

*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>	13
1.7	Another Semantic Corner-Case: Deadlock	13
1.7.1	A Technique to rewrite Interleaves to MPrefixes	14
1.8	Yet another Semantic Corner-Case: A Deadlock-Freeness Proof	18
1.8.1	On Coinduction for deadlock-freeness.	18
1.9	A more serious Example: An MPI Program	24
1.10	An Architectural Extension: Parameterized Threads	26
1.11	Denotational Semantics of <i>IMP_{concur}</i> -P, a parametric version of <i>IMP_{concur}</i>	26
1.12	Conclusion and Future Work	27
2	Appendix: A Syntactic Frontend for <i>IMP_{concur}</i>	29
2.1	Introduction	29
2.2	Interface	29
2.3	Parsing and Term-Reading	30
2.4	Context Checking	32
2.5	Semantic Check	38
2.6	Semantic Encoding	57
2.7	Impression	65
2.8	Tests	74

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
⋗

```

$$\begin{aligned} Thread3 &= SKIP ; \\ &(\text{"var-local"} := (\lambda\sigma. 4 + 42) ; \\ &(\text{"test"} := (\lambda\sigma. 2 * \sigma \text{"var-local"}) ; \\ &(\text{"test"} := (\lambda\sigma. \sigma \text{"test"} + 3) ; \\ &IF (\lambda\sigma. \sigma \text{"x"} > 5) \\ &THEN WHILE (\lambda\sigma. \sigma \text{"x"} > 5) DO \text{"test"} := (\lambda\sigma. \sigma \text{"test"} - 3) \\ &ELSE \text{"test"} := (\lambda\sigma. \sigma \text{"test"} + 3)))) \end{aligned}$$

1.3 Events, States and Semaphores ("Bricks").

definition *GVARs* where $\langle GVARs \equiv \{e. \exists x. \exists y. e = read_{global} \ x \ y\} \cup \{e. \exists x. \exists y. e = upd_{global} \ x \ y\} \rangle$

lemma $L4$ [simp]: $\langle \text{unlock } n \in \text{LOCKS} \rangle$ unfolding LOCKS-def by auto

lemma *L8 [simp]:* $\langle \text{upd}_{global} \ n \ m \in \text{GVARs} \rangle$ **unfolding** *LOCKS-def GVARs-def*
by *auto*

lemma $L9[simp]$: $\langle read_{local} \ n \ m \notin LOCKS \rangle$ **unfolding** $LOCKS-def$ $GVARs-def$
by *auto*
lemma $L10[simp]$: $\langle read_{local} \ n \ m \notin GVARs \rangle$ **unfolding** $LOCKS-def$ $GVARs-def$
by *auto*
lemma $L11[simp]$: $\langle update_{local} \ n \ m \notin LOCKS \rangle$ **unfolding** $LOCKS-def$ $GVARs-def$
by *auto*
lemma $L12[simp]$: $\langle update_{local} \ n \ m \notin GVARs \rangle$ **unfolding** $LOCKS-def$ $GVARs-def$
by *auto*

1.3.1 A simple Model of a Semaphore Family

definition $semaphore :: \langle int \Rightarrow evs \ process \rangle$
where $\langle semaphore \equiv \mu X. (\lambda n. lock \ n \rightarrow unlock \ n \rightarrow X \ n) \rangle$

lemma $sema-rec : \langle semaphore \ n = lock \ n \rightarrow unlock \ n \rightarrow semaphore \ n \rangle$
by $(subst \ cont-process-rec[OF \ semaphore-def[THEN \ meta-eq-to-obj-eq]]) \ simp-all$

1.3.2 A simple Model of a Global Variable Family

definition $global-vars :: \langle \sigma \Rightarrow SV \Rightarrow evs \ process \rangle$
where $\langle global-vars \equiv \mu X. (\lambda \sigma. (\lambda id. ((read_{global} \ id)!(\sigma \ id) \rightarrow X \ \sigma \ id) \sqcap ((upd_{global} \ id)?v \rightarrow X \ (\sigma(id := v)) \ id))) \rangle$

lemma $global-vars-rec : \langle global-vars \ \sigma \ n = ((read_{global} \ n)!(\sigma \ n) \rightarrow global-vars \ \sigma \ n) \rangle$
 $\sqcap ((upd_{global} \ n)?v \rightarrow global-vars \ (\sigma(n := v)) \ n) \rangle$
by $(subst \ cont-process-rec[OF \ global-vars-def[THEN \ meta-eq-to-obj-eq]]) \ simp-all$

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

definition $local-vars :: \sigma \Rightarrow SV \Rightarrow evs \ process$
where $\langle local-vars \equiv \mu X. (\lambda \sigma. (\lambda id. ((read_{local} \ id)!(\sigma \ id) \rightarrow X \ \sigma \ id) \sqcap ((update_{local} \ id)?v \rightarrow X \ (\sigma(id := v)) \ id))) \rangle$

lemma $local-vars-rec : \langle local-vars \ \sigma \ n = ((read_{local} \ n)!(\sigma \ n) \rightarrow local-vars \ \sigma \ n) \rangle$
 $\sqcap ((update_{local} \ n)?v \rightarrow local-vars \ (\sigma(n := v)) \ n) \rangle$
by $(subst \ cont-process-rec[OF \ local-vars-def[THEN \ meta-eq-to-obj-eq]]) \ simp-all$

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 $Sem_0 :: com \Rightarrow (\sigma \Rightarrow evs \ process) \Rightarrow \sigma \Rightarrow evs \ process$

$$\begin{aligned}
\text{where } Sem_0 \text{ SKIP } C &= C \\
Sem_0 (x := E) C &= (\lambda \sigma. C (\sigma(x := E \sigma))) \\
Sem_0 (P ; Q) C &= (Sem_0 P (Sem_0 Q C)) \\
Sem_0 (IF E THEN C1 ELSE C2) C &= (\lambda \sigma. \text{if } E \sigma \\
&\quad \text{then } Sem_0 C1 C \sigma \\
&\quad \text{else } Sem_0 C2 C \sigma) \\
Sem_0 (WHILE E DO B) C &= (\mu X. (\lambda \sigma. \text{if } E \sigma \\
&\quad \text{then } Sem_0 B X \sigma \\
&\quad \text{else } C \sigma)) \\
Sem_0 (call F) C &= (\lambda \sigma. C (F \sigma)) \\
Sem_0 (com.STOP) C &= (\lambda \sigma. Constant-Processes.STOP) \\
Sem_0 (com.lock n) C &= (\lambda \sigma. evs.lock n \rightarrow C \sigma) \\
Sem_0 (com.unlock n) C &= (\lambda \sigma. evs.unlock n \rightarrow C \sigma) \\
Sem_0 (STORE E TO X_{glo}) C &= (\lambda \sigma. upd_{global} X_{glo} (E \sigma) \rightarrow C \sigma) \\
Sem_0 (LOAD X_{loc} FROM X_{glo}) C &= (\lambda \sigma. (read_{global} X_{glo})?x \\
&\quad \rightarrow C(\sigma(X_{loc}:=x)))
\end{aligned}$$

definition *initial-state* :: $\sigma \langle \sigma_0 \rangle$

where $\langle \sigma_0 \equiv (\lambda -. 0::int) \rangle$

definition *some-state* :: $\sigma \langle \sigma_{expl} \rangle$

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** $\langle Sem \text{ Thread} \equiv Sem_0 \text{ Thread } (\lambda -. Skip) \sigma_0 \rangle$

— 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* $\equiv (\parallel \parallel idx \in \# \text{ mset } ["a", "b", "c"]. \text{ global-vars } \sigma_0 \text{ idx}$
 $\parallel \parallel idx \in \# \text{ mset } [1..4]. \text{ semaphore } idx)$
 $\parallel$
 $(Sem \text{ Thread1} \parallel Sem \text{ Thread2})$
 $\rangle$

ML $\langle (* \dots \text{ and this reads at the term-level as follows } : *)$

$val - \$ - \$ temp = Thm.concl-of @\{thm \text{ Thread3-sem-def}\}$

```

    val temp2 = term <||| idx ∈ # mset ["a", "b", "c"]. global-vars σ0 idx
  >

```

Tests:

lemma *Thread2-sem* :

```

  Sem(Thread2) = evs.lock 3 → (μ x. (λσ::σ. evs.lock 4 → x σ)) σexpl
  unfolding Thread2-def Sem-def some-state-def initial-state-def
  by (simp add: fun-upd-def)

```

schematic-goal *K* : *Thread3-sem* = ?*X*

```

  unfolding Thread3-sem-def Sem-def
  unfolding Thread1-def Thread2-def
  apply(rule trans)
  apply (simp add: List.upto.simps cong: HOL.if-cong)
  by (rule refl)

```

lemma *Thread3-sem* :

```

  < Thread3-sem = ((global-vars σ0 "a" ||| (global-vars σ0 "b" ||| global-vars σ0
    "c"))
    ||| (semaphore 1 ||| (semaphore 2 ||| (semaphore 3 ||| semaphore
    4))))
    ||
    ((evs.lock 3 → (μ x. (λσ. if 3 < σ "e"
      then evs.lock 4 → x σ
      else updglobal "b" (σ "d" + 1)
        → readglobal "b"?x
        → evs.unlock 3 → Skip))
      ((λ-. 0::int)("a" := 3::int)))
    |||
    (evs.lock 3 → (μ x. (λσ. evs.lock 4 → x σ)) ((λ-. 0::int)("a" :=
    3::int))))
  >
  unfolding Thread3-sem-def Sem-def Thread1-def Thread2-def initial-state-def
  by (simp add: List.upto.simps cong: HOL.if-cong)

```

ML<

```

val x = HOLogic.dest-eq (HOLogic.dest-Trueprop ( Thm.concl-of @{thm K} )) ;
val y = @{thm Thread3-sem} |> Thm.concl-of
    |> HOLogic.dest-Trueprop
    |> HOLogic.dest-eq
  >

```

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**

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

definition *Thread4-sem* **where**

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

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

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

unfolding *Thread4-sem-def Thread4-def Sem-def*

by (*simp add: List.upto.simps Fixrec.fix-id cong: HOL.if-cong*)

1.7 Another Semantic Corner-Case: Deadlock

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

definition *Thread5* **where**

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

definition *Thread5-sem* **where**

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

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

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

by(*subst cont-process-rec[where $P = \langle (\mu x. (\lambda\sigma. \text{ evs.lock } (1::int) \rightarrow x \sigma)) \rangle$, OF refl]*) *simp-all*

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

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

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 $- || -$ into the form $Mprefix\ A\ P$ in order to apply the rule:

$$Mprefix\ ?A\ ?P\ ||\ Mprefix\ ?B\ ?Q = \Box_{x \in (?A \cap ?B)} \rightarrow (?P\ x\ ||\ ?Q\ x)$$

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 $(a \rightarrow P\ ||\ b \rightarrow Q)$
 $= Mprefix\ \{a, b\}\ (\lambda x. \text{if } x = a \text{ then } (P\ ||\ b \rightarrow Q)$
 $\qquad\qquad\qquad \text{else } (a \rightarrow P)\ ||\ Q)$
apply(*simp add: write0-Inter-write0*)
by(*subst (3) write0-def, subst write0-Det-Mprefix, simp add: assms*)

lemma *inter2Mprefix2:*

assumes $* : a \notin A$
shows $((a \rightarrow P)\ ||\ Mprefix\ A\ Q)$
 $= Mprefix\ (insert\ a\ A)\ (\lambda x. \text{if } x = a \text{ then } (P\ ||\ Mprefix\ A\ Q)$
 $\qquad\qquad\qquad \text{else } ((a \rightarrow P)\ ||\ Q\ x))$
apply(*subst write0-def, simp add: Mprefix-Inter-Mprefix*)
apply(*subst Mprefix-Det-Mprefix, simp add: assms*)
by(*fold write0-def, simp*)

lemma *inter2Mprefix3:*

$a \neq b \implies ((a \rightarrow P\ a) \Box (b \rightarrow Q\ b))$
 $= Mprefix\ \{a, b\}\ (\lambda x. \text{if } x = a \text{ then } P\ a \text{ else } Q\ b)$
apply(*simp add: Mprefix-singl[symmetric]*)
by (*smt (verit, ccfv-threshold) Mprefix-Un-distrib Mprefix-singl*
 $Un-insert-right\ insert-commute\ sup-bot.right-neutral$)

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 → P
 $=$ *evs.lock 1 → ((evs.unlock 1 → semaphore 1 ||| MultiInter (mset [2..4]) semaphore) || P)*

(**is** ?lhs = *evs.lock 1 → ?rhs*)

proof –

have * : ?lhs = *evs.lock 1 →*
 $((\text{evs.unlock } 1 \rightarrow \text{semaphore } 1$
 $||| \Box x \in \{\text{evs.lock } 2, \text{evs.lock } 3, \text{evs.lock } 4\}$
 $\rightarrow (\text{if } x = \text{evs.lock } 2$
 $\text{then evs.unlock } 2 \rightarrow \text{semaphore } 2$
 $|||$
 $\Box x \in \{\text{evs.lock } 3, \text{evs.lock } 4\}$
 $\rightarrow (\text{if } x = \text{evs.lock } 3$
 $\text{then evs.unlock } 3 \rightarrow \text{semaphore } 3 |||$
 $\text{semaphore } 4$
 $\text{else semaphore } 3 ||| \text{evs.unlock } 4 \rightarrow$
 $\text{semaphore } 4)$
 $\text{else semaphore } 2$
 $|||$
 $(\text{if } x = \text{evs.lock } 3$
 $\text{then evs.unlock } 3 \rightarrow \text{semaphore } 3 ||| \text{semaphore } 4$
 $\text{else semaphore } 3 ||| \text{evs.unlock } 4 \rightarrow \text{semaphore}$
 $4))) || P)$
(**is** - = *evs.lock 1 → ?rhs'*)
apply(*rule trans*)
apply(*simp add: List.upto.simps*)
apply(*subst sema-rec[where n=⟨1⟩,*
 $\text{subst sema-rec[where n=⟨2⟩},$
 $\text{subst sema-rec[where n=⟨3⟩},$
 $\text{subst sema-rec[where n=⟨4⟩}]$)
apply(*rule trans*)
apply(*simp add: inter2Mprefix1 inter2Mprefix2*)
apply(*simp add: sema-rec[where n=⟨1⟩,symmetric]*
 $\text{sema-rec[where n=⟨2⟩,symmetric]$
 $\text{sema-rec[where n=⟨3⟩,symmetric]$
 $\text{sema-rec[where n=⟨4⟩,symmetric]$
cong: HOL.if-cong)
apply(*subst write0-def[THEN meta-eq-to-obj-eq, of evs.lock (1::int)]*)
by(*subst Mprefix-Par-Mprefix, simp add: Mprefix-singl*) — et bang!
define *X* **where** *X* = ?rhs'
have ** : ?rhs = *X*
apply(*subst write0-def[THEN meta-eq-to-obj-eq]*)
apply(*simp add: List.upto.simps*)
apply(*subst sema-rec[where n=⟨1⟩,subst sema-rec[where n=⟨2⟩,*
 $\text{subst sema-rec[where n=⟨3⟩},\text{subst sema-rec[where n=⟨4⟩}]$)

