

Asynchronous Communication I

INF4140 - Models of concurrency

Asynchronous Communication, lecture 10

Høsten 2014

7.11.2014



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?

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**

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**

Why *without channels*? actually

What kind of system? Agent network systems

Two kinds of settings for concurrent systems, based on the notion of:

- **processes** — without self identity, but with named **channels**. Channels often 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 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 **agent/network** systems!*

Two kinds of settings for concurrent systems, based on the notion of:

- **processes** — without self identity, but with named **channels**. Channels often 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 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 **agent/network** systems!*

The remarks about “networks” are not really convincing.

New syntax 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.
- *choice operator* $[]$ to select between alternative statement lists, starting with receive statements.

Here m is a message name, B the name of an agent, 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.

Example: Coin machine (Cont)

```
1 loop
2   while b < 10
3   do
4     (await U: five; b:=b+5)
5     []
6     (await U: one; b:=b+1)
7   od;
8   send U: ten;
9   b:=b-10
10 end
```

- choice operator `[]`¹
 - selects 1 *enabled* branch
 - non-deterministic choice if both branches are enabled

¹In the literature, also `+` as notation can often be found.

- behavior of a concurrent system: may be described as **set of executions**,
- 1 execution: **sequence of atomic interaction 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

²message sequence (charts) in UML etc.

- behavior of a concurrent system: may be described as set of executions,
- 1 execution: sequence of atomic interaction 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

²message sequence (charts) in UML, etc.

an execution sequence may be graphically pictured by UML message sequence diagrams.

Say something about true concurrency, remember

- very well known and widely used “format” to describe “languages” (= sets finite “words” over given a given “alphabet”)
-

A way to describe (sets of) traces

Example (Reg-Expr)

- a , b : atomic interactions.
- Assume them to “run” concurrently

⇒ two possible interleavings, described by

$$[[a.b] + [b.a]] \quad (1)$$

Parallel composition of a^* and b^* :

$$(a + b)^* \quad (2)$$

Remark: notation for reg-expr's

Different notations exist. E.g.: some write $a|b$ 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

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** traces³
- *non-terminating* systems: infinite traces
- trace set semantics in the general case: both finite and infinite traces
- 2 conceptually important classes of properties⁴
 - **safety** (“nothing wrong will happen”)
 - **liveness** (“something good will happen”)

³Be aware: typically an *infinite* set of finite traces.

⁴Safety etc. it's not a property, it's a “property/class of properties”

└ Asynchronous Communication I

└ 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* traces³
- non-terminating systems: *infinite* traces
- trace set semantics in the general case: both finite and infinite traces
- 2 conceptually important classes of properties⁴
 - *safety* ("nothing wrong will happen")
 - *liveness* ("something good will happen")

³Be aware: typically an infinite set of finite traces.

⁴Safety etc. it's not a property. It's a "property/class of properties"

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

Safety and liveness & histories

- often: concentrate on *finite traces*
- reasons
 - conceptually/theoretically simpler
 - connection to monitoring
 - 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 ”.

	traces	histories
a “sloppy” B may or may not give a reply, in which case there will be an infinite trace with only “hi- B ”	$hi_B^\infty + hi_B^+ hi_A$	$hi_B^* + hi_B^+ hi_A$
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.		
an “eager” B will reply within a fixed number of “ hi_B ”s, for instance before A says “ hi_B ” three times.		

- 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).

Asynchronous Communication I

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 ".

	traces	histories
<ul style="list-style-type: none"> a "sloppy" B may or may not give a reply, in which case there will be an infinite trace with only "hi-B" 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" after finitely many "hi-B". an "eager" B will reply within a fixed number of "hi-B", for instance before A says "hi" three times. 	$hi^* \cup hi^* hi$	$hi^* \cup hi^* hi$

• 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).

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 complex

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 ”.

	traces	histories
a “sloppy” B may or may not give a reply, in which case there will be an infinite trace with only “hi- B ”	$hi_B^\infty + hi_B^+ hi_A$	$hi_B^* + hi_B^+ hi_A$
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.		
an “eager” B will reply within a fixed number of “hi- B ”s, for instance before A says “hi- B ” three times.		

