A Corrected Failure-Divergence Model for CSP in Isabelle/HOL[1]

H. Tej, B. Wolff

Universität Bremen, FB3
Postfach 330440
D-28334 Bremen
{bu,ht}@informatik.uni-bremen.de

Abstract. We present a failure-divergence model for CSP following the concepts of [BR 85]. Its formal representation within higher order logic in the theorem prover Isabelle/HOL [Pau 94] revealed an error in the basic definition of CSP concerning the treatment of the termination symbol tick.
A corrected model has been formally proven consistent with Isabelle/-HOL. Moreover, the changed version maintains the essential algebraic properties of CSP. As a result, there is a proven correct implementation of a "CSP workbench" within Isabelle.

1 Introduction

In his invited lecture at FME'96, C.A.R. Hoare presented his view on the status quo of formal methods in industry. With respect to formal proof methods, he ruled that they "are now sufficiently advanced that a [...] formal methodologist could occasionally detect [...] obscure latent errors before they occur in practice" and asked for their publication as a possible "milestone in the acceptance of formal methods" in industry.

In this paper, we report of a larger verification effort as part of the UniForM project [Kri+95]. It revealed an obscure latent error that was not detected within a decade. It can not be said that the object of interest is a "large software system" whose failure may "cost millions", but it is a well-known subject in the center of academic interest considered foundational for several formal methods *tools*: the theory of the failure-divergence model of CSP ([Hoa 85], [BR 85]). And indeed we hope that this work may further encourage the use of formal proof methods at least in the academic community working on formal methods.

Implementations of proof support for a formal method can roughly be divided into two categories. In *direct tools* like FDR [For 95], the logical rules of a method (possibly integrated into complex proof techniques) are hard-wired into the code of their implementation. Such tools tend to be difficult to modify and to formally reason about, but can possess enviable automatic proof power in specific problem domains and comfortable user interfaces.

---

1 This work has been supported by the German Ministery for Education and Research (BMBF) as part of the project **UniForM** under grant No. FKZ 01 IS 521 B2.

The other category can be labelled as *logical embeddings*. Formal methods such as CSP or Z can be logically embedded into an LCF-style tactical theorem prover such as *HOL* [GM 93] or *Isabelle*[Pau94]. Coming with an open system design going back to Milner, these provers allow for user-programmed extensions in a logically sound way. Their strength is flexibility, generality and expressiveness that makes them to *symbolic programming environments*.

In this paper we present a tool of the latter category (as a step towards a future combination with the former). After a brief introduction into the failure divergence semantics in the traditional CSP-literature, we will discuss the revealed problems and present a correction. Although the error is not "mathematically deep", it stings since its correction affects many definitions. It is shown that the corrected CSP still fulfils the desired algebraic laws. The addition of fixpoint-theory and specialised tactics extends the embedding in Isabelle/HOL to a formally proven consistent proof environment for CSP. Its use is demonstrated in a final example.

2 The Failure Divergence Semantics

In this section, we follow closely the presentation of [Cam 91], whose contribution is a formal, machine-assisted version of a subset of CSP based on [BR 85] and [Ros 88] *without* the sequential operator, the parallel interleave operator and a proof-theory based on fixpoint induction. With [Cam 91], we share some major design decisions, in particular the choice of the alternative process ordering in [Ros 88] (see below).

In its trace semantics model it is not possible to describe certain concepts that commonly arise when reasoning about concurrent programs. In particular, it is not possible to express non-determinism, or to distinguish deadlock from infinite internal activity. The failure-divergence model incorporates the information available in the trace-semantics, and in addition introduces the notions of *refusal* and *divergence* to model such concepts.

Example 2.1: Non-Determinism

Let *a* and *b* be any two events in some set of events Σ. The two processes

$$(a \rightarrow Stop) \,\Box\, (b \rightarrow Stop) \tag{1}$$

and

$$(a \rightarrow Stop) \sqcap (b \rightarrow Stop) \tag{2}$$

cannot be distinguished under the trace semantics, in which both processes are capable of performing the same sequences of events, i.e. both have the same set of traces $\{\langle\rangle,\langle a\rangle,\langle b\rangle\}$. This is because both processes can either engage in *a* and then *Stop*, or engage in *b* and then *Stop*. We would, however, like to distinguish between a deterministic choice of *a* or *b* (1) and a non-deterministic choice of *a* or *b* (2).

This can be done by considering the events that a process can refuse to engage in when these events are offered by the environment; it cannot refuse either, so we say its *maximal refusal set* is the set containing all elements of Σ other than *a* and *b*, written $\Sigma\backslash\{a,b\}$, i.e. it can refuse all elements in Σ other than *a* or *b*. In the case of the non-deterministic process (2), however, we wish to express that if the environment offers the event *a* say, the process non-deterministically chooses either to engage in *a*,

/ to refuse it and engage in *b* (likewise for b). We say therefore, that process (2) has two maximal refusal sets, $\Sigma\backslash\{a\}$ and $\Sigma\backslash\{b\}$, because it can refuse to engage in either *a* or *b*, but not both. The notion of refusal sets is in this way used to distinguish non-determinism from determinism.

Example 2.2: Infinite Chatter

Consider the infinite process

$$\mu\ X.\ a \rightarrow X$$

which performs an infinite stream of *a*'s. If one now conceals the event *a* in this process by writing

$$(\mu\ X.\ a \rightarrow X) \setminus a \qquad (3)$$

it no longer becomes possible to distinguish the behaviour of this process from that of the deadlock process *Stop*. We would like to be able to make such a distinction, since the former process has clearly not stopped but is engaging in an unbounded sequence of internal actions invisible to the environment. We say the process has diverged, and introduce the notion of a *divergence set* to denote all sequences events that can cause a process to diverge. Hence, the process *Stop* is assigned the divergence set {}, since it can not diverge, whereas the process (3) above diverges on any sequence of events since the process begins to diverge immediately, i.e. its divergence set is Σ^*, where Σ^* denotes the set of all sequences with elements in Σ. Divergence is undesirable and so it is essential to be able to express it to ensure that it is avoided.

2.3 The Original Version of CSP-Semantics

The Semantic Domain. In the model of CSP presented in [BR 85] a process communicates with its environment by engaging in events drawn from some *alphabet* Σ. In the failure-divergence semantics a process is characterised by:

- its *failures* — these are sets of pairs (s,X), where s is a possible sequence of events a process can engage in (a *trace*), and X is the set of events that process can refuse to engage in (the *refusals*) after having engaged in *s*,
- its *divergences* — these are the traces after which a process may diverge.

Processes are therefore represented by pairs *(F,D)*, where *F* is a failure set and *D* is a divergence set.

The failures and divergences of a process must satisfy six well-definedness conditions (following [Ros 88]): (i) the initial trace of a process must be empty, (ii) the prefixes of all traces of a process are themselves traces of that process, i.e. traces are *prefix-closed*, (iii) a process can refuse all subsets of a refusal set, (iv) all events which are impossible to perform in the next step can be included in a refusal set, (v) a divergence set is *suffix closed*, and (vi) once a process has diverged, it can engage in, or refuse, any sequence of events.

