

*IMP*_{concur} - A Semantics for a Language with
Concurrency based on HOL-CSP

Burkhart Wolff

September 29, 2026

Abstract

The theory IMP_{concur} provides a programming-language aspect to HOL-CSP. It extends the well-known imperative language IMP originated by Glenn Winskell by concurrent primitives like `lock` and `unlock` for semaphores, thread-local and system global program variables and the notion of threads that run in interleaving semantics. Threads can be parametric and used to construct concurrent process architectures.

Representing IMP_{concur} semantics can be done by a straight-forward translation into HOL-CSP; thus, IMP_{concur} can be seen as a thin layer on top of the theory of Concurrent Sequential Processes of Hoare, Brookes and Roscoe in its formalization HOL-CSP in Isabelle/HOL.

Notwithstanding its simplicity, IMP_{concur} covers many aspects of concurrency, non-termination, divergence, synchronization, states and race-conditions. We believe that IMP_{concur} may be relevant for readers interested in the link between programming languages and process algebras. Resulting new ways to concurrent program verification are an interesting object for future study. We also present a language extension for dynamic reconfigurations of thread- architectures.

This session is complemented by a syntactic frontend for IMP_{concur} , called *CIL*, which has been authored by Mathilde Needham and Zineddine Kermadj on their internship project in summer 2025.

Contents

1	<i>IMP_{concur}</i>-A Shallow Embedding in HOL-CSP	7
1.1	Introduction	7
1.2	The Syntax of <i>IMP_{concur}</i>	8
1.3	Events, States and Semaphores ("Bricks").	9
1.3.1	A simple Model of a Semaphore Family	10
1.3.2	A simple Model of a Global Variable Family	10
1.3.3	A simple Model of a Local Variable Family (Optional)	10
1.4	Denotational Semantics of <i>IMP_{concur}</i>	10
1.5	A Thread-System as HOL-CSPM-Architecture	11
1.6	A Semantic Corner-Case of <i>IMP_{concur}</i>	12
1.7	Another Semantic Corner-Case: Deadlock	13
1.7.1	A Technique to rewrite Interleaves to MPrefixes	13
1.8	Yet another Semantic Corner-Case: A Deadlock-Freeness Proof	15
1.8.1	On Coinduction for deadlock-freeness.	16
1.9	A more serious Example: An MPI Program	17
1.10	An Architectural Extension: Parameterized Threads	19
1.11	Denotational Semantics of <i>IMP_{concur}</i> -P, a parametric version of <i>IMP_{concur}</i>	19
1.12	Conclusion and Future Work	20
2	Appendix: A Syntactic Frontend for <i>IMP_{concur}</i>	21
2.1	Introduction	21
2.2	Interface	21
2.3	Parsing and Term-Reading	22
2.4	Context Checking	22
2.5	Semantic Check	22
2.6	Semantic Encoding	22
2.7	Impression	22
2.8	Tests	22

Chapter 1

IMP_{concur}-A Shallow Embedding in HOL-CSP

```
theory IMP-concur
imports HOL-CSPM
```

```
begin
```

1.1 Introduction

IMP_{concur} is intended to provide a more common (though still theoretic) programming-language aspect to CSP, and could be an interesting object of study in itself. This thin layer over HOL-CSP establishes a link between programming languages and process-algebras, a link that had already been present in the influential OCCAM language.

IMP_{concur} comprises the standard elements of the IMP - language (*SKIP*, *assignment*, *IF - THEN - ELSE*, *WHILE - DO* - and the sequential composition - ; -), plus the new features:

1. semaphores with *lock* and *unlock*, and
2. thread-global shared variables accessible via *LOAD* and *STORE* operations.

This file contains:

1. the abstract and concrete syntax of *IMP_{concur}*
2. bricks-specifications for semaphores and global shared variables,
3. a denotational semantics of *IMP_{concur}* converting *IMP_{concur}*-program systems into HOL-CSPM, and

4. some examples and tests.

The general theory should be developed elsewhere; this sample file shows the global construction principle of a combined semantics.

1.2 The Syntax of IMP_{concur}

type-synonym $SV = string$ — Shared Variables

type-synonym $V = string$ — (Thread-)Local Variables

type-synonym $MV = int$ — Mutual exclusion variables (MUTEX ids)

type-synonym $D = int$

type-synonym $\sigma = \langle V \Rightarrow D \rangle$

type-synonym $E_{arith} = \langle \sigma \Rightarrow D \rangle$

type-synonym $E_{bool} = \langle \sigma \Rightarrow bool \rangle$

type-synonym $F = \sigma \Rightarrow \sigma$

datatype $com = SKIP$

```
| assign V Earith      ( - := - [90,90]80)
| seq    com com      (infixr ; 78)
| cond   Ebool com com (IF (-)/ THEN (-)/ ELSE (-)/
                        [0,0,79]79)
| while  Ebool com      (WHILE (-)/ DO (-) [0,80]80)
| call   F

| STOP
| lock   MV
| unlock MV
| send   Earith SV      (STORE - TO - [90,90]80)
| rec    V SV          (LOAD - FROM - [90,90]80)
```

Note that it is straight forward to formulate the assert primitive in IMP_{concur} :

definition *assert where* $\langle assert E \equiv IF E THEN SKIP ELSE STOP \rangle$

... which reduces the checking of assertions to the proof of deadlock-freeness.