- 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: $\{[hi_B]^\infty\}, \{[hi_B]^+ [hi_A]\}$

Asynchronous Communication I

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 ".

	traces	histories
<ul style="list-style-type: none"> a "sloppy" B may or may not give a reply, in which case there will be an infinite trace with only "hi-B" 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_B" after finitely many "hi_A". an "eager" B will reply within a fixed number of "hi_A", for instance before A says "hi_A" three times. 	$hi_A^* \cdot hi_B$ $hi_A^* \cdot hi_A$	$hi_A^* \cdot hi_A$ $hi_A^* \cdot hi_A$

- 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: $\{(hi_A)^*\}, \{(hi_A)^* \cdot hi_B\}$

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 complex

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 ”.

	traces	histories
a “sloppy” B may or may not give a reply, in which case there will be an infinite trace with only “hi- B ”	$hi_B^\infty + hi_B^+ hi_A$	$hi_B^* + hi_B^+ hi_A$
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.		
an “eager” B will reply within a fixed number of “hi- B ”s, for instance before A says “hi- B ” three times.		

- 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: $\{[hi_B]^*\}, \{[hi_B]^+ [hi_A]\}$

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 ".

	traces	histories
<ul style="list-style-type: none"> a "sloppy" B may or may not give a reply, in which case there will be an infinite trace with only "hi-B" 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_{n}" after finitely many "hi_{n}". an "eager" B will reply within a fixed number of "hi_{n}", for instance before A says "hi_{n}" three times. 	$hi^* \cdot hi^* \cdot hi^*$	$hi^* \cdot hi^* \cdot hi^*$

- 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: $\{hi^*\} \cup \{hi^n\} \cup \{hi^n\}$
 Histories: $\{hi^*\}, \{hi^n\}, \{hi^n\}$

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 complex

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 ”.

	traces	histories
a “sloppy” B may or may not give a reply, in which case there will be an infinite trace with only “hi- B ”	$hi_B^\infty + hi_B^+ hi_A$	$hi_B^* + hi_B^+ hi_A$
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.		
an “eager” B will reply within a fixed number of “hi- B ”s, for instance before A says “hi- B ” three times.		

- 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: $\{[hi_B]^*\}, \{[hi_B]^+ [hi_A]\}$

Asynchronous Communication I

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 ".

	traces	histories
<ul style="list-style-type: none"> a "sloppy" B may or may not give a reply, in which case there will be an infinite trace with only "hi-B" 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_{n}" after finitely many "hi_{n}". an "eager" B will reply within a fixed number of "hi_{n}", for instance before A says "hi_{n}" three times. 	$hi^* \cdot hi^* \cdot hi^*$	$hi^* \cdot hi^* \cdot hi^*$

- 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: $\{hi^*\} \cup \{hi^n\} \cup \{hi^n\}$
 Histories: $\{hi^*\}, \{hi^n\}, \{hi^n\}$

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 complex

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 ”.

	traces	histories
a “sloppy” B may or may not give a reply, in which case there will be an infinite trace with only “hi- B ”	$hi_B^\infty + hi_B^+ hi_A$	$hi_B^* + hi_B^+ hi_A$
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.		
an “eager” B will reply within a fixed number of “hi- B ”s, for instance before A says “hi- B ” three times.		

- 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: $\{[hi_B]^*\}, \{[hi_B]^+ [hi_A]\}$

Asynchronous Communication I

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 ".

	traces	histories
<ul style="list-style-type: none"> a "sloppy" B may or may not give a reply, in which case there will be an infinite trace with only "hi-B" 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_{n}" after finitely many "hi_{n}". an "eager" B will reply within a fixed number of "hi_{n}", for instance before A says "hi_{n}" three times. 	$hi^* \cdot hi^* \cdot hi^*$	$hi^* \cdot hi^* \cdot hi^*$

- 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: $\{hi^*\} \cup \{hi^n\} \cup \{hi^n\}$
 Histories: $\{hi^*\}, \{hi^n\}, \{hi^n\}$

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 complex

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 ”.

	traces	histories
a “sloppy” B may or may not give a reply, in which case there will be an infinite trace with only “hi- B ”	$hi_B^\infty + hi_B^+ hi_A$	$hi_B^* + hi_B^+ hi_A$
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.		
an “eager” B will reply within a fixed number of “hi- B ”s, for instance before A says “hi- B ” three times.		