More formally, given a (possibly infinite) set of events Σ and sets *F* and *D* such that:

$$F \subseteq \Sigma^* \times \mathbb{P}(\Sigma)$$
$$D \subseteq \Sigma^*$$

then using a set theory and predicate calculus notation similar to that adopted in [Ros 88], the above six well-definedness conditions for processes are stated as:

$$\text{is\_well\_defined}(F,D) \overset{\text{def}}{\Leftrightarrow}$$

$$(\langle\rangle,\{\}) \in F \qquad \text{(i)}$$

$$\wedge \quad \forall s,t.\ (s \hat{}\ t, \{\}) \in F \Rightarrow (s, \{\}) \in F \qquad \text{(ii)}$$

$$\wedge \quad \forall s,X,Y.\ (s, X) \in F \wedge Y \subseteq X \Rightarrow (s, Y) \in F \qquad \text{(iii)}$$

$$\wedge \quad \forall s,X,Y.\ (s, X) \in F \wedge (\forall c \in Y.\ ((s \hat{}\ \langle c\rangle,\ \{\}) \notin F)) \Rightarrow (s, X \cup Y) \in F \qquad \text{(iv)}$$

$$\wedge \quad \forall s,t.\ s \in D \Rightarrow s \hat{}\ t \in D \qquad \text{(v)}$$

$$\wedge \quad \forall s,t.\ s \in D \Rightarrow (s \hat{}\ t, X) \in F \qquad \text{(vi)}$$

where $\langle\rangle$ denote the empty trace, and the notation $s \hat{}\ t$ is used to represent the concatenation of two traces *s* and *t*.

In the model originally presented in [BR 85], the converse of (iii) is also a well-definedness condition. This condition, which is shown formally below, states that a set is refusable if all its finite subsets are refusable:

$$(\forall Y \in \mathbb{F}(X).\ (s,Y) \in F) \Rightarrow (s,X) \in F \qquad (4)$$

In [Ros 88], Roscoe explains that this condition can in fact be omitted from the definition of process, but if this is done, a coarser, more complex ordering on processes must be defined since the ordering used in [BR 85] is no longer a *complete* partial order otherwise. This ordering will be called *process ordering.*

We prefer to use the process ordering of [Ros 88] (extended in [RB 89]), since we plan to investigate combinations of Z and CSP. In such a combination, Z is used to specify system-transitions via pre-and postconditions. Therefore we need a model that can cope with unbounded nondeterminism. In such a setting, a separation of the process-ordering from the refinement ordering seems unavoidable (see the detailed discussion of the counter examples in [Ros 88]). We will show in chapter 5 how one can live with two orderings from a proof theory point of view.

The semantics of the operators. We will consider the set of well-formed processes over alphabet Σ as a type *process Σ.* Then the language of CSP can now be given by the following signature (using infix notation):

Stop, Skip	: process Σ	
$\_\rightarrow\_$	: $\Sigma \times$ process $\Sigma \rightarrow$ process Σ	(* prefix of single event *)
$\Box\_:\_\rightarrow\_$	: $\mathbb{P}\ \Sigma \times (\Sigma \rightarrow$ process $\Sigma) \rightarrow$ process Σ	(* multi-prefix of events*)
$\_\backslash\_$	: process $\Sigma \times \mathbb{P}\ \Sigma \rightarrow$ process Σ	(* hide *)
$\_\Box\_$, $\_\sqcap\_$,		(* det. & non-det. choice *)
$\_\ ;\_$	: process $\Sigma \times$ process $\Sigma \rightarrow$ process Σ	(* seq. composition*)
$\mu\ \_$	: (process $\Sigma \rightarrow$ process $\Sigma) \rightarrow$ process Σ	(* recursion *)
…		

The signature above is the precise equivalent of the "grammar" in [BR 85] in higher-order abstract syntax. We will write $\Box x : A \rightarrow P\ x$ for $(\Box\_:\_\rightarrow\_)\ A\ (\lambda x.P\ x)$.

In the traditional CSP-literature, a distinguishable particular element $\surd$ ("tick") within Σ is required. It is used to indicate the termination of a process. It is crucial that it has not been distinguished on the level of the definition of the semantic domains.

Let $\mathcal{D}$ be the projection into the divergences and $\mathcal{F}$ be the projection into the failures of a process. The semantics for the CSP-operators can now be given following the lines of the example below:

$$\begin{aligned}
\mathcal{D}(P;Q) &= \mathcal{D}\,P \cup \{\, s \hat{} t \mid \text{tick-free}(s) \wedge s\hat{}\langle\surd\rangle \in \text{traces}(\mathcal{F}\,P) \wedge t \in \mathcal{D}\,Q\} \\
\mathcal{F}(P;Q) &= \{(s, X) \mid \text{tick-free}(s) \wedge (s, X \cup \{\surd\}) \in \mathcal{F}\,P\} \\
&\cup \{(s \hat{} t, X) \mid (s\hat{}\langle\surd\rangle,\{\}) \in \mathcal{F}\,P \wedge \text{tick-free}(s) \wedge (t,X) \in \mathcal{F}\,Q) \\
&\cup \{(s, X) \mid s \in \mathcal{D}\,P)\}
\end{aligned}$$

where *traces* denotes the projection into the traces and the predicate *tick-free* discriminates traces not containing tick.

Of course, for any operator it has to be shown that the results of $\mathcal{D}$ and $\mathcal{F}$, when composed to a pair, form in fact an object of type *process*, i.e. it remains to be shown that failures and divergences produced by an operator according to definitions above respect the well-formedness condition of the semantical domain, i.e. *is_well_defined($\mathcal{F}P$, $\mathcal{D}P$)*. (Theorem 1 in [BR 85]).

In fact, this is not possible for the sequential operator.

2.4 The Problem

The problem is that from the definition one can not prove the following part of *is_well_defined*:

$$(s \hat{} t,\{\}) \in \mathcal{F}(P;Q) \Rightarrow (s,\{\}) \in \mathcal{F}(P;Q)$$

Consider the following case:

- $(s \hat{} t, \{\}) \in \mathcal{F}(P;Q)$ and
- $s \hat{} t \in \mathcal{D}\,P$ and
- $s \notin \mathcal{D}\,P$ and
- s is not tick-free, i.e there exist s' and s'' such that $s = s' \hat{} \langle\surd\rangle \hat{} s''$

From the definition for ; and the *is_well_defined* we can only prove that:

$$(s',\{\}) \in \mathcal{F}(P;Q)$$

but we can say nothing about *(s,{})*.

The problem is independent from axiom (4).

Conceptually, this is a consequence of an incoherent treatment of tick-freeness in divergence sets and failure sets. Although this is extremely ugly, our intuition that ticks "can appear only at the end of a trace" ([Hoa 85], pp.57, paragraph 1.9.7) has to be formally represented in the notion of well-formedness (which was, to our knowledge, never done in the CSP-Literature).

This means that the sequential operator of CSP in the sense of the definition does not form a process. This problem has meanwhile been recognised by other researchers of the CSP community ([Ros 96]), together with the fact that the problem ranges over "traditional CSP literature". Roscoe independently found this error recently and proposes a solution similar to ours.

3 Isabelle/HOL

3.1 Higher Order Logic (HOL)

In this section, we will give a short overview of the concepts and the syntax. Our logical language HOL goes back to [Chu 40]; a more recent presentation is [And 86]. In the formal methods community, it has achieved some acceptance, especially in hardware-verification. HOL is a classical logic with equality formed over the usual logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and = for negation, conjunction, disjunction, implication and equality. It is based on total functions denoted by λ-abstractions like "$\lambda x.x$". Function application is denoted by $f\ a$. Every term in the logic must be typed, in order to avoid Russels paradox. Isabelle's type discipline incorporates polymorphism with type-classes (as in Haskell). HOL extends predicate calculus in that universal and existential quantification $\forall$x. P x rsp. $\exists$x. P x can range over functions.