Two (non-sensical) example threads:

definition *Thread1 where*

```
< Thread1  $\equiv$  SKIP ;
    "a" := ( $\lambda\sigma. 1 + 2$ );
    IF ( $\lambda\sigma. \sigma$  "d" > 3) THEN lock 4 ELSE lock 3;
    WHILE ( $\lambda\sigma. \sigma$  "e" > 3) DO lock 4 ;
    STORE ( $\lambda\sigma. \sigma$  "d" + 1) TO "b";
    LOAD "a" FROM "b";
    unlock 3
>
```

```

⋖ Thread2 ≡ SKIP ;
    "a" := (λσ. 1 + 2);
    IF (λσ. σ "d" > 3) THEN lock 4 ELSE lock 3;
    WHILE (λσ. True) DO lock 4 ;
    STORE (λσ. σ "d" + 1) TO "b";
    LOAD "a" FROM "b";
    unlock 3
⋗

```

 $\langle ML \rangle$

The common set of events in a thread-system

lemma *L12[simp]*: $\langle \text{update}_{\text{local}} \ n \ m \notin \text{GVARS} \rangle \ \langle \text{proof} \rangle$

1.3.1 A simple Model of a Semaphore Family

definition $\text{semaphore} :: \langle \text{int} \Rightarrow \text{evs process} \rangle$

where $\langle \text{semaphore} \equiv \mu X. (\lambda n. \text{lock } n \rightarrow \text{unlock } n \rightarrow X \ n) \rangle$

lemma $\text{sema-rec} : \langle \text{semaphore } n = \text{lock } n \rightarrow \text{unlock } n \rightarrow \text{semaphore } n \rangle$
 $\langle \text{proof} \rangle$

1.3.2 A simple Model of a Global Variable Family

definition $\text{global-vars} :: \langle \sigma \Rightarrow SV \Rightarrow \text{evs process} \rangle$

where $\langle \text{global-vars} \equiv \mu X. (\lambda \sigma. (\lambda id. ((\text{read}_{\text{global}} \ id)!(\sigma \ id) \rightarrow X \ \sigma \ id) \sqcap ((\text{upd}_{\text{global}} \ id)?v \rightarrow X \ (\sigma(id := v)) \ id))) \rangle$

lemma $\text{global-vars-rec} : \langle \text{global-vars } \sigma \ n = ((\text{read}_{\text{global}} \ n)!(\sigma \ n) \rightarrow \text{global-vars } \sigma \ n) \rangle$
 $\sqcap ((\text{upd}_{\text{global}} \ n)?v \rightarrow \text{global-vars } (\sigma(n := v)) \ n) \rangle$
 $\langle \text{proof} \rangle$

1.3.3 A simple Model of a Local Variable Family (Optional)

definition $\text{local-vars} :: \sigma \Rightarrow SV \Rightarrow \text{evs process}$

where $\langle \text{local-vars} \equiv \mu X. (\lambda \sigma. (\lambda id. ((\text{read}_{\text{local}} \ id)!(\sigma \ id) \rightarrow X \ \sigma \ id) \sqcap ((\text{update}_{\text{local}} \ id)?v \rightarrow X \ (\sigma(id := v)) \ id))) \rangle$

lemma $\text{local-vars-rec} : \langle \text{local-vars } \sigma \ n = ((\text{read}_{\text{local}} \ n)!(\sigma \ n) \rightarrow \text{local-vars } \sigma \ n) \rangle$
 $\sqcap ((\text{update}_{\text{local}} \ n)?v \rightarrow \text{local-vars } (\sigma(n := v)) \ n) \rangle$
 $\langle \text{proof} \rangle$

1.4 Denotational Semantics of IMP_{concur}

The denotational semantics of IMP_{concur} is a straight recursive interpretation in the domain of HOL-CSP processes. Thread local variables were represented in this presentation by updates on a thread-local environment, not by interactions with processes. The recursive definition of the semantic function fits on a post-card:

fun $\text{Sem}_0 :: \text{com} \Rightarrow (\sigma \Rightarrow \text{evs process}) \Rightarrow \sigma \Rightarrow \text{evs process}$

where $\text{Sem}_0 \text{ SKIP } C = C$
 $\text{Sem}_0 (x := E) C = (\lambda \sigma. C (\sigma(x := E \ \sigma)))$
 $\text{Sem}_0 (P ; Q) C = (\text{Sem}_0 P (\text{Sem}_0 Q C))$
 $\text{Sem}_0 (\text{IF } E \text{ THEN } C1 \text{ ELSE } C2) C = (\lambda \sigma. \text{if } E \ \sigma$
 $\quad \text{then } \text{Sem}_0 C1 \ C \ \sigma$
 $\quad \text{else } \text{Sem}_0 C2 \ C \ \sigma)$
 $\text{Sem}_0 (\text{WHILE } E \text{ DO } B) C = (\mu X. (\lambda \sigma. \text{if } E \ \sigma$
 $\quad \text{then } \text{Sem}_0 B \ X \ \sigma$
 $\quad \text{else } C \ \sigma))$

