CIMP

Peter Gammie

August 16, 2018

Abstract

CIMP extends the small imperative language IMP with control non-determinism and constructs for synchronous message passing.

Contents

1 Lifted predicates 1

2 CIMP syntax and semantics 4

  2.1 Syntax ................................................................. 4
  2.2 Process semantics .................................................... 6
  2.3 System steps ......................................................... 9
  2.4 Assertions ............................................................ 10
      2.4.1 Control predicates ............................................ 10
      2.4.2 Invariants ..................................................... 12
      2.4.3 Relating reachable states to the initial programs .......... 13
  2.5 Simple-minded Hoare Logic/VCG for CIMP .......................... 15
      2.5.1 VCG rules .................................................... 17
      2.5.2 Cheap non-interference rules ................................. 20

3 One-place buffer example 22

4 Unbounded buffer example 24

5 Concluding remarks 26

References 28

1 Lifted predicates

Typically we define predicates as functions of a state. The following provide a somewhat comfortable imitation of Isabelle/HOL’s operators.

abbreviation (input)

pred-pair :: ('a ⇒ 'b) ⇒ ('a ⇒ 'c) ⇒ 'a ⇒ 'b × 'c (infixr ⊗ 60) where
a ⊗ b ≡ λs. (a s, b s)
abbreviation (input)
pred-in :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b set) ⇒ 'a ⇒ bool (infix in 50) where
        a in A ≡ λs. a s ∈ A s

abbreviation (input)
pred-subseteq :: ('a ⇒ 'b set) ⇒ ('a ⇒ 'b set) ⇒ 'a ⇒ bool (infix subseteq 50) where
        A subseteq B ≡ λs. A s ⊆ B s

abbreviation (input)
pred-union :: ('a ⇒ 'b set) ⇒ ('a ⇒ 'b set) ⇒ 'a ⇒ 'b set (infixl union 65) where
        a union b ≡ λs. a s ∪ b s

abbreviation (input)
pred-diff :: ('a ⇒ 'b set) ⇒ ('a ⇒ 'b set) ⇒ 'a ⇒ 'b set (infixr diff 65) where
        a diff b ≡ λs. a s − b s

abbreviation (input)
pred-comp :: (('b ⇒ 'c) ⇒ 'a ⇒ 'd) ⇒ ('b ⇒ 'a ⇒ 'c) ⇒ 'a ⇒ 'd (infixl o 55) where
        f o g ≡ λs. f (λb. g b s) s

abbreviation (input)
pred-app :: ('b ⇒ 'a ⇒ 'c) ⇒ ('a ⇒ 'b set) ⇒ 'a ⇒ bool (infixl ⊘ 100) where
        f ⊘ g ≡ λs. f (g s) s

abbreviation (input)
pred-eq :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ bool (infix eq 40) where
        a eq b ≡ λs. a s = b s

abbreviation (input)
pred-neq :: ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ bool (infix neq 40) where
        a neq b ≡ λs. a s ≠ b s

abbreviation (input)
pred-lt :: ('a ⇒ 'b::ord) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ bool (infix lt 40) where
        a lt b ≡ λs. a s < b s

abbreviation (input)
pred-and :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool (infixr and 35) where
        a and b ≡ λs. a s ∧ b s

abbreviation (input)
pred-or :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool (infixr or 30) where
        a or b ≡ λs. a s ∨ b s

abbreviation (input)
pred-not :: ('a ⇒ bool) ⇒ 'a ⇒ bool (not - [40] 40) where
        not a ≡ λs. ¬a s
abbreviation (input)
pred-imp :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool (infixr imp 25) where
  a imp b ≡ λ s. a s → b s

abbreviation (input)
pred-iff :: ('a ⇒ bool) ⇒ ('a ⇒ bool) ⇒ 'a ⇒ bool (infixr iff 25) where
  a iff b ≡ λ s. a s ↔ b s

abbreviation (input)
pred-K :: 'b ⇒ 'b (⟨⟩) where
  ⟨f⟩ ≡ λ s. f

abbreviation (input)
pred-conjoin :: ('a ⇒ bool) list ⇒ 'a ⇒ bool where
  pred-conjoin xs ≡ foldr (and) xs (True)

abbreviation (input)
pred-singleton :: ('a ⇒ 'b) ⇒ 'a ⇒ 'b set where
  pred-singleton x ≡ λ s. {x s}

abbreviation (input)
pred-empty :: ('a ⇒ 'b set) ⇒ 'a ⇒ bool (empty - [40] 40) where
  empty a ≡ λ s. a s = {}

abbreviation (input)
pred-map-empty :: ('a ⇒ ('b ⇒ 'c option)) ⇒ 'a ⇒ bool (map'-empty - [40] 40) where
  map-empty a ≡ λ s. a s = Map.empty

abbreviation (input)
pred-list-null :: ('a ⇒ 'b list) ⇒ 'a ⇒ bool (list'-null - [40] 40) where
  list-null a ≡ λ s. a s = []

abbreviation (input)
pred-null :: ('a ⇒ 'b option) ⇒ 'a ⇒ bool (null - [40] 40) where
  null a ≡ λ s. a s = None

abbreviation (input)
pred-ex :: ('b ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool (binder EXS 10) where
  EXS x. P x ≡ λ s. ∃ x. P x s

abbreviation (input)
pred-all :: ('b ⇒ 'a ⇒ bool) ⇒ 'a ⇒ bool (binder ALLS 10) where
  ALLS x. P x ≡ λ s. ∀ x. P x s

abbreviation (input)
pred-If :: ('a ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'b ((If (-)/ Then (-)/ Else (-)))
where If P Then x Else y ≡ λ s. if P s then x s else y s

2 CIMP syntax and semantics

We define a small sequential programming language with synchronous message passing primitives for describing the individual processes. This has the advantage over raw transition systems in that it is programmer-readable, includes sequential composition, supports a program logic and VCG (§2.5), etc. These processes are composed in parallel at the top-level.

CIMP is inspired by IMP, as presented by Winskel (1993) and Nipkow and Klein (2014), and the classical process algebras CCS (Milner 1980, 1989) and CSP (Hoare 1985). Note that the algebraic properties of this language have not been developed.

As we operate in a concurrent setting, we need to provide a small-step semantics (§2.2), which we give in the style of structural operational semantics (SOS) as popularised by Plotkin (2004). The semantics of a complete system (§2.3) is presently taken simply to be the states reachable by interleaving the enabled steps of the individual processes, subject to message passing rendezvous. We leave a trace or branching semantics to future work.

2.1 Syntax

Programs are represented using an explicit (deep embedding) of their syntax, as the semantics needs to track the progress of multiple threads of control. Each (atomic) basic command (§2.2) is annotated with a location, which we use in our assertions (§2.4.1). These locations need not be unique, though in practice they likely will be.

Processes maintain local states of type state. These can be updated with arbitrary relations of state ⇒ state set with LocalOp, and conditions of type state ⇒ bool are similarly shallowly embedded. This arrangement allows the end-user to select their own level of atomicity.

The sequential composition operator and control constructs are standard. We add the infinite looping construct Loop so we can construct single-state reactive systems; this has implications for fairness assertions.


type-synonym 's bexp = 's ⇒ bool
datatype ('answer, 'location, 'question, 'state) com
  = Request 'location 'state ⇒ 'question 'answer ⇒ 'state ⇒ 'state set
  (§-] Request -
  [0, 70, 70] 71)
  | Response 'location 'question ⇒ 'state ⇒ ('state × 'answer) set
  (§-] Response
  [0, 70] 71)
  | LocalOp 'location 'state ⇒ 'state set
  (§-] LocalOp - [0, 70]
  71)
  | Cond1 'location 'state bexp ('answer, 'location, 'question, 'state) com
  (§-] IF - THEN
  - FI [0, 0] 71)
  | Cond2 'location 'state bexp ('answer, 'location, 'question, 'state) com
  ('answer, 'location, 'question, 'state) com
  (§-] IF -/ THEN
  -/ ELSE -/ FI [0, 0] 71)
We provide a one-armed conditional as it is the common form and avoids the need to discover a label for an internal SKIP and/or trickier proofs about the VCG.

In contrast to classical process algebras, we have local state and distinct send and receive actions. These provide an interface to Isabelle/HOL’s datatypes that avoids the need for binding (ala the π-calculus of Milner (1989)) or large non-deterministic sums (ala CCS (Milner 1980, §2.8)). Intuitively the sender asks a ’question’ with a Request command, which upon rendezvous with a receiver’s Response command receives an ’answer’. The ’question’ is a deterministic function of the sender’s local state, whereas a receiver can respond non-deterministically. Note that CIMP does not provide a notion of channel; these can be modelled by a judicious choice of ’question’.

We also provide a binary external choice operator. Internal choice can be recovered in combination with local operations (see Milner (1980, §2.3)).

We abbreviate some common commands: SKIP is a local operation that does nothing, and the floor brackets simplify deterministic LocalOps. We also adopt some syntax magic from Makarius’s Hoare and Multiquote theories in the Isabelle/HOL distribution.

**abbreviation** SKIP-syn ((↓- / SKIP 70) where

\[
\{l\} \text{SKIP} \equiv \{l\} \text{LocalOp} (\lambda s. \{s\})
\]

**abbreviation** (input) DetLocalOp :: ’location ⇒ (’state ⇒ ’state)

\[
⇒ (’answer, ’location, ’question, ’state) \text{com} (\{↓\} [\cdot]) \text{where}
\]

\[
\{l\} [f] \equiv \{l\} \text{LocalOp} (\lambda s. \{fs\})
\]

**syntax**

-quote :: ’b ⇒ (’a ⇒ ’b) (⇒ [0] 1000)
-antquote :: (’a ⇒ ’b) ⇒ ’b (¬ [1000]) 1000)
-Assign :: ’location ⇒ idt ⇒ ’b ⇒ (’answer, ’location, ’question, ’state) \text{com} ((\{↓\} ’- :=/ -) [0, 0, 70] 71)
-NonDetAssign :: ’location ⇒ idt ⇒ ’b set ⇒ (’answer, ’location, ’question, ’state) \text{com} ((\{↓\} ’- :∈/ -) [0, 0, 70] 71)

**abbreviation** (input) NonDetAssign :: ’location ⇒ (’val ⇒ ’val) ⇒ ’state ⇒ ’state) ⇒

(’state ⇒ ’val set)

⇒ (’answer, ’location, ’question, ’state) \text{com where}

\[
\text{NonDetAssign} l \text{upd} es \equiv \{l\} \text{LocalOp} (\lambda s. \{ \text{upd} \langle e \rangle s | e. e \in es s \})
\]

**translations**

\[
\{l\} \ ‘x := e ⇒ \text{CONST DetLocalOp} l \langle -\text{update-name} x (\lambda -. e)\rangle
\]
parse-translation ∥

let

fun antique-tr i (Const (@{syntax-const -antiquote}, -) $ t as Const (@{syntax-const -antiquote}, -)) $ t) =
  antique-tr i t $ Bound i

| antique-tr i (t $ u) = antique-tr i t $ antique-tr i u
| antique-tr i (Abs (x, T, t)) = Abs (x, T, antique-tr (i + 1) t)

and skip-antiquote-tr i ((c as Const (@{syntax-const -antiquote}, -)) $ t) =
  c $ skip-antiquote-tr i t

| skip-antiquote-tr i t = antique-tr i t;

fun quote-tr [t] = Abs (s, dummyT, antique-tr 0 (Term.incr-boundvars 1 t))

| quote-tr ts = raise TERM (quote-tr, ts);

in ({@{syntax-const -quote}, K quote-tr} end)

2.2 Process semantics

Here we define the semantics of a single process’s program. We begin by defining the type of externally-visible behaviour:

datatype (′answer, ′question) seq-label
  = sl-Internal (τ)
  | sl-Send ′question ′answer (≪-, -≫)
  | sl-Receive ′question ′answer (≫-, ≪)

We define a labelled transition system (an LTS) using an execution-stack style of semantics that avoids special treatment of the SKIPS introduced by a traditional small step semantics (such as Winskel (1993, Chapter 14)) when a basic command is executed. This was suggested by Thomas Sewell; Pitts (2002) gave a semantics to an ML-like language using this approach.

type-synonym (′answer, ′location, ′question, ′state) local-state
  = (′answer, ′location, ′question, ′state) com list × ′state

inductive

small-step :: (′answer, ′location, ′question, ′state) local-state
         ⇒ (′answer, ′question) seq-label
         ⇒ (′answer, ′location, ′question, ′state) local-state ⇒ bool (· → · [55, 0, 56]

where

  Request: [ α = action s; s' ∈ val β s ] ⇒ ( {l} Request action val # cs, s) →_{α, β} (cs, s')

| Response: (s', β) ∈ action α s ⇒ ( {l} Response action # cs, s) →_{α, β} (cs, s')

| LocalOp: s' ∈ R s ⇒ ( {l} LocalOp R # cs, s) →_τ (cs, s')
The representation of evaluation contexts follows Berghofer (2012). This style of operational semantics was originated by Felleisen and Hieb (1992). The following projections operate on local states. These are internal to CIMP and should not appear to the end-user.

**abbreviation** cPGM :: ('answer, 'location, 'question, 'state) local-state ⇒ ('answer, 'location, 'question, 'state) com list where
cPGM ≡ fst

**abbreviation** cLST :: ('answer, 'location, 'question, 'state) local-state ⇒ 'state where
cLST s ≡ snd s

To reason about system transitions we need to identify which basic statement gets executed next. To that end we factor out the recursive cases of the small-step semantics into contexts, which identify the basic-com commands with immediate externally-visible behaviour. Note that non-determinism means that more than one basic-com can be enabled at a time. The representation of evaluation contexts follows Berghofer (2012). This style of operational semantics was originated by Felleisen and Hieb (1992).

**type-synonym** ('answer, 'location, 'question, 'state) ctxt
  = ('answer, 'location, 'question, 'state) com ⇒ ('answer, 'location, 'question, 'state) com

**inductive-set**
  ctxt :: ( ('answer, 'location, 'question, 'state) ctxt
       × ( ('answer, 'location, 'question, 'state) com ⇒ ('answer, 'location, 'question, 'state) com list ) ) set

where
  C-Hole: (id, []) ∈ ctxt
  | C-Loop: (E, fctxt) ∈ ctxt ⇒ (λt. LOOP DO E t OD, λt. fctxt t ⊕ [LOOP DO E t OD]) ∈ ctxt
  | C-Seq: (E, fctxt) ∈ ctxt ⇒ (λt. E t;; u, λt. fctxt t ⊕ [u]) ∈ ctxt
  | C-Choose1: (E, fctxt) ∈ ctxt ⇒ (λt. E t ∪ u, fctxt) ∈ ctxt

7
While we only use this result left-to-right (to decompose a small step into a basic one), this

\[
\text{C-Choose2: } (E, fctxt) \in \text{ctxt} \implies (\lambda t. u \sqcup E t, \text{fctxt}) \in \text{ctxt}
\]

**inductive**

\[
\text{basic-com :: ('answer, 'location, 'question, 'state) com } \Rightarrow \text{bool}
\]

**where**

\[
\text{basic-com } (\{t\} \text{ REQUEST action val})
\]

\[
\text{basic-com } (\{t\} \text{ RESPONSE action})
\]

\[
\text{basic-com } (\{t\} \text{ LOCALOP R})
\]

\[
\text{basic-com } (\{t\} \text{ IF } b \text{ THEN } c \text{ ELSE } c2 \text{ FI})
\]

\[
\text{basic-com } (\{t\} \text{ WHILE } b \text{ DO } c \text{ OD})
\]

We can decompose a small step into a context and a basic-com.

**fun**

\[
\text{decompose-com :: ('answer, 'location, 'question, 'state) com } \Rightarrow \text{Basic-com ('}\text{answer, 'location, 'question, 'state) com list') set}
\]

**where**

\[
\text{decompose-com } (\text{LOOP DO } c1 \text{ OD}) = \{ (c, \lambda t. \text{ LOOP DO } \text{ictxt } t \text{ OD}, \lambda t. \text{fctxt } t @ [\text{LOOP DO } \text{ictxt } t \text{ OD}], c \text{fctxt } t, \text{ictxt } t) \in \text{decompose-com } c1 \}
\]

\[
\text{decompose-com } (c1 \sqcup c2) = \{ (c, \lambda t. \text{ictxt } t ; c2, \lambda t. \text{fctxt } t @ \{c2\}, c \text{fctxt } \text{ictxt } t, \text{ictxt } t, \text{fctxt } t) \in \text{decompose-com } c1 \}
\]

\[
\text{decompose-com } (c1 \sqcup c2) = \{ (c, \lambda t. \text{ictxt } t \sqcup c2, \text{fctxt } t) | c \text{fctxt } \text{ictxt } t, \text{ictxt } t, \text{fctxt } t) \in \text{decompose-com } c1 \}
\]

\[
\text{definition}
\]

\[
\text{decomposeLS :: ('answer, 'location, 'question, 'state) local-state } \Rightarrow \text{Basic-com ('}\text{answer, 'location, 'question, 'state) local-state') set}
\]

**where**

\[
\text{decomposeLS } s \equiv \text{case } \text{cPGM } s \text{ of } c \# - \Rightarrow \text{decompose-com } c | - \Rightarrow \{}
\]

**theorem** context-decompose:

\[
s \overset{\alpha}{\Rightarrow} s' \iff (\exists (c, \text{ictxt, fctxt}) \in \text{decomposeLS } s.
\]

\[
\text{cPGM } s = \text{ictxt } c \# t l (\text{cPGM } s)
\]

\[
\land \text{ basic-com } c
\]

\[
\land (c \# \text{fctxt } c @ t l (\text{cPGM } s), \text{cLST } s) \overset{\alpha}{\Rightarrow} s'
\]

While we only use this result left-to-right (to decompose a small step into a basic one), this
2.3 System steps

A global state maps process names to process’ local states. One might hope to allow processes to have distinct types of local state, but there remains no good solution yet in a simply-typed setting; see Schirmer and Wenzel (2009).

\textbf{type-synonym} ('answer', 'location', 'proc', 'question', 'state) global-state
\[ \text{= 'proc } \Rightarrow ('answer', 'location', 'question', 'state) local-state \]

\textbf{type-synonym} ('proc', 'state) local-states
\[ \text{= 'proc } \Rightarrow 'state \]

An execution step of the overall system is either any enabled internal $\tau$ step of any process, or a communication rendezvous between two processes. For the latter to occur, a Request action must be enabled in process $p1$, and a Response action in (distinct) process $p2$, where the request/response labels $\alpha$ and $\beta$ (semantically) match.

We also track global communication history here to support assertional reasoning (see §2.4).

\textbf{type-synonym} ('answer', 'question) event = 'question \times 'answer
\textbf{type-synonym} ('answer', 'question) history = ('answer', 'question) event list

\textbf{type-synonym} ('answer', 'location', 'proc', 'question', 'state) system-state
\times ('answer', 'question) history

\textbf{inductive-set}
\[ \text{system-step} ::= ( ('answer', 'ls', 'proc', 'question', 'state) system-state } \]
\times ( ('answer', 'ls', 'proc', 'question', 'state) system-state ) set

\textbf{where}
\begin{align*}
\text{LocalStep} & : [ s \ p \rightarrow_{\tau} ls'; s' = s(p := ls'); h' = h ] \Rightarrow ((s, h), (s', h')) \in \text{system-step} \\
\text{CommunicationStep} & : [ s \ p1 \rightarrow_{\alpha, \beta\preceq} ls1'; s \ p2 \rightarrow_{\beta, \alpha \preceq} ls2'; p1 \neq p2; \ s' = s(p1 := ls1', p2 := ls2'); h' = h @ [(\alpha, \beta)] ] \Rightarrow ((s, h), (s', h')) \in \text{system-step}
\end{align*}

\textbf{abbreviation}
\[ \text{system-step-syn} ::= ('answer', 'ls', 'proc', 'question', 'state) system-state } \]
\Rightarrow ( 'answer', 'ls', 'proc', 'question', 'state) system-state \Rightarrow \text{bool} (- s\Rightarrow - [55, 56] \ 55) \]
\textbf{where}
\[ sh \ s\Rightarrow sh' \equiv (sh, sh') \in \text{system-step} \]

\textbf{abbreviation}
\[ \text{system-steps-syn} ::= ('answer', 'ls', 'proc', 'question', 'state) system-state } \]
\Rightarrow ( 'answer', 'ls', 'proc', 'question', 'state) system-state \Rightarrow \text{bool} (- s\Rightarrow* - [55, 56] \ 55) \]
\textbf{where}
\[ \text{sh} \Rightarrow^* \text{sh}' \equiv (\text{sh}, \text{sh}') \in \text{system-step}^* \]

In classical process algebras matching communication actions yield \( \tau \) steps, which aids nested parallel composition and the restriction operation (Milner 1980, \S 2.2). As CIMP does not provide either we do not need to hide communication labels. In CCS/CSP it is not clear how one reasons about the communication history, and it seems that assertional reasoning about these languages is not well developed.

### 2.4 Assertions

We now develop a technique for showing that a CIMP system satisfies a single global invariant, following Lamport (1980); Lamport and Schneider (1984) (and the later Lamport (2002)) and closely related work by Cousot and Cousot (1980) and Levin and Gries (1981), which suggest the incorporation of a history variable. Cousot and Cousot (1980) apparently contains a completeness proof. Lamport mentions that this technique was well-known in the mid-80s when he proposed the use of prophecy variables (see his webpage bibliography). See de Roever, de Boer, Hannemann, Hooman, Lakhnech, Poel, and Zwiers (2001) for an extended discussion of some of this.

Achieving the right level of abstraction is a bit fiddly; we want to avoid revealing too much of the program text as it executes. Intuitively we wish to expose the processes’s present control locations and local states only. Lamport avoids these issues by only providing an axiomatic semantics for his language.

