Concurrency Theory

Winter Semester 2019/20

Lecture 3: Hennessy-Milner Logic

Joost-Pieter Katoen and Thomas Noll
Software Modeling and Verification Group
RWTH Aachen University

https://moves.rwth-aachen.de/teaching/ws-19-20/ct/

Software Modeling and Verification Chair | RWTH AACHEN UNIVERSITY

Outline of Lecture 3

Syntax of CCS I

Definition (Syntax of CCS)

- Let A be a set of (action) names.
- $\overline{A} := \{\overline{a} \mid a \in A\}$ denotes the set of co-names.
- $Act := A \cup \overline{A} \cup \{\tau\}$ is the set of actions with the silent (or: unobservable) action τ.
- Let Pid be a set of process identifiers.
- The set Prc of process expressions is defined by the following syntax:

$$
\begin{array}{rlll}
P ::= & \mathrm{nil} & & \text{(inaction)} \\
\mid & \alpha.P & & \text{(prefixing)} \\
\mid & P_1 + P_2 & & \text{(choice)} \\
\mid & P_1 \parallel P_2 & & \text{(parallel composition)} \\
\mid & P \setminus L & & \text{(restriction)} \\
\mid & P[f] & & \text{(relabelling)} \\
\mid & C & & \text{(process call)}
\end{array}
$$

where $\alpha \in Act$, $\emptyset \neq L \subseteq A$, $C \in Pid$, and $f : Act \to Act$ such that $f(\tau) = \tau$ and $f(\overline{a}) = \overline{f(a)}$ for each $a \in A$.

Syntax of CCS II

Definition (continued)

- A (recursive) process definition is an equation system of the form

$$(C_i = P_i \mid 1 \leq i \leq k)$$

where $k \geq 1$, $C_i \in Pid$ (pairwise distinct), and $P_i \in Prc$ (with identifiers from $\{C_1, \ldots, C_k\}$).

Notational Conventions:

- $\overline{\overline{a}}$ means a
- $\sum_{i=1}^{n} P_i$ ($n \in \mathbb{N}$) means $P_1 + \ldots + P_n$ (where $\sum_{i=1}^{0} P_i :=$ nil)
- $P \setminus a$ abbreviates $P \setminus \{a\}$
- $[a_1 \mapsto b_1, \ldots, a_n \mapsto b_n]$ stands for $f : Act \to Act$ with $f(a_i) = b_i$ for $i \in [n]$ and $f(\alpha) = \alpha$ otherwise
- restriction and relabelling bind stronger than prefixing, prefixing stronger than composition, composition stronger than choice:

$$P \setminus a + b.Q \parallel R \quad \text{means} \quad (P \setminus a) + ((b.Q) \parallel R)$$

Labelled Transition Systems

Goal: represent behaviour of system by (infinite) graph

- nodes = system states
- edges = transitions between states

Definition (Labelled transition system)

An (*Act*-)labelled transition system (LTS) is a triple $(S, Act, \longrightarrow)$ consisting of

- a set S of states
- a set Act of (action) labels
- a transition relation $\longrightarrow \subseteq S \times Act \times S$

For $(s, \alpha, s') \in \longrightarrow$ we write $s \stackrel{\alpha}{\longrightarrow} s'$. An LTS is called finite if S is so.

Remarks:

- sometimes an initial state $s_0 \in S$ is distinguished ("$LTS(s_0)$")
- (finite) LTSs correspond to (finite) automata without final states

Semantics of CCS I

Definition (Semantics of CCS)

A process definition $(C_i = P_i \mid 1 \leq i \leq k)$ determines the LTS $(Prc, Act, \longrightarrow)$ whose transitions can be inferred from the following rules ($P, P', Q, Q' \in Prc$, $\alpha \in Act$, $\lambda \in A \cup \overline{A}$, $a \in A$):

$$\text{(Act)}\ \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \text{(Sum}_1\text{)}\ \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \qquad \text{(Sum}_2\text{)}\ \frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'}$$

$$\text{(Par}_1\text{)}\ \frac{P \xrightarrow{\alpha} P'}{P \parallel Q \xrightarrow{\alpha} P' \parallel Q} \qquad \text{(Par}_2\text{)}\ \frac{Q \xrightarrow{\alpha} Q'}{P \parallel Q \xrightarrow{\alpha} P \parallel Q'} \qquad \text{(Com)}\ \frac{P \xrightarrow{\lambda} P' \quad Q \xrightarrow{\overline{\lambda}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

