Asynchronous Communication 1

INF4140

15.11.12

Lecture 10

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* view of components
  - describing overall understanding of a (sub)system
- Histories from the *inside* 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 kinds of settings for concurrent systems, based on the notion of:

- *process* — without self identity, but with named channels. Channels are usually FIFO.
- *object (agent)* — 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 a very general setting. The process/channel setting may be obtained by representing each combination of object and message kind as a channel.

So, in the following we consider *agent/network* systems!

Programming asynchronous agent systems

Standard sequential language extended with statements for sending and receiving:

- **send statement:** **send** $B : m(e)$
  means that the current agent sends message m to agent B where e is an (optional) list of actual parameters.
- **fixed receive statement:** **await** $B : m(w)$
  wait for a message m from a specific agent B, 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.
- We may use a **choice operator** [] to select between alternative statement lists, starting with receive statements.

Here m is a message name, B and e expressions, X and w variables.

Example: Coin Machine

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.

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

Here, the choice operator, [], selects the first enabled branch, (and makes a non-deterministic choice if both branches are enabled).

Interleaving semantics of concurrent systems

- a concurrent system may be described semantically by a *set of executions*,
- where each execution is captured by the *sequence of atomic interaction events*, often called *trace*.

This is called *interleaving semantics*, because in each interaction sequence, all interactions are ordered sequentially, and there is no concurrency (for a given sequence).

Concurrency is expressed by the set of all possible interleavings.

Example: If interactions a and b are concurrent, the regular expression

$$[[a; b] \mid [b; a]]$$

expresses the two possible interleavings.

The parallel composition of a^* and b^* is $[a \mid b]^*$

Safety and liveness considerations: 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 systems give rise to finite traces
- non-terminating systems may give rise to infinite traces (if the interaction goes on)

Thus the trace set semantics of a system with both terminating and non-terminating processes, contain both finite and infinite traces. This kind of semantics expresses both

- *safety* ("nothing wrong will happen")
- *liveness* ("something good will happen")

Safety and liveness considerations: Histories

For practical specification, it is convenient to deal with finite traces, and the notion of a *trace up to a given execution point*, often called a *history*, is useful.

Note: In contrast to the book, histories are here finite initial parts of a trace (prefixes).

Note: The set of histories for all possible choices of execution consists of finite sequences, and is

- *prefix closed*, i.e. if a history h is in the set, then any prefix (initial part) of h is also in the set.

Sets of histories express safety, but not liveness.

Simple example: histories and trace set

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

- a "sloppy" B may or may not give a reply, in which case there will be an infinite trace with only "hi-B" (here comma denotes union).
  Trace set: $\{[hi - B]^{\infty}\}, \{[hi - B]^{+} [hi - A]\}$
  Histories: $\{[hi - B]^{*}\}, \{[hi - B]^{+} [hi - A]\}$
- a "lazy" B will reply eventually, but there is 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:
  $\emptyset, \{[hi - B]\}, \{[hi - B] [hi - A]\}, \{[hi - B] [hi - B]\}, \{[hi - B] [hi - B] [hi - A]\}$

Prefix closure and history functions

A history is a finite sequence of events. We use the following functions:

$$\begin{array}{llll}
\varepsilon & : & \rightarrow \mathit{Hist} & \text{— the empty history (constructor)} \\
_;_ & : \mathit{Hist} * \mathit{Event} & \rightarrow \mathit{Hist} & \text{— append right (constructor)} \\
\#_ & : \mathit{Hist} & \rightarrow \mathit{Nat} & \text{— length} \\
_/_ & : \mathit{Hist} * \mathit{Set} & \rightarrow \mathit{Hist} & \text{— projection by set of events} \\
_ \leq _ & : \mathit{Hist} * \mathit{Hist} & \rightarrow \mathit{Bool} & \text{— prefix relation} \\
_ < _ & : \mathit{Hist} * \mathit{Hist} & \rightarrow \mathit{Bool} & \text{— strict prefix relation}
\end{array}$$

where _ gives argument positions. Definitions (inductive wrt. ε and $\_;\_$):

$$\begin{array}{ll}
\#\varepsilon & = 0 \\
\#(h;x) & = (\#h) + 1 \\
\varepsilon/s & = \varepsilon \\
(h;x)/s & = \textbf{if } x \in s \textbf{ then } (h/s);x \textbf{ else } (h/s) \textbf{ fi} \\
h \leq h' & = (h = h') \vee h < h' \\
h < \varepsilon & = \mathit{false} \\
h < (h';x) & = h \leq h'
\end{array}$$

where x is an event, s a set of events, and h a sequence of events.
$<$ 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:
An 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:

