# CSP Revisited

Florian Kammüller

Technische Universität Berlin
Institut für Softwaretechnik und Theoretische Informatik

**Abstract.** In this paper we revisit the formalization of Communicating Sequential Processes (CSP) [2] in Isabelle/HOL. We offer a simple alternative embedding of this specification language for distributed processes that makes use of as many standard features of the underlying Higher Order Logic of Isabelle, like datatypes and the formalization of fixpoints due to Tarski.

## 1 Introduction

The specification language CSP for Communicating Sequential Processes is a classical tool for specifying and reasoning about parallel communicating processes.

Although there exists the FDR tool for the analysis of CSP specifications there has always been a considerable amount of parallel activities to mechanize the CSP language in HOL systems. The first formalization of this theory in Isabelle/HOL [4] dates back to the early nineties [1]. Later the work by Tej and Wolff contributed a fully fledged support tool. Yet there has been ongoing research on formalizing CSP in Isabelle/HOL [3].

The reason to offer yet another formalization is twofold. Firstly, the earlier formalization by Wolff has been frozen in with Isabelle version 98 because the formalization was quite closely intertwined with the Isabelle source code. Thus it became more and more difficult for the developers to adjust their CSP tool repeatedly to newer Isabelle versions. We believe that it must have been for this particular reason that more recently a new project was initiated where the authors reformalized the CSP calculus. However, contemplating this more recent formalization we found that it uses sometimes contrived implementations for the definition of recursive processes relying implicitly on Tarski's fixpoint theory. However, they do not use this theory although it is available in Isabelle/HOL (theory Fixedpoint.thy). There might be pragmatic reasons not to build on it but we find that this unnecessarily endangers consistency. As we also use CSP in teaching, we have a particular interest to show the concepts of CSP in a transparent light while offering students a tool environment.

For these reasons we construct CSP again in Isabelle/HOL while keeping the formalization as lightweight and practical as possible. Therefore we aimed at using where possible the powerful datatype package in combination with primitice recursive definitions of the operators and reusing as much as psooible existing predefined theories like the fixpoint theory provided in Isabelle/HOL.

In this paper we present first the definition of the CSP syntax using a simple datatype definition in Section 2. Then we define the semantics of CSP in Section 3. In this section we also briefly summarize the two extensions of the simple trace semantics to failures and divergences. For the semantics of recursive processes we use the existing theory of fixpoints in Isabelle/HOL. In Section 4 we consider the notion of process refinement that is central to CSP for stepwise development of specifications. Finally, in Section 5 we compare our approach in particular to the earlier formalization by Wolff and draw some general conclusions.

## 2 The Syntax of CSP Processes

The first most decisive design decision in any CSP specification is the definition of the possible communication events $\Sigma$. A process in CSP is then completely defined by the events it can communicate. There is one basic process called `STOP` that does not communicate anything. It represents the deadlock process. Other processes may be constructed from existing processes and events. To build a process that communicates sequentially the *prefixing* operator $\rightarrow$ is used. Given an event $a \in \Sigma$ and a process $P$ the constructed process $a \rightarrow P$ communicates first the event $a$ and after that it behaves like process $P$. Given two processes $P$ and $Q$ we can construct the deterministic choice $P \Box Q$ and the nondeterministic choice $P \sqcap Q$ between the two processes. Parallel composition of processes is also possible. There are four possibilities for parallel composition. The interleave operator combines two processes $P$ and $Q$ in such a way that the resulting process $P|||Q$ enables all possible interleavings of the events communicated by either of the two. By contrast, in the parallel composition $P \parallel Q$ the resulting process communicate only events that can be communicated by either of the two processes simultaneously. That is in the interleaving there is no *synchronization* between the communciation of $P$ and $Q$. For a more precise control over independence of communication on the one side and synchronization on the other side there are the operators generalized and alphabetized parallel (here `genpar` and `alphpar`). In the generalized parallel operator the set $A$ of events over which the processes $P$ and $Q$ must agree is parameterized. That is in `genpar` $P$ $A$ $Q$ the processes $P$ and $Q$ must synchronize over all events in $A$ but can communicate independently all other events outside $A$. The hiding operator finally enables to hide selected events from the visible part of a communication.

The following datatype definition defines the syntax of CSP processes.

```
datatype α process =
    STOP
  | prefix α (α process)                (infixl "→" 50)
  | dchoice (α process) (α process)     (infixl "□" 50)
  | nchoice (α process) (α process)     (infixl "⊓" 50)
  | parallel (α process) (α process) (infixl "‖" 50)
  | interleave (α process)(α process) (infixl "|||" 50)
  | genpar   (α process) (α set) (α process)
  | hiding   (α process) (α set)    (infixl "\" 50)
```

# 3 Semantics

The first semantical model of CSP is the trace model. There are also the failures and the failures and divergences model that are refined models for CSP processes, but all build on the traces model.