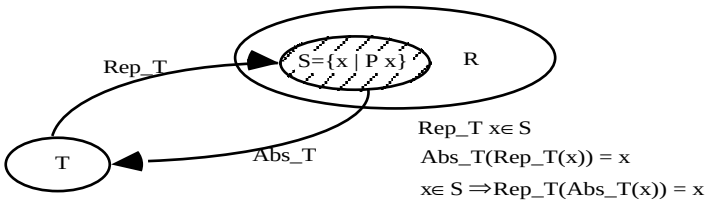
I3.2 Conservative Extensions in HOL

The introduction of new axioms while building a new theory may easily lead to inconsistency. Here, a *theory* is a pair of a signature Σ and a set of formulas *Ax* (the axioms). A theory extension can be characterised by a relation on theories:

$$(\Sigma, Ax) \rightsquigarrow (\Sigma \cup \Delta\Sigma, Ax \cup \Delta Ax)$$

Fortunately there are a number of syntactic schemes for theory extensions that maintain the consistency of the extended one — such schemes are called *conservative extensions schemes*. (For a more formal account the reader is referred to [GM 93]; one may also find a proof of soundness there). Some syntactic schemes for theory-increments $\Delta\Sigma$ and ΔAx are:

- the *constant definition* $c \equiv t$ of a fresh constant symbol c and a closed expression t not containing c and not containing a free type variable that does not occur in the type of c,
- the *type definition* (a set of axioms stating an isomorphism between a non-empty subset $S = \{x :: R \mid P\ x\}$ of a base-type R and the type T to be defined),



- a set of equations forming a *primitive recursive scheme* over a fresh constant symbol f.

The basic idea of these extension schemes is to avoid logical paradoxes by avoiding general recursive axioms provoking them. Desired properties have to be derived from conservative extensions. We will build up all theories by the above extension schemes, which constitutes a consistency proof w.r.t. HOL.

3.3 Isabelle

Isabelle is a *generic* theorem prover that supports a number of logics, among them first-order logic (FOL), Zermelo-Fränkel set theory (ZF), constructive type theory (CTT), the Logic of Computable Functions (LCF), and others. We only use its set-up for higher order logic (HOL). Isabelle supports natural deduction style. Its principal inference techniques are resolution (based on higher-order unification) and term-rewriting. Isabelle provides syntax for hierarchical theories (containing signatures and axioms).

In the sequel, all Isabelle input and output will be denoted in this FONT throughout this paper — enriched by the usual mathematical notation for ∀, ∃,... instead of ASCII-transcriptions.[2]

Isabelle belongs to the family of LCF-style theorem provers. This means it is a set of data types and function definitions in the ML-environment (or: "data-base"). The crucial one is the abstract data type "thm" (protected by the ML type discipline) that contains the formulas accepted by Isabelle as theorems. thm-objects can only be constructed via operations of the logical kernel of Isabelle. This architecture allows to provide user-programmed extensions of Isabelle without corrupting the logical kernel.

Technically, the proofs were done by ML-scripts performing sequences of kernel operations. These scripts were attached to the theory documents that constitute a larger system of theories, "the CSP-theory" in our case. While Isabelle is loading the theory documents and checking the proof-scripts, Isabelle can produce an HTML-document allowing to browse the CSP-theory.

4 Formalising CSP Semantics in HOL

Our formalisation of CSP profits from the powerful logical language HOL in several aspects:

- Higher order abstract syntax leads to a more compact notation avoiding auxiliary instruments like environments, updates, substitutions and process-and alphabet-variables. These issues were handled uniformly and precisely by the type-discipline.
- The data type invariant is_process (corresponding to is_well_defined in 2.3.1) can be encapsulated within a HOL-type. This leads to explicit treatment of notational assumptions and makes them amenable to static type checking.
- As we will see in the next chapter, HOL can cope with the issue of *admissibility* (as a prerequisite for fixpoint induction) in an extremely elegant way.

4.1 A Corrected Version for CSP Semantics

Whenever we changed a definition or a theorem, we will mark this by * in the sequel. The modified process invariant reads as follows[3]:

---

2 We do not distinguish quantifications and implications at the different logical levels throughout this paper; see [Pau 94].

3 In his "Notes on CSP" [Ros 96], Roscoe proposes two additional conditions. We have also proved formally that the CSP-theory is consistent on this basis.

$$\text{is\_process}(F, D) \stackrel{\text{def}}{\Leftrightarrow}$$

$$\begin{array}{lll}
\wedge & (\langle\rangle,\{\}) \in F & \text{(i)} \\
\wedge & \forall s,X.\ (s,X) \in F \Rightarrow \text{front-tick-free}(s) & (*) \\
\wedge & \forall s,t.\ (s \hat{}\ t, \{\}) \in F \Rightarrow (s, \{\}) \in F & \text{(ii)} \\
\wedge & \forall s,X,Y.\ (s, X) \in F \wedge Y \subseteq X \Rightarrow (s, Y) \in F & \text{(iii)} \\
\wedge & \forall s,X,Y.\ (s, X) \in F \wedge (\forall c \in Y.\ ((s \hat{}\ \langle c\rangle,\{\}) \notin F)) \Rightarrow (s, X \cup Y) \in F & \text{(iv)} \\
\wedge & \forall s,t.\ s \in D \wedge \text{tick-free}(s) \wedge \text{front-tick-free}(t) \Rightarrow s \hat{}\ t \in D & \text{(v*)} \\
\wedge & \forall s,t.\ s \in D \Rightarrow (s, X) \in F & \text{(vi)}
\end{array}$$

The condition * requires all traces to be front-tick-free (i.e. $\surd$ can occur at most at the end of a trace). Note that from (v *) and (vi) follows also front-tick-freeness for all divergences.

4.2 The Type Process

The encapsulation of the data type invariants *is_process* of the previous section within a type is accomplished by a type definition (see section 3.2). Note that Isabelle's notation for type constructor instances differs from the one used throughout this paper.

We introduce a type abbreviation trace Σ as synonym for $(\Sigma \oplus \{\surd\})^*$ where $\oplus$ denotes the disjoint sum on sets. Further, a type abbreviation p Σ will be used for the product of failures and divergences:

$$\text{p}\ \Sigma \stackrel{\text{def}}{=} \mathbb{P}(\text{trace}\ \Sigma \times \mathbb{P}\ (\Sigma \oplus \{\surd\})) \times \mathbb{P}\ (\text{trace}\ \Sigma)$$

The set of all these tuples of p Σ represents the base type R of the type definition scheme. According to this extension scheme, fresh constant symbols are introduced:

$$\begin{array}{l}
\text{Abs_process} : \text{p}\ \Sigma \rightarrow \text{process}\ \Sigma \\
\text{Rep_process} : \text{process}\ \Sigma \rightarrow \text{p}\ \Sigma
\end{array}$$

together with the new axioms:

$$\begin{array}{lll}
\text{Rep_process}\ X : \{p.\ \text{is_process}\ p\} & & \text{(Rep_process)} \\
\text{Abs_process}(\text{Rep_process}(X)) = X & & \text{(Abs_inverse)} \\
\text{is_process}\ X \Rightarrow \text{Rep_process}(\text{Abs_process}(X)) = X & & \text{(Rep_inverse)}
\end{array}$$

In Isabelle, this whole instance of the conservative extension scheme is abbreviated with the following statement in the theory ProcessType:

```
subtype (process) process Σ= "{p. is_process p}"
```

A first important theorem of this extension is[4]:

$$\text{is\_process}\ (\text{Rep\_process}\ P) \qquad \text{(is\_process\_Rep)}$$

---

[4] In fact, the methodology entails a proof obligation that the type is non empty, i.e. that there is a witness for which is_process holds. This trivial proof is omitted here.

We proceed with the definitions of the projections for failures and divergences:

$\mathcal{D}\,P \equiv$ snd (Rep_process P)
$\mathcal{F}\,P \equiv$ fst (Rep_process P)

where fst and snd are the usual projections into cartesian products.

