# Fair Must Testing for I/O Automata

Rob van Glabbeek

School of Comp. Sc. and Engineering, Univ. of New South Wales, Sydney, Australia
rvg@cs.stanford.edu

**Abstract.** The concept of must testing is naturally parametrised with a chosen completeness criterion or fairness assumption. When taking weak fairness as used in I/O automata, I show that it characterises exactly the fair preorder on I/O automata as defined by Lynch & Tuttle.

**Keywords:** I/O automata · Must testing · Fairness.

*This paper is dedicated to Frits Vaandrager at the occasion of his 60th birthday. I fondly remember my days at CWI as a starting computer scientist, sharing an office with Frits. Here I had the rare privilege of sharing all my ideas with Frits at the time they were formed, and receiving instantaneous meaningful feedback. This feedback has had a great impact on my work.*

*I take the opportunity to also pass best wishes and warmest thoughts to Frits from Ursula Goltz, whom I am visiting while finishing this paper. My joint work with Ursula was inspired by my work with Frits on connecting Petri nets and process algebra.*

## 1 Introduction

May- and must-testing was proposed by De Nicola & Hennessy in [2]. It yields semantic equivalences where two processes or automata are distinguished if and only if they react differently on certain tests. The tests are processes that additionally feature success states. Such a test $T$ is applied to a process $A$ by taking the CCS parallel composition $T|A$, and implicitly applying a CCS restriction operator to it that removes the remnants of uncompleted communication attempts. The outcome of applying $T$ to $A$ is deemed successful if and only if this composition yields a process that may, respectively must, reach a success state. It is trivial to recast this definition of may- and must-testing equivalence using the CSP parallel composition $\|$ [8] instead of the one from CCS.

I/O automata [9] are a model of concurrency that distinguishes output actions, which are under the control of a given automaton, from input actions, which are stimuli from the environment on which an automaton might react. The parallel composition $\|$ of I/O automata, exactly like the one of CSP, imposes synchronisation on actions the composed automata have in common. However, it allows forming the composition $A\|B$ only when $A$ and $B$ have no output actions in common. This makes it impossible to synchronise on actions $c$ where both $A$ and $B$ have the option not to allow $c$ in certain states.

Must testing equivalence for CCS and CSP partially discerns *branching time*, in the sense that is distinguishes the processes $\tau.(a+b)$ and $\tau.a+\tau.b$ displayed in Figure 1. This is not the case for I/O automata, as the synchronisations between test and tested automaton that are necessary to make such distinctions are ruled out by the restriction described above.

It is not a priori clear how a given process or automaton *must* reach a success state. For all we know it might stay in its initial state and never take any transition leading to this success state. To this end one must employ an assumption saying that under appropriate circumstances certain enabled transitions will indeed be taken. Such an assumption is called a *completeness criterion* [5]. The theory of testing from [2] implicitly employs a default completeness criterion that in [7] is called *progress*. However, one can parameterise the notion of must testing by the choice of any completeness criterion, such as the many notions of *fairness* classified in [7].

Lynch & Tuttle [9] defined a trace and a fair preorder on I/O automata, which were meant to reason about safety and liveness properties, respectively, just like the may- and must testing preorders of [2]. Unsurprisingly, as formally shown in Section 5 of this paper, the trace preorder on I/O automata is characterised exactly by may testing. Segala [12] has studied must-testing on I/O automata, employing the default completeness criterion, and found that on a large class of I/O automata it characterises the *quiescent trace preorder* of Vaandrager [13]. It does not exactly characterise the fair preorder, however.

In my analysis this is due to the choice of progress as the completeness criterion employed for must testing, whereas the fair preorder of I/O automata is based on a form of weak fairness. In this work I study must testing on I/O automata based on the same form of weak fairness, and find that it characterises the fair preorder exactly.

Although I refer to must-testing with fairness as the chosen completeness criterion as *fair must testing*, it should not be confused with the notion of *fair testing* employed in [1,10]. The latter is also known as *should testing*. It incorporates a concept of fairness that is much stronger than the notion of fairness from I/O automata, called *full fairness* in [7].

