# A Generic Theorem Prover of CSP Refinement*

Yoshinao Isobe[1] and Markus Roggenbach[2]

[1] AIST, Japan
y-isobe@aist.go.jp
[2] University of Wales Swansea, United Kingdom
M.Roggenbach@Swan.ac.uk

**Abstract.** 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 for infinite state systems, which may also involve infinite non-determinism. Semantically, Csp-Prover supports both the theory of complete metric spaces as well as the theory of complete partial orders. Both these theories are implemented for infinite product spaces. Technically, Csp-Prover is based on the theorem prover Isabelle. It provides a deep encoding of Csp. The tool's architecture follows a generic approach which makes it easy to adapt it for various Csp models besides those studied here: the stable failures model $\mathcal{F}$ and the traces model $\mathcal{T}$.

## 1 Introduction

Among the various frameworks for the description and modelling of reactive systems, process algebra plays a prominent rôle. It has proved to be suitable at the level of requirement specification, at the level of design specifications, and also for formal refinement proofs [2]. In this context, the process algebra Csp [11, 21] has successfully been applied in various areas, ranging from train control systems [5] over software for the international space station [3, 4] to the verification of security protocols [23].

Concerning tool support for Csp, the model checker FDR [15] is without doubt the standard proof tool for Csp. It allows for refinement proofs as well as for deadlock and livelock analysis. However, in general FDR restricts Csp specifications to finite state systems[1] and allows only the use of concrete data types. Furthermore, in practical applications it is often hard to deal with the state explosion problem. In this context, the use of theorem provers has been suggested e.g. by [26, 25, 8, 24] in order to complement the well-established technique of model checking.

* Supported by Royal Society with Short Visit Grants.

[1] On the LHS of a refinement check, see [22] for a precise characterisation of the possible infinite state processes on the RHS.

N. Halbwachs and L. Zuck (Eds.): TACAS 2005, LNCS 3440, pp. 108–123, 2005.

In this paper we describe a new tool Csp-Prover. Its generic architecture makes it suitable for various denotational Csp models. The implemented theories of complete metric spaces and complete partial orders allow Csp-Prover to deal with infinite state systems with unbounded non-determinism. Using the theorem prover Isabelle [16], Csp-Prover can also analyse specifications in Csp based on abstract data types. We demonstrate the relevance of these properties by proofs in the context of an industrial case study.

The paper is organised as follows: First, we describe the theorem prover Isabelle and give a short overview on the process algebra Csp. Then, the generic architecture of Csp-Prover is discussed in detail. Sect. 5 demonstrates how to apply Csp-Prover in various settings. Finally, we relate Csp-Prover to similar tools.

## 2 The Theorem Prover Isabelle

Isabelle [18] is an interactive theorem prover. Theorems to be proved are entered as *goals*. A goal can be manipulated by proof-commands referring to a set of predefined inference *rules* producing new goals. Such rules can be combined to form *proof tactics*. A proof is completed, if by application of rules and tactics the only open goal is the truth value `True`. Successfully proved theorems can be stored and used later as new rules.

To extend an existing logic, Isabelle offers mechanisms to define new types, functions, predicates etc. The keyword **typedef** defines a new type as a non-empty subset of an existing type:

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

Here, `P` is a predicate over the existing type `SuperType`, and `SubType` is the newly defined type by the subset. In addition, the keyword **datatype** is used for recursive type definitions with type-constructors, for example,

```
datatype 'a list = Nil  |  Cons 'a "'a list"
```

where `Nil` and `Cons` are type constructors, and `'a` is a type variable. Type classes can be defined by

```
axclass SubClass < SuperClass
  name1: (a condition)   name2: (a condition) ···
```

where `SubClass` is included in `SuperClass` and contains only types which satisfy conditions named $\texttt{name}_{1,2,\ldots}$. Everything in `SuperClass` is inherited by `SubClass`. Another possibility to use this inheritance is to declare that a type forms an instance of `SubClass` by the keyword **instance**. Such a declaration requires a proof that the type satisfies the conditions of `SubClass` and all its super classes.

Theorems, together with definitions and proof-commands needed for their proof, can be stored in *theory-files*. Isabelle organises such files in a rule-database, to which other theory files may refer. Such a theory-file generally has the format

**theory** $\mathtt{T} = \mathtt{B}_1 + \cdots + \mathtt{B}_n$: *declarations, definitions, and proofs* **end**

where $\mathtt{B}_1, \cdots, \mathtt{B}_n$ are names of *parent theories* of theory T. Everything used in parent theories is available in their children. This allows for a hierarchical organisation of theory-files.