```

    apply(simp add: inter2Mprefix1 inter2Mprefix2)
    by (metis X-def sema-rec Mprefix-singl)
  show ?thesis
    by (metis * ** X-def)
qed

lemma contractMultiInter2:
  ((evs.unlock 1 → semaphore 1 ||| MultiInter (mset [2..4]) semaphore)
   || evs.lock 1 → P)
  = Constant-Processes.STOP
  (is (?R || ?S) = -)
proof -
  have 1 : ?R =  $\Box x \in \{ \text{evs.unlock } 1, \text{evs.lock } 2, \text{evs.lock } 3, \text{evs.lock } 4 \}$ 
    → (if x = evs.unlock 1
      then semaphore 1
        |||  $\Box x \in \{ \text{evs.lock } 2, \text{evs.lock } 3, \text{evs.lock } 4 \}$ 
          → (if x = evs.lock 2
            then evs.unlock 2 → semaphore 2
              |||  $\Box x \in \{ \text{evs.lock } 3, \text{evs.lock } 4 \}$ 
                → (if x = evs.lock 3
                  then evs.unlock 3 → semaphore 3 ||| semaphore 4
                    else semaphore 3 ||| evs.unlock 4 → semaphore 4)
                else semaphore 2
                  ||| (if x = evs.lock 3
                    then evs.unlock 3 → semaphore 3 ||| semaphore 4
                      else semaphore 3 ||| evs.unlock 4 → semaphore 4))
            else evs.unlock 1 → semaphore 1
              ||| (if x = evs.lock 2
                then evs.unlock 2 → semaphore 2
                  |||  $\Box x \in \{ \text{evs.lock } 3, \text{evs.lock } 4 \}$ 
                    → (if x = evs.lock 3 then evs.unlock 3 → semaphore 3 |||
                      semaphore 4
                    else semaphore 3 ||| evs.unlock 4 → semaphore 4)
                  else semaphore 2
                    ||| (if x = evs.lock 3
                      then evs.unlock 3 → semaphore 3 ||| semaphore 4
                        else semaphore 3 ||| evs.unlock 4 → semaphore 4)))
          else semaphore 3 ||| evs.unlock 4 → semaphore 4)
        else semaphore 2
          ||| (if x = evs.lock 3
            then evs.unlock 3 → semaphore 3 ||| semaphore 4
              else semaphore 3 ||| evs.unlock 4 → semaphore 4)))
    apply(simp add: List.upto.simps)
    apply(subst sema-rec[where n=⟨1⟩],subst sema-rec[where n=⟨2⟩],
      subst sema-rec[where n=⟨3⟩],subst sema-rec[where n=⟨4⟩])
    apply(simp add: inter2Mprefix1 inter2Mprefix2)
    by(simp add: sema-rec[where n=⟨1⟩,symmetric]
      sema-rec[where n=⟨2⟩,symmetric]
      sema-rec[where n=⟨3⟩,symmetric]
      sema-rec[where n=⟨4⟩,symmetric]
      cong: HOL.if-cong)
  show ?thesis
    apply(subst(2) write0-def[THEN meta-eq-to-obj-eq])

```

```

    apply(subst 1)
    by (simp add: Mprefix-Par-Mprefix)
qed

```

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:  $\langle \neg \text{deadlock-free Thread5-sem} \rangle$ 
  unfolding Thread5-sem-def Sem-def Thread5-def
proof –
  show  $\langle \neg \text{deadlock-free}$ 
    ( MultiInter (mset [1..4]) semaphore
      || Sem0 (WHILE  $\lambda\sigma$ . True DO com.lock 1) ( $\lambda$ -. Skip)  $\sigma_0$ )  $\rangle$ 
    (is  $\langle \neg \text{deadlock-free } (?SEMA \parallel ?P) \rangle$ )
  proof –
    have 1 :  $\langle ?P = \text{evs.lock 1} \rightarrow ?P \rangle$  using Thread5-while-rec by auto
    have 2 :  $\langle (?SEMA \parallel \text{evs.lock 1} \rightarrow ?P)$ 
      =  $\text{evs.lock 1} \rightarrow ((\text{evs.unlock 1} \rightarrow \text{semaphore 1}$ 
        || MultiInter (mset [2..4]) semaphore) ||  $?P) \rangle$ 
      (is  $?lhs = \text{evs.lock 1} \rightarrow ?R'$ )
    by (metis contractMultiInter1)
    have 3 :  $\langle \neg \text{deadlock-free Thread5-sem} \rangle = \langle \neg \text{deadlock-free } ?R' \rangle$ 
    by (metis 1 2 Sem-def Thread5-def Thread5-sem-def deadlock-free-write0-iff)
    have 4 :  $\langle ?R' = \text{Constant-Processes.STOP} \rangle$ 
    by (metis 1 contractMultiInter2)
    show ?thesis
    using 3 4 Sem-def Thread5-def Thread5-sem-def non-deadlock-free-STOP by
    auto
  qed
qed

```

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

```

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

```

```

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  $\langle \alpha(Q) \subseteq S$ 
   $\implies \alpha(Q) \cap T = \{\}$ 

```

$$\begin{aligned} &\Rightarrow \text{Initials } Q \cap \alpha((M\text{prefix } T \ R)) = \{\} \\ &\Rightarrow ((P \parallel Q) \llbracket S \rrbracket (M\text{prefix } T \ R)) = P \llbracket S \rrbracket (M\text{prefix } T \ R) \end{aligned}$$

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**

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

lemma *Thread6-while-rec* :

$$\begin{aligned} &(\mu x::'b \Rightarrow \text{evs process. } (\lambda\sigma. \text{evs.unlock } (1::\text{int}) \rightarrow x \ \sigma)) \ A \\ &= \text{evs.unlock } 1 \rightarrow (\mu x. (\lambda\sigma. \text{evs.unlock } (1::\text{int}) \rightarrow x \ \sigma)) \ A \\ &\text{by}(\text{subst cont-process-rec}[\text{where } P = \langle (\mu x. (\lambda\sigma. \text{evs.unlock } (1::\text{int}) \rightarrow x \ \sigma)) \rangle, \\ &\quad \text{OF refl}] \text{ simp-all} \end{aligned}$$

definition *Thread6-sem* **where**

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

$$\parallel$$

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

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

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*: $DF \ UNIV = (\mu x. \sqcap a \in UNIV \rightarrow \sqcap a \in UNIV \rightarrow x)$

```

unfolding DF-def
proof(rule FD-antisym)
  have * :  $\langle X = (X \wedge X) \rangle$  for X by simp
  show  $\langle (\mu x. \sqcap a \in UNIV \rightarrow x) \sqsubseteq_{FD} (\mu x. \sqcap a \in UNIV \rightarrow \sqcap a \in UNIV \rightarrow x) \rangle$ 
    apply(subst *)
    apply(subst cont-process-rec[where P =  $\langle (\mu x. \sqcap a \in UNIV \rightarrow x) \rangle$ 
      and f =  $\langle \lambda x. \sqcap a \in UNIV \rightarrow x \rangle$ ], simp-all)
    apply(rule fix-ind[where F =  $\langle (\lambda x. \sqcap a \in UNIV \rightarrow x) \rangle$ ], simp, simp)
    apply(subst cont-process-rec[where P =  $\langle (\mu x. \sqcap a \in UNIV \rightarrow \sqcap a \in UNIV \rightarrow$ 
 $x) \rangle$ 
      and f =  $\langle \lambda x. \sqcap a \in UNIV \rightarrow \sqcap a \in UNIV \rightarrow x \rangle$ ], simp-all)
    apply(rule mono-Mndetprefix-FD, simp)
    apply(subst cont-process-rec[where P =  $\langle (\mu x. \sqcap a \in UNIV \rightarrow \sqcap a \in UNIV \rightarrow$ 
 $x) \rangle$ 
      and f =  $\langle \lambda x. \sqcap a \in UNIV \rightarrow \sqcap a \in UNIV \rightarrow x \rangle$ ],
      simp-all)
    by (simp add: mono-Mndetprefix-FD)
next
  show  $\langle (\mu x. \sqcap a \in UNIV \rightarrow \sqcap a \in UNIV \rightarrow x) \sqsubseteq_{FD} (\mu x. \sqcap a \in UNIV \rightarrow x) \rangle$ 
    apply(rule fix-ind[where F =  $\langle (\lambda x. \sqcap a \in UNIV \rightarrow \sqcap a \in UNIV \rightarrow x) \rangle$ ],
    apply (simp-all)
    apply(subst cont-process-rec[where P =  $\langle (\mu x. \sqcap a \in UNIV \rightarrow x) \rangle$ 
      and f =  $\langle \lambda x. \sqcap a \in UNIV \rightarrow x \rangle$ ], simp-all)
    apply(subst cont-process-rec[where P =  $\langle (\mu x. \sqcap a \in UNIV \rightarrow x) \rangle$ 
      and f =  $\langle \lambda x. \sqcap a \in UNIV \rightarrow x \rangle$ ], simp-all)
    apply(rule mono-Mndetprefix-FD, simp)
    by(rule mono-Mndetprefix-FD, simp)
qed

```

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

```

lemma deadlock-free-1-coinduct :
   $\langle (\bigwedge x. x \sqsubseteq_{FD} P \implies \sqcap a \in UNIV \rightarrow x \sqsubseteq_{FD} P) \implies \text{deadlock-free } (P::('a, 'b) \text{ process}_{ptick}) \rangle$ 
  apply(rule DF-Univ-freeness[of UNIV], simp, unfold DF-def)
  by(rule fix-ind)(simp-all)

```

```

lemma deadlock-free-2-coinduct :
   $\langle (\bigwedge x. x \sqsubseteq_{FD} P \implies \sqcap a \in UNIV \rightarrow \sqcap a \in UNIV \rightarrow x \sqsubseteq_{FD} P) \implies \text{deadlock-free } (P::('a, 'b) \text{ process}_{ptick}) \rangle$ 
  apply(rule DF-Univ-freeness[of UNIV], simp-all add: lasso-equiv1)
  by(rule fix-ind)(simp-all)

```

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  $\langle \text{evs.unlock } 1 \rightarrow P2 = P2 \rangle$ 
  shows  $\langle \text{MultiInter } (\text{mset } [1..4]) \text{ semaphore} \rangle$ 

```

```

||
□x∈{evs.lock 1, evs.unlock 1}
  → (if x = evs.lock 1 then P1 ||| evs.unlock 1 → P2 else evs.lock 1 →
P1 ||| P2) =
  evs.lock 1 →
  ( (evs.unlock 1 → semaphore 1 ||| MultiInter (mset [2..4]) semaphore)
    || (P1 ||| P2))
(is ?lhs = evs.lock 1 → ?rhs)
proof -
  have * : ?lhs = evs.lock 1 →
    ((evs.unlock 1 → semaphore 1
      ||| □x∈{evs.lock 2, evs.lock 3, evs.lock 4}
        → (if x = evs.lock 2
            then evs.unlock 2 → semaphore 2
            |||
              □x∈{evs.lock 3, evs.lock 4}
                → (if x = evs.lock 3
                    then evs.unlock 3 → semaphore 3 |||
semaphore 4
                    else semaphore 3 ||| evs.unlock 4 →
semaphore 4)
                else semaphore 2
            |||
              (if x = evs.lock 3
                then evs.unlock 3 → semaphore 3 ||| semaphore 4
                else semaphore 3 ||| evs.unlock 4 → semaphore 4)))
      || (P1 ||| evs.unlock 1 → P2))
    (is - = evs.lock 1 → ?rhs')
  apply(rule trans)
  apply(simp add: List.upto.simps)
  apply(subst sema-rec[where n=⟨1⟩],subst sema-rec[where n=⟨2⟩],
    subst sema-rec[where n=⟨3⟩],subst sema-rec[where n=⟨4⟩])
  apply(rule trans)
  apply(simp add: inter2Mprefix1 inter2Mprefix2)
  apply(simp add: sema-rec[where n=⟨1⟩,symmetric]
    sema-rec[where n=⟨2⟩,symmetric]
    sema-rec[where n=⟨3⟩,symmetric]
    sema-rec[where n=⟨4⟩,symmetric]
    cong: HOL.if-cong)
  apply(subst write0-def[THEN meta-eq-to-obj-eq, of evs.lock (1::int)])
  by(subst Mprefix-Par-Mprefix, simp add: Mprefix-singl)
define X where X = ?rhs'
have ** : ?rhs = X
  apply(subst write0-def[THEN meta-eq-to-obj-eq])
  apply(simp add: List.upto.simps)
  apply(subst sema-rec[where n=⟨1⟩],subst sema-rec[where n=⟨2⟩],
    subst sema-rec[where n=⟨3⟩],subst sema-rec[where n=⟨4⟩])
  apply(simp add: inter2Mprefix1 inter2Mprefix2)
  apply(simp only: X-def)

```

1.8. YET ANOTHER SEMANTIC CORNER-CASE: A DEADLOCK-FREENESS PROOF 21

```

    by (metis sema-rec assms Mprefix-singl)
  show ?thesis
    by (metis * ** X-def)
qed

```

