User Guide Csp-Prover2004

CSP-Prover Document
Version: DRAFT October 22, 2004

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

Contents

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-Prover2004-2.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-Prover2004-2
```

by an unpacking command (e.g. `tar zxvf Csp-Prover2004-2.tar.gz`).

Figure 1 shows the contents of Csp-Prover2004-2. 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.

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

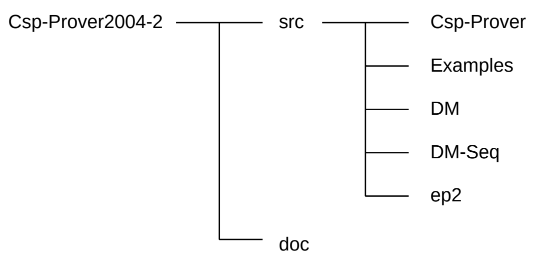


Figure 1: The directory tree of Csp-Prover2004-2

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-Prover2004-2/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, 7 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-Prover2004-2/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

   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.

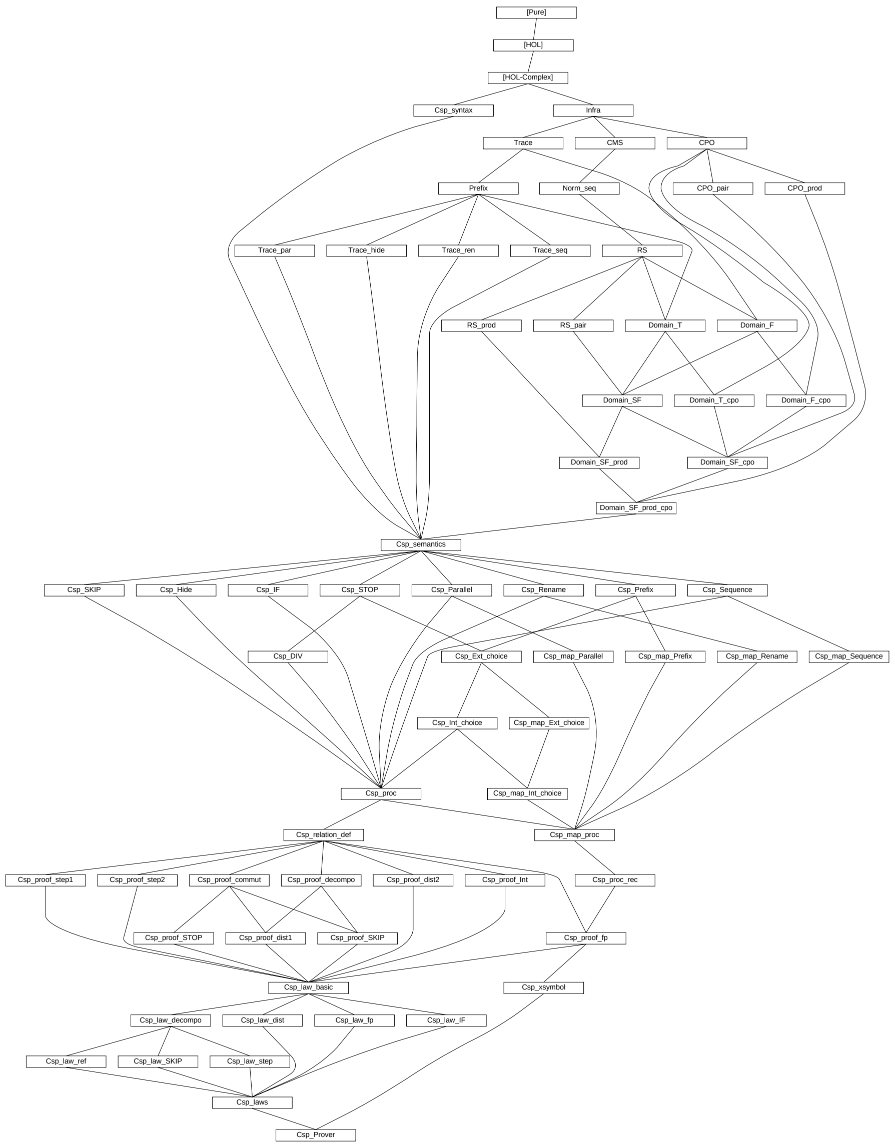


Figure 2: The dependency-graph for CSP-Prover

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] $\rightarrow$ [Logics] $\rightarrow$ [Csp-Prover].

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 □ 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-Prover2004-2/src/Examples/Inc_nat.thy
   ```