$$\begin{aligned}
|Sem_0 (call F) C &= (\lambda \sigma. C (F \sigma)) \\
|Sem_0 (com.STOP) C &= (\lambda \sigma. Constant-Processes.STOP) \\
|Sem_0 (com.lock n) C &= (\lambda \sigma. evs.lock n \rightarrow C \sigma) \\
|Sem_0 (com.unlock n) C &= (\lambda \sigma. evs.unlock n \rightarrow C \sigma) \\
|Sem_0 (STORE E TO X_{glo}) C &= (\lambda \sigma. upd_{global} X_{glo} (E \sigma) \rightarrow C \sigma) \\
|Sem_0 (LOAD X_{loc} FROM X_{glo}) C &= (\lambda \sigma. (read_{global} X_{glo})?x \\
&\rightarrow C(\sigma(X_{loc}:=x)))
\end{aligned}$$

definition *initial-state* :: σ ($\langle \sigma_0 \rangle$)
where $\langle \sigma_0 \equiv (\lambda -. 0::int) \rangle$

definition *some-state* :: σ ($\langle \sigma_{expl} \rangle$)
where $\langle \sigma_{expl} v \equiv \text{if } v = "a" \text{ then } 3 \text{ else } 0 \rangle$

In case that an **assume**-primitive on initial states has to be modeled (the usual construct for modeling preconditions), this can be done by the Hilbert-Choice operator *SOME* $\sigma. E$ on initial local and global states.

definition *Sem* **where** $\langle Sem Thread \equiv Sem_0 Thread (\lambda -. Skip) \sigma_0 \rangle$
— final continuation: *Skip*, execution starts with initialized local vars for simplicity.

1.5 A Thread-System as HOL-CSPM-Architecture

An example of a global IMP_{concur} -System with 2 threads, 3 global variables and 4 semaphores:

definition *Thread3-sem* $\equiv$ ($||| idx \in \# mset ["a","b","c"] . global-vars \sigma_0 idx$
 $||| idx \in \# mset [1..4] . semaphore idx$)
 $||$
 $(Sem Thread1 ||| Sem Thread2)$
 $\rangle$

$\langle ML \rangle$

Tests:

lemma *Thread2-sem* :
 $Sem(Thread2) = evs.lock 3 \rightarrow (\mu x. (\lambda \sigma::\sigma. evs.lock 4 \rightarrow x \sigma)) \sigma_{expl}$
 $\langle proof \rangle$

schematic-goal *K* : $Thread3-sem = ?X$
 $\langle proof \rangle$

lemma *Thread3-sem :*

$$\begin{aligned}
\langle \text{Thread3-sem} = & ((\text{global-vars } \sigma_0 \text{ ''a''} \parallel (\text{global-vars } \sigma_0 \text{ ''b''} \parallel \text{global-vars } \sigma_0 \\
& \text{''c''})) \\
& \parallel (\text{semaphore } 1 \parallel (\text{semaphore } 2 \parallel (\text{semaphore } 3 \parallel \text{semaphore} \\
& 4)))) \\
& \parallel \\
& ((\text{evs.lock } 3 \rightarrow (\mu x. (\lambda \sigma. \text{if } 3 < \sigma \text{ ''e''} \\
& \text{then evs.lock } 4 \rightarrow x \sigma \\
& \text{else upd}_{\text{global}} \text{ ''b'' } (\sigma \text{ ''d''} + 1) \\
& \rightarrow \text{read}_{\text{global}} \text{ ''b''?} x \\
& \rightarrow \text{evs.unlock } 3 \rightarrow \text{Skip})) \\
& ((\lambda -. 0::\text{int})(\text{''a''} := 3::\text{int}))) \\
& \parallel \\
& (\text{evs.lock } 3 \rightarrow (\mu x. (\lambda \sigma. \text{evs.lock } 4 \rightarrow x \sigma)) ((\lambda -. 0::\text{int})(\text{''a''} := \\
& 3::\text{int})))) \\
& \rangle \\
& \langle \text{proof} \rangle
\end{aligned}$$

$\langle ML \rangle$

1.6 A Semantic Corner-Case of IMP_{concur}

We study the case of three global variables and 4 locks and a computation that spends its time in an infinite loop without communicating. (The result would be the same if inside the loop, only computations on local variables were done.

definition *Thread4 where*

$$\text{Thread4} = \text{WHILE } (\lambda \sigma. \text{True}) \text{ DO SKIP}$$

definition *Thread4-sem where*

$$\begin{aligned}
\text{Thread4-sem} \equiv & (\parallel \text{idx} \in \# \text{ mset } [\text{''a''}, \text{''b''}, \text{''c''}]. \text{ global-vars } \sigma_0 \text{ idx} \\
& \parallel \\
& \parallel \text{idx} \in \# \text{ mset } [1..4]. \text{ semaphore idx }) \\
& \parallel \\
& (\text{Sem Thread4})
\end{aligned}$$

The following theory shows that this type of programs collapses to $\perp$.

theorem *bang :* $\langle \text{Thread4-sem} = \perp \rangle$

$\langle \text{proof} \rangle$

1.7 Another Semantic Corner-Case: Deadlock

For simplicity of the subsequent proof, we model a system without global variables:

definition *Thread5* **where**

$$\text{Thread5} = \text{WHILE } (\lambda\sigma. \text{True}) \text{ DO } (\text{com.lock } 1)$$

definition *Thread5-sem* **where**

$$\text{Thread5-sem} \equiv (\parallel \mid \mid \mid \text{idx} \in \# \text{mset } [1..4]. \text{semaphore idx})$$

$$\parallel$$

$$\text{Sem Thread5}$$

lemma *Thread5-while-rec* : $(\mu x :: 'b \Rightarrow \text{evs process. } (\lambda\sigma. \text{evs.lock } (1::\text{int}) \rightarrow x \sigma))$
 A

$$= \text{evs.lock } 1 \rightarrow (\mu x. (\lambda\sigma. \text{evs.lock } (1::\text{int}) \rightarrow x \sigma)) A$$

<proof>

This process system will deadlock after the first attempt to get *com.lock 0*. This can be formally proven in HOL-CSP by stating:

$$\neg \text{deadlock-free Thread5-sem.}$$

The proof is shown in the following subsection.