## 3.1 Trace Model

For an alphabet $\Sigma$ the set $\Sigma^*$ denotes the set of all finite concatenations of elements of $\Sigma$ including the empty element. A word in $\Sigma^*$ is a concatenation $\langle a_1, a_2, \ldots \rangle$ for elements $a_1, a_2, \ldots \in \Sigma$. The most natural presentation of traces in Isabelle/HOL is given by the list datatype, a well supported theory of Isabelle/HOL.

```
types α trace = α list
```

In order to define the first simple trace semantics of processes we use that classical filter operator on traces annotated as $\restriction$.

```
consts
  filter ::   [α trace, α set] ⇒ α trace (infixl "↾" 50)
```

Given a trace $s$ and a set of events $A$ it returns the trace that is obtained from $s$ by deleting all events that are not in $A$. This semantics can be given by the following two primitive recursive definitions. Note, that we reuse here the empty list `[]` to represent the empty trace: the above `types` definition of traces keeps the underlying `list` syntax available for traces. Similarly, we reuse `#`, the list constructor, as concatenation for traces.[1]

```
primrec
  filter_nil : "[] ↾ A = []"
  filter_cons: "(a # l) ↾ A = (if a: A then a # (l ↾ A) else (l ↾ A))"
```

Next we declare the following constant tor the trace semantics of CSP processes.

```
  traces :: α process ⇒ (α trace) set  ("⟦ _ ⟧" [21] 20)
```

The idea of this first coarse semantics of CSP is to assign to each process the set of traces that record the possible behaviour of this process at each moment in time. That is, the empty trace is always an element of the traces of any process as it represents the moment when nothing yet has been communicated. It is also implied that the traces of any process are *prefixed-closed*: if a trace $t$ is in the set of `traces`$(P)$, then all prefixes of $t$ must be contained in `traces`$(P)$.

The traces of a process are now defined by underlying each of the possible constructors with a semantics. It is worth noting at this point that we can formulate these semantical rules again by primitive recursion. In the following we explain the pieces of a whole `primrec` section line by line at the same time

[1] We may easily introduce syntactic sugar to cover the original list notions with the CSP specific synax.

sheding some light on the trace semantics. The STOP process' semantics is that it does never communicate anything: yet only the empty trace describes this behaviour.

```
primrec
  traces_STOP : "⟦ STOP ⟧ = { [] }"
```

The traces for prefixing a process $P$ by $a$ are built by juxtaposing the $a$ in front of all traces of $P$; the empty trace has to be inserted afterwards again as it is lost in this procedure.

```
  traces_prefix:
  "⟦ a → P ⟧ = { [] } ∪ { t | ∃ t'. t = a # t' ∧ t' ∈ ⟦ P ⟧ }"
```

The deterministic choice between two processes $P$ and $Q$ opens up two branches in the behaviour of the process: hence the traces of the two individual processes are unified.

```
  traces_dchoice: "⟦ P □ Q ⟧ = ⟦ P ⟧ ∪ ⟦ Q ⟧"
```

The nondeterministic choice is meant to be a choice which is not influenced by the environment, but is internally taken (by the system). However, considering the traces, which build nothing more than a *transcript* of possible behaviours, the way thechoice has been taken at a certain point is lost: the traces for the nondeterministic choice are identical to the traces of the determninsitic choce.

```
  traces_nchoice: "⟦ P ⊓ Q ⟧ = ⟦ P ⟧ ∪ ⟦ Q ⟧"
```

This shortcoming of the trace semantics leads to the extension of the semantical model by failures (see below).

The four different forms of parallel operators are defined as follows. In general the processes have to agree on each event they communicate: their traces must be shared, i.e. the traces of the resulting parallel composition is the intersection of the traces of each individual process.

```
  traces_parallel : "⟦ P ‖ Q ⟧ = ⟦ P ⟧ ∩ ⟦ Q ⟧"
```

In the other extreme, the interleaving of two processes that are not synchronized at all, each of the combined processes has its own contribution to a trace of the combination. Technically we collect those combined traces $s$ as restrictions to individual sets of events traces for each of the constituent processes such that these individual sets unify to the set of all traces that occur in $s$.

```
  traces_inter : "⟦ P ||| Q ⟧ =
  { s | ∃ X Y . s ↾ X ∈ ⟦ P ⟧ ∧ s ↾ Y ∈ ⟦ Q ⟧ ∧ set s = X ∪ Y }"
```

