A Complete Axiomatic Semantics for the CSP Stable-Failures Model

Yoshinao Isobe[1] and Markus Roggenbach[2,⋆]

[1] National Institute of Advanced Industrial Science and Technology, Japan
y-isobe@aist.go.jp
[2] University of Wales Swansea, United Kingdom
M.Roggenbach@Swan.ac.uk

Abstract. Traditionally, the various semantics of the process algebra Csp are formulated in denotational style. For many Csp models, e.g., the traces model, equivalent semantics have been given in operational style. A Csp semantics in axiomatic style, however, has been considered problematic in the literature.

In this paper we present a sound and complete axiomatic semantics for Csp with unbounded nondeterminism over an alphabet of arbitrary size.

This result is connected in various ways with our tool Csp-Prover: (1) the Csp dialect under discussion is the input language of Csp-Prover; (2) all theorems presented have been verified with Csp-Prover; (3) Csp-Prover implements the given axiom system.

1 Introduction

Among the various frameworks for the description and modelling of reactive systems, process algebra plays a prominent role. Here, the process algebra Csp [2,8] has successfully been applied in various areas, ranging from train control systems over software for the international space station to the verification of security protocols.

Traditionally, Csp semantics such as the traces model, the failures-divergences model, or the stable-failures model, are formulated in denotational style, c.f. [8]. However, the success of the model checker FDR [6], which clearly is the standard proof tool for Csp, relies on the formulation of operational semantics equivalent to the given denotational ones.

A similar success story with theorem proving for Csp, see, e.g., [1,4,5,9,10] for various approaches, will require an axiomatic (or algebraic) formulation of the Csp models. A *complete* axiomatic semantics for Csp, however, is considered problematic in the literature. There are issues concerning normalisation. The best known results apply for finitely nondeterministic Csp over a finite alphabet of communications only [8]. Consequently, all the implementations listed above are based on a denotational semantics. While this is satisfactory from a theoretical point of view (every true proposition over the denotational semantics

⋆ This cooperation was supported by the EPSRC Project EP/D037212/1.

C. Baier and H. Hermanns (Eds.): CONCUR 2006, LNCS 4137, pp. 158–172, 2006.

can be proven within the theorem prover — up to the incompleteness of the underlying logic[1]), the actual proof-practise relies on an known to be *incomplete* set of algebraic laws and proof rules derived from the denotational semantics implemented.

In this paper we present a sound and complete axiomatic semantics for Csp with unbounded nondeterminism over an alphabet of arbitrary size. Here, we consider full Csp, where the generic internal choice operator has been replaced by a restricted one (this is necessary in order to obtain a *set* of processes rather than a *class*), and where recursion is replaced by infinite nondeterminism over depth-finite processes. We show in Theorem 1 that this language is expressive with respect to the stable-failures domain.

The considered Csp dialect is the input language of our tool Csp-Prover [3,4,5]. Csp-Prover is an interactive theorem prover which supports refinement proofs over various denotational semantics of the process algebra Csp. In the context of this paper, we use Csp-Prover to verify that our axiom system is sound (in this process we found some of the Csp laws established in the literature to be incorrect — see Section 3) as well as to show that the two transformations involved in the completeness proof are semantics preserving.

The paper is organised as follows: First, we introduce our Csp dialect and show that it is expressive. In Section 3 we present a sound axiom system $\mathcal{A}_{\mathcal{F}}$ for stable-failures equivalence. The proof that the axiom system $\mathcal{A}_{\mathcal{F}}$ is complete involves two steps: (1) sequentialisation, see Section 4, and (2) normalisation of sequential processes, see Section 5. Finally, we briefly discuss how to verify the theorems given in this paper with Csp-Prover.

2 The CSP-Dialect

This section summarises syntax and semantics of the input language of Csp-Prover. Especially, we show that it is expressive and that it can deal with infinitely many mutual recursive processes.

2.1 Syntax

Fig. 1 shows the syntax of Csp implemented in Csp-Prover: given an alphabet of communications Σ and the data type of natural numbers Nat, we form a set $Sel(\Sigma)$ of *selectors* to be explained below. $Proc_{\Sigma}$ denotes the set of the processes whose alphabet is Σ.

The set $Sel(\Sigma)$ of *selectors* used in the replicated internal choice is defined as the disjoint sum of the powerset over Σ and the set of the natural numbers:

$$Sel(\Sigma) = \{(set)\, A \mid A \subseteq \Sigma\} \uplus \{(nat)\, n \mid n \in Nat\}$$

Note that replicated internal choice takes a *subset* of $Sel(\Sigma)$ as its parameter.

[1] The traditional formulation of the denotational Csp semantics involves higher-order concepts such as chain-completeness or metric-completeness.

$P ::=$	$Skip$	%% successful terminating process
	$\mid Stop$	%% deadlock process
	$\mid Div$	%% divergence
	$\mid a \rightarrow P$	%% action prefix
	$\mid ?\, x : A \rightarrow P(x)$	%% prefix choice
	$\mid P \,\Box\, P$	%% external choice
	$\mid P \sqcap P$	%% internal choice
	$\mid !!\, s : S \bullet P(s)$	%% replicated internal choice
	$\mid$ *if* b *then* P *else* P	%% conditional
	$\mid P \,[\![X]\!]\, P$	%% generalized parallel
	$\mid P \setminus X$	%% hiding
	$\mid P[[r]]$	%% relational renaming
	$\mid P \fatsemi P$	%% sequential composition
	$\mid P \downarrow n$	%% depth restriction

where $A, X \subseteq \Sigma$, $S \subseteq Sel(\Sigma)$, b is a condition, $r \in \mathbb{P}(\Sigma \times \Sigma)$, and $n \in Nat$.

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

One difference from conventional Csp is that we replace the generic internal choice $\sqcap \mathcal{P}$ by a replicated internal choice $!!\, s : S \bullet P(s)$, i.e., instead of having internal choice over an arbitrary class of processes $\mathcal{P} \subseteq Proc_\Sigma$, internal choice is restricted to run over an indexed set of processes $P(s) : Sel(\Sigma) \Rightarrow Proc_\Sigma$ only, where the index set S is a subset of $Sel(\Sigma)$. The other difference is that we introduce restriction $\downarrow$ as a basic operator. Restriction plays an important role in full-normalisation. In the stable-failures model, restriction cannot be defined in terms of the other basic operators, see [8].

The following shortcuts have proven to be useful:

$$
\begin{aligned}
!set\ A : \mathcal{A} \bullet P(A) &= !!\, s : \{(set)\, A \mid A \in \mathcal{A}\} \bullet P((set)^{-1}(s)) \\
!nat\ n : N \bullet P(n) &= !!\, s : \{(nat)\, n \mid n \in N\} \bullet P((nat)^{-1}(s)) \\
!\ x : A \bullet P(x) &= !set\ X : \{\{x\} \mid x \in A\} \bullet P(contents(X))
\end{aligned}
$$

where $\mathcal{A} \subseteq \mathbb{P}(\Sigma)$, $N \subseteq Nat$, $A \subseteq \Sigma$, and $contents(\{x\}) = x$. Moreover, if the range of the selector is the universe, the universe is often omitted, for example we write $!nat\ n \bullet P(n)$ instead of $!nat\ n : Nat \bullet P(n)$.

