

# 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>

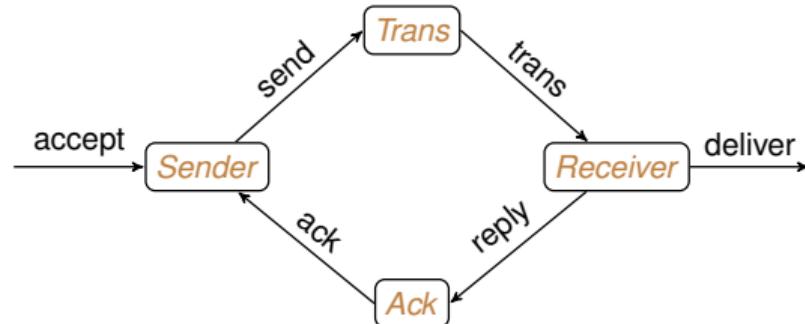
Thomas Noll, Peter Thiemann

Winter 2025/26

# Outline of Lecture 8

- 1 Modelling the ABP
- 2 Analysing the ABP
- 3 Modifying Channel Properties
- 4 Hennessy-Milner Logic (HML)

# 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 (“Alternating-Bit Protocol”).

# 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*:

$$Trans = \sum_{f \in F} send_f. (\overline{trans}_f. Trans + \overline{trans}_{\perp}. Trans)$$

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

$$Ack = \sum_{b \in \mathbb{B}} reply_b. (\overline{ack}_b. Ack + \overline{ack}_{\perp}. Ack)$$

# 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  $b \in \mathbb{B}$ :

$$\text{Sender} = \text{Send}_0$$

$$\text{Send}_b = \sum_{d \in D} \text{accept}_d \cdot \text{Send}_{db}$$

$$\text{Send}_{db} = \overline{\text{send}_{db}} \cdot \text{Wait}_{db}$$

$$\text{Wait}_{db} = \text{ack}_b \cdot \text{Send}_{1-b} + \text{ack}_{1-b} \cdot \text{Send}_{db} + \text{ack}_{\perp} \cdot \text{Send}_{db}$$

# 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.

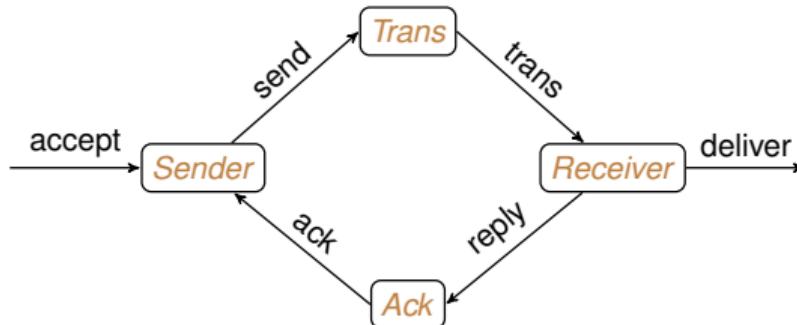
## The receiver process

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

$$Receiver = Recv_0$$

$$\begin{aligned} Recv_b = & \sum_{d \in D} trans_{db} \cdot \overline{deliver}_d \cdot \overline{reply}_b \cdot Recv_{1-b} \\ & + \sum_{d \in D} trans_{d(1-b)} \cdot \overline{reply}_{1-b} \cdot Recv_b \\ & + trans_{\perp} \cdot \overline{reply}_{1-b} \cdot Recv_b \end{aligned}$$

# 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

$$ABP = (\text{Sender} \parallel \text{Trans} \parallel \text{Ack} \parallel \text{Receiver}) \setminus L$$

$$\begin{aligned} L = & \{ \text{send}_{db}, \text{trans}_{db} \mid d \in D, b \in \mathbb{B} \} \\ & \cup \{ \text{reply}_b, \text{ack}_b \mid b \in \mathbb{B} \} \cup \{ \text{trans}_\perp, \text{ack}_\perp \} \end{aligned}$$

$$\text{Spec} = \overline{\text{accept}_d} \cdot \overline{\text{deliver}_d} \cdot \text{Spec}$$