The encapsulation of well-formedness within a type has the price that the constant definitions of the semantic operators are slightly unconventional. The definition of the prefix-operator in Isabelle theory notation, for instance, proceeds as follows:

```
Prefix = ProcessType +
consts   "→"  :: Σ → process Σ → process Σ            (infix 75)
defs
     Prefix_def    " a → P ≡
          Abs_process({(s,X) | s = ⟨⟩ ∧ ev a ∉ X} ∪
                          {(s, X) | s ≠ ⟨⟩ ∧ hd s = ev a ∧ (tl s,X)∈ ℱ P},
                          {d | d ≠ ⟨⟩ ∧ hd d = ev a ∧ tl s ∈ 𝒟 P} )"
end
```

The first line indicates that the theory Prefixis a hierarchical extension of the theory ProcessType. The pragma (infix 75) sets up Isabelle's powerful parsing machinery to parse the prefix operator the way it is used throughout this paper. The next axiom is declared to be a constant definition (Isabelle checks the syntactic side conditions) containing the abstracted tuple of failures and divergences, where hd and tl are the usual projection in lists and ev is just the injection of an element into $\Sigma \oplus \{\surd\}$.

From this definition, the traditional equations for $\mathcal{F}$ and $\mathcal{D}$ are *derived* as theorems:

$\mathcal{D}(a \rightarrow P)$ = {(⟨ev a⟩^s, X) | s ∈ $\mathcal{D}$ P}
$\mathcal{F}(a \rightarrow P)$ = {(⟨⟩,X) | ev a ∉ X}} ∪ {(⟨ev a⟩^s,X) | (s,X) ∈ $\mathcal{F}$ P}

The proof requires Rep_inverse and hence a proof of is_process for the prefix operator. We follow this technique to develop conservatively $\mathcal{F}$ and $\mathcal{D}$ for all operators.

4.3 The Semantics of the CSP-Operators

In the sequel, we will omit the technical definitions like Prefix_def and start with a listing of the derived theorems for the process projections, bearing in mind that they already subsume the proof of process well-formedness.

$\mathcal{D}$(Bot) = {d | front-tick-free d} (*)

$\mathcal{F}$(Bot) = {(s,X) | front-tick-free s} (*)

$\mathcal{D}$(Skip) = {}

$\mathcal{F}$(Skip) = {(⟨⟩,X) | $\surd \notin$ X} $\cup$ {(⟨$\surd$⟩,X)}

$\mathcal{D}$(Stop) = {}

$\mathcal{F}$(Stop) = {(⟨⟩,X)}

$\mathcal{D}$(□x:A→P x)= {(⟨ev a⟩^s, X) | a $\in$ A $\wedge$ s $\in$ $\mathcal{D}$ P}

$\mathcal{F}$(□x:A→P x)= {(⟨⟩,X) | X $\cap$ ev A = {}}

$\cup$ {(⟨ev a⟩^s,X) | a $\in$ A $\wedge$ (s,X) $\in$ $\mathcal{F}$ P}

$\mathcal{D}$(P ; Q) = $\mathcal{D}$ P $\cup$ {s ^ t | s^⟨$\surd$⟩ $\in$ traces($\mathcal{F}$ P) $\wedge$ t $\in$ $\mathcal{D}$ Q} (*)

$\mathcal{F}$(P ; Q) = {(s,X) | tick-free(s) $\wedge$ (s, X $\cup${$\surd$}) $\in$ $\mathcal{F}$ P}

$\cup$ {(s^t,X) | (s^⟨$\surd$⟩,{}) $\in$ $\mathcal{F}$ P $\wedge$ (t,X) $\in$ $\mathcal{F}$ Q}

$\cup$ {(s,X) | s $\in$ $\mathcal{D}$(P;Q)} (*)

$\mathcal{D}$(P ⊓ Q) = $\mathcal{D}$ P $\cup$ $\mathcal{D}$ Q

$\mathcal{F}$(P ⊓ Q) = $\mathcal{F}$ P $\cup$ $\mathcal{F}$ Q

$\mathcal{D}$(P □ Q) = $\mathcal{D}$ P $\cup$ $\mathcal{D}$ Q

$\mathcal{F}$(P □ Q) = {(⟨⟩,X) | (⟨⟩,X) $\in$ $\mathcal{F}$ P $\cap$ $\mathcal{F}$ Q}

$\cup$ {(s,X) | s $\neq$ ⟨⟩ $\wedge$ (s,X) $\in$ $\mathcal{F}$ P $\cup$ $\mathcal{F}$ Q}

$\cup$ {(⟨⟩,X) | ⟨⟩ $\in$ $\mathcal{D}$ P $\cup$ $\mathcal{D}$ Q}

$\mathcal{D}$(P \ A) = {s | $\exists$ t u.front-tick-free u $\wedge$ s=(hide t(ev A))^u $\wedge$
(t $\in$ $\mathcal{D}$ P $\wedge$ (u = ⟨⟩ $\vee$ tick-free t) $\vee$
($\exists$ M. M $\notin$ $\mathbb{F}$ {x.True} $\wedge$ t $\in$ M $\wedge$
($\forall$ w$\in$M,w'$\in$M.t$\leq$w $\wedge$ (w$\leq$w' $\vee$ w'$\leq$w)) $\wedge$
($\forall$ w $\in$ M. hide w (ev A) = hide t (ev A) $\wedge$
w $\in$ traces($\mathcal{F}$ P))))} (*)

$\mathcal{F}$(P \ A) = {(s, X) | $\exists$ t. s = hide t (ev A) $\wedge$
(t, X $\cup$ ev A) $\in$ $\mathcal{F}$ P}

$\cup$ {(s, X) | s $\in$ $\mathcal{D}$(P\A) } (*)

$\mathcal{D}$(P [|A |] Q) = {s | $\exists$ t u r w. front-tick-free w $\wedge$
(tick-free r $\vee$ w=⟨⟩) $\wedge$ s = r^w $\wedge$
r inter((t,u),(ev A) $\cup$ {$\surd$}) $\wedge$
(t $\in$ $\mathcal{D}$ P $\wedge$ u $\in$ traces($\mathcal{F}$ Q) $\vee$
t $\in$ $\mathcal{D}$ Q $\wedge$ u $\in$ traces($\mathcal{F}$ P))} (*)

$\mathcal{F}$(P [|A |] Q) = {(s,R) | $\exists$ t u X Y. (t,X) $\in$ $\mathcal{F}$ P $\wedge$ (u,Y) $\in$ $\mathcal{F}$ Q $\wedge$
s inter ((t,u), (ev A) $\cup$ {$\surd$}) $\wedge$
R=((X$\cup$Y) $\cap$ ((ev A) $\cup$ {$\surd$})) $\cup$ X $\cap$ Y}

$\cup$ {(s,R) | s $\in$ $\mathcal{D}$(P [|A |] Q)} (*)

hide t A yields the trace obtained from t when concealing all events contained in A. The expression r inter ((t,u),A) means that r is obtained from t and u by synchronising their events which are contained in A and interleaving those which are not.

We adopt the more recent concept of the parallel interleave operator P[|A |]Q from the CSP-literature and define the parallel operator and the interleave operator as special cases:

$$P \;|||\; Q \equiv P[|\,\{\}\,|]\; Q \qquad\qquad P \;||\; Q \equiv P\; [|\{x \mid True\}|]\; Q$$

4.4 The Generic Theory of Fixpoints

The keystone of any denotational semantics is its fixpoint theory that gives semantics to systems of (mutual) recursive equations. Meanwhile, many embeddings of denotational constructions in HOL-Systems have been described in the literature; in the Isabelle/HOL world alone, there is HOLCF [Reg 94]. However, HOLCF is a logic of *continuous functions*, while the fixpoint-theory is only a very small part of it. In contrast to HOLCF, we aim at a more lightweight approach that is parameterized (generic) with the underlying domain-theory (here: processes). Beyond the advantage of a separation of concerns, this paves the way for the reuse of this theory in other problem domains and for a future combination of CSP with pure functional programming. It is also possible with little effort to exchange the fixpoint-theory by another, for example, based on metric spaces via Banach-fixpoints.