In [6] another mode of testing was proposed, called *reward testing*. Reward-testing equivalence combines the distinguishing power of may as well as must testing, and additionally makes some useful distinctions between processes that are missed by both may and must testing [6]. As for must testing, its definition is naturally parametrised by a completeness criterion. When applied to I/O automata, using as completeness criterion the form of fairness that is native to I/O automata, it turns out that reward testing is not stronger than must testing, and also characterises the fair preorder.

## 2 I/O automata

An I/O automaton is a labelled transition system equipped with a nonempty set of start states, with each action that may appear as transition label classified as an input, an output or an internal action. Input actions are under the control of the environment of the automaton, whereas output and internal actions, together called *locally-controlled* actions, are under the control of the automaton itself. I/O automata are *input enabled*, meaning that in each state each input action of the automaton can be performed. This indicates that the environment may perform such actions regardless of the state of the automaton; an input transition merely indicates how the automaton reacts on such an event. To model that certain input actions have no effect in certain states, one uses self-loops.

I/O automata employ a partition of the locally-controlled actions into *tasks* to indicate which sequences of transitions denote *fair* runs. A run is fair unless it has a suffix on which some task is enabled in every state, yet never taken.

**Definition 1** An *input/output automaton* (or *I/O automaton*) $A$ is a tuple $(acts(A), states(A), start(A), steps(A), part(A))$ with

- $acts(A)$ a set of *actions*, partitioned into three sets $in(A)$, $out(A)$ and $int(A)$ of *input actions*, *output actions* and *internal actions*, respectively,
- $states(A)$ a set of *states*,
- $start(A) \subseteq states(A)$ a nonempty set of start states,
- $steps(A) \subseteq states(A) \times acts(A) \times states(A)$ a *transition relation* with the property that $\forall s \in states(A).\ \forall a \in in(A).\ \exists (s, a, s') \in steps(A)$, and
- $part(A) \subseteq \mathscr{P}(local(A))$ a partition of the set $local(A) := out(A) \cup int(A)$ of *locally-controlled actions* of $A$ into *tasks*.

Let $ext(A) := in(A) \cup out(A)$ be the set of *external actions* of $A$.

An action $a \in acts(A)$ is *enabled* in a state $s \in states(A)$ if $\exists (s, a, s') \in steps(A)$. A task $\mathcal{T} \in part(A)$ is *enabled* in $s$ if some action $a \in \mathcal{T}$ is enabled is $s$.

**Definition 2** 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 $s_0 \in start(A)$ and $(s_i, a_{i+1}, s_{i+1}) \in steps(A)$ for all $i < length(\alpha)$. Here $length(\alpha) \in \mathbb{N} \cup \{\infty\}$ denotes the number of action occurrences in $\alpha$. The sequence $a_1, a_2, \ldots$ obtained by dropping all states from $\alpha$ is called $sched(\alpha)$. An execution $\alpha$ of $A$ is *fair* if, for each suffix $\alpha' = s_k, a_{k+1}, s_{k+1}, a_{k+2}, \ldots$ of $\alpha$ (with $k \in \mathbb{N} \wedge k \leqslant length(\alpha)$) and each task $\mathcal{T} \in part(A)$, if $\mathcal{T}$ is enabled in each state of $\alpha'$, then $\alpha'$ contains an action from $\mathcal{T}$.

In [9] two semantic preorders are defined on I/O automata, here called $\sqsubseteq_T$ and $\sqsubseteq_F$, the *trace* and the *fair preorder*. In [9] $S \sqsubseteq_T I$ and $S \sqsubseteq_F I$ are denoted "*I implements S*" and "*I solves S*", respectively. Here $S$ is an I/O automaton that is (a step closer to) the specification of a problem, and $I$ one that is (a step closer to) its implementation. The preorder $\sqsubseteq_T$ is meant to reason about safety properties: if $S \sqsubseteq_T I$ then $I$ has any safety property that $S$ has. In the same way, $\sqsubseteq_F$ is for reasoning about liveness properties. In [12] and much subsequent work $S \sqsubseteq_F I$ is written as $I \sqsubseteq_F S$. Here I put $I$ on the right, so as to orient the refinement symbol $\sqsubseteq$ in the way used in CSP [8], and in the theory of testing [2].