2.2 Semantics

In this paper, we concentrate on the denotational stable-failures model $\mathcal{F}$ of Csp. Its domain $\mathcal{F}_\Sigma$ is given as the set of all pairs (T, F) that satisfy certain healthiness conditions, where $T \subseteq \Sigma^{*\checkmark}$ and $F \subseteq \Sigma^{*\checkmark} \times \mathbb{P}(\Sigma^{\checkmark})$[2], see [8] for the details. The semantics of a process P is denoted by $[\![P]\!]_{\mathcal{F}}$, where the map $[\![\cdot]\!]_{\mathcal{F}} : Proc_\Sigma \rightarrow \mathcal{F}_\Sigma$ is expressed in terms of two functions: $[\![P]\!]_{\mathcal{F}} = (traces(P), failures(P))$. Our definitions of *traces* and *failures* are identical to those given in [8]. However, we

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

need to add semantical clauses for our two new operators, namely replicated internal choice[3] and depth restriction:

$$\begin{aligned} \mathit{traces}(!!\ s : S \bullet P(s)) &= \bigcup\{\mathit{traces}(P(s)) \mid s \in S\} \cup \{\langle\rangle\} \\ \mathit{failures}(!!\ s : S \bullet P(s)) &= \bigcup\{\mathit{failures}(P(s)) \mid s \in S\} \\ \mathit{traces}(P \downarrow n) &= \mathit{traces}(P) \downarrow n \\ \mathit{failures}(P \downarrow n) &= \mathit{failures}(P) \downarrow n \end{aligned}$$

where the restriction functions over traces and failures are given as follows:

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

Note that on the domain, which general internal choice and replicated internal choice share, they have the same semantics, see the semantical clauses for general internal choice in the stable-failures model as defined in [8] (note $\mathcal{P} \neq \emptyset$):

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

Process equivalence $=_{\mathcal{F}}$ and process refinement $\sqsubseteq_{\mathcal{F}}$ over the stable failures model are then defined as usual:

$$\begin{aligned} P =_{\mathcal{F}} Q &\Leftrightarrow \mathit{traces}(P) = \mathit{traces}(Q) \wedge \mathit{failures}(P) = \mathit{failures}(Q), \\ P \sqsubseteq_{\mathcal{F}} Q &\Leftrightarrow \mathit{traces}(P) \supseteq \mathit{traces}(Q) \wedge \mathit{failures}(P) \supseteq \mathit{failures}(Q). \end{aligned}$$

2.3 Expressiveness

At first glance, the above defined input language of Csp-Prover seems to be weaker than full Csp as the generic internal choice operator $\sqcap \mathcal{P}$ is missing. However, we can show our language to be expressive.

First, we define a function $\mathit{Proc}_{\mathcal{T}(n)}$ on sets of traces and a function $\mathit{Proc}_{\mathcal{F}(n)}$ on the domain $\mathcal{F}_\Sigma$, inductively on n as follows:

$$\begin{aligned} \mathit{Proc}_{\mathcal{T}(0)}(T) &= \mathit{Div} \\ \mathit{Proc}_{\mathcal{T}(n+1)}(T) &= ((!\, x : \mathit{head}(T) \bullet (x \rightarrow \mathit{Proc}_{\mathcal{T}(n)}(\mathit{tail}(T, x)))) \,\Box\, \mathit{Div}) \\ &\quad \sqcap (\mathit{if}\ (\langle\checkmark\rangle \in T)\ \mathit{then}\ \mathit{Skip}\ \mathit{else}\ \mathit{Div}) \\ \mathit{Proc}_{\mathcal{F}(0)}(T, F) &= !\mathit{set}\ A : \mathit{accept}(T, F) \bullet (?\, x : A \rightarrow \mathit{Div}) \\ \mathit{Proc}_{\mathcal{F}(n+1)}(T, F) &= (!\, x : \mathit{head}(F) \bullet (x \rightarrow \mathit{Proc}_{\mathcal{F}(n)}(\mathit{tail}(T, x), \mathit{tail}(F, x)))) \,\Box\, \mathit{Div} \end{aligned}$$

where *head*, *tail*, and *accept* are defined as

$$\begin{aligned} \mathit{head}(T) &= \{x \in \Sigma \mid \exists\, t.\, \langle x\rangle \frown t \in T\} \\ \mathit{head}(F) &= \{x \in \Sigma \mid \exists\, t\ X.\, (\langle x\rangle \frown t, X) \in F\} \\ \mathit{tail}(T, x) &= \{t \mid \langle x\rangle \frown t \in T\} \\ \mathit{tail}(F, x) &= \{(t, X) \mid (\langle x\rangle \frown t, X) \in F\} \\ \mathit{accept}(T, F) &= \{(\Sigma - Y) \mid (\langle\rangle, Y) \in F \wedge \checkmark \in Y \wedge (\forall\, x \notin Y.\, \langle x\rangle \in T)\}. \end{aligned}$$

[3] To make the implementation easier we allow the empty set $\emptyset$ as a set S of selectors. Consequently we need to add $\{\langle\rangle\}$ to the set of traces. However, this makes sense only in models with a refinement top.

Intuitively, $A \in \mathit{accept}(T, F)$ is the set of communications which are not refused by F and can be performed by T. Next, define a function $\mathit{Proc}_{\mathcal{F}}$ as follows:

$$\mathit{Proc}_{\mathcal{F}}(T, F) = (!\mathit{nat}\ n \bullet \mathit{Proc}_{\mathcal{T}(n)}(T)) \ \sqcap\ (!\mathit{nat}\ n \bullet \mathit{Proc}_{\mathcal{F}(n)}(T, F))$$

With these functions defined, we show that $[\![\cdot]\!]_{\mathcal{F}}$ is surjective on $\mathcal{F}_{\Sigma}$:

Theorem 1. *For all* $(T, F) \in \mathcal{F}_{\Sigma}$, $[\![\mathit{Proc}_{\mathcal{F}}(T, F)]\!]_{\mathcal{F}} = (T, F)$.

Proof sketch. We prove by induction on n : if $t \in \mathit{traces}(\mathit{Proc}_{\mathcal{T}(n)}(T))$ or $t \in \mathit{traces}(\mathit{Proc}_{\mathcal{F}(n)}(T, F))$ for some n, then $t \in T$. Then we show by induction on the length of t : if $t \in T$ then $t \in \mathit{traces}(\mathit{Proc}_{\mathcal{T}(\mathit{length}(t))}(T))$. Hence, $\mathit{traces}(\mathit{Proc}_{\mathcal{F}}(T, F)) = T$. Equality for *failures* follows by a similar argument. □

2.4 Recursive Processes

Infinite processes can be effectively expressed by fixed points. For example, a buffer *Buffer*, which iteratively receives a real number r from the channel *in* and sends it to a channel *out* together with an increasing natural number *id*, can be defined by using a solution f of the following system of equations[4]:

$$\begin{array}{l} f\,(\mathit{Empty}\,(\mathit{id})) \ =_{\mathcal{F}}\ \mathit{in}\,?\,r \ \rightarrow (f\,(\mathit{Full}\,(r, \mathit{id}))) \\ f\,(\mathit{Full}\,(r, \mathit{id})) \ =_{\mathcal{F}}\ \mathit{out}\,(r, \mathit{id}) \ \rightarrow (f\,(\mathit{Empty}\,(\mathit{id} + 1))) \end{array}$$

where *Empty* and *Full* are *names*, and f is a function whose domain is

$$\mathit{Dom}(f) = \{\mathit{Empty}\,(\mathit{id}) \mid \mathit{id} \in \mathit{Nat}\} \cup \{\mathit{Full}\,(r, \mathit{id}) \mid r \in \mathit{Real}, \mathit{id} \in \mathit{Nat}\}$$

and whose range is the set of all processes. Any solution f is a fixed point $(\mathit{Fix}\ \mathit{fun})$ of the function $\mathit{fun} : (\mathit{Dom}(f) \Rightarrow \mathit{Proc}_{\Sigma}) \Rightarrow (\mathit{Dom}(f) \Rightarrow \mathit{Proc}_{\Sigma})$ given as:

$$\begin{array}{l} \mathit{fun}(f)(\mathit{Empty}\,(\mathit{id})) := \mathit{in}\,?\,r \ \rightarrow (f\,(\mathit{Full}\,(r, \mathit{id}))) \\ \mathit{fun}(f)(\mathit{Full}\ (r, \mathit{id})) := \mathit{out}\,(r, \mathit{id}) \ \rightarrow (f\,(\mathit{Empty}\,(\mathit{id} + 1))) \end{array}$$

Therefore, the process *Buffer*, which initially has no data and whose initial *id* is zero, is given as $(\mathit{Fix}\ \mathit{fun})(\mathit{Empty}\,(0))$.