1.7.1 A Technique to rewrite Interleaves to MPrefixes

Note that the following description is not necessarily the most effective manner to establish deadlock-freeness via fixed-point induction; it is clear that it will not scale up to large process systems. (For a more powerful and automated approach, we refer to the handling via Proc-Omata [1] or the approach in [2]). Our proceeding has, though, the advantage to use only very basic means and is therefore a self-contained demonstration.

A key obstacle for the establishment of deadlock-properties is that we need to bring both sides of the parallel composition operator - $\parallel$ - into the form *MPrefix A P* in order to apply the rule:

$$\text{MPrefix } ?A \text{ ?P } \parallel \text{MPrefix } ?B \text{ ?Q} = \Box_{x \in (?A \cap ?B)} \rightarrow (?P \ x \parallel ?Q \ x)$$

The *MPrefix*-terms are typically quite large if-then-else cascades representing some form of decision diagram that collapse to small terms after synchronization.

lemma *inter2MPrefix1*:

assumes $a \neq b$

shows $(a \rightarrow P \parallel b \rightarrow Q)$

$$= \text{MPrefix } \{a, b\} (\lambda x. \text{if } x = a \text{ then } (P \parallel b \rightarrow Q))$$

$else (a \rightarrow P) ||| Q$

$\langle proof \rangle$

lemma *inter2Mprefix2*:

assumes $* : a \notin A$
shows $((a \rightarrow P) ||| Mprefix\ A\ Q)$
 $= Mprefix\ (insert\ a\ A)\ (\lambda x. \text{if } x = a \text{ then } (P ||| Mprefix\ A\ Q)$
 $\quad\quad\quad else ((a \rightarrow P) ||| Q\ x))$
 $\langle proof \rangle$

lemma *inter2Mprefix3*:

$a \neq b \implies ((a \rightarrow P\ a) \sqcap (b \rightarrow Q\ b))$
 $= Mprefix\ \{a, b\}\ (\lambda x. \text{if } x = a \text{ then } P\ a \text{ else } Q\ b)$
 $\langle proof \rangle$

After these generalities, we turn to the core lemmas relevant to the subsequent proofs. They construct compact normalforms for states (represented by CSPM-expressions) linked by the events *evs.lock 1* and *evs.unlock 1*. They proceed by blowing both sides up into a lower-level normalform in which their equality can be demonstrated.

lemma *contractMultiInter1* :

$MultiInter\ (mset\ [1..4])\ semaphore\ ||\ evs.lock\ 1 \rightarrow P$
 $= evs.lock\ 1 \rightarrow ((evs.unlock\ 1 \rightarrow semaphore\ 1\ ||| MultiInter\ (mset\ [2..4])$
 $semaphore) || P)$
 $(is\ ?lhs = evs.lock\ 1 \rightarrow ?rhs)$
 $\langle proof \rangle$

lemma *contractMultiInter2*:

$((evs.unlock\ 1 \rightarrow semaphore\ 1\ ||| MultiInter\ (mset\ [2..4])\ semaphore)$
 $||\ evs.lock\ 1 \rightarrow P)$
 $= Constant-Processes.STOP$
 $(is\ (?R || ?S) = -)$
 $\langle proof \rangle$

After these preparatory lemmas, the path to the deadlock is straight-forward: it proceeds by unrolling the while-loop two times; unfolding *semaphore 1* and its fixpoint two times; simplifying to *com.STOP* since *evs.lock 0* and *evs.unlock 0* are forced to be synchronized, propagating this result to the refinement level and reduce it to a contradiction of the refinement statement with the *DF UNIV* process. We skip the formal proof here since it is out of scope of this presentation.

theorem *bullocks*: $\langle \neg \text{deadlock-free Thread5-sem} \rangle$

$\langle proof \rangle$

This proof can be drastically shortened / automatized if we would use a more advanced background theory of HOL-CSP, for example:

term $\langle \text{Initials } Q \subseteq \text{ev} \text{ ' } S$
 $\implies \text{Initials } Q \cap \text{ev} \text{ ' } T = \{\}$
 $\implies ((P \sqcap Q) \llbracket S \rrbracket (Mprefix \ T \ R)) = P \llbracket S \rrbracket (Mprefix \ T \ R) \rangle$

term $\langle \alpha(Q) \subseteq S$
 $\implies \alpha(Q) \cap T = \{\}$
 $\implies \alpha(Q) \cap \alpha(R) = \{\}$
 $\implies ((P \parallel Q) \llbracket S \rrbracket R) = P \llbracket S \rrbracket R \rangle$

term $\langle \alpha(Q) \subseteq S$
 $\implies \alpha(Q) \cap T = \{\}$
 $\implies \text{Initials } Q \cap \alpha((Mprefix \ T \ R)) = \{\}$
 $\implies ((P \parallel Q) \llbracket S \rrbracket (Mprefix \ T \ R)) = P \llbracket S \rrbracket (Mprefix \ T \ R) \rangle$