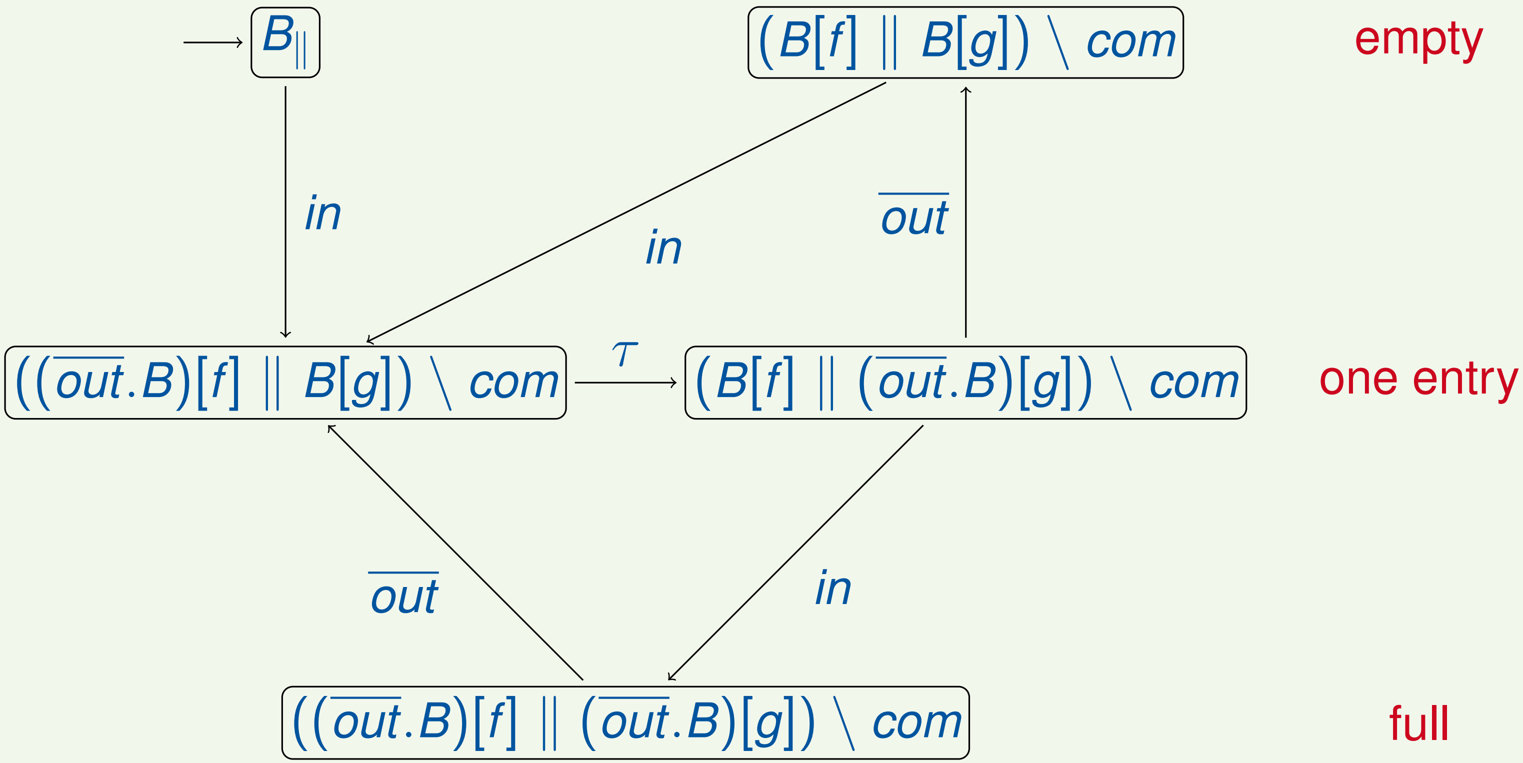
$$\text{(Res)}\ \frac{P \xrightarrow{\alpha} P' \quad (\alpha, \overline{\alpha} \notin L)}{P \setminus L \xrightarrow{\alpha} P' \setminus L} \qquad \text{(Rel)}\ \frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \qquad \text{(Call)}\ \frac{P \xrightarrow{\alpha} P' \quad (C = P)}{C \xrightarrow{\alpha} P'}$$

