

# The HOL-CSP Refinement Toolkit

Safouan Taha      Burkhart Wolff      Lina Ye

September 13, 2023



# Abstract

Recently, a modern version of Roscoes and Brookes [3] Failure-Divergence Semantics for CSP has been formalized in Isabelle [10].

We use this formal development called HOL-CSP2.0 to analyse a family of refinement notions, comprising classic and new ones. This analysis enables to derive a number of properties that allow to deepen the understanding of these notions, in particular with respect to specification decomposition principles for the case of infinite sets of events. The established relations between the refinement relations help to clarify some obscure points in the CSP literature, but also provide a weapon for shorter refinement proofs. Furthermore, we provide a framework for state-normalisation allowing to formally reason on parameterised process architectures.

As a result, we have a modern environment for formal proofs of concurrent systems that allow for the combination of general infinite processes with locally finite ones in a logically safe way. We demonstrate these verification-techniques for classical, generalised examples: The CopyBuffer for arbitrary data and the Dijkstra's Dining Philosopher Problem of arbitrary size.

If you consider to cite this work, please refer to [11].



# Contents

<b>1</b>	<b>Context</b>	<b>7</b>
1.1	Introduction . . . . .	7
1.2	The Global Architecture of CSP_RefTk . . . . .	9
1.3	Non-terminating Runs . . . . .	10
1.4	Lifelock Freeness . . . . .	11
1.5	New laws . . . . .	11
<b>2</b>	<b>Normalisation of Deterministic CSP Processes</b>	<b>13</b>
2.1	Deterministic normal-forms with explicit state . . . . .	13
2.2	Interleaving product lemma . . . . .	13
2.3	Synchronous product lemma . . . . .	14
2.4	Consequences . . . . .	14
<b>3</b>	<b>Examples</b>	<b>15</b>
3.1	CopyBuffer Refinement over an infinite alphabet . . . . .	15
3.1.1	The Copy-Buffer vs. reference processes . . . . .	15
3.1.2	... and abstract consequences . . . . .	15
3.2	Generalized Dining Philosophers . . . . .	16
3.2.1	Preliminary lemmas for proof automation . . . . .	16
3.2.2	The dining processes definition . . . . .	16
3.2.3	Translation into normal form . . . . .	17
3.2.4	The normal form for the global philosopher network . . . . .	21
3.2.5	The complete process system under normal form . . . . .	22
3.2.6	And finally: Philosophers may dine ! Always ! . . . . .	22
<b>4</b>	<b>Conclusion</b>	<b>25</b>



# Chapter 1

## Context

### 1.1 Introduction

Communicating Sequential Processes CSP is a language to specify and verify patterns of interaction of concurrent systems. Together with CCS and LOTOS, it belongs to the family of *process algebras*. CSP's rich theory comprises denotational, operational and algebraic semantic facets and has influenced programming languages such as Limbo, Crystal, Clojure and most notably Golang [5]. CSP has been applied in industry as a tool for specifying and verifying the concurrent aspects of hardware systems, such as the T9000 transputer [1].

The theory of CSP, in particular the denotational Failure/Divergence Denotational Semantics, has been initially proposed in the book by Tony Hoare [6], but evolved substantially since [2, 3, 8].

Verification of CSP properties has been centered around the notion of *process refinement orderings*, most notably  $\sqsubseteq_{FD}$ - and  $\sqsubseteq$ -. The latter turns the denotational domain of CSP into a Scott cpo [9], which yields semantics for the fixed point operator  $\mu x. f(x)$  provided that  $f$  is continuous with respect to  $\sqsubseteq$ -. Since it is possible to express deadlock-freeness and livelock-freeness as a refinement problem, the verification of properties has been reduced traditionally to a model-checking problem for a finite set of events  $A$ .

We are interested in verification techniques for arbitrary event sets  $A$  or arbitrarily parameterized processes. Such processes can be used to model dense-timed processes, processes with dynamic thread creation, and processes with unbounded thread-local variables and buffers. Events may even be higher-order objects such as functions or again processes, paving the way for the modeling of re-programmable compute servers or dynamic distributed computing architectures. However, this adds substantial complexity to the process theory: when it comes to study the interplay of different denotational models, refinement-orderings, and side-conditions for continuity, paper-and-pencil proofs easily reach their limits of precision.

Several attempts have been undertaken to develop the formal theory of CSP in an interactive proof system, mostly in Isabelle/HOL [4, 12, 7]. This work is based on the most recent instance in this line, HOL-CSP 2.0, which has been published as AFP submission [10] and whose development is hosted at <https://gitlri.lri.fr/burkhart.wolff/hol-csp2.0>.

The present AFP Module is an add-on on this work and develops some support for

1. advanced induction schemes (mutual fixed-point Induction, K-induction),
2. bridge-Lemmas between the classical refinement relations in the FD-semantics, which allow for reduced refinement proof complexity in certain cases, and
3. a theory of explicit state normalisation which allows for proofs over certain communicating networks of arbitrary size.