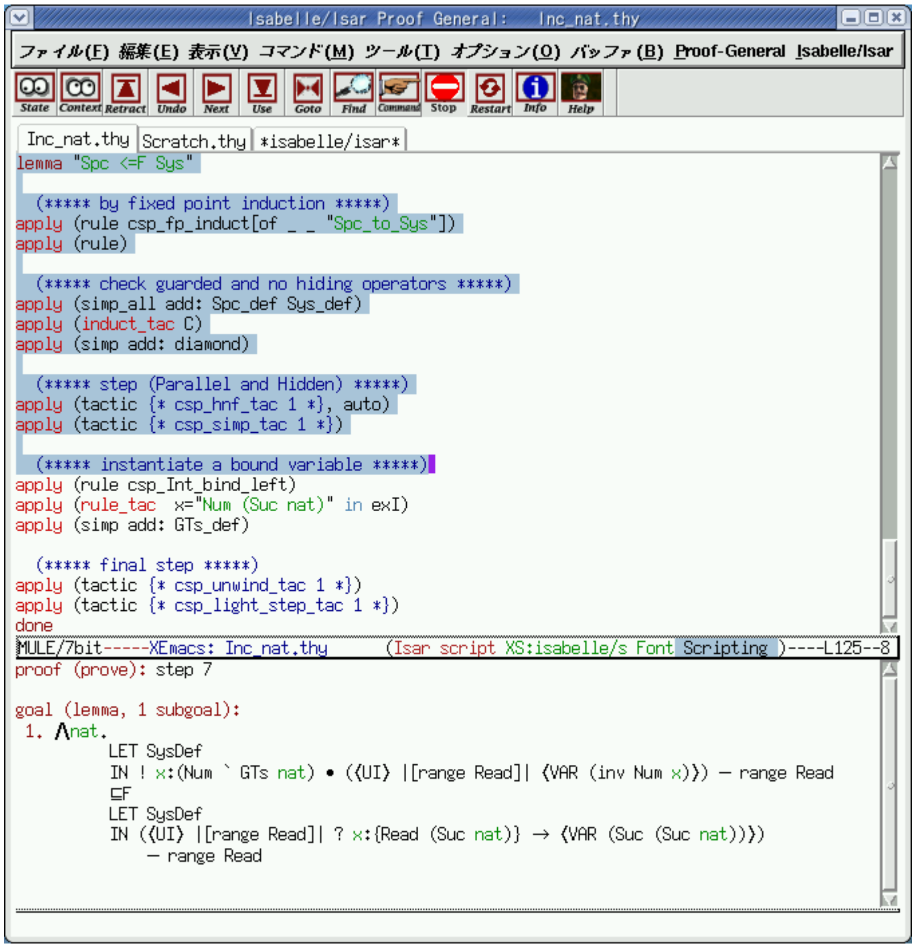
3. Copy the commands from "`Inc_nat.thy`" and paste them to the isabelle window line by line until the proof finishes.

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:


Isabelle/Isar Proof General: Inc_nat.thy
ファイル(F) 編集(E) 表示(V) コマンド(M) ツール(T) オプション(O) バッファ(B) Proof-General Isabelle/Isar
State
Context
Retract
Undo
Next
Use
Goto
Find
Command
Stop
Restart
Info
Help
Inc_nat.thy
Scratch.thy
isabelle/isar

```
lemma "Spc <=F Sys"

  (***** by fixed point induction *****)
apply (rule csp_fp_induct[of _ _ "Spc_to_Sys"])
apply (rule)

  (***** check guarded and no hiding operators *****)
apply (simp_all add: Spc_def Sys_def)
apply (induct_tac C)
apply (simp add: diamond)

  (***** step (Parallel and Hidden) *****)
apply (tactic {* csp_hnf_tac 1 *}, auto)
apply (tactic {* csp_simp_tac 1 *})

  (***** instantiate a bound variable *****)
apply (rule csp_Int_bind_left)
apply (rule_tac  x="Num (Suc nat)" in exI)
apply (simp add: GTs_def)

  (***** final step *****)
apply (tactic {* csp_unwind_tac 1 *})
apply (tactic {* csp_light_step_tac 1 *})
done
MULE/7bit-----XEmacs: Inc_nat.thy       (Isar script XS:isabelle/s Font Scripting )----L125--8
proof (prove): step 7

goal (lemma, 1 subgoal):
 1. ⋀nat.
       LET SysDef
       IN ! x:(Num ` GTs nat) • ({UI} |[range Read]| {VAR (inv Num x)}) — range Read
       ⊑F
       LET SysDef
       IN ({UI} |[range Read]| ? x:{Read (Suc nat)} → {VAR (Suc (Suc nat))})
          — range Read
```