Csp offers two standard approaches to deal with fixed-points: complete partial orders (cpo) with Tarski's fixed point theorem or complete metric spaces (cms) with Banach's fixed point theorem. The limits $(\mathit{Fix}\ \mathit{fun})$ and $(\mathit{Fix}!\ \mathit{fun})$ of the converging sequences in Tarski's and Banach's fixed point theorems can be defined in our Csp-dialect as follows:

$$\begin{array}{rl} (\mathit{Fix}\ \mathit{fun})(x) := & !\mathit{nat}\ n \bullet ((\mathit{fun}^{(n)}(\lambda\, y.\, \mathit{Div}))(x)) \\ (\mathit{Fix}!\ \mathit{fun})(x) := & !\mathit{nat}\ n \bullet (((\mathit{fun}^{(n)}(\lambda\, y.\, \mathit{Any}))(x)) \downarrow n) \end{array}$$

where *Div* plays the role of the bottom element in the cpo approach and *Any* stands for any process, which corresponds to the arbitrary initial point of Banach's theorem. Then, as expected, the following properties hold:

[4] $\mathit{in}\,?\,r \rightarrow P(r)$ is a syntactic sugar for $?\,x : \{\mathit{in}(r) \mid r \in \mathit{Real}\} \rightarrow P(\mathit{in}^{-1}(x))$.

1. Let $fun \in ProcFun_\Sigma$. Then $(Fix\, fun)(x) =_\mathcal{F} (fun\,(Fix\, fun))(x)$ for all x; furthermore, for any f with $(\forall x. f(x) =_\mathcal{F} fun(f)(x))$ holds $f(x) \sqsubseteq_\mathcal{F} (Fix\, fun)(x)$. Thus, $(Fix\, fun)$ is the greatest fixed point on $\sqsubseteq_\mathcal{F}$, in other words, it is the least fixed point in the semantic domain.
2. Let $fun \in ProcFun_\Sigma$ be guarded and without hiding operator. Then we have $(Fix!\ fun)(x) =_\mathcal{F} (fun\,(Fix!\ fun))(x)$ for all x; furthermore, for every f, if $(\forall\, x.\ f(x) =_\mathcal{F} fun(f)(x))$ then $f(x) =_\mathcal{F} (Fix!\ fun)(x)$. Thus, $(Fix!\ fun)$ is the unique fixed point on $=_\mathcal{F}$.

Here, $ProcFun_\Sigma$ is the set of functions fun such that for all x, $(\lambda f.\ fun(f)(x))$ is a *process-function*. Each process-function $P(f)$ is a process that may contain a process-function variable f, see, e.g., the above example *Buffer*.

Thus, both ways of CSP of dealing with systems of recursive equations, the cpo approach using Tarski's fixed point theorem as well as the cms approach using Banach's fixed point theorem, are expressible in the input language of CSP-Prover.

3 Axiom System

In this section, we present a sound axiom system $\mathcal{A}_\mathcal{F}$ for the CSP stable-failures model. The completeness of $\mathcal{A}_\mathcal{F}$ is shown later in the Sections 4 and 5.

We write $\mathcal{A}_\mathcal{F} \vdash P = Q$ if the equality of two processes P and Q can be proven by equational and inductive reasoning from the axiom system $\mathcal{A}_\mathcal{F}$. Fig. 2 summarizes changes from the axiom system for finite processes given in [8]. The superscript $*$ denotes modified laws, the superscript $+$ denotes added laws.

Our axiom system $\mathcal{A}_\mathcal{F}$ replaces the usual unwinding laws for recursive processes such as $(\mu X.\, P(X)) = P(\mu X.\, P(X))$ by new axioms (Tarski-fix) and (Banach-fix). While the unwinding laws have proven to be handy for verifying practical systems with CSP-Prover, at the same time they cause problems with normalization: see the discussion on infinite unwinding of divergent processes such as $(\mu X.\, X)$ in [8], p. 273. Our laws (Tarski-fix) and (Banach-fix), however, transforms recursive processes to unbounded nondeterministic processes. Such processes can then be analyzed via induction on n. We give an example of how to normalise a divergent process at the end of Section 5.

Secondly, we found that the well-known laws ($[\![X]\!]$-$\triangleright$-split) and ($[\![X]\!]$-$\triangleright$-input) (P.288–289 in [8]) are not correct:

$$(P \triangleright P') \,[\![\, X \,]\!]\, (Q \triangleright Q')$$
$$= (P \,[\![\, X \,]\!]\, Q) \triangleright ((P' \,[\![\, X \,]\!]\, (Q \triangleright Q')) \sqcap ((P \triangleright P') \,[\![\, X \,]\!]\, Q')), \qquad ([\![X]\!]\text{-}\triangleright\text{-split})$$

$$(P \triangleright P') \,[\![\, X \,]\!]\, (?x : A \to Q(x))$$
$$= (?x : (A - X) \to ((P \triangleright P') \,[\![\, X \,]\!]\, Q(x)))$$
$$\quad \Box\ ((P \,[\![\, X \,]\!]\, (?x : A \to Q(x))) \triangleright (P' \,[\![\, X \,]\!]\, (?x : A \to Q(x)))). \quad ([\![X]\!]\text{-}\triangleright\text{-input})$$

For example, instantiate the processes in ($[\![X]\!]$-$\triangleright$-split) as follows: $P = a \to Stop$, $P' = Stop$, $Q = Stop$, $Q' = b \to Stop$, and $X = \emptyset$, where $a \neq b$. In this case, the semantics of the left hand side of ($[\![X]\!]$-$\triangleright$-split) does not contain the failure $(\langle a \rangle, b)$

$$(\mathit{Fix}\ \mathit{fun})(x) = !\mathit{nat}\ n \bullet ((\mathit{fun}^{(n)}(\lambda\, y.\ \mathit{Div}))(x)) \qquad (\text{Tarski-fix})^+$$
$$(\mathit{Fix}!\ \mathit{fun})(x) = !\mathit{nat}\ n \bullet (((\mathit{fun}^{(n)}(\lambda\, y.\ P))(x)) \downarrow n) \qquad (\text{Banach-fix})^+$$

---

if $P = (?x : A \to P'(x)) \triangleright P''$ and $Q = (?x : B \to Q'(x)) \triangleright Q''$ then
$$\begin{aligned} & P \,[\![X]\!]\, Q \\ & = (?x : ((X \cap A \cap B) \cup (A - X) \cup (B - X)) \to \\ & \qquad \mathit{if}\ (x \in X)\ \mathit{then}\ (P'(x) \,[\![X]\!]\, Q'(x)) \\ & \qquad \mathit{else\ if}\ (x \in A \cap B)\ \mathit{then}\ ((P'(x) \,[\![X]\!]\, Q) \sqcap (P \,[\![X]\!]\, Q'(x))) \\ & \qquad\qquad \mathit{else\ if}\ (x \in A)\ \mathit{then}\ (P'(x) \,[\![X]\!]\, Q)\ \mathit{else}\ (P \,[\![X]\!]\, Q'(x))) \\ & \quad \triangleright ((P'' \,[\![X]\!]\, Q) \sqcap (P \,[\![X]\!]\, Q'')) \end{aligned} \qquad (|[X]|\text{-}\triangleright\text{-split})^*$$