- 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: $\{[hi_B]^*\}, \{[hi_B]^+ [hi_A]\}$

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 ".

	traces	histories
<ul style="list-style-type: none"> a "sloppy" B may or may not give a reply, in which case there will be an infinite trace with only "hi-B" 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_{i}" after finitely many "hi_{j}". an "eager" B will reply within a fixed number of "hi_{j}", for instance before A says "hi_{j}" three times. 	$hi^* \cdot hi^* \cdot hi^*$	$hi^* \cdot hi^* \cdot hi^*$

- 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: $\{hi^*\} \cup \{hi^i\} \cup \{hi^i\}$
 Histories: $\{hi^i\}, \{hi^i\}, \{hi^i\}$

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 complex

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 ”.

	traces	histories
a “sloppy” B may or may not give a reply, in which case there will be an infinite trace with only “hi- B ”	$hi_B^\infty + hi_B^+ hi_A$	$hi_B^* + hi_B^+ hi_A$
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.		
an “eager” B will reply within a fixed number of “hi- B ”s, for instance before A says “hi- B ” three times.		

- 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: $\{[hi_B]^*\}, \{[hi_B]^+ [hi_A]\}$

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 ".

	traces	histories
<ul style="list-style-type: none"> a "sloppy" B may or may not give a reply, in which case there will be an infinite trace with only "hi-B" 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_{n}" after finitely many "hi_{n}". an "eager" B will reply within a fixed number of "hi_{n}", for instance before A says "hi_{n}" three times. 	$hi^* \cdot hi^* \cdot hi^*$	$hi^* \cdot hi^* \cdot hi^*$

- 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: $\{hi^*\} \cup \{hi^n \mid n \in \mathbb{N}\}$
 Histories: $\{hi^n \mid n \in \mathbb{N}\}$

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 complex

Let use the following conventions

- events $x : Event$ is an event,
- set of events: $A : 2^{Event}$
- history $h : Hist$

A set of events is assumed to be fixed.

Definition (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)

Let use the following conventions

- events x : Event is an event,
- set of events: A , 2^{Event}
- history h : Hist

A set of events is assumed to be fixed.

[Definition \(Histories\)](#)

Histories (over the given set of events) is given inductively over the constructors ϵ (empty history) and $_{\cdot} _$ (appending of an event to the right of the history)

Event is a bad terminology

Functions over histories

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
$_ \leq _$: $Hist * Hist$	$\rightarrow Bool$	prefix relation
$_ < _$: $Hist * Hist$	$\rightarrow Bool$	strict prefix relation

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

$$\# \epsilon = 0$$

$$\#(h; x) = \#h + 1$$

$$\epsilon / A = \epsilon$$

$$(h; x) / s = \text{if } x \in A \text{ then } (h/s); x \text{ else } (h/s) \text{ fi}$$

$$h \leq h' = (h = h') \vee h < h'$$

$$h < \epsilon = \text{false}$$

$$h < (h'; x) = h \leq h'$$

2014-11-21

INF4140

└ Asynchronous Communication I

└ Functions over histories

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
$_ \leq _$	Hist + Hist	\rightarrow Bool prefix relation
$_ < _$	Hist + Hist	\rightarrow Bool strict prefix relation

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

$\# \epsilon$	$= 0$
$\#(h; x)$	$= \#h + 1$
ϵ/A	$= \epsilon$
$(h; x)/s$	$=$ if $x \in A$ then $(h/s); x$ else $(h/s) \# \epsilon$
$h \leq h'$	$= (h = h') \vee h < h'$
$h < \epsilon$	$= \text{false}$
$h < (h'; x)$	$= h \leq h'$

$<$ 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:

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

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

$$h \leq h' \Rightarrow (I(h') \Rightarrow I(h)) \quad (4)$$

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

Remark:

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

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

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:

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

• mostly interested in prefix-closed invariants!