```

lemma contractMultiInter1'' :
  assumes evs.lock 1  $\rightarrow$  P1 = P1
  shows <
    (evs.unlock 1  $\rightarrow$  semaphore 1 ||| MultiInter (mset [2..4]) semaphore)
    ||  $\Box x \in \{ \text{evs.lock } 1, \text{evs.unlock } 1 \}$ 
     $\rightarrow$  (if x = evs.lock 1
        then P1 ||| evs.unlock 1  $\rightarrow$  P2
        else evs.lock 1  $\rightarrow$  P1 ||| P2)
  =
    (evs.unlock 1  $\rightarrow$  ((MultiInter (mset [1..4]) semaphore)
      || (P1 ||| P2)))
  >
  (is (?R || ?S) = evs.unlock 1  $\rightarrow$  ?rhs)
proof -
  have 1 : ?R =  $\Box x \in \{ \text{evs.unlock } 1, \text{evs.lock } 2, \text{evs.lock } 3, \text{evs.lock } 4 \}$ 
     $\rightarrow$  (if x = evs.unlock 1
      then semaphore 1
        |||  $\Box x \in \{ \text{evs.lock } 2, \text{evs.lock } 3, \text{evs.lock } 4 \}$ 
         $\rightarrow$  (if x = evs.lock 2
          then evs.unlock 2  $\rightarrow$  semaphore 2
            |||  $\Box x \in \{ \text{evs.lock } 3, \text{evs.lock } 4 \}$ 
             $\rightarrow$  (if x = evs.lock 3
              then evs.unlock 3  $\rightarrow$  semaphore 3 ||| semaphore 4
            else semaphore 3 ||| evs.unlock 4  $\rightarrow$  semaphore 4)
          else semaphore 2
            ||| (if x = evs.lock 3
              then evs.unlock 3  $\rightarrow$  semaphore 3 ||| semaphore 4
              else semaphore 3 ||| evs.unlock 4  $\rightarrow$  semaphore 4))
        else evs.unlock 1  $\rightarrow$  semaphore 1
          ||| (if x = evs.lock 2
            then evs.unlock 2  $\rightarrow$  semaphore 2
              |||  $\Box x \in \{ \text{evs.lock } 3, \text{evs.lock } 4 \}$ 
               $\rightarrow$  (if x = evs.lock 3
                then evs.unlock 3  $\rightarrow$  semaphore 3 ||| semaphore 4
                else semaphore 3 ||| evs.unlock 4  $\rightarrow$  semaphore 4)
            else semaphore 2
              ||| (if x = evs.lock 3
                then evs.unlock 3  $\rightarrow$  semaphore 3 ||| semaphore 4
                else semaphore 3 ||| evs.unlock 4  $\rightarrow$  semaphore 4)))
    apply (simp add: List.upto.simps)
    apply (subst sema-rec[where n=⟨1⟩], subst sema-rec[where n=⟨2⟩],

```

```

      subst sema-rec[where  $n=\langle 3 \rangle$ ],subst sema-rec[where  $n=\langle 4 \rangle$ ])
apply(simp add: inter2Mprefix1 inter2Mprefix2)
by(simp add: sema-rec[where  $n=\langle 1 \rangle$ ,symmetric]
      sema-rec[where  $n=\langle 2 \rangle$ ,symmetric]
      sema-rec[where  $n=\langle 3 \rangle$ ,symmetric]
      sema-rec[where  $n=\langle 4 \rangle$ ,symmetric]
      cong: HOL.if-cong)
have 2:  $evs.unlock\ 1 \rightarrow ?rhs = \Box x \in \{evs.lock\ 1\} \rightarrow ?rhs$  by (metis write0-def)
have 3:  $?rhs = \Box x \in \{evs.lock\ 1, evs.lock\ 2, evs.lock\ 3, evs.lock\ 4\}$ 
       $\rightarrow$  (if  $x = evs.lock\ 1$ 
        then  $evs.unlock\ 1 \rightarrow semaphore\ 1$ 
        |||  $\Box x \in \{evs.lock\ 2, evs.lock\ 3, evs.lock\ 4\}$ 
         $\rightarrow$  (if  $x = evs.lock\ 2$ 
          then  $evs.unlock\ 2 \rightarrow semaphore\ 2$ 
          |||  $\Box x \in \{evs.lock\ 3, evs.lock\ 4\}$ 
           $\rightarrow$  (if  $x = evs.lock\ 3$ 
            then  $evs.unlock\ 3 \rightarrow semaphore\ 3$  |||
            semaphore 4
            else semaphore 3 |||  $evs.unlock\ 4 \rightarrow$ 
            semaphore 4)
          else semaphore 2
          ||| (if  $x = evs.lock\ 3$ 
            then  $evs.unlock\ 3 \rightarrow semaphore\ 3$  ||| semaphore
            4
            else semaphore 3 |||  $evs.unlock\ 4 \rightarrow semaphore$ 
            4))
        else semaphore 1
        ||| (if  $x = evs.lock\ 2$ 
          then  $evs.unlock\ 2 \rightarrow semaphore\ 2$ 
          |||  $\Box x \in \{evs.lock\ 3, evs.lock\ 4\}$ 
           $\rightarrow$  (if  $x = evs.lock\ 3$ 
            then  $evs.unlock\ 3 \rightarrow semaphore\ 3$  ||| semaphore 4
            else semaphore 3 |||  $evs.unlock\ 4 \rightarrow semaphore\ 4$ )
          else semaphore 2
          ||| (if  $x = evs.lock\ 3$ 
            then  $evs.unlock\ 3 \rightarrow semaphore\ 3$  ||| semaphore 4
            else semaphore 3 |||  $evs.unlock\ 4 \rightarrow semaphore\ 4$ )))
        || (P1 ||| P2)
apply(simp add: List.upto.simps)
apply(subst sema-rec[where  $n=\langle 1 \rangle$ ],subst sema-rec[where  $n=\langle 2 \rangle$ ],
      subst sema-rec[where  $n=\langle 3 \rangle$ ],subst sema-rec[where  $n=\langle 4 \rangle$ ])
apply(simp add: inter2Mprefix1 inter2Mprefix2)
by(simp add: sema-rec[where  $n=\langle 1 \rangle$ ,symmetric]
      sema-rec[where  $n=\langle 2 \rangle$ ,symmetric]
      sema-rec[where  $n=\langle 3 \rangle$ ,symmetric]
      sema-rec[where  $n=\langle 4 \rangle$ ,symmetric]
      cong: HOL.if-cong)
show ?thesis
apply(subst 1, subst 2)

```

1.8. YET ANOTHER SEMANTIC CORNER-CASE: A DEADLOCK-FREENESS PROOF 23

```

apply(subst Mprefix-Par-Mprefix, simp-all)
apply(simp-all add: Mprefix-singl cong: HOL.if-cong)
apply(subst sema-rec[where n=⟨1⟩])
apply(subst inter2Mprefix2, simp)
apply(subst inter2Mprefix2, simp)
apply(subst assms)
apply(subst sema-rec[where n=⟨1⟩, symmetric])
apply(subst 3)
by (metis (no-types, lifting) empty-iff evs.distinct(1) insert-iff
      inter2Mprefix2 mono-Mprefix-eq)
qed

```

And now we attack the main theorem:

```

theorem bullocks2 : ⟨deadlock-free Thread6-sem⟩
  unfolding Thread6-sem-def Sem-def Thread5-def Thread6-def
proof –
  show ⟨deadlock-free
    ( MultiInter (mset [1..4]) semaphore
      || ( Sem0 (WHILE λσ. True DO com.lock 1) (λ-. Skip) σ0
          ||| Sem0 (WHILE λσ. True DO com.unlock 1) (λ-. Skip) σ0)))
    (is ⟨deadlock-free (?SEMA || (?P1 ||| ?P2))⟩)
  proof –
    have 1 : ⟨?P1 = evs.lock 1 → ?P1⟩ using Thread5-while-rec by auto
    have 2 : ⟨?P2 = evs.unlock 1 → ?P2⟩ using Thread6-while-rec by auto
    have 3 : ⟨?P1 ||| ?P2 = □x∈{evs.lock 1, evs.unlock 1}
      → (if x = evs.lock 1 then ?P1 ||| evs.unlock 1 → ?P2
          else evs.lock 1 → ?P1 ||| ?P2)⟩
      apply(subst 1, subst 2) by(subst inter2Mprefix1, simp-all)
    have 4 : ⟨(?SEMA || ...)
      = evs.lock 1 → ((evs.unlock 1 → semaphore 1 ||| MultiInter (mset
[2..4]) semaphore)
      || (?P1 ||| ?P2))⟩
      (is ⟨- = evs.lock 1 → ?P3⟩)
      using 2 contractMultiInter1' by auto
    have 5 : ⟨(evs.unlock 1 → semaphore 1 ||| MultiInter (mset [2..4]) semaphore)
|| (?P1|||?P2)
      =
      (evs.unlock 1 → ((MultiInter (mset [1..4]) semaphore) || (?P1|||?P2)))⟩
      using 1 3 contractMultiInter1'' by fastforce
    show ?thesis
    apply(rule deadlock-free-2-coinduct)
    apply(subst 3, subst 4)
    apply(rule Mndetprefix-FD-write0[of ⟨evs.lock 1⟩, THEN trans-FD], simp)
    apply(rule mono-write0-FD)
    apply(subst 5)
    by (metis Mndetprefix-FD UNIV-I mono-write0-FD)
qed
qed

```

1.9 A more serious Example: An MPI Program

The example is drawn from Steven F. Siegel's 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 an integer 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:

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

consts `array` :: `⟨string × int ⇒ string⟩`

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

consts `array_offset` :: `⟨string × int ⇒ int⟩`

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

term `[1..N]`

definition `sums` **where** `⟨sums A = fold (+) A (0::int)⟩`

definition `Thread7` **where**

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>.

```

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

```

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-sem where*

Thread7-sem A A' $\equiv$

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

... where *A'*, the array-cell-environment, should be set to its content *A*: $A'(array("A",idx)) = A!(idx-1)$ 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 $\langle \forall i \in \text{set}[1..NP]. A'(\text{array}("A", idx)) = A!(\text{nat}(idx-1)) \rangle$
and $\langle \text{length } A = \text{nat } NP \rangle$
shows $\langle \text{deadlock-free } (\text{Thread7-sem } A \ A') \rangle$
unfolding $\text{deadlock-free-def Thread7-sem-def Sem-def Thread7-def}$
oops

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

fun $Sem_{a0} :: com_a \Rightarrow (\sigma \Rightarrow \text{evs process}) \Rightarrow \sigma \Rightarrow \text{evs process}$
where $\langle Sem_{a0} \ SKIP \ C = C \rangle$
 $\langle Sem_{a0} \ (x :=_a E) \ C = (\lambda \sigma. C \ (\sigma((x \ \sigma) := E \ \sigma))) \rangle$
 $\langle Sem_{a0} \ (P ;; Q) \ C = (Sem_{a0} \ P \ (Sem_{a0} \ Q \ C)) \rangle$
 $\langle Sem_{a0} \ (IF_a \ E \ THEN \ C1 \ ELSE \ C2) \ C = (\lambda \sigma. \text{if } E \ \sigma$
 $\quad \text{then } Sem_{a0} \ C1 \ C \ \sigma$
 $\quad \text{else } Sem_{a0} \ C2 \ C \ \sigma) \rangle$
 $\langle Sem_{a0} \ (WHILE_a \ E \ DO \ B) \ C = (\mu X. (\lambda \sigma. \text{if } E \ \sigma$
 $\quad \text{then } Sem_{a0} \ B \ X \ \sigma$
 $\quad \text{else } C \ \sigma)) \rangle$
 $\langle Sem_{a0} \ (com_a.call \ F) \ C = (\lambda \sigma. C \ (F \ \sigma)) \rangle$

$$\begin{aligned}
| \langle Sem_{a0} (com_a.lock\ n)\ C &= (\lambda\ \sigma. evs.lock\ (n\ \sigma) \rightarrow C\ \sigma) \rangle \\
| \langle Sem_{a0} (com_a.unlock\ n)\ C &= (\lambda\ \sigma. evs.unlock\ (n\ \sigma) \rightarrow C\ \sigma) \rangle \\
| \langle Sem_{a0} (STORE_a\ E\ TO\ X_{glo})\ C &= (\lambda\ \sigma. upd_{global}\ (X_{glo}\ \sigma)\ (E\ \sigma) \rightarrow \\
C\ \sigma) \rangle & \\
| \langle Sem_{a0} (LOAD_a\ X_{loc}\ FROM\ X_{glo})\ C &= (\lambda\ \sigma. (read_{global}\ (X_{glo}\ \sigma))?\ x \\
&\rightarrow C(\sigma((VarName(X_{loc}\ \sigma)):=x))) \rangle
\end{aligned}$$

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

2.3 Parsing and Term-Reading

ML

val quiet-mode = false

fun message quiet-mode s = if quiet-mode then () else writeln s;

datatype ('a, 'b) sum = inl of 'a | inr of 'b;

exception Undeclared-Variable of string

local open HOLogic in

*fun map-opt f - (SOME a) = f a
| map-opt - a NONE = a*

fun map-option f = map-opt (SOME o f) NONE

(remove this *)*

*fun map-option f (SOME a) = SOME(f a)
| map-option - NONE = NONE*

*val parse-var-decl = (Parse.binding --| Parse.\$\$\$: -- (Parse.position (Parse.term)))
-- (Scan.option (Parse.\$\$\$ = |-- Parse.term))
: ((binding*(string*Position.T))*string option) parser*

*(*declarations de locks*)*

*val parse-locks = Parse.binding --| Parse.\$\$\$: -- (Parse.position (Parse.term))
: (binding*(string*Position.T)) parser*

datatype a = Skip

*| Assign of binding*string
| Unlock of binding
| Lock of binding
| Send of binding*string
| Receive of binding*binding
| Ifelse of (string*(a list))*(a list)
| While of string*(a list)*

*val parse-lock = **keyword** $\langle LOCK \rangle$ |-- Parse.binding --| Parse.\$\$\$;*

```

    >> (fn (x) => Lock(x))
    : a parser
val parse-unlock = keyword⟨UNLOCK⟩ |-- Parse.binding --| Parse.$$$ ;
    >> (fn (x) => Unlock(x))
    : a parser
val parse-assign = Parse.binding --| Parse.$$$ = -- Parse.term --| Parse.$$$
;
    >> (fn (x) => Assign(x))
    : a parser
val parse-send = Parse.binding --| keyword⟨->⟩ -- Parse.term --| Parse.$$$
;
    >> (fn (x) => Send(x))
    : a parser
val parse-receive = Parse.binding --| keyword⟨<-⟩ -- Parse.binding --| Parse.$$$
;
    >> (fn (x) => Receive(x))
    : a parser
val parse-skip = keyword⟨SKIP⟩ |-- Parse.$$$ ;
    >> (fn - => Skip) : a parser

fun parse-actions TL = ( Scan.repeat ( parse-lock
    || parse-unlock
    || parse-assign
    || parse-send
    || parse-receive
    || parse-skip
    || parse-if-else
    || parse-while
  )) TL

and parse-if-else TL = ((keyword⟨IF⟩ |-- Parse.term --| keyword⟨THEN⟩
  -- parse-actions --| keyword⟨ELSE⟩ -- parse-actions--|
keyword⟨DONE⟩)
  >>(fn (x) => Ifelse(x))) TL
and parse-while TL = (( keyword⟨WHILE⟩ |-- Parse.term --| keyword⟨DO⟩
  -- parse-actions --| keyword⟨DONE⟩)
  >> (fn (x) => While(x))) TL