if $P = (?x : A \to P'(x)) \triangleright P''$ and $Q = ?x : B \to Q'(x)$ then
$$\begin{aligned} & P \,[\![X]\!]\, Q \\ & = (?x : ((X \cap A \cap B) \cup (A - X) \cup (B - X)) \to \\ & \qquad \mathit{if}\ (x \in X)\ \mathit{then}\ (P'(x) \,[\![X]\!]\, Q'(x)) \\ & \qquad \mathit{else\ if}\ (x \in A \cap B)\ \mathit{then}\ ((P'(x) \,[\![X]\!]\, Q) \sqcap (P \,[\![X]\!]\, Q'(x))) \\ & \qquad\qquad \mathit{else\ if}\ (x \in A)\ \mathit{then}\ (P'(x) \,[\![X]\!]\, Q)\ \mathit{else}\ (P \,[\![X]\!]\, Q'(x))) \\ & \quad \triangleright (P'' \,[\![X]\!]\, Q) \end{aligned} \qquad (|[X]|\text{-}\triangleright\text{-input})^*$$

$$!!\, s : \emptyset \bullet P(s) = \mathit{Div} \qquad (!!\text{-emptyset})^+$$
$$\text{if } S \neq \emptyset \text{ and } (\forall\, s \in S.\ P(s) = Q) \text{ then } !!\, s : S \bullet P(s) = Q \qquad (!!\text{-const})^*$$
$$!!\, s : (S_1 \cup S_2) \bullet P(s) = (!!\, s : S_1 \bullet P(s)) \sqcap (!!\, s : S_2 \bullet P(s)) \qquad (!!\text{-union-}\sqcap)^*$$
$$\begin{aligned} & !!\, s : S \bullet (?\, x : A(s) \to P(s, x)) \\ & = !\mathit{set}\ X : \{A(s) \mid s \in S\} \bullet \\ & \quad (?\, x : X \to (!!\, s : \{s \in S \mid x \in A(s)\} \bullet P(s, x))) \end{aligned} \qquad (!!\text{-input-}!\mathrm{set})^+$$
$$\text{if } \forall\, s \in S.\ Q(s) \in \{\mathit{Skip}, \mathit{Div}\} \text{ then}$$
$$!!\, s : S \bullet (P(s) \,\Box\, Q(s)) = (!!\, s : S \bullet P(s)) \,\Box\, (!!\, s : S \bullet Q(s)) \qquad (!!\text{-}\Box\text{-Dist})^+$$
$$\text{if } Q \in \{\mathit{Skip}, \mathit{Div}\} \text{ then}$$
$$(!\mathit{set}\ X : \mathcal{X} \bullet (?\, x : X \to P(x))) \,\Box\, Q = (?\, x : \bigcup \mathcal{X} \to P(x)) \,\Box\, Q \qquad (!!\text{-input-Dist})^+$$

$$P \downarrow 0 = \mathit{Div} \qquad (\downarrow\text{-zero})^+$$
$$(P \downarrow n) \downarrow m = P \downarrow \min(n, m) \qquad (\downarrow\text{-min})^+$$
$$P = !\mathit{nat}\ n \bullet (P \downarrow n) \qquad (!\mathrm{nat}\text{-}\downarrow)^+$$
$$(!!\, s : S \bullet P(s)) \downarrow n = !!\, s : S \bullet (P(s) \downarrow n) \qquad (\downarrow\text{-Dist})^+$$
$$(P_1 \sqcap P_2) \downarrow n = (P_1 \downarrow n) \sqcap (P_2 \downarrow n) \qquad (\downarrow\text{-dist})^+$$
$$(P_1 \,\Box\, P_2) \downarrow n = (P_1 \downarrow n) \,\Box\, (P_2 \downarrow n) \qquad (\downarrow\text{-}\Box\text{-dist})^+$$
$$\mathit{Skip} \downarrow (n + 1) = \mathit{Skip} \qquad (\mathrm{skip}\text{-}\downarrow)^+$$
$$\mathit{Div} \downarrow n = \mathit{Div} \qquad (\mathrm{div}\text{-}\downarrow)^+$$
$$(?\, x : A \to P(x)) \downarrow (n + 1) = ?\, x : A \to (P(x) \downarrow n) \qquad (\downarrow\text{-step})^+$$

$$?\ x : A \to P(x) = ((?\, x : A \to P(x)) \,\Box\, \mathit{Div}) \sqcap (?\, x : A \to \mathit{Div}) \qquad (?\text{-div})^+$$
$$\begin{aligned} & !!\, s : S \bullet (!\mathit{set}\ X : \mathcal{X}(s) \bullet (?\, x : X \to \mathit{Div})) \\ & = !\mathit{set}\ X : \bigcup \{\mathcal{X}(s) \mid s \in S\} \bullet (?\, x : X \to \mathit{Div}) \end{aligned} \qquad (!!\text{-}!\mathrm{set\text{-}div})^+$$
$$\text{if } \mathcal{X} \subseteq \mathcal{Y} \text{ and } (\forall\, Y \in \mathcal{Y}.\ \exists\, X \in \mathcal{X}.\ X \subseteq Y \subseteq A) \text{ then}$$
$$\begin{aligned} & ((?\, x : A \to P(x)) \,\Box\, Q) \sqcap (!\mathit{set}\ X : \mathcal{X} \bullet (?\, x : X \to \mathit{Div})) \\ & = ((?\, x : A \to P(x)) \,\Box\, Q) \sqcap (!\mathit{set}\ X : \mathcal{Y} \bullet (?\, x : X \to \mathit{Div})) \end{aligned} \qquad (?\text{-}!\mathrm{set\text{-}}\subseteq)^+$$

Fig. 2. The axiom system $\mathcal{A}_{\mathcal{F}}$ (differences from [8])

because $(Stop \triangleright (b \rightarrow Stop))$ can perform b even after P has performed a. On the other hand, the semantics of the right hand side of ($[\![X]\!]$-$\triangleright$-split) contains the failure $(\langle a \rangle, b)$ because it has a subexpression $((a \rightarrow Stop) \,[\![\emptyset]\!]\, Stop)$. Therefore, the law ($[\![X]\!]$-$\triangleright$-split) does not hold. Similarly, there is a counter example (e.g. $P = Stop$, $P' = b \rightarrow Stop$, $A = \{a\}$, $Q(a) = Stop$, and $X = \emptyset$) for the law ($[\![X]\!]$-$\triangleright$-input). Hence, we modified the laws ($[\![X]\!]$-$\triangleright$-split) and ($[\![X]\!]$-$\triangleright$-input) as shown in Fig. 2. The modified laws are less generic than the original ones, but they are expressive enough to gain completeness.

Thirdly, we added laws (!!-$\cdots$) for replicated internal choice $!!\, s : S \bullet P(s)$, as shown in Fig. 2, by modifying the laws for the (generic) internal choice $\sqcap$. The laws (!!-input-!set), (!!-$\Box$-Dist), and (!!-input-Dist) are used for replacing replicated internal choice by (external) prefix choice, considering the effects of *Skip* or *Div*. These laws were added instead of the following law for the binary internal choice (P.289 in [8]): $(P \,\Box\, Skip) \sqcap (Q \,\Box\, Skip) = (P \,\Box\, Q \,\Box\, Skip)$.

Furthermore, we added the laws ($\downarrow$-$\cdots$) for the restriction operator as shown in Fig. 2. The most important law is (!nat-$\downarrow$) which is used for finitising the depth of infinite processes.

Finally, we added the laws (?-div), (!!-!set-div), and (?-!set-$\subseteq$) for normalising sequential processes. The law (?-!set-$\subseteq$) is used to satisfy the condition (N_3) of full normal forms, stated in Definition 2 in Section 5.

The presented axiom system $\mathcal{A}_{\mathcal{F}}$ is sound:

Theorem 2. *Let $P, Q \in Proc_{\Sigma}$. Then $\mathcal{A}_{\mathcal{F}} \vdash P = Q$ implies $P =_{\mathcal{F}} Q$.*

4 Full Sequentialisation

In this section, we define a method to fully sequentialise a process. The purpose of this transformation is to remove hiding. Hiding operators can cause a problem when normalising processes with the help of depth restriction operators: $P \downarrow n =_{\mathcal{F}} Q \downarrow n$ does not necessarily imply $(P \setminus X) \downarrow n =_{\mathcal{F}} (Q \setminus X) \downarrow n$ due to hidden communications.