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

which is prefix closed if I is *historically monotonic*:

$$h \leq h' \Rightarrow (I(h') \Rightarrow I(h))$$

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

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

Thus, a monotonic invariant I' defines the following prefix closed trace set:

$$\{h \mid I'(h)\}$$

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

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

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

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

Legality and other functions on histories

Legality (sometimes called well-definedness) can be defined by the following function on histories:

$$legal \quad : Hist \quad \rightarrow Bool \quad \text{— all reception is legal}$$

Definition (inductive wrt. ε and _; _):

$$\begin{aligned} legal(\varepsilon) &= true \\ legal(h; (A \uparrow B : m)) &= legal(h) \\ legal(h; (A \downarrow B : m)) &= legal(h) \land \#(h/\{A \downarrow B : m\}) < \#(h/\{A \uparrow B : m\}) \end{aligned}$$

where m is message and h a history.
A received message has been sent but not yet consumed.
Note: when m includes parameters, legality ensures that the values received are the same as those sent.
Example of a legal history (coin machine C 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, initialized to the 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 actual 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

Global invariant: By a predicate I on the global history, we may specify desired system behavior:

> *"at any point in an execution H the property $I(H)$ is satisfied"*

- By *logging* the history at run-time, as above, we may *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 can we prove such properties by analysis of the program text?
- How can we monitor, or prove correctness properties, *component-wise*?

Semantics: Inside view: Local histories

Definition (α_A): The events *visible to* an agent A, denoted α_A, are the events *local* to A, i.e.

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

Definition (h_A): The *local history* of A , denoted h_A, is the subsequence of all events in an execution which are 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 same as the projection to A -events of the history observed globally.

Note: Each event is visible to one, and only one, agent!

Coin Machine Example: Local Events

The events visible to C are:

$$U{\downarrow}C : \mathit{five} \quad -- \quad$$ C consumes the message "five"

$$U{\downarrow}C : \mathit{one} \quad -- \quad$$ C consumes the message "one"

$$C{\uparrow}U : \mathit{ten} \quad -- \quad$$ C sends the message "ten"

The events visible to U are:

$$U{\uparrow}C : \mathit{five} \quad -- \quad$$ U sends the message "five" to C

$$U{\uparrow}C : \mathit{one} \quad -- \quad$$ U sends the message "one to C

$$C{\downarrow}U : \mathit{ten} \quad -- \quad$$ 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

The code of the agent C is given below, using b as a local variable initialized to 0.

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

Local 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:

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

Coin Machine Example: Loop Invariants

Loop invariant for the outer loop:

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

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:

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

Histories: 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

$$legal(H) \wedge_i \ h_{Ai} = H/\alpha_{Ai}$$

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 local specification of Ai is given by a predicate on the local history $I_{Ai}(h_{Ai})$ describing a property which *holds before all local interaction points*.

I may have the form of an implication, expressing the output events from Ai depends on a condition on its input events.
From local invariants to a global invariant:
if each agent satisfies $I_{Ai}(h_{Ai})$, the total system will satisfy:

$$legal(H) \wedge_i \ I_{Ai}(H/\alpha_{Ai})$$

Coin machine example: from Local to Global Invariant

before each send/receive:

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

Local Invariant of C in terms of h alone:

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

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

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

$$I_C(H/\alpha_C) = 0 \leq sum(H/\alpha_C/\downarrow) - sum(H/\alpha_C/\uparrow) < 15$$

Shorthand notation:

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

Coin machine example: from Local to Global Invariant

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

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

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

Global Invariant of the system U and C:

$$I(H) = legal(H) \wedge I_C(H/\alpha_C) \wedge I_U(H/\alpha_U)$$

implying:

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

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 in this system the coin machine will have balance ≤ 10.

Coin Machine Example: Loop Invariants (Alternative version)

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:

$$\begin{array}{lcl}
rec(\varepsilon) & = & 0 \\
rec(h; (U{\downarrow}C : five)) & = & rec(h) + 5 \\
rec(h; (U{\downarrow}C : one)) & = & rec(h) + 1 \\
rec(h; (C{\uparrow}U : ten)) & = & rec(h) \\
sent(\varepsilon) & = & 0 \\
sent(h; (U{\downarrow}C : five)) & = & sent(h) \\
sent(h; (U{\downarrow}C : one)) & = & sent(h) \\
sent(h; (C{\uparrow}U : ten)) & = & sent(h) + 10
\end{array}$$

Loop invariant for the inner loop:

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

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 foil).

Remark: 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

$$sendevents(h/\downarrow) \;\leq\; h/\uparrow$$

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.

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

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

$$sendevents(h/A\downarrow B) \leq \; h/A\uparrow B$$

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