Concurrency Theory

Winter 2025/26

Lecture 8: The Alternating-Bit Protocol & Hennessy-Milner Logic

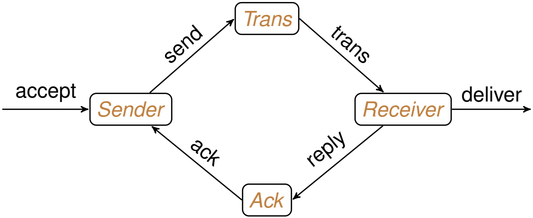
Thomas Noll, Peter Thiemann
Programming Languages Group
University of Freiburg

https://proglang.github.io/teaching/25ws/ct.html

Thomas Noll, Peter Thiemann

Winter 2025/26

The Alternating-Bit Protocol (ABP)



Working principle of ABP

- Goal: reliable communication of data (from some finite set *D*) between *Sender* and *Receiver*.
- No direct communication between *Sender* and *Receiver*; all messages must travel through the channels (which are unreliable).
- *Sender* transfers data to *Receiver* via transport channel *Trans*.
- *Receiver* confirms reception via acknowledgement channel *Ack*.
- In addition to the actual data transmitted, a control bit is used to ensure reliability. After each successful transmission, this bit toggles its value

The Channels

- Channels are unidirectional and can transfer only one message each time (i.e., no message overtaking is possible).
- Channels are unreliable, i.e., messages can be lost, corrupted, or duplicated. However transmission errors are detected.

The channel processes

- *Trans* transmits frames of the following form (where $\mathbb{B} := \{0, 1\}$):

$$F = \{db \mid d \in D, b \in \mathbb{B}\}.$$

- It detects transmission errors and reports them to *Receiver*:

$$Trans = \sum_{f \in F} send_f.\left(\overline{trans_f}.Trans + \overline{trans_\perp}.Trans\right)$$

- *Ack* behaves like *Trans* but only carries control bits:

$$Ack = \sum_{b \in \mathbb{B}} reply_b.\left(\overline{ack_b}.Ack + \overline{ack_\perp}.Ack\right)$$

The Sender

- After accepting data, *Sender* sends it via *Trans* with a control bit b (which is initially set to 0) repeatedly until the acknowledgement b is received over *Ack*. For the next data item, control bit $1 - b$ is used, and so on.
- When *Sender* receives an acknowledgement containing the same bit as the message it is currently transmitting, it stops transmitting that message, flips the protocol bit, and repeats the protocol for the next message.
- Otherwise, it re-initiates transmission.

The sender process

For $d \in D$ and $b \in \mathbb{B}$:

$$\begin{aligned}
\mathit{Sender} &= \mathit{Send}_0 \\
\mathit{Send}_b &= \textstyle\sum_{d \in D} \mathit{accept}_d.\mathit{Send}_{db} \\
\mathit{Send}_{db} &= \overline{\mathit{send}_{db}}.\mathit{Wait}_{db} \\
\mathit{Wait}_{db} &= \mathit{ack}_b.\mathit{Send}_{1-b} + \mathit{ack}_{1-b}.\mathit{Send}_{db} + \mathit{ack}_\perp.\mathit{Send}_{db}
\end{aligned}$$

The Receiver

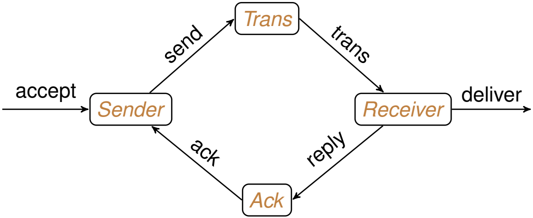
- *Receiver* gets the data item with a control bit b or the information that the data is corrupted.
- In the first case, if b is the expected value of the control bit (which is initially set to 0 also on the receiving side), b is returned via *Ack*.
- Otherwise, the transmission is re-initiated by returning the “wrong” control bit $1 - b$ to Sender.
- When a message is received for the first time, *Receiver* delivers it, while subsequent messages with the same control bit are simply acknowledged.