First, we define the set $SeqProc_{\Sigma}$ of processes in *full sequential forms*. Processes in full sequential form are built using the various CSP choice operators and the basic processes *Skip*, *Stop* and *Div* only. More formally:

Definition 1. *The set $SeqProc_{\Sigma}$ is defined as the smallest set satisfying*

1. $(?\, x : A \rightarrow P(x)) \,\Box\, Q \in SeqProc_{\Sigma}$,
   if $P(x) \in SeqProc_{\Sigma}$ for all $x \in A$ and $Q \in \{Skip, Div, Stop\}$.
2. $!!\, s : S \bullet P(s) \in SeqProc_{\Sigma}$,
   if $P(s) \in SeqProc_{\Sigma}$ for all $s \in S$, where $S \neq \emptyset$.

If $P \in SeqProc_{\Sigma}$, we say that P is in full sequential form.

As the set A in the first condition is allowed to be the empty set, we have for example $(?\, x : \emptyset \rightarrow Div) \,\Box\, Skip \in SeqProc_{\Sigma}$.

Next, we define for each CSP operator *op* a sequentialising function f_{op}. Applying f_{op} to a CSP process P which has *op* as its out-most operator, the function will transform this process into a semantically equivalent CSP process $f_{op}(P)$ in

$$
\begin{array}{rl}
(Pr)_{seq} & = (?\,x : \emptyset \to Div) \,\Box\, Pr \qquad (Pr \in \{Skip, Div, Stop\}) \\
a \to_{seq} P_1 & = (?\,x : \{a\} \to P_1) \,\Box\, Stop \\
?\,x : A \to_{seq} P(x) & = (?\,x : A \to P(x)) \,\Box\, Stop \\
!!\,s : S \bullet_{seq} P'(s) & = \begin{cases} (Div)_{seq} & ;\ S = \emptyset \\ !!\,s : S \bullet P'(s) & ;\ \text{otherwise} \end{cases} \\
P_1 \sqcap_{seq} P_2 & = !nat\ n : \{0,1\} \bullet (\text{if } (n = 0) \text{ then } P_1 \text{ else } P_2)
\end{array}
$$

$$
\begin{array}{rl}
P_1 \,\Box_{seq}\, Pr & = !!\,s : S_1 \bullet (R'_1(s) \,\Box_{seq}\, Pr) \qquad (Pr \in \{P_2, R_2\}) \\
R_1 \,\Box_{seq}\, P_2 & = !!\,s : S_2 \bullet (R_1 \,\Box_{seq}\, R'_2(s)) \\
R_1 \,\Box_{seq}\, R_2 & = (?\,x : (A_1 \cup A_2) \to \\
& \quad \text{if } (x \in A_1 \cap A_2) \text{ then } P'_1(x) \sqcap_{seq} P'_2(x) \\
& \quad \text{else if } (x \in A_1) \text{ then } P'_1(x) \text{ else } P'_2(x)) \\
& \Box \text{ if } (Q_1 = Skip \vee Q_2 = Skip) \text{ then } Skip \\
& \quad \text{else if } (Q_1 = Div \vee Q_2 = Div) \text{ then } Div \text{ else } Stop \\
Pr_1 \triangleright_{seq} Pr_2 & = (Pr_1 \sqcap_{seq} (Stop)_{seq}) \,\Box_{seq}\, Pr_2 \qquad (Pr_i \in \{P_i, R_i\})
\end{array}
$$

$$
\begin{array}{rl}
P_1 \,[\![\, X \,]\!]_{seq}\, Pr & = !!\,s : S_1 \bullet (R'_1(s) \,[\![\, X \,]\!]_{seq}\, Pr) \qquad (Pr \in \{P_2, R_2, Skip, Div\}) \\
R_1 \,[\![\, X \,]\!]_{seq}\, P_2 & = !!\,s : S_2 \bullet (R_1 \,[\![\, X \,]\!]_{seq}\, R'_2(s)) \\
R_1 \,[\![\, X \,]\!]_{seq}\, Skip & = ((?\,x : (A_1 - X) \to (P'_1(x) \,[\![\, X \,]\!]_{seq}\, Skip)) \,\Box\, Q_1) \\
R_1 \,[\![\, X \,]\!]_{seq}\, Div & = ((?\,x : (A_1 - X) \to (P'_1(x) \,[\![\, X \,]\!]_{seq}\, Div)) \,\Box\, Div) \\
R_1 \,[\![\, X \,]\!]_{seq}\, R_2 & = \text{if } (Q_1 = Stop \wedge Q_2 = Stop) \text{ then } R_1 \,[\![\, X \,]\!]^{step}_{seq}\, R_2 \\
& \quad \text{else } (R_1 \,[\![\, X \,]\!]^{step}_{seq}\, R_2) \\
& \qquad \triangleright_{seq} (\text{if } (Q_1 = Stop) \text{ then } (R_1 \,[\![\, X \,]\!]_{seq}\, Q_2) \\
& \qquad\quad \text{else if } (Q_2 = Stop) \text{ then } (R_2 \,[\![\, X \,]\!]_{seq}\, Q_1) \\
& \qquad\quad \text{else } (R_1 \,[\![\, X \,]\!]_{seq}\, Q_2) \sqcap_{seq} (R_2 \,[\![\, X \,]\!]_{seq}\, Q_1)) \\
R_1 \,[\![\, X \,]\!]^{step}_{seq}\, R_2 & = ?\,x : ((X \cap A_1 \cap A_2) \cup (A_1 - X) \cup (A_2 - X)) \to \\
& \quad (\text{if } (x \in X) \text{ then } (P'_1(x) \,[\![\, X \,]\!]_{seq}\, P'_2(x)) \\
& \qquad \text{else if } (x \in A_1 \cap A_2) \\
& \qquad\quad \text{then } ((P'_1(x) \,[\![\, X \,]\!]_{seq}\, R_2) \sqcap_{seq} (R_1 \,[\![\, X \,]\!]_{seq}\, P'_2(x))) \\
& \qquad\quad \text{else if } (x \in A_1) \\
& \qquad\qquad \text{then } (P'_1(x) \,[\![\, X \,]\!]_{seq}\, R_2) \text{ else } (R_1 \,[\![\, X \,]\!]_{seq}\, P'_2(x))) \\
& \Box\ Stop \\
P_1 \setminus_{seq} X & = !!\,s : S_1 \bullet (R'_1(s) \setminus_{seq} X) \\
R_1 \setminus_{seq} X & = \text{if } (Q_1 = Stop) \text{ then} \\
& \quad \text{if } (A_1 \cap X = \emptyset) \\
& \quad \text{then } ((?\,x : A_1 \to (P'_1(x) \setminus_{seq} X)) \,\Box\, Q_1) \\
& \quad \text{else } ((?\,x : (A_1 - X) \to (P'_1(x) \setminus_{seq} X)) \,\Box\, Q_1) \\
& \qquad \triangleright_{seq} (!\,x : (A_1 \cap X) \bullet_{seq} (P'_1(x) \setminus_{seq} X)) \\
& \quad \text{else } (((?\,x : (A_1 - X) \to (P'_1(x) \setminus_{seq} X)) \,\Box\, Q_1) \\
& \qquad \sqcap_{seq} (!\,x : (A_1 \cap X) \bullet_{seq} (P'_1(x) \setminus_{seq} X)))
\end{array}
$$

In this figure, it is assumed that
$P(x) \in SeqProc_\Sigma$ for all $x \in A$,
$P'(s) \in SeqProc_\Sigma$ for all $s \in S$,
$P_i = !!\,s : S_i \bullet R'_i(s) \in SeqProc_\Sigma$ for each $i \in \{1, 2\}$, and
$R_i = (?\,x : A_i \to P'_i(x)) \,\Box\, Q_i \in SeqProc_\Sigma$ for each $i \in \{1, 2\}$.