• a history invariant is **historically monotonic**:

$$h \leq h' \Rightarrow (I(h) \Rightarrow I(h')) \quad (4)$$

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

Remark:

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

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

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

$U \uparrow C : five$ — 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”

- note all global sequences/histories “make sense”
- depends on the programming language/communication model
- sometimes called well-definedness, well-formedness or similar
- $legal : Hist \rightarrow Bool$

Definition (Legal history)

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

where m is message and h a history.

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

Example (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 ,
- 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 .

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 to **prove** such properties by analysing the program?
- How can we monitor, or prove correctness properties, **component-wise**?

Definition (Local events)

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 by A)
- $B \downarrow A : m$: any reception by A . (input by A)

Definition (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.

- Each event is visible to one, and only one, agent!

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: Sloppy coin machine

```
1 loop
2   while b < 10
3   do
4     (await U: five; b:=b+5)
5     []
6     (await U: one; b:=b+1)
7   od;
8   send U: ten;
9   b:=b-10
10 end
```

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

$U \downarrow C : \text{five}$ — C consumes the message “five”
 $U \downarrow C : \text{one}$ — C consumes the message “one”
 $C \uparrow U : \text{ten}$ — C sends the message “ten”

```

1 loop
2   while b < 10
3   do
4     (await U: five; b:=b+5)
5   []
6     (await U: one; b:=b+1)
7   od;
8   send U: ten;
9   b:=b-10
10 end

```

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"

? 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:

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

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

$$\begin{aligned} \text{sum}(\varepsilon) &= 0 \\ \text{sum}(h; (\dots : \text{five})) &= \text{sum}(h) + 5 \\ \text{sum}(h; (\dots : \text{one})) &= \text{sum}(h) + 1 \\ \text{sum}(h; (\dots : \text{ten})) &= \text{sum}(h) + 10 \end{aligned}$$

Loop invariant for the inner loop:

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

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

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

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 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:

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

Coin machine example: from local to global invariant

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

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

Local Invariant of C in terms of h alone:

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

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

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

$$I_C(H/\alpha_C) = 0 \leq \text{sum}(H/\alpha_C/\downarrow) - \text{sum}(H/\alpha_C/\uparrow) < 15 \quad (9)$$

Shorthand notation:

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

Coin machine example: from local to global invariant

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

$$I_U(h) = 0 \leq \text{sum}(h/\uparrow) - \text{sum}(h/\downarrow) \leq 10$$

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

- Global Invariant of the system U and C :

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

implying:

Overall

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

since $\text{legal}(H)$ gives:

$$\text{sum}(H/U\downarrow C) \leq \text{sum}(H/U\uparrow C) \text{ and}$$

$$\text{sum}(H/C\downarrow U) \leq \text{sum}(H/C\uparrow U).$$

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

2014-11-21

INF4140

Asynchronous Communication I

Coin machine example: from local to global invariant

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

$$I_U(h) = 0 \leq \text{sum}(h \uparrow) - \text{sum}(h \downarrow) \leq 10$$

$$I_U(h/ou) = 0 \leq \text{sum}(H/U \uparrow) - \text{sum}(H/U \downarrow) \leq 10$$

- Global Invariant of the system U and C :

$$I(H) = \text{legal}(H) \wedge I_C(H/oc) \wedge I_U(H/ou) \quad (10)$$

implying:

Overall

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

since $\text{legal}(H)$ gives:

$$\text{sum}(H/U \downarrow C) \leq \text{sum}(H/U \uparrow C) \text{ and}$$

$$\text{sum}(H/C \downarrow U) \leq \text{sum}(H/C \uparrow U)$$

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

I think there was an error in the slides $I_U(h)$

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:

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

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 slide).

Remark: In “black-box” specifications, we consider observable events only, abstracting away from internal events. Then, legality of sending may be strengthened:

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

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

$$\text{sendevents}(h/\downarrow) \leq h/\uparrow$$

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.

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

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

$$\text{sendevents}(h/A\downarrow B) \leq h/A\uparrow B$$

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

Asynchronous Communication II

INF4140 - Models of concurrency

Asynchronous Communication, lecture 11

Høsten 2014

14.11.2014



- 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

- 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**

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

New syntax 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.
- *choice operator* $[]$ to select between alternative statement lists, starting with receive statements.

