INF4140 - Models of concurrency

Fall 2016

November 29, 2016

Abstract

This is the "handout" version of the slides for the lecture (i.e., it's a rendering of the content of the slides in a way that does not waste so much paper when printing out). The material is found in [Andrews, 2000]. Being a handout-version of the slides, some figures and graph overlays may not be rendered in full detail, I remove most of the overlays, especially the long ones, because they don't make sense much on a handout/paper. Scroll through the real slides instead, if one needs the overlays.

This handout version also contains more remarks and footnotes, which would clutter the slides, and which typically contains remarks and elaborations, which may be given orally in the lecture.

1 Asynchronous Communication I

14. 11. 2016

Asynchronous Communication: Semantics, specification and reasoning

Where are we?

- part one: shared variable systems
  - programming
  - synchronization
  - reasoning by invariants and Hoare logic
- part two: communicating systems
  - message passing
  - channels
  - rendezvous

What is the connection?

- What is the semantic understanding of message passing?
- How can we understand concurrency?
- How to understand a system by looking at each component?
- How to specify and reason about asynchronous systems?

Overview

Clarifying the semantic questions above, by means of histories:

- describing interaction
- capturing interleaving semantics for concurrent systems
- Focus: asynchronous communication systems without channels

Plan today

- histories from the outside (global) view of components
  - describing overall understanding of a (sub)system

- Histories from the inside (local) view of a component
  - describing local understanding of a single process
- The connection between the inside and outside view
  - the composition rule

What kind of system? Agent network systems

Two flavors of message-passing concurrent systems, based on the notion of:

- processes — without self identity, but with named channels. Channels often FIFO.
- objects (agents) — with self identity, but without channels, sending messages to named objects through a network. In general, a network gives no FIFO guarantee, nor guarantee of successful transmission.

We use the latter here, since it is a very general setting. The process/channel setting may be obtained by representing each combination of object and message kind as a channel.

In the following we consider systems with agents connected by a network!

Programming asynchronous agent systems

New syntax statements for sending and receiving:

- *send statement:* `send` $A : m(e)$ means that the current agent sends message m to agent A where e is an (optional) list of actual parameters.
- *fixed receive statement:* `await` $A : m(w)$ wait for a message m from a specific agent A, and receive parameters in the variable list w. We say that the message is then consumed.
- *open receive statement:* `await` $X?m(w)$ wait for a message m from any agent X and receive parameters in w (consuming the message). The variable X will be set to the agent that sent the message.
- *choice operator* [ ] to select between alternative statement lists, each starting with a receive statement.

Here m is a message name, A the name of an agent, e expressions, X and w variables.

Async. communication constructs

Syntax (s statement list, e expression)

s	::=	`send` $A : m(e)$	send to A
	|	`await` $A : m(w)$	receive from A
	|	`await` $X?m(w)$	receive from someone
	|	`await` $?m(w)$	annonymous receive
	|	$s\,[\,]\,s$	choice
	|	$x := e \mid s;s \mid (s)$	assignment and seq. composition
	|	$if... \mid while... \mid ...$	other statements

Note: As commented by Shukun, the original syntax was not good. Now we use "?" for unknown channels and ":" for known channels.

Channel comm. in Go

- no "named" sender or receiver: go-routines are anonymous
- instead Go uses channel names
- choice operator: `select`
- different syntax (of course):
  - `<- c` : receive over `c`
  - `c <- e` : send `e` over `c`
- similar non-determinism by select-case:

```
    select {                    // comparable to our choice [ ]
      case msg := <-c1:         // receive on c1 and store in msg
           ...
      case msg := <-c2:
           ...
      case msg := <-c3:
           ...
      default:                  // optional branch if
         ...                    // nothing else works
      }
}
```

Example: Coin machine (from Exam 05)

Consider an agent C which changes "5 krone" coins and "1 krone" coins into "10 krone" coins. It receives *five* and *one* messages and sends out *ten* messages as soon as possible, in the sense that the number of messages sent out should equal the total amount of kroner received divided by 10.

We imagine here a fixed user agent U, both producing the *five* and *one* messages and consuming the *ten* messages. The code of the agent C is given below, using b (*balance*) as a local variable initialized to 0.

Example: Coin machine (Cont)

```
loop
  while b < 10
  do
    (await U:five; b:=b+5)
  []
    (await U:one;  b:=b+1)
  od;
  send U:ten;
  b:=b-10                   // use b in next iteration
end
```

- the choice operator [ ][1]
  - selects an *enabled* branch, if any (and otherwise waits)
  - non-deterministic choice if both branches are enabled

Interleaving semantics of concurrent systems

- behavior of a concurrent system: may be described as set of executions,
- each execution: sequence of atomic communication events,
- other names for it: trace, history, execution, (interaction) sequence ...[2]

  Interleaving semantics

  Concurrency is expressed by the set of all *possible interleavings*.
- remember also: "sequential consistency" from the WMM part.
- note: for each interaction sequence, all interactions are ordered sequentially, and their "visible" concurrency

Note: an execution sequence may be graphically pictured by UML message sequence diagrams. Say something about true concurrency, remember

[1] In the literature, also + as notation can often be found. [ ] taken because of "ASCII" version of □, which can be found in publications.

[2] message sequence (charts) in UML etc.

Regular expressions (to express traces)

- very well known and widely used "format" to describe "languages" (= sets of finite "words" over given a given "alphabet")
- formed by
  - $*$ (repetition)
  - $+$ (choice)
  - ; (sequential composition)

(Note: we do not use | for choice, to not confuse with parallel composition.)

A way to describe (sets of) traces

Example 1 (Reg-Expr). • a, b: atomic interactions.

- Assume them to "run" concurrently

$\Rightarrow$ two possible interleavings, described by

$$[[a; b] + [b; a]] \tag{1}$$

Parallel composition of a^* and b^*:

$$(a + b)^* \tag{2}$$

Note:

Remark: notation for reg-expr's
Different notations exist. E.g.: $a|b$ for the *alternative/non-deterministic* choice between a and b. We use $+$ instead

- to avoid confusion with parallel composition
- be consistent with common use of regexp. for describing concurrent behavior

Note: earlier version of this lecture used |.

Safety and liveness & traces

We may let each interaction sequence reflect all interactions in an execution, called the trace, and the set of all possible traces is then called the trace set.