The receiver process

For $d \in D$ and $b \in \mathbb{B}$:

$$
\begin{aligned}
Receiver &= Recv_0 \\
Recv_b &= \sum_{d \in D} trans_{db}.\overline{deliver_d}.\overline{reply_b}.Recv_{1-b} \\
&+ \sum_{d \in D} trans_{d(1-b)}.\overline{reply_{1-b}}.Recv_b
\end{aligned}
$$

The Overall Protocol



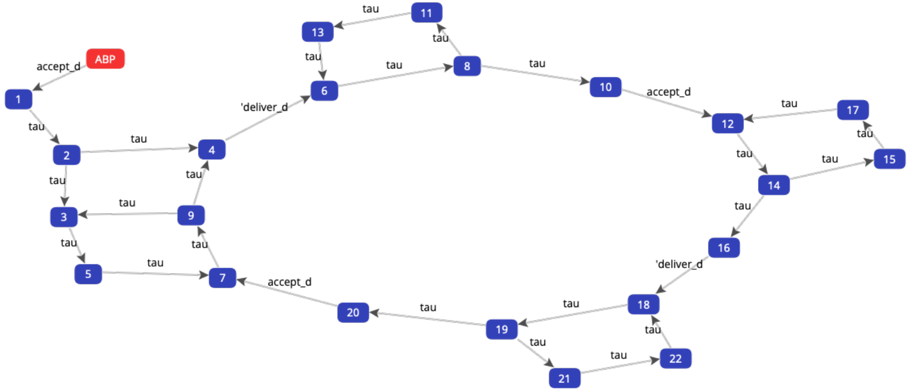
- The only actions visible to the environment are *accept* and *deliver* steps on the *Sender* and *Receiver* side, respectively; everything else is considered to be internal communication.
- The protocol is expected to behave like a one-place buffer.

The protocol and its specification

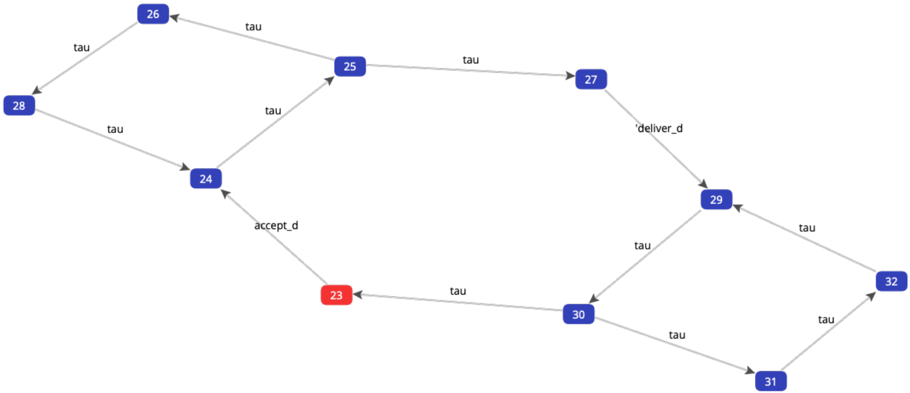
$$ABP = (Sender \parallel Trans \parallel Ack \parallel Receiver) \setminus L$$
$$L = \{send_{db}, trans_{db} \mid d \in D, b \in \mathbb{B}\}$$
$$\cup \{reply_b, ack_b \mid b \in \mathbb{B}\} \cup \{trans_\perp, ack_\perp\}$$

$$Spec = accept_d.\overline{deliver_d}.Spec$$

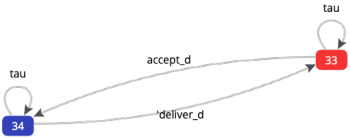
The State Space



The State Space Modulo Strong Bisimulation



The State Space Modulo Weak Bisimulation



Properties

Correctness

The protocol and its specification are weakly bisimilar:

$$ABP \approx Spec.$$

Absence of deadlocks

ABP is deadlock free (formal details later):

$$ABP \models NoDeadlock$$
$$NoDeadlock \overset{max}{=} \langle Act \rangle \mathrm{tt} \wedge [Act] NoDeadlock$$