## 3 The CSP Dialect Chosen for CSP-Prover

This section briefly summarises Csp syntax, Csp semantics and how to analyse process equations in Csp, following closely [21]. It also gives a first overview on what has been implemented in Csp-Prover.

```
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 channel c (*)
    | c !! x -> P(x)          %% non-deterministic sending x on channel 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
```

**Fig. 1.** Syntax of basic Csp processes in Csp-Prover

### 3.1 Syntax

The process algebra Csp [11, 21] 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. Fig. 1 shows that the Csp dialect implemented by Csp-Prover covers all these features[2]. 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

[2] The syntactical differences to Csp-M, as e.g. in the sequential composition P ;; P, avoid overloading of symbols also used by Isabelle.

for a boolean value, `X:'a set` is a subset of `'a`, and `r:('a * 'a) set` denotes a binary relation over communications. Derived operators are marked by (*) . In Csp, *recursive processes* are either defined by process equations or by so-called $\mu$-recursion. Here, Csp-Prover currently offers only the former mechanism. Each recursive process has the form `LET df IN P`, where the body process `P` can contain process names whose behaviours are defined by fixed points of the function `df`. The most convenient way to define this 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 parametrised process expressions can – on the semantical side of Csp – give rise to infinite systems of equations.

### 3.2 Semantics

Csp is a language with many semantics, different in their style as well as in their ability to distinguish between different reactive behaviour. There are operational, denotational and algebraic approaches, ranging from the simple finite traces model $\mathcal{T}$ to such complex semantics as the infinite traces model with failures/divergences $\mathcal{U}$. For a general theorem prover on Csp the denotational semantics are of special interest. Even under the restriction to finitely nondeterministic Csp the algebraic approach does not work quite cleanly for the three main models: the traces model $\mathcal{T}$, the failure-divergence model $\mathcal{N}$, and the stable failures model $\mathcal{F}$ (see [21] for the details).

The current prototype of Csp-Prover concentrates on the denotational stable-failures semantics $\mathcal{F}$: This semantics allows for analysis of deadlock-freedom and of liveness properties (for which the traces model $\mathcal{T}$ is too weak). Furthermore, the semantic domain of $\mathcal{F}$ is a complete metric space (cms) as well as complete partial order (cpo) even in the case of infinite alphabets[3]. For recursively defined processes, both approaches, cms and cpo, allow to prove the existence of solutions and to analyse these solutions by powerful induction principles.

Given a set of communications $A$, the domain of the *stable failures model* $\mathcal{F}$ is a set of pairs $(T, F)$ satisfying certain healthiness conditions, where $T \subseteq A^{*\checkmark}$ and $F \subseteq A^{*\checkmark} \times \mathbb{P}(A^{\checkmark})$[4]. In such a pair $(T, F)$, $T$ is the set of traces a process can execute, while the elements $(s, X) \in F$ describe sets of communications $X$ which a process can fail to accept after executing the trace $s$. The healthiness conditions state e.g. that the sets $T$ need to be non-empty and prefix-closed,

[3] The semantic domain of the failure-divergence model $\mathcal{N}$ is not a cpo on refinement order for infinite alphabets; however it is a cms independent of the alphabet size. Another problem is that in $\mathcal{N}$ the semantics clauses for hiding work only under special conditions.

[4] $A^{\checkmark} := A \cup \{\checkmark\}$, $A^{*\checkmark} := A^* \cup \{s \frown \langle\checkmark\rangle \mid s \in A^*\}$.

that a trace occurring in $F$ needs to be a trace in $T$, that after termination a process may refuse to engage in any events.

Typical examples of the semantic clauses of $\mathcal{F}$ are:

$$
\begin{aligned}
\mathit{traces}(\texttt{STOP}) &= \{\langle\rangle\} \\
\mathit{failures}(\texttt{STOP}) &= \{(\langle\rangle, X) \mid X \subseteq A^{\checkmark}\} \\
\mathit{traces}(\texttt{a -> P}) &= \{\langle\rangle\} \cup \{\langle \texttt{a}\rangle \frown s \mid s \in \mathit{traces}(\texttt{P})\} \\
\mathit{failures}(\texttt{a -> P}) &= \{(\langle\rangle, X) \mid \texttt{a} \notin X\} \cup \{(\langle \texttt{a}\rangle \frown s, X) \mid (s, X) \in \mathit{failures}(\texttt{P})\}
\end{aligned}
$$

Our implementation uses the traces semantics and the stable-failures semantics as they are defined in [21]. As Isabelle allows only for consistent theories, our encoding can also be seen as a proof for the well-formedness of these semantics[5].