(*thread T1 : where var-local-decl
  actions action1;
  ... ;
  actionN;
  thread T2: actions action1;
  ... ;
  actionN;
*)
val parse-threads = ( keyword⟨thread⟩ |-- (Scan.option (Parse.binding)) --|

```

```

Parse.$$$ :
    -- (Scan.option (keyword <any> | -- Scan.repeat1
(parse-var-decl) ))
    -- (keyword <actions> | -- parse-actions))

type raw-absy = (((binding (*system name*)
    *
    ((binding * (string * Position.T)) * string option)list) (*globals*)
    *
    (binding * (string * Position.T)) list) (*locks*)
    *
    ((binding option * (*a thread's name*)
    ((binding * (string * Position.T)) *string option)list option) (*a thread's
locals*)
    *
    a list (*a thread's actions*)
    ) list) (*threads lists*)

type raw-thread-absy = (binding option * (*a thread's name*)
    ((binding * (string * Position.T)) *string option)list option)*
(*a thread's locals*)
    a list (*a thread's actions*)

val parse-system-spec = (
    Parse.binding
    --| keyword <globals>    -- (Scan.repeat parse-var-decl)
    --| keyword <locks>      -- (Scan.repeat parse-locks)
    -- (Scan.repeat1 parse-threads)
    --| keyword <end;>
) : raw-absy parser

end

```

```

typed decl pre-event-type

```

2.4 Context Checking

```

ML<
val pre-event-type = @{typ <pre-event-type>}

```

```

datatype a-term = SkipA
    | AssignA of binding*term
    | UnlockA of binding
    | LockA of binding
    | SendA of binding*term
    | ReceiveA of binding*binding

```

```

| IfelseA of (term*(a-term list))*(a-term list)
| WhileA of term*(a-term list)
| SeqA of a-term * a-term

type thread-absy = {nom-thread : binding ,
                    locals-decl : ((binding * typ) * term option) list ,
                    actions      : a-term list,
                    locstab      : (term option) Symtab.table
                  }
type absy = {system      : binding,
             globals-decls : (binding * typ * Position.T * term option) list,
             varstab      : (term option) Symtab.table,
             locks-decls  : (binding * term * Position.T) list,
             threads-decls : thread-absy list
           }

val SPY = Unsynchronized.ref(NONE:absy option)
val SPYG = Unsynchronized.ref([Bound 0])

fun pre-const ((bdg, typ), -) = (bdg, typ, NoSyn)
fun pre-const' (b, trm) = (b, fastype-of trm, Mixfix.NoSyn)
fun pre-const-bar (b, trm, -) = (Binding.suffix-name ' b,
                                fastype-of trm, Mixfix.NoSyn)
fun pre-const2 (b, typ, -, -) = (b, typ, Mixfix.NoSyn)
fun pre-const3 (b, trm, -) = (b, fastype-of trm, Mixfix.NoSyn)

fun pre-event (b, trm, -) = (b, fastype-of trm --> pre-event-type,
                             Mixfix.NoSyn)

(* Got replaced but would be cool to know why isn't it working

fun check-vars term varstab locstab =
  let
    val vars = Term.add-free-names term []
    fun is-declared v = Symtab.defined varstab v orelse Symtab.defined locstab v
    val undeclared = List.filter (not o is-declared) vars
  in
    if null undeclared then ()
    else error (Undeclared variable(s) used: ^ String.concatWith , undeclared)
  end*)

fun read-term-err thy msg bind str =
  let val pos = Position.here(Binding.pos-of bind)
  in (Syntax.read-term-global thy str
      handle ERROR s => error(error in ^msg^:^pos ^\n^ s ) )
  end

```

```

fun read-term-err-pos thy msg - str pos =
  (Syntax.read-term-global thy str
   handle ERROR s => error(error in ^msg^:^(Position.here pos) ^\n^
s ) )

```

(* Utilitaire pour lire un terme à partir d'un string *)

```

fun get-term-and-vars thy str =
  let
    val ctxt = Proof-Context.init-global thy
    val term = Syntax.read-term ctxt str
    val vars = Term.add-free-names term []
  in
    (term, vars)
  end

```

(* Résout une variable selon priorité locale ou globale *)

```

fun resolve-var prefer-global var locstab varstab =
  let
    val loc = Symtab.lookup locstab var
    val global = Symtab.lookup varstab var
  in
    if prefer-global then
      (case global of
        SOME (SOME t) => SOME t
      | - => (case loc of
        SOME (SOME t) => SOME t
        | - => NONE))
    else
      (case loc of
        SOME (SOME t) => SOME t
      | - => (case global of
        SOME (SOME t) => SOME t
        | - => NONE))
  end

```

(* Variante stricte qui renvoie une erreur si non trouvé *)

```

fun resolve-var-strict prefer-global var locstab varstab =
  case resolve-var prefer-global var locstab varstab of
    SOME t => t
  | NONE => error (Variable ^var^ non déclarée)

```

```

fun check-thread thy thy3 vars-tab (((SOME thread-name, locals-opt), actions-list):
raw-thread-absy) =

```

```

  let
    fun loc-decls-conv ((bdg, (s, pos)), SOME init-term) =
      let
        val typ = Syntax.read-typ-global thy s
        val inferred-type = fastype-of init-term

```

```

    val ctxt = Proof-Context.init-global thy
    val inferred-str = Syntax.string-of-typ ctxt inferred-type
    val declared-str = Syntax.string-of-typ ctxt typ
  in
    if inferred-type <> typ then
      error (Type mismatch in variable ^ Binding.name-of bdg ^
        : expected ^ declared-str ^, but got ^ inferred-str ^ at ^
Position.here pos )
    else
      ((bdg, typ), SOME init-term)
  end
| loc-decls-conv ((bdg, (s, pos)), NONE) =
  ((bdg, Syntax.read-typ-global thy s), NONE)

fun action-conv - - Skip = SkipA
| action-conv - - (Lock b) = LockA b
| action-conv - - (Unlock b) = UnlockA b
| action-conv varstab locstab (Assign (bdg, str)) =
  let
    val (term, vars) = get-term-and-vars thy str
    val assign-term = term
    val - =
      if Symtab.defined locstab (Binding.name-of bdg)
      orelse Symtab.defined varstab (Binding.name-of bdg)
      then ()
      else error (Variable ^ (Binding.name-of bdg) ^ is not declared (in
assign binding))
  in
    AssignA (bdg, assign-term)
  end

| action-conv varstab locstab (Send (b, str)) =
  let
    val (term, -) = get-term-and-vars thy str
    val - =
      if Symtab.defined locstab (Binding.name-of b)
      orelse Symtab.defined varstab (Binding.name-of b)
      then ()
      else error (Variable ^ (Binding.name-of b) ^ is not declared (in send))
  in
    SendA (b, term)
  end

| action-conv varstab locstab (Receive (blv, bsv)) =
  let
    val - =
      if Symtab.defined locstab (Binding.name-of blv)
      then if (Symtab.defined varstab (Binding.name-of bsv)) then ()

```

```

      else error (Variable ^ (Binding.name-of bsv) ^ is not declared
(in receive binding))
      else error (Variable ^ (Binding.name-of blv) ^ is not declared (in
receive binding))
    in
      ReceiveA (blv, bsv)
    end

| action-conv varstab locstab (Ifelse ((cond-str, then-branch), else-branch)) =
  let
    val (term, vars) = get-term-and-vars thy cond-str
    val cond-term = term
    val then-branch' = map (action-conv varstab locstab) then-branch
    val else-branch' = map (action-conv varstab locstab) else-branch
  in
    IfelseA ((cond-term, then-branch'), else-branch')
  end

| action-conv varstab locstab (While (cond-str, body)) =
  let
    val (term, vars) = get-term-and-vars thy cond-str
    (* val cond-term =
      case vars of
        [x] => resolve-var-strict false x locstab varstab
      | - => term*)
    val cond-term = term
    val actions = map (action-conv varstab locstab) body
  in
    WhileA (cond-term, actions)
  end

val (loc-decls', thy3') = case locals-opt of
  NONE => ([], thy3)
| SOME locals =>
  let
    val locals-terms = map (fn ((bdg,(s,pos)), init-opt-str) =>
      let
        val init-opt-term = case init-opt-str of
          NONE => NONE
        | SOME str => SOME (read-term-err thy local init bdg str)
      in
        ((bdg, (s,pos)), init-opt-term)
      end) locals
  in
    val decls = map loc-decls-conv locals-terms
    val thy4 = Sign.add-consts (map pre-const decls) thy3
  in
    (decls, thy4)
  end

```

```

fun conv-decl-init ((n, -), init-opt) = (Binding.name-of n, init-opt)
val locstab' = Symtab.make (map conv-decl-init loc-decls')
val actions' = map (action-conv vars-tab locstab') actions-list in
{ nom-thread = thread-name, locals-decl = loc-decls', actions = actions', locstab
= locstab' }
end
| check-thread - - ((NONE, -), -) =
  error Current syntax restriction: threads must have label.

```

```

val context-check = fn ((system: binding ,
  globals : ((binding * (string * Position.T)) * string option)list),
  locks : (binding * (string * Position.T)) list),
  threads : raw-thread-absy list)
=>
fn thy : theory =>
let val read = read-term-err thy
  fun global-decls-conv ((bdg, (s, pos)), SOME init-term) =
    let
      val typ = Syntax.read-typ-global thy s
      val inferred-type = fastype-of init-term
      val ctxt = Proof-Context.init-global thy
      val inferred-str = Syntax.string-of-typ ctxt inferred-type
      val declared-str = Syntax.string-of-typ ctxt typ
    in
      if inferred-type <> typ then
        error (Type mismatch in variable ^ Binding.name-of bdg ^
          : expected ^ declared-str ^, but got ^ inferred-str ^
          at ^Position.here pos)
      else
        (bdg, typ, pos, SOME init-term)
    end
  | global-decls-conv ((bdg, (s, pos)), NONE) =
    (bdg, Syntax.read-typ-global thy s, pos, NONE)
val (global-decls', thy3') =
  let
    val global-terms = map (fn ((bdg,(s,pos)), init-opt-str)
=>
      let
        val init-opt-term = case init-opt-str of
          NONE => NONE
        | SOME str => SOME (read-term-err thy local
init bdg str)
      in
        ((bdg, (s,pos)), init-opt-term)
      end) global-terms

```

```

      val decls = map global-decls-conv global-terms
      val thy4 = Sign.add-consts (map pre-const2 decls) thy
    in
      (decls, thy4)
    end

    fun conv-decl-init (n,-, - , init-opt) = (Binding.name-of n,
init-opt)

    val varstab    = Symtab.make (map conv-decl-init global-decls')
    val lock-decls' = map (fn (b, (str,pos)) => (b, read lock-decl b
str, pos)) locks

    val thy'       = Sign.add-consts (map pre-const2 global-decls')
thy

    val thy''      = thy' |> Sign.add-consts (map pre-event
lock-decls')

    val threads' = map (check-thread thy thy'' varstab) threads
    val res = {system = system,
globals-decls = global-decls',
varstab = varstab,
locks-decls = lock-decls',
threads-decls = threads'}
    in SPY:=(SOME (res)); res end

  >

```

2.5 Semantic Check

```

ML<
(* Redéfinir les types de base de la sémantique *)
type SV = string (* Shared Variables *)
type V = string  (* Thread-Local Variables *)
type MV = int    (* Mutex IDs *)
type D = int     (* Data type – simplifié à int *)

type state = string -> int

fun safe-lookup var state =
  case state var of
    ~1 => raise Undeclared-Variable var
  | v => v

type arith-expr = state -> int
type bool-expr = state -> bool

datatype com =
  SKIP

```

```

| Assign of string * arith-expr
| Seq of com * com
| Cond of bool-expr * com * com
| While of bool-expr * com
| Lock of string
| Unlock of string
| Store of arith-expr * string
| Load of string * string

(* Table de mapping pour les locks *)
val lock-table = Unsynchronized.ref (Symtab.empty : int Symtab.table)
val lock-counter = Unsynchronized.ref 1
(*
type absy = {system          : binding,
              globals-decls  : (binding * typ * Position.T * term option) list,
              varstab        : (term option) Symtab.table,
              locks-decls    : (binding * term * Position.T) list,
              threads-decls  : thread-absy list
            }

fun get-lock-id (lock-name : string) : int =
  case Symtab.lookup (!lock-table) lock-name of
    SOME id => id
  | NONE =>
    let
      val new-id = !lock-counter
      val - = lock-counter := new-id + 1
      val - = lock-table := Symtab.update (lock-name, new-id) (!lock-table)
    in
      new-id
    end
  *)

fun declared-lock (name: string) (machine: absy) : bool =
  List.exists (fn (b, -, -) => Binding.name-of b = name) (#locks-decls machine)

fun get-lock-id (lock-name: string) (machine: absy) : int =
  if not (declared-lock lock-name machine) then
    error (Lock ^ lock-name ^ not declared)
  else
    case Symtab.lookup (!lock-table) lock-name of
      SOME id => id
    | NONE =>
      let
        val new-id = !lock-counter
        val - = lock-counter := new-id + 1
        val - = lock-table := Symtab.update (lock-name, new-id) (!lock-table)
      in
        new-id
      end

```

```

    end;
(* Évaluation des termes Isabelle/HOL *)
fun eval-term-to-int (thy : theory) (term : term) (state : state) : int =
  let
    (* Fonction récursive pour évaluer les termes *)
    fun eval t =
      case t of
        Const (@{const-name Groups.plus-class.plus}, -) $ t1 $ t2 =>
          (eval t1) + (eval t2)
      | Const (@{const-name Groups.minus-class.minus}, -) $ t1 $ t2 =>
          (eval t1) - (eval t2)
      | Const (@{const-name Groups.times-class.times}, -) $ t1 $ t2 =>
          (eval t1) * (eval t2)
      | Const (@{const-name HOL.Trueprop}, -) $ t1 => eval t1
      | @term 0::nat => 0
      | @term 1::nat => 1
      | Free (name, -) => safe-lookup name state
      | Bound i => i (* Placeholder pour les variables liées *)
      | - =>
          (* Essayer d'extraire les constantes numériques *)
          (case try (HOLogic.dest-number #> snd) t of
            SOME n => n
          | NONE => 0) (* Valeur par défaut *)
    in
      eval term
    end

fun eval-term-to-bool (thy : theory) (term : term) (state : state) : bool =
  let
    fun eval t =
      case t of
        @term True => true
      | @term False => false
      | Const (@{const-name HOL.conj}, -) $ t1 $ t2 =>
          (eval t1) andalso (eval t2)
      | Const (@{const-name HOL.disj}, -) $ t1 $ t2 =>
          (eval t1) orelse (eval t2)
      | Const (@{const-name HOL.Not}, -) $ t1 =>
          not (eval t1)
      | Const (@{const-name HOL.eq}, -) $ t1 $ t2 =>
          (eval-term-to-int thy t1 state) = (eval-term-to-int thy t2 state)
      | Const (@{const-name Orderings.ord-class.less}, -) $ t1 $ t2 =>
          (eval-term-to-int thy t1 state) < (eval-term-to-int thy t2 state)
      | Const (@{const-name Orderings.ord-class.less-eq}, -) $ t1 $ t2 =>
          (eval-term-to-int thy t1 state) <= (eval-term-to-int thy t2 state)
      | Free (name, -) =>
          (case try (fn () => safe-lookup name state) () of
            SOME 0 => false
          | SOME - => true
          | _ => false)
    in
      eval term
    end
  end

```