Fig. 3. Sequentialising functions (part)

full sequential form, provided its subprocesses are already in full sequential form. Here, we actually prove a stronger proposition, namely $\mathcal{A}_{\mathcal{F}} \vdash P = f_{op}(P)$. For example, we have $\mathcal{A}_{\mathcal{F}} \vdash P \,[\![X]\!]\, Q = f_{par}(P \,[\![X]\!]\, Q)$, which according to Theorem 2 implies $P \,[\![X]\!]\, Q =_{\mathcal{F}} f_{par}(P \,[\![X]\!]\, Q)$, and $f_{par}(P \,[\![X]\!]\, Q) \in SeqProc_{\Sigma}$ for $P, Q \in SeqProc_{\Sigma}$. For convenience, we use infix notation to write the functions f_{op}, for example we write $P \,[\![X]\!]_{seq}\, Q$ instead of $f_{par}(P \,[\![X]\!]\, Q)$. Note the inductive structure of the sequentialising functions presented in Fig. 3.

Finally, we define an overall sequentialisation function $Seq : Proc_{\Sigma} \Rightarrow SeqProc_{\Sigma}$ inductively on the syntactic structure of processes:

$$\begin{aligned}
Seq(P) &= (P)_{seq} && (P \in \{Skip, Div, Stop\}) \\
Seq(a \rightarrow P) &= a \rightarrow_{\mathrm{seq}} Seq(P) \\
Seq(?x : A \rightarrow P(x)) &= ?x : A \rightarrow_{\mathrm{seq}} Seq(P(x)) \\
Seq(P \oplus Q) &= Seq(P) \oplus_{\mathrm{seq}} Seq(Q) && (\oplus \in \{\Box, \sqcap, [\![X]\!], {}^{s}_{s}\}) \\
Seq(!!\, s : S \bullet P(s)) &= !!\, s : S \bullet_{\mathrm{seq}} Seq(P(s)) && \cdots
\end{aligned}$$

For this function Seq, Theorem 3 holds:

Theorem 3. *$Seq(P) \in SeqProc_{\Sigma}$ and $\mathcal{A}_{\mathcal{F}} \vdash P = Seq(P)$ for all $P \in Proc_{\Sigma}$.*

Proof sketch. First we show that each sequentialising function f_{op} indeed sequentialises processes, e.g., if $P, Q \in SeqProc_{\Sigma}$ then $\mathcal{A}_{\mathcal{F}} \vdash P \,[\![X]\!]\, Q = P \,[\![X]\!]_{seq}\, Q$ and $P \,[\![X]\!]_{seq}\, Q \in SeqProc_{\Sigma}$, by induction on the structures of full sequential forms P and Q. Equality can often be derived by using the *distributive-laws* and *step-laws* taking into account the special role of *Skip* and *Div*. From this, the result on $Seq(P)$ follows easily. □

5 Full Normalisation

Semantically equivalent processes $P =_{\mathcal{F}} Q$ in full sequential form can still be different syntactically: Let $A \neq B$ and $R(x, y) = (x \rightarrow_{seq} y \rightarrow_{seq} (Skip)_{seq})$. Then $!\, x : A \bullet_{seq} (!\, y : B \bullet_{seq} R(x, y)) \neq !\, x : B \bullet_{seq} (!\, y : A \bullet_{seq} R(y, x))$, are both in full sequential form, although the two processes have the same semantics; semantically it does not matter in which order the selector sets are defined. Therefore, the next step is to study normalisation.

First, we define a new full normal form, which differs slightly from the full normal form for finite processes presented in [8]:

Definition 2. *A process $P \in Proc_{\Sigma}$ is said to be in full normal form if and only if P has the form $((?x : A \rightarrow P(x)) \,\Box\, Q) \sqcap (!set\ X : \mathcal{X} \bullet (?x : X \rightarrow Div))$ and the following four conditions $(N_1), \ldots, (N_4)$ are satisfied: (N_1) for all x, if $x \in A$ then $P(x)$ is already in full normal form else $P(x)$ is Div[5], (N_2) $\bigcup \mathcal{X} \subseteq A$, (N_3) $\forall X. ((\exists X_0 \in \mathcal{X}. X_0 \subseteq X \subseteq A) \Rightarrow X \in \mathcal{X})$, and (N_4) $Q \in \{Skip, Div\}$.*

The set of full normal forms is denoted by $NormProc_{\Sigma}$.

[5] $?\, x : A \rightarrow P(x) =_{\mathcal{F}} ?\, x : A \rightarrow Q(x)$ implies $P(x) =_{\mathcal{F}} Q(x)$ for all $x \in A$. However, for $x \in \Sigma \backslash A$ we do not necessarily have $P(x) =_{\mathcal{F}} Q(x)$. Since $P(x)$ and $Q(x)$ are total functions over Σ, for values outside of A we need to fix them to some constant as, e.g., *Div* in order to obtain uniqueness.

Our definition 2 differs from [8] only in condition (N_3). [8] requires all elements of $\mathcal{X}$ to be incomparable. In fact, if $\mathcal{X}$ is *finite*, we can replace our set $\mathcal{X}$ in the full normal form by the incomparable set $\{\bigcap\{X_0 \in \mathcal{X} \mid X_0 \subseteq X\} \mid X \in \mathcal{X}\}$ without changing the semantics of the process. However, if $\mathcal{X}$ is *infinite*, the semantics may change: $\bigcap\{X_0 \in \mathcal{X} \mid X_0 \subseteq X\}$ is not always contained in $\mathcal{X}$. Therefore, we require (N_3) instead of incomparability.

Next we prove that for processes in $NormProc_\Sigma$ syntactic and semantic equality are the same:

Theorem 4. *For all $P, Q \in NormProc_\Sigma$, $P =_\mathcal{F} Q$ if and only if $P = Q$.*

Proof. Almost identical to the proof presented in [8]. □

While for every *finitely* nondeterministic process P with a finite alphabet, there is a process P' in full normal form such that $P =_\mathcal{F} P'$, this does not hold for *infinitely* nondeterministic processes with an arbitrary alphabet as follows.

Theorem 5. *There exist $P \in Proc_\Sigma$ with $P \neq_\mathcal{F} P'$ for all $P' \in NormProc_\Sigma$.*

Proof. Consider the process $Loop_a := (Fix\ fun_a)(A)$ with $fun_a(f)(A) := a \rightarrow f(A)$, thus, $Loop_a$ satisfies the equation $Loop_a =_\mathcal{F} a \rightarrow Loop_a$. Note that $(Fix\ fun_a)$ is expressed by infinite nondeterminism over Nat. Assume there exists some $P' \in NormProc_\Sigma$ with $Loop_a =_\mathcal{F} P'$. Define

$$\begin{aligned} P'' := ((?\, x : \{a\} \rightarrow (\text{if } (x = a) \text{ then } P' \text{ else } Div)) \,\Box\, Div) \\ \sqcap\, (!set\ X : \{\{a\}\} \bullet (?\, x : X \rightarrow Div)) \end{aligned}$$

Consequently $P' \neq P''$. On the other hand, we can prove that $P'' \in NormProc_\Sigma$ and $P' =_\mathcal{F} P''$ — contradiction to Theorem 4. □

To deal with this weakness, we define an *extended* full normal form.

Definition 3. *A process P is in extended full normal form iff P is of the form $!nat\ n \bullet P'(n)$, where the processes $P'(n)$ are in full normal form and $P'(n) =_\mathcal{F} P \downarrow n$ for all $n \in Nat$. We denote the set of extended full normal forms by $XNormProc_\Sigma$.*

The extended full normal form consists of an infinite nondeterministic choice between a family of fully normalised processes $P(n)$, where the depth of the processes $P(n)$ is restricted to n by the restriction operator $\downarrow$.

First we give an example of extended full normal form of an infinite process.