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

```
/usr/local/Csp-Prover2004-2/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-Prover2004-2/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
```

```
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.

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)` denotes 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 introduced as syntactic sugar:

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

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

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

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

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

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:

```
c ? x : X -> P(x)  =  ? y : {(c x) | x ∈ X} -> P(c⁻¹(y))
    c ? x -> P(x)  =  c ? x : UNIV -> P(x)
```

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

```
c !! x : X -> P(x)  =  ! y : {(c x) | x ∈ X} -> P(c⁻¹(y))
    c !! x -> P(x)  =  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:

```
! x : X -> P(x)  =  ! 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` ∈ `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:

  ```
  f !! x : X .. P(x)  =  ! y : {(f x) | x ∈ X} .. P(f⁻¹(y))
      f !! x -> P(x)  =  f !! x : UNIV .. P(x)
  ```

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 `LET df IN P` is defined as follows:

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

datatype
 ('n,'a) procRec
  = Letin "('n,'a) procDef" "('n,'a) proc"   ("LET _ IN _")
```

Intuitively, `LET df IN P` behaves like the body process `P`, where each process name `C` in `P` behaves like a process `df(C)`. The most convenient way to define the function 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) procDef"
primrec
   SysDef      (UI) = Read ? m -> Num ! m -> <UI>"
   SysDef   (VAR n) = Read ! n -> <VAR (Suc n)>"

consts
 Sys :: "(SysName, Event) procRec"
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 $\widehat{\mathrm{P}}$ 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 Domain

Csp has a special event `Tick` which represents a successful termination. So, the (extended) set of events (communications) consists of user-defined events, whose type is `'a`, and `Tick` as follows:

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

In the stable failures model, the behavior of each process is expressed by the order of events that it can perform and a set of refusal events at each state. The order of events is represented by a sequence (i.e. trace) of events defined as follows:

```
typedef 'a trace = "s::('a event) list. Tick : set(butlast s)"
```

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. Therefore, `Tick` does not occur in every trace except the last of the trace.

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"
```

Finally, the set of failures components is given as a type `'a dom_SF` in CSP-Prover. The type `'a dom_SF` and the (infinite) product `('i,'a) dom_SF_prod` are defined from the set `'a dom_T` of traces and the set `'a dom_F` of failures as follows:

```
typedef 'a dom_T = "{T::('a trace set).  CT1(T)}"
```

```
typedef 'a dom_F = "{F::('a failure set).  CF1(F)}"
```

```
types   'a dom_TF = "'a dom_T * 'a dom_F"
typedef 'a dom_SF = "{SF::('a dom_TF). CT2(SF) & CF2(SF) & CF3(SF)}"
```

```
types ('i,'a) dom_SF_prod = "'i => 'a dom_SF"
```

where the type `'i` represents the indexing set of the product space, which is used for infinite systems of equations, and the conditions `CT1`, `CF1`, $\cdots$, `CF3` are defined as follows:

```
 CT1(T) = prefix_closed T & T ≠ {}
 CF1(F) = ∀s X Y. (s,X) ∈ F & Y ⊆ X → (s,Y) ∈ F
CT2(SF) = ∀s.  s @t [✓]t ∈ fst SF & notick s
           → (s,UNIV−{✓}) ∈ snd SF & (s @t [✓]t ,UNIV) ∈ snd SF
CF2(SF) = ∀s X Y. (s,X) ∈ snd SF & notick s & (∀a∈ Y. s @t [a]t ∉ fst SF)
           → (s,X ∪ Y) ∈ snd SF
CF3(SF) = ∀ s X. (s,X) ∈ snd SF → s ∈ fst SF.
```

The set of traces and the set of failures are extracted from a domain on $\mathcal{F}$ by the following functions `Tof` and `Fof`, respectively.

