IN5170/IN9170 – Models of concurrency

Fall 2021

Compiled: November 11, 2021

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 [**?**]. 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.

Contents

1 Asynchronous Communication I

10.11.2021

Asynchronous Communication

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*?
  - $\Rightarrow$ *scalability*
- How to specify and reason about asynchronous systems?
- in particular, *compositional reasoning*.

Overview for today

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

- agent systems
- 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 or process 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. Examples: Go, CSP
- 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 next look at

- Go, briefly
- agent systems, in more detail including reasoning aspects
  - compositional (modular) reasoning

Channel Communication in Go

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

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

1.1 Agent systems

Agent Systems

A High-Level Introduction

- syntax
- examples
- semantics and reasoning

Motivation

- We focus on agent systems, 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.

Programming asynchronous agent systems

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, similar to **select** in Go ("angelic choice")

Here m is a message name, A an agent expression, X an agent reference, e an expression (list), and w a variable (list).

Async. communication constructs

Syntax (s statement list, e expression/expression list)

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
	|	$x := e \mid s; s \mid (s)$	assign., seq.composition
	|	$m(e)$	local call to method m
	|	`if` e `then` s `else` s `fi`	if statement
	|	`while` e `do` s `od` | `loop` s `end`	loop statements

1.2 Example: Coin machine

Example: Coin machine (from Exam 05)

Consider an agent C which

- changes "5 krone" coins and "1 krone" coins into "10 krone" coins.
- 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

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

1.3 Histories

Interleaving semantics of concurrent systems

- The behavior of a concurrent system: may be described as a set of executions,
- each execution: sequence of atomic communication events,
- other names for it: trace, history, execution, (interaction) sequence — similar to message sequence (charts) in UML

  Interleaving semantics

  Concurrency is expressed by the set of all *possible interleavings*.

- note: for each interaction sequence, all interactions are ordered sequentially, showing their "visible" concurrent actions

Regular expressions (to express traces and histories)

- well known and widely used "format" to describe "languages" (= sets of finite "words" over a given "*alphabet*")
- formed by
  - [...] (meta-parentheses)
  - superscript * (repetition) as in $[..]^*$
  - superscript $^+$ (nonempty repetition) as in $[..]^+$
  - $+$ (choice) [2]
  - ; (sequential composition)

Examples:

$$[a; b]^*$$
$$a + [a; b]$$
$$[a]^* + [b]^*$$
$$[a]^+ + [b]^*$$

A way to describe (sets of) traces

Example 1 (Regular expressions). • 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^*:

$$[a + b]^* \tag{2}$$

[2] Note: we do not use | for choice, to not confuse with parallel composition.

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 general: *both finite and infinite traces*
- two conceptually important classes of properties[4]
  - safety ("nothing wrong will happen")
  - liveness ("something good will happen")

Note: infinite traces are difficult to work with and reason with

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)
- the set of all histories for an execution 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: $\{\epsilon\}, \{[hi_B]\}, \{[hi_B]; [hi_A]\}, \{[hi_B]; [hi_B]\}, \{[hi_B]; [hi_B]; [hi_A]\}$

Histories = sequences of communication events

We use the following conventions

- communication events, $a : Event$ is an event
- set of communication events: A (i.e. $2^{\#Event}$)
- history $h : Hist$ – sequence of events

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

Definition 2 (Histories)**.** Histories (over a given set of events) are defined inductively over the constructors

- ϵ (empty history) and
- _; _ (appending an event at the end of a history)

Note: Functions over histories can then be defined inductively.

[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"

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**
$h \preceq h'$	$= (h = h') \vee h \prec h'$
$h \prec \varepsilon$	$= false$
$h \prec (h';x)$	$= h \preceq h'$

Invariants and Prefix Closed Trace Sets

May use *invariants* to define trace sets:

A (history) invariant $I(h)$ is a predicate over a history h, 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 if

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

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

---

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

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

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

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

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

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

C – coin machine
U – user

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 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. (Inside A we simply write h.)

- 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"

Note: disjoint event sets!

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.

"program 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.

***Composition* rule for histories: from inside to outside view**

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

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

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 is maintained by all local interaction points. I_{A_i} may have the form of an implication, expressing that the output events from A_i depends on a condition on its input events.

Composition rule 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:

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

Example revisited: Coin Machine.

Let us focus on C

```
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):

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

Coin machine example: Loop invariants for C

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:

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

Loop invariant for the inner loop:

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

Note: $h/\uparrow$ denotes the subsequence of messages sent (by C) **Note:** $h/\downarrow$ denotes the subsequence of messages received (by C)

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:

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

This gives the global invariant:

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

Coin machine user U: 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 global invariant:

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

  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.

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:

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

Final Remarks

Remark 1 (Self-communication may be considered internal:)**.** *In "black-box" specifications, we consider observable events only, abstracting away from internal events. Then, the alphabet of an agent A is restricted by requiring the other agent to be different from A:*

$$\alpha_A = \{A{\uparrow}B{:}m \texttt{ where } A \neq B\} \cup \{B{\downarrow}A{:}m \texttt{ where } A \neq B\}$$

Remark 2 (FIFO/non-FIFO ordering:)**.** *The 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).

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 $A{\uparrow}B$.