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

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 event
- **Local reasoning**: we do not know the global invariant
 - For **await**: unknown parameter values
 - For *open receive*: unknown sender

⇒ use **non-deterministic** assignment

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

where variable x may be given any (type correct) value

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 event
- Local reasoning: we do not know the global invariant
 - For await: unknown parameter values
 - For open receive: unknown sender

⇒ use non-deterministic assignment

$x ::= \text{some}$ (11)

where variable x may be given any (type correct) value

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 $H \dots$), 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.

- each send statement **send** $B : m$ in A is treated as:

$$h := (h; A \uparrow B : m) \quad (12)$$

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

$$w := \text{some} ; h := (h; B \downarrow A : m(\vec{x})) \quad (13)$$

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

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

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

where the usage of $X := \text{some}$ expresses that A may receive the message from any agent

⁵where \vec{x} is a sequence of variables

Non-det assignment

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

- as said: await/send have been **expressed** by manipulating h , using non-det assignments
- ⇒ rules for await/send statements

Non-det assignment

$$\frac{}{\{ \forall x. Q \} \times \Rightarrow \text{some } \{ Q \}} \text{ND-Assign}$$

- as said: await/send have been expressed by manipulating A , using non-det assignments
- rules for await/send statements

Not too important: the statement is also known as “havoc”

Derived Hoare rules for send and receive

$$\frac{}{\{ Q_{h \leftarrow h; A \uparrow B : m} \} \mathbf{send} B : m \{ Q \}} \text{Send}$$

$$\frac{}{\{ \forall \vec{x} . Q_{h \leftarrow h; B \downarrow A : m(\vec{x})} \} \mathbf{await} B : m(\vec{x}) \{ Q \}} \text{Receive}_1$$

$$\frac{}{\{ \forall \vec{x}, X . Q_{h \leftarrow h; X \downarrow A : m(\vec{x})} \} \mathbf{await} X ? m(\vec{x}) \{ Q \}} \text{Receive}_2$$

- As before: A is 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.
- **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:

$$\frac{}{\{ Q_{h \leftarrow h; (B \downarrow A : m)} \} \mathbf{await} B : m \{ Q \}} \text{Receive}$$

Hoare rules for local reasoning

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

Rule for $[]$

$$\frac{\{ P_1 \} S_1 \{ Q \} \quad \{ P_2 \} S_2 \{ Q \}}{\{ P_1 \wedge P_2 \} (S_1 [] S_2) \{ Q \}} \text{Nondet}$$

Remark: We may reason similarly backwards over conditionals:⁶

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

⁶We 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.

The Hoare rule for non-deterministic choice ($\{\} \{\}$) isRule for $\{\}$

$$\frac{\{P_1\} S_1 \{Q\} \quad \{P_2\} S_2 \{Q\}}{\{P_1 \wedge P_2\} (S_1 \{\} S_2) \{Q\}} \text{Nondet}$$

Remark: We may reason similarly backwards over conditionals.⁶

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

⁶You 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.

I remove that as precondition, i.e., in the logic, **if b then P_1 else P_2** and replaced it as shown in If', because I feel it's confusing. In earlier versions they used that as abbreviations for the pre-condition as shown.

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.

Example: Coin machine (Cont)

```
1 loop
2   while b < 10
3   do
4     (await U: five; b:=b+5)
5     []
6     (await U: one; b:=b+1)
7   od;
8   send U: ten;
9   b:=b-10
10 end
```

- **choice** operator `[]`¹
 - selects 1 *enabled* branch
 - **non-deterministic** choice if both branches are enabled

¹In the literature, also `+` as notation can often be found

Coin Machine Example: Events

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

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

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

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

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

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

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:

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

Coin machine example: Loop invariants

Loop invariant for the outer loop:

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

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

$$\begin{aligned} \text{sum}(\varepsilon) &= 0 \\ \text{sum}(h; (\dots : \text{five})) &= \text{sum}(h) + 5 \\ \text{sum}(h; (\dots : \text{one})) &= \text{sum}(h) + 1 \\ \text{sum}(h; (\dots : \text{ten})) &= \text{sum}(h) + 10 \end{aligned}$$