```

      | NONE => false)
    | - => error (Unsupported boolean expression: ^ Pretty.string-of (Syntax.pretty-term-global
thy t) )
                                     (*Syntax.string-of-term-global thy t)*)

  in
    eval term
  end
(* Fixed record pattern matching *)
(*fun process-absy absy-data =
  let
    val {system, globals-decls, locks-decls, threads-decls, varstab} = absy-data
  in
    (* Process the fields *)
    (system, globals-decls, locks-decls, threads-decls)
  end
*)
(* Fonction de conversion de l'AST vers com *)
fun convert-to-com (thy : theory) (actions : a-term list)
  (varstab : (term option) Symtab.table)
  (locstab : (term option) Symtab.table) : com =
  let
    fun term-to-arith-expr (t : term) : arith-expr =
      fn state => eval-term-to-int thy t state

    fun term-to-bool-expr (t : term) : bool-expr =
      fn state => eval-term-to-bool thy t state

    fun convert-action (action : a-term) : com =
      case action of
        SkipA => SKIP
      | AssignA (bdg, term) =>
        let
          val name = Binding.name-of bdg
          val is-local = Symtab.defined locstab name
        in
          if is-local
          then Assign (name, term-to-arith-expr term) (* local *)
          else Store (term-to-arith-expr term, name) (* global *)
        end
      | LockA bdg =>
        Lock (Binding.name-of bdg)
      | UnlockA bdg =>
        Unlock (Binding.name-of bdg)
      | SendA (bdg, term) =>
        Store (term-to-arith-expr term, Binding.name-of bdg)
      | ReceiveA (local-bdg, shared-bdg) =>
        Load (Binding.name-of local-bdg, Binding.name-of shared-bdg)
      | IfelseA ((cond-term, then-actions), else-actions) =>
        Cond (term-to-bool-expr cond-term,

```

```

      convert-actions-to-seq then-actions,
      convert-actions-to-seq else-actions)
| WhileA (cond-term, body-actions) =>
  While (term-to-bool-expr cond-term, convert-actions-to-seq body-actions)

and convert-actions-to-seq (actions : a-term list) : com =
  case actions of
  [] => SKIP
| [action] => convert-action action
| action :: rest => Seq (convert-action action, convert-actions-to-seq rest)
in
  convert-actions-to-seq actions
end

fun check-lock-ownership (thread-name : string) (actions : a-term list) =
  let
    fun check [] - = NONE
    | check (LockA b :: rest) held =
      let val l = Binding.name-of b in
        if List.exists (fn x => x = l) held then
          SOME (Thread ^ thread-name ^ : LOCK ^ l ^ already held)
        else
          check rest (l :: held)
      end
    | check (UnlockA b :: rest) held =
      let val l = Binding.name-of b in
        if List.exists (fn x => x = l) held then
          check rest (remove (op =) l held)
        else
          SOME (Thread ^ thread-name ^ : tries to UNLOCK ^ l ^ without
holding it)
      end
    | check (IfelseA ((-, ifBody), elseBody) :: rest) held =
      (case check ifBody held of
        SOME err => SOME err
      | NONE =>
        (case check elseBody held of
          SOME err => SOME err
        | NONE => check rest held))
    | check (WhileA (-, body) :: rest) held =
      (case check body held of
        SOME err => SOME err
      | NONE => check rest held)
    | check (- :: rest) held = check rest held
  in
    check actions []
  end

```

```

(* Alternative CPS-style semantics *)
datatype csp-event =
  | LockEvent of string * int
  | UnlockEvent of string * int
  | ReadEvent of string * int
  | UpdateEvent of string * int
  | AssignEvent of string * int
  | SkipEvent
(* Mise à jour de l'état fonctionnel *)
fun update-state state x v = fn y => if y = x then v else state y

(* Semantique CPS complète *)
fun sem-cps - SKIP cont =
  (fn state => SkipEvent :: cont state)

| sem-cps - (Assign (x, e)) cont =
  (fn state =>
    let
      val v = e state
    in
      AssignEvent (x, v) :: cont (update-state state x v)
    end)

| sem-cps machine (Seq (P, Q)) cont =
  sem-cps machine P (sem-cps machine Q cont)

| sem-cps machine (Cond (cond, t1, t2)) cont =
  (fn state =>
    if cond state then sem-cps machine t1 cont state
    else sem-cps machine t2 cont state)

| sem-cps machine (While (cond, body)) cont =
  (let
    fun loop state =
      if cond state then
        sem-cps machine body loop state
      else cont state
  in loop end)

| sem-cps machine (Lock id) cont =
  (fn state => LockEvent (id, get-lock-id id machine) :: cont state)

| sem-cps machine (Unlock id) cont =
  (fn state => UnlockEvent (id, get-lock-id id machine) :: cont state)

| sem-cps - (Store (e, x)) cont =
  (fn state =>
    let
      val v = e state

```

```

      val new-state = update-state state x v
    in
      UpdateEvent (x, v) :: cont new-state
    end)
| sem-cps - (Load (x, y)) cont =
  (fn state =>
    let
      val v = safe-lookup y state
      val new-state = update-state state x v
    in
      ReadEvent (y, v) :: cont new-state
    end)
type csp-trace = csp-event list

(* Fonction Sem adaptée avec gestion des événements CSP *)
fun sem-step (thy : theory) (machine: absy) (cmd : com) (cont : state * csp-trace
-> state * csp-trace)
  (s : state, trace : csp-trace) : state * csp-trace =
  case cmd of
    SKIP => cont (s, trace)
  | Assign (x, expr) =>
    let
      val new-value = expr s
      val new-state = fn v => if v = x then new-value else s v
    in
      cont (new-state, trace)
    end
  | Seq (p, q) =>
    sem-step thy machine p (fn (s', trace') => sem-step thy machine q cont (s',
trace')) (s, trace)
  | Cond (guard, c1, c2) =>
    if guard s
    then sem-step thy machine c1 cont (s, trace)
    else sem-step thy machine c2 cont (s, trace)
  | While (guard, body) =>
    if guard s
    then sem-step thy machine body (fn (s', trace') =>
      sem-step thy machine (While (guard, body)) cont (s', trace')) (s, trace)
    else cont (s, trace)
  | Lock lock-name =>
    let
      val lock-id = get-lock-id lock-name machine
      val new-trace = LockEvent (lock-name, lock-id) :: trace
    in
      cont (s, new-trace)
    end
  | Unlock lock-name =>
    let
      val lock-id = get-lock-id lock-name machine

```

```

    val new-trace = UnlockEvent (lock-name, lock-id) :: trace
  in
    cont (s, new-trace)
  end
| Store (expr, var) =>
  let
    val value = expr s
    val new-state = update-state s var value
    val new-trace = UpdateEvent (var, value) :: trace
  in
    cont (new-state, new-trace)
  end
| Load (local-var, global-var) =>
  let
    (*TODO*)
    val read-value = s global-var
    val new-state = fn v => if v = local-var then read-value else s v
    val new-trace = ReadEvent (global-var, read-value) :: trace
  in
    cont (new-state, new-trace)
  end
end

```

(* Analyses sémantiques avancées *)

(* Détection de deadlocks simples *)

```

fun detect-potential-deadlock (threads : thread-absy list) : string list =
  let
    fun extract-locks-from-actions (actions : a-term list) : string list =
      let
        fun extract-from-action action =
          case action of
            LockA bdg => [Binding.name-of bdg]
          | UnlockA bdg => []
          | IfelseA ((-, then-actions), else-actions) =>
              (extract-locks-from-actions then-actions) @
              (extract-locks-from-actions else-actions)
          | WhileA (_, body-actions) =>
              extract-locks-from-actions body-actions
          | - => []
        in
          in
            List.concat (map extract-from-action actions)
          end
        end
      end
    fun get-thread-locks (thread : thread-absy) : string * string list =
      let
        val {nom-thread, actions, ...} = thread
        val locks = extract-locks-from-actions actions
      in
        (Binding.name-of nom-thread, locks)
      end
    end
  end

```

```

end

val thread-locks = map get-thread-locks threads

(* Détection simple : si deux threads utilisent les mêmes verrous *)
fun find-shared-locks [] = []
  | find-shared-locks ((t1, locks1) :: rest) =
    let
      fun check-overlap (t2, locks2) =
        let
          val shared = List.filter (fn l => List.exists (fn l2 => l = l2) locks2)
            locks1
        in
          if null shared then NONE
          else SOME (Potential deadlock between  $\hat{t1}$  and  $\hat{t2}$ 
            on locks:  $\hat{String.concatWith}$  , shared)
          end
        end
      val warnings = List.mapPartial check-overlap rest
    in
      warnings @ find-shared-locks rest
    end

in
  find-shared-locks thread-locks
end

(* Détection de race conditions *)
fun detect-race-conditions (threads : thread-absy list) : string list =
  let
    fun extract-global-vars-from-actions (actions : a-term list) : (string * bool) list
      =
        let
          fun extract-from-action action =
            case action of
              SendA (bdg, -) => [(Binding.name-of bdg, true)] (* write *)
            | ReceiveA (-, bdg) => [(Binding.name-of bdg, false)] (* read *)
            | IfelseA ((-, then-actions), else-actions) =>
              (extract-global-vars-from-actions then-actions) @
              (extract-global-vars-from-actions else-actions)
            | WhileA (-, body-actions) =>
              extract-global-vars-from-actions body-actions
            | - => []
          in
            List.concat (map extract-from-action actions)
          end
        end
    fun get-thread-global-access (thread : thread-absy) : string * (string * bool) list
      =
        let
          val {nom-thread, actions, ...} = thread

```

```

    val accesses = extract-global-vars-from-actions actions
  in
    (Binding.name-of nom-thread, accesses)
  end

  val thread-accesses = map get-thread-global-access threads

  (* Détection : si deux threads accèdent à la même variable et au moins un écrit *)
  fun find-races [] = []
  | find-races ((t1, accesses1) :: rest) =
    let
      fun check-race (t2, accesses2) =
        let
          fun has-conflict var =
            let
              val t1-writes = List.exists (fn (v, is-write) => v = var andalso
is-write) accesses1
              val t1-reads = List.exists (fn (v, is-write) => v = var andalso not
is-write) accesses1
              val t2-writes = List.exists (fn (v, is-write) => v = var andalso
is-write) accesses2
              val t2-reads = List.exists (fn (v, is-write) => v = var andalso not
is-write) accesses2
            in
              (t1-writes orelse t1-reads) andalso (t2-writes orelse t2-reads) andalso
              (t1-writes orelse t2-writes)
            end
          end
        let
          val all-vars = map fst (accesses1 @ accesses2)
          val conflicting-vars = List.filter has-conflict all-vars
        in
          if null conflicting-vars then []
          else [Potential race condition between  $\wedge t1 \wedge$  and  $\wedge t2 \wedge$ 
on variables:  $\wedge$  String.concatWith , conflicting-vars]
          end
        val warnings = List.concat (map check-race rest)
      in
        warnings @ find-races rest
      end
    in
      find-races thread-accesses
    end

  (* Vérification de cohérence des types *)
  fun type-check-thread (thy : theory) (thread : thread-absy) (vars-decl) : string list =
    let
      val {nom-thread, locals-decl, actions, locstab} = thread
      fun check-action-types (action, errors) =
        case action of
          AssignA (bdg, term) =>

```

```

let
  val var-name = Binding.name-of bdg
  val term-type = fastype-of term
  val expected-type-opt =
    case find-first (fn ((bdg', -), -) => Binding.name-of bdg' =
var-name) locals-decl of
      SOME ((-, typ), -) => SOME typ
    | NONE => NONE
in
  case expected-type-opt of
    SOME expected-type =>
      if term-type = expected-type then
        errors
      else
        (Type mismatch in assignment to ^var-name^
         : expected ^Syntax.string-of-typ-global thy expected-type^
         , got ^Syntax.string-of-typ-global thy term-type) :: errors
    | NONE =>
        (Variable ^var-name^ is not locally declared) :: errors
end

| SendA (bdg, term) =>
  let
    val var-name = Binding.name-of bdg
    val term-type = fastype-of term
    val expected-type-opt =
      case find-first (fn ( bdg', -, -, -) => Binding.name-of bdg' =
var-name) vars-decl of
        SOME ( -, typ, -, -) => SOME typ
      | NONE => NONE
  in
    case expected-type-opt of
      SOME expected-type =>
        if term-type = expected-type then
          errors
        else
          (Type mismatch in assignment to ^var-name^
           : expected ^Syntax.string-of-typ-global thy expected-type^
           , got ^Syntax.string-of-typ-global thy term-type) :: errors
      | NONE =>
          (Variable ^var-name^ is not globally declared) :: errors
    end
  end

| ReceiveA (local-bdg, shared-bdg) =>
  let
    val local-var = Binding.name-of local-bdg
    val shared-var = Binding.name-of shared-bdg
    val local-type-opt =
      case find-first (fn ((bdg', -), -) => Binding.name-of bdg' =

```