## 1.2 The Global Architecture of CSP\_RefTk

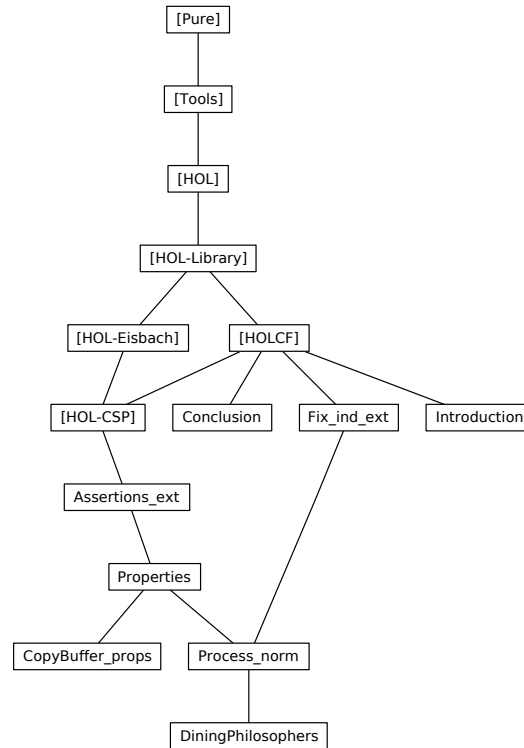


Figure 1.1: The overall architecture: HOLCF, HOL-CSP, and CSP\_RefTk

The global architecture of CSP\_RefTk is shown in [Figure 1.1](#). The entire package resides on:

1. HOL-Eisbach from the Isabelle/HOL distribution,
2. HOLCF from the Isabelle/HOL distribution, and
3. HOL-CSP 2.0 from the Isabelle Archive of Formal Proofs.

The theories `Assertion_ext` and `Fixind_ext` are extensions of the corresponding theories in HOL-CSP.

```

__ *****
Project : CSP-RefTK - A Refinement Toolkit for HOL-CSP * Version : 1.0 * * Au-
Author  : Burkhart Wolff, Safouan Taha, Lina Ye. * * This file : Theorems on DF
and LF * * Copyright (c) 2020 Université Paris-Saclay, France * * All rights re-
served. * * Redistribution and use in source and binary forms, with or without *
modification, are permitted provided that the following conditions are * met: * *

```

\* Redistributions of source code must retain the above copyright \* notice, this list of conditions and the following disclaimer. \* \* \* Redistributions in binary form must reproduce the above \* copyright notice, this list of conditions and the following \* disclaimer in the documentation and/or other materials provided \* with the distribution. \* \* \* Neither the name of the copyright holders nor the names of its \* contributors may be used to endorse or promote products derived \* from this software without specific prior written permission. \* \* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS \* "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT \* LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR \* A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT \* OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, \* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT \* LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, \* DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY \* THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT \* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE \* OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.  
 \*\*\*\*\*

```
theory Properties
  imports HOL-CSP.Assertions
begin
```

### 1.3 Non-terminating Runs

**definition** *non-terminating* :: 'a process  $\Rightarrow$  bool  
 where *non-terminating*  $P \equiv \text{RUN UNIV} \sqsubseteq_T P$

**corollary** *non-terminating-refine-DF*: *non-terminating*  $P = \text{DF UNIV} \sqsubseteq_T P$   
**and** *non-terminating-refine-CHAOS*: *non-terminating*  $P = \text{CHAOS UNIV} \sqsubseteq_T P$   
 <proof>

**lemma** *non-terminating-is-right*: *non-terminating*  $(P::'a \text{ process}) \longleftrightarrow (\forall s \in \mathcal{T} P. \text{tickFree } s)$   
 <proof>

**lemma** *nonterminating-implies-div-free*: *non-terminating*  $P \Longrightarrow \mathcal{D} P = \{\}$   
 <proof>

**lemma** *non-terminating-implies-F*: *non-terminating*  $P \Longrightarrow \text{CHAOS UNIV} \sqsubseteq_F P$   
 <proof>

**corollary** *non-terminating-F*: *non-terminating*  $P = \text{CHAOS UNIV} \sqsubseteq_F P$   
 <proof>

**corollary** *non-terminating-FD*: *non-terminating*  $P = \text{CHAOS UNIV} \sqsubseteq_{FD} P$

*<proof>*

## 1.4 Lifelock Freeness

**thm** *lifelock-free-def*

**corollary** *lifelock-free-is-non-terminating*: *lifelock-free*  $P =$  *non-terminating*  $P$   
*<proof>*

**lemma** *div-free-divergence-refine*:

$$\mathcal{D} P = \{\} \longleftrightarrow \text{CHAOS}_{\text{SKIP}} \text{ UNIV} \sqsubseteq_D P$$

$$\mathcal{D} P = \{\} \longleftrightarrow \text{CHAOS UNIV} \sqsubseteq_D P$$

$$\mathcal{D} P = \{\} \longleftrightarrow \text{RUN UNIV} \sqsubseteq_D P$$

$$\mathcal{D} P = \{\} \longleftrightarrow \text{DF}_{\text{SKIP}} \text{ UNIV} \sqsubseteq_D P$$

$$\mathcal{D} P = \{\} \longleftrightarrow \text{DF UNIV} \sqsubseteq_D P$$

*<proof>*

**definition** *lifelock-free-v2* :: 'a process  $\Rightarrow$  bool

**where** *lifelock-free-v2*  $P \equiv \text{CHAOS}_{\text{SKIP}} \text{ UNIV} \sqsubseteq_{FD} P$

**lemma** *div-free-is-lifelock-free-v2*: *lifelock-free-v2*  $P \longleftrightarrow \mathcal{D} P = \{\}$   
*<proof>*

**lemma** *lifelock-free-is-lifelock-free-v2*: *lifelock-free*  $P \Longrightarrow$  *lifelock-free-v2*  $P$   
*<proof>*

