Fair Must Testing for I/O Automata

Rob van Glabbeek

University of Edinburgh, UK
University of New South Wales, Sydney, Australia

23 September 2022

The Theory of Testing [DH84]

Consider a model of concurrency that features *automata* and *tests*.
It does not matter how they are defined, provided the following:

- Each test has a set of *states*.
- Some states are *success states*.

The Theory of Testing [DH84]

Consider a model of concurrency that features *automata* and *tests*.
It does not matter how they are defined, provided the following:

- Each test has a set of *states*.
- Some states are *success states*.
  Normally, automata are tests without success states.
  That is, tests are automata enriched with success states.

The Theory of Testing [DH84]

Consider a model of concurrency that features *automata* and *tests*.
It does not matter how they are defined, provided the following:

- Each test has a set of *states*.
- Some states are *success states*.
- It also has a set of *executions*,
  which are (annotated) sequences of states.

The Theory of Testing [DH84]

Consider a model of concurrency that features *automata* and *tests*.
It does not matter how they are defined, provided the following:

- Each test has a set of *states*.
- Some states are *success states*.
- It also has a set of *executions*,
  which are (annotated) sequences of states.
- Some of the executions are classified as *complete*.

The Theory of Testing [DH84]

Consider a model of concurrency that features *automata* and *tests*.
It does not matter how they are defined, provided the following:

- Each test has a set of *states*.
- Some states are *success states*.
- It also has a set of *executions*,
  which are (annotated) sequences of states.
- Some of the executions are classified as *complete*.
- There is a partial function $[\_\|\_]$ of type $\mathit{tests} \times \mathit{automata} \rightharpoonup \mathit{tests}$.
  $[T\|A]$ is the *application* of test T to automata A. It is of type test.

The Theory of Testing [DH84]

Consider a model of concurrency that features *automata* and *tests*.
It does not matter how they are defined, provided the following:

- Each test has a set of *states*.
- Some states are *success states*.
- It also has a set of *executions*,
  which are (annotated) sequences of states.
- Some of the executions are classified as *complete*.
- There is a partial function $[\_\|\_]$ of type $\mathit{tests} \times \mathit{automata} \rightharpoonup \mathit{tests}$.
  $[T\|A]$ is the *application* of test T to automata A. It is of type test.

Now A **may pass** T if $[T\|A]$ has an execution with a success state.

↑
is defined and

$$A \equiv_{\text{may}} B \quad :\Leftrightarrow \quad \forall T.\ (A \text{ **may pass** } T \Leftrightarrow B \text{ **may pass** } T)$$

$$\wedge\ \mathit{type}(A) = \mathit{type}(B).$$

The Theory of Testing [DH84]

Consider a model of concurrency that features *automata* and *tests*.
It does not matter how they are defined, provided the following:

- Each test has a set of *states*.
- Some states are *success states*.
- It also has a set of *executions*,
  which are (annotated) sequences of states.
- Some of the executions are classified as *complete*.
- There is a partial function $[\_\|\_]$ of type $\mathit{tests} \times \mathit{automata} \rightharpoonup \mathit{tests}$.
  $[T\|A]$ is the *application* of test T to automata A. It is of type test.

Now A **may pass** T if $[T\|A]$ has an execution with a success state.
↑
is defined and

$$A \equiv_{\mathrm{may}} B \quad :\Leftrightarrow \quad \forall T.\ (A \textbf{ may pass } T \Leftrightarrow B \textbf{ may pass } T)$$
$$A \sqsubseteq_{\mathrm{may}} B \quad :\Leftrightarrow \quad \forall T.\ (A \textbf{ may pass } T \Rightarrow B \textbf{ may pass } T) \wedge \mathit{type}(A) = \mathit{type}(B).$$

The Theory of Testing [DH84]

Consider a model of concurrency that features *automata* and *tests*.
It does not matter how they are defined, provided the following:

- Each test has a set of *states*.
- Some states are *success states*.
- It also has a set of *executions*,
  which are (annotated) sequences of states.
- Some of the executions are classified as *complete*.
- There is a partial function $[\_\|\_]$ of type $tests \times automata \rightharpoonup tests$.
  $[T\|A]$ is the *application* of test T to automata A. It is of type test.

Now A **may pass** T if $[T\|A]$ has an execution with a success state.
A **must pass** T if each complete execution of $[T\|A]$ has a success state.

$$A \equiv_{\text{must}} B \quad :\Leftrightarrow \quad \forall T.\ (A \textbf{ must pass } T \Leftrightarrow B \textbf{ must pass } T)$$

$$A \sqsubseteq_{\text{must}} B \quad :\Leftrightarrow \quad \forall T.\ (A \textbf{ must pass } T \Rightarrow B \textbf{ must pass } T) \wedge \mathit{type}(A) = \mathit{type}(B).$$

The Theory of Testing [DH84]

Consider a model of concurrency that features *automata* and *tests*.
It does not matter how they are defined, provided the following:

- Each test has a set of *states*.
- Some states are *success states*.
- It also has a set of *executions*,
  which are (annotated) sequences of states.
- Some of the executions are classified as *complete*.
- There is a partial function $[\_\|\_]$ of type $\mathit{tests} \times \mathit{automata} \rightharpoonup \mathit{tests}$.
  $[T\|A]$ is the *application* of test T to automata A. It is of type test.

Now A **may pass** T if $[T\|A]$ has an execution with a success state.
A **must pass** T if each complete execution of $[T\|A]$ has a success state.
A **should pass** T [BRV95,NC95] if each finite execution of $[T\|A]$ can be extended into an execution with a success state.

$$A \equiv_{\text{should}} B \quad :\Leftrightarrow \quad \forall T.\ (A \textbf{ should pass } T \Leftrightarrow B \textbf{ should pass } T)$$

$$A \sqsubseteq_{\text{should}} B \quad :\Leftrightarrow \quad \forall T.\ (A \textbf{ should pass } T \Rightarrow B \textbf{ should pass } T) \wedge \mathit{type}(A) = \mathit{type}(B).$$

Testing for CCS [DH84]

automata: [. . .]
tests: [. . .]
states of a test: [. . .]
success states: [. . .]
executions: [. . .]
complete executions: [. . .]
application: [. . .]

Testing for CCS [DH84]

automata: CCS expressions, over an alphabet $\mathcal{A}$ of actions
tests: [...]
states of a test: [...]
success states: [...]
executions: [...]
complete executions: [...]
application: [...]

Testing for CCS [DH84]

automata: CCS expressions, over an alphabet $\mathcal{A}$ of actions
tests: CCS expressions, over the alphabet $\mathcal{A} \uplus \{w\}$
states of a test: [. . .]
success states: [. . .]
executions: [. . .]
complete executions: [. . .]
application: [. . .]

Testing for CCS [DH84]

automata: CCS expressions, over an alphabet $\mathcal{A}$ of actions
tests: CCS expressions, over the alphabet $\mathcal{A} \uplus \{w\}$
states of a test: the reachable CCS expressions
success states: [. . .]
executions: [. . .]
complete executions: [. . .]
application: [. . .]

Testing for CCS [DH84]

automata: CCS expressions, over an alphabet $\mathcal{A}$ of actions
tests: CCS expressions, over the alphabet $\mathcal{A} \uplus \{w\}$
states of a test: the reachable CCS expressions
success states: those states in which w is enabled
executions: [. . .]
complete executions: [. . .]
application: [. . .]

Testing for CCS [DH84]

automata: CCS expressions, over an alphabet $\mathcal{A}$ of actions
tests: CCS expressions, over the alphabet $\mathcal{A} \uplus \{w\}$
states of a test: the reachable CCS expressions
success states: those states in which w is enabled
executions: determined by the operational semantics of CCS
complete executions: [. . .]
application: [. . .]

Testing for CCS [DH84]

automata: CCS expressions, over an alphabet $\mathcal{A}$ of actions
tests: CCS expressions, over the alphabet $\mathcal{A} \uplus \{w\}$
states of a test: the reachable CCS expressions
success states: those states in which w is enabled
executions: determined by the operational semantics of CCS
complete executions: either infinite, of ending in deadlock
application: [. . .]
Deadlock: a state without outgoing transitions

Testing for CCS [DH84]

automata: CCS expressions, over an alphabet $\mathcal{A}$ of actions
tests: CCS expressions, over the alphabet $\mathcal{A} \uplus \{w\}$
states of a test: the reachable CCS expressions
success states: those states in which w is enabled
executions: determined by the operational semantics of CCS
complete executions: either infinite, of ending in deadlock
application: the CCS context $(\_|\_)\backslash\mathcal{A}$

Automata

An *automaton/test* A is $(\mathit{acts}(A), \mathit{states}(A), \mathit{start}(A), \mathit{steps}(A))$ with

- $\mathit{acts}(A)$ a set of *actions*,

- $\mathit{states}(A)$ a set of *states*,
- $\mathit{start}(A) \subseteq \mathit{states}(A)$ a nonempty set of *start states*, and
- $\mathit{steps}(A) \subseteq \mathit{states}(A) \times \mathit{acts}(A) \times \mathit{states}(A)$ a *transition relation*.

Automata are also known as *process graphs*,
state/transition diagrams, or
sets of states in *labelled transition systems*.

Automata

An *automaton/test* A is $(\mathit{acts}(A), \mathit{states}(A), \mathit{start}(A), \mathit{steps}(A))$ with

- $\mathit{acts}(A)$ a set of *actions*, partitioned into two sets $\mathit{ext}(A)$ and $\mathit{int}(A)$ of *external actions* and *internal actions*, respectively,
- $\mathit{states}(A)$ a set of *states*,
- $\mathit{start}(A) \subseteq \mathit{states}(A)$ a nonempty set of *start states*, and
- $\mathit{steps}(A) \subseteq \mathit{states}(A) \times \mathit{acts}(A) \times \mathit{states}(A)$ a *transition relation*.

Automata are also known as *process graphs*,
state/transition diagrams, or
sets of states in *labelled transition systems*.

An *execution* of an I/O automaton A is an alternating sequence $\alpha = s_0, a_1, s_1, a_2, \ldots$ of states and actions, either being infinite or ending with a state, such that $s_0 \in \mathit{start}(A)$ and $(s_i, a_{i+1}, s_{i+1}) \in \mathit{steps}(A)$ for all $i < \mathit{length}(\alpha)$.

Testing for Automata

automata: CCS expressions, over an alphabet $\mathcal{A}$ of actions
tests: CCS expressions, over the alphabet $\mathcal{A} \uplus \{w\}$
states of a test: the reachable CCS expressions
success states: those states in which w is enabled
executions: determined by the operational semantics of CCS
complete executions: either infinite, of ending in deadlock
application: the CCS context $(T|A)\backslash\mathcal{A}$

Testing for Automata

automata: automata A (see previous slide) with $w \notin \mathit{acts}(A)$
tests: CCS expressions, over the alphabet $\mathcal{A} \uplus \{w\}$
states of a test: the reachable CCS expressions
success states: those states in which w is enabled
executions: determined by the operational semantics of CCS
complete executions: either infinite, of ending in deadlock
application: the CCS context $(T|A)\backslash\mathcal{A}$

Testing for Automata

automata: automata A (see previous slide) with $w \notin \mathit{acts}(A)$
tests: automata T (see previous slide) with $w \in \mathit{ext}(T)$
states of a test: the reachable CCS expressions
success states: those states in which w is enabled
executions: determined by the operational semantics of CCS
complete executions: either infinite, of ending in deadlock
application: the CCS context $(T|A)\backslash\mathcal{A}$

Testing for Automata

automata: automata A (see previous slide) with $w \notin \mathit{acts}(A)$
tests: automata T (see previous slide) with $w \in \mathit{ext}(T)$
states of a test: $\mathit{states}(A)$
success states: those states in which w is enabled
executions: determined by the operational semantics of CCS
complete executions: either infinite, of ending in deadlock
application: the CCS context $(T|A)\backslash\mathcal{A}$

Testing for Automata

automata: automata A (see previous slide) with $w \notin \mathit{acts}(A)$
tests: automata T (see previous slide) with $w \in \mathit{ext}(T)$
states of a test: $\mathit{states}(A)$
success states: those states in which w is enabled
executions: determined by the operational semantics of CCS
complete executions: either infinite, of ending in deadlock
application: the CCS context $(T|A)\backslash\mathcal{A}$

Testing for Automata

automata: automata A (see previous slide) with $w \notin \mathit{acts}(A)$
tests: automata T (see previous slide) with $w \in \mathit{ext}(T)$
states of a test: $\mathit{states}(A)$
success states: those states in which w is enabled
executions: determined by $\mathit{steps}(A)$
complete executions: either infinite, of ending in deadlock
application: the CCS context $(T|A)\backslash\mathcal{A}$

Testing for Automata

automata: automata A (see previous slide) with $w \notin \mathit{acts}(A)$
tests: automata T (see previous slide) with $w \in \mathit{ext}(T)$
states of a test: $\mathit{states}(A)$
success states: those states in which w is enabled
executions: determined by $\mathit{steps}(A)$
complete executions: either infinite, of ending in deadlock
application: the CCS context $(T|A)\backslash\mathcal{A}$

Testing for Automata

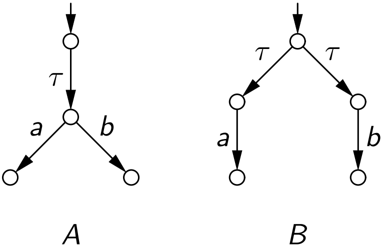
automata: automata A (see previous slide) with $w \notin acts(A)$
tests: automata T (see previous slide) with $w \in ext(T)$
states of a test: $states(A)$
success states: those states in which w is enabled
executions: determined by $steps(A)$
complete executions: either infinite, of ending in deadlock
application: the CCS context $(T|A)\backslash\mathcal{A}$ with $\mathcal{A} = ext(A) \cup ext(T)\backslash\{w\}$

Testing for Automata

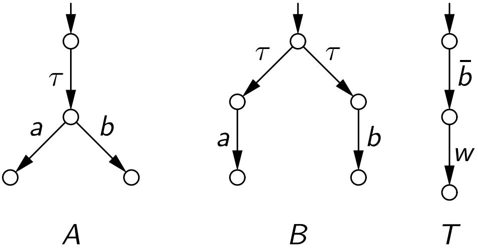
automata: automata A (see previous slide) with $w \notin \mathit{acts}(A)$
tests: automata T (see previous slide) with $w \in \mathit{ext}(T)$
states of a test: $\mathit{states}(A)$
success states: those states in which w is enabled
executions: determined by $\mathit{steps}(A)$
complete executions: either infinite, of ending in deadlock
application: the CCS context $(T|A)\backslash\mathcal{A}$ with $\mathcal{A} = \mathit{ext}(A) \cup \mathit{ext}(T)\backslash\{w\}$

The theory of testing for CCS is a special case
of the theory of testing for automata.

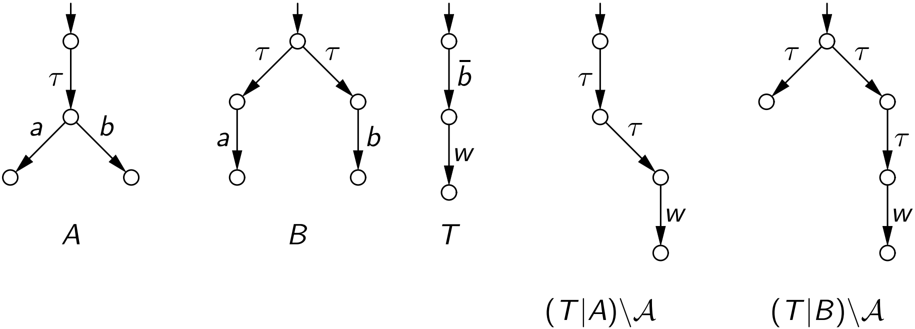
Example: discerning branching time



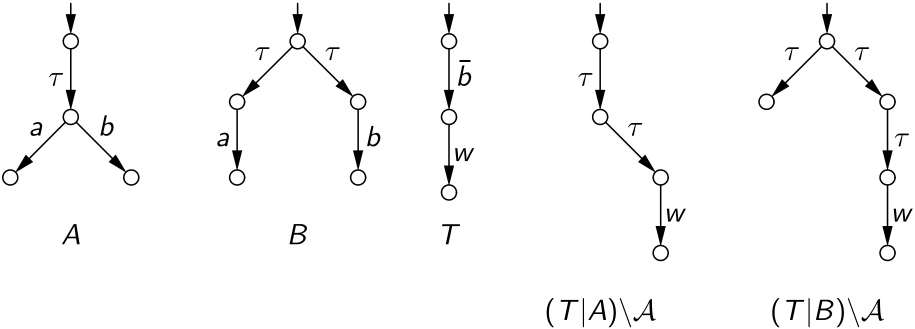
Example: discerning branching time



Example: discerning branching time

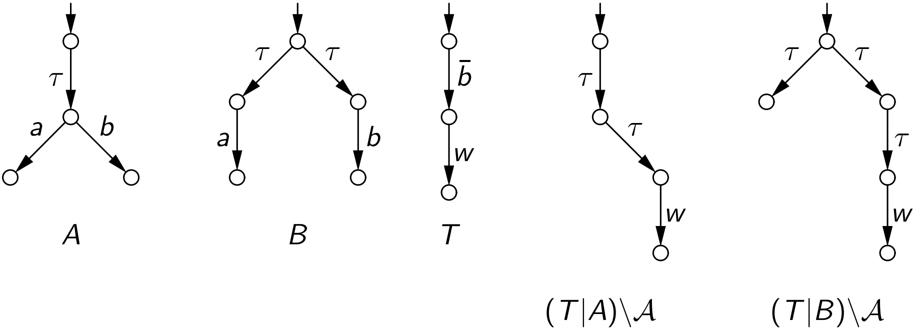


Example: discerning branching time



A **must pass** T but $\neg(B$ **must pass** $T)$

Example: discerning branching time



A **must pass** T but $\neg(B$ **must pass** $T)$

Thus $A \not\sqsubseteq_{\mathrm{must}} B$ and $A \not\eqsim_{\mathrm{must}} B$.

Testing for Automata

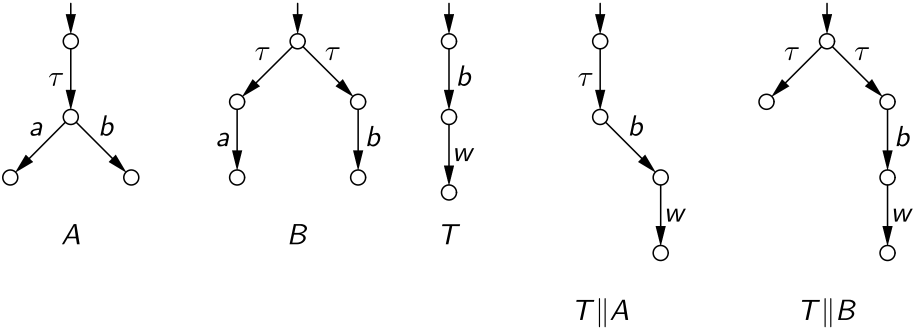
automata: automata A with $w \notin \mathit{acts}(A)$
tests: automata T with $w \in \mathit{ext}(T)$
states of a test: $\mathit{states}(A)$
success states: those states in which w is enabled
executions: determined by $\mathit{steps}(A)$
complete executions: either infinite, of ending in deadlock
application: the CCS context $(T|A)\backslash\mathcal{A}$ with $\mathcal{A} = \mathit{ext}(A) \cup \mathit{ext}(T)\backslash\{w\}$

Testing for Automata

automata: automata A with $w \notin \mathit{acts}(A)$
tests: automata T with $w \in \mathit{ext}(T)$
states of a test: $\mathit{states}(A)$
success states: those states in which w is enabled
executions: determined by $\mathit{steps}(A)$
complete executions: either infinite, of ending in deadlock
application: the CSP parallel composition $T \| A$

This operator enforces synchronisation on $\mathit{ext}(T) \cap \mathit{ext}(A)$.

Example: discerning branching time



A **must pass** T but $\neg(B$ **must pass** $T)$

Thus $A \not\sqsubseteq_{\text{must}} B$ and $A \not\eqsim_{\text{must}} B$.

I/O Automata [LT89]

Automata $A = (acts(A), states(A), start(A), steps(A) \qquad\qquad)$
in which

- $acts(A)$ is partitioned into $in(A)$ and $out(A)$,
- such that in each state each action from $in(A)$ is enabled,

I/O Automata [LT89]

Automata $A = (\mathit{acts}(A), \mathit{states}(A), \mathit{start}(A), \mathit{steps}(A), \mathit{part}(A))$ in which

- $\mathit{acts}(A)$ is partitioned into $\mathit{in}(A)$ and $\mathit{out}(A)$,
- such that in each state each action from $\mathit{in}(A)$ is enabled,
- equipped with a partition $\mathit{part}(A)$ of the set $\mathit{local}(A) := \mathit{out}(A) \cup \mathit{int}(A)$ of *locally-controlled actions* of A into *tasks*.

I/O Automata [LT89]

Automata $A = (\mathit{acts}(A), \mathit{states}(A), \mathit{start}(A), \mathit{steps}(A), \mathit{part}(A))$ in which

- $\mathit{acts}(A)$ is partitioned into $\mathit{in}(A)$ and $\mathit{out}(A)$,
- such that in each state each action from $\mathit{in}(A)$ is enabled,
- equipped with a partition $\mathit{part}(A)$ of the set $\mathit{local}(A) := \mathit{out}(A) \cup \mathit{int}(A)$ of *locally-controlled actions* of A into *tasks*.

An execution α of A is *fair* if, for each suffix α' of α
and each task $\mathcal{T} \in \mathit{part}(A)$,
if $\mathcal{T}$ is enabled in each state of α',
then α' contains an action from $\mathcal{T}$.

I/O Automata [LT89]

Automata $A = (\mathit{acts}(A), \mathit{states}(A), \mathit{start}(A), \mathit{steps}(A), \mathit{part}(A))$ in which

- $\mathit{acts}(A)$ is partitioned into $\mathit{in}(A)$ and $\mathit{out}(A)$,
- such that in each state each action from $\mathit{in}(A)$ is enabled,
- equipped with a partition $\mathit{part}(A)$ of the set $\mathit{local}(A) := \mathit{out}(A) \cup \mathit{int}(A)$ of *locally-controlled actions* of A into *tasks*.

An execution α of A is *fair* if, for each suffix α' of α
and each task $\mathcal{T} \in \mathit{part}(A)$,
if $\mathcal{T}$ is enabled in each state of α',
then α' contains an action from $\mathcal{T}$.

Parallel composition $A \| B$ of I/O automata is as for CSP, or standard automata, but is defined only when $\mathit{out}(A) \cap \mathit{out}(B) = \emptyset$.

Fundamental Preorders on I/O Automata [LT89]

I/O automata are a *typed* model of concurrency: automata will be compared only when they have the same input and output actions.

Fundamental Preorders on I/O Automata [LT89]

I/O automata are a *typed* model of concurrency: automata will be compared only when they have the same input and output actions.

$\mathit{trace}(\alpha)$ is the (in)finite sequence of external actions in execution α.
$\mathit{fintraces}(A) := \{\mathit{trace}(\alpha) \mid \alpha$ is a finite execution of $A\}$.
$\mathit{fairtraces}(A) := \{\mathit{trace}(\alpha) \mid \alpha$ is a fair execution of $A\}$.

$$S \sqsubseteq_T I \;:\Leftrightarrow\; \mathit{in}(S){=}\mathit{in}(I) \wedge \mathit{out}(S){=}\mathit{out}(I) \wedge \mathit{fintraces}(I) \subseteq \mathit{fintraces}(S)$$

$$S \sqsubseteq_F I \;:\Leftrightarrow\; \mathit{in}(S){=}\mathit{in}(I) \wedge \mathit{out}(S){=}\mathit{out}(I) \wedge \mathit{fairtraces}(I) \subseteq \mathit{fairtraces}(S)\,.$$

One writes $A \equiv_T B$ if $A \sqsubseteq_T B \wedge B \sqsubseteq_T A$, and similarly for $\equiv_F$.

Fundamental Preorders on I/O Automata [LT89]

I/O automata are a *typed* model of concurrency: automata will be compared only when they have the same input and output actions.

$\mathit{trace}(\alpha)$ is the (in)finite sequence of external actions in execution α.
$\mathit{fintraces}(A) := \{\mathit{trace}(\alpha) \mid \alpha \text{ is a finite execution of } A\}$.
$\mathit{fairtraces}(A) := \{\mathit{trace}(\alpha) \mid \alpha \text{ is a fair execution of } A\}$.

$$S \sqsubseteq_T I \;:\Leftrightarrow\; \mathit{in}(S){=}\mathit{in}(I) \wedge \mathit{out}(S){=}\mathit{out}(I) \wedge \mathit{fintraces}(I) \subseteq \mathit{fintraces}(S)$$

$$S \sqsubseteq_F I \;:\Leftrightarrow\; \mathit{in}(S){=}\mathit{in}(I) \wedge \mathit{out}(S){=}\mathit{out}(I) \wedge \mathit{fairtraces}(I) \subseteq \mathit{fairtraces}(S)\,.$$

One writes $A \equiv_T B$ if $A \sqsubseteq_T B \wedge B \sqsubseteq_T A$, and similarly for $\equiv_F$.

These preorders capture *safety* and *liveness* properties, respectively.

Fundamental Preorders on I/O Automata [LT89]

I/O automata are a *typed* model of concurrency: automata will be compared only when they have the same input and output actions.

$\mathit{trace}(\alpha)$ is the (in)finite sequence of external actions in execution α.
$\mathit{fintraces}(A) := \{\mathit{trace}(\alpha) \mid \alpha \text{ is a finite execution of } A\}$.
$\mathit{fairtraces}(A) := \{\mathit{trace}(\alpha) \mid \alpha \text{ is a fair execution of } A\}$.

$$S \sqsubseteq_T I \;\; :\Leftrightarrow \;\; \mathit{in}(S)=\mathit{in}(I) \wedge \mathit{out}(S)=\mathit{out}(I) \wedge \mathit{fintraces}(I) \subseteq \mathit{fintraces}(S)$$

$$S \sqsubseteq_F I \;\; :\Leftrightarrow \;\; \mathit{in}(S)=\mathit{in}(I) \wedge \mathit{out}(S)=\mathit{out}(I) \wedge \mathit{fairtraces}(I) \subseteq \mathit{fairtraces}(S)\,.$$

One writes $A \equiv_T B$ if $A \sqsubseteq_T B \wedge B \sqsubseteq_T A$, and similarly for $\equiv_F$.

These preorders capture *safety* and *liveness* properties, respectively.

By [GH19, Thm. 6.1] each finite execution can be extended into a fair execution. As a consequence, $A \sqsubseteq_F B \Rightarrow A \sqsubseteq_T B$.

Testing for I/O Automata [Seg97]

automata: automata A with $w \notin \mathit{acts}(A)$
tests: automata T with $w \in \mathit{acts}(T)$
states of a test: $\mathit{states}(A)$
success states: those states in which w is enabled
executions: determined by $\mathit{steps}(A)$
complete executions: either infinite, of ending in deadlock
application: the parallel composition $T \| A$

Testing for I/O Automata [Seg97]

automata: I/O automata A with $w \notin acts(A)$
tests: I/O automata T with $w \in acts(T)$
states of a test: $states(A)$
success states: those states in which w is enabled
executions: determined by $steps(A)$
complete executions: either infinite, of ending in deadlock
application: the parallel composition $T \| A$

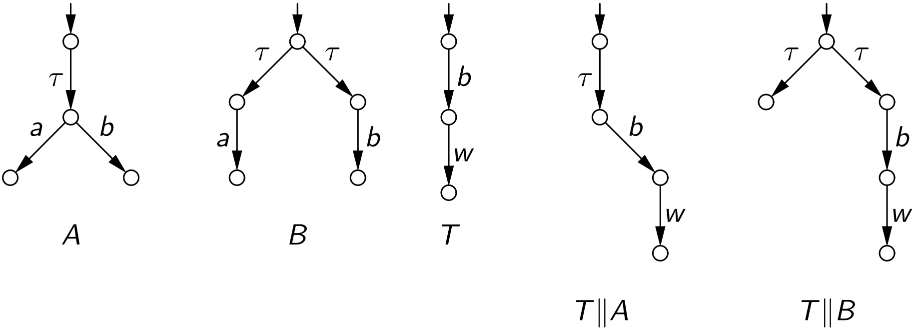
Testing for I/O Automata [Seg97]

automata: I/O automata A with $w \notin acts(A)$
tests: I/O automata T with $w \in out(A)$
states of a test: $states(A)$
success states: those states in which w is enabled
executions: determined by $steps(A)$
complete executions: either infinite, of ending in deadlock
application: the parallel composition $T \| A$

Testing for I/O Automata [Seg97]

automata: I/O automata A with $w \notin acts(A)$
tests: I/O automata T with $w \in out(A)$
states of a test: $states(A)$
success states: those states in which w is enabled
executions: determined by $steps(A)$
complete executions: either infinite, of ending in deadlock
application: the I/O parallel composition $T \| A$

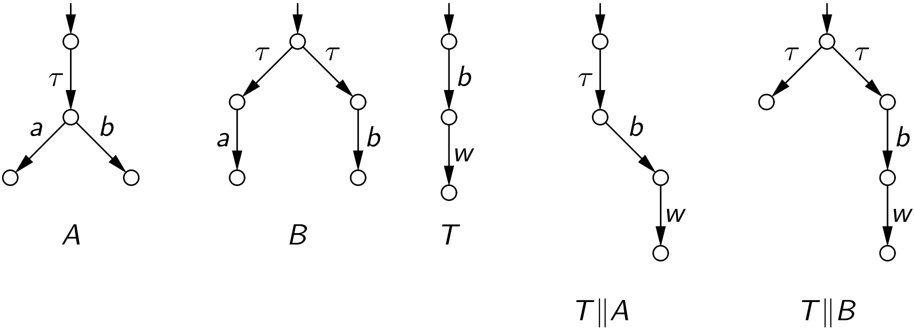
Example: discerning branching time impossible



A **must pass** T but $\neg(B$ **must pass** $T)$

Thus $A \not\sqsubseteq_{\mathrm{must}} B$ and $A \not\eqsim_{\mathrm{must}} B$.

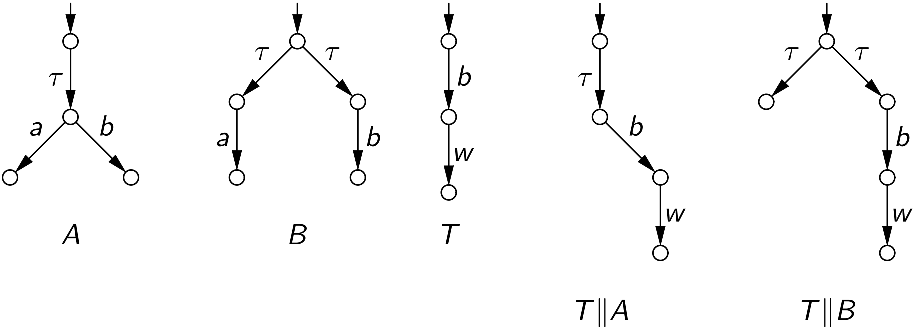
Example: discerning branching time impossible



Such a T does not exists:

- if $a \notin ext(A)$ or $a \notin ext(T)$ then neither A nor B **must pass** T.
- $a \in in(A)$ or $a \in in(T)$ violates input enabledness
- if $a \in out(A) \cap out(T)$ then $T \| A$ is undefined.

Example: discerning branching time impossible



Such a T does not exists:

- if $a \notin$ *ext*(A) or $a \notin$ *ext*(T) then neither A nor B **must pass** T.
- $a \in$ *in*(A) or $a \in$ *in*(T) violates input enabledness
- if $a \in$ *out*$(A) \cap$ *out*(T) then $T \| A$ is undefined.

In fact, $A \equiv_{\text{must}} B$.

Safety and Liveness

The native preorders $\sqsubseteq_T$ and $\sqsubseteq_F$ from [LT89] are meant to capture safety and liveness properties, respectively.

Safety and Liveness

The native preorders $\sqsubseteq_T$ and $\sqsubseteq_F$ from [LT89] are meant to capture safety and liveness properties, respectively.

May and must testing can be seen as aiming at the same.

Safety and Liveness

The native preorders $\sqsubseteq_T$ and $\sqsubseteq_F$ from [LT89] are meant to capture safety and liveness properties, respectively.

May and must testing can be seen as aiming at the same.

Indeed, we have $A \equiv_{\mathrm{may}} B$ iff $A \equiv_T B$.
(Trivial, or see my paper.)

Safety and Liveness

The native preorders $\sqsubseteq_T$ and $\sqsubseteq_F$ from [LT89] are meant to capture safety and liveness properties, respectively.

May and must testing can be seen as aiming at the same.

Indeed, we have $A \equiv_{\mathrm{may}} B$ iff $A \equiv_T B$.
(Trivial, or see my paper.)

Yet, $\equiv_{\mathrm{must}}$ and $\equiv_F$ are incomparable [Seg97].

Safety and Liveness

The native preorders $\sqsubseteq_T$ and $\sqsubseteq_F$ from [LT89] are meant to capture safety and liveness properties, respectively.

May and must testing can be seen as aiming at the same.

Indeed, we have $A \equiv_{\mathrm{may}} B$ iff $A \equiv_T B$.
(Trivial, or see my paper.)

Yet, $\equiv_{\mathrm{must}}$ and $\equiv_F$ are incomparable [Seg97].

$A =$  $B =$  $\quad \mathit{acts}(A) = \emptyset \quad \mathit{acts}(B) = \mathit{int}(B) = \{\tau\}$.

Safety and Liveness

The native preorders $\sqsubseteq_T$ and $\sqsubseteq_F$ from [LT89] are meant to capture safety and liveness properties, respectively.

May and must testing can be seen as aiming at the same.

Indeed, we have $A \equiv_{\mathrm{may}} B$ iff $A \equiv_T B$.
(Trivial, or see my paper.)

Yet, $\equiv_{\mathrm{must}}$ and $\equiv_F$ are incomparable [Seg97].

$A =$  $B =$ 

$\mathit{acts}(A) = \emptyset$ $\mathit{acts}(B) = \mathit{int}(B) = \{\tau\}$.

Then $A \equiv_F B$, yet $A \not\sqsubseteq_{\mathrm{must}} B$.

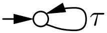
Safety and Liveness

The native preorders $\sqsubseteq_T$ and $\sqsubseteq_F$ from [LT89] are meant to capture safety and liveness properties, respectively.

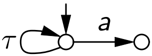
May and must testing can be seen as aiming at the same.

Indeed, we have $A \equiv_{\mathrm{may}} B$ iff $A \equiv_T B$.
(Trivial, or see my paper.)

Yet, $\equiv_{\mathrm{must}}$ and $\equiv_F$ are incomparable [Seg97].

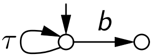
$A =$  $B =$  $\mathit{acts}(A) = \emptyset$ $\mathit{acts}(B) = \mathit{int}(B) = \{\tau\}$.

Then $A \equiv_F B$, yet $A \not\sqsubseteq_{\mathrm{must}} B$.



$\equiv_{must}$

$\not\equiv_F$



A Third Fundamental Preorder on I/O Automata [Va91]

$\mathit{trace}(\alpha)$ is the (in)finite sequence of external actions in execution α.
$\mathit{fintraces}(A) := \{\mathit{trace}(\alpha) \mid \alpha$ is a finite execution of $A\}$.
$\mathit{fairtraces}(A) := \{\mathit{trace}(\alpha) \mid \alpha$ is a fair execution of $A\}$.

$$S \sqsubseteq_T I \;:\Leftrightarrow\; \mathit{in}(S)=\mathit{in}(I) \wedge \mathit{out}(S)=\mathit{out}(I) \wedge \mathit{fintraces}(I) \subseteq \mathit{fintraces}(S)$$

$$S \sqsubseteq_F I \;:\Leftrightarrow\; \mathit{in}(S)=\mathit{in}(I) \wedge \mathit{out}(S)=\mathit{out}(I) \wedge \mathit{fairtraces}(I) \subseteq \mathit{fairtraces}(S)\,.$$

A Third Fundamental Preorder on I/O Automata [Va91]

trace(α) is the (in)finite sequence of external actions in execution α.
fintraces(A) := $\{$*trace*(α) $\mid$ α is a finite execution of $A\}$.
fairtraces(A) := $\{$*trace*(α) $\mid$ α is a fair execution of $A\}$.
An execution α is *quiescent* if it is finite and its last state enables only input actions.
qtraces(A) := $\{$*trace*(α) $\mid$ α is a quiescent execution of $A\}$.

$$S \sqsubseteq_T I \;:\Leftrightarrow\; in(S)=in(I) \wedge out(S)=out(I) \wedge fintraces(I) \subseteq fintraces(S)$$

$$S \sqsubseteq_F I \;:\Leftrightarrow\; in(S)=in(I) \wedge out(S)=out(I) \wedge fairtraces(I) \subseteq fairtraces(S)\,.$$

A Third Fundamental Preorder on I/O Automata [Va91]

trace(α) is the (in)finite sequence of external actions in execution α.
fintraces(A) := $\{$*trace*(α) $\mid \alpha$ is a finite execution of $A\}$.
fairtraces(A) := $\{$*trace*(α) $\mid \alpha$ is a fair execution of $A\}$.
An execution α is *quiescent* if it is finite and its last state enables only input actions.
qtraces(A) := $\{$*trace*(α) $\mid \alpha$ is a quiescent execution of $A\}$.

$$S \sqsubseteq_T I \;:\Leftrightarrow\; in(S)=in(I) \wedge out(S)=out(I) \wedge fintraces(I) \subseteq fintraces(S)$$

$$S \sqsubseteq_F I \;:\Leftrightarrow\; in(S)=in(I) \wedge out(S)=out(I) \wedge fairtraces(I) \subseteq fairtraces(S)\,.$$

$$S \sqsubseteq_Q I :\Leftrightarrow S \sqsubseteq_T I \wedge qtraces(I) \subseteq qtraces(S)\,.$$

A Third Fundamental Preorder on I/O Automata [Va91]

$trace(\alpha)$ is the (in)finite sequence of external actions in execution α.
$fintraces(A) := \{trace(\alpha) \mid \alpha$ is a finite execution of $A\}$.
$fairtraces(A) := \{trace(\alpha) \mid \alpha$ is a fair execution of $A\}$.
An execution α is *quiescent* if it is finite and its last state enables only input actions.
$qtraces(A) := \{trace(\alpha) \mid \alpha$ is a quiescent execution of $A\}$.

$$S \sqsubseteq_T I \;:\Leftrightarrow\; in(S){=}in(I) \wedge out(S){=}out(I) \wedge fintraces(I) \subseteq fintraces(S)$$

$$S \sqsubseteq_F I \;:\Leftrightarrow\; in(S){=}in(I) \wedge out(S){=}out(I) \wedge fairtraces(I) \subseteq fairtraces(S)\,.$$

$$S \sqsubseteq_Q I :\Leftrightarrow S \sqsubseteq_T I \wedge qtraces(I) \subseteq qtraces(S)\,.$$

Theorem [Seg97]: Let A and B be finitely branching and strongly convergent I/O automata. Then $A \sqsubseteq_{\mathrm{must}} B$ iff $A \sqsubseteq_Q B$.

Safety and Liveness

The native preorders $\sqsubseteq_T$ and $\sqsubseteq_F$ from [LT89] are meant to capture safety and liveness properties, respectively.

May and must testing can be seen as aiming at the same.

Indeed, we have $A \equiv_{\mathrm{may}} B$ iff $A \equiv_T B$.
(Trivial, or see my paper.)

Yet, $\equiv_{\mathrm{must}}$ and $\equiv_F$ are incomparable [Seg97].

Safety and Liveness

The native preorders $\sqsubseteq_T$ and $\sqsubseteq_F$ from [LT89] are meant to capture safety and liveness properties, respectively.

May and must testing can be seen as aiming at the same.

Indeed, we have $A \equiv_{\mathrm{may}} B$ iff $A \equiv_T B$.
(Trivial, or see my paper.)

Yet, $\equiv_{\mathrm{must}}$ and $\equiv_F$ are incomparable [Seg97].

In my analysis, this is because the classical theory of testing and I/O automata are based on different notions of a complete execution.

Testing for I/O Automata based on fairness

automata: I/O automata A with $w \notin \mathit{acts}(A)$
tests: I/O automata T with $w \in \mathit{out}(A)$
states of a test: $\mathit{states}(A)$
success states: those states in which w is enabled
executions: determined by $\mathit{steps}(A)$
complete executions: either infinite, of ending in deadlock
application: the I/O parallel composition $T \| A$

Testing for I/O Automata based on fairness

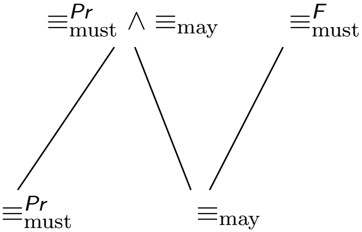
automata: I/O automata A with $w \notin acts(A)$
tests: I/O automata T with $w \in out(A)$
states of a test: $states(A)$
success states: those states in which w is enabled
executions: determined by $steps(A)$
complete executions: the fair ones
application: the I/O parallel composition $T \| A$

Testing for I/O Automata based on fairness

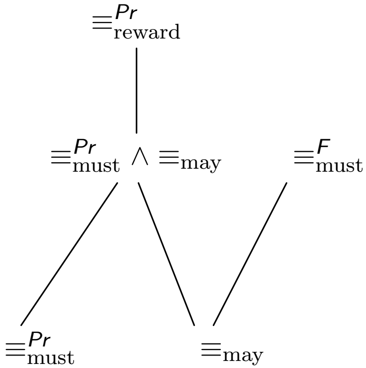
automata: I/O automata A with $w \notin \mathit{acts}(A)$
tests: I/O automata T with $w \in \mathit{out}(A)$
states of a test: $\mathit{states}(A)$
success states: those states in which w is enabled
executions: determined by $\mathit{steps}(A)$
complete executions: the fair ones
application: the I/O parallel composition $T \| A$

Theorem: Now $A \sqsubseteq_{\mathrm{must}} B$ iff $A \sqsubseteq_F B$.

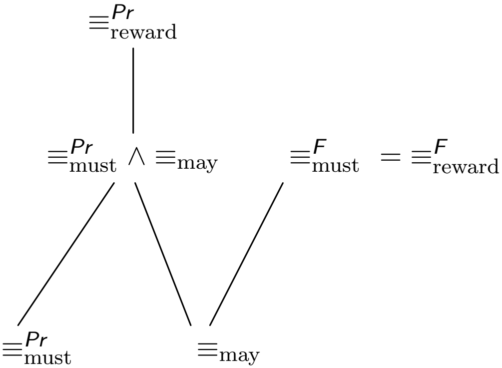
A lattice of testing equivalences for I/O automata



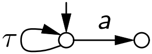
A lattice of testing equivalences for I/O automata



A lattice of testing equivalences for I/O automata

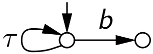


Counterexamples

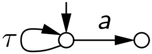


$$\equiv^{Pr}_{\mathrm{must}}$$

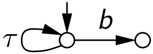
$$\not\equiv_{\mathrm{may}}$$



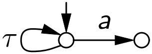
Counterexamples



$$\not\equiv^{Pr}_{\mathrm{reward}}$$
$$\equiv^{Pr}_{\mathrm{must}}$$
$$\not\equiv_{\mathrm{may}}$$
$$\not\equiv^{F}_{\mathrm{must}}$$



Counterexamples

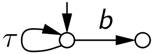


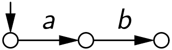
$\not\equiv^{Pr}_{\mathrm{reward}}$

$\equiv^{Pr}_{\mathrm{must}}$

$\not\equiv_{\mathrm{may}}$

$\not\equiv^{F}_{\mathrm{must}}$

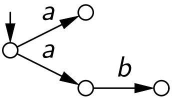




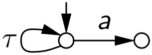
$\not\equiv^{Pr}_{\mathrm{must}}$

$\equiv_{\mathrm{may}}$

$\not\equiv^{F}_{\mathrm{must}}$



Counterexamples

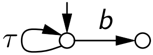


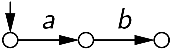
$\not\equiv^{Pr}_{\text{reward}}$

$\equiv^{Pr}_{\text{must}}$

$\not\equiv_{\text{may}}$

$\not\equiv^{F}_{\text{must}}$



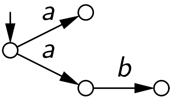


$\not\equiv^{Pr}_{\text{reward}}$

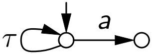
$\not\equiv^{Pr}_{\text{must}}$

$\equiv_{\text{may}}$

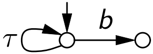
$\not\equiv^{F}_{\text{must}}$

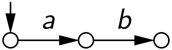


Counterexamples

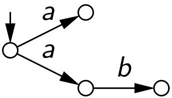


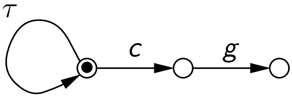
$$\not\equiv^{Pr}_{\mathrm{reward}}$$
$$\equiv^{Pr}_{\mathrm{must}}$$
$$\not\equiv_{\mathrm{may}}$$
$$\not\equiv^{F}_{\mathrm{must}}$$



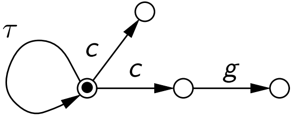


$$\not\equiv^{Pr}_{\mathrm{reward}}$$
$$\not\equiv^{Pr}_{\mathrm{must}}$$
$$\equiv_{\mathrm{may}}$$
$$\not\equiv^{F}_{\mathrm{must}}$$

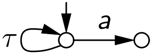




$$\not\equiv^{Pr}_{\mathrm{reward}}$$
$$\equiv_{\mathrm{may}}$$
$$\equiv^{Pr}_{\mathrm{must}}$$



Counterexamples

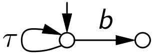


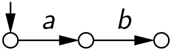
$\not\equiv^{Pr}_{\mathrm{reward}}$

$\equiv^{Pr}_{\mathrm{must}}$

$\not\equiv_{\mathrm{may}}$

$\not\equiv^{F}_{\mathrm{must}}$



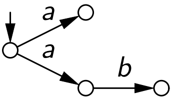


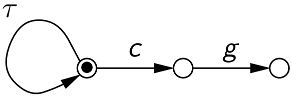
$\not\equiv^{Pr}_{\mathrm{reward}}$

$\not\equiv^{Pr}_{\mathrm{must}}$

$\equiv_{\mathrm{may}}$

$\not\equiv^{F}_{\mathrm{must}}$



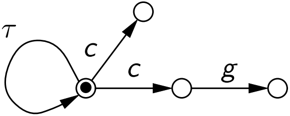


$\not\equiv^{Pr}_{\mathrm{reward}}$

$\equiv_{\mathrm{may}}$

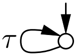
$\equiv^{Pr}_{\mathrm{must}}$

$\not\equiv^{F}_{\mathrm{must}}$



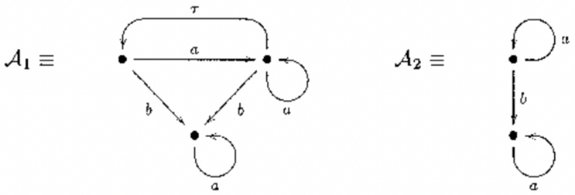
Counterexamples

$$\not\equiv^{Pr}_{\mathrm{reward}}$$
$$\not\equiv^{Pr}_{\mathrm{must}}$$
$$\equiv_{\mathrm{may}}$$
$$\equiv^{F}_{\mathrm{must}}$$



Counterexamples

EXAMPLE 5.2. Consider the I/O automata



where a is an input action, b is an output action, τ is an internal action, and the partitions of the locally controlled actions contain a single class. The I/O automata $\mathscr{A}_1$ and $\mathscr{A}_2$ are equivalent according to the quiescent preorder since they have the same external traces and their quiescent traces are all finite external traces containing at least a b action. The external trace a^∞, however, is a fair trace of $\mathscr{A}_1$ but not a fair trace of $\mathscr{A}_2$.

$\mathcal{A}_1 \equiv_{\text{reward}} \mathcal{A}_2$ but $\mathcal{A}_1 \not\equiv^F_{\text{must}} \mathcal{A}_2$.

$\mathcal{A}_1 \equiv_{\text{must}} \mathcal{A}_2$

$\mathcal{A}_1 \equiv_{\text{may}} \mathcal{A}_2$

Conclusion

When using the native notion of fairness from I/O automata as completeness criterion in the definition of must testing,

must testing exactly characterises the fair preorder from [LT89].

Upgrading to reward testing here does not yield extra distinctions.

Future work: extend with time and probabilities.