I/O automata are a typed model of concurrency, in the sense that two automata will be compared only when they have the same input and output actions.

**Definition 3** Let $\mathit{trace}(\alpha)$ be the finite or infinite sequence of external actions resulting from dropping all internal actions in $\mathit{sched}(\alpha)$, and let $\mathit{fintraces}(A)$ be the set $\{\mathit{trace}(\alpha) \mid \alpha \text{ is a finite execution of } A\}$. Likewise $\mathit{fairtraces}(A) := \{\mathit{trace}(\alpha) \mid \alpha \text{ is a fair execution of } A\}$. Now

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

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

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

By [7, Thm. 6.1] each finite execution can be extended into a fair execution. As a consequence, $A \sqsubseteq_F B \Rightarrow A \sqsubseteq_T B$.

The parallel composition of I/O automata [9] is similar to the one of CSP [8]: participating automata $A_i$ and $A_j$ synchronise on actions in $\mathit{acts}(A_i) \cap \mathit{acts}(A_j)$, while for the rest allowing arbitrary interleaving. However, it is defined only when the participating automata have no output actions in common.

**Definition 4** A collection $\{A_i\}_{i \in I}$ of I/O automata is *strongly compatible* if
- $\mathit{int}(A_i) \cap \mathit{acts}(A_j) = \emptyset$ for all $i, j \in I$ with $i \neq j$, and
- $\mathit{out}(A_i) \cap \mathit{out}(A_j) = \emptyset$ for all $i, j \in I$ with $i \neq j$,
- no action is contained in infinitely many sets $\mathit{acts}(A_i)$.

The *composition* $A = \prod_{i \in I} A_i$ of a countable collection $\{A_i\}_{i \in I}$ of strongly compatible I/O automata is defined by
- $\mathit{int}(A) := \bigcup_{i \in I} \mathit{int}(A_i)$,
- $\mathit{out}(A) := \bigcup_{i \in I} \mathit{out}(A_i)$,
- $\mathit{in}(A) := \bigcup_{i \in I} \mathit{in}(A_i) - \mathit{out}(A)$,
- $\mathit{states}(A) := \prod_{i \in I} \mathit{states}(A_i)$,
- $\mathit{start}(A) := \prod_{i \in I} \mathit{start}(A_i)$,
- $\mathit{steps}(A)$ is the set of triples $(\vec{s_1}, a, \vec{s_2})$ such that, for all $i \in I$, if $a \in \mathit{acts}(A_i)$ then $(\vec{s_1}[i], a, \vec{s_2}[i]) \in \mathit{steps}(A_i)$, and if $a \notin \mathit{acts}(A_i)$ then $\vec{s_1}[i] = \vec{s_2}[i]$, and
- $\mathit{part}(A) := \bigcup_{i \in I} \mathit{part}(A_i)$.

Clearly, composition of I/O automata is associative: when writing $A_1 \| A_2$ for $\prod_{i \in \{1,2\}} A_i$ then $(A \| B) \| C \cong A \| (B \| C)$, for some notion of isomorphism $\cong$, included in $\equiv_T$ and $\equiv_F$. Moreover, as shown in [9], composition is monotone for $\sqsubseteq_T$ and $\sqsubseteq_F$, or in other words, $\sqsubseteq_T$ and $\sqsubseteq_F$ are precongruences for composition:

$$\text{if } A_i \sqsubseteq_T B_i \text{ for all } i \in I, \text{ then } \prod_{i \in I} A_i \sqsubseteq_T \prod_{i \in I} B_i\,, \text{ and}$$

$$\text{if } A_i \sqsubseteq_F B_i \text{ for all } i \in I, \text{ then } \prod_{i \in I} A_i \sqsubseteq_F \prod_{i \in I} B_i\,.$$