**corollary** *deadlock-free-v2-is-lifelock-free-v2*: *deadlock-free-v2*  $P \Longrightarrow$  *lifelock-free-v2*  $P$   
*<proof>*

## 1.5 New laws

**lemma** *non-terminating-Seq*:

$$\text{non-terminating } P \Longrightarrow (P ; Q) = P$$

*<proof>*

**lemma** *non-terminating-Sync*:

$$\text{non-terminating } P \Longrightarrow \text{lifelock-free-v2 } Q \Longrightarrow \text{non-terminating } (P \llbracket A \rrbracket Q)$$

*<proof>*

**lemmas** *non-terminating-Par* = *non-terminating-Sync*[**where**  $A = \langle \text{UNIV} \rangle$ ]  
**and** *non-terminating-Inter* = *non-terminating-Sync*[**where**  $A = \langle \{\} \rangle$ ]

**end**



## Chapter 2

# Normalisation of Deterministic CSP Processes

**theory** *Process-norm*

**imports** *Properties HOL-CSP.Induction-ext*

**begin**

### 2.1 Deterministic normal-forms with explicit state

**abbreviation** *P-dnorm*  $\tau\ v \equiv (\mu\ X.\ (\lambda\ s.\ \square\ e \in (\tau\ s) \rightarrow X\ (v\ s\ e)))$

**notation**  $P\text{-dnorm}\ (P_{norm}[\_,\_] 60)$

**lemma** *dnorm-cont[simp]*:

**fixes**  $\tau::'\sigma::type \Rightarrow 'event::type\ set$  **and**  $v::'\sigma \Rightarrow 'event \Rightarrow 'sigma$   
**shows**  $cont\ (\lambda X.\ (\lambda s.\ \square e \in (\tau\ s) \rightarrow X\ (v\ s\ e)))$  (**is**  $cont\ ?f$ )  
(*proof*)

### 2.2 Interleaving product lemma

**lemma** *dnorm-inter*:

**fixes**  $\tau_1::'\sigma_1::type \Rightarrow 'event::type\ set$  **and**  $\tau_2::'\sigma_2::type \Rightarrow 'event\ set$   
**and**  $v_1::'\sigma_1 \Rightarrow 'event \Rightarrow 'sigma_1$  **and**  $v_2::'\sigma_2 \Rightarrow 'event \Rightarrow 'sigma_2$   
**defines**  $P: P \equiv P_{norm}[\tau_1, v_1]$  (**is**  $P \equiv fix.\ (\Lambda\ X.\ ?P\ X)$ )  
**defines**  $Q: Q \equiv P_{norm}[\tau_2, v_2]$  (**is**  $Q \equiv fix.\ (\Lambda\ X.\ ?Q\ X)$ )

**assumes** *indep*:  $\langle \forall\ s_1\ s_2.\ \tau_1\ s_1 \cap \tau_2\ s_2 = \{\} \rangle$

**defines** *Tr*:  $\tau \equiv (\lambda(s_1, s_2).\ \tau_1\ s_1 \cup \tau_2\ s_2)$

**defines** *Up*:  $v \equiv (\lambda(s_1, s_2)\ e.\ \text{if } e \in \tau_1\ s_1 \text{ then } (v_1\ s_1\ e, s_2)$   
 $\text{else if } e \in \tau_2\ s_2 \text{ then } (s_1, v_2\ s_2\ e) \text{ else } (s_1, s_2))$

**defines** *S*:  $S \equiv P_{norm}[\tau, v]$  (**is**  $S \equiv fix.\ (\Lambda\ X.\ ?S\ X)$ )

**shows**  $(P \ s_1 \ ||| \ Q \ s_2) = S \ (s_1, s_2)$

*<proof>*

## 2.3 Synchronous product lemma

**lemma** *dnorm-par*:

**fixes**  $\tau_1 :: 's_1 :: type \Rightarrow 'event :: type \ set$  **and**  $\tau_2 :: 's_2 :: type \Rightarrow 'event \ set$   
**and**  $v_1 :: 's_1 \Rightarrow 'event \Rightarrow 's_1$  **and**  $v_2 :: 's_2 \Rightarrow 'event \Rightarrow 's_2$   
**defines**  $P: P \equiv P_{norm} \llbracket \tau_1, v_1 \rrbracket$  (**is**  $P \equiv fix \cdot (\Lambda \ X. ?P \ X)$ )  
**defines**  $Q: Q \equiv P_{norm} \llbracket \tau_2, v_2 \rrbracket$  (**is**  $Q \equiv fix \cdot (\Lambda \ X. ?Q \ X)$ )