```

local-var) locals-decl of
    SOME ((-, typ), -) => SOME typ
  | NONE => NONE

val shared-type-opt =
  (* DOUBLE CHECK IF THIS CAN SAFELY BE REMOVED

  case Symtab.lookup varstab shared-var of
    SOME (SOME t) => fastype-of t
  | NONE =>
    case Symtab.lookup (!SPY |> Option.valOf |> #varstab) shared-var of
      SOME (SOME t) => fastype-of t
    | - => error (Shared variable ^ shared-var ^ not declared)
    | SOME NONE => error (Variable ^ shared-var ^ declared but has
no initial value)*)
  case find-first (fn (bdg', -, -, -) => Binding.name-of bdg' = shared-var)
vars-decl of
    SOME ( -, typ, -, -) => SOME typ
  | NONE => NONE

in
  case (local-type-opt, shared-type-opt) of
    (SOME local-type, SOME shared-type) =>
      if local-type = shared-type then
        errors
      else
        (Type mismatch in receive: cannot assign ^ shared-var ^ to ^
local-var ^
      (expected ^ Syntax.string-of-typ-global thy local-type ^
      , got ^ Syntax.string-of-typ-global thy shared-type ^)) :: errors
  | (NONE, -) =>
    (Local variable ^ local-var ^ not declared) :: errors
  | (-, NONE) =>
    (Shared variable ^ shared-var ^ not declared) :: errors
  end
| IfelseA ((cond, then-branch), else-branch) =>
  let
    val cond-type =
      let
        val t = cond
      in
        case cond of
          Free (x, -) =>
            (case Symtab.lookup locstab x of
              SOME (SOME t') => fastype-of t'
            | SOME NONE => error (Variable ^ x ^ declared but has no
initial value)
            | NONE =>
              (case Symtab.lookup (!SPY |> Option.valOf |> #varstab) x of

```

```

      SOME (SOME t') => fastype-of t'
    | SOME NONE => error (Global variable  $\hat{x}$  declared but
has no initial value)
    | NONE => fastype-of t)) (* fallback to actual term type *)
  | - => fastype-of t
end

val cond-error =
  if cond-type = @{typ bool} then []
  else [Condition must be boolean, got:  $\hat{x}$  Syntax.string-of-typ-global thy
cond-type]

val then-errors = List.foldl check-action-types [] then-branch
val else-errors = List.foldl check-action-types [] else-branch
in
  cond-error @ then-errors @ else-errors @ errors
end

| WhileA (cond, body) =>
  let
    val cond-type =
      let
        val t = cond
      in
        case cond of
          Free (x, -) =>
            (case Symtab.lookup locstab x of
              SOME (SOME t') => fastype-of t'
            | SOME NONE => error (Variable  $\hat{x}$  declared but has no
initial value)
            | NONE =>
              (case Symtab.lookup (!SPY |> Option.valOf |> #varstab) x of
                SOME (SOME t') => fastype-of t'
              | SOME NONE => error (Global variable  $\hat{x}$  declared
but has no initial value)
              | NONE => fastype-of t)) (* fallback *)
            | - => fastype-of t
          end
        val cond-error =
          if cond-type = @{typ bool} then []
          else [While condition must be boolean, got:  $\hat{x}$  Syntax.string-of-typ-global
thy cond-type]

        val body-errors = List.foldl check-action-types [] body
      in
        cond-error @ body-errors @ errors
      end
    end
  end

```

```

    | - => errors

in
  List.foldl check-action-types [] actions
end

fun build-initial-state-for-thread
  (globals : (binding * typ * Position.T * term option) list)
  (locals : ((binding * typ) * term option) list) (thy: theory) : state =
let
  fun term-to-int t-opt =
    case t-opt of
      NONE => 0
    | SOME t => eval-term-to-int thy t (fn - => 0)

  val global-vals = map (fn (bdg, -, -, t-opt) =>
    (Binding.name-of bdg, term-to-int t-opt)) globals

  val local-vals = map (fn ((bdg, -), t-opt) =>
    (Binding.name-of bdg, term-to-int t-opt)) locals

  val all-vals = local-vals @ global-vals
in
  fn var => case AList.lookup (op =) all-vals var of
    SOME v => v
  | NONE => 0
end

(* Fonction de vérification sémantique principale complète *)
fun semantic-check (absy-data : absy) (thy : theory) : theory =
let
  val {system, globals-decls, varstab, locks-decls, threads-decls} = absy-data

  val - = message false (=== SEMANTIC ANALYSIS FOR SYSTEM: ^
    Binding.name-of system ^ ===)

  (* Vérification 1: Analyse des deadlocks *)
  val deadlock-warnings = detect-potential-deadlock threads-decls
  val - = if null deadlock-warnings then
    message false ✓ No potential deadlocks detected
  else
    (message false DEADLOCK WARNINGS;;
     map (message false) deadlock-warnings; ())

  (* Vérification 2: Analyse des race conditions *)
  val race-warnings = detect-race-conditions threads-decls
  val - = if null race-warnings then
    message false ✓ No race conditions detected
  else

```

```

(message false RACE CONDITION WARNINGS;;
 map (message false) race-warnings; ())

(* Vérification 3: Vérification des types pour chaque thread *)
fun check-thread (machine : absy) (thread : thread-absy) : unit =
let
  val {system, globals-decls, varstab, locks-decls, threads-decls} = machine
  val {nom-thread, locals-decl, actions, locstab} = thread
  val thread-name = Binding.name-of nom-thread

  val - = message false (Analyzing thread: ^ thread-name)

  (* Conversion vers com pour analyse sémantique *)
  val com-program = convert-to-com thy actions varstab locstab

  val initial-state = build-initial-state-for-thread globals-decls locals-decl thy

  val cps-trace = sem-cps machine com-program (fn - => []) initial-state

  val - = message false ( ✓ CPS-style trace for ^ thread-name ^:)
  val - = List.app (fn ev => message false ( ^ @ {make-string} ev)) cps-trace
  (* Vérification des types *)
  val type-errors = type-check-thread thy thread globals-decls
  val - = if null type-errors then
    message false ( ✓ Type checking passed for ^ thread-name)
  else
    (message false ( ✗ Type errors in ^ thread-name ^:);
     map (fn err => message false ( ^ err)) type-errors; ())
  (* Vérification que lock et unlock respectent des règles d'appartenance *)
  val - = case check-lock-ownership thread-name actions of
    NONE => ()
  | SOME msg => error msg

  (* Simulation d'exécution pour détecter d'autres problèmes *)
  val initial-state = build-initial-state-for-thread globals-decls locals-decl thy
  val (final-state, execution-trace) =
    sem-step thy machine com-program (fn (s, t) => (s, t)) (initial-state, [])

  val - = message false ( ✓ Execution simulation completed for ^ thread-name)
  val - = message false ( Generated ^
    Int.toString (length execution-trace) ^ CSP events)

in
  ()
end

val - = map (check-thread absy-data) (threads-decls)

(* Vérification 4: Cohérence globale du système *)
val - = message false === GLOBAL SYSTEM CONSISTENCY ===

```

```

(* Vérifier que tous les verrous utilisés sont déclarés *)
fun check-lock-declarations () =
  let
    val declared-locks = map (fn (bdg, -, -) => Binding.name-of bdg) locks-decls

    fun extract-used-locks-from-thread (thread : thread-absy) =
      let
        fun extract-from-actions actions =
          List.concat (map (fn action =>
            case action of
              LockA bdg => [Binding.name-of bdg]
            | UnlockA bdg => [Binding.name-of bdg]
            | IfelseA ((-, then-acts), else-acts) =>
              extract-from-actions then-acts @ extract-from-actions else-acts
            | WhileA (-, body-acts) => extract-from-actions body-acts
            | - => []) actions)
          in
            extract-from-actions (#actions thread)
          end

        val used-locks = List.concat (map extract-used-locks-from-thread threads-decls)
        val undeclared-locks = List.filter (fn used =>
          not (List.exists (fn decl => decl = used) declared-locks)) used-locks
      in
        if null undeclared-locks then
          message false ✓ All used locks are properly declared
        else
          (message false ✗ Undeclared locks used;;
           map (fn lock => message false ( ^ lock)) undeclared-locks; ())
      end
  end

val - = check-lock-declarations ()

fun check-var-declarations () =
  let
    (* ----- 1. Ensemble des noms globaux ----- *)
    val global-names =
      map (fn (bdg, -, -, -) => Binding.name-of bdg) globals-decls

    (* ----- 2. Noms locaux d'un thread ----- *)
    fun local-names-of-thread ({locals-decl, ...} : thread-absy) : string list =
      map (fn ((bdg, -), -) => Binding.name-of bdg) locals-decl

    (* ----- 3. Variables utilisées dans des actions ----- *)
    fun extract-vars (actions : a-term list) : string list =

```

```

List.concat (map (fn action =>
  case action of
    AssignA (bdg, -)      => [Binding.name-of bdg]
  | SendA   (bdg, -)      => [Binding.name-of bdg]
  | ReceiveA (lbdg, sbdg) => [Binding.name-of lbdg,
                             Binding.name-of sbdg]
  | IfelseA ((-, t1), t2) => extract-vars t1 @ extract-vars t2
  | WhileA  (-, body)     => extract-vars body
  | -        => [] actions)

(* ----- 4. Manquantes dans un thread donné -----
----- *)
fun undeclared-in-thread (thr : thread-absy) : string list =
  let
    val declared = global-names @ local-names-of-thread thr
    val used     = extract-vars (#actions thr)
  in
    List.filter (fn v => not (List.exists (fn d => d = v) declared)) used
  end

(* ----- 5. Agrégation globale -----
----- *)
val missing =
  List.concat (map undeclared-in-thread threads-decls)
|> sort-distinct string-ord
in
  if null missing then
    message false ✓ All used variables are properly declared (globals or locals)
  else
    (message false ✗ Undeclared variables (neither global nor local));
    List.app (fn v => message false ( ^ v)) missing
  end

(* Appel de la vérification dans semantic-check -----
----- *)
val - = check-var-declarations ()

val - = message false === SEMANTIC CHECK COMPLETED ===
in
  thy
end

fun semantics (absy-cl: theory -> absy) (thy : theory) =
  let
    val absy-data = absy-cl thy
  in
    semantic-check absy-data thy
  end
end

```

```

fun sequence-actions [] = SkipA
  | sequence-actions [a] = a
  | sequence-actions (a::rest) = SeqA (a, sequence-actions rest)
(*
(*Fonction pour générer le code CSP complet *)
fun generate-csp-code (absy-data : absy) (thy : theory) : string =
let
  val {system, globals-decls, locks-decls, threads-decls, varstab} = absy-data
  val system-name = Binding.name-of system

  (*Génération des événements CSP *)
  fun generate-events () =
    let
      val lock-events = map (fn (bdg, -, -) =>
        let val name = Binding.name-of bdg
        in channel lock- ^ name ^, unlock- ^ name ^ : Int
        end) locks-decls

      val global-events = map (fn (bdg, -, -, -) =>
        let val name = Binding.name-of bdg
        in channel read- ^ name ^, update- ^ name ^ : Int
        end) globals-decls
    in
      String.concatWith \n (lock-events @ global-events)
    end

  (*Conversion d'une action en CSP *)
  fun action-list-to-csp [] state thy = (SKIP, state)
    | action-list-to-csp [a] state thy = action-to-csp a state thy
    | action-list-to-csp (a::rest) state thy =
      let
        val (s1, state1) = action-to-csp a state thy
        val (s2, state2) = action-list-to-csp rest state1 thy
      in
        if (a = SkipA) then
          (s2, state2)
        else
          (s1 ^ → ^ s2, state2)
        end
      end

  and action-to-csp action state thy =
    case action of
      SkipA => (SKIP, state)
    | LockA bdg => (lock- ^ Binding.name-of bdg ^ !0, state)
    | UnlockA bdg => (unlock- ^ Binding.name-of bdg ^ !0, state)
    | AssignA (bdg, term) =>
      let
        val name = Binding.name-of bdg
        val typ = fastype-of term

```

```

    val v = if typ = @{typ bool}
      then if eval-term-to-bool thy term state then 1 else 0
      else eval-term-to-int thy term state
    val new-state = update-state state name v
  in
    (update- ^ name ^! ^ Int.toString v, new-state)
  end
| SendA (bdg, term) =>
  let
    val name = Binding.name-of bdg
    val typ = fastype-of term
    val v = if typ = @{typ bool}
      then if eval-term-to-bool thy term state then 1 else 0
      else eval-term-to-int thy term state
    val new-state = update-state state name v
  in
    (update- ^ name ^! ^ Int.toString v, new-state)
  end
| ReceiveA (local-bdg, global-bdg) =>
  let
    val global-var-name = Binding.name-of global-bdg
    val v = safe-lookup global-var-name state
    val local-var-name = Binding.name-of local-bdg
    val new-state = update-state state local-var-name v
  in
    (read- ^ global-var-name ^! ^ Int.toString v, new-state)
  end
| IfelseA ((cond, a1), a2) =>
  let
    val b = eval-term-to-bool thy cond state
    val val-str = if b then true else false
    val (then-code, -) = action-list-to-csp a1 state thy
    val (else-code, -) = action-list-to-csp a2 state thy
  in
    ({ ( ^ val-str ^ & ^ then-code ^ ) [] (¬ ^ val-str ^ & ^ else-code ^ ) },
state)
  end
| WhileA (cond, body) =>
  let
    val val-str = if eval-term-to-bool thy cond state then true else false
    val (body-code, -) = action-list-to-csp body state thy
  in
    ({ ( ^ val-str ^ & μ X. ^ body-code ^ -> X ) [] (¬ ^ val-str ^ & (SKIP) )
}, state)
  end
end

```

(* Conversion d'un thread vers CSP *)

fun thread-to-csp globals-decls (thread : thread-absy) thy : string =

```

    let
      val {nom-thread, actions, locals-decl, ...} = thread
      val thread-name = Binding.name-of nom-thread

      val initial-state = build-initial-state-for-thread globals-decls locals-decl thy
      val (code, -) = action-list-to-csp actions initial-state thy
    in
      thread-name ^ = ^ code
    end

    fun generate-threads globals-decls threads-decls thy =
      String.concatWith \n (map (fn t => thread-to-csp globals-decls t thy)
        threads-decls)

    (*Génération du système global *)
    fun generate-system () =
      let
        val thread-names = map (fn t => Binding.name-of (#nom-thread t))
          threads-decls
      in
        system-name ^ = ^ String.concatWith ||| thread-names
      end

    in
      String.concatWith \n\n [
        -- Channels,
        generate-events (),
        ,
        -- Threads,
        generate-threads globals-decls threads-decls thy,
        ,
        -- System,
        generate-system ()
      ]
    end
  *)
>

```

2.6 Semantic Encoding

definition *SKIP-process* :: *evs process*
where *SKIP-process* ≡ *Skip*

ML_<

(* Convertir une liste de a-term en une seule commande de type com *)
fun *sequence-actions* [] = *SkipA*
 | *sequence-actions* [a] = a
 | *sequence-actions* (a :: rest) = *SeqA* (a, *sequence-actions* rest)