Our formalisation of fixpoint theory in HOL will use a particular concept of Isabelle/HOL, namely polymorphism with (axiomatic) *type classes*. This is a constraint on a type variable (similar to the functional programming language Haskell) restricting it to the class of types fulfilling certain syntactic and semantic requirements.

For example, the type class $\alpha :: po$ (partial ordering) can restrict the class of all types α to those for which there is a symbol $\leq : \alpha \times \alpha \to bool$ that enjoy the property $x \leq x$ (refl_ord), $x \leq y \wedge y \leq x \Rightarrow x = y$ (antisym_ord) and $x \leq y \wedge y \leq z \Rightarrow x \leq z$ (trans_ord). Showing that a particular type (say nat with its standard ordering $\leq$) is an *instance* of this type-class, i.e. nat::po is a legal type assertion, requires the proof of the above properties follow from the definition of $\leq : nat \times nat \to bool$. Once this proof has been done while establishing the instance judgement, Isabelle can use this semantic information during static type checking.

We apply this construction to the class cpo that is an extension of po. It requires the symbol $\bot : \alpha$::cpo and the semantic properties $\bot \leq x$ (least) and directed $X \Rightarrow X \neq \{\} \wedge \exists b.\ X <<|\ b$ (complete). Here, directed : (a::po) set $\to$ bool and "is least upper bound" _<<|_:(a::po)set $\to$ a $\to$ bool are defined in the usual way for the class of partial orderings, together with lub : (a::po)set $\to \alpha$ defined as lub S $\equiv \varepsilon x.\ S <<|\ x$. For the class of cpo's, the crucial notions for continuity cont : (α::cpo $\to \beta$::cpo) $\to$ bool and the fixpoint operator fix : (α::cpo $\to \alpha$) $\to \alpha$ are defined in the usual way.

From the definition of continuity it is easy to show several proof-rules like cont(λx.x) (cont_id) and cont(λx.c) (cont_const_fun), stating the identity or any constant function to be continuous.

The first key result of the fixpoint theory is the proof of the fixpoint theorem:

$$cont\ f \Rightarrow fix\ f = f(fix\ f)$$

from the definition of fix f ≡ lub($\bigcup_{i\in\mathbb{N}}$ $f^i\bot$). The second key result is the fixpoint induction theorem, that can be used as general proof principle (see chapter 5).

A third result consists in the fact that the definitions $x \le y \equiv fst\ x \le fst\ y \wedge snd\ x \le snd\ y$ and $\bot \equiv (\bot, \bot)$ extend cpo's to product cpo's. From these definition the instance judgement for the type constructor "×" itself can be proved:

instance "×" : (cpo,cpo)cpo

On this basis Isabelle's parser can parse mutual recursive definitions of the scheme:

letrec $x_1 = E_1(x_1,\ldots,x_n)$
. . .
$x_n = E_n(x_1,\ldots,x_n)$
in $F(x_1,\ldots,x_n)$

as $let(x_1,\ldots,x_n)=fix\ \lambda(x_1,\ldots,x_n).(E_1(x_1,\ldots,x_n),\ldots,E_n(x_1,\ldots,x_n))\ in\ F(x_1,\ldots,x_n)$. Note that the necessary inference that $(x_1,\ldots,x_n)$ forms a cpo is done by Isabelles type inference and not by tactical theorem proving.

Similarly, the usual extension of cpo's to function spaces can be constructed. This adds arbitrary abstractions to an instance of the fixpoint theory with a concrete language; for CSP, this means an optional extension to "Higher Order CSP" allowing the expression of process schemes within this language (similar algorithmic schemes like *map* and *fold* in functional programming languages).

4.5 The Process Instance of the Fixpoint Theory

The crucial point of the instantiation is the definition of the process ordering. As already mentioned, instead of the usual refinement ordering (which is a partial ordering):

$$P \sqsubseteq Q \equiv \mathcal{F}P \supseteq \mathcal{F}Q \wedge \mathcal{D}P \supseteq \mathcal{D}Q$$

we use the more complex process ordering of [Ros 88] since otherwise the operators will not be continuous in presence of unbounded nondeterminism. A prerequisite is the definition "refusals after" $\mathcal{R}$: process $\Sigma \to$ trace $\Sigma \to \mathbb{P}(\mathbb{P}\ (\Sigma \oplus \{\surd\}))$:

$$\mathcal{R}P\ s \equiv \{\ X \mid (s,X) \in \mathcal{F}\ P\}$$

Then the process ordering is introduced as:

$$\begin{aligned} P \le Q \equiv\ & \mathcal{D}P \supseteq \mathcal{D}Q \\ & \wedge\ s \notin \mathcal{D}P \Rightarrow \mathcal{R}P\ s = \mathcal{R}Q\ s \\ & \wedge\ \mu(\mathcal{D}P) \subseteq traces\ \mathcal{F}Q \end{aligned}$$

where μ T denotes the set of minimal elements of a set T of finite traces. The difference between these orderings is that ≤ orders just approximation, but not non-determinism, i.e.:

$$Bot \le a \to Bot \le a \to a \to Bot \ldots$$

but:

$$a \to Bot \not\le a \to Bot \sqcap b \to Bot \not\le a \to Bot \sqcap b \to Bot \sqcap c \to Bot \not\le \ldots$$

Note that the chain outlined above is ordered w.r.t. ⊑, however.

The well-known theorem:

$$P \leq Q \Rightarrow P \sqsubseteq Q \qquad \text{(ord\_imp\_ref)}$$

expresses that the process ordering is just a coarser ordering than the refinement ordering.

The definition of $\leq$ proves to be an instance of po. With Bot identified with $\bot$, the type α process is proven to form an instance of the type class cpo. As a consequence we inherit all definitions and theorems from the generic fixpoint theory. The CSP-operator μ is just identified with fix:(process $\Sigma \rightarrow$ process $\Sigma) \rightarrow$ process Σ.

A quite important consequence of ord_imp_ref is that the fixpoints (which are known to uniquely exist in the generic fixpoint theory) have a very particular form in the process-instance:

$$\text{fix f} = \text{Abs\_process } (\bigcap_{i \in \mathbb{N}} \mathcal{F}(f^i \text{ Bot}), \bigcap_{i \in \mathbb{N}} \mathcal{D}(f^i \text{ Bot})) \qquad \text{(fix\_eq\_lim\_proc)}$$

i.e. if a fixpoint exists w.r.t. $\leq$, than it coincides with the fixpoint w.r.t. $\sqsubseteq$.

The most complex part of the entire theory is the proof of continuity for the CSP-operators. The required properties have the following form:

cont F $\Rightarrow$ cont (λ x. a $\rightarrow$ F x)	(cont_prefix)
cont F $\wedge$ cont G $\Rightarrow$ cont (λ x. F x $\Box$ G x)	(cont_ndet)
cont F $\wedge$ cont G $\Rightarrow$ cont (λ x. F x $\sqcap$ G x)	(cont_det)
cont F $\wedge$ cont G $\Rightarrow$ cont (λ x. F x ; G x)	(cont_seq)
cont F $\wedge$ cont G $\Rightarrow$ cont (λ x. F x [|A |] G x)	(cont_parint)
cont F $\wedge$ finite A $\Rightarrow$ cont (λ x. F x \ A)	(cont_hide)
. . .	

Especially the last two theorems can pass as "highly non-trivial" even by mathematically rigorous standards; as formal proofs, they must be considered as hard. Phrases like "By Königs lemma follows the existence of finitely many traces of the form ... " required weeks of intensive work.