# Outline of Lecture 8

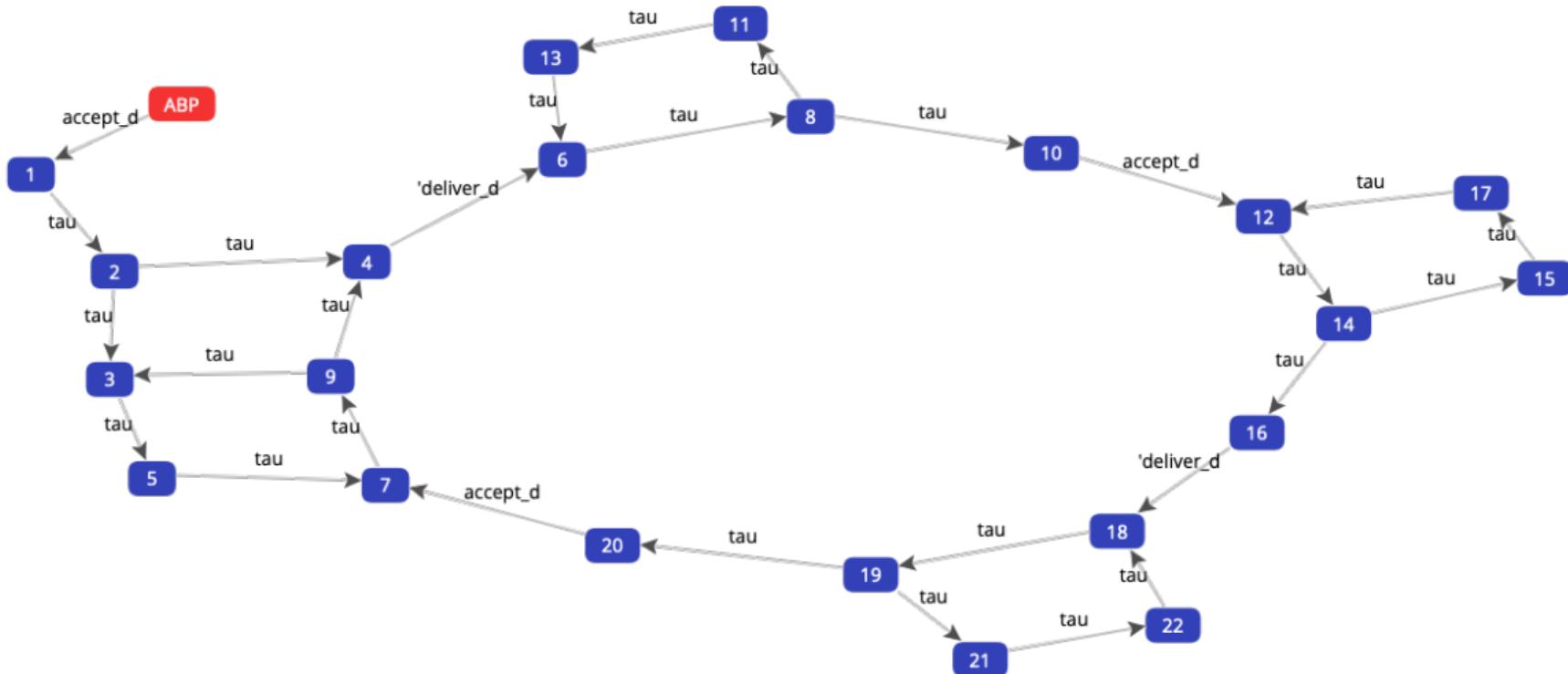
1 Modelling the ABP

2 Analysing the ABP

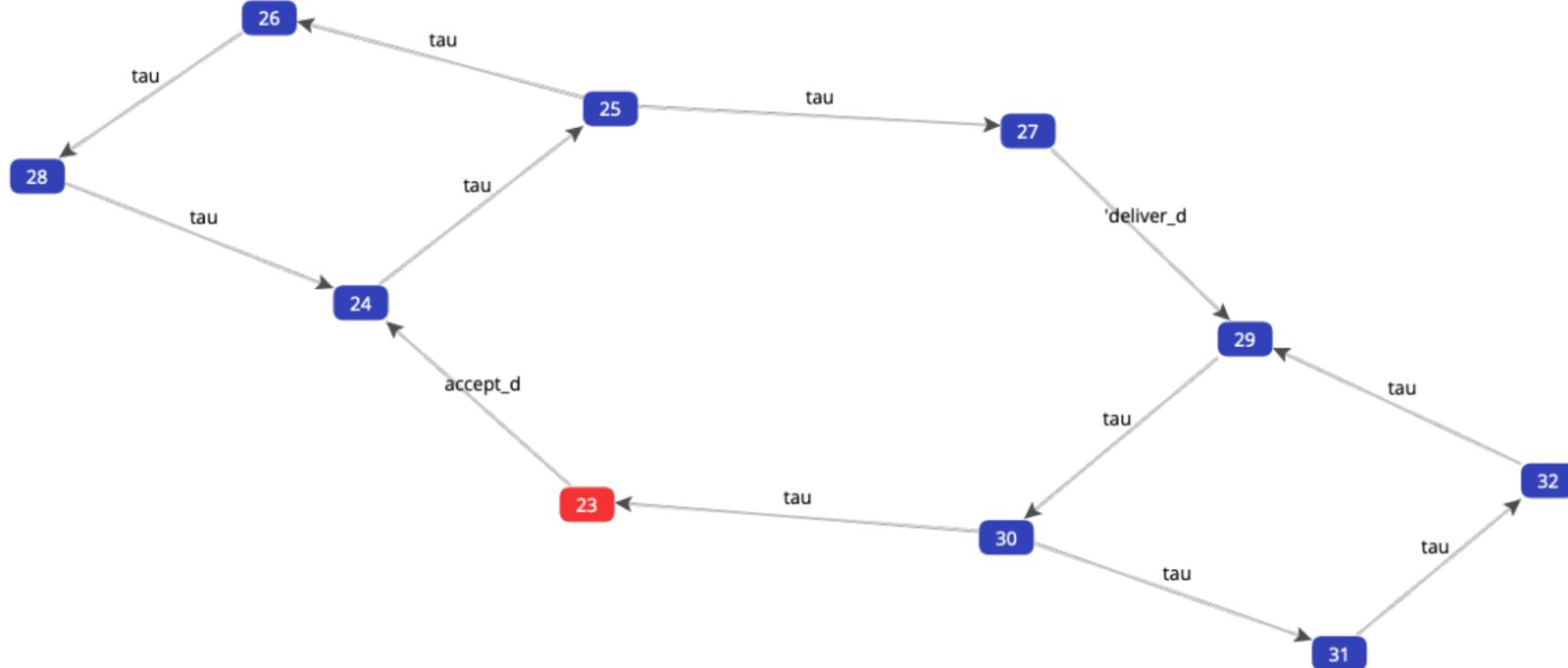
3 Modifying Channel Properties

4 Hennessy-Milner Logic (HML)

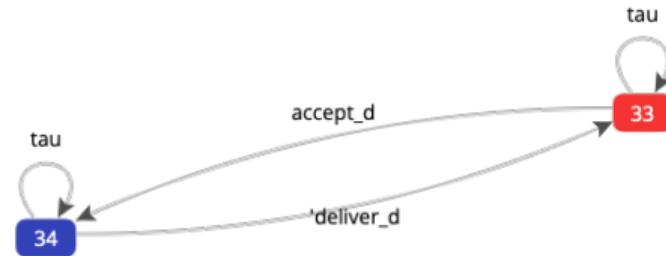
# The State Space



# The State Space Modulo Strong Bisimulation



# The State Space Modulo Weak Bisimulation



# Properties

## Correctness

The protocol and its specification are **weakly bisimilar**:

$$ABP \approx Spec.$$

# Properties

## Correctness

The protocol and its specification are **weakly bisimilar**:

$$ABP \approx Spec.$$

## Absence of deadlocks

$ABP$  is **deadlock free** (formal details later):

$$\begin{aligned} ABP &\models NoDeadlock \\ NoDeadlock &\stackrel{\max}{=} \langle Act \rangle tt \wedge [Act] NoDeadlock \end{aligned}$$

# Properties

## Correctness

The protocol and its specification are **weakly bisimilar**:

$$ABP \approx Spec.$$

## Absence of deadlocks

*ABP* is **deadlock free** (formal details later):

$$\begin{aligned} ABP &\models NoDeadlock \\ NoDeadlock &\stackrel{\max}{=} \langle Act \rangle tt \wedge [Act] NoDeadlock \end{aligned}$$

## Existence of livelocks

*ABP* contains **livelocks** (formal details later):

$$\begin{aligned} ABP &\models HasLivelock \\ HasLivelock &\stackrel{\min}{=} CanDiverge \vee \langle Act \rangle HasLivelock \\ CanDiverge &\stackrel{\max}{=} \langle \tau \rangle CanDiverge \end{aligned}$$