**defines**  $Tr: \tau \equiv (\lambda (s_1, s_2). \ \tau_1 \ s_1 \ \cap \ \tau_2 \ s_2)$   
**defines**  $Up: v \equiv (\lambda (s_1, s_2) \ e. \ (v_1 \ s_1 \ e, \ v_2 \ s_2 \ e))$   
**defines**  $S: S \equiv P_{norm} \llbracket \tau, v \rrbracket$  (**is**  $S \equiv fix \cdot (\Lambda \ X. ?S \ X)$ )

**shows**  $(P \ s_1 \ || \ Q \ s_2) = S \ (s_1, s_2)$

*<proof>*

## 2.4 Consequences

**inductive-set**  $\mathfrak{R}$  **for**  $\tau :: 's :: type \Rightarrow 'event :: type \ set$   
**and**  $v :: 's \Rightarrow 'event \Rightarrow 's$   
**and**  $\sigma_0 :: 's$   
**where** *rbase*:  $\sigma_0 \in \mathfrak{R} \ \tau \ v \ \sigma_0$   
 $|$  *rstep*:  $s \in \mathfrak{R} \ \tau \ v \ \sigma_0 \Longrightarrow e \in \tau \ s \Longrightarrow v \ s \ e \in \mathfrak{R} \ \tau \ v \ \sigma_0$

— Deadlock freeness

**lemma** *deadlock-free-dnorm-* :

**fixes**  $\tau :: 's :: type \Rightarrow 'event :: type \ set$   
**and**  $v :: 's \Rightarrow 'event \Rightarrow 's$   
**and**  $\sigma_0 :: 's$   
**assumes** *non-reachable-sink*:  $\forall s \in \mathfrak{R} \ \tau \ v \ \sigma_0. \ \tau \ s \neq \{\}$   
**defines**  $P: P \equiv P_{norm} \llbracket \tau, v \rrbracket$  (**is**  $P \equiv fix \cdot (\Lambda \ X. ?P \ X)$ )  
**shows**  $s \in \mathfrak{R} \ \tau \ v \ \sigma_0 \Longrightarrow deadlock\text{-free}\text{-}v2 \ (P \ s)$

*<proof>*

**lemmas** *deadlock-free-dnorm = deadlock-free-dnorm-[rotated, OF rbase, rule-format]*

**end**

# Chapter 3

## Examples

### 3.1 CopyBuffer Refinement over an infinite alphabet

```
theory CopyBuffer-props
  imports HOL-CSP.CopyBuffer Properties
begin
```

#### 3.1.1 The Copy-Buffer vs. reference processes

```
lemma DF-COPY: (DF (range left  $\cup$  range right))  $\sqsubseteq_{FD}$  COPY
  <proof>
```

#### 3.1.2 ... and abstract consequences

```
corollary df-COPY: deadlock-free COPY
  and lf-COPY: lifelock-free COPY
  <proof>
```

```
corollary df-v2-COPY: deadlock-free-v2 COPY
  and lf-v2-COPY: lifelock-free-v2 COPY
  and nt-COPY: non-terminating COPY
  <proof>
```

```
lemma DF-SYSTEM: DF UNIV  $\sqsubseteq_{FD}$  SYSTEM
  <proof>
```

```
corollary df-SYSTEM: deadlock-free SYSTEM
  and lf-SYSTEM: lifelock-free SYSTEM
  <proof>
```

```
corollary df-v2-SYSTEM: deadlock-free-v2 SYSTEM
  and lf-v2-SYSTEM: lifelock-free-v2 SYSTEM
  and nt-SYSTEM: non-terminating SYSTEM
  <proof>
```

end

## 3.2 Generalized Dining Philosophers

```
theory DiningPhilosophers
  imports Process-norm
begin
```

### 3.2.1 Preliminary lemmas for proof automation

```
lemma Suc-mod:  $n > 1 \implies i \neq \text{Suc } i \text{ mod } n$ 
  <proof>
```

```
lemmas suc-mods = Suc-mod Suc-mod[symmetric]
```

```
lemma l-suc:  $n > 1 \implies \neg n \leq \text{Suc } 0$ 
  <proof>
```

```
lemma minus-suc:  $n > 0 \implies n - \text{Suc } 0 \neq n$ 
  <proof>
```

```
lemma numeral-4-eq-4:  $4 = \text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } 0)))$ 
  <proof>
```

```
lemma numeral-5-eq-5:  $5 = \text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } (\text{Suc } 0))))$ 
  <proof>
```

### 3.2.2 The dining processes definition

```
locale DiningPhilosophers =
```

```
  fixes  $N::\text{nat}$ 
  assumes  $N\text{-g1}[simp] : N > 1$ 
```

```
begin
```

```
datatype dining-event = picks (phil:nat) (fork:nat)
  | putsdown (phil:nat) (fork:nat)
```

```
definition RPHIL::  $\text{nat} \Rightarrow \text{dining-event process}$ 
  where RPHIL  $i = (\mu X. (\text{picks } i \ i \rightarrow (\text{picks } i \ (i-1) \rightarrow (\text{putsdown } i \ (i-1) \rightarrow (\text{putsdown } i \ i \rightarrow X))))))$ 
```

```
definition LPHIL0::  $\text{dining-event process}$ 
  where LPHIL0 =  $(\mu X. (\text{picks } 0 \ (N-1) \rightarrow (\text{picks } 0 \ 0 \rightarrow (\text{putsdown } 0 \ 0 \rightarrow (\text{putsdown } 0 \ (N-1) \rightarrow X))))))$ 
```

```
definition FORK ::  $\text{nat} \Rightarrow \text{dining-event process}$ 
```