1.8 Yet another Semantic Corner-Case: A Deadlock-Freeness Proof

We slightly modify the previous example by adding the following thread into the architecture:

definition *Thread6* **where**

Thread6 = *WHILE* ($\lambda\sigma. \text{True}$) *DO* (*com.unlock* 1)

lemma *Thread6-while-rec* :

$(\mu x :: 'b \Rightarrow \text{evs process. } (\lambda\sigma. \text{evs.unlock } (1::\text{int}) \rightarrow x \ \sigma)) \ A$
 $= \text{evs.unlock } 1 \rightarrow (\mu x. (\lambda\sigma. \text{evs.unlock } (1::\text{int}) \rightarrow x \ \sigma)) \ A$

<proof>

definition *Thread6-sem* **where**

Thread6-sem $\equiv (\parallel \text{idx} \in \# \text{mset } [1..4]. \text{semaphore idx})$

$\parallel$

$(\text{Sem } \text{Thread5} \parallel \text{Sem } \text{Thread6})$

The architecture *Thread6*, however, will not deadlock, although each single thread will. This shows that in IMP_{concur} locks can mutually deblock each other and have — together with the global variables — global visibility inside the thread system.

The statement for this claim looks as follows:

term $\langle \text{deadlock-free } \text{Thread6-sem} \rangle$

We will present in this theory a straight-forward proof approach (HOL-CSP has more sophisticated and also more automatic methods than the suggested one). We need a little background theory concerning *deadlock-free*'ness here; in principle, it can be established via a consequence of the fixpoint induction.

A complication to be tackled is that we need a format of this induction that takes two events — instead of just one — to arrive at the anchor of the induction, namely $evs.lock$ and $evs.unlock$.

1.8.1 On Coinduction for deadlock-freeness.

In the following, we derive the necessary deadlock-coinduction:

lemma *lasso-equiv1*: $DF\ UNIV = (\mu\ x.\ \sqcap a \in UNIV \rightarrow \sqcap a \in UNIV \rightarrow x)$
 $\langle proof \rangle$

Here is the necessary versions of *deadlock-free-ness* induction for cases with a lasso-length 1 and 2:

lemma *deadlock-free-1-coinduct* :
 $\langle (\bigwedge x.\ x \sqsubseteq_{FD} P \implies \sqcap a \in UNIV \rightarrow x \sqsubseteq_{FD} P)$
 $\implies deadlock\text{-}free\ (P::('a, 'b)\ process_{tick}) \rangle$
 $\langle proof \rangle$

lemma *deadlock-free-2-coinduct* :
 $\langle (\bigwedge x.\ x \sqsubseteq_{FD} P \implies \sqcap a \in UNIV \rightarrow \sqcap a \in UNIV \rightarrow x \sqsubseteq_{FD} P)$
 $\implies deadlock\text{-}free\ (P::('a, 'b)\ process_{tick}) \rangle$
 $\langle proof \rangle$

And we need two more technical lemmas (analogously to the proofs in the previous session) that enable us to represent the intermediate process states by compact process expressions in HOL-CSPM.

lemma *contractMultiInter1'*:
assumes $\langle evs.unlock\ 1 \rightarrow P2 = P2 \rangle$
shows $\langle MultiInter\ (mset\ [1..4])\ semaphore$
 $\parallel$
 $\sqcap x \in \{evs.lock\ 1,\ evs.unlock\ 1\}$
 $\rightarrow (if\ x = evs.lock\ 1\ then\ P1\ \parallel\ evs.unlock\ 1 \rightarrow P2\ else\ evs.lock\ 1 \rightarrow$
 $P1\ \parallel\ P2) =$
 $evs.lock\ 1 \rightarrow$
 $(\ (evs.unlock\ 1 \rightarrow semaphore\ 1\ \parallel\ MultiInter\ (mset\ [2..4])\ semaphore)$
 $\parallel\ (P1\ \parallel\ P2)) \rangle$
(is $?lhs = evs.lock\ 1 \rightarrow ?rhs)$
 $\langle proof \rangle$

lemma *contractMultiInter1''* :
assumes $evs.lock\ 1 \rightarrow P1 = P1$
shows $\langle$
 $(evs.unlock\ 1 \rightarrow semaphore\ 1\ \parallel\ MultiInter\ (mset\ [2..4])\ semaphore)$
 $\parallel\ \sqcap x \in \{evs.lock\ 1,\ evs.unlock\ 1\}$
 $\rightarrow (if\ x = evs.lock\ 1$
 $then\ P1\ \parallel\ evs.unlock\ 1 \rightarrow P2$
 $else\ evs.lock\ 1 \rightarrow P1\ \parallel\ P2)$
 $\rangle$

$$\begin{aligned}
&= \\
&\quad (evs.unlock\ 1 \rightarrow ((MultiInter\ (mset\ [1..4])\ semaphore) \\
&\quad \quad \parallel (P1\ \parallel\ P2))) \\
&\quad \rangle \\
&\quad (\text{is } (?R \parallel ?S) = evs.unlock\ 1 \rightarrow ?rhs) \\
&\langle proof \rangle
\end{aligned}$$

And now we attack the main theorem:

theorem *bullocks2* : $\langle deadlock\text{-}free\ Thread6\text{-}sem \rangle$
 $\langle proof \rangle$

1.9 A more serious Example: An MPI Program

The example is drawn from Steven F. Siegels paper: “Parameterized Verification of Deterministic MPI Programs” <https://arxiv.org/pdf/2607.18049>, 2026. It addresses the verification of message passing programs (MPI) ¹

The informal spec of the running example reads as follows: The program *cycsum* is a cyclic sum reduction to all processes. Each process stores a int number in *x*, then repeatedly sends to its left, receives from its right, and adds the received value to sum. At the end, each process holds the sum of the original array *A* of values.