### 3.3 Analysing CSP Recursion

Consider the recursive equations in Csp defined by the following functions:

```
primrec df1    (P) = a -> <P> |~| b -> SKIP
primrec df2    (Q) = a -> <Q>
```

For such equations the natural questions are: (1) Do there exist solutions for `P` and `Q` in, say, the Csp model $\mathcal{F}$? (2) Are these solutions uniquely determined? (3) How to prove properties on these solutions, e.g. that `Q` refines `P`? To deal with these questions, Csp employs two different techniques: complete metric spaces (cms) and complete partial orders (cpo) which are both implemented in Csp-Prover. These two approaches follow a similar pattern: the first step consists of proving that the domain of a given Csp model is a cms or a cpo, respectively. As a particularity of Csp, metric spaces are introduced in terms of so-called restriction spaces. The second step consists of proving that the various Csp operators satisfy the pre-requisite properties, namely contractiveness for cms and continuity for cpo. Finally, a fixed point theorem is used to deal with question (1). In the case of cms this is Banach's theorem, while it is Tarski's theorem within the cpo approach. For question (2) Banach's theorem leads to a unique fixed point, while Tarski's theorem does not guarantee uniqueness. Here, the least fixed point is chosen in the Csp models $\mathcal{T}$ and $\mathcal{F}$. To answer question (3) both the cms and the cpo approach offer as a technique the so-called fixed point induction.

Up to now the described framework works only for one *single equation*. But how about an *infinite system of equations* like the recursive process `Inc` illustrated in Sect. 3.1? For such infinite systems, infinite products of cms and cpo, respectively, are required. Furthermore, the pre-requisite properties of the fixed point theorems need to be proved only on the base of component functions.

[5] In an earlier encoding of Csp in Isabelle [26] it was necessary to correct an up-to-then established Csp semantics.

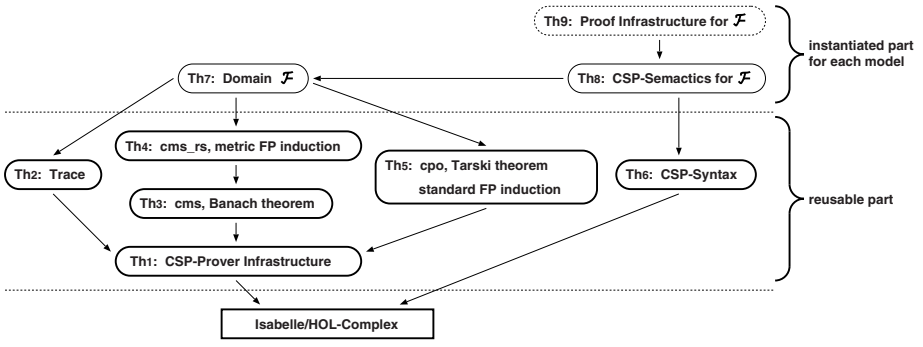


**Fig. 2.** The theory map of Csp-Prover instantiated with the stable-failures model $\mathcal{F}$

## 4 A Generic Theorem Prover for CSP Refinement

Csp-Prover extends the Isabelle [18] theory HOL-Complex (which is HOL [16] extended with a definitional development of the real and complex numbers) by a hierarchy of theory-files encoding Csp, see Fig. 2. The prototype discussed here supports the stable-failures model $\mathcal{F}$ as well as the traces model $\mathcal{T}$. Csp-Prover has a generic architecture divided into a large *reusable part* $\mathtt{Th}_{1,\ldots,6}$ independent of specific Csp models and an *instantiated part* $\mathtt{Th}_{7,8,9}$ for each specific Csp model.

The reusable part contains Banach's fixed point theorem and the metric fixed point induction rule based on complete metric spaces (cms) as well as Tarski's fixed point theorem and the standard fixed point induction rule based on complete partial orders (cpo). Furthermore, it provides infinite product constructions for these spaces. Thus, when Csp-Prover is instantiated with the domain of a new Csp model, the fixed point theorems, the induction rules, as well as the product constructions are available for free, provided the domain is a cms or a cpo. Additionally, the reusable part provides guidelines in form of proof obligations on how to show that a domain is a cms or a cpo. Here, especially the theory on restriction spaces plays an important rôle for proofs concerning cms.

Another contribution of the reusable part is the definition of the Csp syntax as a recursive type. Here, instantiating Csp-Prover with a new model requires to provide its semantic clauses defined inductively over this type. This means that the syntax is *deeply encoded*, thus structural induction on processes is supported.

### 4.1 Reusable Part

The reusable part consists of a theory of traces ($\mathtt{Th}_2$), fixed point theories on cms and cpo ($\mathtt{Th}_{3,4,5}$), the definitions of the Csp syntax ($\mathtt{Th}_6$), and fundamental theorems on limits and least upper bounds ($\mathtt{Th}_1$). In this Section, we concentrate on how to encode cms and restriction spaces. Trace theory is similar to the data

type of lists[6]. The discussion of the syntax definition is postponed to Sect. 4.3, where it is considered in the context of semantic clauses. For modelling cpos in Isabelle we refer to [26, 25].