Loop invariant for the inner loop:

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

Inner loop

let I_i (“inner invariant”) abbreviate equation (6)

```
1  {  $I_i$  }
2  while  $b < 10$  {  $b \leq 10 \wedge I_i$  }
3  {  $(I_i \ b \leftarrow (b+5))_{h \leftarrow h; U \downarrow C: \text{five}}$   $\wedge$   $(I_i \ b \leftarrow (b+1))_{h \leftarrow h; U \downarrow C: \text{one}}$  }
4  do
5    ( await  $U: \text{five}$ ; {  $I_i \ b \leftarrow b+1$  }
6     $b := b+5$  )
7  []
8    ( await  $U: \text{one}$ ;  $b := b+1$ )
9    {  $I_i$  }
10 od;
11 {  $I_i \wedge b \geq 10$  }
12 {  $(I_o \ b \leftarrow b-10)_{h \leftarrow h; C \uparrow U: \text{ten}}$  }
13 send  $U: \text{ten}$ ;
```

Must prove the implication:

$$b < 10 \wedge I_i \Rightarrow (I_i \ b \leftarrow (b+5))_{h \leftarrow h; U \downarrow C: \text{five}} \wedge (I_i \ b \leftarrow (b+1))_{h \leftarrow h; U \downarrow C: \text{one}}$$

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

```
1 { Io }
2 loop
3   { Io }
4   { Ii }
5   while b < 10 { b ≤ 10 ∧ Ii }
6   { (Ii b ← (b+5))h ← h; U ↓ C: five ∧ (Ii b ← (b+1))h ← h; U ↓ C: one }
7   do
8     ( await U: five; { Ii b ← b+1 }
9     b := b+5 )
10  []
11  ( await U: one; b := b+1 )
12  { Ii }
13  od;
14  { Ii ∧ b ≥ 10 }
15  { (Io b ← b-10)h ← h; C ↑ U: ten }
16  send U: ten;
17  { Io b ← b-10 }
18  b := b-10
19  { Io }
20 end
```

Verification conditions (as usual):

- $I_o \Rightarrow I_i$, and
- $I_i \wedge b \geq 10 \Rightarrow (I_o \ b \leftarrow (b-10))_{h \leftarrow h; C \uparrow U: ten}$

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
- ⇒ 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.

- 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
- if reasoning gives the condition $Q(\vec{x}, h)$ immediately after a send or receive statement, we basically need to ensure:

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

- 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.

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

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

3 kinds of verification conditions

$$(h = (h'; A \uparrow B : m(e)) \wedge I_A(h') \wedge Q(\vec{x}, h)) \Rightarrow I_A(h) \quad (16)$$

$$(h = (h'; B \downarrow A : m(\vec{y})) \wedge I_A(h') \wedge Q(\vec{x}, h)) \Rightarrow I_A(h) \quad (17)$$

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

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

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)):

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

Consider the statement **send** $U : \text{ten}$ in C

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

\Rightarrow Verification condition, corresponding to equation (16):

$$h = h'; (C \uparrow U : \text{ten}) \wedge I_C(h') \wedge I_o \text{ } b \leftarrow (b-10) \Rightarrow I_C(h) \quad (19)$$

└ Asynchronous Communication II

└ 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)):

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

Consider the statement `send U : fan` in C

- Hoare analysis of the outer loop gave the condition I_C \wedge $I_{b \rightarrow (a-10)}$ immediately after the statement
 - history ends with the event $C \uparrow U : fan$
- ⇒ Verification condition, corresponding to equation (16)

$$h = h', (C \uparrow U : fan) \wedge I_C(h') \wedge I_{b \rightarrow (a-10)} \Rightarrow I_C(h) \quad (16)$$