The collection of the above theorems (together with cont_id and cont_const_fun) is used to instantiate Isabelle's simp_tac procedure (see [Pau 94]), that applies them in a backward-chaining technique similarly to PROLOG-interpreters. This yields a tactical program that decides the continuity of arbitrary CSP-expressions with finite hide-sets as required for the application of the Knaster-Tarski theorem or for the fixpoint induction.

4.6 Laws

From the definitions of the CSP-operators the usual CSP-laws can be derived as formally proven theorems. Among them there is also the list drawn from [BR 85]:

$P \Box P = P$ | $P \Box Q = Q \Box P$

$P \Box (Q \Box R) = (P \Box Q) \Box R$ | $P \Box (Q \sqcap R) = (P \Box Q) \sqcap (P \Box R)$

$P \sqcap (Q \Box R) = (P \sqcap Q) \Box (P \sqcap R)$ | $P \Box Stop = P$

$a \rightarrow (P \sqcap Q) = a \rightarrow P \sqcap a \rightarrow Q$ | $a \rightarrow P \Box a \rightarrow Q = a \rightarrow P \sqcap a \rightarrow Q$

$P \sqcap P = P$ | $P \sqcap Q = Q \sqcap P$

$P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$

$P \parallel Q = Q \parallel P$ | $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$

$P \parallel (Q \sqcap R) = P \parallel Q \sqcap P \parallel R$

$a \rightarrow P \parallel b \rightarrow Q = Stop$ if $a \neq b$ | $a \rightarrow P \parallel b \rightarrow Q = a \rightarrow (P \parallel Q)$ if $a = b$

$P \parallel Stop = Stop$

$P \,|||\, Q = Q \,|||\, P$ | $P \,|||\, (Q \,|||\, R) = (P \,|||\, Q) \,|||\, R$

$P \,|||\, (Q \sqcap R) = P \,|||\, Q \sqcap P \,|||\, R$

$a \rightarrow P \,|||\, b \rightarrow Q = a \rightarrow (P \,|||\, b \rightarrow Q) \Box b \rightarrow (a \rightarrow P \,|||\, Q)$

$Skip ; P = P$ | $Stop ; P = Stop$

$(a \rightarrow P); Q = a \rightarrow (P ; Q)$ | $P ; (Q ; R) = (P ; Q) ; R$

$P ; (Q \sqcap R) = (P ; Q) \sqcap (P ; R)$ | $(Q \sqcap R) ; P = (Q ; P) \sqcap (R ; P)$

$P \setminus \{a\} \setminus \{a\} = P \setminus \{a\}$ | $P \setminus \{a\} \setminus \{b\} = P \setminus \{b\} \setminus \{a\}$

$(a \rightarrow P) \setminus \{b\} = a \rightarrow P \setminus \{b\}$ if $a \neq b$ | $(a \rightarrow P) \setminus \{a\} = (P \setminus \{a\})$

$(Q \sqcap R) \setminus A = Q \setminus A \sqcap R \setminus A$

Note that the law $P \,|||\, Stop = P$ (as in [BR 85]) does not hold as a consequence of its definition based on the parallel interleave operator. Instead, we have:

$$P|||Stop=P;Stop.$$

5 Proof Support for CSP

Fixpoint theory comes with a general induction principle called fixpoint induction. We will see that it can be expressed particularly elegant in HOL. Moreover, it will be shown that fixpoint induction can be used as proof principle for refinement proofs.

5.1 Fixpoint Induction

The idea of this proof principle is to induce a property *P* over ascending chains in directed sets. If *P* is *admissible*, i.e. if validity of *P* for all elements of a directed set *Y* always implies validity of *P* for the least upper bound of *Y*, then the task of proving a property *P* for a fixpoint fix *f* reduces to prove *P* for all its approximations.

Admissibility is a second order concept and can not be represented inside a first-order logic. In the days of the late Edinburgh LCF-prover, the task was resolved by built-in syntactical checks over predicates, the principles of which had been worked out by meta-theoretic reasoning. These checks were a constant source of errors and annoyance since they inherently conflicted with the overall design goal to keep the core of a theorem-prover small and simple.

In HOL admissibility adm: (α::cpo → bool) → bool is just an ordinary predicate (to our knowledge, the idea of an object-logical representation of admissibility is due to [Reg 94]) defined by:

$$adm\ P \equiv \forall Y.\ directed\ Y \Rightarrow (\forall x : Y.\ P\ x) \Rightarrow P(lub\ Y) \qquad (adm\_def)$$

which leads naturally to a list of theorems that implement the reminiscent syntactic checks in ordinary derived proof-rules *inside* the logic:

$$adm\ (\lambda x.c) \qquad (adm\_const\_fun)$$
$$adm\ P \wedge adm\ Q \Rightarrow adm\ (\lambda x.\ P\ x \wedge Q\ x) \qquad (adm\_conj)$$
$$adm\ P \wedge adm\ Q \Rightarrow adm\ (\lambda x.\ P\ x \vee Q\ x) \qquad (adm\_disj)$$
$$cont\ f \wedge cont\ g \Rightarrow adm\ (\lambda x.\ f\ x \leq g\ x) \qquad (adm\_ord)$$

etc.

Admissibility is used in the fixpoint induction principle in the following way:

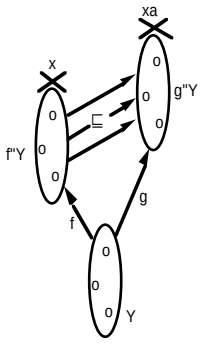
$$|[\ cont\ f \wedge adm\ P \wedge (\forall\ x.\ P\ x \Rightarrow P(f\ x))\ |] \Rightarrow P\ (fix\ f) \qquad (fix\_ind)$$

The crucial question arises, if the refinement ordering is also admissible. This is vital for the applicability of fixpoint induction for the highly desirable refinement proofs. To our knowledge, this question has not been risen so far in the literature.

Of course, such a property cannot be proven in the generic fixpoint theory (as all theorems above) but only in the process instance.

Proposition: The refinement ordering is admissible, i.e.

$$cont\ f \wedge cont\ g \Rightarrow adm\ (\lambda x.\ f\ x \sqsubseteq g\ x) \qquad (adm\_ref\_ord)$$



Proof-Sketch: Let f and g be continuous, Y be directed and let (∀x:Y. f x ⊑ g x) hold. Let f''Y and g''Y denote the image sets of Y w.r.t. f and g. Then the figure aside gives an overview over the situation.

Here *x* and *xa* denote the lub's w.r.t. ≤. As a consequence of ord_imp_ref and of transitivity of ⊑, both *x* and *xa* must be upper bounds w.r.t. ⊑ for f''Y. The question arises if they are also related via ⊑. The answer is positive as a consequence of fix_eq_lim_proc and the definition of ⊑, i.e. *x* is also *least* upper bound w.r.t. ⊑.

This fact gives us that living with two orders in CSP (as a price for unbounded nondeterminism) is perhaps inelegant and uncomfortable, but perfectly possible.

5.2 Take Lemmas

Fixpoint induction proofs are usually quite ingenious proofs. In this section we will discuss a more specialised proof-scheme that is more amenable to automated reasoning. This principle will also shed some light on the potential of model-checking techniques (seen from the perspective of symbolic reasoning).