In the theory file $\mathtt{Th}_3$, first the class of metric spaces is defined as a type-class `ms` which satisfies conditions of diagonal, symmetry, and triangle inequality. Next, the class of complete metric spaces is defined as a type-subclass `cms` of the class `ms` by adding the completeness condition `complete_ms`, which requires every Cauchy sequence `xs` to converge to some point `x`:

```
axclass cms < ms
    complete_ms: "∀xs. cauchy xs⟶(∃x. xs convergeTo x)"
```

Then, Banach's fixed point theorem is proved, i.e. that any contraction map `f` over `cms` has a unique fixed point and the sequence obtained by iteratively applying `f` to any value `x0` converges to the fixed point.

```
theorem Banach_thm: "contraction (f::('a::cms⇒'a))
    ⟹ (f hasUFP ∧ (λn. (f^n) x0) convergeTo (UFP f))"
```

A way for deriving a metric space from a restriction space is given in [21]. Thus, if a space is an instance of the class `rs` of restriction spaces then the space is also an instance of `ms_rs` which is the multiple-inheritance from `ms` and `rs`. An important result on `ms_rs` is that the completeness of `ms_rs` is preserved by the constructors `*` (binary product) and `fun` (function space). For example, if type `T` is an instance of `cms_rs`, then the function type `I` ⇒ `T` is also an instance of `cms_rs` for an arbitrary type `I`. This theorem is expressed in Isabelle by

```
instance fun :: (type,cms_rs) cms_rs
```

The function type `I` ⇒ `T1` is used to deal with infinite product spaces whose index set is `I`, which are required to deal with infinite state systems (see Sect. 3.3). Take for example the Csp model $\mathcal{F}$. Here we need that $\mathcal{F}^{\mathtt{I}}$ is a cms for any infinite index set `I`, which is intuitively the set of (infinitely many) process names. The above property expresses: if $\mathcal{F}$ is an instance of `cms_rs` then $\mathcal{F}^{\mathtt{I}}$ is also an instance of `cms_rs`.

Finally, the following metric fixed point induction rule on `cms_rs` is proved (see [21] on continuity and constructiveness for restriction spaces).

```
theorem cms_fixpoint_induction: "[| (R::'a::cms_rs⇒bool) x ;
    continuous_rs R ; constructive_rs f ; inductivefun R f |]
      ⟹ f hasUFP ∧ R (UFP f)"
```

[6] The event type of Csp consists of communications `(Ev a)` and the event ✓ for successful termination. The trace type is defined as the subset of event lists such that ✓ cannot occur except in the last place of a list.

### 4.2 Instantiated Part

The instantiated part consists of instantiated domain theories ($\mathtt{Th}_7$ in Fig. 2), semantic clauses ($\mathtt{Th}_8$), and a proof infrastructure ($\mathtt{Th}_9$). The proof infrastructure contains many CSP laws such as step laws, distributive laws, fixed-point induction rules, etc. Furthermore, it provides a powerful tactic `csp_hnf_tac` for translating any expression to a head normal form. As these rules and tactics are proved in Isabelle, they are guaranteed to be sound with respect to the chosen CSP semantics.

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)}"
types 'a failure = "'a trace * 'a event set"
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`, $\cdots$, `HC_F4` are predicates which exactly represent the healthiness conditions `T1`, $\cdots$, `F4` given in [21].

In order to apply Banach's theorem and the metric induction rule to the model $\mathcal{F}$, it is required to prove that the infinite product `('i,'a) domSF_prod` of `'a domSF` is an instance of `cms_rs`, where `('i,'a) domSF_prod` is a synonym of `('i ⇒ 'a domSF)` and the type `'i` represents the indexing set of the product space. This is proved as follows: (1) `domT` and `domF` are instances of `cms_rs`, (2) `domTF` is also an instance of `cms_rs`, thus there exists a limit of each Cauchy sequence in `domSF` ($\subset$ `domTF`), (3) the limit is contained in `domSF`, thus `domSF` is also an instance of `cms_rs`, and (4) `domSF_prod` is an instance of `cms_rs`. Here, the proofs of (2) and (4) follow by preservation of `cms_rs` under the constructors `*` and `fun` as mentioned in Sect. 4.1. This example shows how the provided infrastructure in terms of restriction spaces discharges certain proof obligations when a new CSP model is integrated in CSP-Prover.

### 4.3 Deep Encoding

The CSP syntax is defined as a recursive type `('n,'a) proc` by the command **datatype** as shown in Fig. 3, where `'n` and `'a` are type variables for process-names and communications, respectively. This syntax encoding style implies that structural induction over processes is available by the Isabelle's proof command `induct_tac`. Recursive processes take the form `LET:fp df IN P`, 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`. In the current CSP-Prover, `LET df IN P` is an abbreviation of `LET:Ufp df IN P`.

```
datatype ('n,'a) proc = STOP
                      | SKIP
                      | Act_prefix "'a" "('n,'a) proc"     ("_ -> _")
                      | ...
                      | Proc_name  "'n"                    ("<_>")

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 _")
```

**Fig. 3.** Syntax definition of processes

```
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)"
                      ⋮
  "⟦<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)
                      ⋮
  "⟦<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 *)
```