The intermediate solution between the two – some portion $A$ of events needs to be synchronized – is expressed in a similar manner. The restriction set of events of each of the processes $P$ and $Q$ includes here the set $A$ thus enforcing that the events of $A$ are communicated by both.

```
traces_genpar : "⟦ genpar P A Q ⟧ =
{ s . ∃ X Y. s ↾ (A ∪ X) ∈ ⟦ P ⟧ ∧ s ↾ (A ∪ Y) ∈ ⟦ Q ⟧
          ∧ set s = X ∪ A ∪ Y }"
```

Hiding a set of events $A$ in a process $P$ simply means that in all traces of the original process $P$ each event that is in $A$ has to be eliminated. This can be encoded in the language of Isabelle/HOL by mapping a function that filters the complement of $A$ with respect to $\Sigma$ out of any trace $s$ over the set of all traces of $P$. Here - is the symmetric set difference and ‘ the operator building the image of a function applied to a set.

```
traces_hide: "⟦ P \ A ⟧ = (λ s. s ↾ (Σ - A))‘(⟦ P ⟧)"
```

### 3.2 Recursion

To construct non-trivial processes the use of recursion is enabled. Users of CSP may casually write recursive eqation for the definition of processes, like the following process communicating sequences of $a$'s of arbitrary length.

```
P = a → P
```

Semantically such a recursive equation is resolved in the classical way as the the fixpoint of a corresponding functorial. For the example, the fixpoint of the functorial $\lambda s.a{\rightarrow}s$ is chosen. Now, as we have defined the semantics of our CSP processes as sets (of traces) we have to consider functorials over such sets. As powersets with the subset relation form complete partially ordered sets we can use the theory of fixpoints in Isabelle/HOL to assign a meaning to recursive functions.

Now an interesting problem arises. As we base our embedding on a syntactical characterization of CSP operators by a datatype a recursive rescription is also syntactical. However, the fixpoints are only defined in the semantics, because it is there that we arrive at a function representing the meaning (syntactically the recursion leads to ever longer nested calls). In order to achieve this we define the set of all trace sets that are ever reached by any process over an alphabet $\Sigma$.

```
constdefs
  Traces :: ((α trace) set) set
  "Traces == {A :: ('a trace) set. ∃ P. traces P = A}"
```

We define a higher order function that transforms a functorial over syntactic processes over a functiorial over traces, i.e. in the semantics. We need to reconstruct the syntactical process that has a certain semantics in order to define the latter semantical factorial. To this end, we use the Hilbert operator, which enables the selection of some syntactical process that is mapped to a given trace set $y$.

```
recursion_prep :: (α process ⇒ α process) ⇒ ((α trace)set ⇒ (α trace)set)
"recursion_prep f == (λ y :: (α trace)set . if y ∈ Traces then
                    ⟦ f (SOME z. traces z = y)⟧ else {})"
```

To reassure ourselves that this transformation function works we prove the following lemma.

```
y ∈ Traces ⟹ ∃ z. traces z = y
```

CSP uses the notation $\mu$ for the least fixed point operator. Given the transformation function `recursion_prep` we can use the predefined least fixed point operator to assign a semantics to recursive process definitions.

```
constdefs
  μ :: "(α process ⇒ α process) ⇒ (α trace) set"
  "μ f == lfp (recursion_prep f)"
```

### 3.3 Divergences

Through hiding in combination with recursion arises another problem in the behavioural specification of processes. Consider the following recursive process.

```
P = (a → P) \ a
```

Process `P` does not communicate anything visible, as the only event $a$ that it communicates is hidden. This process behaves like the process `STOP` when considering just its traces.[2] However, in difference to `STOP` which does not communicate at all `P` has an endless invisible activity of communcation going on.

The behaviour expressed by process `P` is considered as a *divergence* in CSP. A process that may behave at a certain point like the above process is considered as nonterminating. To differentiate this behaviour from a process like `STOP` that does not do anything, divergences are explicitly introduced as the third layer of semantics in CSP.

### 3.4 Failures and Divergences

As we have seen in the previous sections there are two phenomena not expressible in the simple trace model of CSP.

- deterministic and nondeterministic choice are not distinguishable
- deadlock and livelock are not distinguishable.

For the former reason CSP considers a second extension of its semantical model by sets of events that may be refused at each moment during a process. These so-called *refusal sets* combined with a trace representing the particular moment when this refusal is possible form so-called *failures*. The latter reason gives rise to the extension of the semantical model by a third component: the *divergences*. These are sets of traces after which a process can behave like the above in that it produces an infinite invisible activity.