- terminating system: finite traces[3]
- *non-terminating* systems: infinite traces
- trace set semantics in the general case: both finite and infinite traces
- 2 conceptually important classes of properties[4]
  - safety ("nothing wrong will happen")
  - liveness ("something good will happen")

Note: Remark: By means of (sets of) finite sequences, we may only express safety properties, when allowing non-terminating systems. Perhaps remind: partial correctness etc.

[3] Be aware: typically an *infinite* set of finite traces.
[4] Safety etc. it's not a property, it's a "property/class of properties"

Safety and liveness & histories

- often: concentrate on *finite traces*
- reasons
  - conceptually/theoretically simpler
  - connection to run-time monitoring/run-time verification
  - connection to checking (violations of) safety prop's
- our terminology: history = trace up to a given execution point (thus finite)
- Note: In contrast to the book, histories are here finite initial parts of a trace (prefixes)
- sets of histories are **prefix closed**:

  if a history h is in the set, then every prefix (initial part) of h is also in the set.
- sets of histories: can be used capture safety, but not liveness

Simple example: histories and trace set

Consider a system of two agents, A and B, where agent A says "hi-B" repeatedly until B replies "hi-A".

- "sloppy" B: may or may not reply, in which case there will be an infinite trace with only "hi-B" (here comma denotes $\cup$).

  Trace set: $\{[hi_B]^\infty\}, \{[hi_B]^+\,[hi_A]\}$ Histories: $\{[hi_B]^*\}, \{[hi_B]^+\,[hi_A]\}$
- "lazy" B: will reply eventually, but no limit on how long A may need to wait. Thus, each trace will end with "hi_A" after finitely many "hi_B"'s. Trace set: $\{[hi_B]^+\,[hi_A]\}$ Histories: $\{[hi_B]^*\}, \{[hi_B]^+\,[hi_A]\}$
- an "eager" B will reply within a fixed number of "hi_B"'s, for instance before A says "hi_B" three times. Trace set: $\{[hi_B]\,[hi_A]\}, \{[hi_B]\,[hi_B]\,[hi_A]\}$ Histories: $\emptyset, \{\epsilon\}\{[hi_B]\}, \{[hi_B]\,[hi_A]\}, \{[hi_B]\,[hi_B]\}, \{[hi_B]\,[hi_B]\,[hi_A]\}$

> **Note:** With safety we may distinguish the "eager" from the rest, but not the "lazy" from the "sloppy". With liveness we may distinguish all. Since infinite sequences are mathematically and practically complex, we don't consider them.

Histories = sequences of communication events

Let use the following conventions

- communication events $a : \mathit{Event}$ is an event
- set of communication events: $A : 2^{\mathit{Event}}$
- history $h : \mathit{Hist}$

For a given program, the set of (communication) events *Event* is assumed to be fixed.

Definition 2 (Histories)**.** Histories (over the given set of events) is given inductively over the constructors ϵ (empty history) and _; _ (appending of an event to the right of the history)

> **Note:** Terminology: Event here means communication event.

Functions over histories (for specification purpose)

function	type		
ϵ	:	$\rightarrow Hist$	the empty history (constructor)
$\_;\_$	: $Hist * Event$	$\rightarrow Hist$	append right (constructor)
$\#\_$	: $Hist$	$\rightarrow Nat$	length
$\_/\_$	: $Hist * Set$	$\rightarrow Hist$	projection by set of events
$\_ \preceq \_$	: $Hist * Hist$	$\rightarrow Bool$	prefix relation
$\_ \prec \_$	: $Hist * Hist$	$\rightarrow Bool$	strict prefix relation

Inductive definitions (inductive wrt. ε and $\_;\_$):

$\#\epsilon$	$= 0$
$\#(h;x)$	$= \#h + 1$
ϵ/A	$= \epsilon$
$(h;x)/A$	$=$ `if` $x \in A$ `then` $(h/A);x$ `else` (h/A) `fi`
$h \preceq h'$	$= (h = h') \vee h \prec h'$
$h \prec \varepsilon$	$= false$
$h \prec (h';x)$	$= h \preceq h'$

Note: $<$ and $\leq$ denote strict and non-strict prefix-relations:
$h \leq h'$ expresses that sequence h is a prefix (initial part) of h'.

Invariants and Prefix Closed Trace Sets

May use *invariants* to define trace sets:
A (history) invariant I is a predicate over a histories, supposed to hold at all times:

"At any point in an execution h the property $I(h)$ is satisfied"

It defines the following set:

$$\{h \mid I(h)\} \tag{3}$$

- mostly interested in *prefix-closed invariants*!
- a history invariant is historically monotonic:

$$h \leq h' \Rightarrow (I(h') \Rightarrow I(h)) \tag{4}$$

- I history-monotonic $\Rightarrow$ set from equation (3) prefix closed

Remark: A non-monotonic predicate I may be transformed to a monotonic one I':

$$\begin{array}{ll} I'(\varepsilon) & = I(\varepsilon) \\ I'(h';x) & = I(h') \wedge I(h';x) \end{array}$$

Note: Basically, the words prefix-closed and historically monotonic are equivalent

Semantics: Outside view: global histories over events

Consider asynchronous communication by messages from one agent to another: Since message passing may take some time, the sending and receiving of a message m are semantically seen as two distinct atomic interaction events of type Event:

- $A{\uparrow}B{:}m$ denotes that A sends message m to B
- $A{\downarrow}B{:}m$ denotes that B receives (consumes) message m from A

A global history, H, is a finite sequence of such events, requiring that it is legal, i.e.

each reception is preceded by a corresponding send-event.

For instance, the history

$$[(A{\uparrow}B{:}hi), (A{\uparrow}B{:}hi), (A{\downarrow}B{:}hi), (A{\uparrow}B{:}hi), (B{\uparrow}A{:}hi)]$$

is legal and expresses that A has sent "hi" three times and that B has received one of these and has replied "hi".

Note: a concrete message may also have parameters, say messagename(parameterlist) where the number and types of the parameters are statically checked.