Recap: Calculus of Communicating Systems

Semantics of CCS II

Example (continued)

Complete LTS of parallel two-place buffer:



Outline of Lecture 3

Recap: Calculus of Communicating Systems

Infinite State Spaces

Process Traces

Hennessy-Milner Logic

Closure under Negation

HML and Process Traces

Infinite State Spaces

The Power of Recursive Definitions

So far: only finite state spaces

The Power of Recursive Definitions

So far: only finite state spaces

Example 3.1 (Counter)

$$C = up.(C \parallel down.\mathrm{nil})$$

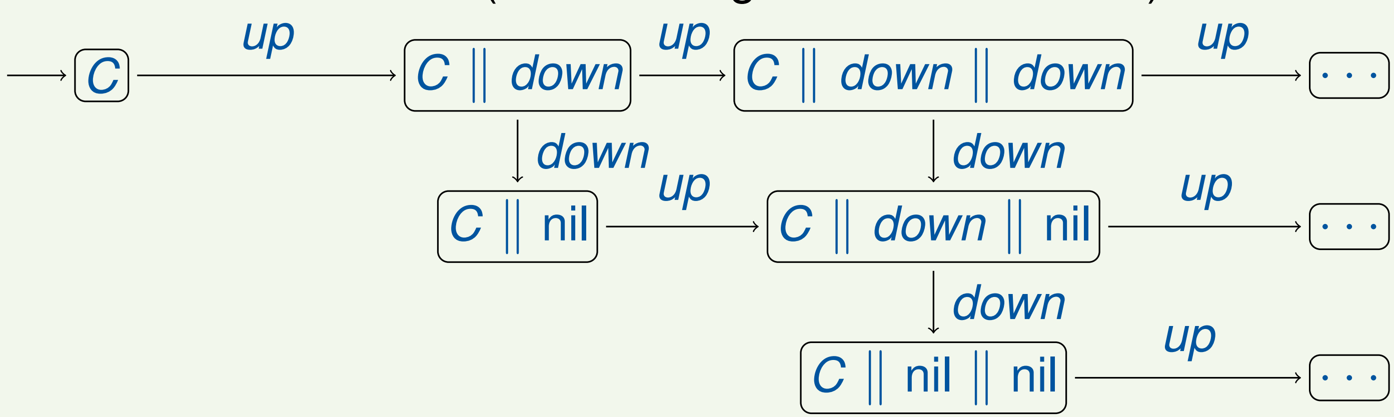
The Power of Recursive Definitions

So far: only finite state spaces

Example 3.1 (Counter)

$$C = up.(C \parallel down.\text{nil})$$

gives rise to infinite LTS (abbreviating $down := down.\text{nil}$):



Software Modeling and Verification Chair
RWTH AACHEN UNIVERSITY

Infinite State Spaces

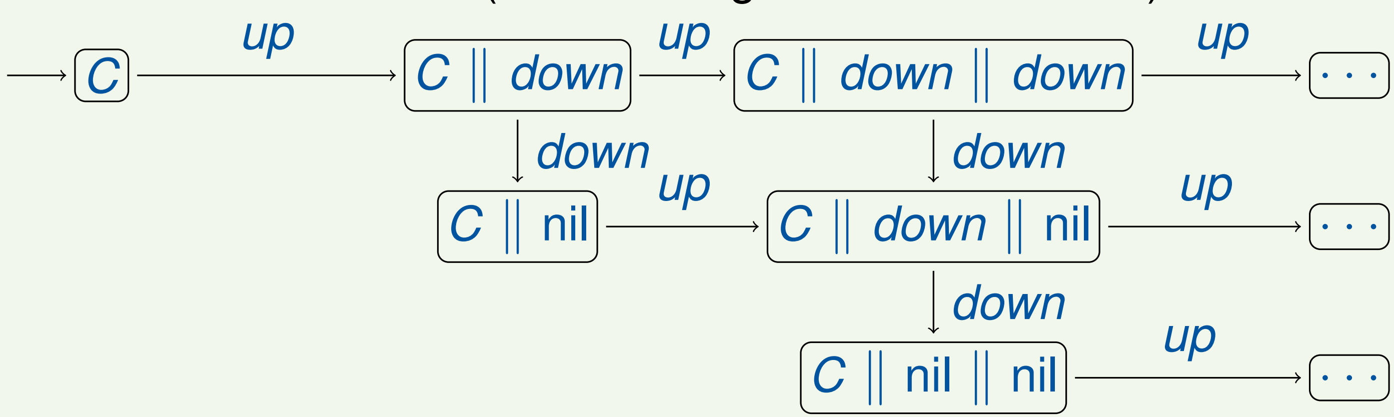
The Power of Recursive Definitions

So far: only finite state spaces

Example 3.1 (Counter)