```
consts
  Tof ::  "'a dom_SF => 'a dom_T" "Tof SF == fst (Rep_dom_SF SF)"
  Fof ::  "'a dom_SF => 'a dom_F" "Fof SF == snd (Rep_dom_SF SF)"
```

where `Rep_dom_SF` is a function which converts the type of `'a dom_SF` into `'a dom_TF`.

8 Semantics

The semantics is given by translating process-expressions in the model $\mathcal{F}$ step by step as shown in Fig.6. At first, each process "`P::('n,'a) proc`" is translated to a function "$[\![\mathtt{P}]\!]_{\mathtt{TF}}$`::('n,'a) dom_SF_prod⇒'a dom_TF`" which is *recursively* defined on the structure of `P` by the keyword **primrec**. The recursive definition of $[\![\mathtt{P}]\!]_{\mathtt{TF}}$ exactly complies with the semantic clause of the model $\mathcal{F}$, where the functions `STOPsf`, `SKIPsf`, `->sf`, `EXTCHsf`, and so on are used just for improving readability, for example, `->sf`, and `EXTCHsf` are defined as shown in Fig.7. (the definition of all the operators are written soon.)

```
consts
  eval_TF :: "('n,'a) proc⇒('n,'a) dom_SF_prod⇒'a dom_TF" ("⟦_⟧TF")
  eval_SF :: "('n,'a) proc⇒('n,'a) dom_SF_prod⇒'a dom_SF" ("⟦_⟧SF")

primrec
  "⟦STOP⟧TF        = (λe.  STOPsf)"
  "⟦SKIP⟧TF        = (λe.  SKIPsf)"
  "⟦a -> P⟧TF      = (λe.  (a ->sf (⟦P⟧TF e)))"
  "⟦? :X -> Pf⟧TF = (λe.  EXTCHsf{a ->sf ⟦Pf a⟧TF e |a.  a∈X}∪{STOPsf})"
  "⟦P [+] Q⟧TF     = (λe.  EXTCHsf{⟦P⟧TF e, ⟦Q⟧TF e})"
  "⟦P |~| Q⟧TF     = (λe.  INTCHsf{⟦P⟧TF e, ⟦Q⟧TF e})"
  "⟦! :X .. Pf⟧TF = (λe.  INTCHsf{⟦Pf a⟧TF e |a.  a∈X}∪{DIVsf}))"
                          ⋮
         (the other operators are written soon)
                          ⋮
  "⟦<C>⟧TF         = (λe.  e C)"

defs eval_SF_def :  "⟦P⟧SF == (λe.  Abs_dom_SF(⟦P⟧TF e))"

consts
  eval_DF :: "('n⇒('m,'a) proc)⇒('m,'a) dom_SF_prod ⇒('n,'a) dom_SF_prod"   ("⟦_⟧DF")
  eval_RC :: "('n,'a) procRec⇒'a dom_SF"                                      ("⟦_⟧RC")

defs eval_DF_def :  "⟦df⟧DF == (λe.  (λC. (⟦df C⟧SF e)))"
primrec "⟦LET df IN P⟧RC = ⟦P⟧SF (UFP ⟦df⟧DF)"
```

Figure 6: The mapping from each process to a domain

```
consts
   Prefixt  :: "'a⇒'a trace set⇒'a trace set"     ("_ ->t _")
   Prefixf  :: "'a⇒'a failure set⇒'a failure set" ("_ ->f _")
   Prefixsf :: "'a⇒'a dom_TF⇒'a dom_TF"           ("_ ->sf _")
defs
   Prefixt_def  "a ->t T    == insert []t {[Ev a]t @t s |s.  s:  T"}
   Prefixf_def  "a ->f F    == {([]t,X) |X. Ev a ∉ X} ∪
                               {([Ev a]t @t s,X) |s X. (s,X) ∈ F}"
   Prefixsf_def "a ->sf TF  == (Abs_dom_T (a ->t Rep_dom_T (fst TF)),
                                Abs_dom_F (a ->f Rep_dom_F (snd TF)))"
consts
   Ext_choicef  :: "'a trace set set⇒'a failure set set ⇒'a failure set"  ("EXTCHf _ _")
   Ext_choicesf :: "'a dom_TF set⇒'a dom_TF"                                ("EXTCHsf _")
defs
   Ext_choicef_def
     "EXTCHf Ts Fs  == {([]t,X) |X. ([]t,X) : Inter Fs} ∪
                       {(s,X) |s X. s ≠ []t & (s,X)∈Union Fs} ∪
                       {([]t,X) |X. ✓ ∉X & [✓]t ∈ Union Ts}"
   Ext_choicesf_def
     "EXTCHsf TFs == (Abs_dom_T (Union (Rep_dom_T ‘ fst ‘ TFs)),
                     Abs_dom_F (EXTCHf (Rep_dom_T ‘ fst ‘ TFs) (Rep_dom_F ‘ snd ‘ TFs)))"
```

Figure 7: The meaning of Csp-operators

In Fig.6, $[\![\mathtt{P}]\!]_{\mathtt{SF}}$ is the same as $[\![\mathtt{P}]\!]_{\mathtt{TF}}$ except that the type is converted by `Abs_dom_SF`. The translation $[\![\_]\!]_{\mathtt{SF}}$ over "`('n,'a) proc`" is extended to $[\![\_]\!]_{\mathtt{DF}}$ over "`'m⇒('n,'a) proc`". The meaning of "`LET df IN P`" is given by `P`, where the meaning of process-names in `P` given by `df`. The meaning of `df` is defined by fixed points, for example, a process name `A` defined as `(df A = a -> A)` is a process `x` such that `(x =F a -> x)`, where `=F` is the equality in the model $\mathcal{F}$. In the current version of Csp-Prover, the unique fixed point `UFP` is used based on the Banach's fixed point theorem. The function `UFP` is a partial function, and it returns the fixed point of a function `f` if `f` has a unique fixed point, else it returns `(the None)` which represents "undefined". This is a usual technique using the `option` type to define a partial function in Isabelle.

9 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 2004.

$$\begin{aligned} \texttt{R1 <=F R2} &= \texttt{[[R2]]RC} \subseteq \texttt{[[R1]]RC} \\ \texttt{R1 =F R2} &= \texttt{[[R2]]RC} = \texttt{[[R1]]RC} \end{aligned}$$

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.

9.1 Fixed point induction

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, we can verify `(LET df1 IN P1) <=F (LET df2 IN P2)` by proving