Coin Machine Example: Events

$$
\begin{array}{lll}
U{\uparrow}C{:}five & -- & U \text{ sends the message “five” to C} \\
U{\downarrow}C{:}five & -- & C \text{ consumes the message “five”} \\
 & & \\
U{\uparrow}C{:}one & -- & U \text{ sends the message “one to C} \\
U{\downarrow}C{:}one & -- & C \text{ consumes the message “one”} \\
 & & \\
C{\uparrow}U{:}ten & -- & C \text{ sends the message “ten”} \\
C{\downarrow}U{:}ten & -- & U \text{ consumes the message “ten”}
\end{array}
$$

Legal histories

- not all global sequences/histories “make sense”
- depends on the programming language/communciation model
- sometimes called well-definedness, well-formedness or similar
- $legal : Hist \to Bool$

Definition 3 (Legal history)**.**

$$
\begin{array}{ll}
legal(\epsilon) & = true \\
legal(h; (A{\uparrow}B{:}m)) & = legal(h) \\
legal(h; (A{\downarrow}B{:}m)) & = legal(h) \wedge \#(h/\{A{\downarrow}B{:}m\}) < \#(h/\{A{\uparrow}B{:}m\})
\end{array}
$$

where m is message and h a history.

- when m include parameters, legality ensures that the values received are the same as those sent.

Example of legal history (coin machine C and user U):

$$[(U{\uparrow}C{:}five), (U{\uparrow}C{:}five), (U{\downarrow}C{:}five), (U{\downarrow}C{:}five), (C{\uparrow}U{:}ten)]$$

Outside view: logging the global history

How to “calculate” the global history at *run-time:*

- introduce a global variable H,
- initialize: to empty sequence
- for each execution of a send statement in A, update H by

  $$H := H; (A{\uparrow}B{:}m)$$

  where B is the destination and m is the l message
- for each execution of a receive statement in B, update H by

  $$H := H; (A{\downarrow}B{:}m)$$

  where m is the message and A the sender. The message must be of the kind requested by B.

Outside View: Global Properties

specify desired system behavior by predicate I on the global history,

Global invariant

“at any point in an execution H, property $I(H)$ is satisfied”

Enforcement

- run-time logging the history: monitor an executing system. When $I(H)$ is violated we may
  - report it
  - stop the system, or
  - interact with the system (for inst. through fault handling)
- How to prove such properties by analysing the program?
- How can we monitor, or prove correctness properties, component-wise ?

Semantics: Inside view: Local histories

Definition 4 (Local events)**.** Events visible to an agent A, (written α_A = the events local to A:

- $A{\uparrow}B{:}m$: any send-events from A (output from A)
- $B{\downarrow}A{:}m$: any reception by A (input to A)

Definition 5 (Local history)**.** Given a global history: The local history of A, written h_A, is the subsequence of all events visible to A

- Conjecture: Correspondence between global and local view:

$$h_A = H/\alpha_A$$

  i.e. at any point in an execution the history observed locally in A is the projection to A -events of the history observed globally.

- Note: Each event is visible to one, and only one, agent! This will allow compositional reasoning.

Coin Machine Example: Local Events

The events *visible to* C are:

$U{\downarrow}C{:}five$	C consumes the message "five"
$U{\downarrow}C{:}one$	C consumes the message "one"
$C{\uparrow}U{:}ten$	C sends the message "ten"

The events visible to U are:

$U{\uparrow}C{:}five$	U sends the message "five" to C
$U{\uparrow}C{:}one$	U sends the message "one to C
$C{\downarrow}U{:}ten$	U consumes the message "ten"

How to relate local and global views

From global specification to implementation: First, set up the goal of a system: by one or more global histories. Then implement it. For each component: use the global histories to obtain a local specification, guiding the implementation work.

"construction from specifications" From implementation to global specification: First, make or reuse components.

Use the local knowledge for the desired components to obtain global knowledge. Working with invariants: The specifications may be given as invariants over the history.

- Global invariant: in terms of all events in the system
- Local invariant (for each agent): in terms of events visible to the agent

Need *composition* rules connecting local and global invariants.

Example revisited: Coin Machine

```
loop
  while b < 10
  do
    (await U:five; b:=b+5)
  []
    (await U:one;  b:=b+1)
  od;
  send U:ten;
  b:=b-10              // use b in next iteration
end
```

Coin Machine:

interactions *visible to C* (i.e. those that may show up in the local history):

$$\begin{array}{lll} U{\downarrow}C{:}five & -- & C \text{ consumes the message "five"} \\ U{\downarrow}C{:}one & -- & C \text{ consumes the message "one"} \\ C{\uparrow}U{:}ten & -- & C \text{ sends the message "ten"} \end{array}$$

Note: We are interested in a program satisfying the following *local history invariant*:

$$I(h) \hat{=} 0 \leq rec(h) - sent(h) < 10$$

Coin machine example: Loop invariants

Loop invariant for the outer loop:

$$sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 5 \tag{5}$$

where *sum* (the sum of values in the messages) is defined as follows:

$$\begin{array}{lcl} sum(\varepsilon) & = & 0 \\ sum(h;(\ldots:five)) & = & sum(h)+5 \\ sum(h;(\ldots:one)) & = & sum(h)+1 \\ sum(h;(\ldots:ten)) & = & sum(h)+10 \end{array}$$

Loop invariant for the inner loop:

$$sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 15 \tag{6}$$

Histories: from inside to outside view

From local histories to global history: if we know all the local histories h_{Ai} in a system ($i = 1...n$), we have the global knowlegde

$$legal(H) \wedge (\bigwedge_i \ h_{A_i} = H/\alpha_{A_i})$$

i.e. the global history H must be legal and correspond to all the local histories. This may be used to reason about the global history.

Local invariant A_i: a local specification of A_i is given by a predicate on the local history $I_{A_i}(h_{A_i})$ describing a property which holds before all local interaction points. I may have the form of an implication, expressing the output events from A_i depends on a condition on its input events.