$$C = up.(C \parallel down.\mathrm{nil})$$

gives rise to infinite LTS (abbreviating $down := down.\mathrm{nil}$):



Sequential "specification": $C_0 = up.C_1$

$$C_n = up.C_{n+1} + down.C_{n-1} \qquad (n > 0)$$

Outline of Lecture 3

Process Traces I

Goal: reduce processes to the action sequences they can perform

Definition 3.2 (Trace language)

For every $P \in Prc$, let

$$Tr(P) := \{w \in Act^* \mid \text{ex. } P' \in Prc \text{ such that } P \xrightarrow{w} P'\}$$

be the trace language of P (where $\xrightarrow{w} := \xrightarrow{a_1} \circ \ldots \circ \xrightarrow{a_n}$ for $w = a_1 \ldots a_n$).

$P, Q \in Prc$ are called trace equivalent if $Tr(P) = Tr(Q)$.

Process Traces I

Goal: reduce processes to the action sequences they can perform

Definition 3.2 (Trace language)

For every $P \in Prc$, let

$$Tr(P) := \{w \in Act^* \mid \text{ex. } P' \in Prc \text{ such that } P \xrightarrow{w} P'\}$$

be the trace language of P (where $\xrightarrow{w} := \xrightarrow{a_1} \circ \ldots \circ \xrightarrow{a_n}$ for $w = a_1 \ldots a_n$).

$P, Q \in Prc$ are called trace equivalent if $Tr(P) = Tr(Q)$.

Example 3.3 (One-place buffer)

$B = in.\overline{out}.B$

$\implies Tr(B) = (in \cdot \overline{out})^* \cdot (in + \varepsilon)$

Process Traces II

Remarks:

- The trace language of $P \in Prc$ is accepted by the LTS of P, interpreted as a (finite or infinite) automaton with initial state P and where every state is final.

Process Traces II

Remarks:

- The trace language of $P \in Prc$ is accepted by the LTS of P, interpreted as a (finite or infinite) automaton with initial state P and where every state is final.
- Trace equivalence is obviously an equivalence relation
(i.e., reflexive, symmetric, and transitive).

Process Traces II

Remarks:

- The trace language of $P \in Prc$ is accepted by the LTS of P, interpreted as a (finite or infinite) automaton with initial state P and where every state is final.
- Trace equivalence is obviously an equivalence relation (i.e., reflexive, symmetric, and transitive).
- Trace equivalence identifies processes with identical LTSs: the trace language of a process consists of the (finite) paths in the LTS. Thus:

$$LTS(P) = LTS(Q) \implies Tr(P) = Tr(Q)$$

Process Traces II

Remarks:

- The trace language of $P \in Prc$ is accepted by the LTS of P, interpreted as a (finite or infinite) automaton with initial state P and where every state is final.
- Trace equivalence is obviously an equivalence relation (i.e., reflexive, symmetric, and transitive).
- Trace equivalence identifies processes with identical LTSs: the trace language of a process consists of the (finite) paths in the LTS. Thus:

$$LTS(P) = LTS(Q) \implies Tr(P) = Tr(Q)$$

- Later we will see: trace equivalence is too coarse, i.e., identifies too many processes $\implies$ bisimulation

Outline of Lecture 3

Recap: Calculus of Communicating Systems

Infinite State Spaces

Process Traces

Hennessy-Milner Logic

Closure under Negation

HML and Process Traces

Motivation

Goal: check processes for simple properties

- action a is initially enabled
- action b is initially disabled
- a deadlock never occurs
- always sends a reply after receiving a request

Hennessy-Milner Logic

Motivation

Goal: check processes for simple properties

- action *a* is initially enabled
- action *b* is initially disabled
- a deadlock never occurs
- 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 3.4 (Syntax of HML)

The set *HMF* of Hennessy-Milner formulae over a set of actions *Act* is defined by the following syntax:

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

where $\alpha \in Act$.