Example 1. For $n \in Nat$ define the process $Inc(n)$ as $(Fix\ fun_{inc})(n)$ with $fun_{inc}(f)(n) := !nat\ m : GT(n) \bullet m \rightarrow f(m)$ and $GT(n) = \{m \mid n < m\}$. $Inc(n)$ produces a sequence of natural numbers $> n$ where the increment is nondeterministically chosen. Let $Ninc(n)$ be the process $!nat\ i \bullet Ninc(i, n)$ with

$$\begin{aligned} Ninc(0, n) &:= (Div)_{norm} \\ Ninc(i+1, n) &:= ((?\, m : GT(n) \rightarrow (\text{if } (n < m) \text{ then } Ninc(i, m) \text{ else } Div)) \,\Box\, Div) \\ &\quad \sqcap\, (!set\ N : \{N \mid N \neq \emptyset, N \subseteq GT(n)\} \bullet (?\, m : N \rightarrow Div)) \end{aligned}$$

$$
\begin{aligned}
!!s : S \bullet^{(0)}_{norm} P(s) &= (Div)_{norm} \\
!!s : S \bullet^{(n+1)}_{norm} P(s) &= ((?x : A' \rightarrow \\
&\quad \text{if } (x \in A') \text{ then } (!!s : \{s \in S \mid x \in A(s)\} \bullet^{(n)}_{norm} P'(s,x)) \\
&\quad \text{else } Div) \\
&\quad \Box \text{ if } (\exists\, s \in S.Q(s) = Skip) \text{ then } Skip \text{ else } Div) \\
&\quad \sqcap\, !set\ X : (complete(A', \mathcal{X}')) \bullet (?x : X \rightarrow Div) \\
\text{where} & \\
(Div)_{norm} &= ((?x : \emptyset \rightarrow Div) \,\Box\, Div) \sqcap (!set\ X : \emptyset \bullet (?x : X \rightarrow Div)), \\
complete(A', \mathcal{X}') &= \{X \mid \exists\, X_0 \in \mathcal{X}'.\ X_0 \subseteq X \subseteq A'\}, \\
\forall\, s \in S.\ P(s) &= ((?x : A(s) \rightarrow P'(s,x)) \,\Box\, Q(s)) \sqcap \\
&\quad (!set\ X : \mathcal{X}(s) \bullet (?x : X \rightarrow Div)) \in NormProc_\Sigma, \\
A' &= \bigcup\{A(s) \mid s \in S\}, \text{ and } \mathcal{X}' = \bigcup\{\mathcal{X}(s) \mid s \in S\}.
\end{aligned}
$$

Fig. 4. Normalising function for replicated internal choice

Here, $(Div)_{norm}$ is the full normal form of Div as defined in Fig. 4. $Ninc(n) =_{\mathcal{F}} Inc(n)$ and $Ninc(n) \in XNormProc_{Nat}$.

Next we prove that for processes in $XNormProc_\Sigma$ syntactic and semantic equality are the same:

Theorem 6. *For all $P, Q \in XNormProc_\Sigma$, $P =_{\mathcal{F}} Q$ if and only if $P = Q$.*

Proof. Let $P, Q \in XNormProc_\Sigma$ and $P =_{\mathcal{F}} Q$. Thus, for some P' and Q', $P = !nat\, n \bullet P'(n)$ and $Q = !nat\, n \bullet Q'(n)$. Further, for all n, $P'(n) =_{\mathcal{F}} P \downarrow n =_{\mathcal{F}} Q \downarrow n =_{\mathcal{F}} Q'(n)$. Thus, $P'(n) = Q'(n)$ by Theorem 4. Hence, $P = Q$. □

Then we define a function that transforms processes of the form $!!\, s : S \bullet P(s)$ into full normal form, see Fig. 4. Note that the function $!!\, s : S \bullet^{(n)}_{norm} P(s)$ is defined inductively on n (and not on the process structure). The reason for this is that structural induction on processes is not possible over a family of processes $P(s)$. The following lemma shows that our transformation up to depth n indeed yields a process in full normal form and is semantics preserving:

Lemma 1. *If $P(s) \in NormProc_\Sigma$ for all $s \in S$, then for any n, $!!\, s : S \bullet^{(n)}_{norm} P(s) \in NormProc_\Sigma$, and $\mathcal{A}_{\mathcal{F}} \vdash\ (!!\, s : S \bullet P(s)) \downarrow n = !!\, s : S \bullet^{(n)}_{norm} P(s)$.*

Proof sketch. By induction on n. The transformation by $\mathcal{A}_{\mathcal{F}}$ is established in three steps: (1) for the first subexpression $((?x : A(s) \rightarrow P(s,x)) \,\Box\, Q(s))$ of $P(s)$, the nondeterminism over S can be rewritten to (external) prefix choice by (!!-input-!set), (!!-□-Dist), and (!!-input-Dist). (2) for the second subexpression $(!set\ X : \mathcal{X}(s) \bullet (?\, x : X \rightarrow Div))$ of $P(s)$, the two nondeterminism by S and $\mathcal{X}(s)$ can be rewritten to one nondeterminism by (!!-!set-div). (3) Finally, (?-!set-⊆) is applied for replacing $\mathcal{X}'$ by $complete(A', \mathcal{X}')$. □

Finally, for each $n \in Nat$ we define a function $Norm_{(n)}(P)$ inductively on the structure of P, see Fig. 5. The following lemma shows that the function $Norm_{(n)}$

$$
\begin{aligned}
&Norm_{(0)}(Pr) = (Div)_{norm} && (Pr \in SeqProc_\Sigma)\\
&Norm_{(n+1)}(P) = !!\, s : S \bullet^{(n+1)}_{norm} Norm_{(n+1)}(R'(s))\\
&Norm_{(n+1)}(R) = (?\, x : A \to (\text{if } x : A \text{ then } Norm_{(n)}(P'(x)) \text{ else } Div)\\
&\qquad\qquad \Box\ (\text{if } (Q = Skip) \text{ then } Skip \text{ else } Div))\\
&\qquad\qquad \sqcap\ (!set\ X : (\text{if } Q = Stop \text{ then } \{A\} \text{ else } \emptyset) \bullet (?\, x : X \to Div))\\
&\text{where } P = !!\, s : S \bullet R'(s) \in SeqProc_\Sigma,\\
&\qquad R = ((?x : A \to P'(x)) \Box Q) \in SeqProc_\Sigma,
\end{aligned}
$$

Fig. 5. Normalising function

transforms a process in full sequential forms whose depth is restricted to n into full normal form.

Lemma 2. *Let $P \in SeqProc_\Sigma$. Then for any n, $Norm_{(n)}(P) \in NormProc_\Sigma$ and $\mathcal{A}_\mathcal{F} \vdash P \downarrow n = Norm_{(n)}(P)$.*

Proof sketch. By induction on the structure of the full sequential form P. If P has the form $!!\, s : S \bullet R'(s)$, it can be normalised by Lemma 1. Otherwise, P is of the form $(?\, x : A \to P'(x)) \Box Q$. If $Q \neq Stop$ then it can be transformed to a full normal form by ($\sqcap$-unit) and (!!-emptyset), otherwise by (?-div). □

With the function $XNorm$ defined as

$$XNorm(P) = !nat\ n \bullet (Norm_{(n)}(Seq(P))),$$

we finally obtain the expected theorem:

Theorem 7. *Let $P \in Proc_\Sigma$. Then, $XNorm(P) \in XNormProc_\Sigma$ and $\mathcal{A}_\mathcal{F} \vdash P = XNorm(P)$.*

Proof sketch. By the law (!nat-↓), Theorem 3, and Lemma 2, $\mathcal{A}_\mathcal{F} \vdash P = !nat\ n \bullet (P \downarrow n) = !nat\ n \bullet (Seq(P) \downarrow n) = !nat\ n \bullet (Norm_{(n)}(Seq(P))) = XNorm(P)$, and for all n, $Norm_{(n)}(Seq(P)) =_\mathcal{F} XNorm(P) \downarrow n$. □

From this follows as a corollary that the axiom system $\mathcal{A}_\mathcal{F}$ is sound and complete for stable-failures equivalence.