From local invariants to a global invariant: if each agent satisfies $I_{A_i}(h_{A_i})$, the total system will satisfy the global invariant:

$$legal(H) \wedge (\bigwedge_i \ I_{A_i}(H/\alpha_{A_i}))$$

Coin machine example: from local to global invariant

before each send/receive: (see eq. (6))

$$sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 15$$

Local Invariant of C in terms of h *alone:*

$$I_C(h) = \exists b.\ (sum(h/\downarrow) = sum(h/\uparrow) + b \ \wedge \ 0 \leq b < 15) \tag{7}$$

$$I_C(h) = 0 \leq sum(h/\downarrow) - sum(h/\uparrow) < 15 \tag{8}$$

For a global history H ($h = H/\alpha_C$) we have:

$$I_C(H/\alpha_C) \Leftrightarrow 0 \leq sum(H/\alpha_C/\downarrow) - sum(H/\alpha_C/\uparrow) < 15 \tag{9}$$

Shorthand notation:

$$0 \leq sum(H/\downarrow C) - sum(H/C\uparrow) < 15$$

Coin machine example: from local to global invariant

- Local invariant of a careful user U (with exact change):

$$I_U(h) = 0 \leq sum(h/\uparrow) - sum(h/\downarrow) \leq 10$$
$$I_U(H/\alpha_U) = 0 \leq sum(H/U\uparrow) - sum(H/\downarrow U) \leq 10$$

- Global invariant of the system U and C:

$$I(H) = legal(H) \wedge I_C(H/\alpha_C) \wedge I_U(H/\alpha_U) \tag{10}$$

implying:

Overall

$$0 \leq sum(H/U\downarrow C) - sum(H/C\uparrow U) \leq sum(H/U\uparrow C) - sum(H/C\downarrow U) \leq 10$$

since $legal(H)$ gives: $sum(H/U\downarrow C) \leq sum(H/U\uparrow C)$ and $sum(H/C\downarrow U) \leq sum(H/C\uparrow U)$.

So, globally, this system will have balance ≤ 10.

Note: This slide then is about the user. The overall picture is how the local invariants give rise to a global one. The coin machine is not careful, but the user is. We don't have code, but we can stipulate a local invariant in the same way as for the coin machine, but here we simply mention the h. The first two equations are two different versions of the same thing, knowing that the local history is a projection of the global one, using the general fact $\alpha_U = \{U\uparrow\} \cup \{\downarrow U\}$.
Afterwards, one can get the global one. The final overall one uses legality.
In the final equation: the sums talk about what:

1. C receives
2. C sends
3. U sends
4. U receives

Coin machine example: Loop invariants (Alternative)

Loop invariant for the outer loop:

$$rec(h) = sent(h) + b \ \wedge \ 0 \leq b < 5$$

where rec (the total amount received) and $sent$ (the total amount sent) are defined as follows:

$$\begin{array}{lcl} rec(\epsilon) & = & 0 \\ rec(h;(U\downarrow C{:}five)) & = & rec(h) + 5 \\ rec(h;(U\downarrow C{:}one)) & = & rec(h) + 1 \\ rec(h;(C\uparrow U{:}ten)) & = & rec(h) \end{array}$$

$$\begin{array}{lcl} sent(\epsilon) & = & 0 \\ sent(h;(U\downarrow C{:}five)) & = & sent(h) \\ sent(h;(U\downarrow C{:}one)) & = & sent(h) \\ sent(h;(C\uparrow U{:}ten)) & = & sent(h) + 10 \end{array}$$

Loop invariant for the inner loop:

$$rec(h) = sent(h) + b \ \wedge \ 0 \leq b < 15$$

Legality

The above definition of legality reflects networks where you may not assume that messages sent will be delivered, and where the order of messages sent need not be the same as the order received.

Perfect networks may be reflected by a stronger concept of legality (see next slide).

Remark 1 (Self-communication may be considered internal:)**.** *In "black-box" specifications, we consider observable events only, abstracting away from internal events. Then, legality of sending may be strengthened:*

$$legal(h;(A\uparrow B{:}m)) = legal(h) \wedge A \neq B$$

Using Legality to Model Network Properties

If the network delivers messages in a FIFO fashion, one could capture this by strengthening the legality-concept suitably, requiring

$$sendevents(h/\downarrow) \preceq h/\uparrow$$

where the projections $h/\uparrow$ and $h/\downarrow$ denote the subsequence of messages sent and received, respectively, and *sendevents* converts receive events to the corresponding send events.

$$\begin{array}{ll} sendevents(\varepsilon) & = \varepsilon \\ sendevents(h;(A{\uparrow}B{:}m)) & = sendevents(h) \\ sendevents(h;(A{\downarrow}B{:}m)) & = sendevents(h);(A{\uparrow}B{:}m) \end{array}$$

Channel-oriented systems can be mimicked by requiring FIFO ordering of communication for each pair of agents:

$$sendevents(h/A\downarrow B) \preceq h/A\uparrow B$$

where $A\downarrow B$ denotes the set of receive-events with A as source and B as destination, and similarly for $A{\uparrow}B$.

2 Asynchronous Communication II

21.11.2016

Overview: Last time

- semantics: histories and trace sets
- specification: invariants over histories
  - global invariants
  - local invariants
  - the connection between local and global histories
- example: Coin machine
  - the main program
  - formulating local invariants

Overview: Today

- Analysis of **send**/**await** statements
- Verifying local history invariants
- example: Coin Machine
  - proving loop invariants
  - the local invariant and a global invariant
- example: Mini bank

Agent/network systems (Repetition)

We consider general agent/network systems:

- Concurrent agents:
  - with self identity
  - no variables shared between agents
  - communication by message passing
- Network:
  - no channels
  - no FIFO guarantee
  - no guarantee of successful transmission

Local reasoning by Hoare logic (a.k.a program logic)

We adapt Hoare logic to reason about local histories in an agent A:

- Introducing a local (logical) variable h, initialized to empty ϵ
  - h represents the *local* history of A
- For **send**/**await**-statement: define the effect on h.
  - extending the h with the corresponding communication event
- *Local reasoning*: we do not know the global invariant
  - For **await**: unknown parameter values
  - For *open receive*: unknown sender

$\Rightarrow$ use non-deterministic assignment

$$x := \texttt{some} \tag{11}$$

Note: As suggested by Daniel, the word "any" might be more appropriate than "some".

Note: In the previous lecture: we had introduced histories, corresponding operators, and invariants over histories. The operations and the concept of invariants were discussed independent from whether it's local or global. For global histories, we wrote H, resp. we said it's a global *variable* (ironically, so to say, for purposes of logics, there is a shared variable, namely H ...), for local ones which are projections from the H are written h_A. But what we have not yet done is "local reasoning". So far we learnt how the global and the local histories (projections) hang together. It's also clear that the projected local history cannot talk about agent local variables.