Meaning of HML Constructs

- All processes satisfy tt.

Meaning of HML Constructs

- All processes satisfy tt.
- No process satisfies ff.

Meaning of HML Constructs

- All processes satisfy tt.
- No process satisfies ff.
- A process satisfies $F \land G$ iff it satisfies F and G.

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.

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 affords an α-labelled transition to a state satisfying F (possibility).

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 affords 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).

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 affords 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, $\langle\emptyset\rangle F := \mathrm{ff}$ and $[\emptyset]F := \mathrm{tt}$

Semantics of HML

Definition 3.5 (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:

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

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

$$\langle\cdot\alpha\cdot\rangle(T) := \{s \in S \mid \exists s' \in T : s \stackrel{\alpha}{\longrightarrow} s'\}$$
$$[\cdot\alpha\cdot](T) := \{s \in S \mid \forall s' \in S : s \stackrel{\alpha}{\longrightarrow} s' \implies s' \in T\}$$

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

Definition 3.5 (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:

$$[\![tt]\!] := S \qquad [\![ff]\!] := \emptyset$$
$$[\![F_1 \land F_2]\!] := [\![F_1]\!] \cap [\![F_2]\!] \qquad [\![F_1 \lor F_2]\!] := [\![F_1]\!] \cup [\![F_2]\!]$$
$$[\![\langle\alpha\rangle F]\!] := \langle\cdot\alpha\cdot\rangle([\![F]\!]) \qquad [\![[\alpha]F]\!] := [\cdot\alpha\cdot]([\![F]\!])$$

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

$$\langle\cdot\alpha\cdot\rangle(T) := \{s \in S \mid \exists s' \in T : s \xrightarrow{\alpha} s'\}$$
$$[\cdot\alpha\cdot](T) := \{s \in S \mid \forall s' \in S : s \xrightarrow{\alpha} s' \implies s' \in T\}$$

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.

Example 3.6 ($\langle\cdot\alpha\cdot\rangle$, $[\cdot\alpha\cdot]$ operators)

on the board

Simple Properties Revisited

Example 3.7

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

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

Simple Properties Revisited

Example 3.7

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

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

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

$$\begin{aligned}
[\![[b]\mathrm{ff}]\!] &= [\cdot b \cdot][\![\mathrm{ff}]\!] = [\cdot b \cdot](\emptyset) \\
&= \{s \in S \mid \forall s' \in S : s \xrightarrow{b} s' \implies s' \in \emptyset\} \\
&= \{s \in S \mid \nexists s' \in S : s \xrightarrow{b} s'\} =: \{s \in S \mid s \stackrel{b}{\not\longrightarrow}\}
\end{aligned}$$

Simple Properties Revisited

Example 3.7

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

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

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

$$\begin{aligned} [\![[b]\mathrm{ff}]\!] &= [\cdot b \cdot][\![\mathrm{ff}]\!] = [\cdot b \cdot](\emptyset) \\ &= \{s \in S \mid \forall s' \in S : s \xrightarrow{b} s' \implies s' \in \emptyset\} \\ &= \{s \in S \mid \nexists s' \in S : s \xrightarrow{b} s'\} =: \{s \in S \mid s \stackrel{b}{\not\longrightarrow}\} \end{aligned}$$

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

Simple Properties Revisited

Example 3.7

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

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

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

$$\begin{aligned}[\![[b]\mathrm{ff}]\!] &= [\cdot b\cdot][\![\mathrm{ff}]\!] = [\cdot b\cdot](\emptyset)\\ &= \{s\in S \mid \forall s'\in S : s\xrightarrow{b} s' \implies s'\in\emptyset\}\\ &= \{s\in S \mid \nexists s'\in S : s\xrightarrow{b} s'\} =: \{s\in S \mid s\stackrel{b}{\not\longrightarrow}\}\end{aligned}$$

3. Absence of deadlock:
   - initially: $\langle Act\rangle\mathrm{tt}$
   - always: later (requires recursion)
4. Responsiveness:
   - initially: $[request]\langle\overline{reply}\rangle\mathrm{tt}$
   - always: later (requires recursion)

Outline of Lecture 3

Closure under Negation

Observation: negation is *not* one of the HML constructs
Reason: HML is closed under negation

Closure under Negation

Observation: negation is *not* one of the HML constructs
Reason: HML is closed under negation

Lemma 3.8

For every $F \in HMF$ *there exists* $F^c \in HMF$ *such that* $[\![F^c]\!] = S \setminus [\![F]\!]$ *for every LTS* $(S, Act, \longrightarrow)$.