# Outline of Lecture 8

1 Modelling the ABP

2 Analysing the ABP

3 Modifying Channel Properties

4 Hennessy-Milner Logic (HML)

# Modifying Channel Properties

## Lossy channels (without detection)

$$\begin{aligned} Trans &= \sum_{f \in F} send_f. (\overline{trans}_f.Trans + \tau.Trans) \\ Ack &= \sum_{b \in B} reply_b. (\overline{ack}_b.Ack + \tau.Ack) \end{aligned}$$

# Modifying Channel Properties

## Lossy channels (without detection)

$$\begin{aligned} Trans &= \sum_{f \in F} send_f. (\overline{trans}_f.Trans + \tau.Trans) \\ Ack &= \sum_{b \in B} reply_b. (\overline{ack}_b.Ack + \tau.Ack) \end{aligned}$$

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)

# Modifying Channel Properties

## Lossy channels (without detection)

$$\begin{aligned} Trans &= \sum_{f \in F} send_f. (\overline{trans}_f. Trans + \tau. Trans) \\ Ack &= \sum_{b \in B} reply_b. (\overline{ack}_b. Ack + \tau. Ack) \end{aligned}$$

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)

$$\begin{aligned} Trans &= \sum_{f \in F} send_f. (\overline{trans}_f. Trans + \overline{trans}_f. \overline{trans}_f. Trans) \\ Ack &= \sum_{b \in B} reply_b. (\overline{ack}_b. Ack + \overline{ack}_b. \overline{ack}_b. Ack) \end{aligned}$$

# Modifying Channel Properties

## Lossy channels (without detection)

$$\begin{aligned} Trans &= \sum_{f \in F} send_f. (\overline{trans}_f. Trans + \tau. Trans) \\ Ack &= \sum_{b \in B} reply_b. (\overline{ack}_b. Ack + \tau. Ack) \end{aligned}$$

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)

$$\begin{aligned} Trans &= \sum_{f \in F} send_f. (\overline{trans}_f. Trans + \overline{trans}_f. \overline{trans}_f. Trans) \\ Ack &= \sum_{b \in B} reply_b. (\overline{ack}_b. Ack + \overline{ack}_b. \overline{ack}_b. Ack) \end{aligned}$$

Effects:

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

# Outline of Lecture 8

1 Modelling the ABP

2 Analysing the ABP

3 Modifying Channel Properties

4 Hennessy-Milner Logic (HML)

## Equivalence-checking approach

$$\textit{Impl} \equiv \textit{Spec}$$

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

# Verifying Correctness of Concurrent Systems

## Equivalence-checking approach

$$\textit{Impl} \equiv \textit{Spec}$$

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

## Model-checking approach

$$\textit{Impl} \models \textit{Prop}$$

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

**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.

**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:

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

where  $\alpha \in Act$ .

# Meaning of HML Constructs

- All processes satisfy  $\text{tt}$ .

# Meaning of HML Constructs

- All processes satisfy  $\text{tt}$ .
- No process satisfies  $\text{ff}$ .

# Meaning of HML Constructs

- All processes satisfy  $\text{tt}$ .
- No process satisfies  $\text{ff}$ .
- A process satisfies  $F \wedge G$  iff it satisfies  $F$  and  $G$ .

# Meaning of HML Constructs

- All processes satisfy  $\text{tt}$ .
- No process satisfies  $\text{ff}$ .
- 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.

# Meaning of HML Constructs

- All processes satisfy  $\text{tt}$ .
- No process satisfies  $\text{ff}$ .
- 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  $\alpha \in \text{Act}$  iff it there exists an  $\alpha$ -labelled transition to a state satisfying  $F$  (possibility).

# Meaning of HML Constructs

- All processes satisfy  $\text{tt}$ .
- No process satisfies  $\text{ff}$ .
- 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  $\alpha \in \text{Act}$  iff it there exists an  $\alpha$ -labelled transition to a state satisfying  $F$  (possibility).
- A process satisfies  $[\alpha]F$  for some  $\alpha \in \text{Act}$  iff all its  $\alpha$ -labelled transitions lead to a state satisfying  $F$  (necessity).

# Meaning of HTML Constructs

- All processes satisfy  $\text{tt}$ .
- No process satisfies  $\text{ff}$ .
- 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  $\alpha \in \text{Act}$  iff it there exists an  $\alpha$ -labelled transition to a state satisfying  $F$  (possibility).
- A process satisfies  $[\alpha]F$  for some  $\alpha \in \text{Act}$  iff all its  $\alpha$ -labelled transitions lead to a state satisfying  $F$  (necessity).