#### 2.4.1 Control predicates

Following Lamport (1980)\(^1\), we define the \( \text{at} \) predicate, which holds of a process when control resides at that location. Due to non-determinism processes can be \( \text{at} \) a set of locations; it is more like “a statement with this location is enabled”, which incidentally handles non-unique locations. Lamport’s language is deterministic, so he doesn’t have this problem. This also allows him to develop a stronger theory about his control predicates.

```plaintext
primrec
  atC :: ('answer, 'location, 'question, 'state) com \Rightarrow 'location \Rightarrow bool
where
  atC (\{}l'\} Request action val) = (\lambda l. \_ \_ l = l')
| atC (\{}l'\} Response action) = (\lambda l. \_ \_ l = l')
| atC (\{}l'\} LocalOp f) = (\lambda l. \_ \_ l = l')
| atC (\{}l'\} IF - THEN - FI \_ \_ \_ = (\lambda l. \_ \_ l = l')
| atC (\{}l'\} IF - THEN - ELSE - FI \_ \_ \_ = (\lambda l. \_ \_ l = l')
| atC (\{}l'\} WHILE - DO - OD \_ \_ \_ = (\lambda l. \_ \_ l = l')
| atC (\{}l'\} LOOP DO c OD \_ \_ \_ = atC c
| atC (c1;; c2 \_ \_ \_ = atC c1
| atC (c1 \_ \_ c2 \_ \_ \_ = (atC c1 or atC c2)
```

primrec atL :: (answer, location, question, state) com list ⇒ location ⇒ bool where
  atL [] = (False)
| atL (c # -) = atC c

abbreviation atLS :: (answer, location, question, state) local-state ⇒ location ⇒ bool where
  atLS ≡ λs. atL (cPGM s)

We define predicates over communication histories and a projection of global states. These are uncurried to ease composition.

type-synonym (location, proc, state) pred-local-state
= proc ⇒ ((location ⇒ bool) × state)

record (answer, location, proc, question, state) pred-state
= local-states :: (location, proc, state) pred-local-state
  hist :: (answer, question) history

type-synonym (answer, location, proc, question, state) pred
= (answer, location, proc, question, state) pred-state ⇒ bool

definition mkP :: (answer, location, proc, question, state) system-state ⇒ (answer, location, proc, question, state) pred-state ⇒ bool where
  mkP ≡ λs p l. fst (local-states s p) l

We provide the following definitions to the end-user.

AT maps process names to a predicate that is true of locations where control for that process resides. The abbreviation at shuffles its parameters; the former is simplifier-friendly and eta-reduced, while the latter is convenient for writing assertions.

definition AT :: (answer, location, proc, question, state) pred-state ⇒ proc ⇒ location ⇒ bool where
  AT ≡ λs p l. at p l s ≡ AT s p l

abbreviation at :: proc ⇒ location ⇒ (answer, location, proc, question, state) pred where
  at p l s ≡ AT s p l

Often we wish to talk about control residing at one of a set of locations. This stands in for, and generalises, the in predicate of Lamport (1980).

definition atS :: proc ⇒ location set ⇒ (answer, location, proc, question, state) pred where
  atS ≡ λp ls s. ∃l∈ls. at p l s

A process is terminated if it not at any control location.

abbreviation terminated :: proc ⇒ (answer, location, proc, question, state) pred where
  terminated p s ≡ ∀l. ¬at p l s

The LST operator (written as a postfix ↓) projects the local states of the processes from a pred-state, i.e. it discards control location information.
Conversely the LSTP operator lifts predicates over local states into predicates over pred-state. Levin and Gries (1981, §3.6) call such predicates universal assertions.

**type-synonym** (′proc, ′state) state-pred
= (′proc, ′state) local-states ⇒ bool

**definition** LST :: (′answer, ′location, ′proc, ′question, ′state) pred-state
⇒ (′proc, ′state) local-states (↓ [1000] 1000) where
s↓ ≡ snd ◦ local-states s

**abbreviation** (input) LSTP :: (′proc, ′state) state-pred
⇒ (′answer, ′location, ′proc, ′question, ′state) pred where
LSTP P ≡ λs. P (LST s)

By default we ask the simplifier to rewrite atS using ambient AT information.

**lemma** atS-state-cong[cong]:
[ [ AT s p = AT s' p ] ] ⇒ atS p ls s ←→ atS p ls s'
by (auto simp: atS-def)

We provide an incomplete set of basic rules for label sets.

**lemma** atS-simps:
¬atS p { } s
atS p {l} s ←→ at p l s
[ [ at p l s; l ∈ ls ] ] ⇒ atS p ls s ←→ True
(∀l. at p l s ←→ l /∈ ls) ⇒ atS p ls s ←→ False
by (auto simp: atS-def)

**lemma** atS-mono:
[ [ atS p ls s; ls ⊆ ls' ] ] ⇒ atS p ls' s
by (auto simp: atS-def)

**lemma** atS-un:
atS p (l ∪ l') s ←→ atS p l s ∨ atS p l' s
by (auto simp: atS-def)

### 2.4.2 Invariants

A complete system consists of one program per process, and a (global) constraint on their initial local states. From these we can construct the set of initial global states and all those reachable by system steps (§2.3).

**type-synonym** (′answer, ′location, ′proc, ′question, ′state) programs
= ′proc ⇒ (′answer, ′location, ′question, ′state) com

**type-synonym** (′answer, ′location, ′proc, ′question, ′state) system
= (′answer, ′location, ′proc, ′question, ′state) programs
× (′proc, ′state) state-pred

12
definition
initial-states :: ('answer, 'location, 'proc, 'question, 'state) system
⇒ ('answer, 'location, 'proc, 'question, 'state) global-state set

where
initial-states sys ≡
{ f . (∀ p. cPGM (f p) = [fst sys p]) ∧ snd sys (cLST ◦ f) }

definition
reachable-states :: ('answer, 'location, 'proc, 'question, 'state) system
⇒ ('answer, 'location, 'proc, 'question, 'state) system-state set

where
reachable-states sys ≡ system-step∗ " (initial-states sys × {[]})"

The following is a slightly more convenient induction rule for the set of reachable states.

lemma reachable-states-system-step-induct[consumes 1,
case-names init LocalStep CommunicationStep]:

assumes r: (s, h) ∈ reachable-states sys
assumes i: ∀ s, s ∈ initial-states sys ⇒ P s []
assumes l: ∀ s h ls’, p. [(s, h) ∈ reachable-states sys; P s h; s p → τ ls’] ⇒ P (s(p := ls’)') h
assumes c: ∀ s h ls1’ ls2’ p1 p2 α β. 
[(s, h) ∈ reachable-states sys; P s h; s p1 →≪α, β≫ ls1’; s p2 →≫α, β≪ ls2’; p1 ≠ p2] ⇒ P (s(p1 := ls1’', p2 := ls2’')) (h @ [⟨α, β⟩])

shows P s h

2.4.3 Relating reachable states to the initial programs

To usefully reason about the control locations presumably embedded in the single global invariant, we need to link the programs we have in reachable state s to the programs in the initial states. The fragments function decomposes the program into statements that can be directly executed (§2.2). We also compute the locations we could be at after executing that statement as a function of the process’s local state.

We could support Lamport’s after control predicate with more syntactic analysis of this kind.

fun
extract-cond :: ('answer, 'location, 'question, 'state) com ⇒ 'state bexp

where
extract-cond (⟨l⟩ IF b THEN c FI) = b
| extract-cond (⟨l⟩ IF b THEN c1 ELSE c2 FI) = b
| extract-cond (⟨l⟩ WHILE b DO c OD) = b
| extract-cond c = (False)

type-synonym ('answer, 'location, 'question, 'state) loc-comp
= ('answer, 'location, 'question, 'state) com
⇒ 'state ⇒ 'location ⇒ bool
fun lconst :: ('location ⇒ bool) ⇒ ('answer, 'location, 'question, 'state) loc-comp where
  lconst lp b s = lp

definition lcond :: ('location ⇒ bool) ⇒ ('location ⇒ bool)
  ⇒ ('answer, 'location, 'question, 'state) loc-comp where
  lcond lp lp' c s ⇔ if extract-cond c s then lp else lp'

fun
  fragments :: ('answer, 'location, 'question, 'state) com
  ⇒ ('location ⇒ bool)
  ⇒ ( ('answer, 'location, 'question, 'state) com
      × ('answer, 'location, 'question, 'state) loc-comp ) set
where
  fragments ( [] ) IF b THEN c FI ) ls
  = { ( [] ) IF b THEN c' FI, lcond (atC c) ls | c'. True } ∪ fragments c ls
  | fragments ( [] ) IF b THEN c1 ELSE c2 FI ) ls
  = { ( [] ) IF b THEN c1' ELSE c2' FI, lcond (atC c1) (atC c2)) | c1' c2'. True } ∪ fragments c1 ls ∪ fragments c2 ls
  | fragments ( LOOP DO c OD ) ls = fragments c (atC c)
  | fragments ( !l WHILE b DO c OD ) ls
  = { ( !l' WHILE b DO c' OD, lcond (atC c) ls | c'. True } ∪ fragments c (λl. l = l')
  | fragments (c1; c2) ls = fragments c1 (atC c2) ∪ fragments c2 ls
  | fragments (c1 ∪ c2) ls = fragments c1 ls ∪ fragments c2 ls
  | fragments c ls = { (c, lconst ls) }

fun
  fragmentsL :: ('answer, 'location, 'question, 'state) com list
  ⇒ ( ('answer, 'location, 'question, 'state) com
      × ('answer, 'location, 'question, 'state) loc-comp ) set
where
  fragmentsL [] = {}
  fragmentsL [c] = fragments c (False)
  fragmentsL (c # c' # cs) = fragments c (atC c') ∪ fragmentsL (c' # cs)

abbreviation
  fragmentsLS :: ('answer, 'location, 'question, 'state) local-state
  ⇒ ( ('answer, 'location, 'question, 'state) com
       × ('answer, 'location, 'question, 'state) loc-comp ) set
where
  fragmentsLS s ⇔ fragmentsL (cPGM s)

Eliding the bodies of IF and WHILE statements yields smaller (but equivalent) proof obligations.
We show that taking system steps preserves fragments.
lemma reachable-states-fragmentsLS:
  assumes (s, h) ∈ reachable-states sys
  shows fragmentsLS (s p) ⊆ fragments (fst sys p) {False}
Decomposing a compound command preserves fragments too.

fun
  extract-inner-locations :: (answer, location, question, state) com ⇒ (answer, location, question, state) com list

where
  extract-inner-locations ([| l |] IF b THEN c FI) cs = lcond (atC c) (atL cs)
| extract-inner-locations ([| l |] IF b THEN c1 ELSE c2 FI) cs = lcond (atC c1) (atC c2)
| extract-inner-locations (LOOP DO c OD) cs = lconst (atC c)
| extract-inner-locations ([| l |] WHILE b DO c OD) cs = lcond (atC c) (atL cs)
| extract-inner-locations c cs = lconst (atL cs)

lemma small-step-extract-inner-locations:
  assumes basic-com c
  assumes (c ≠ cs, ls) → α ls'
  shows extract-inner-locations c cs ls = atLS ls'
using assms by (fastforce split: lcond-splits)

The headline lemma allows us to constrain the initial and final states of a given small step in terms of the original programs, provided the initial state is reachable.

theorem decompose-small-step:
  assumes s p → α ps'
  assumes (s, h) ∈ reachable-states sys
  obtains c cs ls'
    where (c, ls') ∈ fragments (fst sys p) (False)
    and basic-com c
    and ∀ l. atC c l → atLS (s p) l
    and ls' c (cLST (s p)) = atLS ps'
    and (c ≠ cs, cLST (s p)) → α ps'

Reasoning with reachable-states-system-step-induct and decompose-small-step is quite tedious. We provide a very simple VCG that generates friendlier local proof obligations.

2.5 Simple-minded Hoare Logic/VCG for CIMP

We do not develop a proper Hoare logic or full VCG for CIMP: this machinery merely packages up the subgoals that arise from induction over the reachable states (§2.4.2). This is somewhat in the spirit of ?.

Note that this approach is not compositional: it consults the original system to find matching communicating pairs, and aft tracks the labels of possible successor statements. More serious Hoare logics are provided by Cousot and Cousot (1989); Lamport (1980); Lamport and Schneider (1984).

Intuitively we need to discharge a proof obligation for either Requests or Responses but not both. Here we choose to focus on Requests as we expect to have more local information available about these.

inductive
where

Request: \[
\text{\textbf{\textsf{LocalOp}}: } \bigwedge s \; ps' \; s' \; p' \; l' \; \beta \; s' \; p'.
\]
\[
\text{\textbf{\textsf{Cond1}}: } \bigwedge s' \; \text{\textbf{\textsf{Cond2}}: } \bigwedge s' \; \text{\textbf{\textsf{While}}: } \bigwedge s' \; \text{\textbf{\textsf{bool}} (\cdot, \cdot, \cdot =/ \lnot \text{\textsf{\textbackslash{\text{\textbackslash{l}}}}})}
\]

— There are no proof obligations for the following commands.
We abbreviate invariance with one-sided validity syntax.

**abbreviation** `valid-inv` 
\(-, -, - \models \|-/-\) **where**

\(\text{pgms}, \ p, \ \text{aft} \models \{ l \} \equiv \text{pgms}, \ p, \ \text{aft} \models \{ I \} \)

We tweak `fragments` by omitting `Responses`, yielding fewer obligations.

**fun**

\(\text{vcg-fragments}^\prime :: (\text{answer}, \text{location}, \text{question}, \text{state}) \text{ com}
\)

\(\Rightarrow (\text{location} \Rightarrow \text{bool})
\)

\(\Rightarrow (\text{answer}, \text{location}, \text{question}, \text{state}) \text{ com}
\)

\(\times (\text{answer}, \text{location}, \text{question}, \text{state}) \text{ loc-comp} \) **set**

**where**

\(\text{vcg-fragments}^\prime (\{ l \} \text{ Response action}) \text{ ls} = \{ \}
\)

\(\text{vcg-fragments}^\prime (\{ l \} \text{ IF } b \text{ THEN } c \text{ FI} ) \text{ ls}
\)

\(= \text{vcg-fragments}^\prime \text{ c ls}
\)

\(\cup \{ \{ l \} \text{ IF } b \text{ THEN } c \text{ FI, lcond } (\text{atC c} \text{ ls}) | c'. \text{ True} \} \)

\(\text{vcg-fragments}^\prime (\{ l \} \text{ IF } b \text{ THEN } c1 \text{ ELSE } c2 \text{ FI}) \text{ ls}
\)

\(= \text{vcg-fragments}^\prime \text{ c2 ls } \cup \text{vcg-fragments}^\prime \text{ c1 ls}
\)

\(\cup \{ \{ l \} \text{ IF } b \text{ THEN } c1' \text{ ELSE } c2' \text{ FI, lcond } (\text{atC c1}) \text{ (atC c2)}) | c1' c2'. \text{ True} \} \)

\(\text{vcg-fragments}^\prime (\text{LOOP } \text{DO } c \text{ OD}) \text{ ls} = \text{vcg-fragments}^\prime \text{ c (atC c)}
\)

\(\text{vcg-fragments}^\prime (\text{WHILE } b \text{ DO } c \text{ OD}) \text{ ls}
\)

\(= \text{vcg-fragments}^\prime \text{ c (l. l = l') } \cup \{ \{ l \} \text{ WHILE } b \text{ DO } c' \text{ OD, lcond } (\text{atC c} \text{ ls}) | c'. \text{ True} \} \)

\(\text{vcg-fragments}^\prime \text{ (c1 :: c2) ls} = \text{vcg-fragments}^\prime \text{ c2 ls } \cup \text{vcg-fragments}^\prime \text{ c1 (atC c2)}
\)

\(\text{vcg-fragments}^\prime \text{ (c1 \sqcup c2) ls} = \text{vcg-fragments}^\prime \text{ c1 ls } \cup \text{vcg-fragments}^\prime \text{ c2 ls}
\)

\(\text{vcg-fragments}^\prime \text{ c ls} = \{ (c, \text{ lconst ls})\}
\)

**abbreviation**

\(\text{vcg-fragments} :: (\text{answer}, \text{location}, \text{question}, \text{state}) \text{ com}
\)

\(\Rightarrow (\text{answer}, \text{location}, \text{question}, \text{state}) \text{ com}
\)

\(\times (\text{answer}, \text{location}, \text{question}, \text{state}) \text{ loc-comp} \) **set**

**where**

\(\text{vcg-fragments c} \equiv \text{vcg-fragments}^\prime \text{ c (False)}
\)

The user sees the conclusion of \(V\) for each element of \(\text{vcg-fragments}\).

**lemma** \(\text{VCG}:
\)

**assumes** \(\text{R: } s \in \text{reachable-states sys}
\)

**assumes** \(\text{I: } \forall s \in \text{initial-states sys}.\ \text{I (mkP } (s, [])\)

**assumes** \(\text{V: } \forall p. \ \forall (c, \text{ afts}) \in \text{vcg-fragments } (\text{fst sys p}).\ ((\text{fst sys}, p, \text{ afts} \models \{ I \} \ c)
\)

**shows** \(\text{I (mkP s)}\)
2.5.1 VCG rules

We can develop some (but not all) of the familiar Hoare rules; see Lamport (1980) and the seL4/l4.verified lemma buckets for inspiration. We avoid many of the issues Lamport mentions as we only treat basic (atomic) commands.

category

context

fixes pgms :: ('answer, 'location, 'proc, 'question, 'state) programs
fixed p :: 'proc
fixes afts :: ('answer, 'location, 'question, 'state) loc-comp

begin

abbreviation
valid-syn :: ('answer, 'location, 'proc, 'question, 'state) pred
⇒ ('answer, 'location, 'question, 'state) com
⇒ ('answer, 'location, 'proc, 'question, 'state) pred ⇒ bool where
valid-syn P c Q ≡ pgms, p, afts |= \{P\} c \{Q\}
notation valid-syn (\{\_\}/ \_/ \_/ \_/)

abbreviation
valid-inv-syn :: ('answer, 'location, 'proc, 'question, 'state) pred
⇒ ('answer, 'location, 'proc, 'question, 'state) com ⇒ bool where
valid-inv-syn P c ≡ \{P\} c \{P\}
notation valid-inv-syn (\_/ \_/)

lemma vcg-True:
\{P\} c \{⟨True⟩\}
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemma vcg-conj:
\[ \{I\} c \{Q\}; \{I\} c \{R\} \] \implies \{I\} c \{Q and R\}
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemma vcg-pre-imp:
\[ \Rightarrow s. P s \Rightarrow Q s; \{Q\} c \{R\} \] \implies \{P\} c \{R\}
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemmas vcg-pre = vcg-pre-imp[rotated]

lemma vcg-post-imp:
\[ \Rightarrow s. Q s \Rightarrow R s; \{P\} c \{Q\} \] \implies \{P\} c \{R\}
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemma vcg-prop[intro]:
\{\{P\}\} c
by (cases c) (fastforce intro: vcg-intros)+

lemma vcg-drop-imp:

assumes \{P\} c \{Q\}
shows \{P\} c \{R \imp Q\}
using assms
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemma vcg-conj-lift:
  assumes x: \{P\} c \{Q\}
  assumes y: \{P\} c \{Q'\}
  shows \{P \et P'\} c \{Q \et Q'\}
apply (rule vcg-conj)
apply (rule vcg-pre[OF x], simp)
apply (rule vcg-pre[OF y], simp)
done

lemma vcg-disj-lift:
  assumes x: \{P\} c \{Q\}
  assumes y: \{P\} c \{Q'\}
  shows \{P \et P'\} c \{Q \et Q'\}
using assms
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemma vcg-imp-lift:
  assumes \{P'\} c \{not P\}
  assumes \{Q\} c \{Q\}
  shows \{P' \et Q'\} c \{P \imp Q\}
by (simp only: imp-conv-disj vcg-disj-lift[OF assms])

lemma vcg-ex-lift:
  assumes \(\forall x.\ \{P x\} c \{Q x\}\)
  shows \(\{\l s. \exists x.\ P x s\}\ c \{\l s. \exists x.\ Q x s\}\)
using assms
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemma vcg-all-lift:
  assumes \(\forall x.\ \{P x\} c \{Q x\}\)
  shows \(\{\l s. \forall x.\ P x s\}\ c \{\l s. \forall x.\ Q x s\}\)
using assms
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemma vcg-name-pre-state:
  assumes \(\forall s.\ P s \imp (\=) s\) c \{Q\}
  shows \{P\} c \{Q\}
using assms
by (cases c) (fastforce elim!: vcg-inv intro: vcg-intros)+

lemma vcg-lift-comp:
  assumes f: \(\forall P.\ \{\l s.\ P (f s :: 'a :: type)\}\) c
assumes \( P : \forall x. \{ Q x \} \union \{ P x \} \)
shows \( \{ \lambda s . Q (f s) s \} \union \{ \lambda s . P (f s) s \} \)
apply \( \text{rule vcg-name-pre-state} \)
apply \( \text{rename-tac} s \)
apply \( \text{rule vcg-pre} \)
apply \( \text{rule vcg-post-imp[rotated]} \)
apply \( \text{rule vcg-conj-lift} \)
apply \( \text{rule-tac} x = f s \in P \)
apply \( \text{rule-tac} P = \lambda fs. fs = f s \in f \)
apply \( \text{simp} \)
apply \( \text{simp} \)
done

2.5.2 Cheap non-interference rules

These rules magically construct VCG lifting rules from the easier to prove \( eq\text{-imp} \) facts. We don’t actually use these in the GC, but we do derive \( \text{fun-upd} \) equations using the same mechanism. Thanks to Thomas Sewell for the requisite syntax magic.

As these \( eq\text{-imp} \) facts do not usefully compose, we make the definition asymmetric (i.e., \( g \) does not get a bundle of parameters).

definition \( eq\text{-imp} :: (\forall a \Rightarrow b \Rightarrow c) \Rightarrow (\forall b \Rightarrow e) \Rightarrow \text{bool} \)
\( eq\text{-imp} f g \equiv (\forall s s' . (\forall x. f x s = f x s') \rightarrow (g s = g s')) \)

lemma \( eq\text{-impD} : \)
\[ [\text{eq-imp} f g; \forall x . f x s = f x s'] \implies g s = g s' \]
by \( (\text{simp add: eq-imp-def}) \)

lemma \( eq\text{-imp-vcg} : \)
assumes \( g : \text{eq-imp} f g \)
assumes \( f : \forall x P . \{ P \circ (f x) \} \union \{ c \} \)
shows \( \{ P \circ g \} \union \{ c \} \)
apply \( \text{rule vcg-name-pre-state} \)
apply \( \text{rename-tac} s \)
apply \( \text{rule vcg-pre} \)
apply \( \text{rule vcg-post-imp[rotated]} \)
apply \( \text{rule vcg-all-lift[where} \ 'a='a\text{]} \)
apply \( \text{rule-tac} x = x \text{ and} P = \lambda fs. fs = f x s \in f \text{[rule-format]} \)
apply \( \text{simp} \)
apply \( \text{frule eq-impD[where} f=f, OF g\text{]} \)
apply \( \text{simp} \)
apply \( \text{simp} \)
done

lemma \( eq\text{-imp-vcg-LST} : \)
assumes \( g : \text{eq-imp} f g \)
assumes \( f : \forall x P . \{ P \circ (f x) \circ LST \} \union \{ c \} \)

shows \( \{ P \circ g \circ LST \} c \)
apply (rule vcg-name-pre-state)
apply (rename-tac \( s \))
apply (rule vcg-pre)
apply (rule vcg-post-imp[rotated])
apply (rule vcg-all-lift[where \( a = 'a' \)])
apply simp
apply (frule eq-impD[where \( f = f, OF g \)])
apply simp
apply simp
done

lemma eq-imp-fun-upd:
assumes \( g : \text{eq-imp} \ f \ g \)
assumes \( f : \forall x . \ f \ x \ (s(fld := \text{val})) = f \ x \ s \)
shows \( g \ (s(fld := \text{val})) = g \ s \)
apply (rule eq-impD[OF \( g \)])
done

lemma curry-forall-eq:
\[ \forall f . \ P \ f \] = \[ \forall f . \ P \ (\text{case-prod} \ f) \]
apply safe
apply simp-all
apply (rename-tac \( f \))
apply (drule-tac \( x = x \text{ and } P = \lambda s . \text{fs} = f \ x \ s \text{ in} \ f[\text{rule-format}] \))
apply simp
done

lemma pres-tuple-vcg:
\[ (\forall P . \ P \circ (\lambda s . (f \ s, g \ s))) \] c
\[ \leftrightarrow \] \( (\forall P . \ P \circ f \) c \) \( \land \) (\( \forall P . \ P \circ g \) c)
apply (simp add: curry-forall-eq o-def)
apply safe
apply fast
apply fast
apply (rename-tac \( P \))
apply (rule-tac \( f = f\text{ and }P = \lambda s . \text{fs} \) \( g \ s \) \text{ in} \ vcg-lift-comp, simp, simp)
done

lemma pres-tuple-vcg-LST:
\[ (\forall P . \ P \circ (\lambda s . (f \ s, g \ s)) \circ LST) \] c
\[ \leftrightarrow \] \( (\forall P . \ P \circ f \circ LST) \) c \( \land \) (\( \forall P . \ P \circ g \circ LST \) c)
apply (simp add: curry-forall-eq o-def)
apply safe
apply fast
apply fast
apply (rename-tac P)
apply (rule-tac f=λs. f s↓ and P=λfs s. P fs (g s) for g in vcg-lift-comp, simp, simp)
done

lemmas conj-explode = conj-imp-eq-imp-imp
end

3 One-place buffer example

To demonstrate the CIMP reasoning infrastructure, we treat the trivial one-place buffer example of Lamport and Schneider (1984, §3.3). Note that the semantics for our language is different to Lamport and Schneider’s, who treated a historical variant of CSP (i.e., not the one in Hoare (1985)).

We introduce some syntax for fixed-topology (static channel-based) scenarios.

abbreviation
  Receive :: "location ⇒ "channel ⇒ ("val ⇒ "state ⇒ "state)
          ⇒ (unit, "location, "channel × "val, "state) com (\-/- \-\->)
where
  \{l\} ch>f ≡ \{l\} Response (λquest s. if fst quest = ch then \{(f (snd quest) s, ())\} else {})

abbreviation
  Send :: "location ⇒ "channel ⇒ ("state ⇒ "val)
          ⇒ (unit, "location, "channel × "val, "state) com (\-/- \-\->)
where
  \{l\} ch<f ≡ \{l\} Request (λs. (ch, f s)) (λans s. \{s\})

We further specialise these for our particular example.

abbreviation
  Receive' :: "location ⇒ "channel ⇒ (unit, "location, "channel × "state, "state) com (\-/- \-\->)
where
  \{l\} ch>r ≡ \{l\} ch>(λv -. v)

abbreviation
  Send' :: "location ⇒ "channel ⇒ (unit, "location, "channel × "state, "state) com (\-/- \-\->)
where
  \{l\} ch<a ≡ \{l\} ch<\id

These definitions largely follow Lamport and Schneider (1984). We have three processes communicating over two channels. We enumerate program locations.

datatype ex-chname = ξ12 | ξ23
type-synonym ex-val = nat
type-synonym ex-ch = ex-chname × ex-val
datatype ex-loc = r12 | r23 | s23 | s12
datatype ex-proc = p1 | p2 | p3

22
type-synonym \textit{ex-pgm} = (\textit{unit}, \textit{ex-loc}, \textit{ex-ch}, \textit{ex-val}) \textit{com}

type-synonym \textit{ex-pred} = (\textit{unit}, \textit{ex-loc}, \textit{ex-proc}, \textit{ex-ch}, \textit{ex-val}) \textit{pred}

type-synonym \textit{ex-state} = (\textit{unit}, \textit{ex-loc}, \textit{ex-proc}, \textit{ex-ch}, \textit{ex-val}) \textit{global-state}

type-synonym \textit{ex-system} = (\textit{unit}, \textit{ex-loc}, \textit{ex-proc}, \textit{ex-ch}, \textit{ex-val}) \textit{system}

type-synonym \textit{ex-history} = (\textit{ex-ch} \times \textit{unit}) \textit{list}

primrec \textit{ex-pgms} :: \textit{ex-proc} \Rightarrow \textit{ex-pgm}

where
\textit{ex-pgms} \ p_1 = \{ s_{12} \} \ ξ_{12}^<
\textit{ex-pgms} \ p_2 = \text{LOOP DO} \{ r_{12} \} \ ξ_{12}^<;; \{ s_{23} \} \ ξ_{23}^<< \text{OD}
\textit{ex-pgms} \ p_3 = \{ r_{23} \} \ ξ_{23}^<=

Each process starts with an arbitrary initial local state.

abbreviation \textit{ex-init} :: (\textit{ex-proc} \Rightarrow \textit{ex-val}) \Rightarrow \text{bool}

where
\textit{ex-init} \equiv \langle \text{True} \rangle

abbreviation \textit{ex-system} :: \textit{ex-system}

where
\textit{ex-system} \equiv (\textit{ex-pgms}, \textit{ex-init})

PeteG: I don’t understand how Lamport and Schneider justify their invariants.

The following adapts Kai Engelhardt’s, from his notes titled \textit{Proving an Asynchronous Message Passing Program Correct}, 2011. The history variable tracks the causality of the system, which I feel is missing in Lamport’s treatment. We tack on Lamport’s invariant so we can establish \textit{Etern-pred}.

abbreviation \textit{filter-on-channel} :: \textit{ex-chname} \Rightarrow \textit{ex-history} \Rightarrow \textit{ex-val} \textit{list}

where
\textit{filter-on-channel} \ ch \equiv \text{map} \ (\text{snd} \circ \text{fst}) \circ \text{filter} \ ((=) \ ch \circ \text{fst} \circ \text{fst})

definition \textit{Ip1-0} :: \textit{ex-pred}

where
\textit{Ip1-0} \equiv \text{at p}_1 \ s_{12} \text{ imp } (λs. \text{filter-on-channel} \ ξ_{12} \ (\text{hist} \ s) = [])

definition \textit{Ip1-1} :: \textit{ex-pred}

where
\textit{Ip1-1} \equiv \text{terminated p}_1 \text{ imp } (λs. \text{filter-on-channel} \ ξ_{12} \ (\text{hist} \ s) = \text{LST s p}_1)

definition \textit{Ip2-0} :: \textit{ex-pred}

where
\textit{Ip2-0} \equiv \text{at p}_2 \ r_{12} \text{ imp } (λs. \text{filter-on-channel} \ ξ_{12} \ (\text{hist} \ s) = \text{filter-on-channel} \ ξ_{23} \ (\text{hist} \ s))

definition \textit{Ip2-1} :: \textit{ex-pred}

where
\textit{Ip2-1} \equiv \text{at p}_2 \ s_{23} \text{ imp } (λs. \text{filter-on-channel} \ ξ_{12} \ (\text{hist} \ s) = \text{filter-on-channel} \ ξ_{23} \ (\text{hist} \ s))

\text{at} \ [\text{LST s p}_2]

\land \text{LST s p}_1 = \text{LST s p}_2)

definition \textit{Ip3-0} :: \textit{ex-pred}

where
\textit{Ip3-0} \equiv \text{at p}_3 \ r_{23} \text{ imp } (λs. \text{filter-on-channel} \ ξ_{23} \ (\text{hist} \ s) = [])

definition \textit{Ip3-1} :: \textit{ex-pred}

where
\textit{Ip3-1} \equiv \text{terminated p}_3 \text{ imp } (λs. \text{filter-on-channel} \ ξ_{23} \ (\text{hist} \ s) = \text{LST s p}_2)
∧ LST s p1 = LST s p3)

definition I-pred :: ex-pred where
I-pred ≡ pred-conjoin [ Ip1-0, Ip1-1, Ip2-0, Ip2-1, Ip3-0, Ip3-1 ]

lemmas I-defs = Ip1-0-def Ip1-1-def Ip2-0-def Ip2-1-def Ip3-0-def Ip3-1-def

If process three terminates, then it has process one’s value. This is stronger than Lamport and Schneider’s as we don’t ask that the first process has also terminated.

definition Etern-pred :: ex-pred where
Etern-pred ≡ terminated p3 imp (λs. LST s p1 = LST s p3)

Proofs from here down.

lemma correct-system:
  I-pred sh ⇒ Etern-pred sh
apply (clarsimp simp: Etern-pred-def I-pred-def I-defs)
done

lemma p1: ex-pgms, p1, lconst (False) |= {I-pred} s12 ξ12 ∧ λs. s
apply (rule vcgintros)
apply (rename-tac p')
apply (case-tac p')
  apply (auto simp: I-pred-def I-defs atS-def)
done

lemma p2-1: ex-pgms, p2, lconst (λl. l = r12) |={I-pred} s23 ξ23 ∧ λs. s
apply (rule vcgintros)
apply (rename-tac p')
apply (case-tac p')
  apply (auto simp: I-pred-def I-defs atS-def)
done

lemma (s, h) ∈ reachable-states ex-system ⇒ I-pred (mkP (s, h))
apply (erule VCG)
apply (clarsimp simp: I-pred-def I-defs atS-def)
apply simp
apply (rename-tac p)
apply (case-tac p)
  apply auto
  apply (auto simp: p1 p2-1)
done

4 Unbounded buffer example

This is more literally Kai’s example from his notes titled Proving an Asynchronous Message Passing Program Correct, 2011.
datatype `ex-chname = ξ 12 ∣ ξ 23`

type-synonym `ex-val = nat`

type-synonym `ex-ls = ex-val list`

type-synonym `ex-ch = ex-chname ∗ ex-val`

datatype `ex-loc = π 4 ∣ π 5 ∣ c1 ∣ r12 ∣ r23 ∣ s23 ∣ s12`

datatype `ex-proc = p1 ∣ p2 ∣ p3`

type-synonym `ex-pgm = (unit, ex-loc, ex-ch, ex-ls) com`

type-synonym `ex-pred = (unit, ex-loc, ex-proc, ex-ch, ex-ls) pred`

type-synonym `ex-state = (unit, ex-loc, ex-proc, ex-ch, ex-ls) global-state`

type-synonym `ex-system = (unit, ex-loc, ex-proc, ex-ch, ex-ls) system`

type-synonym `ex-history = (ex-ch ∗ unit) list`

FIXME a bit fake: the local state for the producer process contains all values produced.

primrec `ex-pgms :: ex-proc ⇒ ex-pgm where`
  `ex-pgms p1 = LOOP DO \{c1\} LocalOp (λ xs. \{ xs @ [x] | x. True \}) ;; \{s12\} ξ 12-last OD`
  | `ex-pgms p2 = LOOP DO \{r12\} \{ξ 12\}(λ x xs @ [x])`
  | `ex-pgms p3 = LOOP DO \{r23\} \{ξ 23\}(λ x xs @ [x])`
  | `ex-pgms p3 = LOOP DO \{r23\} \{ξ 23\}(λ x xs @ [x])`
  | (λ ans s. \{tl s\}) FI
  | `ex-pgms p3 = LOOP DO \{r23\} \{ξ 23\}(λ x xs @ [x])`

abbreviation `ex-init :: (ex-proc ⇒ ex-ls) ⇒ bool where`
  `ex-init f ≡ ∀ p. f p = []`

abbreviation `ex-system :: ex-system where`
  `ex-system ≡ (ex-pgms, ex-init)`

definition `filter-on-channel :: ex-chname ⇒ ex-history ⇒ ex-val list where`
  `filter-on-channel ch ≡ map (snd ∘ fst) ∘ filter (\(=\) ch ∘ fst ∘ fst)`

lemma `filter-on-channel-simps [simp]`:
  `filter-on-channel ch [] = []`
  `filter-on-channel ch (xs @ ys) = filter-on-channel ch xs @ filter-on-channel ch ys`
  `filter-on-channel ch (((ch', v), resp) ∘ vals) = (if ch' = ch then [v] else []) @ filter-on-channel ch vals`

by (simp-all add: filter-on-channel-def)

definition `Ip1-0 :: ex-pred where`
  `Ip1-0 ≡ λ s. at p1 c1 s → filter-on-channel ξ 12 (hist s) = s bel p1`

definition `Ip1-1 :: ex-pred where`
  `Ip1-1 ≡ λ s. at p1 s12 s → length (s bel p1) > 0 ∧ butlast (s bel p1) = filter-on-channel ξ 12 (hist s)`

definition `Ip1-2 :: ex-pred where`
  `Ip1-2 ≡ λ s. filter-on-channel ξ 12 (hist s) ≤ s bel p1`

definition `Ip2-0 :: ex-pred where`
\[ \text{Ip2-0} \equiv \lambda s. \text{filter-on-channel } \xi \text{12 (hist } s\text{)} = \text{filter-on-channel } \xi \text{23 (hist } s\text{)} \sqcap s\downarrow p2 \]

**definition** Ip2-1 :: ex-pred where
\[ \text{Ip2-1} \equiv \lambda s. \text{at } p2 s23 s \rightarrow \text{length } (s\downarrow p2) > 0 \]

**definition** Ip3-0 :: ex-pred where
\[ \text{Ip3-0} \equiv \lambda s. s\downarrow p3 = \text{filter-on-channel } \xi \text{23 (hist } s\text{)} \]

**definition** I-pred :: ex-pred where
\[ \text{I-pred} \equiv \text{pred-conjoin } [ \text{Ip1-0}, \text{Ip1-1}, \text{Ip1-2}, \text{Ip2-0}, \text{Ip2-1}, \text{Ip3-0} ] \]

**lemmas** I-defs = I-pred-def Ip1-0-def Ip1-1-def Ip1-2-def Ip2-0-def Ip2-1-def Ip3-0-def

The local state of \( p3 \) is some prefix of the local state of \( p1 \).

**definition** Etern-pred :: ex-pred where
\[ \text{Etern-pred} \equiv \lambda s. s\downarrow p3 \leq s\downarrow p1 \]

**lemma** correct-system:
\[ \text{I-pred } s \implies \text{Etern-pred } s \]

**lemma** \( s \in \text{reachable-states ex-system} \implies \text{I-pred } (\text{mkP } s) \)

### 5 Concluding remarks

Previously Nipkow and Prensa Nieto (1999); Prensa Nieto (2002, 2003)\(^2\) have developed the classical Owicki/Gries and Rely-Guarantee paradigms for the verification of shared-variable concurrent programs in Isabelle/HOL. These have been used to show the correctness of a garbage collector (Prensa Nieto and Esparza 2000).

We instead use synchronous message passing, which is significantly less explored. de Boer, de Roever, and Hannemann (1999); de Roever et al. (2001) provide compositional systems for terminating systems. We have instead adopted Lamport’s paradigm of a single global invariant and local proof obligations as the systems we have in mind are tightly coupled and it is not obvious that the proofs would be easier on a decomposed system; see de Roever et al. (2001, §1.6.6) for a concurring opinion.

Unlike the generic sequential program verification framework Simpl (Schirmer 2004), we do not support function calls, or a sophisticated account of state spaces. Moreover we do no meta-theory beyond showing the simple VCG is sound (§2.5).

### References


\(^2\)The theories are in \$ISABELLE/src/HOL/HoareParallel. 