**Fig. 4.** Semantics definition of processes

The Csp semantics is defined by translating process-expressions into elements of the model $\mathcal{F}$ by a mapping (⟦P⟧SF e) as shown in Fig. 4, where e is an evaluation function for process names in P. The mapping (⟦P⟧SF e) is a pair of mappings (⟦P⟧T e ,, ⟦P⟧F e), where (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. The mappings (⟦P⟧T e) and (⟦P⟧F e) are defined by the

same semantic clauses of the model $\mathcal{F}$ in [21], where subscripts `t` and `f` (e.g. in $[]_t$ and $\in_f$) 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`. Furthermore, the meaning $[\![\texttt{df}]\!]_{\texttt{DF}}$ of each defining function is defined such that the meaning of each process name `C` is $[\![\texttt{df C}]\!]_{\texttt{SF}}$. Finally, the meaning $[\![\texttt{LET:fp df IN P}]\!]_{\texttt{RC}}$ of each recursive process is defined by $[\![\texttt{P}]\!]_{\texttt{SF}}$, where the meaning of each process name in `P` is given by a suitable fixed point of $[\![\texttt{df}]\!]_{\texttt{DF}}$.

## 5 Applications

In this section we demonstrate how CSP-Prover can be used for the verification of reactive systems. First, we discuss deadlock analysis and a refinement proof in the context of an industrial case study. Then we study a mutual exclusion problem arising in the classical example of the dining mathematicians.

### 5.1 Verification in the Context of an Industrial Case Study

The EP2 system[7] is a new industrial standard of electronic payment systems. It consists of seven autonomous entities centred around the EP2 *Terminal*: Cardholder (i.e., customer), Point of Service (i.e., cash register), Attendant, POS Management System, Acquirer, Service Center, and Card, see Fig. 5. These en-

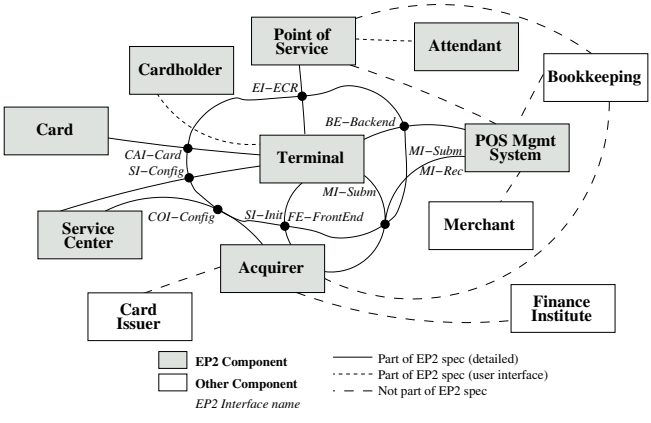


**Fig. 5.** Overview of the EP2 System

tities communicate with the Terminal and, to a certain extent, with one another via *XML-messages* in a fixed format.

[7] EP2 is a joint project established by a number of (mainly Swiss) financial institutes and companies in order to define infrastructure for credit, debit, and electronic purse terminals in Switzerland (www.eftpos2000.ch).

```
1  (* data part *)
2  typedecl init_d      typedecl request_d
3  typedecl response_d  typedecl exit_d
4  datatype Data = Init init_d | Exit exit_d | Request request_d | Response response_d
5  datatype Event = c Data
6
7  (* process part *)
8  datatype ACName = Acquirer | AcConfM | Terminal | TerminalConfM
9  consts ACDef :: "(ACName, Event) procDF"
10 primrec
11   "ACDef (Terminal) = c !! init:(range Init) -> <TerminalConfM>"
12   "ACDef (TerminalConfM) =
13      c ? x -> IF (x:range Request)
14               THEN c !! response:(range Response) -> <TerminalConfM>
15               ELSE IF (x:range Exit) THEN SKIP ELSE STOP"
16   "ACDef (Acquirer) = c ? x:(range Init) -> <AcConfM>"
17   "ACDef (AcConfM) =
18      c !! exit:(range Exit) -> SKIP |~|
19      c !! request:(range Request) -> c ? response:(range Response) -> <AcConfM>"
20
21 constdefs AC :: "(ACName, Event) procRC"
22   "AC == LET ACDef IN (<Acquirer> |[range c]| <Terminal>)"
```

**Fig. 6.** EP2 Specification at the Abstract Component Description Level

In [9], major parts of the EP2 system have been formalised in Csp-Casl [20]. Following the structure of the original EP2 documents, the specifications presented in [9] can be classified to be e.g. on the Architectural Level, on the Abstract Component Description Level, or on the Concrete Component Description Level. In this context, tool support is needed to prove deadlock freedom for the interaction between the various EP2 components.