The first condition of strong compatibility is not a limitation of generality. Each I/O automaton is $\equiv_T$ and $\equiv_F$-equivalent to the result of bijectively renaming its internal actions. Hence, prior to composing a collection of automata, one could rename their internal actions to ensure that this condition is met. Up to $\equiv_T$ and $\equiv_F$ the composition would be independent on the choice of these renamings.

## 3 Testing preorders

Testing preorders [2] are defined between *automata* $A$, defined as in Def. 1, but without the partition $part(A)$ and without the distinction between input and output actions, and therefore also without the input enabling requirement from Item 4. The parallel composition of automata is as in Def. 4, but without the requirement that the participating automata have no output actions in common.

**Definition 5** An *automaton* $A$ is a tuple $(acts(A), states(A), start(A), steps(A))$ with

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

A collection $\{A_i\}_{i\in I}$ of I/O automata is *compatible* if

- $int(A_i) \cap acts(A_j) = \emptyset$ for all $i,j \in I$ with $i \neq j$, and
- no action is contained in infinitely many sets $acts(A_i)$.

The *composition* $A = \prod_{i\in I} A_i$ of a countable collection $\{A_i\}_{i\in I}$ of compatible I/O automata is defined by

- $int(A) := \bigcup_{i\in I} int(A_i)$,
- $ext(A) := \bigcup_{i\in I} ext(A_i)$,
- $states(A) := \prod_{i\in I} states(A_i)$,
- $start(A) := \prod_{i\in I} start(A_i)$, and
- $steps(A)$ is the set of triples $(\vec{s_1}, a, \vec{s_2})$ such that, for all $i \in I$, if $a \in acts(A_i)$ then $(\vec{s_1}[i], a, \vec{s_2}[i]) \in steps(A_i)$, and if $a \notin acts(A_i)$ then $\vec{s_1}[i] = \vec{s_2}[i]$.

A *test* is such an automaton, but featuring a special external action $w$, not used elsewhere. This action is used to mark *success states*: those in which $w$ is enabled. The parallel composition $T\|A$ of a test $T$ and an automaton $A$, if it exists, is itself a test, and $[T\|A]$ denotes the result of reclassifying all its non-$w$ actions as internal. An execution of $[T\|A]$ is *successful* iff it contains a success state.

**Definition 6** An automaton $A$ *may pass* a test $T$, notation $A$ **may** $T$, if $[T\|A]$ has a successful execution. It *must pass* $T$, notation $A$ **must** $T$, if each complete execution[1] of $[T\|A]$ is successful. It *should pass* $T$, notation $A$ **should** $T$, if each finite execution of $[T\|A]$ can be extended into a successful execution.

Write $A \sqsubseteq_{\text{may}} B$ if $ext(A) = ext(B)$ and $A$ **may** $T$ implies $B$ **may** $T$ for each test $T$ that is compatible with $A$ and $B$. The preorders $\sqsubseteq_{\text{must}}$ and $\sqsubseteq_{\text{should}}$ are defined similarly.

[1] The original work on must testing [2] defined an execution to be complete if it either is infinite, of ends in a state without outgoing transitions. Here I will consider the concept of a complete execution as a parameter in the definition of must testing.