**where**  $FORK\ i = (\mu\ X.\ (picks\ i\ i \rightarrow (putsdown\ i\ i \rightarrow X)))$   
 $\square\ (picks\ ((i+1)\ mod\ N)\ i \rightarrow (putsdown\ ((i+1)\ mod\ N)\ i \rightarrow X))$

**abbreviation**  $foldPHILs\ n \equiv fold\ (\lambda\ i\ P.\ P\ |||\ RPHIL\ i)\ [1..<\ n]\ (LPHIL0)$

**abbreviation**  $foldFORKs\ n \equiv fold\ (\lambda\ i\ P.\ P\ |||\ FORK\ i)\ [1..<\ n]\ (FORK\ 0)$

**abbreviation**  $PHILs \equiv foldPHILs\ N$

**abbreviation**  $FORKs \equiv foldFORKs\ N$

**corollary**  $FORKs-def2: FORKs = fold\ (\lambda\ i\ P.\ P\ |||\ FORK\ i)\ [0..<\ N]\ SKIP$   
 $\langle proof \rangle$

**corollary**  $N = 3 \implies PHILs = (LPHIL0\ |||\ RPHIL\ 1\ |||\ RPHIL\ 2)$   
 $\langle proof \rangle$

**definition**  $DINING :: dining-event\ process$

**where**  $DINING = (FORKs\ ||\ PHILs)$

### Unfolding rules

**lemma**  $RPHIL-rec:$

$RPHIL\ i = (picks\ i\ i \rightarrow (picks\ i\ (i-1) \rightarrow (putsdown\ i\ (i-1) \rightarrow (putsdown\ i\ i \rightarrow$   
 $RPHIL\ i))))$   
 $\langle proof \rangle$

**lemma**  $LPHIL0-rec:$

$LPHIL0 = (picks\ 0\ (N-1) \rightarrow (picks\ 0\ 0 \rightarrow (putsdown\ 0\ 0 \rightarrow (putsdown\ 0\ (N-1)$   
 $\rightarrow LPHIL0))))$   
 $\langle proof \rangle$

**lemma**  $FORK-rec: FORK\ i = ( (picks\ i\ i \rightarrow (putsdown\ i\ i \rightarrow (FORK\ i)))$

$\square\ (picks\ ((i+1)\ mod\ N)\ i \rightarrow (putsdown\ ((i+1)\ mod\ N)\ i \rightarrow$   
 $(FORK\ i))))$   
 $\langle proof \rangle$

### 3.2.3 Translation into normal form

**lemma**  $N-pos[simp]: N > 0$

$\langle proof \rangle$

**lemmas**  $N-pos-simps[simp] = suc-mods[OF\ N-g1]\ l-suc[OF\ N-g1]\ minus-suc[OF\ N-pos]$

The one-fork process

**type-synonym**  $id_{fork} = nat$

**type-synonym**  $\sigma_{fork} = nat$

**definition** *fork-transitions*::  $id_{fork} \Rightarrow \sigma_{fork} \Rightarrow$  dining-event set ( $Tr_f$ )  
**where**  $Tr_f\ i\ s =$  (if  $s = 0$  then  $\{picks\ i\ i\} \cup \{picks\ ((i+1)\ mod\ N)\ i\}$   
 else if  $s = 1$  then  $\{putsdown\ i\ i\}$   
 else if  $s = 2$  then  $\{putsdown\ ((i+1)\ mod\ N)\ i\}$   
 else  $\{\}$ )

**declare**  $Un-insert-right[simp\ del]$   $Un-insert-left[simp\ del]$

**lemma**  $ev-id_{fork}x[simp]$ :  $e \in Tr_f\ i\ s \Rightarrow fork\ e = i$   
 ⟨proof⟩

**definition**  $\sigma_{fork-update}$ ::  $id_{fork} \Rightarrow \sigma_{fork} \Rightarrow$  dining-event  $\Rightarrow \sigma_{fork}$  ( $Up_f$ )  
**where**  $Up_f\ i\ s\ e =$  ( if  $e = (picks\ i\ i)$  then 1  
 else if  $e = (picks\ ((i+1)\ mod\ N)\ i)$  then 2  
 else 0 )

**definition**  $FORK_{norm}$ ::  $id_{fork} \Rightarrow \sigma_{fork} \Rightarrow$  dining-event process  
**where**  $FORK_{norm}\ i = P_{norm} \llbracket Tr_f\ i , Up_f\ i \rrbracket$

**lemma**  $FORK_{norm-rec}$ :  $FORK_{norm}\ i = (\lambda\ s.\ \square\ e \in (Tr_f\ i\ s) \rightarrow FORK_{norm}\ i$   
 ( $Up_f\ i\ s\ e$ ))  
 ⟨proof⟩