```

(*
fun subst-with-fresh-env (env-type : typ) (t : term) : term * term =
  let
    (* collect free variable names *)
    val frees = Term.add-free-names t []

    (* generate a fresh variable name  $\sigma$ , adding primes if necessary *)
    fun fresh-name base [] = base
      | fresh-name base used =
          let
            fun bump n =
              let val candidate = base ^ replicate-string n ' ' in
                if member (op =) used candidate then bump (n + 1) else candidate
              end
            in bump 1 end

    val sigma-name = fresh-name  $\sigma$  frees
    val sigma = Free (sigma-name, env-type)

    (* perform substitution: Free(x) becomes sigma $ x *)
    fun subst t =
      case t of
        Free (name, -) => sigma $ HOLogic.mk-string name
      | Const - => t
      | Bound - => t
      | Var - => t
      | Abs (n, T, body) => Abs (n, T, subst body)
      | t1 $ t2 => subst t1 $ subst t2

    val t-subst = subst t
  in
    (sigma, t-subst)
  end
*)

val sigma : term = Free ( $\sigma$ , @{typ string  $\Rightarrow$  int}) (* changer le type ici si besoin *)

```

```

fun subst-with-sigma (t : term) : term =
  let
    fun subst tm =
      case tm of
        Free (name, -)      => sigma $ HOLogic.mk-string name
      | Const -             => tm
      | Bound -             => tm
      | Var -               => tm
      | Abs (n, T, body)    => Abs (n, T, subst body)
      | tm1 $ tm2           => subst tm1 $ subst tm2

```

```

in
  subst t
end

fun lambda-sigma-apply-var (env-type : typ) (var-name : string) : term =
  let
    val bound-sigma = Bound 0
    val body = bound-sigma $ HLogic.mk-string var-name
  in
    Abs ( $\sigma$ , env-type, body)
  end

(*
  Crée une abstraction  $\lambda\sigma:typ. t$ 
  en remplaçant les Free( $\sigma, typ$ ) par Bound 0 dans t
*)
fun make-lambda-sigma (env-type : typ) (sigma-name : string) (t : term) : term =
  let
    fun subst tm =
      case tm of
        Free (n, T) => if n = sigma-name andalso T = env-type then Bound 0 else
      tm
    | Const - => tm
    | Bound - => tm
    | Var - => tm
    | Abs (n, T, body) => Abs (n, T, subst body)
    | t1 $ t2 => subst t1 $ subst t2
  in
    Abs (sigma-name, env-type, subst t)
  end

(* Conversion d'une action abstraite (a-term) vers le terme logique com *)
fun quote-com - SkipA =
  @{term com.SKIP :: com}
  (* | quote-com (AssignA (b, e)) =
    let
      val var-str = HLogic.mk-string (Binding.name-of b)

      val (sigma, e') = subst-with-fresh-env @{typ string => int} e
      val lam = Abs (fst (Term.dest-Free sigma), @{typ string => int}, e')
      val lam = Abs ( $\sigma$ , @{typ string => nat}, e')

    in
      @{term assign} $ var-str $ lam
    end*)
  (* | quote-com (AssignA (b, e)) =
    let
      val var-str = HLogic.mk-string (Binding.name-of b)
      val e' = subst-with-sigma e (*  $\sigma$  "x" ... *)
      val lam = Abs ( $\sigma$ , @{typ string => int}, e')*)

```

```

| quote-com - (AssignA (b, e)) =
  let
    val var-str = HOLogic.mk-string (Binding.name-of b)
    val expr = subst-with-sigma e (* ou autre expression contenant  $\sigma$  "x" *)
    val lam = make-lambda-sigma @{typ string  $\Rightarrow$  int}  $\sigma$  expr
  in
    @{term com.assign} $ var-str $ lam
  end

| quote-com machine (SeqA (c1, c2)) =
  @{term com.seq} $ quote-com machine c1 $ quote-com machine c2

| quote-com machine (LockA b) =
  let
    val n-str = Binding.name-of b
    val n = HOLogic.mk-number @{typ int} (get-lock-id n-str machine)
    (* val n = case Int.fromString n-str of
        SOME i => HOLogic.mk-number @{typ nat} i
      | NONE => error (Invalid lock number: ^n-str) *)
  in
    @{term com.lock :: int  $\Rightarrow$  com} $ n
  end

| quote-com machine (UnlockA b) =
  let
    val n-str = Binding.name-of b
    val n = HOLogic.mk-number @{typ int} (get-lock-id n-str machine)
  in
    @{term com.unlock :: int  $\Rightarrow$  com} $ n
  end

| quote-com - (SendA (sv, e)) =
  let
    val sv-str = HOLogic.mk-string (Binding.name-of sv)
    val expr = subst-with-sigma e
    val lam = make-lambda-sigma @{typ string  $\Rightarrow$  int}  $\sigma$  expr
  in
    @{term com.send :: ( $\sigma \Rightarrow$  int)  $\Rightarrow$  string  $\Rightarrow$  com} $ lam $ sv-str
  end

| quote-com - (ReceiveA (v, sv)) =
  let
    val v-str = HOLogic.mk-string (Binding.name-of v)
    val sv-str = HOLogic.mk-string (Binding.name-of sv)
  in
    @{term com.rec :: string  $\Rightarrow$  string  $\Rightarrow$  com} $ v-str $ sv-str
  end

| quote-com machine (IfelseA ((cond, then-branch), else-branch)) =
  let
    (* val (sigma, e') = subst-with-fresh-env @{typ string  $\Rightarrow$  int} cond *)
    val lam = Abs (fst (Term.dest-Free sigma), @{typ string  $\Rightarrow$  int}, e')*)

```

```

    val expr = subst-with-sigma cond
    val lam = make-lambda-sigma @{typ string ⇒ int} σ expr
  in
    @{term com.cond} $ lam
      $ quote-com machine (sequence-actions then-branch)
      $ quote-com machine (sequence-actions else-branch)
  end

| quote-com machine (WhileA (cond, body)) =
  let
    val expr = subst-with-sigma cond
    val lam = make-lambda-sigma @{typ string ⇒ int} σ expr

    in
      @{term com.while} $ lam
        $ quote-com machine (sequence-actions body)
    end
  end
>

ML
(* Fonction pour déclarer les termes des threads *)
fun declare-thread-terms (thy: theory) (machine : absy) (threads: thread-absy list)
  (varstab: (term option) Symtab.table)
  : theory =
  fold (fn {nom-thread, actions, ...} => fn thy' =>
    let
      val name = Binding.name-of nom-thread
      val bname = Binding.name (name ^ -term)

      (* Convert actions into com *)
      val com-ast = sequence-actions actions
      val com-term = quote-com machine com-ast
      val com-term = Type.constraint @{typ com} com-term

      (* Create a simple definition: thread-name-term = com-term *)
      val lthy = Named-Target.theory-init thy'
      val ((-, term-def), lthy') =
        Local-Theory.define ((bname, NoSyn), ((Binding.empty, []), com-term)) lthy
      val thy'' = Local-Theory.exit-global lthy'

      val - = writeln (✓ Defined command term: ^ Binding.name-of bname)

      (* Optionally, create the semantic term separately *)
      val sem-bname = Binding.name (name ^ -sem)
      val cont = Abs (s, @{typ σ}, @{term SKIP-process})
      val sem-term = @{const Sem0} $ com-term $ cont

      val lthy2 = Named-Target.theory-init thy''
    end
  )

```

```

    val ((-, sem-def), lthy2') =
      Local-Theory.define ((sem-bname, NoSyn), ((Binding.empty, []), sem-term))
  lthy2
    val thy''' = Local-Theory.exit-global lthy2'

    val - = writeln (✓ Defined semantic term: ^ Binding.name-of sem-bname)

  in
    thy'''
  end)
  threads thy
>

```

ML

```

(* Fonction pour générer les termes sémantiques *)
fun generate-terms (absy-data : absy) (thy : theory) : theory =
  let
    val {system, globals-decls, locks-decls, threads-decls, varstab} = absy-data
    val system-name = Binding.name-of system

    (* Fonction pour vérifier l'égalité des termes dans une table *)
    fun term-equals-in-table (tab : (term option) Symtab.table) (b : binding) (t :
    term) : bool =
      case Symtab.lookup tab (Binding.name-of b) of
      SOME (SOME t') => Term.aconv (t, t')
      | - => false

    (* Fonction pour mettre à jour l'état *)
    fun update-state (locstab : (term option) Symtab.table)
      (globtab : (term option) Symtab.table)
      (state : term -> int)
      (x : binding)
      (v : int) : term -> int =
      let
        val x-name = Binding.name-of x
        fun equals-var y =
          case Symtab.lookup locstab x-name of
          SOME (SOME t') => Term.aconv (y, t')
          | - =>
            (case Symtab.lookup globtab x-name of
             SOME (SOME t'') => Term.aconv (y, t'')
             | - => false)
        in
          fn y => if equals-var y then v else state y
        end
      end

    (* Fonction d'évaluation basique d'un terme *)
    fun eval-term (t : term) (state : term -> int) : int =
      case t of

```

```

    Const (@{const-name Groups.zero-class.zero}, -) => 0
  | Const (@{const-name Groups.one-class.one}, -) => 1
  | x => state x (* Évaluation simplifiée, TODO *)

(* Fonction pour convertir une action en terme *)
fun action-to-term (action : a-term)
  (state : term -> int)
  (locstab : term option Symtab.table)
  (globtab : term option Symtab.table)
  : term * (term -> int) =
case action of
  SkipA =>
    (@{term com.SKIP :: com}, state)

  | AssignA (name, assignation) =>
    let
      val v = eval-term assignation state
      val updated-state = update-state locstab globtab state name v
      val name-str = HOLogic.mk-string (Binding.name-of name)
      val assign-term = @{term assign} $ name-str $ assignation
    in
      (assign-term, updated-state)
    end

  | SeqA (SkipA, C) =>
    action-to-term C state locstab globtab

  | SeqA (A, B) =>
    let
      val (term-A, stateA) = action-to-term A state locstab globtab
      val (term-B, stateB) = action-to-term B stateA locstab globtab
      val seq-term = @{term com.seq} $ term-A $ term-B
    in
      (seq-term, stateB)
    end

  | LockA name =>
    let
      val n-str = Binding.name-of name
      val n = HOLogic.mk-number @{typ int} (get-lock-id n-str absy-data)
      val lock-term = @{term com.lock} $ n
    in
      (lock-term, state)
    end

  | UnlockA name =>
    let
      val n-str = Binding.name-of name
      val n = HOLogic.mk-number @{typ int} (get-lock-id n-str absy-data)

```

```

      (* val n = case Int.fromString n-str of
          SOME i => HOLogic.mk-number @{typ nat} i
          | NONE => error (Invalid unlock number: ^ n-str) *)
      val unlock-term = @{term com.unlock} $ n
    in
      (unlock-term, state)
    end

  | - =>
    error (Unsupported action type in action-to-term)

(* État par défaut *)
fun default-state (- : term) = 0

(* Traiter tous les threads *)
val - = List.app (fn {nom-thread, actions, locstab, ...} =>
  case actions of
    [] => writeln (Thread ^ Binding.name-of nom-thread ^ has no actions)
  | - =>
    let
      val - = writeln (== Processing thread ^ Binding.name-of nom-thread ^
==)
      val com-ast = sequence-actions actions
      val (res-term, final-state) = action-to-term com-ast default-state locstab
    varstab
      val - = writeln (✓ Successfully processed thread ^ Binding.name-of
nom-thread)
    in
      ()
    end
  ) threads-decls

(* Déclarer les termes des threads dans la théorie *)
val thy' = declare-thread-terms thy absy-data threads-decls varstab
in
  thy'
end

fun define-cmd (const-name : string) (rhs : term) (thy : theory) : theory =
  let
    val bname = Binding.name const-name (* nom de la constante *)
    val lthy = Named-Target.theory-init thy (* entrée en contexte local *)

    (* definition const-name where const-name = rhs *)
    val ((-, def-thm), lthy') =
      Local-Theory.define ((bname, NoSyn), ((Binding.empty, []), rhs)) lthy

    val thy' = Local-Theory.exit-global lthy' (* on ressort en théorie *)
  end

```

```

    val - = writeln (✓ defined “ ^ const-name ^ ” : ^
                      Syntax.string-of-typ-global thy' (fastype-of rhs))
    val - = writeln ( ^ Syntax.string-of-term-global thy' rhs)
    val - = writeln ( theorem: ^ Thm.string-of-thm-global thy' (snd def-thm))
  in
    thy'
  end
>

```

2.7 Impression

ML_⋄

```
fun mk-sumT (T1, T2) = Type (Sum-Type.sum, [T1, T2])
```

```

val UNIV- = HOLogic.mk-UNIV @ {typ evs}
val empty-evs-set = Const (@ {const-name bot}, @ {typ evs set})

```

```

fun mk-fun-upd (f, x, y) =
  let
    val T-dom = @ {typ string}
    val T-cod = @ {typ int}
  in
    Const (@ {const-name fun-upd},
           (T-dom --> T-cod) --> T-dom --> T-cod --> (T-dom -->
T-cod))
    $ f $ x $ y
  end

```

```

fun make-sigma-term (tab : term option Symtab.table) : term =
  let
    val zero = HOLogic.mk-number @ {typ int} 0
    val base = Abs (-, @ {typ string}, zero)

    fun apply-update (name, SOME value) acc =
      mk-fun-upd (acc, HOLogic.mk-string name, value)
      | apply-update (-, NONE) acc = acc

    val updated = Symtab.fold apply-update tab base
  in
    updated
  end

```

```

val - =
  Outer-Syntax.command command-keyword⋄SYSTEM⋄
    defines System Specification
    (parse-system-spec >>
     (fn system-data => Toplevel.theory (fn thy =>
      let

```

```

*) (* 1. syntactic & semantic checks -----
*)
    val checked = context-check system-data thy
    val thy-sem = semantic-check checked thy

    val lthy0 = Named-Target.theory-init thy-sem

    fun define-cmd ({nom-thread, actions, ...} : thread-absy) lthy =
      let
        val name      = Binding.name-of nom-thread
        val const-bnd = Binding.name (name ^ -cmd)
        val com-term  = quote-com checked (sequence-actions actions)
        val ((-, -), lthy') =
          Local-Theory.define ((const-bnd, NoSyn),
                               ((Binding.empty, []), com-term)) lthy

      (*Affichage*)
      val - = writeln (✓ Defined constant:  ^ Binding.name-of const-bnd)

      val - = writeln (  ^ Syntax.string-of-term-global thy-sem com-term)

      in lthy' end

    val lthy-final = fold define-cmd (#threads-decls checked) lthy0
    val thy'      = Local-Theory.exit-global lthy-final