The may- and must-testing preorders stem from [2], whereas should-testing was added independently in [1] and [10]. I have added the condition $ext(A) = ext(B)$ to obtain preorders that respect the types of automata. A fourth mode of testing, called *reward testing*, was contributed in [6]. It has no notion of success state, and no action $w$; instead, each transition of a test $T$ is tagged with a real number, the reward of taking that transition. A negative reward can be seen as a penalty. Each transition $(s, a, s')$ of $[T\|A]$ with $a \in acts(T)$ inherits its reward from the unique transition of $T$ it projects to; in case $a \notin acts(T)$ it has reward 0. The reward $reward(\alpha)$ of an execution $\alpha$ is the sum of the rewards of the actions in $\alpha$.[2] Now $A \sqsubseteq_{\text{reward}} B$ if $ext(A) = ext(B)$ and for each test $T$ that is compatible with $A$ and $B$ and for each complete execution $\beta$ of $[T\|B]$ there exists a complete execution $\alpha$ of $[T\|A]$ such that $reward(\alpha) \leqslant reward(\beta)$.

In the original work on testing [2,6] the CCS parallel composition $T|A$ was used instead of the CSP parallel composition $T\|A$; moreover, only those executions consisting solely of internal actions mattered for the definitions of passing a test. The present approach is equivalent, in the sense that it trivially gives rise to the same testing preorders.

The may-testing preorder can be regarded as pointing in the opposite direction as the others. Using CCS notation, one has $\tau.P \sqsubsetneq_{\text{may}} \tau.P + \tau.Q$, yet $\tau.P + \tau.Q \sqsubsetneq_{\text{must}} \tau.P$, $\tau.P + \tau.Q \sqsubsetneq_{\text{should}} \tau.P$ and $\tau.P + \tau.Q \sqsubsetneq_{\text{reward}} \tau.P$. The inverse of the may-testing preorder can be characterised as *survival testing*. Here a state in which $w$ is enabled is seen as a *failure state* rather than a success state, and automaton $A$ *survives* test $T$, notation $A$ **surv** $T$, if no execution of $[T\|A]$ passes through a failure state. Write $A \sqsubseteq_{\text{surv}} B$ if $ext(A) = ext(B)$ and $A$ **surv** $T$ implies $B$ **surv** $T$ for each test $T$ that is compatible with $A$ and $B$. By definition, $A \sqsubseteq_{\text{surv}} B$ iff $B \sqsubseteq_{\text{may}} A$.

The only implications between reward, must and may/survival testing are

$$A \sqsubseteq_{\text{reward}} B \;\Rightarrow\; A \sqsubseteq_{\text{must}} B \qquad \text{and} \qquad A \sqsubseteq_{\text{reward}} B \;\Rightarrow\; A \sqsubseteq_{\text{surv}} B\,.$$

Namely, any must test $T$ witnessing $A \not\sqsubseteq_{\text{must}} B$ can be coded as a reward test by assigning a reward $+1$ to all transitions of $T$ leading to a success state (and 0 to all other transitions). Likewise any survival test $T$ witnessing $A \not\sqsubseteq_{\text{surv}} B$ can be coded as a reward test by assigning a reward $-1$ to all transitions of $T$ leading to a failure state.

The notions of may- and should-testing are unambiguously defined above, whereas the notions of must- and reward testing depend on the definition of a complete execution. In [5] I posed that transition systems or automata constitute a good model of distributed systems only in combination with a *completeness criterion*: a selection of a subset of all executions as complete executions, modelling complete runs of the represented system.

The default completeness criterion, employed in [2,6] for the definition of must- and reward testing, deems an execution complete if it either is infinite, of ends in deadlock, a state without outgoing transitions. Other completeness

[2] If $\alpha$ is infinite, its reward can be $+\infty$ or $-\infty$; see [6] for a precise definition.

criteria either classify certain finite executions that do not end in deadlock as complete, or certain infinite executions as incomplete.

The first possibility was explored in [5,7] by considering a set $B$ of actions that might be blocked by the environment in which an automaton is running. Now a finite execution can be deemed complete if all transitions enabled in its last state have labels from $B$. The system might stop at such a state if indeed the environment blocks all those actions. Since in the application to must- and reward testing, all non-$w$ transitions in $[T\|A]$ are labelled with internal actions, which cannot be blocked by the environment, the above possibility of increasing the set of finite complete executions does not apply.