Corollary 1. *Let $P, Q \in Proc_\Sigma$. Then, $\mathcal{A}_\mathcal{F} \vdash P = Q$ if and only if $P =_\mathcal{F} Q$.*

Proof. By Theorems 2, 6, and 7 □

At the end of this section, we give an example to show how to normalise divergent infinite processes by the axiom system $\mathcal{A}_\mathcal{F}$.

Example 2. We normalise the divergent infinite process $(Fix\ count)(0)$ by $\mathcal{A}_\mathcal{F}$, where the function $count :: (Nat \Rightarrow Proc_{Nat}) \Rightarrow (Nat \Rightarrow Proc_{Nat})$ is defined as:

$$count\,(f)\,(n) := (n \to f(n+1)) \setminus \{n\}$$

The process $(Fix\ count)(0)$ increases the natural number n from the initial value 0 – which is hidden to the outside world. For $(Fix\ count)(0)$, we can for example

prove the equality: $(Fix\ count)(0) =_{\mathcal{F}} (0 \rightarrow (1 \rightarrow (Fix\ count)(2)) \setminus \{1\}) \setminus \{0\}$. To do so, first expand the fixed point by the law (Tarski-fix),

$$\mathcal{A}_{\mathcal{F}} \vdash (Fix\ count)(0) = !nat\ n \bullet ((count^{(n)}(\lambda\, y.\, Div))(0))$$

Next, we show by induction on n that $\mathcal{A}_{\mathcal{F}} \vdash count^{(n)}(\lambda\, y.\, Div)(m) = Div$ for all n, m. The base case $(n = 0)$ is trivial because $\mathcal{A}_{\mathcal{F}} \vdash count^{(0)}(\lambda\, y.\, Div)(m) = (\lambda\, y.\, Div)(m) = Div$. The induction case $(n + 1)$ is proven as follows:

$$\begin{aligned}
&\mathcal{A}_{\mathcal{F}} \vdash count^{(n+1)}(\lambda\, y.\, Div)(m) \\
&\quad = count(count^{(n)}(\lambda\, y.\, Div))(m) \\
&\quad = (m \rightarrow (count^{(n)}(\lambda\, y.\, Div))(m+1)) \setminus \{m\} \\
&\quad = (m \rightarrow Div) \setminus \{m\} && \text{by induction} \\
&\quad = (?\, x : \emptyset \rightarrow (Div \setminus \{m\})) \triangleright (!\ x : \{m\} \bullet (Div \setminus \{m\})) && \text{by (hide-step)} \\
&\quad = Stop \triangleright (!\ x : \{m\} \bullet (Div \setminus \{m\})) && \text{by (stop-step)} \\
&\quad = !\ x : \{m\} \bullet Div && \text{by (unit-laws)} \\
&\quad = Div && \text{by (!!-const)}
\end{aligned}$$

Finally, since $\mathcal{A}_{\mathcal{F}} \vdash Div = (Div)_{norm} \in NormProc_{\Sigma}$, we have

$$\mathcal{A}_{\mathcal{F}} \vdash (Fix\ count)(0) = !nat\ n \bullet (Div)_{norm} \in XNormProc_{\Sigma},$$

where $(Div)_{norm}$ given in Fig. 4 and $!nat\ n \bullet (Div)_{norm}$ are the full normal form and the extended full normal form of Div, respectively. □

6 Verification by CSP-Prover

The tool Csp-Prover [3,4] provides a deep encoding of Csp in the generic theorem prover Isabelle [7]. Csp-Prover contains fundamental theorems such as fixed point theorems on complete metric spaces and complete partial order, the definitions of Csp syntax and semantics, and many Csp-laws and semi-automatic proof tactics for verification of refinement relation. Therefore, Csp-Prover can be used for

1. Verification of infinite state systems. For example, we applied Csp-Prover to verify a part of the specification of the EP2 system, which is a new industrial standard of electronic payment systems, see [4].
2. Establishing new theorems on Csp. For example, Csp-Prover assisted us in proving the theorems given in this paper.

All proofs (including the examples) given in this paper have been verified by Csp-Prover. However, Csp-Prover also implements the axiom system $\mathcal{A}_{\mathcal{F}}$, besides the verification of this paper. Therefore, it is possible to prove the stable-failures equivalence over processes by syntactical rewriting with Csp-Prover.

In Isabelle, theorems, together with definitions and proof-scripts needed for their proof, can be stored in *theory-files*. Currently, Csp-Prover consists of three packages of theory-files: `CSP`, `CSP_T`, and `CSP_F`. The package `CSP` is the reusable part independent of specific Csp models. For example, it contains fixed point theorems on cms and cpo, and the definition of Csp syntax. The packages `CSP_T` and `CSP_F` are instantiated parts for the traces model and the stable failures model.

The packages have a hierarchical organisation as: `CSP_F` on `CSP_T` on `CSP` on Isabelle/HOL-Complex. The total number of lines of theory-files in `CSP`, `CSP_T`, and `CSP_F` are about 12,000 lines, 11,000 lines, and 18,000 lines, respectively.

The theorems for sequentialisation and normalisation given in this paper are stored in a new package `FNF_F` implemented on `CSP_F`. The total line number of theory-files in `FNF_F` is about 6,000 lines. All the packages can be downloaded from the web-site [3] of Csp-Prover.

7 Conclusion

We have shown that the Csp-dialect under discussion has the same expressive power as full Csp. We also presented a sound and complete axiom system $\mathcal{A}_{\mathcal{F}}$ of stable-failures equivalence for processes with unbounded nondeterminism over an arbitrary (possibly infinite) alphabet. The theorems presented in this paper have been verified by Csp-Prover.

Our results are of practical relevance for theorem proving for Csp in general: besides having a complete axiom system available, it is also possible to base proof rules and tactics on the extended full normal form. On the theoretical side, the here presented axioms, transformations, and normal forms provide new insight into the semantics of the process algebra Csp.

Acknowledgement. The authors are grateful to Erwin R. Catesbeiana Jr for pointing out the incompleteness problem in the first place and for good advice on how to avoid inconsistencies in the axiom system.

References

1. B. Dutertre and S. Schneider. Using a PVS embedding of CSP to verify authentication protocols. In E. L. Gunter and A. P. Felty, editors, *TPHOL 1997*, LNCS 1275, pages 121–136. Springer, 1997.
2. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
3. Y. Isobe and M. Roggenbach. Webpage on Csp-Prover. `http://staff.aist.go.jp/y-isobe/CSP-Prover/CSP-Prover.html`.
4. Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. In N. Halbwachs and L. D. Zuck, editors, *TACAS 2005*, LNCS 3440, pages 108–123. Springer, 2005.
5. Y. Isobe, M. Roggenbach, and S. Gruner. Extending CSP-Prover by deadlock-analysis: Towards the verification of systolic arrays. In *FOSE 2005*, Japanese Lecture Notes Series 31, pages 257–266. Kindai-kagaku-sha, 2005.
6. F. S. E. Limited. Failures-divergence refinement: FDR2. `http://www.fsel.com/`.
7. L. C. Paulson. *A Generic Theorem Prover*. LNCS 828. Springer, 1994.
8. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998. Or No.68 in `http://web.comlab.ox.ac.uk/oucl/work/bill.roscoe/pubs.html`.
9. S. Schneider. Verifying authentication protocol implementations. In B. Jacobs and A. Rensink, editors, *FMOODS 2002*, volume 209 of *IFIP Conference Proceedings*, pages 5–24. Kluwer, 2002.
10. H. Tej and B. Wolff. A corrected failure-divergence model for CSP in Isabelle/HOL. In J. Fitzgerald, C. Jones, and P. Lucas, editors, *FME'97*, LNCS 1313, pages 318–337. Springer, 1997.