    (* -----
    *)
    (* 2. Construction du réseau GLOBALVARS -----
    *)
    (* val globals-names : term list =
      map (fn (b, -, -, -) => HOLogic.mk-string (Binding.name-of b))
      (#globals-decls checked)

    val globals-list-term =
      HOLogic.mk-list @{typ string} globals-names

    val globals-mset-term =
      Cspm-API.mk-mset globals-list-term

    (*val sumT = Type (Sum-Type.sum, [@{typ sema-evs}, @{typ vars-ev}])*)
    val procT = Cspm-API.mk-processT @{typ evs}
    val sigma0-term = make-sigma-term (#varstab checked)

    (*val GLOBALVARS-const =
      Const (@{const-name GLOBALVARS}, @{typ string} --> procT)

    val lam-GLOBALVARS =

```

```

Abs (idx, @{typ string}, GLOBALVARS-const $ Bound 0)*)

val GLOBALVARS-partial = Const (@{const-name GLOBALVARS},
@{typ  $\sigma$ } --> @{typ string} --> procT)
val lam-GLOBALVARS = Abs (idx, @{typ string}, GLOBALVARS-partial
$ sigma0-term $ Bound 0)

val temp-test = Cspm-API.mk-MultiInter globals-mset-term lam-GLOBALVARS
val - = writeln ([SYSTEM] global test : \n ^
Syntax.string-of-term-global thy' temp-test)
val - = tracing (\n [DEBUG] Type full-term = ^
Syntax.string-of-typ-global @{theory} (fastype-of
(temp-test) ))

```

```

val processes : term list =
  map (fn name => lam-GLOBALVARS $ name) globals-names
(* globals-names : term list de strings *)

val globals-net-term : term option =
  (case processes of
   [] => NONE
  | [t] => SOME t
  | ts => SOME (foldr1 (fn (t1, t2) => Cspm-API.mk-Inter t1 t2) ts)*)

(* 2. Construction du réseau GLOBALVARS compactifié -----
*)
val globals-names : term list =
  map (fn (b, -, -, -) => HLogic.mk-string (Binding.name-of b))
(#globals-decls checked)

```

```

val globals-list-term =
  HLogic.mk-list @{typ string} globals-names

val globals-mset-term =
  Cspm-API.mk-mset globals-list-term

val procT = Cspm-API.mk-processT @{typ evs}
val sigma0-term = make-sigma-term (#varstab checked)

val GLOBALVARS-partial = Const (@{const-name GLOBALVARS},
@{typ  $\sigma$ } --> @{typ string} --> procT)
val lam-GLOBALVARS = Abs (idx, @{typ string}, GLOBALVARS-partial
$ sigma0-term $ Bound 0)

(* Utilisation de MultiSync avec ensemble d'événements vide pour compactifier
*)
val globals-net-term : term option =

```

```

(case globals-names of
  [] => NONE
  | - =>
    let
      val multisync-const = Const (@{const-name MultiSync},
                                   @ {typ evs set} -->
                                   @ {typ string multiset} -->
                                   @ {typ string => (evs) process} -->
                                   @ {typ (evs) process})
      val compact-term = multisync-const $ empty-evs-set $ globals-mset-term
$ lam-GLOBALVARS
    in
      SOME compact-term
    end)

val - = (case globals-net-term of
  SOME t => writeln ([SYSTEM] réseau GLOBALVARS compactifié
: \n ^
                                   Syntax.string-of-term-global thy' t)
  | NONE => writeln ([SYSTEM] Aucune variable globale déclarée))

(* 3. Construction du réseau SEMAPHORES -----
*)

(* extraire les noms de verrous depuis checked *)

val lock-ids-terms : term list =
  map (fn (b, -, -) =>
    HLogic.mk-number @ {typ int} (get-lock-id (Binding.name-of b)
checked))
  (* get-lock-id : binding -> int *)
  (#locks-decls checked)

val locks-list-term =
  HLogic.mk-list @ {typ int} lock-ids-terms

val locks-mset-term =
  Cspm-API.mk-mset locks-list-term (* {# ... #} *)

val SEMAPHORES-const =
  Const (@{const-name SEMAPHORES}, @ {typ int} --> procT)

val lam-SEMAPHORES =
  Abs (idx, @ {typ int}, SEMAPHORES-const $ Bound 0)
val processes : term list = map (fn name => lam-SEMAPHORES $ name)
lock-ids-terms
  (* globals-names : term list de strings *)

```

```

(* val semaphores-net-term : term option =
  case processes of
  [] => NONE
  | [t] => SOME t
  | ts => SOME (foldr1 (fn (t1, t2) => Cspm-API.mk-Inter t1 t2) ts)*)

val semaphores-net-term : term option =
  (case lock-ids-terms of
  [] => NONE
  | - =>
  let
    val empty-evs-set = Const (@{const-name bot}, @{typ evs set})
    val multisync-const = Const (@{const-name MultiSync},
      @{typ evs set} -->
      @{typ int multiset} -->
      @{typ int => (evs) process} -->
      @{typ (evs) process})
    val compact-term = multisync-const $ empty-evs-set $ locks-mset-term
  $ lam-SEMAPHORES
  in
    SOME compact-term
  end)

(*val semaphores-net-term =
  SOME( Cspm-API.mk-MultiInter- sumT locks-mset-term lam-SEMAPHORES
) *)

(*|NONE => writeln ([SYSTEM] réseau SEMAPHORES :\n Vous
n'avez pas de Locks \n) *)

(* 4. Construction du réseau des THREADS -----
*)

(* val thread-names : string list =
  map (fn {nom-thread, ...} => Binding.name-of nom-thread) (#threads-decls
checked)

val thread-consts : term list =
  map (fn name =>
    Const (Sign.full-name thy (Binding.name (name ^ -cmd)), @{typ com}))
  thread-names

val thread-processes : term list =
  map (fn t => sem-const $ t) thread-consts

```

```

val thread-net-term =
  (case thread-processes of
    [] => error No threads declared
  | [t] => t
  | ts => foldr1 (fn (t1, t2) => Cspm-API.mk-Inter t1 t2) ts)*)
(*fun define-cmd ({nom-thread, actions, ...} : thread-absy) lthy =
  let
    val name      = Binding.name-of nom-thread
    val const-bnd = Binding.name (name ^ -cmd)
    val com-term  = quote-com checked (sequence-actions actions)
    val ((lhs, -), lthy') =
      Local-Theory.define ((const-bnd, NoSyn),
        ((Binding.empty, []), com-term)) lthy
    val const-name = Local-Theory.full-name lthy' const-bnd
    val - = writeln (✓ Defined constant: ^ const-name)
  in
    (const-name, lthy') (* swapped! *)
  end*)
fun define-cmd ({nom-thread, actions, ...} : thread-absy) lthy =
  let
    val name      = Binding.name-of nom-thread
    val const-bnd = Binding.name (name ^ -cmd)
    val com-term  = quote-com checked (sequence-actions actions)

    val - = Thm.ctrm-of (Proof-Context.init-global (Local-Theory.exit-global
lthy)) com-term

    val const-type = fastype-of com-term
    val lhs-const = Free (Binding.name-of const-bnd, const-type)
    val def-eq = Logic.mk-equals (lhs-const, com-term)

    val ((lhs-term, (-, def-thm)), lthy') =
      Specification.definition
        {verbose = true}
        (SOME (const-bnd, NONE, NoSyn)) (* binding option *)
        [] (* parameters list *)
        [] (* premises list *)
        ((Binding.empty, []), def-eq) (* attributes and equation *)
        lthy (* local theory *)

    val const-name = Local-Theory.full-name lthy' const-bnd
    val - = writeln (✓ Defined constant: ^ const-name)
  in
    (const-name, lthy')
  end
  (* 1. Définition des constantes de threads, et sortie du local-theory *)
  val (cmd-const-names, lthy-final) =
    fold-map define-cmd (#threads-decls checked) lthy0

```



```

                                @{typ string multiset} -->
                                @{typ string => (evs) process} -->
                                @{typ (evs) process}
    val compact-term = multisync-const $ empty-evs-set $ mset $
lam
    in
        SOME compact-term
    end)
in
    locals-net-term
    (* (case processes of
        [] => NONE
        | [t] => SOME t
        | ts => SOME (foldr1 (fn (t1, t2) => Cspm-API.mk-Inter t1 t2)
ts))
    *)
end

```

```

(* D. compose Sem ti-cmd || LOCALVARS -----
*)

```

```

val sem-const = Const (@{const-name Sem}, @{typ com} --> procT)
val thread-processes : term list =
    ListPair.map (fn (th-absy, cmd-term) =>
        let
            val sem-proc = sem-const $ cmd-term
            val lvars-net = localvars-net th-absy
        in
            case lvars-net of
                NONE => sem-proc
            | SOME t => Cspm-API.mk-Sync sem-proc UNIV- t (* par-const $
sem-proc $ @{term UNIV} $ t*)
        end)
    (#threads-decls checked, thread-consts)

```

```

(* E. réseau des threads (+ locals) combiné avec ||| -----
*)

```

```

val thread-net-term : term =
    (case thread-processes of
        [] => error No threads declared
        | [t] => t
        | ts => foldr1 (fn (t1, t2) => Cspm-API.mk-Inter t1 t2) ts)

    val - = writeln ([SYSTEM] réseau THREADS : \n ^
        Syntax.string-of-term-global thy' thread-net-term)
    (* 5. Composition globale : (SEMAPHORES ||| GLOBALVARS)
|| UNIV || THREADS *)

```

```

val net1 = (case globals-net-term of
  SOME t => SOME( Cspm-API.mk-Sync (the
semaphores-net-term) empty-evs-set t)
  (* par-const $ thread-net-term $ @{term empty}
$t *)
  |NONE => semaphores-net-term)

```

```

val full-system-term = (case net1 of
  SOME t => Cspm-API.mk-Sync t UNIV-
thread-net-term
  (* par-const $ net1 $ @{term UNIV} $ (t) *)
  |NONE => thread-net-term)

```

```

val - = writeln ([SYSTEM] réseau complet :\n ^
  Syntax.string-of-term-global thy' full-system-term)
(* val - = tracing (\n [DEBUG] Type threads = ^
  Syntax.string-of-typ-global @{theory} (fastype-of
(thread-net-term) ))
val - = tracing (\n [DEBUG] Type lam-GLOBALVARS = ^
  Syntax.string-of-typ-global @{theory} (fastype-of (the
globals-net-term) ))
val - = tracing (\n [DEBUG] Type semaphores-net-term = ^
  Syntax.string-of-typ-global @{theory} (fastype-of (the
semaphores-net-term) ))
val - = tracing (\n [DEBUG] Type full-term = ^
  Syntax.string-of-typ-global @{theory} (fastype-of
(full-system-term) ))*)

```

```

val - = Thm.ctrm-of (Proof-Context.init-global thy') full-system-term
val const-bnd = #system checked
val lthy-sys = Named-Target.theory-init thy'

```

```

val const-type = fastype-of full-system-term
val lhs-const = Free (Binding.name-of const-bnd, const-type)
val def-eq = Logic.mk-equals (lhs-const, full-system-term)

```

```

val ((lhs-term, (-, def-thm)), lthy') =
Specification.definition
{verbose = true}
(SOME (const-bnd, NONE, NoSyn)) (* binding option *)
[] (* parameters list *)
[] (* premises list *)
((Binding.empty, []), def-eq) (* attributes and equation *)
lthy-sys (* local theory *)

```

```

(* Exit to global theory *)
val thy'' = Local-Theory.exit-global lthy'

```

```

    in
      thy''
    end)))
  >

```

2.8 Tests

SYSTEM *S*

```

globals
locks
thread empty:
  actions
thread empty2:
  actions
end;

```

SYSTEM *System-A*

```

globals x:⟨int⟩ = ⟨3 :: int⟩ var1:⟨int⟩ = ⟨4::int⟩ var4:⟨int⟩ = ⟨4::int⟩
locks l:⟨()⟩ l2:⟨()⟩ l3:⟨()⟩
thread t1 :
  any y : ⟨int⟩ = ⟨36 :: int⟩ var2 : ⟨int⟩
  actions
  SKIP;
  LOCK l;
  y <- var1;
  UNLOCK l;
  IF ⟨var1 ≤ (3 :: int)⟩ THEN
    WHILE ⟨var1 ≤ (3 :: int)⟩ DO
      LOCK l2;
      SKIP;
      var2 = ⟨var2 + 1 :: int⟩;
      var1 -> ⟨var2 :: int⟩;
      UNLOCK l2;
    DONE
  ELSE
    SKIP;
  DONE
  y = ⟨y+2 :: int⟩;
  y = ⟨y+2 :: int⟩;
  var1 -> ⟨y :: int⟩;

thread t2 :
  any var2:⟨int⟩ = ⟨(4+6) :: int⟩ y : ⟨int⟩ = ⟨28 :: int⟩
  actions
  SKIP;
  LOCK l;

```

```

    UNLOCK l;
    IF  $\langle \mathcal{P} = (5 :: int) \rangle$  THEN
      WHILE  $\langle x - (8 :: int) < x \rangle$  DO
        LOCK l;
        UNLOCK l;
      DONE
    ELSE
      SKIP;
    DONE
     $y = \langle var2 + \mathcal{P} :: int \rangle$ ;
  thread t3:
    any test :int =  $\langle -\mathcal{P} :: int \rangle$  var-local :int =  $\langle 4 :: int \rangle$ 
    actions
      var-local =  $\langle (4+42) :: int \rangle$ ;
      test =  $\langle 2 * var-local :: int \rangle$ ;
      test =  $\langle test + \mathcal{P} :: int \rangle$ ;
    IF  $\langle x > (6 :: int) \rangle$ 
    THEN
      WHILE  $\langle x > (2 :: int) \rangle$  DO
        test =  $\langle test - \mathcal{P} :: int \rangle$ ;
      DONE
    ELSE
      test =  $\langle test + \mathcal{P} :: int \rangle$ ;
    DONE
end;

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

ML<
val temp = term  $\langle Sem(t1-cmd) \rangle$ 
val temp2 = term  $\langle Sem(t1-cmd) \rangle$ 
val temp3 = term  $\langle (t1-cmd) \rangle$ 
val temp4 = term  $\langle empty2-cmd \rangle$ 

val tem2 = Cspm-API.mk-Det temp temp2
>

ML<
val temp = term  $\langle System-A \rangle$ 

val c = Thm.ctrm-of @{context} temp
>

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.