Local invariant reasoning by Hoare Logic

- each send statement $\texttt{send}\ B : m(\vec{x})$ in A is treated as:

$$h := (h; A{\uparrow}B{:}m(\vec{x})) \tag{12}$$

- each fixed receive statement $\texttt{await}\ B : m(\vec{x})$ in A[5] is treated as

$$\vec{x} := \texttt{some}; h := (h; B{\downarrow}A{:}m(\vec{x})) \tag{13}$$

  the usage of $\vec{x} := \texttt{some}$ expresses that A may receive any values for the receive parameters

- each open receive statement $\texttt{await}\ X?m(\vec{x})$ in A is treated as

$$X := \texttt{some}; \texttt{await}\ X : m(\vec{x}) \tag{14}$$

  i.e., $X := \texttt{some}; \vec{x} := \texttt{some}; h := (h; B{\downarrow}A{:}m(\vec{x}))$, where $X := \texttt{some}$ expresses that A may receive the message from any agent.

Rule for non-deterministic assignments

Non-det assignment

$$\frac{}{\{\ \forall x\ .\ Q\ \}\ x := \texttt{some}\ \{\ Q\ \}}\ \text{ND-Assign}$$

- as said: await/send have been expressed by manipulating h, using non-det assignments

$\Rightarrow$ rules for await/send statements

Note: Not too important: the statement is also known as "havoc"

[5] where $\vec{x}$ is a sequence of variables and X an agent variable

Derived Hoare rules for send and receive

$$\frac{}{\{\ Q[h; A{\uparrow}B{:}m(\vec{x})/h]\ \}\ \texttt{send}\ B : m(\vec{x})\ \{\ Q\ \}}\ \textsc{Send}$$

$$\frac{}{\{\ \forall \vec{x}\ .\ Q[h; B{\downarrow}A{:}m(\vec{x})/h]\ \}\ \texttt{await}\ B : m(\vec{x})\ \{\ Q\ \}}\ \textsc{Receive}_1$$

$$\frac{}{\{\ \forall \vec{x}, X\ .\ Q[h; X{\downarrow}A{:}m(\vec{x})/h]\ \}\ \texttt{await}\ X?m(\vec{x})\ \{\ Q\ \}}\ \textsc{Receive}_2$$

- As before: A is the current agent/object, h the local history
- We assume that neither B nor X occur in $\vec{x}$, and that $\vec{x}$ is a list of distinct variables (which is a static check)
- No shared variables. $\Rightarrow$ no interference, and Hoare reasoning can be done as usual in the sequential setting!
- Simplified version, if no parameters in await:

$$\frac{}{\{\ Q[h/h; (B{\downarrow}A{:}m)]\ \}\ \texttt{await}\ B : m\ \{\ Q\ \}}\ \textsc{Receive}$$

Note: The static distinctness check mentioned is typical for assignments with multiple right-hand-sides.

Hoare rules for local reasoning

The Hoare rule for non-deterministic choice ([ ]) is

Rule for [ ]

$$\frac{\{\ P_1\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P_2\ \}\ S_2\ \{\ Q\ \}}{\{\ P_1 \wedge P_2\ \}\ (S_1\ [\,]\ S_2)\ \{\ Q\ \}}\ \textsc{Nondet}$$

Remark: We may reason similarly backwards over conditionals:[6]

$$\frac{\{\ P_1\ \}\ S_1\ \{\ Q\ \} \qquad \{\ P_2\ \}\ S_2\ \{\ Q\ \}}{\{\ (b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_2)\ \}\ \textbf{if}\ \ b\ \ \textbf{then}\ \ S_1\ \ \textbf{else}\ \ S_2\ \ \textbf{fi}\ \ \{\ Q\ \}}\ \textsc{If}'$$

Note: Alternatively, the precondition can be formulated as **if** b **then** P_1 **else** P_2, using if-then-else both at the statement level and at the boolean expression level (which might be a bit confusing).

Coin machine: local events

Invariants may refer to the local history h, which is the sequence of events visible to C that have occurred so far. The events visible to C are:

$U{\downarrow}C{:}five$ $--$ C consumes the message "five"
$U{\downarrow}C{:}one$ $--$ C consumes the message "one"
$C{\uparrow}U{:}ten$ $--$ C sends the message "ten"

[6] We used actually a *different* formulation for the rule for conditionals. Both formulations are equivalent in the sense that (together with the other rules, in particular Consequence, one can prove the same properties.

Inner loop

let I_i ("inner invariant") abbreviate equation (6)

```
{ I_i }
while b < 10   { b < 10 ∧ I_i }
{ (I_i[(b+5)/b])[h; U↓C:five/h] ∧ (I_i[(b+1)/b])[h; U↓C:one/h] }
do
    ( await U:five;   { I_i[b+5/b] }
      b:=b+5 )    { I_i }
[]
   (await U:one;   b:=b+1)
   { I_i }
od;
```

Must prove the implication:

$$b < 10 \wedge I_i \Rightarrow (I_i[(b+5)/b])[h; U{\downarrow}C:five/h] \wedge (I_i[(b+1)/b])[h; U{\downarrow}C:one/h]$$

Note: From precondition I_i for the loop, we have $I_i \wedge b \geq 10$ as the postcondition to the inner loop.

Outer loop

```
{ I_o }
loop
    { I_o }
    { I_i }
    while b < 10   { b < 10 ∧ I_i }
    { (I_i[(b+5)/b])[h; U↓C:five/h] ∧ (I_i[(b+1)/b])[h; U↓C:one/h] }
    do
        ( await U:five;   { I_i[b+5/b] }
          b:=b+5 )    { I_i }
    []
       (await U:one;   b:=b+1)
       { I_i }
    od;
    { I_i ∧ b ≥ 10 }
    { (I_o[b−10/b])[h; C↑U:ten/h] }
    send U:ten;
    { I_o[b−10/b] }
    b:=b−10
    { I_o }
end
```

Backwards verification.