The principle of take lemmas is enclosed in the take operator _↓_ : process Σ → nat → process Σ, that cuts a behaviour of a process up to a depth n, for example:

$$fix\ (\lambda x.\ a \rightarrow x) \downarrow 1 = a \rightarrow Bot.$$

The definition of this operator along the usual lines yields the characterising theorems:

$\mathcal{F}(P \downarrow n) = \mathcal{F}\,P \cup \{ (s, X) \mid s \in \mathcal{D}(P \downarrow n)\}$

$\mathcal{D}(P \downarrow n) = \mathcal{D}\,P \cup \{ s \hat{} t \mid |s|=n \wedge \text{tick-free } s \wedge \text{front-tick-free}(t) \wedge s \in \text{traces } P\}$

From there the following *cutting-rules* are derived:

$P \downarrow 0 = Bot$ $\qquad$ $(a \rightarrow P) \downarrow n = a \rightarrow (P \downarrow n-1)$

$(Q \sqcap R) \downarrow n = (Q \downarrow n \sqcap R \downarrow n)$

…

The principal characteristic of this operator is that it is monotone w.r.t. $\leq$:

$n \leq m \Rightarrow P{\downarrow}n \leq P{\downarrow}m$

This fact allows us to specialise the fixpoint-induction to the $\leq$-take-lemma:

$\forall m\ ((\forall n.\ n < m \ \wedge\ P{\downarrow}n \sqsubseteq Q{\downarrow}n) \Rightarrow P{\downarrow}m \sqsubseteq Q{\downarrow}m) \ \Rightarrow\ P \sqsubseteq Q$

Note the strong similarity of this rule to Noetherian induction. Using this take-lemma, we can perform the following backward-proof example:

$\quad fix(\lambda x.\ a \rightarrow x) \sqsubseteq fix(\lambda x.\ a \rightarrow x \sqcap \lambda x.\ b \rightarrow x)$

$\Leftarrow$ { by $\leq$-take-lemma,$\forall$-intro, $\Rightarrow$-intro}

$\quad |[\forall n.\ n < m \ \wedge\ fix(\lambda x.\ a \rightarrow x){\downarrow}n \sqsubseteq fix(\lambda x.\ a \rightarrow x \sqcap \lambda x.\ b \rightarrow x){\downarrow}n]| \Rightarrow$
$\quad\quad fix(\lambda x.\ a \rightarrow x){\downarrow}m \sqsubseteq fix(\lambda x.\ a \rightarrow x \sqcap \lambda x.\ b \rightarrow x){\downarrow}m$

$\Leftarrow$ { by knaster-tarski}

$\quad |[...]| \Rightarrow (a \rightarrow fix(...)){\downarrow}m \sqsubseteq (a \rightarrow fix(...) \sqcap b \rightarrow fix(...)){\downarrow}m$

$\Leftarrow$ { by cutting rules}

$\quad |[...]| \Rightarrow a \rightarrow (fix(...){\downarrow}m\text{-}1) \sqsubseteq a \rightarrow (fix(...){\downarrow}m\text{-}1) \sqcap b \rightarrow (fix(...){\downarrow}m\text{-}1)$

$\Leftarrow$ { by refinement projection left}

$\quad |[...]| \Rightarrow a \rightarrow (fix(...){\downarrow}m\text{-}1) \sqsubseteq a \rightarrow (fix(...){\downarrow}m\text{-}1)$

$\Leftarrow$ { by refinement monotonicity}

$\quad |[\forall n.\ n < m \ \wedge\ fix(...){\downarrow}n \sqsubseteq fix(...){\downarrow}n]| \Rightarrow fix(...){\downarrow}m\text{-}1 \sqsubseteq fix(...){\downarrow}m\text{-}1$

$\Leftarrow$ { by arithmetic and assumption}

$\quad$ True

Even without knowing anything about tactical programming in Isabelle, it is not hard to see how this proof-technique can be mechanised. The essential difficulties are to unfold fix-terms only in a controlled way, to "drive inside" the take-operator occurrences while decreasing their offsets and to control the necessary backtracking for refinement projection left rsp. refinement projection right.

The technique resembles very much the usual graph-exploration techniques in labelled transition diagrams (as implemented in FDR). The nodes in the graph correspond to equivalence-classes on take-terms, the edges applications of the refinement monotonicity. If problematic pathological cases were avoided (so-called *non-contracting* bodies of fix like $fix(\lambda x.x)$), and if graph-regularity can be assured, this tactical program will be a (proven correct) decision procedure.

6 Example

The following example is drawn from [For 95], pp. 5. It specifies a process *COPY* that behaves like a one place buffer. Then an implementation using a separate sender *SEND* and receiver processes *REC*, communicating via a channel *mid* and an acknowledgement *ack*. Instead of using model-checking for a known, finite alphabet of events, we will prove via fixpoint induction for arbitrary alphabets that the implementation refines the specification. Note, however, that the alphabets must still

be finite because of the hiding operator in *SYSTEM*, which is known to be noncontinuous for infinite alphabets (see [BR 86]).

On the top-level of our CSP theory in Isabelle, new syntax for channels has been introduced. Hence *writing* c!a→P is represented by (c,a)→P and *receiving* c?x→P x is mapped to an appropriate representation with multi-prefixes.

Our can be represented in an Isabelle theory by introducing a data type for all involved channels. This can be done in an ML-like definition:

```
datatype channel = left | right | mid | ack
```

The process COPY : process (channel × Σ) is defined as follows:

```
COPY ≡ (letrec COPY = left?x → right!x → COPY in COPY     (COPY_def)
```

The definition of the implementation reads as follows:

```
SYSTEM ≡ (   letrec    SEND = left?x → mid!x → ack?y → SEND;
                       REC  = mid?x → right!x → ack!x → REC
             in        SEND [| SYN |] REC) \ SYN)     (SYSTEM_def)
```

where SYN ≡ {x | fst x = mid ∨ fst x = ack}.

Now we can state the desired proof-goal COPY ⊑ SYSTEM (under premise P : finite SYN) with COPY acting as specification of the behaviour of SYSTEM.

In the following presentation of the backward-proof, we suppress the required proofs of continuity (which were eliminated by an appropriate tactic). For convenience, we introduce G as abbreviation for the often re-occurring term:

```
(λu.  (left?x→mid!x→ack?y→fst u, mid?x→right!x→ack!x→snd u))
```

Then, the main steps of the refinement proof are:

```
COPY ⊑ SYSTEM
⇐   {by COPY_def, SYSTEM_def,  fix_ind, adm_ref_ord }
1)  Fixpoint induction base:
        Bot  ⊑ SYSTEM
    ⇐   {by Bot ⊑  X }
        True
2)  Fixpoint induction step:
        |[ x ⊑       (fst (fix G) [| SYN |] snd (fix G)) \ SYN]| ⇒
            left?xa  → right!xa  → x
        ⊑
            (fst (fix G) [| SYN |] snd (fix G)) \ SYN
    ⇐   {by knaster_tarski over both fix-terms, fst-snd-simplification}
        |[...]| ⇒
            left ? xa → right ! xa → x
        ⊑
            (left?x → mid!x → ack?y → fst (fix  G)
            [| SYN |]
             mid?x → right!x → ack!x → snd (fix G)) \ SYN
```

$\Leftarrow$ {by distributive laws of the hiding operator, the parallel interleave operator and the Mprefix operator}

$|[\ldots]| \Rightarrow$

left?xa $\rightarrow$ right!xa $\rightarrow$ x

$\sqsubseteq$

left?x $\rightarrow$ right!x$\rightarrow$((fst(fix G)|]SYN[|snd(fix G)\SYN)

$\Leftarrow$ {by monotonicity of multiprefix operator w.r.t refinement order $\sqsubseteq$ and by assumption}

True

The premise P was only used in the proof of admissibility, when applying adm_ref_ord. A careful analysis of its proof reveals that it can be strengthened to cont f $\wedge$ mono g $\Rightarrow$ adm (λx. f x $\sqsubseteq$ g x), while on the other hand a proof of monotonicity for the hide operator with arbitrary sets seems feasible. This seems to suggest that at least the class of typical refinements fix f $\sqsubseteq$ (fix g)\A (provided that f and g continuous) with one outermost hiding operator hiding away an arbitrary internal communication channel introduced by the refinement step can be handled also in the infinite case.