The second possibility was extensively explored in [7], where a multitude of completeness criteria was defined. Most of those can be used as a parameter in the definition of must- and reward testing. So far, the resulting testing preorders have not been explored.[3]

## 4 Testing preorders for I/O automata

Since I/O automata can be seen as special cases of the automata from Section 3, the definitions of Section 3 also apply to I/O automata. The condition $ext(A) = ext(B)$ should then be read as $in(A) = in(B) \wedge out(A) = out(B)$. The only place where it makes an essential difference whether one works with I/O automata or general automata is in judging compatibility between automata and tests. Given two I/O automata $A$ and $B$, let $A \sqsubseteq^{\text{LTS}}_{\text{must}} B$ be defined by first seeing $A$ and $B$ as general automata (by dropping the partitions $part(A)$ and $part(B)$), and then applying the definitions of Section 3, using the default completeness criterion. In contrast, let $A \sqsubseteq^{Pr}_{\text{must}} B$ be defined as Section 3, but only allowing tests that are themselves I/O automata (seeing the special action $w$ as an output action), and that are strongly compatible with $A$ and $B$. The superscript $Pr$ stands for "progress", the name given in [7] to the default completeness criterion. The difference between $\sqsubseteq^{\text{LTS}}_{\text{must}}$ and $\sqsubseteq^{Pr}_{\text{must}}$ is illustrated in Figure 1.

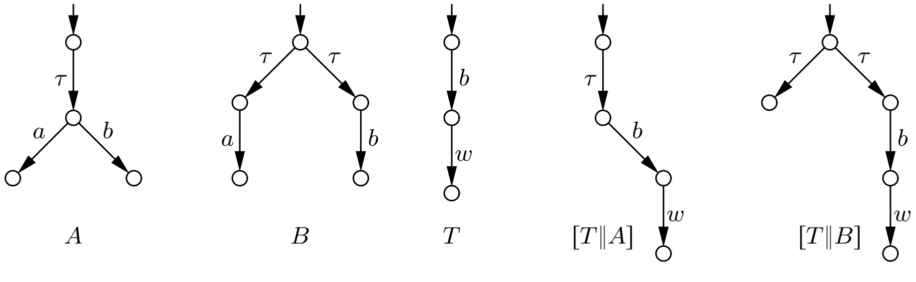



**Fig. 1.** Classic example of how branching time is discerned by must testing

[3] The paper [4] explores these testing preorders; it was written after the present paper.

Here $A$ and $B$ are automata with $acts(A) = acts(B) = \{\tau, a, b\}$, and $T$ is a test with $acts(T) = \{a, b, w\}$. The short arrows point to start states. Test $T$ witnesses that $A \not\sqsubseteq^{\text{LTS}}_{\text{must}} B$, for $A$ **must** $T$, yet $\neg(B$ **must** $T)$. Here it is crucial that $a \in acts(T)$, even though this action labels no transition of $T$, for otherwise the $a$-transition of $A$ would return in $[T \| A]$ and one would not obtain $A$ **must** $T$. To see $A$, $B$ and $T$ as I/O automata, one needs to take $in(A) = in(B) = in(T) = \emptyset$, and thus $a, b \in out(A) \cap out(B) \cap out(T)$. However, this violates the strong compatibility of $T$ with $A$ and $B$, so that $T$ is disqualified as an appropriate test. There is no variant of $T$ that is strongly compatible with $A$ and $B$ and yields the same result; in fact $A \equiv^{Pr}_{\text{must}} B$.