Outer loop (2)

Verification conditions (as usual):

- $I_o \Rightarrow I_i$, and
- $I_i \wedge b \geq 10 \Rightarrow (I_o[(b-10)/b])[h; C{\uparrow}U:ten/h]$
- I_o holds initially since $h = \varepsilon \wedge b = 0 \Rightarrow I_o$

Local history invariant

For each agent (A):

- Predicate $I_A(h)$ over the local communication history (h)
- Describes interactions between A and the surrounding agents
- Must be maintained by *all* history extensions in A
- Last week: Local history invariants for the different agents may be composed, giving a global invariant

Verification idea: "induction":

Init: Ensure that $I_A(h)$ holds initially (i.e., with $h = \varepsilon$)

Preservation: Ensure that $I_A(h)$ holds after each `send`/`await`-statement, assuming that $I_A(h)$ holds before each such statement

Local history invariant reasoning

- to prove properties of the code in agent A
- for instance: loop invariants etc
- the conditions may refer to the local state $\vec{x}$ (a list of variables) and the local history h, e.g., $Q(\vec{x}, h)$.

The local history invariant $I_A(h)$

- must hold immediately *after* each send/receive

$\Rightarrow$ if reasoning gives the condition $Q(v, h)$ immediately after a send or receive statement, we basically need to ensure:

$$Q(\vec{x}, h) \Rightarrow I_A(h) \tag{15}$$

- we may assume that the invariant is satisfied immediately *before* each send/receive point.
- we may also assume that the *last* event of h is the send/receive event.

Note: Remark: the local history invariant *cannot* talk about internal variables (except h, if we think of it as a local variable). The last remark on the slide, about the last element of the history, we will see what that means on the following slides.

Proving the local history invariant

We get 3 verification conditions (+ one for the "beginning")

$$I_A(\epsilon) \tag{16}$$

$$(\ h = (h'; A{\uparrow}B{:}m(e)) \wedge I_A(h') \wedge Q(\vec{x}, h)\ ) \Rightarrow I_A(h) \tag{17}$$

$$(\ h = (h'; B{\downarrow}A{:}m(\vec{y})) \wedge I_A(h') \wedge Q(\vec{x}, h)\ ) \Rightarrow I_A(h) \tag{18}$$

$$(\ h = (h'; X{\downarrow}A{:}m(\vec{y})) \wedge I_A(h') \wedge Q(\vec{x}, h)\ ) \Rightarrow I_A(h) \tag{19}$$

in all three cases: Q is the condition right after the send-, resp. the await-statement, and

- $I_A(\_)$: local history invariant of A
- first conjunct $h = ...$: specifies last communication step
- $I_A(h')$: assumption that invariant holds before the comm.-statement
- 3 communication/sync. statements to consider: `send` $B : m(e)$, `await` $B : m(\vec{y})$, and `await` $X?m(\vec{y})$

Coin machine example: local history invariant

For the coin machine C, consider the local history invariant $I_C(h)$ from last week (see equation (8)):

$$I_C(h) = 0 \leq sum(h/{\downarrow}) - sum(h/{\uparrow}) < 15$$

Consider the statement `send` $U : ten$ in C

- Hoare analysis of the outer loop gave the condition $I_o[(b-10)/b]$ immediately after the statement
- history *ends* with the event $C{\uparrow}U{:}ten$

$\Rightarrow$ Verification condition, corresponding to equation (17):

$$h = h'; (C{\uparrow}U{:}ten) \wedge I_C(h') \wedge I_o[(b-10)/b] \Rightarrow I_C(h) \tag{20}$$

Note: The local history invariant I_C was given before, on slide 9. Remember that the history invariant is not allowed to mention local variables such as b. That was done, however, for the loop-invariants (we had 2 loop invariants, one for the inner and one for the outer loop. There was also quite some discussion in the exercises.

Coin machine example: local history invariant

Expanding I_c and I_o in the VC from equation (20), gives:

$$\begin{array}{l} h = h'; (C{\uparrow}U : ten) \wedge \\ I_C(h') \wedge \\ I_o[(b-10)/b] \\ \quad \Rightarrow I_C(h) \\ \hline h = h'; (C{\uparrow}U : ten) \wedge \\ (0 \leq sum(h'/{\downarrow}) - sum(h'/{\uparrow}) < 15) \wedge \\ (sum(h/{\downarrow}) = sum(h/{\uparrow}) + b - 10 \wedge 0 \leq b - 10 < 5) \\ \quad \Rightarrow 0 \leq sum(h/{\downarrow}) - sum(h/{\uparrow}) < 15 \\ \hline (b = (sum(h/{\downarrow}) - sum(h/{\uparrow}) \wedge \\ 0 \leq b - 10 < 5) \\ \quad \Rightarrow 0 \leq b - 10 < 15 \end{array}$$

Note: Gray parts not needed here.

Coin Machine Example: Summary

Correctness proofs (bottom-up):

- code
- loop invariants (Hoare analysis)
- local history invariant
- verification of local history invariant based on the Hoare analysis

Note: The [ ]-construct was useful (basically necessary) for programming service-oriented systems, and had a simple proof rule.

Example: "Mini bank" (ATM): Informal specification

Client cycle: The client C is making these messages

- put in card, give pin, give amount to withdraw, take cash, take card

Mini Bank cycle: The mini bank M is making these messages

to client: ask for pin, ask for withdrawal, give cash, return card

to central bank: request of withdrawal

Central Bank cycle: The central bank B is making these messages

to mini bank: grant a request for payment, or deny it

There may be many mini banks talking to the same central bank, and there may be many clients using each mini bank (but the mini bank must handle one client at a time).