**lemma**  $FORK-refines-FORK_{norm}$ :  $FORK_{norm}\ i\ 0 \sqsubseteq_{FD}\ FORK\ i$   
 ⟨proof⟩

**lemma**  $FORK_{norm-refines-FORK}$ :  $FORK\ i \sqsubseteq_{FD}\ FORK_{norm}\ i\ 0$   
 ⟨proof⟩

**lemma**  $FORK_{norm-is-FORK}$ :  $FORK\ i = FORK_{norm}\ i\ 0$   
 ⟨proof⟩

The all-forks process in normal form

**type-synonym**  $\sigma_{forks} = nat\ list$

**definition** *forks-transitions*::  $nat \Rightarrow \sigma_{forks} \Rightarrow$  dining-event set ( $Tr_F$ )  
**where**  $Tr_F\ n\ fs = (\bigcup_{i < n} Tr_f\ i\ (fs!i))$

**lemma** *forks-transitions-take*:  $Tr_F\ n\ fs = Tr_F\ n\ (take\ n\ fs)$   
 ⟨proof⟩

**definition**  $\sigma_{forks-update}$ ::  $\sigma_{forks} \Rightarrow$  dining-event  $\Rightarrow \sigma_{forks}$  ( $Up_F$ )  
**where**  $Up_F\ fs\ e = (let\ i=(fork\ e)\ in\ fs[i:=(Up_f\ i\ (fs!i)\ e)])$

**lemma** *forks-update-take*:  $\text{take } n \ (Up_F \ fs \ e) = Up_F \ (\text{take } n \ fs) \ e$   
 ⟨proof⟩

**definition**  $FORKs_{norm} :: nat \Rightarrow \sigma_{forks} \Rightarrow \text{dining-event process}$   
 where  $FORKs_{norm} \ n = P_{norm} \llbracket Tr_F \ n, Up_F \rrbracket$

**lemma** *FORKs<sub>norm</sub>-rec*:  $FORKs_{norm} \ n = (\lambda \ fs. \square \ e \in (Tr_F \ n \ fs) \rightarrow FORKs_{norm} \ n \ (Up_F \ fs \ e))$   
 ⟨proof⟩

**lemma** *FORKs<sub>norm</sub>-0*:  $FORKs_{norm} \ 0 \ fs = STOP$   
 ⟨proof⟩

**lemma** *FORKs<sub>norm</sub>-1-dir1*:  $\text{length } fs > 0 \implies FORKs_{norm} \ 1 \ fs \sqsubseteq_{FD} (FORK_{norm} \ 0 \ (fs!0))$   
 ⟨proof⟩

**lemma** *FORKs<sub>norm</sub>-1-dir2*:  $\text{length } fs > 0 \implies (FORK_{norm} \ 0 \ (fs!0)) \sqsubseteq_{FD} FORKs_{norm} \ 1 \ fs$   
 ⟨proof⟩

**lemma** *FORKs<sub>norm</sub>-1*:  $\text{length } fs > 0 \implies (FORK_{norm} \ 0 \ (fs!0)) = FORKs_{norm} \ 1 \ fs$   
 ⟨proof⟩

**lemma** *FORKs<sub>norm</sub>-unfold*:  
 $0 < n \implies \text{length } fs = Suc \ n \implies$   
 $FORKs_{norm} \ (Suc \ n) \ fs = (FORKs_{norm} \ n \ (\text{butlast } fs) \ || \ (FORK_{norm} \ n \ (fs!n)))$   
 ⟨proof⟩

**lemma** *ft*:  $0 < n \implies FORKs_{norm} \ n \ (\text{replicate } n \ 0) = \text{fold}FORKs \ n$   
 ⟨proof⟩

**corollary** *FORKs-is-FORKs<sub>norm</sub>*:  $FORKs_{norm} \ N \ (\text{replicate } N \ 0) = FORKs$   
 ⟨proof⟩

The one-philosopher process in normal form:

**type-synonym** *phil-id* = nat

**type-synonym** *phil-state* = nat

**definition** *rphil-transitions*:  $\text{phil-id} \Rightarrow \text{phil-state} \Rightarrow \text{dining-event set } (Tr_{rp})$   
 where  $Tr_{rp} \ i \ s =$  ( if  $s = 0$  then {picks  $i \ i$ }  
 else if  $s = 1$  then {picks  $i \ (i-1)$ }  
 else if  $s = 2$  then {putsdown  $i \ (i-1)$ }  
 else if  $s = 3$  then {putsdown  $i \ i$ }

else  $\{\}$ )

**definition** *lphil0-transitions*:: *phil-state*  $\Rightarrow$  *dining-event set* ( $Tr_{lp}$ )

where  $Tr_{lp} s =$  ( if  $s = 0$  then  $\{\text{picks } 0 (N-1)\}$   
 else if  $s = 1$  then  $\{\text{picks } 0 0\}$   
 else if  $s = 2$  then  $\{\text{putsdown } 0 0\}$   
 else if  $s = 3$  then  $\{\text{putsdown } 0 (N-1)\}$   
 else  $\{\}$ )

**corollary** *rphil-phil*:  $e \in Tr_{rp} i s \Rightarrow \text{phil } e = i$

and *lphil0-phil*:  $e \in Tr_{lp} s \Rightarrow \text{phil } e = 0$

*<proof>*

**definition** *rphil-state-update*:: *id\_fork*  $\Rightarrow$   *$\sigma_{fork}$*   $\Rightarrow$  *dining-event*  $\Rightarrow$   *$\sigma_{fork}$*  ( $Up_{rp}$ )

where  $Up_{rp} i s e =$  ( if  $e = (\text{picks } i i)$  then 1  
 else if  $e = (\text{picks } i (i-1))$  then 2  
 else if  $e = (\text{putsdown } i (i-1))$  then 3  
 else 0 )