Closure under Negation

Observation: negation is *not* one of the HML constructs
Reason: HML is closed under negation

Lemma 3.8

For every $F \in HMF$ there exists $F^c \in HMF$ such that $[\![F^c]\!] = S \setminus [\![F]\!]$ for every LTS $(S, Act, \longrightarrow)$.

Proof.

Definition of F^c:

$$\begin{aligned} tt^c &:= ff & ff^c &:= tt \\ (F_1 \wedge F_2)^c &:= F_1^c \vee F_2^c & (F_1 \vee F_2)^c &:= F_1^c \wedge F_2^c \\ (\langle\alpha\rangle F)^c &:= [\alpha]F^c & ([\alpha]F)^c &:= \langle\alpha\rangle F^c \end{aligned}$$

Closure under Negation

Observation: negation is *not* one of the HML constructs
Reason: HML is closed under negation

Lemma 3.8

For every $F \in HMF$ there exists $F^c \in HMF$ such that $[\![F^c]\!] = S \setminus [\![F]\!]$ for every LTS $(S, Act, \longrightarrow)$.

Proof.

Definition of F^c:

$$\begin{aligned} \mathrm{tt}^c &:= \mathrm{ff} & \mathrm{ff}^c &:= \mathrm{tt} \\ (F_1 \wedge F_2)^c &:= F_1^c \vee F_2^c & (F_1 \vee F_2)^c &:= F_1^c \wedge F_2^c \\ (\langle\alpha\rangle F)^c &:= [\alpha]F^c & ([\alpha]F)^c &:= \langle\alpha\rangle F^c \end{aligned}$$

$[\![F^c]\!] = S \setminus [\![F]\!]$: on the board □

Outline of Lecture 3

HML and Process Traces

Lemma 3.9

Let $(Prc, Act, \longrightarrow)$ *be an LTS, and let* $P, Q \in Prc$ *satisfy the same HMF (i.e.,* $\forall F \in HMF : P \models F \iff Q \models F$*). Then* $Tr(P) = Tr(Q)$.

HML and Process Traces

Lemma 3.9

Let $(Prc, Act, \longrightarrow)$ *be an LTS, and let* $P, Q \in Prc$ *satisfy the same HMF (i.e.,* $\forall F \in HMF : P \models F \iff Q \models F$*). Then* $Tr(P) = Tr(Q)$.

Proof.

on the board □

HML and Process Traces

Lemma 3.9

Let $(Prc, Act, \longrightarrow)$ be an LTS, and let $P, Q \in Prc$ satisfy the same HMF (i.e., $\forall F \in HMF : P \models F \iff Q \models F$). Then $Tr(P) = Tr(Q)$.

Proof.

on the board □

Remark: the converse does *not* hold.

Example 3.10

- Let $P := a.(b.\mathrm{nil} + c.\mathrm{nil}) \in Prc$, $Q := a.b.\mathrm{nil} + a.c.\mathrm{nil} \in Prc$
- Then $Tr(P) = Tr(Q) = \{\varepsilon, a, ab, ac\}$

HML and Process Traces

Lemma 3.9

Let $(Prc, Act, \longrightarrow)$ *be an LTS, and let* $P, Q \in Prc$ *satisfy the same HMF (i.e.,* $\forall F \in HMF : P \models F \iff Q \models F$*). Then* $Tr(P) = Tr(Q)$*.*

Proof.

on the board □

Remark: the converse does *not* hold.

Example 3.10

- Let $P := a.(b.\text{nil} + c.\text{nil}) \in Prc$, $Q := a.b.\text{nil} + a.c.\text{nil} \in Prc$
- Then $Tr(P) = Tr(Q) = \{\varepsilon, a, ab, ac\}$
- Let $F := [a](\langle b\rangle \text{tt} \wedge \langle c\rangle \text{tt}) \in HMF$
- Then $P \models F$ but $Q \not\models F$
- [Later: $P, Q \in Prc$ HML-equivalent iff bismilar]