7 Conclusion

We have presented a corrected, shallow embedding of CSP into higher-order logic that nevertheless preserves the algebraic properties of CSP for which we have formal, machine-checked proofs. This embedding forms an implementation of a "CSP Workbench" that allows interactive theorem proving in CSP-specifications with infinite alphabet (complementary to the FDR-tool that allows automatic proofs on specialised, finite CSP-specifications). The collection of theories has been converted directly by Isabelle into a "textbook on CSP theory" available under "http://www.informatik.uni-bremen.de/~bu/isa_doc/CSP/doc/html/index.html".

Some remarks should be given on the amount of verification work. The theory presented so far required one man year (excluding a first attempt of five man months invested in a model much closer to [Hoa 85] that turned out to be infeasible). This effort could probably have been reduced by better expert advice, since our major problems came from wrong theoretic foundations, gaps in proofs etc. and not from the technicalities of "embedding" or proving. Although the effort still may be qualified as considerable, we see a need for more machine assisted verification work, since there is a tendency to dilute the formal core of a research programme, especially a successful one. In the meantime there are so many different variants of CSP, that they are very likely to be incompatible. Due to the high publication pressure, authors tend to modify the definitions according to their needs and cite the proofs from elsewhere ("proof is done analogously to [XY ??]"). In such a situation, research peers can shift more research effort to *canonical* theory representations that were verified by machine assistance.

We are not denying that formal proof activity without mathematical intuition is blind, but we would like to emphasise that intuition tends to delude more often in foundational theories of computer science that in other mathematical research areas, perhaps due to their discrete nature and resulting combinatorial complexity. The treatment of tick is an example for a unintuitive, combinatorically complex part of a complex theory. Obviously, the situation gets even worse if combinations of formal

methods — as envisaged by the UniForM project [Kri+95] — are undertaken. Nevertheless, such combination-methods are particularly desirable since "there is no single theory for all stages of the development of software [...]. Ideas, concepts, methods and calculations will have to be drawn from a wide range of theories, and they are going to have to work together consistently [...]" (again from Hoare's invited lecture at FME'96).

7.1 Future Work

We will investigate to prove the denotational semantics as described in this paper consistent with the operational semantics of FDR [For 95], i.e. we prove consistency with the formal specification of this tool (we are not planning to "prove FDR" w.r.t. this specification). As a result, one can embed the FDR-tool as a proof-oracle (external decision procedure) within Isabelle in order to build up a logically consistent, combined environment for the reasoning over CSP. This is particularly attractive, since both tools deliver complementary deduction support: Isabelle/CSP provides interactive proof support for infinite CSP, while FDR excels at automatic refinement proofs for specialised, finite CSP specifications. In such an environment, general requirements-engineering is possible, followed by a sequence of "massage steps" that make a specification amenable for FDR, concluded by combined proof-efforts of FDR and Isabelle/CSP.

We are interested in designing a transformational methodology in CSP. This means that a collection of "transformation rules" in the sense of [KSW 96a] should be designed that allow the construction of a CSP-process by identifying and refining *design-patterns*.

We work at a safe and semantically clean integration of CSP with other industry-standard specification languages like Z (whose representation in Isabelle/HOL has been worked out in [KSW 96b]). First conceptual studies for such an integration are [Fis 97].

Finally we admit that an encapsulation of the Isabelle/CSP embedding in an integrated *tool* is of crucial importance for further acceptance in industry. Following the lines of [KSW 96a], a generic user interface has been developed that can be instantiated with LCF-style theorem-prover in order to encapsulate them as a specialised tool (see [KLMW 96]). An instance of this technology with Isabelle/CSP has been envisaged. Moreover, an even wider goal of UniForM is to provide a workbench to integrate these tools and to provide them with inter-tool communication, version-management and development-management. We believe that this technology should ease the construction of powerful formal methods tools and simplify the technical side of interchanging information between them.

Acknowledgement. We would like to thank A.W.Roscoe for several hints helping us to bridge big steps in rigorous mathematical proofs. Prof. Bernd Krieg-Brückner, Thomas Santen, Sabine Dick, Christoph Lüth and Clemens Fischer read earlier versions of this paper.

References

[And 86] P.B. Andrews: *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*, Academic Press, 1986.

[BH 95] J. P. Bowen,M. J. Hinchey: Seven more Myths of Formal Methods: Dispelling Industrial Prejudices, in *FME'94: Industrial Benefit of Formal Methods*, proc. 2nd Int. Symposium of Formal Methods Europe, LNCS 873, Springer Verlag 1994, pp. 105-117.

[BR 85] S.D. Brookes, A.W. Roscoe: An improved failures model for communicating processes. In: S.D.Brookes (ed.): Seminar on Semantics of Concurrency. LNCS 197, Springer Verlag, pp. 281-305. 1985.

[Cam 91] A.J. Camillieri: A Higher Order Logic Mechanization of the CSP Failure-Divergence Semantics. G. Birtwistle (ed): *IVth Higher Order Workshop*, Banff 1990. Workshops in Computing, Springer Verlag, 1991.

[Chu 40] A. Church: A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5, 1940, pp. 56-68.

[Fis 97] C. Fischer: Combining CSP and Z. Submitted for publication.

[For 95] Formal Systems (Europe) Ltd: Failures-Divergence Refinement: FDR2, Dec.1995. Preliminary Manual.

[GM 93] M.J.C. Gordon,T.M. Melham: *Introduction to HOL: a Theorem Proving Environment for Higher order Logics*, Cambridge Univ. Press, 1993.

[Hoa 85] C.A.R.Hoare: Communication Sequential Processes.Prentice-Hall, 1985

[KLMW96] Kolyang, C. Lüth, T. Meier, B. Wolff: Generic Interfaces for Formal Development Support Tools. In: Workshop for Verification and Validation Tools, Bremen. to appear in LNCS.

[Kri+95] B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer, A. Baer, : Uniform Workbench — Universelle Entwicklungsumgebung für formale Methoden. Technischer Bericht 8/95, Universität Bremen, 1995. See also the project home-page: http://www.informatik.uni-bremen.de/~uniform.

[KSW 96a] Kolyang, T. Santen, B. Wolff: Correct and User-Friendly Implementations of Transformation Systems. Proc. Formal Methods Europe, Oxford. LNCS 1051, Springer Verlag, 1996.

[KSW 96b] Kolyang, T. Santen, B. Wolff: A structure preserving encoding of Z in Isabelle/HOL. In J. von Wright, J. Grundy and J. Harrison (eds): Theorem Proving in Higher/Order Logics — 9th International Conference, LNCS 1125, pp. 283-298, 1996.

[Pau 94] L. C. Paulson: *Isabelle - A Generic Theorem Prover*. LNCS 828, 1994.

[RB 89] A.W. Roscoe, G. Barett: Unbounded Nondeterminism in CSP. In: M. Main, A.Melton,M.Mislove,D.Schmidt (eds): 9th International Conference in Mathematical Foundations of Programming Semantics. LNCS 442,pp. 160-193, 1989.

[Reg 94] F. Regensburger: HOLCF: Eine konservative Einbettung von LCF in HOL. Phd thesis, Technische Universität München. 1994.

[Ros 88] A.W. Roscoe: An alternative Order for the Failures Model. In: Two Papers on CSP. Technical Monograph PRG-67, Oxford university Computer Laboratory, Programming Research Group, July 1988.

[Ros 96] A.W. Roscoe, e-mail communication with the authors.