[2] and also failures as we will see later

Formally, we define first divergences because failures may be defined reusing divergences. The divergences of a process are a set of traces that record the traces leading up to a divergence point and thereafter admit arbitrary behaviour.[3]

```
consts
  D :: α process ⇒ (α trace) set
```

The divergences of STOP can now exhibit that this process is really a livelock, i.e. does not diverge. The case of the prefixing operator shows the basic idea of divergences: divergences record the behaviour of the preocess up to the point of divergence, thereafter any behaviour is possible in case of divergence. Similarly for the two choice operators alike a possible divergence of one of the combined processes renders the choice divergent. For the parallel operators we give only the more general parallel operator, as the others may be expressed in terms of this.

```
primrec
  Divergences_STOP : "D(STOP) = {}"
  Divergences_prefix: "D (a → P) = { s | ∃ t. s = a # t ∧ t ∈ D(P) }"
  Divergences_dchoice: "D(P □ Q) = (D P) ∪ (D Q)"
  Divergences_nchoice: "D(P ⊓ Q ) = (D P) ∪ (D Q)"
  Divergences_genpar : "D (genpar P A Q) =
                        {s | ∃ X Y. (s ↾ (A ∪ X) ∈ ⟦ P ⟧ ∧ s ∈ D(Q)) ∨
                         (s ↾ (A ∪ Y) ∈ ⟦ Q ⟧ ∧ s ∈ D(P)) }"
  Divergences_hide : "D(P \ A) = {s | ∃ t. t ∈ D(P) ∧ s = t ↾ (Σ - A)}"
```

Failures are pairs of traces and sets of events – the refusals – that a process can refuse to communicate at the point described by the trace.

```
  F :: α process ⇒ (α trace × α set) set
```

The failures of STOP reflect that at the empty trace this process can refuse any set of possible events. The failures of a process $P$ prefixed by event $a$ are characterized as follows: any event other than $a$ amy be refused initially – at the empty trace – afterwards anything that may be refused by $P$. The failures of the nondeterministic choice are now just the union of the failures of each f the combined processes. This definition reflects that an internal choice may refuse a behaviour even if it is for choice. By contrast the deterministic – or external – choice is now differing in the failures model: initially it may only refuse if both processes may refuse, but all further refusals of both processes are then possible. In addition it may refuse anything if it can initially already diverge. For hiding the failures are derived by refering to the failures of the original process adding the hidden events in the refusals and in case of divergence again any failure.

```
primrec
  Failures_STOP :  "F (STOP) = { (s,X) | s = [] ∧ X ⊆ Σ }"
```

[3] In CSP the view is taken that a process that may diverge at some point does not convey any sensible behaviour. Therefore it is assigned any possible behaviour from that point onwards.

```
Failures_prefix: "F (a → P) = { (s,X) | s = [] ∧ a ∉ X } ∪
                  { (s, X) | ∃ s'. s = a # s' ∧ (s',X) ∈ F(P)}"
Failures_dchoice: "F(P □ Q ) = (F P) ∪ (F Q)"
Failures_nchoice: "F(P ⊓ Q) =  {(s,X) | s = [] ∧ (s,X) ∈ (F P)∩(F Q)}
                   ∪  {(s,X) | s ≠ [] ∧  (s,X) ∈ (F P) ∪ (F Q)}
                   ∪  (s,X) | s = [] ∧ [] ∈ D(P) ∪ D(Q) }"
Failures_hide: "F(P \ A) =
                {(s,X) | ∃ t. s = t ↾ (Σ - A) ∧ (t, X ∪ A) ∈ F(P) } ∪
                          {(s,X) | s ∈ D(P \ A)"
```

## 4 Refinement

Refinement of processes is defined on all three levels of the semantical model of CSP. In the earlier papers [1, 5] the authors use the so-called process ordering which is coarser than the classical ordering given by subset relations on the denotations. Tej and Wolff argue that in their mechanization they need this ordering as they need to cope with unbounded nondeterminism. For our principal investigation of the reformalization of CSP using datatype and primitive recursion we stick to the simpler classical refinement ordering.

Process refinement is firstly an important tool for a stepwise development of specifications in CSP: it guarantees that a process that is a refinement behaves inside the behavioural specification of the more abstract process. On the other hand it is also used to abstractly express the behaviour of processes thereby motivating the role of specification versus implementation.