Translating the data part of the specifications given in [9] into adequate Isabelle code, we obtain specifications in the input format of Csp-Prover. Fig. 6 shows the nucleus[8] of the initialisation procedure of the EP2 `Terminal` at the Abstract Component Description Level. The `Terminal` starts the initialisation (line 11) and waits then for data sent by the `Acquirer`. If this data is of type `Request`, the `Terminal` answers with a value of type `Response` (line 14). Another possibility is that the `Acquirer` wants to exit the initialisation (line 15). Any other type of communication sent by the `Acquirer` will lead to a deadlock represented by the process `STOP` (line 15). On the other end of the communication, after receiving an initialisation request (line 16) the `Acquirer` internally decides if it wants to exit the process (line 18) or interact with the `Terminal` by sending a request followed by a response of the `Terminal` (line 19). The system `AC` to be analysed here consists of the parallel composition of the `Terminal` and the `Acquirer` synchronised on the channel `c` (line 22).

It is the defining characteristic of the Abstract Component Description Level that the data involved is *loosely* specified. No specific values are defined (lines 2–5). Semantically this means that – depending on the interpretation of e.g. the type `init_d` – the described systems might involve infinite non-determinism,

[8] For the purpose of this paper, the specification text has been simplified. The complete formalisation and proof can be found in [12].

```
 1  datatype AbsName = Abstract | Loop
 2  consts AbsDef :: "(AbsName, Event) procDF"
 3  primrec
 4    "AbsDef (Abstract) = c !! init:(range Init) -> <Loop>"
 5    "AbsDef (Loop) =
 6       c !! exit:(range Exit) -> SKIP |~|
 7       c !! request:(range Request) -> c !! response:(range Response) -> <Loop>"
 8
 9  constdefs Abs :: "(AbsName, Event) procRC"
10    "Abs == LET AbsDef IN <Abstract>"
```

**Fig. 7.** An abstraction of the process shown in Fig. 6

e.g. if the type `init_d` has infinitely many values, the `Terminal` process of Fig. 6 chooses internally between sending any of these values (line 11). Thus, Csp-Prover has simultaneously to deal with a *class* of specifications: it has to prove that a certain property holds for any possible interpretation of the types involved.

Using Csp-Prover, we can show the above described process `AC` to be stable-failure equivalent to the process `Abs` of Fig. 7. Note that `Abs` is a sequential, i.e. by syntactic characterisation deadlock-free process. As stable failure equivalence preserves deadlocks, establishing this equivalence proves that the interaction of `Terminal` and `Acquirer` on the Abstract Component Description Level is deadlock free[9]. Fig. 8 shows the complete script to prove the stable-failure equivalence `Abs =F AC` (line 14) in Csp-Prover. First, a mapping is defined from the process-names of `Abs` to process expressions in `AC` (line 3-5)[10]. Next, it is shown that the involved recursive processes are guarded and do not use the hiding operator. This is fully automated routine (lines 8–11). After these preparations, `Abs =F AC` is given as a goal (line 14). Using the above mapping, now the recursive processes are unfolded to a base case and step cases by fixed point induction (line 16). Since a step case is produced for each of the process names of `Abs`, the step cases are instantiated by induction on `AbsName` (line 17). Finally, the theorem is proven by Isabelle's tactic `auto`, Csp-Prover's tactic `csp_hnf_tac`, which transforms any expression into a head normal form, and `csp_decompo`, which decomposes Csp-operators (line 18).

### 5.2 The Dining Mathematicians

The dining mathematicians [7] are a classical mutual exclusion problem: There are two mathematicians living at the same place, whose life is focused on two activities, namely thinking (`TH0` and `TH1`, respectively) and eating (`EAT0` and `EAT1`, respectively). As they have a strong dislike for each other, they want never to eat at the same time. To ensure this, they agreed to the following protocol. They both have access to a common variable (`VAR n`) storing integer

[9] In this example, abstraction is convenient to establish deadlock-freedom; in general, Csp-Prover is capable to support e.g. the various deadlock rules stated in [21].

[10] It is hard to automatically derive such correspondences. However, Csp-Prover can assist users to derive them.

```
(* expected correspondence of process-names in Abs to AC *)
consts Abs_to_AC :: "AbsName ⇒ (ACName, Event) proc"
primrec
  "Abs_to_AC (Abstract) = (<Acquirer> |[range c]| <Terminal>)"
  "Abs_to_AC     (Loop) = (<AcConfM> |[range c]| <TerminalConfM>)"

(* guarded and no hiding operator *)
lemma guard_nohide[simp]:
  "!! C. guard (ACDef C) & nohide (ACDef C)"
  "!! C. guard (AbsDef C) & nohide (AbsDef C)"
by (induct_tac C, simp_all, induct_tac C, simp_all)

(* the main theorem *)
theorem ep2: "Abs =F AC"
apply (unfold Abs_def AC_def)
apply (rule csp_fp_induct_cms[of _ _ _ "Abs_to_AC"], simp_all)
apply (induct_tac C)
by (auto simp add: image_iff | tactic {* csp_hnf_tac 1 *} | rule csp_decompo)+
```