A bit more formally, this reads as follows:

```

1 param A;
2 var i, x, sum, left, right;
3 assume isRealArray(A, NP) ∧ x = A[pid];
4 left := (pid + NP - 1) % NP;
5 right := (pid + 1) % NP;
6 i := 1;
7 sum := x;
8 while ( i < NP ) {
9   send x to left;
10  recv x from right;
11  sum := sum + x;
12  x := x + 1;
13 }
14 assert sum = ΣNP-1i=0 A[i];

```

Note that this is a conceptual notation. The full fledged proof in Frama-C involves several transformations and an unverified VCG in the tool-chain; however, it offers automated proofs of the program code annotated by assertions, which comprises 17 pages of dense Frama-C code.

Our modeling in *IMP_{concur}* reads as follows:

¹Message-Passing Interface Forum. 2025. MPI: A Message-Passing Interface Standard, Version 5.0. <https://www.mpi-forum.org/docs/mpi-5.0/mpi50-report.pdf>.

consts $NP :: int$ — Parameter of spec as unspecified constant

consts $array :: \langle string \times int \Rightarrow string \rangle$

— array-name + offset : the only thing we need to know: array is injective.

consts $array_offset :: \langle string \times int \Rightarrow int \rangle$

— the only thing we need to know: $array_offset(array(X,i)) = i$.

term $\langle [1..N] \rangle$

definition $sums$ **where** $\langle sums\ A = fold\ (+)\ A\ (0::int) \rangle$

definition $Thread7$ **where**

```

 $\langle Thread7\ pid\ A = ($ 
   $\text{"left"} := (\lambda\sigma. (pid + NP - 1) \bmod NP)) ;$ 
   $($ 
     $\text{"right"} := (\lambda\sigma. (pid + 1) \bmod NP)) ;$ 
     $($ 
       $\text{"i"} := (\lambda\sigma. 1)) ;$ 
       $com.lock\ pid;$ 
       $LOAD\ \text{"x"}\ FROM\ array(\text{"A"},pid);$ 
       $com.unlock\ pid ;$ 
       $($ 
         $\text{"sum"} := (\lambda\sigma. \sigma\ \text{"x"}) ;$ 
         $WHILE\ (\lambda\sigma. \sigma\ \text{"i"} < NP)\ DO$ 
           $(com.lock\ ((pid - 1 + NP) \bmod NP);$ 
             $STORE\ (\lambda\sigma. \sigma\ \text{"x"})\ TO\ (array(\text{"A"},(pid - 1 + NP) \bmod NP));$ 
             $com.unlock\ ((pid - 1 + NP) \bmod NP) ;$ 
             $com.lock\ ((pid + 1) \bmod NP);$ 
             $LOAD\ \text{"x"}\ FROM\ array(\text{"A"},(pid + 1) \bmod NP);$ 
             $com.unlock\ ((pid + 1) \bmod NP);$ 
             $(\text{"sum"} := (\lambda\sigma. \sigma\ \text{"sum"} + \sigma\ \text{"x"}));$ 
             $(\text{"i"} := (\lambda\sigma. \sigma\ \text{"i"} + 1))$ 
           $);$ 
           $assert\ (\lambda\sigma. \sigma\ \text{"sum"} = sums\ A)$ 
         $\rangle$ 

```

Some remarks: we assume that the *send* and *recv* semantics of the above informal notation implies an implicit lock-unlock mechanism of these global variables. Moreover, the notation is not quite consequent: while MPI programs are explicitly designed for fixed architectures (in our case: a ring), the notation implies access to the local variables which could be modified. In our example *cycsum*, *left* and *right* are constants which have obviously been introduced just to increase readability. The assertion syntax is, as common in PL-annotation languages, a mixture of logical variables and program variables which are syntactically identified: the array *A* is referring to the *content* of the state of the array *A*. This has to be made explicit in the correctness statement.

definition $Thread7\text{-}sem$ **where**

$Thread7\text{-}sem\ A\ A' \equiv$

```

   $($ 
     $|||$   $idx \in \# mset\ [\text{"a"}, \text{"b"}, \text{"c"}].\ global\text{-}vars\ \sigma_0\ idx$ 
     $|||$ 
     $|||$   $idx \in \# mset\ [1..4].\ semaphore\ idx$ 
   $)$ 

```

||

(||| $idx \in \# \text{ mset } [1..NP]. \text{ semaphore } idx$)

... where A' , the array-cell-environment, should be set to its content A :
 $A'(\text{array}("A", idx)) = A!(idx-1)$ for all idx in $[1..NP]$. See remark on the difference between array and its content above.

Again, the functional correctness proof boils down to a deadlock-freeness proof:

theorem *correct*:

assumes $\langle \forall i \in \text{set}[1..NP]. A'(\text{array}("A", i)) = A!(\text{nat}(i-1)) \rangle$
and $\langle \text{length } A = \text{nat } NP \rangle$
shows $\langle \text{deadlock-free } (\text{Thread7-sem } A \ A') \rangle$
 $\langle \text{proof} \rangle$

We consider a formal proof here as out of scope.