Existence of livelocks

ABP contains livelocks (formal details later):

$$ABP \models HasLivelock$$
$$HasLivelock \overset{min}{=} CanDiverge \vee \langle Act \rangle HasLivelock$$
$$CanDiverge \overset{max}{=} \langle \tau \rangle CanDiverge$$

Modifying Channel Properties

Lossy channels (without detection)

$$Trans = \sum_{f \in F} send_f.\,\big(\overline{trans_f}.Trans + \tau.Trans\big)$$
$$Ack = \sum_{b \in \mathbb{B}} reply_b.\,\big(\overline{ack_b}.Ack + \tau.Ack\big)$$

Effects:

- $ABP \not\approx Spec$ (ABP only weakly simulates $Spec$)
- $ABP \not\models NoDeadlock$ (both $Sender$ and $Receiver$ waiting but channels empty)
- $ABP \not\models HasLivelock$ (never re-initiates transmission)

Duplicating channels (without detection)

$$Trans = \sum_{f \in F} send_f.\,\big(\overline{trans_f}.Trans + \overline{trans_f}.\overline{trans_f}.Trans\big)$$
$$Ack = \sum_{b \in \mathbb{B}} reply_b.\,\big(\overline{ack_b}.Ack + \overline{ack_b}.\overline{ack_b}.Ack\big)$$

Effects:

- $ABP \not\approx Spec$ (ABP only weakly simulates $Spec$)
- $ABP \not\models NoDeadlock$
- $ABP \not\models HasLivelock$ (only bounded re-transmission)

Verifying Correctness of Concurrent Systems

Equivalence-checking approach

$$\mathit{Impl} \equiv \mathit{Spec}$$

- $\equiv$ is some equivalence, e.g., $\sim$ or $\approx^c$.
- *Spec* is often expressed in the same language as *Impl*, e.g., CCS.
- *Spec* provides the full specification of the intended behaviour.

Model-checking approach

$$\mathit{Impl} \models \mathit{Prop}$$

- $\models$ is the satisfaction relation.
- *Prop* is a particular feature, often expressed via a logic, e.g., HML.
- *Prop* is a partial specification of the intended behaviour.

Motivation

Goal: check processes for simple properties

- Action a is initially enabled.
- Action b is initially disabled.
- A deadlock never occurs.
- A server always sends a reply after receiving a request.

Approach:

- Formalisation in Hennessy-Milner Logic (HML)
- M. Hennessy, R. Milner: *On Observing Nondeterminism and Concurrency*, ICALP 1980, Springer LNCS 85, 299–309
- Checking by exploration of state space

Syntax of HML

Definition 8.1 (Syntax of HML) (Hennessy & Milner 1985)

The set *HMF* of Hennessy-Milner formulae over a set of actions *Act* is defined as follows:

$$
\begin{array}{rcll}
F & ::= & \mathrm{tt} & \text{(true)} \\
 & | & \mathrm{ff} & \text{(false)} \\
 & | & F_1 \wedge F_2 & \text{(conjunction)} \\
 & | & F_1 \vee F_2 & \text{(disjunction)} \\
 & | & \langle\alpha\rangle F & \text{(diamond)} \\
 & | & [\alpha]F & \text{(box)}
\end{array}
$$

where $\alpha \in Act$.

Meaning of HML Constructs

- All processes satisfy tt.
- No process satisfies ff.
- A process satisfies $F \wedge G$ iff it satisfies F and G.
- A process satisfies $F \vee G$ iff it satisfies either F or G or both.
- A process satisfies $\langle\alpha\rangle F$ for some $\alpha \in Act$ iff it there exists an α-labelled transition to a state satisfying F (possibility).
- A process satisfies $[\alpha]F$ for some $\alpha \in Act$ iff all its α-labelled transitions lead to a state satisfying F (necessity).

Abbreviations for $L = \{\alpha_1, \ldots, \alpha_n\}$ ($n \in \mathbb{N}$):

- $\langle L\rangle F := \langle\alpha_1\rangle F \vee \ldots \vee \langle\alpha_n\rangle F$
- $[L]F := [\alpha_1]F \wedge \ldots \wedge [\alpha_n]F$
- In particular, $\langle\emptyset\rangle F :=$ ff and $[\emptyset]F :=$ tt.
- Thus, a process satisfies
  - $\langle Act\rangle$tt iff it has some outgoing transition, i.e., is not deadlocked, and
  - $[Act]$ff iff it has no outgoing transition, i.e., is deadlocked.