**Fig. 8.** The complete proof script for `AC =F Abs`

```
datatype Event = Eat0 | Back0 | End0 | RD0 int | WR0 int
               | Eat1 | Back1 | End1 | RD1 int | WR1 int | NUM int
syntax       "_CH0" :: "Event set" ("CH0")         "_CH1" :: "Event set" ("CH1")
translations "CH0" == "(range RD0) ∪ (range WR0)"  "CH1" == "(range RD1) ∪ (range WR1)"

datatype SysName = VAR int | TH0 | EAT0 int | TH1 | EAT1 int
consts SysDef :: "(SysName, Event) procDF"
primrec
  "SysDef    (TH0) = RD0 ? n -> IF (EVEN n) THEN Eat0 -> <EAT0 n> ELSE Back0 -> <TH0>"
  "SysDef    (TH1) = RD1 ? n -> IF (ODD n) THEN Eat1 -> <EAT1 n> ELSE Back1 -> <TH1>"
  "SysDef (EAT0 n) = End0 -> WR0 ! (n div 2) -> <TH0>
  "SysDef (EAT1 n) = End1 -> WR1 ! (3 * n + 1) -> <TH1>
  "SysDef  (VAR n) =   WR0 ? n -> <VAR n> [+] WR1 ? n -> <VAR n>
                     [+] RD0 ! n -> <VAR n> [+] RD1 ! n -> <VAR n>"
constdefs Sys :: "int ⇒ (SysName, Event) procRC"
  "Sys == (λn. LET SysDef IN (<TH0> |[CH0]| <VAR n> |[CH1]| <TH1>) -- (CH0 ∪ CH1))"
```

**Fig. 9.** The dining mathematicians: Csp-Prover description of the concrete system

values. If the stored integer (`n`) is even, the first mathematician is allowed to start eating. When finished, the first mathematician sets the stored value to `(n/2)`. A similar procedure holds for the second mathematician, where the check is if the value of the stored variable is odd, and the value written back after eating is `(3n+1)`[11]. Fig. 9 shows this system described in Csp-Prover. Here, each of the process definitions `(EAT0 n)`, `(EAT1 n)`, and `(VAR n)` describes infinitely many equations. The question is: does this now precisely described system exclude the situation where both mathematicians eat together? Or, on a more formal level: has this system a trace where `Eat1` appears between consecutive communications `Eat0` and `End0` (or vice versa)?

The classical argument in analysing this system is to provide an abstraction of the dining mathematicians which clearly has the desired exclusion property. This

[11] The function involved here is the so-called *Collatz function* which is studied in the context of the $3x + 1$ problem, see [13] for a survey.

abstraction `Spc` consists only of three states, which stand for the situations 'both mathematicians think' `TH0_TH1` and 'one mathematician eats while the other is thinking' (`EAT0_TH1` and `TH0_EAT1`, respectively). With Csp-Prover we can show that `(Sys n)` is a stable-failures refinement of `Spc` for any integer `n`, thus "`ALL n. Spc <=F Sys n`". The respective proof script is substantially longer than the proof of `Abs =F AC` shown in Sec. 5.1. But it follows the same strategy: First, the goal is unfolded by fixed point induction. In a second step the resulting proof obligations are translated to head normal forms and automatically discharged. The details are omitted here, the full script as well as the abstraction `Spc` are available at [12]. As `Spc` is again a sequential process, this refinement result also establishes deadlock-freedom of `(Sys n)`.

## 6 Related Work

Based on general purpose theorem provers like Isabelle [18], HOL [10] or PVS [17], various tools for theorem proving over process algebras have been presented.

Closest to our approach are the Csp encodings of Tej/Wolff [26, 25] and Schneider/Dutertre [8, 24]. *Tej/Wolff* suggest a shallow encoding of Csp in Isabelle/HOL based on the cpo approach. Their encoding HOL-CSP is focused on the failure-divergence model of Csp. To deal with recursion it introduces a new process order that implies the standard refinement order. HOL-CSP lacks the possibility of proofs on the syntactic process structure. Thus, powerful tactics as `csp_hnf_tac` in Csp-Prover for transforming process expressions to head normal form are not available. *Schneider/Dutertre*'s encoding of the Csp traces model $\mathcal{T}$ in PVS is tailored to the verification of security protocols. Semantically it uses the cpo approach. It does not consider process termination. Due to its clear focus, refinement proofs of the nature shown in the previous section are out of its scope. Compared to these two encodings, a major advantage of Csp-Prover is its genericity. It is easy to adapt Csp-Prover to any other denotational Csp model. Furthermore, in offering both, the cms and the cpo approach, it allows to use the more convenient and the more promising setting for any proof step.