## 5 May testing

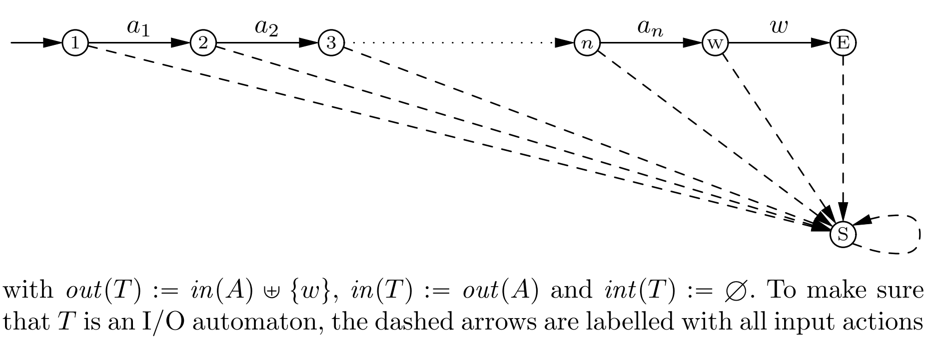
For may-testing on I/O automata there is no difference between $\sqsubseteq^{\text{LTS}}_{\text{may}}$—allowing any test that is compatible with $A$ and $B$—and $\sqsubseteq_{\text{may}}$—allowing only tests that are strongly compatible with $A$ and $B$. These preorders both coincide with the trace preorder $\sqsupseteq_T$.

**Theorem 1** $A \sqsubseteq^{\text{LTS}}_{\text{may}} B$ iff $A \sqsubseteq_{\text{may}} B$ iff $B \sqsubseteq_T A$.

*Proof.* Suppose $B \sqsubseteq_T A$, i.e., $in(A) = in(B) \wedge out(A) = out(B)$ and $fintraces(A) \subseteq fintraces(B)$, and let $T$ be any test compatible with $A$ and $B$. The automaton $T$ need not be an I/O automaton, and even if it is, it need not be strongly compatible with $A$ and $B$. It is well-known that $\sqsubseteq_T$ is a precongruence for composition [8], so $fintraces(T \| A) \subseteq fintraces(T \| B)$. Since $C$ **may** $T$ (for any $C$) iff $w$ occurs in a trace $\sigma \in fintraces(T \| C)$, it follows that $A$ **may** $T$ implies $B$ **may** $T$. Thus $A \sqsubseteq^{\text{LTS}}_{\text{may}} B$.

That $A \sqsubseteq^{\text{LTS}}_{\text{may}} B$ implies $A \sqsubseteq_{\text{may}} B$ is trivial.

Now suppose $A \sqsubseteq_{\text{may}} B$. Then $in(A) = in(B) \wedge out(A) = out(B)$. Let $\sigma = a_1 a_2 \ldots a_n \in fintraces(A)$. Let $T$ be the test automaton




with $out(T) := in(A) \uplus \{w\}$, $in(T) := out(A)$ and $int(T) := \emptyset$. To make sure that $T$ is an I/O automaton, the dashed arrows are labelled with all input actions of $T$, except for $a_i$ (if $a_i \in in(T)$) for the dashed arrow departing from state $i$. By construction, $T$ is strongly compatible with $A$ and $B$. Now $C$ **may** $T$ (for any $C$) iff $\sigma \in fintraces(C)$. Hence $A$ **may** $T$, and thus $B$ **may** $T$, and therefore $\sigma \in fintraces(B)$. □

## 6 Must testing based on progress

**Definition 7** An I/O automaton $T$ is *complementary* to I/O automaton $A$ if $out(T) = in(A) \uplus \{w\}$, $in(T) = out(A)$ and $int(T) \cap int(A) = \emptyset$.

In this case $T$ and $A$ are also strongly compatible, so that $T\|A$ is defined, and $in(T\|A) = \emptyset$. I now show that for the definition of $\sqsubseteq^{Pr}_{\rm must}$ it makes no difference whether one restricts the tests $T$ that may be used to compare two I/O automata $A$ and $B$ to ones that are complementary to $A$ and $B$.

For use in the following proof, define the relation $\equiv$ between I/O automata by $C \equiv D$ iff $states(C) = states(D) \wedge start(C) = start(D) \wedge steps(C) = steps(D)$. Note that $T\|A \equiv T'\|A$ implies that $A$ **must** $T$ iff $A$ **must** $T'$.