```
  CHECK (LET df1 IN P1) <=F (LET df2 IN P2) BY f12
```

where it is defined as follows:

```
 CHECK (LET df1 IN P1) <=F (LET df2 IN P2) BY f12
 =
 (∀C. (nohide (df1 C))) ∧ (∀C. (guard (df1 C))) ∧
 (∀C. (nohide (df2 C))) ∧ (∀C. (guard (df2 C))) ∧
 (LET df2 IN (Rewrite P1 By f12)  <=F LET df2 IN P2 ∧
 (∀C. LET df2 IN (Rewrite (df1 C) By f12) C <=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`. It is hard to automatically find such function `f12` from `df1` and `df2`. Therefore, such function will be given by users.

It is important to note that the definition of "`CHECK ...`" contains only process-names defined by `df2` because each process-name `C` defined by `df1` is replaced with `f12(C)`. 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[of _ _ "Spc_to_Sys"])
```

then the following subgoal is returned:

```
CHECK Spc <=F Sys BY Spc_to_Sys
```

This means that we can prove "`CHECK Spc <=F Sys BY Spc_to_Sys`" instead of "`Spc <=F Sys`". The expression "`CHECK ...`" can be unfolded by a command `apply (unfold CHECKref_def)`, however the rule for unfolding it is added to introduction rules which are automatically applied. Thus, just apply the following command,

```
apply (rule)
```

then 6 sub-goals will be displayed in according to the definition of `"CHECK ..."` as follows:

```
goal (lemma (check_ex1), 6 subgoals):
 1. !!C. nohide (LetD Spc C)
 2. !!C. guard (LetD Spc C)
 3. !!C. nohide (LetD Sys C)
 4. !!C. guard (LetD Sys C)
 5.
    LET LetD Sys IN (Rewrite (InP Spc) By Spc_to_Sys) <=F
    LET LetD Sys IN InP Sys
 6. !!C.
      LET LetD Sys IN (Rewrite ((LetD Spc) C) By Spc_to_Sys) <=F
      LET LetD Sys IN Spc_to_Sys C
```

where `LetD R` and `LetP R` are defined as follows:

```
LetD (LET df IN P)  =  df
LetP (LET df IN P)  =  P
```

This strategy (by `csp_fp_induct`) is also available for verification based on equivalence relation `=F`.

9.2 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 -> P1'` and `? x:Y -> P2'`. To do that, rewriting laws called *step laws* (e.g. see P.32 (1.14) in [Ros98]). are given in Csp. Csp-Prover gives tactics based on step 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), unbounded (by Internal bind), and unconditional 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 small tactics, and they can be individually applied for reducing proof cost.

- `csp_unwind_tac` is a tactic for unfolding 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-goad will be rewritten to the following new sub-goal:

  ```
      Spc <=F
      LET SysDef
      IN (? x:range Read -> Num (inv Read x) -> <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`, where short notations for `??` and `!!` are automatically unfolded. 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 internal binds.

- `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.

9.3 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 internal binds. 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 -> 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)
```

9.4 Internal choice and bind

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.