Alternative to encoding a denotational semantics, [6, 19, 1] base their encodings on an axiomatic semantics of the process algebra. As discussed in Sect. 3.2, such an approach is not an option in the context of Csp.

## 7 Conclusion and Future Work

We have shown a new tool Csp-Prover which supports refinement proofs over various Csp models. Thanks to its powerful semi-automatic and automatic tactics, Csp-Prover has successfully been applied in an industrial case study as well as for a complex example tailored to be a benchmark for refinement proofs.

In the future, we intend to include the failure-divergence model $\mathcal{N}$ in Csp-Prover. Furthermore, we will integrate Csp-Prover with the model checker FDR. In this context the theory of data independence, see e.g. [14], will play an important rôle. Continuing the work on EP2 and applying Csp-Prover to other case

studies, e.g. of the area of train control systems, will help to develop more proof infrastructure to further automate refinement proofs.

## Acknowledgement

The authors would like to thank AIST (Japan) and the Royal Society (UK) for financial support; Faron G. Moller and Kazuhito Ohmaki for initiating our cooperation; Erwin R. Catesbeiana (jr.) for advice on semantical questions; as well as Christoph Lüth, Ranko Lazic, Jan Peleska, Bill Roscoe, and Holger Schlingloff for valuable feedback and advice on our tool.

## References

1. T. Basten and J. Hooman. Process algebra in PVS. In W. Cleaveland, editor, *TACAS'99*, LNCS 1579, pages 270–284. Springer, 1999.
2. J. Bergstra, A. Ponse, and S. Smolka. *Handbook of Process Algebra*. Elsevier, 2001.
3. B. Buth, M. Kouvaras, J. Peleska, and H. Shi. Deadlock analysis for a fault-tolerant system. In *AMAST'97*, LNCS 1349, pages 60–75. Springer, 1997.
4. B. Buth, J. Peleska, and H. Shi. Combining methods for the livelock analysis of a fault-tolerant system. In A. M. Haeberer, editor, *AMAST'98*, LNCS 1548, pages 124–139. Springer, 1998.
5. B. Buth and M. Schrönen. Model-checking the architectural design of a fail-safe communication system for railway interlocking systems. In J. M. Wing, J. Woodcock, and J. Davies, editors, *FM'99*, LNCS 1709. Springer, 1999.
6. A. Camilleri. Combining interaction and automation in process algebra verification. In G. Goos and J. Hartmanis, editors, *TAPSOFT 1991*, LNCS 494, pages 283–295. Springer, 1991.
7. E. M. Clarke and H. Schlingloff. Model checking. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier, 2001.
8. B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In E. L. Gunter and A. Felty, editors, *TPHOL 1997*, LNCS 1275, pages 121–136. Springer, 1997.
9. A. Gimblett, M. Roggenbach, and H. Schlingloff. Towards a formal specification of an electronic payment system in Csp-Casl. In J. L. Fiadeiro, P. Mosses, and F. Orejas, editors, *WADT 2004*, LNCS. Springer, to appear.
10. M. Gordon and T. Melham. *Introduction to HOL*. Cambrige University Press, 1993.
11. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
12. Y. Isobe and M. Roggenbach. Webpage on Csp-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.
13. J. C. Lagarias. The $3x+1$ problem and its generalizations. *Amer. Math. Monthly*, 92:3–23, 1985.
14. R. Lazic. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University Computing Laboratory, 1999.
15. F. S. E. Limited. Failures-divergence refinement: FDR2. `http://www.fsel.com/`.
16. T. Nipkow, L. C. Paulon, and M. Wenzel. *Isabelle/HOL*. LNCS 2283. Springer, 2002.

17. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *CADE'92*, LNAI 607, pages 748–752. Springer, 1992.
18. L. C. Paulson. *A Generic Theorem Prover*. LNCS 828. Springer, 1994.
19. I. P. Rix Groenboom, Chris Hendriks. Algebraic proof assistants in HOL. In B. Möller, editor, *MPC'95*, LNCS 947, pages 305–321. Springer, 1995.
20. M. Roggenbach. CSP-CASL – A new integration of process algebra and algebraic specification. *Theoretical Computer Science*, to appear.
21. A. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
22. A. Roscoe. On the expressive power of CSP refinement. In *Proceedings of AVoCS'03*, Technical Report. Southampton University, 2003.
23. P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. *The Modelling and Analysis of Security Protocols: the CSP Approach*. Addison-Wesley, 2001.
24. S. Schneider. Verifying authentication protocol implementations. In B. Jacobs and A. Rensink, editors, *FMOODS 2002*, *IFIP Conference Proceedings* Vol. 209, pages 5–24. Kluwer, 2002.
25. H. Tej. *HOL-CSP: Mechanised Formal Development of Concurrent Processes*. BISS Monograph Vol. 19. Logos Verlag Berlin, 2003.
26. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97*, LNCS 1313, pages 318–337. Springer, 1997.