The local history invariant I_C was given before, on slide 50. 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 (19), and using definition of sum and using $(sum(h'/\downarrow) - sum(h'/\uparrow) = b$ in the last step

$$\begin{aligned} & h = h'; (C \uparrow U : ten) \wedge \\ & I_C(h') \wedge \\ & I_o \text{ } b \leftarrow (b-10) \\ & \Rightarrow I_C(h) \end{aligned}$$

$$\begin{aligned} & h = h'; (C \uparrow U : ten) \wedge \\ & (0 \leq sum(h'/\downarrow) - sum(h'/\uparrow) < 15) \wedge \\ & (sum(h/\downarrow) = sum(h/\uparrow) + b - 10 \wedge 0 \leq b - 10 < 5) \\ & \Rightarrow 0 \leq sum(h/\downarrow) - sum(h/\uparrow) < 15 \end{aligned}$$

$$\begin{aligned} & h = h'; (C \uparrow U : ten) \wedge \\ & (0 \leq sum(h'/\downarrow) - sum(h'/\uparrow) < 15) \wedge \\ & (sum(h'/\downarrow) = sum(h'/\uparrow) + 10 + b - 10 \wedge 0 \leq b - 10 < 5) \\ & \Rightarrow 0 \leq sum(h'/\downarrow) - sum(h'/\uparrow) - 10 < 15 \end{aligned}$$

$$(0 \leq b < 15) \wedge 0 \leq b - 10 < 5)$$

$$\Rightarrow 0 \leq b - 10 < 15$$

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).

Consider a client C , mini bank M and central bank B :

Example of successful cycle:

[$C \downarrow M : card_in(n)$, $M \downarrow C : pin$, $C \downarrow M : pin(x)$,
 $M \downarrow C : amount$, $C \downarrow M : amount(y)$, $M \downarrow B : request(n, x, y)$, $B \downarrow M : grant$,
 $M \downarrow C : cash(y)$, $M \downarrow C : card_out$]

where n is name, x pin code, and y cash amount, provided by clients.

Example of unsuccessful cycle:

[$C \downarrow M : card_in(n)$, $M \downarrow C : pin$, $C \downarrow M : pin(x)$,
 $M \downarrow C : amount$, $C \downarrow M : amount(y)$, $M \downarrow B : request(n, x, y)$, $B \downarrow M : deny$,
 $M \downarrow C : card_out$]

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: $[C \uparrow M : \text{card_in}(n), M \downarrow C : \text{pin}, C \uparrow M : \text{pin}(x), M \downarrow C : \text{amount}, C \uparrow M : \text{amount}(y), M \downarrow C : \text{cash}(y), M \downarrow C : \text{card_out}]$
- Mini Bank: $[C \downarrow M : \text{card_in}(n), M \uparrow C : \text{pin}, C \downarrow M : \text{pin}(x), M \uparrow C : \text{amount}, C \downarrow M : \text{amount}(y), M \uparrow B : \text{request}(n, x, y), B \downarrow M : \text{grant}, M \uparrow C : \text{cash}(y), M \uparrow C : \text{card_out}]$
- Central Bank: $[M \downarrow B : \text{request}(n, x, y), B \uparrow M : \text{grant}]$

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

Mini bank example: Local histories (2)

The unsuccessful cycle:

- Client: [$C \uparrow M : card_in(n)$, $M \downarrow C : pin$, $C \uparrow M : pin(x)$,
 $M \downarrow C : amount$, $C \uparrow M : amount(y)$, $M \downarrow C : card_out$]
- Mini Bank: [$C \downarrow M : card_in(n)$, $M \uparrow C : pin$, $C \downarrow M : pin(x)$,
 $M \uparrow C : amount$, $C \downarrow M : amount(y)$, $M \uparrow B : request(n, x, y)$,
 $B \downarrow M : deny$, $M \uparrow C : card_out$]
- 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 names

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

```
1 Loop
2   await X?request(n,x,y);
3   if   pin[n]=x and bal[n]>y
4   then bal[n]:=bal[n]-y;
5         send X:grant;
6   else send X:deny
7   fi
8 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:

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

- with $+$ for choice, $[...]^*$ for repetition
- Defines cycles: *request* answered with either *grant* or *deny*
- notation $[regExp \text{ some } X, n, x, y]^*$ 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 $h \text{ is } R$ denote that h matches the structure described by R .

Example (for events a , b , and c):

- we have $(a; b; a; b) \text{ is } [a, b]^*$
- we have $(a; c; a; b) \text{ is } [a, [b|c]]^*$
- we do *not* have $(a; b; a) \text{ is } [a, b]^*$

Loop invariant of Central Bank (B):

Let $Cycle_B$ denote the regular expression:

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

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):