Mini bank example: Global histories

Consider a client C, mini bank M and central bank B: Example of successful cycle:
$[\ C{\updownarrow}M : card\_in(n); M{\updownarrow}C : pin; C{\updownarrow}M : pin(x);\quad M{\updownarrow}C : amount; C{\updownarrow}M : amount(y); M{\updownarrow}B : request(n,x,y); B{\updownarrow}M : grant; M{\updownarrow}C : cash(y); M{\updownarrow}C : card\_out\ ]$
where n is name, x pin code, and y cash amount, provided by clients. Example of unsuccessful cycle: $[\ C{\updownarrow}M : card\_in(n); M{\updownarrow}C : amount; C{\updownarrow}M : amount(y); M{\updownarrow}B : request(n,x,y); B{\updownarrow}M : deny;\quad M{\updownarrow}C : card\_out\ ]$
Notation: $A{\updownarrow}B : m$ denotes the sequence $A{\uparrow}B : m; A{\downarrow}B : m$

Mini bank example: Local histories (1)

From the global histories above, we may extract the corresponding local histories: The successful cycle:

- Client: $[C{\uparrow}M : card\_in(n); M{\downarrow}C : pin; C{\uparrow}M : pin(x);\quad M{\downarrow}C : amount; C{\uparrow}M : amount(y); M{\downarrow}C : cash(y); M{\downarrow}C : card\_out]$
- Mini Bank: $[C{\downarrow}M : card\_in(n); M{\uparrow}C : pin; C{\downarrow}M : pin(x);\quad M{\uparrow}C : amount; C{\downarrow}M : amount(y); M{\uparrow}B : request(n,x,y); B{\downarrow}M : grant; M{\uparrow}C : cash(y); M{\uparrow}C : card\_out]$
- Central Bank: $[M{\downarrow}B : request(n,x,y); B{\uparrow}M : grant]$

The local histories may be used as guidelines when implementing the different agents.

Mini bank example: Local histories (2)

The unsuccessful cycle:

- Client: $[C{\uparrow}M{:}card\_in(n); M{\downarrow}C{:}pin; C{\uparrow}M{:}pin(x); \quad M{\downarrow}C{:}amount; C{\uparrow}M{:}amount(y); M{\downarrow}C{:}card\_out]$
- Mini Bank: $[C{\downarrow}M{:}card\_in(n); M{\uparrow}C{:}pin; C{\downarrow}M{:}pin(x); \quad M{\uparrow}C{:}amount; C{\downarrow}M{:}amount(y); M{\uparrow}B{:}request(n,x,y); B{\downarrow}M{:}deny; M{\uparrow}C{:}card\_out]$
- Central Bank: $[M{\downarrow}B{:}request(n,x,y); B{\uparrow}M{:}deny]$

Note: many other executions possible, say when clients behaves differently, difficult to describe all at a global level (remember the formula of week 1).

Mini bank example: implementation of Central Bank

Sketch of simple central bank. **Program variables:**

```
pin -- array of pin codes, indexed by client names
bal -- array of account balances, indexed by client nam

X : Agent, n:  Client_Name, x:  Pin_Code, y:  Natural
```

```
Loop
  await X?request(n,x,y);
  if    pin[n]=x and bal[n]>y
  then  bal[n]:=bal[n]-y;
        send  X:grant;
  else send X:deny
  fi
end
```

Note: the mini bank X may vary with each iteration.

Mini bank example: Central Bank (B)

Consider the (extended) regular expression $Cycle_B$ defined by:

$$[\ X{\downarrow}B{:}request(n,x,y); [\ B{\uparrow}X{:}grant\ +\ B{\uparrow}X{:}deny\ ]\ \textbf{some}\ X,n,x,y\ ]^*$$

- with + for choice, $[...]^*$ for repetition
- Defines cycles: *request* answered with either *grant* or *deny*
- notation $[regExp\ \textbf{some}\ X,n,x,y]^*$ means that the values of X, n, x, and y are fixed in each cycle, but may vary from cycle to cycle.

Notation: Given an extended regular expression R. Let h **is** R denote that h matches the structure described by R. Example (for events a, b, and c):

- we have $(a;b;a;b)$ **is** $[a;b]^*$
- we have $(a;c;a;b)$ **is** $[a;[b+c]]^*$
- we do *not* have $(a;b;a)$ **is** $[a;b]^*$ (but it is a prefix)

Loop invariant of Central Bank (B): Let $Cycle_B$ denote the regular expression:

$$[\ X{\downarrow}B{:}request(n,x,y); [\ B{\uparrow}X{:}grant\ +\ B{\uparrow}X{:}deny\ ]\ \textbf{some}\ X,n,x,y\ ]^*$$

Loop invariant: h **is** $Cycle_B$

Proof of loop invariant (entry condition): Must prove that it is satisfied initially: ε **is** $Cycle_B$, which is trivial.

Proof of loop invariant (invariance):

loop $\{h$ **is** $Cycle_B\}$
  await `X?request(n,x,y);`
  if `pin[n]=x and bal[n]>y`
    then `bal[n]:=bal[n]-y;` **send** `X:grant;`
    else send `X:deny`
  fi
$\{h$ **is** $Cycle_B\}$
end

Loop invariant of the central bank (B):

```
loop
  { h is Cycle_B }
  { ∀ X,n,x,y . if pin[n] = x ∧ bal[n] > y then h''_1 is Cycle_B else h''_2 is Cycle_B }
  await X?request(n,x,y);
    { if pin[n] = x ∧ bal[n] > y then h'_1 is Cycle_B else h'_2 is Cycle_B }
  if    pin[n]=x and bal[n]>y
  then  bal[n]:=bal[n]-y;
        { (h; B↑X:grant) is Cycle_B }
        send X:grant;
  else  { (h; B↑X:deny) is Cycle_B }
        send X:deny;
  fi
  { h is Cycle_B }
end
```

$$\begin{aligned} h''_1 &= h; X{\downarrow}B{:}request(n,x,y); B{\uparrow}X{:}grant \\ h'_1 &= h; B{\uparrow}X{:}grant \end{aligned}$$

Analogously (with *deny*) for h'_2 and h''_2

Hoare analysis of central bank loop (cont.)

$$h \,\mathbf{is}\, Cycle_B \Rightarrow \forall X,n,x,y\,.\,\mathbf{if}\ pin[n] = x \wedge bal[n] > y$$

Verification condition:

$$\mathbf{then}\ (h; X{\downarrow}B{:}request(n,x,y); B{\uparrow}X{:}grant)\,\mathbf{is}\,Cycle_B$$
$$\mathbf{else}\ \ (h; X{\downarrow}B{:}request(n,x,y); B{\uparrow}X{:}deny)\,\mathbf{is}\,Cycle_B$$

where $Cycle_B$ is

$$[\ X{\downarrow}B{:}request(n,x,y); [\ B{\uparrow}X{:}grant\ +\ B{\uparrow}X{:}deny\ ]\ \mathbf{some}\ X,n,x,y\ ]^*$$

The condition follows by the general rule (regExp R and events a and b):

$$h\ \mathbf{is}\ R^* \wedge (a;b)\ \mathbf{is}\ R \Rightarrow (h;a;b)\ \mathbf{is}\ R^*$$

since $(X{\downarrow}B{:}request(n,x,y); B{\uparrow}X{:}grant)\,\mathbf{is}\,Cycle_B$ and $(X{\downarrow}B{:}request(n,x,y); B{\uparrow}X{:}deny)\,\mathbf{is}\,Cycle_B$

Local history invariant for the central bank (B)

$Cycle_B$ is

$$[\ X{\downarrow}B{:}request(n,x,y); [\ B{\uparrow}X{:}grant\ +\ B{\uparrow}X{:}deny\ ]\ \mathbf{some}\ X,n,x,y\ ]^*$$

Define the history invariant for B by:

$$h \leq Cycle_B$$

Let $h \leq R$ denote that h is a prefix of the structure described by R.