1.10 An Architectural Extension: Parameterized Threads

With a tiny bit of tinkering, we can extend the language IMP_{concur} to a language allowing to express dynamic calculations of architectural connections:

type-synonym $LV_{expr} = \langle \sigma \Rightarrow V \rangle$ — Local Variable Expr

type-synonym $SV_{expr} = \langle \sigma \Rightarrow MV \rangle$ — Semaphore Expr

type-synonym $GV_{expr} = \langle \sigma \Rightarrow \text{string} \rangle$ — Global Variable Expr

datatype $com_a = SKIP$

<i>assign</i>	$LV_{expr} \ E_{arith}$	$(- :=_a - [90,90]80)$
<i>seq</i>	$com_a \ com_a$	(infixl ; ; 78)
<i>cond</i>	$E_{bool} \ com_a \ com_a$	$(IF_a (-) / THEN (-) / ELSE (-) / [0,0,79]79)$
<i>while</i>	$E_{bool} \ com_a$	$(WHILE_a (-) / DO (-) [0,80]80)$
<i>call</i>	F	
<i>lock</i>	SV_{expr}	
<i>unlock</i>	SV_{expr}	
<i>send</i>	$E_{arith} \ GV_{expr}$	$(STORE_a - TO - [90,90]80)$
<i>rec</i>	$E_{arith} \ GV_{expr}$	$(LOAD_a - FROM - [90,90]80)$

1.11 Denotational Semantics of IMP_{concur} -P, a parametric version of IMP_{concur}

consts $VarName :: \langle int \Rightarrow string \rangle$ — converts a reference to a global var name

$\mathbf{fun} \text{ Sem}_{a0} :: com_a \Rightarrow (\sigma \Rightarrow \text{evs process}) \Rightarrow \sigma \Rightarrow \text{evs process}$
 $\mathbf{where} \langle \text{Sem}_{a0} \text{ SKIP } C \rangle = C \rangle$
 $\langle \text{Sem}_{a0} (x :=_a E) C \rangle = (\lambda \sigma. C (\sigma((x \sigma) := E \sigma))) \rangle$
 $\langle \text{Sem}_{a0} (P ;; Q) C \rangle = (\text{Sem}_{a0} P (\text{Sem}_{a0} Q C)) \rangle$
 $\langle \text{Sem}_{a0} (IF_a E \text{ THEN } C1 \text{ ELSE } C2) C \rangle = (\lambda \sigma. \text{if } E \sigma$
 $\quad \text{then Sem}_{a0} C1 C \sigma$
 $\quad \text{else Sem}_{a0} C2 C \sigma) \rangle$
 $\langle \text{Sem}_{a0} (\text{WHILE}_a E \text{ DO } B) C \rangle = (\mu X. (\lambda \sigma. \text{if } E \sigma$
 $\quad \text{then Sem}_{a0} B X \sigma$
 $\quad \text{else } C \sigma)) \rangle$
 $\langle \text{Sem}_{a0} (com_a.call F) C \rangle = (\lambda \sigma. C (F \sigma)) \rangle$
 $\langle \text{Sem}_{a0} (com_a.lock n) C \rangle = (\lambda \sigma. \text{evs.lock } (n \sigma) \rightarrow C \sigma) \rangle$
 $\langle \text{Sem}_{a0} (com_a.unlock n) C \rangle = (\lambda \sigma. \text{evs.unlock } (n \sigma) \rightarrow C \sigma) \rangle$
 $\langle \text{Sem}_{a0} (\text{STORE}_a E \text{ TO } X_{glo}) C \rangle = (\lambda \sigma. \text{upd}_{global} (X_{glo} \sigma) (E \sigma) \rightarrow$
 $C \sigma) \rangle$
 $\langle \text{Sem}_{a0} (\text{LOAD}_a X_{loc} \text{ FROM } X_{glo}) C \rangle = (\lambda \sigma. (\text{read}_{global} (X_{glo} \sigma)) ? x$
 $\quad \rightarrow C(\sigma((\text{VarName}(X_{loc} \sigma)) := x))) \rangle$

1.12 Conclusion and Future Work

We have shown the language and semantics of IMP_{concur} , a thin layer over HOL-CSP giving this process-algebra a more programming language flavor. It is conceived to show the close link between both worlds considered quite distinct by many.

The embedding provides:

1. a clear semantic foundation via HOL-CSP to HOL, comprising computation and concurrency,
2. an end-to-end verification of a verification method inside a highly trustable interactive proof assistant,
3. a path to theorem proving of functional properties, via deadlock-freeness proofs,
4. a path to model-checking of functional properties, via deadlock-freeness inside FDR4 (unpublished so far).

An interesting direction for future work is, for example, how a Hoare Calculus for IMP_{concur} would look like, or how a conversion to FDR4 could be used in a SPIN-like manner for finitized programs as model-checking-based backend for practical purposes.

Chapter 2

Appendix: A Syntactic Frontend for IMP_{concur}

```
theory CIL
  imports IMP-concur
  keywords SYSTEM :: thy-decl
    and globals locks LOCK UNLOCK thread WHILE IF ELSE THEN
      end; any actions DO <- -> SKIP DONE

begin
```

2.1 Introduction

This theory defines syntactic support for a Concurrent Imperative Language (CIL) on top of IMP_{concur} , a semantic language with threads, thread-local and thread-global states on which the threads may synchronize via semaphores (locks and unlocks). IMP_{concur} itself is based on HOL-CSPM; the semantics is constructed in a way that a conversion to Proc-Omata is possible.

This appendix has been authored by Mathilde Needham and Zineddine Kermadj on their internship in summer 2025.

2.2 Interface

```
definition SEMAPHORES
  where ⟨SEMAPHORES idx ≡ Renaming (semaphore idx) id id⟩
    — lifting semaphores to the global alphabet

definition GLOBALVARS :: σ ⇒ string ⇒ evs process
  where ⟨GLOBALVARS σinit idx ≡ Renaming (global-vars σinit idx) (λx. x)
    (λy. y)⟩
```

definition *LOCALVARS*

where $\langle LOCALVARS \ \sigma_{init} \ idx \equiv Renaming \ (local-vars \ \sigma_{init} \ idx) (\lambda x. x) (\lambda y. y) \rangle$

2.3 Parsing and Term-Reading

$\langle ML \rangle$

typed decl *pre-event-type*

2.4 Context Checking

$\langle ML \rangle$

2.5 Semantic Check

$\langle ML \rangle$

2.6 Semantic Encoding

definition *SKIP-process* :: *evs process*

where *SKIP-process* $\equiv Skip$

$\langle ML \rangle$

2.7 Impression

$\langle ML \rangle$

2.8 Tests

SYSTEM *S*

globals

locks

thread *empty*:

actions

thread *empty2*:

actions

end;

SYSTEM *System-A*

globals *x*: $\langle int \rangle = \langle 3 :: int \rangle$ *var1*: $\langle int \rangle = \langle 4 :: int \rangle$ *var4*: $\langle int \rangle = \langle 4 :: int \rangle$

locks *l*: $\langle () \rangle$ *l2*: $\langle () \rangle$ *l3*: $\langle () \rangle$

thread *t1* :

```

any  $y : \langle int \rangle = \langle 36 :: int \rangle$   $var2 : \langle int \rangle$ 
actions
SKIP;
LOCK  $l$ ;
 $y \leftarrow var1$ ;
UNLOCK  $l$ ;
IF  $\langle var1 \leq (3 :: int) \rangle$  THEN
  WHILE  $\langle var1 \leq (3 :: int) \rangle$  DO
    LOCK  $l2$ ;
    SKIP;
     $var2 = \langle var2 + 1 :: int \rangle$ ;
     $var1 \rightarrow \langle var2 :: int \rangle$ ;
    UNLOCK  $l2$ ;
  DONE
ELSE
  SKIP;
DONE
 $y = \langle y + 2 :: int \rangle$ ;
 $y = \langle y + 2 :: int \rangle$ ;
 $var1 \rightarrow \langle y :: int \rangle$ ;

thread  $t2$  :
  any  $var2 : \langle int \rangle = \langle (4 + 6) :: int \rangle$   $y : \langle int \rangle = \langle 28 :: int \rangle$ 
  actions
  SKIP;
  LOCK  $l$ ;

  UNLOCK  $l$ ;
  IF  $\langle 3 = (5 :: int) \rangle$  THEN
    WHILE  $\langle x - (8 :: int) < x \rangle$  DO
      LOCK  $l$ ;
      UNLOCK  $l$ ;
    DONE
  ELSE
    SKIP;
  DONE
   $y = \langle var2 + 3 :: int \rangle$ ;

thread  $t3$ :
  any  $test : int = \langle -3 :: int \rangle$   $var-local : int = \langle 4 :: int \rangle$ 
  actions
     $var-local = \langle (4 + 42) :: int \rangle$ ;
     $test = \langle 2 * var-local :: int \rangle$ ;
     $test = \langle test + 3 :: int \rangle$ ;
  IF  $\langle x > (6 :: int) \rangle$ 
  THEN
    WHILE  $\langle x > (2 :: int) \rangle$  DO
       $test = \langle test - 3 :: int \rangle$ ;
    DONE
  ELSE

```

```

      test =  $\langle test + 3 :: int \rangle$ ;
    DONE
  end;

```

```

thm t1-cmd-def empty2-cmd-def
thm System-A-def

```

```

 $\langle ML \rangle$ 

```

```

find-theorems (500)name : CIL

```

```

lemmas X = System-A-def [simplified t1-cmd-def t2-cmd-def t3-cmd-def Sem-def]

```

Bibliography

- [1] B. Ballenghien and B. Wolff. A theory of proc-omata - and proof methods for process architectures. In C. Anutariya and M. M. Bonsangue, editors, *Theoretical Aspects of Computing - ICTAC 2024 - 21st International Colloquium, Bangkok, Thailand, November 25-29, 2024, Proceedings*, volume 15373 of *Lecture Notes in Computer Science*, pages 272–289. Springer, 2024.
- [2] F. Yan, B. Ballenghien, S. Foster, A. Cavalcanti, J. Baxter, and B. Wolff. Automated verification of robot software models with assume-guarantee reasoning in isabelle/hol. In E. Komendantskaya and T. Nipkow, editors, *17th International Conference on Interactive Theorem Proving, ITP 2026, Lisbon, Portugal, July 26-29, 2026*, volume 382 of *LIPICs*, pages 10:1–10:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2026.