The basic idea is simple. A more abstract process is a more general description of the allowed behaviour. Thus, for the most intuitive, the trace model, it is clear that the semantics of an abstract process contains *more* possible behaviour than its refinement. To put it the other way round: an implementation (or refinement) of a specification may pick out some of the behaviour that is granted as allowed behaviour by its specification. Hence, the refinement $\sqsubseteq_T$ on the trace semantics of processes is simply given by the *subset relation* on the semantical denotations of two processes. The reading of $P \sqsubseteq_T Q$ is however somewhat unnaturally defined as *P is refined by Q* thereby inversing the direction of the $\subseteq$.

```
constdefs
  ⊑_T :: [α process, α process] ⇒ bool (infixl 50)
  "P ⊑_T Q == ⟦ Q ⟧ ⊆ ⟦ P ⟧"
```

For failures, a similar idea applies. Failures contain the traces of the trace model in their first component. Hence it is clear that here the same subsumption relation should hold. With respect to the refusals possible at each point the subsumption does in fact hold in the same sense: if the abstract process may at some point – after a possible trace – refuse certain sets of events, then its implementation may chose to refuse a subset of these possible refusal sets. This concept realizes the idea that the abstract process is also more general in the *must* behaviour. The implementation can only be less chosy. Hence, the failures refinement is also simply given by subsumption of the failures of the two processes.

```
constdefs
  ⊑F :: [α process, α process] ⇒ bool" (infixl 50)
  "P ⊑F Q == F Q ⊆ F P"
```

Finally, for the divergences the reasoning is more intricate. Divergence is seen as indeterminate behaviour. If a process may diverge at a certain point this is interpreted in the CSP world as undefined or unspecified behaviour. Therefore it turns out nicely here that in the divergence model processes have been assigned all possible behaviours up from possible points of divergence. Hence a process that may diverge at a point given by trace $t$ is always more abstract than a process that has identical behaviour up to that point $t$ but does not diverge then, i.e. has just a subset of all combinatorically possible trace continuations. Again the divergence refinement is the subset relation of the divergences subsuming the failures refinement.

```
  ⊑FD :: [α process, α process] ⇒ bool" (infixl 50)
  "P ⊑FD Q == F Q ⊆ F P ∧ D Q ⊆ D P"
```

## 5 Discussion

We first discuss the differences to the earlier formalization [5] of CSP in Isabelle/HOL before we draw some general conclusions.

### 5.1 Comparison

In brief, our formalization works the other way round than the formalization of Tej and Wolff. They use the wellformedness rules of CSP as a big conjunction to define a type of processes in a type definition. Then they define the properties of the operators using the abstraction and representation functions given by the type definition. This is a rather technical process of transfering properties from one domain to another. In our formalization using the datatype definition, we just go the latter step. We use the datatype feature of Isabelle and the list database to build up a wellformed process definition. Then we can use the same definitions they use inside the abstraction functions to define the semantical annotations as functions over the datatype of process. Our approach saves some work by using the now well established features of Isabelle/HOL.

We also use the predefined fixedpoint theory of isabelle/HOL to define the semantics of recursive processes. Although there we have to twist our model in a slightly unnatural way, we arrive at reusing the given fixedpoint constructor because our semantical model is based on sets whereas Tej and Wolff use a self defined type as their semantics. Therefore they have to recreate the entire Tarski fixpoint theory using axiomatic type classes in order to build an instantiation to their process type.

### 5.2 Conclusions

Compared to the very well elaborated theory of CSP given by Tej and Wolff our formalization is just a first experimental sketch to test feasibility of this approach. We believe it is important to keep a formalization lightweight and as simple as possible. The historical development shows that it is well worth trying to keep things simple. Although initially Isabelle/HOL has been intended to be a tool for such deep integrated instantiations as the HOL-CSP tool by Tej and Wolff, it seems nowadays that the applications must not be dependent too much on the inner workings of the implementations. Otherwise, decisive changes in the development of Isabelle/HOL make it impossible to keep such instantiations up to date.

## References


1. A. J. Camillieri. A Higher Order Logic Mechanization of the CSP Failure-Divergence Semantics. *IVth Higher Order Workshop.* Workshops in Computing, Springer, 1991.
2. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
3. Y. Isobe and M. Roggenbach. A Generic Theorem Prover of CSP Refinement. *11th International Conference of Tools and Algorithms for the Construction and Analysis of Systems, TACAS'05.* LNCS 3440, Springer, 2005.
4. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, Springer LNCS, **2283**, 2002.
5. H. Tej and B. Wolff. A Corrected Failure-Divergence Model for CSP in Isabelle/HOL. *Industrial Applications and Strengthened Foundations of Formal Methods, FME'97.* LNCS 1313, Springer, 1997.