**definition** *lphil0-state-update*::  *$\sigma_{fork}$*   $\Rightarrow$  *dining-event*  $\Rightarrow$   *$\sigma_{fork}$*  ( $Up_{lp}$ )

where  $Up_{lp} s e =$  ( if  $e = (\text{picks } 0 (N-1))$  then 1  
 else if  $e = (\text{picks } 0 0)$  then 2  
 else if  $e = (\text{putsdown } 0 0)$  then 3  
 else 0 )

**definition** *RPHIL<sub>norm</sub>*:: *id\_fork*  $\Rightarrow$   *$\sigma_{fork}$*   $\Rightarrow$  *dining-event process*

where  $RPHIL_{norm} i = P_{norm} \llbracket Tr_{rp} i, Up_{rp} i \rrbracket$

**definition** *LPHIL0<sub>norm</sub>*::  *$\sigma_{fork}$*   $\Rightarrow$  *dining-event process*

where  $LPHIL0_{norm} = P_{norm} \llbracket Tr_{lp}, Up_{lp} \rrbracket$

**lemma** *RPHIL<sub>norm</sub>-rec*:  $RPHIL_{norm} i = (\lambda s. \square e \in (Tr_{rp} i s) \rightarrow RPHIL_{norm} i (Up_{rp} i s e))$

*<proof>*

**lemma** *LPHIL0<sub>norm</sub>-rec*:  $LPHIL0_{norm} = (\lambda s. \square e \in (Tr_{lp} s) \rightarrow LPHIL0_{norm} (Up_{lp} s e))$

*<proof>*

**lemma** *RPHIL-refines-RPHIL<sub>norm</sub>*:

assumes *i-pos*:  $i > 0$

shows  $RPHIL_{norm} i 0 \sqsubseteq_{FD} RPHIL i$

*<proof>*

**lemma** *LPHIL0-refines-LPHIL0<sub>norm</sub>*:  $LPHIL0_{norm} 0 \sqsubseteq_{FD} LPHIL0$

*<proof>*



**lemma** *PHILs<sub>norm</sub>-1*:  $\text{length } ps > 0 \implies \text{PHILs}_{norm} 1 ps = (\text{LPHIL0}_{norm} (ps!0))$   
 ⟨proof⟩

**lemma** *PHILs<sub>norm</sub>-unfold*:

**assumes**  $n\text{-pos}: 0 < n$

**shows**  $\text{length } ps = \text{Suc } n \implies$

$\text{PHILs}_{norm} (\text{Suc } n) ps = (\text{PHILs}_{norm} n (\text{butlast } ps)) ||| (\text{RPHIL}_{norm} n (ps!n))$   
 ⟨proof⟩

**lemma** *pt*:  $0 < n \implies \text{PHILs}_{norm} n (\text{replicate } n 0) = \text{foldPHILs } n$   
 ⟨proof⟩

**corollary** *PHILs-is-PHILs<sub>norm</sub>*:  $\text{PHILs}_{norm} N (\text{replicate } N 0) = \text{PHILs}$   
 ⟨proof⟩

### 3.2.5 The complete process system under normal form

**definition** *dining-transitions*::  $\text{nat} \Rightarrow \sigma_{phil} \times \sigma_{forks} \Rightarrow \text{dining-event set } (Tr_D)$   
**where**  $Tr_D n = (\lambda(ps,fs). (Tr_P n ps) \cap (Tr_F n fs))$

**definition** *dining-state-update*::

$\sigma_{phil} \times \sigma_{forks} \Rightarrow \text{dining-event} \Rightarrow \sigma_{phil} \times \sigma_{forks} (Up_D)$

**where**  $Up_D = (\lambda(ps,fs) e. (Up_P ps e, Up_F fs e))$

**definition** *DINING<sub>norm</sub>*::  $\text{nat} \Rightarrow \sigma_{phil} \times \sigma_{forks} \Rightarrow \text{dining-event process}$   
**where**  $\text{DINING}_{norm} n = P_{norm} [[Tr_D n, Up_D]]$

**lemma** *ltsDining-rec*:  $\text{DINING}_{norm} n = (\lambda s. \square e \in (Tr_D n s) \rightarrow \text{DINING}_{norm} n (Up_D s e))$   
 ⟨proof⟩

**lemma** *DINING-is-DINING<sub>norm</sub>*:  $\text{DINING} = \text{DINING}_{norm} N (\text{replicate } N 0, \text{replicate } N 0)$   
 ⟨proof⟩

### 3.2.6 And finally: Philosophers may dine ! Always !

**corollary** *lphil-states*:  $Up_{lp} r e = 0 \vee Up_{lp} r e = 1 \vee Up_{lp} r e = 2 \vee Up_{lp} r e = 3$

**and** *rphil-states*:  $Up_{rp} i r e = 0 \vee Up_{rp} i r e = 1 \vee Up_{rp} i r e = 2 \vee Up_{rp} i r e = 3$

⟨proof⟩

**lemma** *dining-events*:

$e \in Tr_D N s \implies$

$(\exists i \in \{1..<N\}. e = \text{picks } i i \vee e = \text{picks } i (i-1) \vee e = \text{putsdown } i i \vee e = \text{putsdown } i (i-1))$