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

- $\langle L \rangle F := \langle \alpha_1 \rangle F \vee \dots \vee \langle \alpha_n \rangle F$
- $[L]F := [\alpha_1]F \wedge \dots \wedge [\alpha_n]F$
- In particular,  $\langle \emptyset \rangle F := \text{ff}$  and  $[\emptyset]F := \text{tt}$ .
- Thus, a process satisfies
  - $\langle \text{Act} \rangle \text{tt}$  iff it has some outgoing transition, i.e., is not deadlocked, and
  - $[\text{Act}] \text{ff}$  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  $F$ ,  $\llbracket F \rrbracket \subseteq S$ , is defined by:

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

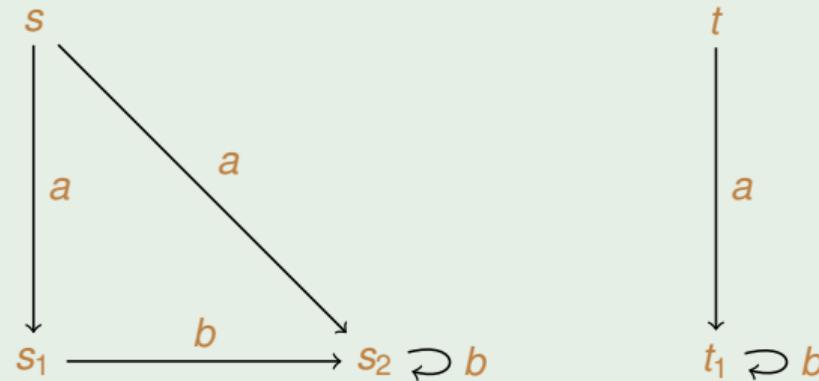
where  $\langle \cdot \alpha \cdot \rangle, [\cdot \alpha \cdot] : 2^S \rightarrow 2^S$  are given by

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

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

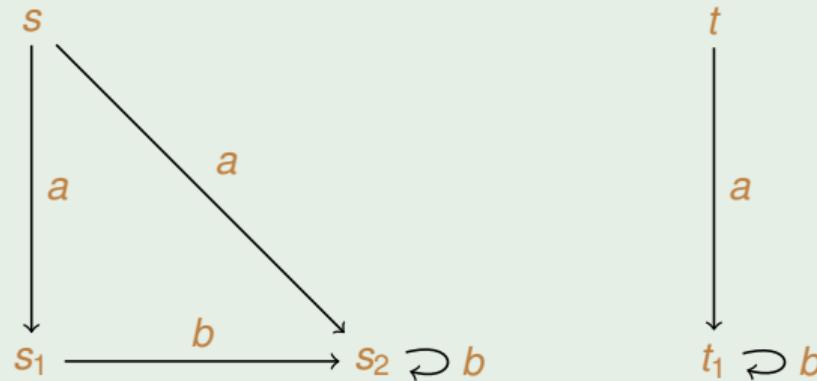
## Semantics of HML II

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



## Semantics of HML II

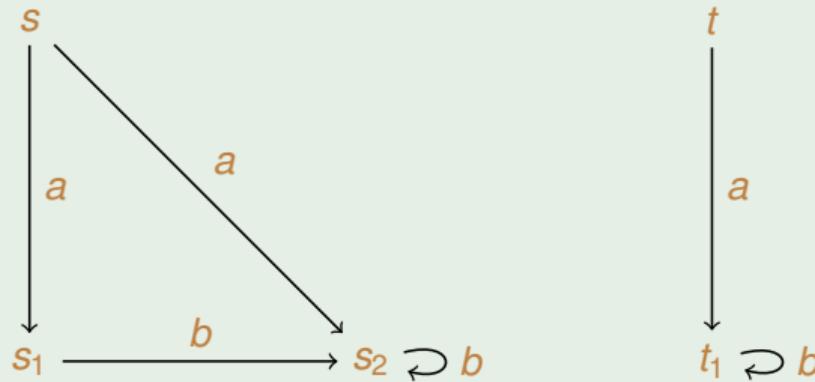
Example 8.3 ( $\langle \cdot a \cdot \rangle$ ,  $[\cdot a \cdot]$  operators)



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

## Semantics of HML II

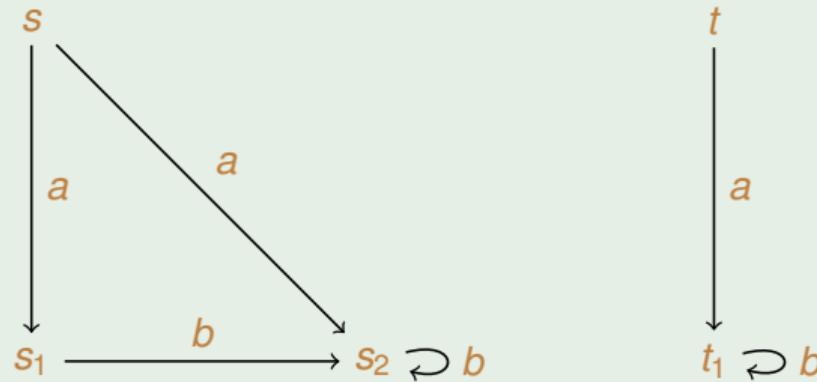
Example 8.3 ( $\langle \cdot a \cdot \rangle$ ,  $[\cdot a \cdot]$  operators)



- (1)  $\langle \cdot a \cdot \rangle(\{s_1, t_1\}) = \{s, t\}$
- (2)  $[\cdot a \cdot](\{s_1, t_1\}) = \{s_1, s_2, t, t_1\}$

## Semantics of HML II

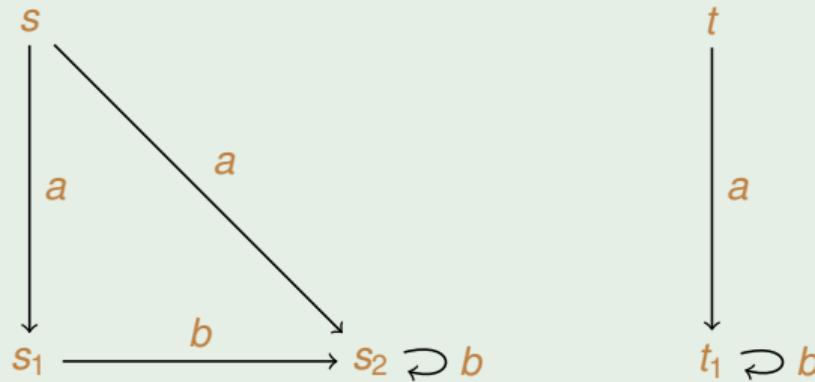
Example 8.3 ( $\langle \cdot \alpha \cdot \rangle$ ,  $[\cdot \alpha \cdot]$  operators)



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

# Semantics of HML II

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



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

## Example 8.4

- (1) Action  $a$  is initially enabled:  $\langle a \rangle tt$

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

# Simple Properties Revisited

## Example 8.4

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

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

(2) Action  $b$  is initially disabled:  $[b]ff$

$$\begin{aligned} \llbracket [b]ff \rrbracket &= [\cdot b \cdot] \llbracket ff \rrbracket \\ &= [\cdot b \cdot](\emptyset) \\ &= \{s \in S \mid \forall s' \in S : s \xrightarrow{b} s' \Rightarrow s' \in \emptyset\} \\ &= \{s \in S \mid \nexists s' \in S : s \xrightarrow{b} s'\} =: \{s \in S \mid s \not\xrightarrow{b}\} \end{aligned}$$

## Example 8.5

(1) Absence of deadlock:

- initially:  $\langle Act \rangle tt$
- always: later  
(requires recursion)

# Simple Properties Revisited

## Example 8.4

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

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

(2) Action  $b$  is initially disabled:  $[b]ff$

$$\begin{aligned} \llbracket [b]ff \rrbracket &= [\cdot b \cdot] \llbracket ff \rrbracket \\ &= [\cdot b \cdot](\emptyset) \\ &= \{s \in S \mid \forall s' \in S : s \xrightarrow{b} s' \Rightarrow s' \in \emptyset\} \\ &= \{s \in S \mid \nexists s' \in S : s \xrightarrow{b} s'\} =: \{s \in S \mid s \not\xrightarrow{b}\} \end{aligned}$$

## Example 8.5

(1) Absence of deadlock:

- initially:  $\langle Act \rangle tt$
- always: later  
(requires recursion)

(2) Responsiveness:

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