```
1 loop
2   {  $h$  is  $Cycle_B$  }
3   {  $\forall X, n, x, y$ . if  $pin[n] = x \wedge bal[n] > y$  then  $h'_1$  is  $Cycle_B$  else  $h'_2$  is  $Cycle_B$  }
4   await X?request( $n, x, y$ );
5   { if  $pin[n] = x \wedge bal[n] > y$  then  $h'_1$  is  $Cycle_B$  else  $h'_2$  is  $Cycle_B$  }
6   if pin[ $n$ ]= $x$  and bal[ $n$ ] $>y$ 
7
8     then bal[ $n$ ]:=bal[ $n$ ]- $y$ ;
9         { ( $h; B \uparrow X : grant$ ) is  $Cycle_B$  }
10        send X:grant;
11        { ( $h; B \uparrow X : grant$ ) is  $Cycle_B$  }
12    else
13        { ( $h; B \uparrow X : deny$ ) is  $Cycle_B$  }
14    fi
15    {  $h$  is  $Cycle_B$  }
16 end
```

$$h'_1 = h; X \downarrow B : request(n, x, y); B \uparrow X : grant$$

$$h'_2 = h; B \uparrow X : grant$$

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

Verification condition:

h is $Cycle_B \Rightarrow \forall X, n, x, y . \text{if } pin[n] = x \wedge bal[n] > y$
 then $(h; X \downarrow B : request(n, x, y); B \uparrow X : grant)$ is $Cycle_B$
 else $(h; X \downarrow B : request(n, x, y); B \uparrow X : deny)$ is $Cycle_B$

where $Cycle_B$ is

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

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

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

since $(X \downarrow B : request(n, x, y); B \uparrow X : grant)$ is $Cycle_B$

and $(X \downarrow B : request(n, x, y); B \uparrow X : deny)$ is $Cycle_B$

Local history invariant for the central bank (B)

$Cycle_B$ is

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

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

$h \leq \text{Cycle}_B$

- 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**)

$h \text{ is } \text{Cycle}_B \Rightarrow h \leq \text{Cycle}_B$ which is trivial.

2. **after await request**

if ... then $(h; B \uparrow X : \text{grant}) \text{ is } \text{Cycle}_B$ **else** $(h; B \uparrow X : \text{deny}) \text{ is } \text{Cycle}_B$
 $\Rightarrow h \leq \text{Cycle}_B$ which follows from $(h; a) \text{ is } R \Rightarrow h \leq R$.

Note: We have now proved that the implementation of B satisfies the local history invariant, $h \leq \text{Cycle}_B$.

Mini bank example: Local invariant of Client (C)

$Cycle_C$:

[$C \uparrow X : card_in(n)$
+ $X \downarrow C : pin, C \uparrow X : pin(x)$
+ $X \downarrow C : amount, C \uparrow X : amount(y')$
+ $X \downarrow C : cash(y)$
+ $X \downarrow C : card_out \text{ some } X, y, y']^*$

History invariant:

$$h_C \leq Cycle_C$$

Note: The values of C , n and x are fixed from cycle to 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↓M : card_in(n), M↑C : pin, C↓M : pin(x),  
  M↑C : amount, C↓M : amount(y),  
  if y ≤ 0 then ε else  
  M↑B : request(n, x, y), [B↓M : deny + B↓M : grant, M↑C : cash(y) ] fi,  
  M↑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:

$H \leq [C \Downarrow M : card_in(n), M \Downarrow C : pin, C \Downarrow M : pin(x),$

$M \Downarrow C : amount, C \Downarrow M : amount(y),$

if $y \leq 0$ **then** $M \Downarrow C : card_out$

else $M \Downarrow B : request(n, x, y), [B \Downarrow M : deny, M \Downarrow C : card_out$

$+ B \Downarrow M : grant, M \Downarrow C : cash(y), [M \Downarrow C : cash(y) ||| M \Downarrow C : card_out]]$ **fi**

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*).

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**

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