- intuition: if $h \leq R$ we may find some extension h' such that $(h;h')$ **is** R
- h **is** $R \Rightarrow h \leq R$ (but not vice versa)
- $(h;a)$ **is** $R \Rightarrow h \leq R$
- Example: $(a;b;a) \ \leq\ [a;b]^*$

Central Bank: Verification of the local history invariant

$h \leq Cycle_B$

- As before, we need to ensure that the history invariant is implied after each send/receive statement.
- Here it is enough to assume the conditions after each send/receive statement in the verification of the loop invariant

This gives 2 proof conditions:

1. **after send grant/deny** (i.e. after **fi**)
   h **is** $Cycle_B \Rightarrow h \leq Cycle_B$ which is trivial.
2. **after await request**
   if ... **then** $(h; B{\uparrow}X{:}grant)\,\mathbf{is}\,Cycle_B$ **else** $(h; B{\uparrow}X{:}deny)\,\mathbf{is}\,Cycle_B$
   $\Rightarrow h \leq Cycle_B$ which follows from $(h;a)\,\mathbf{is}\,R \Rightarrow h \leq R$.

Note: We have now proved that the implementation of B satisfies the local history invariant, $h\ \leq Cycle_B$.

Mini bank example: Local invariant of Client (C)

$Cycle_C$: [$C{\uparrow}X{:}card\_in(n)$ + $X{\downarrow}C{:}pin; C{\uparrow}X{:}pin(x)$ + $X{\downarrow}C{:}amount; C{\uparrow}X{:}amount(y')$ + $X{\downarrow}C{:}cash(y)$ + $X{\downarrow}C{:}card\_out$ **some** X, y, y']*

History invariant:

$$h_C \leq Cycle_C$$

Note: The values of C, n and x are fixed in each cycle.

Note: The client is willing to receive cash and cards, and give card, at any time, and will respond to *pin*, and *amount* messages from a mini bank X in a sensible way, without knowing the protocol of the particular mini bank. This is captured by + for different choices.

Mini bank example: Local invariant for Mini bank (M)

$Cycle_M$: [$C{\downarrow}M{:}card\_in(n); M{\uparrow}C{:}pin; C{\downarrow}M{:}pin(x);$ $M{\uparrow}C{:}amount; C{\downarrow}M{:}amount(y);$ **if** $y \leq 0$ **then** ε **else** $M{\uparrow}B{:}request(n,x,y); [B{\downarrow}M{:}deny$ + $B{\downarrow}M{:}grant; M{\uparrow}C{:}cash(y)$] **fi** ; $M{\uparrow}C{:}card\_out$ **some** C, n, x, y]*

History invariant:

$$h_M \leq Cycle_M$$

Note: communication with a fixed central bank. The client may vary with each cycle.

Mini bank example: obtaining a global invariant

Consider the parallel composition of C, B, M. Global invariant:
$legal(H) \wedge H/\alpha_C \leq Cycle_C \wedge H/\alpha_M \leq Cycle_M \wedge H/\alpha_B \leq Cycle_B$

Assuming no other agents, this invariant may *almost* be formulated by:
$H \leq [C{\updownarrow}M : card\_in(n); M{\updownarrow}C : pin; C{\updownarrow}M : pin(x);$ $M{\updownarrow}C : amount; C{\updownarrow}M : amount(y);$ **if** $y \leq 0$ **then** $M{\updownarrow}C : card\_out$ **else** $M{\updownarrow}B : request(n,x,y); [B{\updownarrow}M : deny; M{\updownarrow}C : card\_out$ + $B{\updownarrow}M : grant; M{\uparrow}C{:}cash(y); [M{\downarrow}C{:}cash(y)$ ||| $M{\updownarrow}C : card$ **some** n, x, y]*

where ||| gives all possible interleavings. However, we have no guarantee that the cash and the card events are received by C before another cycle starts. Any next client may actually take the cash of C.

For proper clients it works OK, but improper clients may cause the Mini Bank to misbehave. Need to incorporate assumptions on the clients, or make an improved mini bank.

Improved mini bank based on a discussion of the global invariant

The analysis so far has discovered some weaknesses:

- The mini bank does not know when the client has taken his cash, and it may even start a new cycle with another client before the cash of the previous cycle is removed. This may be undesired, and we may introduce a new event, say *cash_taken* from C to M, representing the removal of cash by the client. (This will enable the mini bank to decide to take the cash back within a given amount of time.)
- A similar discussion applies to the removal of the card, and one may introduce a new event, say *card_taken* from C to M, so that the mini bank knows when a card has been removed. (This will enable the mini bank to decide to take the card back within a given amount of time.)
- A client may send improper or unexpected events. These may be lying in the network unless the mini bank receives them, and say, ignores them. For instance an old misplaced amount message may be received in (and interfere with) a later cycle. An improved mini bank could react to such message by terminating the cycle, and in between cycles it could ignore all messages (except card_in).

Summary

Concurrent agent systems, without network restrictions (need not be FIFO, message loss possible).

- Histories used for semantics, specification and reasoning
- correspondence between global and local histories, both ways
- parallel composition from local history invariants
- extension of Hoare logic with send/receive statements
- avoid interference, may reason as in the sequential setting
- Bank example, showing
  - global histories may be used to exemplify the system, from which we obtain local histories, from which we get useful coding help
  - specification of local history invariants
  - verification of local history invariants from Hoare logic + verification conditions (one for each send/receive statement)
  - composition of local history invariants to a global invariant

References

[Andrews, 2000] Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley.