User Guide CSP-Prover 2005

CSP-Prover Document
Version: DRAFT February 13, 2005

Yoshinao Isobe, Markus Roggenbach
E-mail address for comments: y-isobe@aist.go.jp, M.Roggenbach@swan.ac.uk

Contents

1 Introduction **2**

2 Installing Isabelle2004 **2**

3 Setting up CSP-Prover **3**

4 Starting CSP-Prover **5**

5 Small demonstrations **6**

6 Syntax **7**

7 Trace **11**

8 Domain **12**

9 Semantics **13**

10 Verification **13**

10.1 Fixed point induction based on cms 14

10.2 Fixed point induction based on cpo 16

10.3 Expanding 16

10.4 Decomposition 18

10.5 Internal choice 19

1 Introduction

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

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

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

Currently Csp-Prover offers as Csp semantics the traces model and stable failure models.

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

2 Installing Isabelle2004

Csp-Prover is encoded in Isabelle2004/HOL-Complex. To install the interactive theorem prover Isabelle follow the instructions of the Isabelle Web page:

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

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

```
Isabelle2004.tar.gz
ProofGeneral-3.5.tar.gz
polyml_base.tar.gz
polyml_x86-linux.tar.gz
HOL_x86-linux.tar.gz
```

```
HOL-Complex_x86-linux.tar.gz
```

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

```
tar -C /usr/local -xzf Isabelle2004.tar.gz
tar -C /usr/local -xzf ProofGeneral.tar.gz
tar -C /usr/local -xzf polyml_base.tar.gz
tar -C /usr/local -xzf polyml_x86-linux.tar.gz
tar -C /usr/local -xzf HOL_x86-linux.tar.gz
tar -C /usr/local -xzf HOL-Complex_x86-linux.tar.gz
```

Isabelle/Isar/HOL is started by

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

Proof General is started by

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

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

3 Setting up CSP-Prover

Download the file `CSP-Prover2005-2.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-Prover2005-2
```

by an unpacking command (e.g. `tar zxvf CSP-Prover2005-2.tar.gz`).

Figure 1 shows the contents of CSP-Prover2005-2. The directories are used as follows:

- `CSP-Prover` contains the theory files for Csp-Prover
- `Examples` contains small examples for testing Csp-Prover.
- `DM` contains the theory files for an example to verify a classical mutual exclusion problem called the Dining mathematicians[CS01].
- `DM-Seq` is another version of the Dining Mathematicians. A sequential behavior `Seq` equivalent to a concurrent behavior `Sys` is given between a specification `Spc` and `Sys`.
- `ep2` contains the theory files for an industrial case study on an electronic payment system **ep2**[ep202].
- `doc` contains documentation about Csp-Prover.

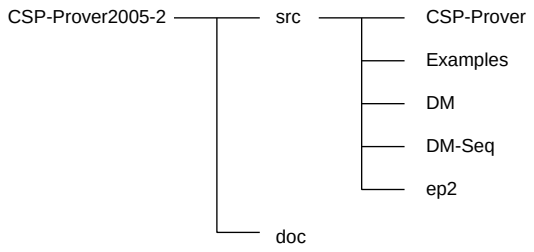

Figure 1: The directory tree of CSP-Prover2005-2

It is recommended to make a heap file: `CSP-Prover`, although you can directly load `CSP_Prover.thy`, and then prove it, and then use it. If you make the heap file once, you do not prove them again before using them. The heap file can be made as follows (or you can also download the heap file for Linux from `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`):

1. Go to the directory `CSP-Prover` by

   ```
   cd /usr/local/CSP-Prover2005-2/src/CSP-Prover
   ```

2. Make the heap file `CSP-Prover` by

   ```
   isatool usedir -b HOL-Complex CSP-Prover
   ```

   The heap file will be made in your isabelle directory. If you did not specify the directory, it is probably

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

   It may take time to make the heap file. For example, 10 minutes by Pentium M (1.5GHz).

In addition, if you like to comfortably read theory files of `CSP-Prover` by browsers (e.g. Netscape, mozilla, $\cdots$), you can make html files for them as follows:

1. Go to the directory `src` by

   ```
   cd /usr/local/CSP-Prover2005-2/src
   ```

2. Make html-files by

   ```
   isatool usedir -i true HOL-Complex CSP-Prover
   ```

3. Browse theory files and theory dependency-graphs by

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

   or the theory dependency-graphs by

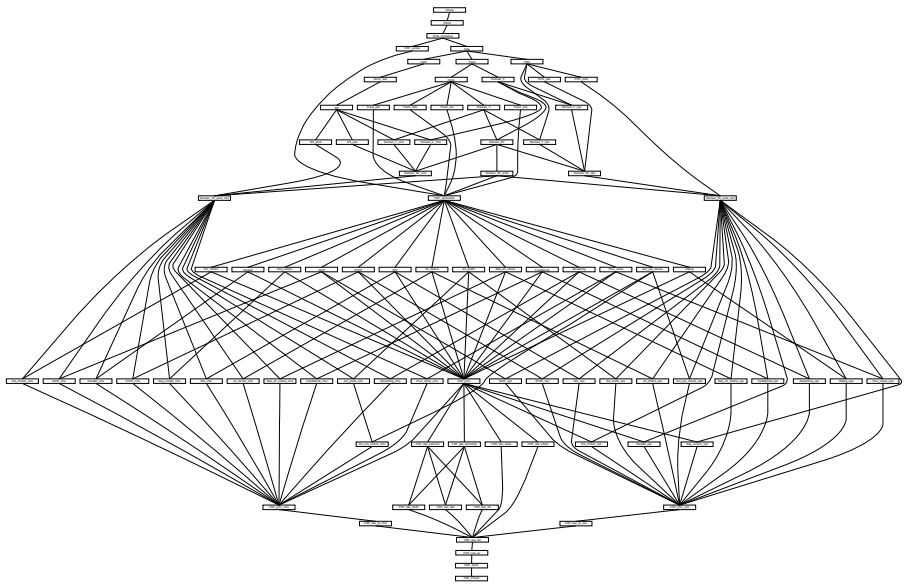

Figure 2: The dependency-graph for Csp-Prover

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

where Java is needed for displaying graphs. The dependency-graph created by isatool for Csp-Prover is shown in Figure 2.

4 Starting CSP-Prover

You can start Csp-Prover in a shell window by

```
isabelle -I CSP-Prover
```

or start it in Proof General[Asp00] by

```
Isabelle -l CSP-Prover
```

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

```
Isabelle -u false -l CSP-Prover
```

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

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

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

5 Small demonstrations

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

1. Start Csp-Prover in the shell window by

   ```
   isabelle -I CSP-Prover
   ```

2. Open the following example in the editor window:

   ```
   /usr/local/CSP-Prover2005-2/src/Examples/Inc_nat.thy
   ```

3. Copy the commands from "`Inc_nat.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-Prover by

   ```
   Isabelle -l CSP-Prover
   ```

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

   ```
   /usr/local/CSP-Prover2005-2/src/Examples/Inc_nat.thy
   ```

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

More conventional Csp-syntax can be displayed as shown in Figure 3 if you use Proof General and activate X-symbols from the menu bar.

Similarly, try to prove another example:

```
/usr/local/CSP-Prover2005-2/src/Examples/Test_Seq.thy
```

The examples `ep2` and `DM` are explained in the web-page:

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

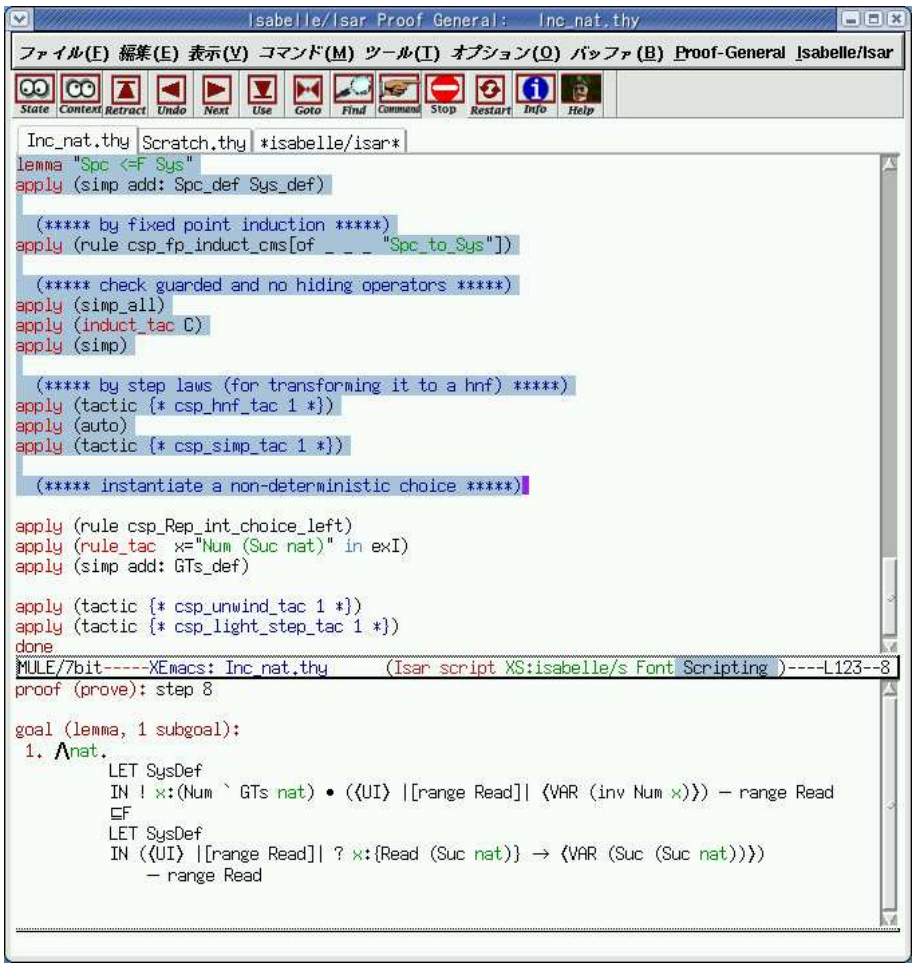

Figure 3: A screen shot of a proof by CSP-Prover

6 Syntax

The process algebra CSP [Hoa85, Ros98] is defined relative to a given set of communications. Its *basic processes* are built from primitive processes like `SKIP` and `STOP`. CSP includes communication primitives like sending and receiving values over a communication channel, distinguishes between internal and external choice between two processes, offers a variety of parallel operators, sequential composition of processes, and various other features like renaming and hiding. Figure 4 shows that the CSP dialect implemented by CSP-Prover covers all these features. This syntax definition involves certain Isabelle notations: given a type `'a` as set of communications, `a:'a` is a single communication, `c:('v⇒'a)` de-

```
P ::= SKIP                  %% successful terminating process
    | STOP                  %% deadlock process
    | a -> P                %% action prefix
    | c ! v -> P            %% sending v over channel c (*)
    | c ? x : X -> P(x)     %% receiving x∈X on channel c (*)
    | c !! x : X -> P(x)    %% non-deterministic sending x∈X on c (*)
    | c !! x -> P(x)        %% non-deterministic sending x on c (*)
    | ? x : X -> P(x)       %% external prefix choice
    | ! x : X -> P(x)       %% internal prefix choice (*)
    | P [+] P               %% external choice
    | P |~| P               %% internal choice
    | ! x : X .. P(x)       %% replicated internal choice
    | IF b THEN P ELSE P    %% conditional
    | P |[X]| P             %% generalized parallel
    | P ||| P               %% interleaving (*)
    | P || P                %% synchronous parallel (*)
    | P -- X                %% hiding
    | P [[r]]               %% relational renaming
    | P ;; P                %% sequential composition
    | P [> P                %% (untimed) timeout (*)
    | <C>                   %% process name
```

Figure 4: Syntax of basic CSP processes in CSP-Prover.

notes a channel name, `v:'v` is a passed value, `b:bool` stands for a boolean value, `X:'a set` is a subset of `'a`, `r:('a * 'a) set` denotes a binary relation over communications, and `C:'n` is a process name for recursive behaviors. Derived operators are marked by (*).

The set of processes is encoded to a recursive type `('n,'a) proc` by the keyword **primrec** as shown in Figure 5, where `'n` and `'a` are types of process-names and communications, respectively. This means that structural induction over processes is is available in CSP-Prover by the command `induct_tac`. Operators involving bound variables such as `(? x : X -> P)` are defined by **syntax** and **translations** as shown in Figure 6. Derived operators such as sending and receiving values are also given as syntactic sugar by **syntax** and **translations** as follows:

- Sending a value: `(c ! v -> P)` sends a value `c` to a channel `c`, and thereafter behaves like `P`. It is a syntactic sugar of `((c v)-> P)`.
- Receiving values: `(c ? x : X -> P(x))` receives a value `v` from a channel `c` and thereafter behaves like `P(v)` if `v` $\in$ `X`. If `X` is the universe, `(: X)` can be omitted. They are defined as follows:

  $$\texttt{c ? x : X -> P(x)} \;=\; \texttt{? y : } \{\texttt{(c x) | x} \in \texttt{X}\} \texttt{ -> P(c}^{-1}\texttt{(y))}$$
  $$\texttt{c ? x -> P(x)} \;=\; \texttt{c ? x : UNIV -> P(x)}$$

- Non-deterministic Sending a value: `(c !! x : X -> P(x))` non-deterministically

```
datatype
 ('n,'a) proc
   = STOP
   | SKIP
   | Act_prefix       "'a" "('n,'a) proc"            ("_ -> _")
   | Ext_pre_choice   "'a set" "'a ⇒ ('n,'a) proc"   ("? : _ -> _")
   | Ext_choice       "('n,'a) proc" "('n,'a) proc"  ("(_ [+] _)")
   | Int_choice       "('n,'a) proc" "('n,'a) proc"  ("(_ |~| _)")
   | Rep_int_choice   "'a set" "'a ⇒ ('n,'a) proc"   ("! : _ .. _")
   | ...
   | Name             "'n"                           ("<_>")
```

Figure 5: The process type defined in Csp-Prover.

```
syntax
 "@Ext_pre_choice" ::
  "pttrn ⇒ 'a set ⇒ ('n,'a) proc ⇒ ('n,'a) proc" ("? _ : _ -> _")

 "@Rep_int_choice" ::
  "pttrn ⇒ 'a set ⇒ ('n,'a) proc ⇒ ('n,'a) proc" ("! _ : _ .. _")

translations
 "? x : X -> P == ? :X -> (λx. P)"
 "! x : X .. P == ! :X .. (λx. P)"
```

Figure 6: Operators involving bound variables.

sends a value `v` to a channel `c` such as `v` $\in$ `X`, and thereafter behaves like `P(v)`. If `X` is the universe, `(: X)` can be omitted. They are defined as follows:

$$\texttt{c !! x : X -> P(x)} \;=\; \texttt{! y : \{(c x) | x} \in \texttt{X\} -> P(c}^{-1}\texttt{(y))}$$
$$\texttt{c !! x -> P(x)} \;=\; \texttt{c !! x : UNIV -> P(x)}$$

- Internal Prefix choice: `(! x : X -> P(x))` requires that, for some `a` $\in$ `X`, an event `a` can be executed and thereafter it behaves like `P(a)`. It is defined as follows:

$$\texttt{! x : X -> P(x)} \;=\; \texttt{! x : X .. x -> P(x)}$$

- Replicated Internal choice with type-conversion: `(f !! x : X .. P(x))` requires that it behaves like `P(v)` for some `v` $\in$ `X`, where `f` is used as a type converter translating the type of v to the event type. If `X` is the universe, `(: X)` can be omitted. They are defined as follows:

$$\texttt{f !! x : X .. P(x)} \;=\; \texttt{! y : \{(f x) | x} \in \texttt{X\} .. P(f}^{-1}\texttt{(y))}$$
$$\texttt{f !! x -> P(x)} \;=\; \texttt{f !! x : UNIV .. P(x)}$$

Here, the short notations for `?` and `!` can be unfolded by

```
apply (simp add: csp_ss_def)
```

If you like to automatically unfold such short-notations, then you can use the declaration: `declare csp_ss_def[simp]`.

The other derived operators for timeout or parallelism are also given as follows:

- Timeout: `(P [> Q)` behaves like `P` for a short time before it opts to behave like `Q`. It is defined as follows:

  ```
  P [> Q  =  (P |~| STOP) [+] Q
  ```

- Synchronous Parallel: `(P || Q)` is a parallel composition, where every event must synchronize between `P` and `Q`. It is defined as follows:

  ```
  P || Q  =  (P |[UNIV]| Q)
  ```

- Interleaving: `(P ||| Q)` is a parallel composition, where `P` and `Q` have no communication. It is defined as follows:

  ```
  P ||| Q  =  (P |[{}]| Q)
  ```

In Csp, *recursive processes* are either defined by process equations or by so-called μ-recursion. Here, Csp-Prover currently offers only the former mechanism, and they take the form `(LET:fp df IN P)` as follows:

```
type ('n,'a) procDF = "'n ⇒ ('n,'a) proc"

datatype fp_type = Ufp | Lfp

datatype
  ('n,'a) procRC = Letin "fp_type" "('n,'a) procDF" "('n,'a) proc"
                                                  ("LET:_ _ IN _")
```

their type is `('n,'a) procRC`. Here, the function `df` binds process names to processes, it has the type `('n,'a) procDF`. And `fp` is a variable instantiated by either `Ufp` or `Lfp`, and specifies which fixed point of `df` is used for giving the meaning of process names: i.e. the unique fixed point by `Ufp` and the least fixed point by `Lfp`. Thus, intuitively, `(LET:fp df IN P)` behaves like the body process `P`, where each process name `C` in `P` behaves like a process `df C`. In the current Csp-Prover, `(LET df IN P)` is an abbreviation of `(LET:Ufp df IN P)`.

The most convenient way to define such function `df` is to use Isabelle's keyword **primrec** for defining recursive functions. For example, a process `Inc` which iteratively sends an increasing natural number `n` to a channel `c` is defined as follows:

```
primrec df       (Loop n) = c ! n -> <Loop (n+1)>
defs "Inc_def:        Inc == LET df IN <Loop 0>"
```

Such a parametrised process expressions can – on the semantical side of Csp – give rise to infinite systems of equations.

Example 6.1 *The recursive process* `Sys` *in* `Inc_nat` *used in Section 5 is defined as follows:*

```
datatype Event = Num nat | Read nat
datatype SysName = UI | VAR nat

consts
 SysDef :: "(SysName, Event) procDF"
primrec
   SysDef      (UI) = Read ? m -> Num ! m -> <UI>"
   SysDef   (VAR n) = Read ! n -> <VAR (Suc n)>"

consts
 Sys :: "(SysName, Event) procRC"
defs Sys_def:
 "Sys == LET SysDef
         IN (<UI> |[range Read]| <VAR 0>) -- (range Read)"
```

□

It is often required to replace each process-name `C` in a process `P` with `f(C)`. It is expressed as `(Rewrite P By f)`, where `P` and `f` have types `('n,'a) proc` and `'n => ('m,'a) proc`, respectively. Therefore, `(Rewrite P By f)` is a process, whose type is `('m,'a) proc`, obtained by replacing each process `<C>` with a process `f(C)`.

7 Trace

CSP has a special event `Tick` which represents a successful termination. So, the set of events consists of communications, whose type is `'a`, and `Tick` as follows:

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

Then, a trace is a list of events such that `Tick` does not occur except in the last place of the list. Furthermore, undefined element `None` is added to the type of traces because concatenations of two traces are not always traces (e.g. the concatenation of `[Tick]` and `[Tick]`). Therefore, the type of traces is defined as follows:

```
typedef 'a Some_trace = {ss::'a event list. Tick ∉ set (butlast ss)}
types   'a trace = "'a Some_trace option"
```

where the function `butlast` removes the last element of s and the function `set` transforms a list to a set of elements contained in the list. And, the type `'b option` consists of `Some 'b` and `None`.

Then, the type converters between `'a list` and `'a trace` are defined as follows:

```
consts
  Abs_trace = "'a event list => 'a trace"
  Rep_trace ::  "'a trace => 'a event list"

defs
  Abs_trace_def : "Abs_trace s == if s ∈ Some_trace
                                  then Some (Abs_Some_trace s)
                                  else None"
  Rep_trace_def : "Rep_trace s == Rep_Some_trace (the s)"
```

8 Domain

In the stable failures model, the behavior of each process is expressed by the set of traces that it can perform and a set of failures. A failure is a pair (t, X) of a trace t and a set X of refusal events. Intuitively, a failure (t, X) represents that events included in X cannot be performed after performing t. The type of failures is easily defined as follows:

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

In the current CSP-Prover, the domains of the stable-failures model $\mathcal{F}$ and the traces model $\mathcal{T}$ are instantiated as types `'a domSF` and `'a domT`, respectively, where `'a` is the type of communications. Here, the type `'a domT` can be reused for defining `'a domSF`:

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

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

```
types   'a domTF = "'a domT * 'a domF"
typedef 'a domSF = "{SF::('a domTF). HC_T2(SF) & HC_T3(SF)
                                     & HC_F3(SF) & HC_F4(SF)}"
```

where `HC_T1`, `HC_F2`, $\cdots$, `HC_F4` are predicates which exactly represents healthiness conditions given in [Ros98] and defined as follows:

$$
\begin{aligned}
\texttt{HC_T1(T)} &= \texttt{prefix_closed T \& T} \neq \{\} \texttt{ \& None} \notin \texttt{T} \\
\texttt{HC_F2(F)} &= (\forall \texttt{s X Y. ((s,X)} \in \texttt{F \& Y} \subseteq \texttt{X)} \rightarrow \texttt{(s,Y)} \in \texttt{F) \&} \\
&\quad\ (\forall \texttt{X. (None, X)} \notin \texttt{F)} \\
\texttt{HC_T2(SF)} &= \forall \texttt{s X. (s,X)} \in_{\texttt{f}} \texttt{snd SF} \rightarrow \texttt{s} \in_{\texttt{t}} \texttt{fst SF} \\
\texttt{HC_T3(SF)} &= \forall \texttt{s. s @}_{\texttt{t}} \texttt{[}\checkmark\texttt{]}_{\texttt{t}} \in_{\texttt{t}} \texttt{fst SF \& notick s} \\
&\quad\ \rightarrow (\forall \texttt{X. (s @}_{\texttt{t}} \texttt{[}\checkmark\texttt{]}_{\texttt{t}} \texttt{,X)} \in_{\texttt{f}} \texttt{snd SF)} \\
\texttt{HC_F3(SF)} &= \forall \texttt{s X Y. (s,X)} \in_{\texttt{f}} \texttt{snd SF \& notick s \&} \\
&\quad\ (\forall \texttt{a} \in \texttt{Y. s @}_{\texttt{t}} \texttt{[a]}_{\texttt{t}} \notin_{\texttt{t}} \texttt{fst SF)} \rightarrow \texttt{(s,X} \cup \texttt{Y)} \in_{\texttt{f}} \texttt{snd SF} \\
\texttt{HC_F4(SF)} &= \forall \texttt{s. s @}_{\texttt{t}} \texttt{[}\checkmark\texttt{]}_{\texttt{t}} \in_{\texttt{t}} \texttt{fst SF \& notick s} \\
&\quad\ \rightarrow \texttt{(s, Evset)} \in_{\texttt{f}} \texttt{snd SF}
\end{aligned}
$$

where `Evset` is `UNIV`$-\{\checkmark\}$ and subscripts `t` and `f` (e.g. in $\in_{\texttt{t}}$ and $\in_{\texttt{f}}$) are attached to operators on `domT` and `domF`, in order to avoid conflicts with the

operators on Isabelle's built-in types such as `list` and `set`.

Here, a pair constructor for `'a domSF` is given as follows:

```
consts
 pairSF ::  "'a domT => 'a domF => 'a domSF" ("(0_ ,, _)" [51,52] 0)
 fstSF  ::  "'a domSF => 'a domT"
 sndSF  ::  "'a domSF => 'a domT"

defs
 pairSF_def :  "(S,,F) == Abs_domSF (S, F)"
 fstSF_def  :  "fstSF == fst o Rep_domSF"
 sndSF_def  :  "sndSF == snd o Rep_domSF"
```

Intuitively, `(T ,, F)` requires `T` and `F` to satisfy healthiness conditions `T1` and `F2`, respectively, and the pair of them to satisfy `T2`, `T3`, `F3`, and `F4`.

Furthermore, in order to analyze infinite systems, the infinite product `('i,'a) domSF_prod` of `'a domSF` is defined as a synonym as follows:

```
types ('i,'a) domSF_prod = "'i => 'a domSF"
```

where the type `'i` represents the indexing set of the product space.

9 Semantics

The Csp semantics is defined by translating process-expressions into elements of the model $\mathcal{F}$ by a mapping $(\llbracket \mathtt{P} \rrbracket_{\mathtt{SF}}\ \mathtt{e})$ as shown in Fig. 7, where `e` is an evaluation function for process names in `P`. The mapping $(\llbracket \mathtt{P} \rrbracket_{\mathtt{SF}}\ \mathtt{e})$ is a pair of mappings $(\llbracket \mathtt{P} \rrbracket_{\mathtt{T}}\ \mathtt{e}\ ,,\ \llbracket \mathtt{P} \rrbracket_{\mathtt{F}}\ \mathtt{e})$. The mappings $(\llbracket \mathtt{P} \rrbracket_{\mathtt{T}}\ \mathtt{e})$ and $(\llbracket \mathtt{P} \rrbracket_{\mathtt{F}}\ \mathtt{e})$ are *recursively* defined by the same semantic clauses of the model $\mathcal{F}$ in [Ros98]. Furthermore, the meaning $\llbracket \mathtt{df} \rrbracket_{\mathtt{DF}}$ of each defining function is defined such that the meaning of each process name `C` is $\llbracket \mathtt{df\ C} \rrbracket_{\mathtt{SF}}$. Finally, the meaning $\llbracket \mathtt{LET:fp\ df\ IN\ P} \rrbracket_{\mathtt{RC}}$ of each recursive process is defined by $\llbracket \mathtt{P} \rrbracket_{\mathtt{SF}}$, where the meaning of each process name in `P` is given by a suitable fixed point of $\llbracket \mathtt{df} \rrbracket_{\mathtt{DF}}$.

(the definition of all the operators will be written.)

10 Verification

You can verify the refinement relation `<=F` (whose X-symbol is $\sqsubseteq$`F`) and the equivalence relation `=F`, which are defined as follows, based on the stable failures model in the current Csp-Prover 2005.

$$\begin{aligned} (\mathtt{R1\ <=F\ R2}) &= (\llbracket \mathtt{R2} \rrbracket_{\mathtt{RC}} \subseteq \llbracket \mathtt{R1} \rrbracket_{\mathtt{RC}}) \\ (\mathtt{R1\ =F\ R2}) &= (\llbracket \mathtt{R2} \rrbracket_{\mathtt{RC}} = \llbracket \mathtt{R1} \rrbracket_{\mathtt{RC}}) \end{aligned}$$

