNfun}(p) \triangleleft f \rrbracket \\ & \implies Q =_{\mathcal{F}} \$p \end{aligned}$	(induct-right-eq)
<b>"cspF_fp_induct_left"</b>	
$\begin{aligned} & \llbracket \text{FPmode} = \text{CMSmode}; \text{guardedfun PNfun}; \\ & \quad f(p) \sqsubseteq_{\mathcal{F}} Q; \bigwedge p. \text{PNfun}(p) \triangleleft f \sqsubseteq_{\mathcal{F}} f(p) \rrbracket \\ & \implies \$p \sqsubseteq_{\mathcal{F}} Q \end{aligned}$	(induct-left-ref)
$\begin{aligned} & \llbracket \text{FPmode} = \text{CMSmode}; \text{guardedfun PNfun}; \\ & \quad f(p) =_{\mathcal{F}} Q; \bigwedge p. \text{PNfun}(p) \triangleleft f =_{\mathcal{F}} f(p) \rrbracket \\ & \implies \$p =_{\mathcal{F}} Q \end{aligned}$	(induct-left-eq)

Figure 19: CSP fixed-point laws

```

1  datatype DFName = DF
2
3  consts DFfun :: "DFName  $\Rightarrow$  (DFName, Event) proc"
4  primrec "DFfun (DF) = ! x  $\rightarrow$  $DF"
5  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  $\$DF \sqsubseteq_{\mathcal{F}} \text{Buffer}$ ,



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  $\$DF \sqsubseteq_{\mathcal{F}} \text{Buffer}$  as the main goal and unfolding the definition of **Buffer**, the following goal is displayed:

```
goal (lemma (manual_proof_Buffer), 1 subgoal):
  1. $DF \sqsubseteq_{\mathcal{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 \sqsubseteq_{\mathcal{F}} ?f(\$Empty 0)
  2. \bigwedge p. ?f(p) \sqsubseteq_{\mathcal{F}} (Bufferfun(p))\triangleleft ?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 \Rightarrow (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 \sqsubseteq_{\mathcal{F}} Buffer_to_DF (Empty 0)
  2. \bigwedge p. Buffer_to_DF(p) \sqsubseteq_{\mathcal{F}} (Bufferfun(p))\triangleleft 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 <sup>13</sup>:

<sup>13</sup>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`.

```
goal (lemma (manual_proof_Buffer), 2 subgoals):
1.  $\$DF \sqsubseteq_{\mathcal{F}} ? a : (\text{range left}) \rightarrow \$DF$ 
2.  $\bigwedge_{\text{real nat.}} \$DF \sqsubseteq_{\mathcal{F}} \text{right } (\text{real}, \text{nat}) \rightarrow \$DF$ 
```

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 *methods* to automatically apply CSP laws.

The most powerful CSP-method is `cspF_auto` which automatically applies CSP-laws to unguarded subexpressions in both sides. The method `cspF_auto` applies the following CSP laws with the following priority (i.e. the assumptions has the highest priority).

1. Assumptions, (i.e. `asm_full_simp_tac` is used)
2. The law `cspF_choice_IF`, which consists of (`cspF_IF`), (`cspF_idem`), (`cspF_unit`), etc to simplify processes.
3. Laws specified by users. (explained below)
4. The law `cspF_all_dist`, which consists of `cspF_dist`, `cspF_Dist`, etc, to distribute operators on choice operators. (see `CSP_F_law_dist.thy`)
5. The laws for renaming prefixes by short notations  $\longleftrightarrow$  and  $\Longleftrightarrow$ .
6. The law `cspF_SKIP_DIV_sort`, which is derived from (`cspF_commut`) and (`cspF_assoc`) to sort processes over  $\square$  to the form  $?x : X \rightarrow P(x) \square \text{SKIP}$  or  $?x : X \rightarrow P(x) \square \text{DIV}$  if unguarded `SKIP` or `DIV` exists.
7. 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`.
8. The law `cspF_step`, to sequentialise processes.
9. The auxiliary laws which delay rewriting internal choice.
10. The law `cspF_unwind` to unwind recursive processes.

(see the ML-function `cspF_auto_core` in `CSP_F_tactic.thy` for more details)

Since the method `cspF_auto` is powerful, it is sometimes required to apply only specific laws to avoid excessive rewriting. Therefore, the following sub methods are also provided, where numbers refer to the CSP-laws in `cspF_auto` above.

- `cspF_simp` applies CSP-laws 2 and 3.
- `cspF_asm` applies CSP-laws 1, 2 and 3.
- `cspF_dist` applies CSP-laws 2, 3, and 4.
- `cspF_ren` applies CSP-laws 2, 3, and 5.
- `cspF_unwind` applies CSP-laws 2, 3, and 10.
- `cspF_step` applies CSP-laws 2, 3, 5, 6, 7, and 8.
- `cspF_hsf` applies CSP-laws 2, 3, 4, 5, 6, 7, and 8.

You can apply CSP-methods by the command **apply** as usual, for example,

```
apply (cspF_auto)
```

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

```
apply (cspF_auto)+
```

Take a look at the proof-script of the lemma `tactical_proof` in `Test_proof.thy` in `Test`. By the method, the lemma is easily proven only by two lines:

```
apply (cspF_auto)+  
apply (auto)
```

If you want to apply CSP-methods only to the left-hand side (resp., right-hand side), attach the `_left` (resp., `_right`) to the last of the names of the methods. For example, the following command applies CSP-laws only to the left hand:

```
apply (cspF_auto_left)
```

If you want to apply CSP-laws *laws*, which you have already proven, to each unguarded subexpression<sup>14</sup>, specify them as an argument of methods as follows:

```
apply (cspF_simp laws)
```

For example, assume that the following law has already been proven:

```
lemma new_law: "(P □ Q) □ P =F (P □ Q)"
```

Then, you can apply the law to each subexpression in the first subgoal by

```
apply (cspF_simp new_law)
```

This option can be used for separating a large proof into some partial proofs as shown in the package `DM`. To simultaneously apply two or more proven laws,

<sup>14</sup>For example, CSP-laws can be applied to each subprocess  $P$  and  $Q$  in  $P \sqcap Q$ , but cannot be applied to  $R$  in  $a \rightarrow R$ . This constraint is used for avoiding infinite rewriting by unwinding.

the command **lemmas** will be useful, for example to combine **law1** and **law2** to **laws**:

```
lemmas laws = law1 law2
```

All the above methods applies CSP-laws only to unguarded subexpressions, but it is sometimes useful to *deeply* simplify guarded subexpressions even if it takes a lot of time. Therefore, The following methods are also provided:

- **cspF\_simp\_deep** is the deep version of **cspF\_simp**, thus it applies CSP-laws 2 and 3 to all the subexpressions.
- **cspF\_ren\_deep** is the deep version of **cspF\_ren**, thus it applies CSP-laws 2, 3 and 5 to all the subexpressions.

All the method explained above can be applied to both verifications of the process equivalence  $P =_{\mathcal{F}} Q$  and the process refinement  $P \sqsubseteq_{\mathcal{F}} Q$ . Next, the methods for the process refinement are given:

- **cspF\_refine\_asm\_left** (resp., **cspF\_refine\_asm\_right**) rewrites the left (resp., right) hand side by the transitive law with using assumptions.
- **cspF\_refine\_left** (resp., **cspF\_refine\_right**) rewrites the left (resp., right) hand side by the transitive law without using assumptions.

These methods also have an argument for adding the laws already proven. For example, assume that the following law has already been proven:

```
lemma new_ref_law: "P ▷ Q ⊆ℱ P □ Q"
```

Then, you can apply the law to the left hand side by the following command.

```
apply (cspF_refine_left new_ref_law)
```

By this command,  $\triangleright$  in the left hand side is replaced by  $\square$ . For example, by this command,

$$(P_1 \triangleright P_2) \square P_3 \sqsubseteq_{\mathcal{F}} (Q_1 \triangleright Q_2) \square Q_3 \quad (*_1)$$

is rewritten to

$$(P_1 \square P_2) \square P_3 \sqsubseteq_{\mathcal{F}} (Q_1 \triangleright Q_2) \square Q_3. \quad (*_2)$$

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

On the other hand, when the law **new\_ref\_law** is applied to the right hand side by the following command,  $\square$  is replaced by  $\triangleright$ .

```
apply (cspF_refine_right new_ref_law)
```

For example, by this command, the subgoal  $(*_1)$  is rewritten to

$$(P_1 \triangleright P_2) \square P_3 \sqsubseteq_{\mathcal{F}} (Q_1 \triangleright Q_2) \triangleright Q_3. \quad (*_3)$$

Note that  $(*_3)$  implies  $(*_1)$ .

**[version 5]:** In CSP-Prover version 4, ML functions are used for giving semi-automatic commands and they can be applied by the method **tactic**, for example,

```
apply (tactic {* cspF_simp_tac 1 *})
```

Therefore, the following command applications

```
apply (tactic {* cspF_..._tac 1 *})
apply (tactic {* cspF_simp_with_tac "law" 1 *}),
```

for the CSP-Prover version 4 have to be replaced by

```
apply (cspF_...)
apply (cspF_simp law)
```

for the new CSP-Prover version 5.

## 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.

- [IR08] Yoshinao Isobe and Markus Roggenbach. Csp-prover – a proof tool for the verification of scalable concurrent systems. *Computer Software – JSSST Journal*, 25(4):85–92, 2008.
- [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  $P_1 \circ P_2$  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)  $(a \rightarrow \text{SKIP}) \circ \$p$
- (2)  $\text{SKIP} \circ (a \rightarrow \$p)$
- (3)  $\text{SKIP} \circ \$p$

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.

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

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  $\Rightarrow$  bool"

primrec
  "gSKIP(STOP)          = True"
  "gSKIP(SKIP)          = False"
  "gSKIP(DIV)           = True"
  "gSKIP( $a \rightarrow P$ )      = True"
  "gSKIP( $? : X \rightarrow P$ ) = True"
  "gSKIP( $P \sqcap Q$ )       = gSKIP( $P$ )  $\wedge$  gSKIP( $Q$ )"
  "gSKIP( $P \sqcap Q$ )       = gSKIP( $P$ )  $\wedge$  gSKIP( $Q$ )"
  "gSKIP( $!! : C \bullet P$ )   = ( $\forall c \in C. \text{gSKIP}(P(c))$ )"
  "gSKIP(IF  $b$  THEN  $P$  ELSE  $Q$ ) = gSKIP( $P$ )  $\wedge$  gSKIP( $Q$ )"
  "gSKIP( $P \parallel X \parallel Q$ ) = gSKIP( $P$ )  $\vee$  gSKIP( $Q$ )"
  "gSKIP( $P \setminus X$ )     = False"
  "gSKIP( $P[[r]]$ )         = gSKIP( $P$ )"
  "gSKIP( $P \circledast Q$ )       = gSKIP( $P$ )  $\vee$  gSKIP( $Q$ )"
  "gSKIP( $P \lfloor n$ )         = gSKIP( $P$ )  $\vee n = 0$ "
  "gSKIP( $\$p$ )             = False"

```

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

```

consts guarded :: "('p,'a) proc  $\Rightarrow$  bool"

primrec
  "guarded(STOP)        = True"
  "guarded(SKIP)        = True"
  "guarded(DIV)         = True"
  "guarded( $a \rightarrow P$ )     = noHide( $P$ )"
  "guarded( $? : X \rightarrow P$ ) = ( $\forall a \in X. \text{noHide}(P(a))$ )"
  "guarded( $P \sqcap Q$ )      = guarded( $P$ )  $\wedge$  guarded( $Q$ )"
  "guarded( $P \sqcap Q$ )      = guarded( $P$ )  $\wedge$  guarded( $Q$ )"
  "guarded( $!! : C \bullet P$ ) = ( $\forall c \in C. \text{guarded}(P(c))$ )"
  "guarded(IF  $b$  THEN  $P$  ELSE  $Q$ ) = guarded( $P$ )  $\wedge$  guarded( $Q$ )"
  "guarded( $P \parallel X \parallel Q$ ) = guarded( $P$ )  $\wedge$  guarded( $Q$ )"
  "guarded( $P \setminus X$ )   = noPN( $P$ )"
  "guarded( $P[[r]]$ )       = guarded( $P$ )"
  "guarded( $P \circledast Q$ )    = (guarded( $P$ )  $\wedge$  gSKIP( $P$ )  $\wedge$  noPN( $Q$ ))  $\vee$ 
    (guarded( $P$ )  $\wedge$  guarded( $Q$ ))"
  "guarded( $P \lfloor n$ )       = guarded( $P$ )  $\vee n = 0$ "
  "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( $P \circledast Q$ )` 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  $\Rightarrow$  ('q,'a) proc)  $\Rightarrow$  bool"
defs
  guardedfun_def: "guardedfun(f) == ( $\forall p$ . guarded(f(p)))"
```

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