Definition 8.2 (Semantics of HML)

Let $(S, Act, \longrightarrow)$ be an LTS and $F \in HMF$.
The set of processes in S that satisfy F, $[\![F]\!] \subseteq S$, is defined by:

$$
\begin{array}{rclcrcl}
[\![\mathrm{tt}]\!] & := & S & \qquad & [\![\mathrm{ff}]\!] & := & \emptyset \\
[\![F_1 \wedge F_2]\!] & := & [\![F_1]\!] \cap [\![F_2]\!] & & [\![F_1 \vee F_2]\!] & := & [\![F_1]\!] \cup [\![F_2]\!] \\
[\![\langle\alpha\rangle F]\!] & := & \langle\cdot\alpha\cdot\rangle([\![F]\!]) & & [\![[\alpha]F]\!] & := & [\cdot\alpha\cdot]([\![F]\!])
\end{array}
$$

where $\langle\cdot\alpha\cdot\rangle, [\cdot\alpha\cdot] : 2^S \to 2^S$ are given by

$$
\begin{aligned}
\langle\cdot\alpha\cdot\rangle(T) &:= \{s \in S \mid \exists s' \in T : s \xrightarrow{\alpha} s'\} \\
[\cdot\alpha\cdot](T) &:= \{s \in S \mid \forall s' \in S : s \xrightarrow{\alpha} s' \Rightarrow s' \in T\}
\end{aligned}
$$

We write $s \models F$ iff $s \in [\![F]\!]$. Two HML formulae are equivalent (written $F \equiv G$) iff they are satisfied by the same processes in every LTS.

Semantics of HML II

Example 8.3 ($\langle\cdot\alpha\cdot\rangle$, $[\cdot\alpha\cdot]$ operators)

$$
\begin{tikzcd}
s \arrow[d, "a"] \arrow[dr, "a"] & & t \arrow[d, "a"] \\
s_1 \arrow[r, "b"] & s_2 \arrow[loop right, "b"] & t_1 \arrow[loop right, "b"]
\end{tikzcd}
$$

(1) $\langle\cdot a\cdot\rangle(\{s_1, t_1\}) = \{s, t\}$

(2) $[\cdot a\cdot](\{s_1, t_1\}) = \{s_1, s_2, t, t_1\}$

(3) $\langle\cdot b\cdot\rangle(\{s_1, t_1\}) = \{t_1\}$

(4) $[\cdot b\cdot](\{s_1, t_1\}) = \{s, t, t_1\}$

Simple Properties Revisited

Example 8.4

(1) Action a is initially enabled: $\langle a\rangle\mathrm{tt}$

$$\begin{aligned}\llbracket\langle a\rangle\mathrm{tt}\rrbracket &= \langle\cdot a\cdot\rangle\llbracket\mathrm{tt}\rrbracket\\ &= \langle\cdot a\cdot\rangle(S)\\ &= \{s\in S \mid \exists s'\in S : s \xrightarrow{a} s'\} =: \{s \in\end{aligned}$$

(2) Action b is initially disabled: $[b]\mathrm{ff}$

$$\begin{aligned}\llbracket[b]\mathrm{ff}\rrbracket &= [\cdot b\cdot]\llbracket\mathrm{ff}\rrbracket\\ &= [\cdot b\cdot](\emptyset)\\ &= \{s\in S \mid \forall s'\in S : s \xrightarrow{b} s' \Rightarrow s'\in\emptyset\}\\ &= \{s\in S \mid \nexists s'\in S : s \xrightarrow{b} s'\} =: \{s \in\end{aligned}$$

Example 8.5

(1) Absence of deadlock:
- initially: $\langle Act\rangle\mathrm{tt}$
- always: later (requires recursion)

(2) Responsiveness:
- initially: $[request]\langle\overline{reply}\rangle\mathrm{tt}$
- always: later (requires recursion)