```
consts
  evalT  ::  "'a proc ⇒ ('n,'a) domSF_prod ⇒ 'a domT"  ("⟦_⟧T")
  evalF  ::  "'a proc ⇒ ('n,'a) domSF_prod ⇒ 'a domF"  ("⟦_⟧F")
  evalSF ::  "'a proc ⇒ ('n,'a) domSF_prod ⇒ 'a domSF" ("⟦_⟧SF")

primrec
  "⟦STOP⟧T          = (λe. {[]t}t)"
  "⟦SKIP⟧T          = (λe. {[]t, [✓]t}t)"
  "⟦a -> P⟧T        = (λe. {t. t=[]t ∨ (∃s. t=[Ev a]t @t s ∧ s ∈t ⟦P⟧T e) }t)"
  "⟦? : X -> Pf⟧T   = (λe. {t. t=[]t ∨ (∃a s. t=[Ev a]t @t s ∧ s ∈t ⟦Pf a⟧T e ∧ a∈X) }t)"
                        ⋮
  "⟦<C>⟧T           = (λe. fstSF (e C))"            (* note: fstSF (T ,, F) = T *)

primrec
  "⟦STOP⟧F          = (λe. {f. ∃X. f=([]t, X) }f)
  "⟦SKIP⟧F          = (λe. {f. (∃X. f=([]t, X) ∧ X ⊆ Evset) ∨ (∃X. f=([✓]t, X)) }f)
  "⟦a -> P⟧F        = (λe. {f. (∃X. f=([]t, X) ∧ Ev a ∉ X) ∨
                              (∃s X. f=([Ev a]t @t s, X) ∧ (s, X) ∈f ⟦P⟧F e) }f)
  "⟦? : X -> Pf⟧F   = (λe. {f. (∃Y. f=([]t, Y) ∧ Ev`X ∩ Y = {}) ∨
                              (∃a s Y. f=([Ev a]t @t s, Y) ∧ (s, Y)∈f⟦Pf a⟧F e) ∧ a∈X) }f)
                        ⋮
  "⟦<C>⟧F           = (λe. sndSF (e C))"            (* note: sndSF (T ,, F) = F *)

defs evalSF_def :
  "⟦P⟧SF == (λe. (⟦P⟧T e ,, ⟦P⟧F e))"

consts
  evalDF ::  "('n⇒('m,'a) proc)⇒('m,'a) domSF_prod ⇒('n,'a) domSF_prod"   ("⟦_⟧DF")
  evalRC ::  "('n,'a) procRC⇒'a domSF"                                     ("⟦_⟧RC")

defs evalDF_def :
  "⟦df⟧DF == (λe. (λC. (⟦df C⟧SF e)))"

recdef evalRC "measure (λx. 0)"
  "⟦LET:Ufp df IN P⟧RC = ⟦P⟧SF (UFP ⟦df⟧DF)"          (* based on cms *)
  "⟦LET:Lfp df IN P⟧RC = ⟦P⟧SF (LFP ⟦df⟧DF)"          (* based on cpo *)
```

Figure 7: The mapping from each process to a domain

Csp-Prover gives many Csp-rules for verifying the relations by rewriting Csp-expressions. You can use these Csp-rules in Isabelle by loading the main theory `CSP_Prover`, for example, as follows

```
theory T = CSP_Prover:
```

This means that your theory `T` will be proven by `CSP-Prover`.

In Csp-Prover, proofs mainly consist of three phases: (1) unfolding recursive processes, (2) expanding processes to head-normal-forms (hnf), and (3) decomposing them. These are explained in the rest of this section.

10.1 Fixed point induction based on cms

It is hard to verify `(LET df1 IN P1) <=F (LET df2 IN P2)` only by rewriting `P1` and `P2` because they are different processes names defined `df1` and `df2`, respectively. This problem is solved by applying fixed point induction. Therefore,

`(LET:fp df1 IN P1) <=F (LET:fp df2 IN P2)` can be verified by proving the following subgoals for both cases `fp=Ufp` and `fp=Lfp`:

```
1. !!C. nohide (df1 C)
2. !!C. guard (df1 C)
3. !!C. nohide (df2 C)
4. !!C. guard (df2 C)
5. LET df2 IN (Rewrite P1 By f12) <=F LET df2 IN P2
6. !!C. LET df2 IN (Rewrite (df1 C) By f12) <=F LET df2 IN (f12 C)
```

where `f12` is a function which takes a process-name in `df1` and returns a process-expression containing process-names defined by `df2`, such that for each process name `C`, `f12(C)` refines `C`. Such function will be given by users because it is hard to automatically find such function `f12` from `df1` and `df2`, but Csp-Prover will be able to assist they to find it.

It is important to note that expressions of the form `(LET df1 IN ...)` is not contained in the subgoals. It allows us to verify the refinement between recursive processes containing different process-names.

For example, the function `Spc_to_Sys` (an instance of `f12` above) which relates `SpcDef` to `SysDef` is defined as follows

```
primrec
  "Spc_to_Sys (Cspc n)
      = (<UI> |[range Read]| <VAR n>) -- (range Read)"
```

in the example `Example/Inc_nat` (also see Section 5 and Example 6.1). Then, when a goal is given as follows:

```
Spc <=F Sys
```

and the following command is applied:

```
apply (rule csp_fp_induct_cms[of _ _ _ "Spc_to_Sys"])
```

then the following subgoals are returned:

```
goal (lemma, 6 subgoals):
 1. !!C. nohide (SpcDef C)
 2. !!C. guard (SpcDef C)
 3. !!C. nohide (SysDef C)
 4. !!C. guard (SysDef C)
 5.
    LET SysDef IN Rewrite (<Cspc 0>) By Spc_to_Sys <=F
    LET SysDef IN (<UI> |[range Read]| <VAR 0>) -- range Read
 6. !!C.
         LET SysDef IN Rewrite (SpcDef C) By Spc_to_Sys <=F
         LET SysDef IN Spc_to_Sys C
```

The rule `csp_fp_induct_cms` can be applied to verification on equality such as `(LET df1 IN P1) =F (LET df2 IN P2)`. Furthermore, Csp-Prover provides

a symmetric rule with `(csp_fp_induct_cms)`. Therefore, in order to verify a refinement `(LET:fp df1 IN P1) <=F (LET:fp df2 IN P2)` , the symmetric rule `(csp_fp_induct_rev)` infers the following subgoals:

```
1. !!C. nohide (df1 C)
2. !!C. guard (df1 C)
3. !!C. nohide (df2 C)
4. !!C. guard (df2 C)
5. LET df1 IN P1 <=F LET df1 IN (Rewrite P2 By f21)
6. !!C. LET df1 IN (f21 C) <=F LET df1 IN (Rewrite (df2 C) By f21)
```

where `f21` is a function which takes a process-name in `df2` and returns a process-expression containing process-names defined by `df1`.

10.2 Fixed point induction based on cpo

Csp-Prover also provides a fixed point induction rule `(csp_fp_induct_cpo)` based on cpo for verification on refinement with least fixed points. By the rule, `(LET:Lfp df1 IN P1) <=F (LET:Lfp df2 IN P2)` can be verified by proving the following subgoals:

```
1. LET:Lfp df1 IN P1 <=F LET:Lfp df1 IN (Rewrite P2 By f21)
2. !!C. LET:Lfp df1 IN (f21 C) <=F
        LET:Lfp df1 IN (Rewrite (df2 C) By f21)
```

This rule is similar to the rule `(csp_fp_induct_rev)` based on cms, but the rule based on cpo does not require processes to be guarded and has no hiding. This induction rule is demonstrated by the example `Test_Lfp.thy` in the directory

```
CSP-Prover2005-2/src/Examples
```

Since the least fixed point is used in cpo, the rule `(csp_fp_induct_cpo)` cannot be applied to verification on equality and there is no symmetric rule such as `(csp_fp_induct_cms)`. Users can select a fixed point induction rule according to situations.

10.3 Expanding

To verify `(LET df1 IN P1) <=F (LET df2 IN P2)`, it is useful to transform `P1` and `P2` to head-normal-forms (hnf) such as `(? x:X -> P)` or `(P |~| Q)` or `(! x:X .. P)`. To do that, rewriting laws called *step laws* (e.g. see P.32 (1.14) in [Ros98]) and *distributive laws* are given in Csp. Csp-Prover gives tactics based on the laws for getting hnfs. The most powerful tactic is "`csp_hnf_tac`". This tactic applies step laws Csp-expressions which are unguarded (by Prefix or Prefix choice) because of avoiding excessive expanding, which makes expressions to be unreadable. User defined tactics are applied in Isar-mode as follows:

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

where 1 represents that this tactic is applied to the first subgoal.

The tactic csp_hnf_tac contains the following smaller tactics, and they can be individually applied for reducing proof cost.

- csp_unwind_tac is a tactic for unwinding process-names, if every definitions of process-names are guarded and have no hiding.

  For example, assume that a sub-goal is given as follows:

  ```
  Spc <=F
  LET SysDef IN (<UI> |[range Read]| <VAR 0>) -- range Read
  ```

  where Spc and SysDef are defined in Inc_nat (also see Example 6.1). Now, apply the tactic as follows:

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

  Then, the sub-goal will be rewritten to the following new sub-goal:

  ```
  Spc <=F
  LET SysDef
  IN (Read ? m -> Num m -> <UI>
       |[range Read]| Read ! 0 -> <VAR (Suc 0)>)
      -- range Read
  ```

  As shown this result, the unguarded process-names UI and VAR are replaced with their process-expressions defined by SysDef. It is important that the process-names in the new sub-goal are not unfolded because they are guarded. This technique works for avoiding infinite rewriting.

  Note: before applying this tactic, it should be proven that every definitions of process-names are guarded and have no hiding. They are easily proven, for example, by

  ```
  lemma guardSysDef[simp]: "!!C. guard (SysDef C)"
  by (induct_tac C, simp_all)
  ```

  but they are not automatically proven by (auto).

- csp_step_tac is a tactic for transforming processes of the form STOP, a -> P, P [+] Q, P |[X]| Q, P -- X, P [[r]], or P ;; Q, to processes (hnfs) of the form ? x:A -> P', used as follows:

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

  csp_light_step_tac is a tactic for transforming processes of the form STOP, or a -> P, to processes (hnfs) of the form ? x:A -> P'. Since the proof cost of csp_step_tac is often high, csp_light_step_tac is sometimes used instead of csp_step_tac.

- `csp_dist_tac` is a tactic for distributing unguarded operators over internal choices and replicated internal choices.
- `csp_simp_tac` is a tactic for simplify processes by mainly evaluating conditions. Simplification rules can be manually added or deleted as follows:

  ```
  apply (tactic {* csp_simp_tac 1 *})
  apply (tactic {* csp_simp_add_tac "name1" 1 *})
  apply (tactic {* csp_simp_del_tac "name2" 1 *})
  apply (tactic {* csp_simp_add_del_tac "name1" "name2" 1 *})
  ```

  where `name1` and `name2` are theory-names added to and deleted from simplification rules, respectively. Do not forget to convert the theory-names to strings by double-quotations.
- `csp_rule_tac` is a tactic for applying an introduction rule to sub-expressions as follows:

  ```
  apply (tactic {* csp_rule_tac "name" 1 *})
  ```

  where `name` is a name of introduction rule.
- `csp_asm_tac` is a tactic for applying assumptions.

10.4 Decomposition

It is possible to decompose process-expressions to sub-expressions and verify the sub-expressions because of their congruency and monotonicity.

- `csp_rm_head` is a rule for removing the same head of head-normal forms like

  ```
  LET df IN ? x:X -> P =F LET df IN ? x:Y -> Q
  ```

  Since this is added to (automatically applied) introduction rules, it can be easily applied as follows:

  ```
  apply (rule)
  ```

  Then, the above sub-goal is decomposed to 2 subgoal:

  ```
  1. X = Y
  2. !!a. a : Y ==> LET df IN P =F LET df IN Q
  ```

  The first goal may be solved by Isabelle set-theory and a tactic like `csp_hnf_tac` can be applied to the second goal again, to transforms them to hnfs.
- `csp_decompo` is more powerful rule than `csp_rm_head`, and it decomposes expressions if their outermost operators are the same. For example, the goal

  ```
  LET df IN P1 [+] P2 =F LET df IN Q1 [+] Q2
  ```

is decomposed the two sub-goals

```
1. LET df IN P1 =F LET df IN Q1
2. LET df IN P2 =F LET df IN Q2
```

by the proof command

```
apply (rule csp_decompo)
```

However, this rule is unsafe because unexpected decomposition can be done. For example, do not apply this rule to the goal

```
   LET df IN a -> SKIP [+] b -> SKIP
=F LET df IN b -> SKIP [+] a -> SKIP
```

- `csp_free_decompo` is a rule for decomposing expressions if they are not the forms of `a -> P`, `? x:X -> P`, `! x:X .. P`, and `IF b THEN P ELSE Q`, thus only unguarded expressions (by events or conditions) can be decomposed. This rule can be used for avoiding excessive decompositions, which are often caused by `csp_decompo`.
- `csp_decompo_subset` is a rule for removing replicated internal choices. Thus, the goal

  ```
  LET df IN ! x:X .. P(x) <=F LET df IN ! x:Y .. Q(x)
  ```

  is rewritten to the following sub-gaols

  ```
  1. Y <= X
  2. !!a. a:Y ==> LET df IN P(a) <=F LET df IN Q(a)
  ```

  by the proof command

  ```
  apply (rule csp_decompo_subset)
  ```

  This rule also works for the expressions whose operators are internal and external prefix choices as follows:

  ```
  LET df IN ! x:X .. x -> P(x) <=F LET df IN ? x:Y -> Q(x)
  ```

  This goal is rewritten to the following sub-gaols by this rule:

  ```
  1. Y ~={}
  2. Y <= X
  3. !!a. a:Y ==> LET df IN P(a) <=F LET df IN Q(a)
  ```

10.5 Internal choice

By distributive laws, unguarded internal choices can be outermost. Then, the outermost internal choices can be removed by the following rules.

- `csp_Int_choice_right` is a rule for removing internal choices. Since this rule is safe, it is added to (automatically applied) introduction rules [intro!]. For example, the goal

  ```
  LET df IN P <=F LET df IN Q1 |~| Q2
  ```

  is rewritten to the following sub-gaols

  ```
  1. LET df IN P <=F LET df IN Q1
  2. LET df IN P <=F LET df IN Q2
  ```

  by the proof command `apply (rule)`.

- `csp_Rep_int_choice_right` is a rule for removing replicated internal choices. This rule is similar to `csp_Int_choice_right`. For example, the goal

  ```
  LET df IN P <=F LET df IN ! x:X .. (Q x)
  ```

  is rewritten to the following sub-gaol

  ```
  1. !!a. a : X ==> LET df IN P <=F LET df IN (Q a)
  ```

  by the proof command `apply (rule)`.

- Internal choices on the left hand side have to be carefully removed because users have to choice a suitable rule from the following three rules:

  - `csp_Int_choice_left1` : for choosing `P` from `P |~| Q`.
  - `csp_Int_choice_left2` : for choosing `Q` from `P |~| Q`.
  - `csp_Int_choice_left3` : for choosing both of `P` and `Q` from `P |~| Q`.

  For example, the goal

  ```
  LET df IN P1 |~| P2 <=F LET df IN Q
  ```

  is rewritten to the following sub-gaol

  ```
  1. LET df IN P1 <=F LET df IN Q
  ```

  by the proof command

  ```
  apply (rule csp_Int_choice_left1)
  ```

  and it is rewritten to the following sub-gaol

  ```
  1. LET df IN P1 [+] P2 <=F LET df IN Q
  ```

  by the proof command

  ```
  apply (rule csp_Int_choice_left3)
  ```

- `csp_Rep_int_choice_left` is a rule for replacing replicated internal choices on the left hand side with existential quantifiers. For example, the goal

  ```
  LET df IN ! x:X .. (P x) <=F LET df IN Q
  ```

is rewritten to the following sub-gaol

```
1. EX a. a : X & LET df IN P a <=F LET df IN Q
```

by the proof command:

```
apply (rule csp_Rep_int_choice_left)
```

Sorry. This document has not been completed yet.

References

[Asp00] David Aspinall. Proof general: A generic tool for proof development. In Susanne Graf and Michael Schwartzbach, editors, *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.

[Hoa85] Charles Antony Richard Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.