$\vee (e = \text{picks } 0 \ 0 \vee e = \text{picks } 0 \ (N-1) \vee e = \text{putsdown } 0 \ 0 \vee e = \text{putsdown } 0 \ (N-1))$   
 ⟨proof⟩

**definition** *inv-dining*  $ps \ fs \equiv$   
 $(\forall i. \text{Suc } i < N \longrightarrow ((fs!(\text{Suc } i) = 1) \longleftrightarrow ps!\text{Suc } i \neq 0)) \wedge (fs!(N-1) = 2 \longleftrightarrow ps!0 \neq 0)$   
 $\wedge (\forall i < N - 1. \quad fs!i = 2 \longleftrightarrow ps!\text{Suc } i = 2) \wedge (fs!0 = 1 \longleftrightarrow ps!0 = 2)$   
 $\wedge (\forall i < N. \ fs!i = 0 \vee fs!i = 1 \vee fs!i = 2)$   
 $\wedge (\forall i < N. \ ps!i = 0 \vee ps!i = 1 \vee ps!i = 2 \vee ps!i = 3)$   
 $\wedge \text{length } fs = N \wedge \text{length } ps = N$

**lemma** *inv-DINING*:  $s \in \mathfrak{R} (Tr_D \ N) \ Up_D (\text{replicate } N \ 0, \text{replicate } N \ 0) \implies$   
*inv-dining*  $(fst \ s) \ (snd \ s)$   
 ⟨proof⟩

**lemma** *inv-implies-DF*: *inv-dining*  $ps \ fs \implies Tr_D \ N \ (ps, fs) \neq \{\}$   
 ⟨proof⟩

**corollary** *DF-DINING*: *deadlock-free-v2 DINING*  
 ⟨proof⟩

**end**

**end**





## Chapter 4

# Conclusion

We presented a formalisation of the most comprehensive semantic model for CSP, a 'classical' language for the specification and analysis of concurrent systems studied in a rich body of literature. For this purpose, we ported [12] to a modern version of Isabelle, restructured the proofs, and extended the resulting theory of the language substantially. The result HOL-CSP 2 has been submitted to the Isabelle AFP [10], thus a fairly sustainable format accessible to other researchers and tools.

We developed a novel set of deadlock - and livelock inference proof principles based on classical and denotational characterizations. In particular, we formally investigated the relations between different refinement notions in the presence of deadlock - and livelock; an area where traditional CSP literature skates over the nitty-gritty details. Finally, we demonstrated how to exploit these results for deadlock/livelock analysis of protocols.

We put a large body of abstract CSP laws and induction principles together to form concrete verification technologies for generalized classical problems, which have been considered so far from the perspective of data-independence or structural parametricity. The underlying novel principle of "trading rich structure against rich state" allows one to convert processes into classical transition systems for which established invariant techniques become applicable.

Future applications of HOL-CSP 2 could comprise a combination with model checkers, where our theory with its derived rules can be used to certify the output of a model-checker over CSP. In our experience, labelled transition systems generated by model checkers may be used to steer inductions or to construct the normalized processes  $P_{norm}[\tau, \nu]$  automatically, thus combining efficient finite reasoning over finite sub-systems with globally infinite systems in a logically safe way.



# Bibliography

- [1] G. Barrett. Model checking in practice: the t9000 virtual channel processor. *IEEE Transactions on Software Engineering*, 21(2):69–78, Feb 1995.
- [2] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, 1984.
- [3] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency*, pages 281–305, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.
- [4] A. J. Camilleri. A higher order logic mechanization of the csp failure-divergence semantics. In G. Birtwistle, editor, *IV Higher Order Workshop, Banff 1990*, pages 123–150, London, 1991. Springer London.
- [5] A. Donovan and B. Kernighan. *The Go Programming Language*. Addison-Wesley Professional Computing Series. Pearson Education, 2015.
- [6] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
- [7] Y. Isobe and M. Roggenbach. Csp-prover: a proof tool for the verification of scalable concurrent systems. *Information and Media Technologies*, 5(1):32–39, 2010.
- [8] A. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1998.
- [9] D. Scott. Continuous lattices. In F. W. Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, pages 97–136, Berlin, Heidelberg, 1972. Springer.
- [10] S. Taha, L. Ye, and B. Wolff. HOL-CSP Version 2.0. *Archive of Formal Proofs*, Apr. 2019. <http://isa-afp.org/entries/HOL-CSP.html>.

- [11] S. Taha, L. Ye, and B. Wolff. Philosophers may Dine - Definitively! In C. A. Furia, editor, *Integrated Formal Methods (iFM)*, number 12546 in Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2020.
- [12] H. Tej and B. Wolff. A corrected failure divergence model for CSP in Isabelle/HOL. In J. S. Fitzgerald, C. B. Jones, and P. Lucas, editors, *Formal Methods Europe (FME)*, volume 1313 of *Lecture Notes in Computer Science*, pages 318–337, Heidelberg, 1997. Springer-Verlag.