**Proposition 1** $A \sqsubseteq^{Pr}_{\rm must} B$ iff $in(A) = in(B) \wedge out(A) = out(B)$ and $A$ **must** $T$ implies $B$ **must** $T$ for each test $T$ that is complementary to $A$ and $B$.

*Proof.* Suppose $A \sqsubseteq^{Pr}_{\rm must} B$. Then $in(A){=}in(B) \wedge out(A){=}out(B)$ and $A$ **must** $T$ implies $B$ **must** $T$ for each test $T$ that is strongly compatible with $A$ and $B$, and thus certainly for each test $T$ that is complementary to $A$ and $B$.

Now suppose $in(A) = in(B) \wedge out(A) = out(B)$ but $A \not\sqsubseteq^{Pr}_{\rm must} B$. Then there is a test $T$, strongly compatible with $A$ and $B$, such that $A$ **must** $T$, yet $\neg(B$ **must** $T)$. It suffices to find a test $T''$ with the same properties that is moreover complementary to $A$ and $B$.

First modify $T$ into $T'$ by adding $ext(A)\backslash ext(T)$ to $in(T')$, while adding a loop $(s, a, s)$ to $steps(T')$ for each state $s \in states(T')$ and each $a \in ext(A)\backslash ext(T)$. Now $T\|A = T'\|A$ and $T\|B = T'\|B$, and thus $A$ **must** $T'$, yet $\neg(B$ **must** $T')$. Moreover, $ext(A) = ext(B) \subseteq ext(T')$.

Modify $T'$ further into $T''$ by reclassifying any action $a \in in(T') \cap in(A)$ as an output action of $T''$ and any $a \in ext(T')\backslash(ext(A) \uplus \{w\})$ as an internal action of $T''$. How $part(T'')$ is defined is immaterial. Then $T'\|A \equiv T''\|A$ and $T'\|B \equiv T''\|B$, and thus $A$ **must** $T''$, yet $\neg(B$ **must** $T'')$. Now $out(T'') = in(A) \uplus \{w\}$, $in(T'') = out(A)$, $int(T'') \cap int(A) = \emptyset$ and $int(T'') \cap int(B) = \emptyset$. □

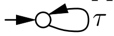
Using the characterisation of Prop. 1 as definition, the preorder $\sqsubseteq^{Pr}_{\rm must}$ on I/O automata has been studied by Segala [12, Section 7]. There it was related to the *quiescent trace* preorder $\sqsubseteq_Q$ defined by Vaandrager [13]. Similar as for the preorders of Section 2, I write $S \sqsubseteq_Q I$ for what was denoted $I \sqsubseteq_Q S$ in [12], and $I \sqsubseteq_{qT} S$ in [13].

**Definition 8** An execution $\alpha$ is *quiescent* if it is finite and its last state enables only input actions. Let $qtraces(A){:=}\{trace(\alpha) \mid \alpha$ is a quiescent execution of $A\}$. Now

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

An I/O automaton is *finitely branching* iff each of its states enables finitely many transitions; it is *strongly convergent* if it has no infinite execution $\alpha$ with $trace(\alpha)$ finite, i.e., no execution with an infinite suffix of only internal actions.

**Theorem 2 ([12, Thm. 7.3])** Let $A$ and $B$ be finitely branching and strongly convergent I/O automata. Then $A \sqsubseteq^{Pr}_{\text{must}} B$ iff $A \sqsubseteq_Q B$.

Note that an execution is quiescent iff it is fair and finite. By [12, Thm. 5.7], if $A$ is strongly convergent then $A \sqsubseteq_F B$ implies $A \sqsubseteq_Q B$. (For let $A \sqsubseteq_F B$. If $\sigma \in$ *qtraces*$(B)$, then $\sigma \in$ *fairtraces*$(B) \subseteq$ *fairtraces*$(A)$ so $A$ has a fair execution $\alpha$ with *trace*$(\alpha) = \sigma$. As $A$ is strongly convergent, $\alpha$ is finite. Hence $\sigma \in$ *qtraces*$(A)$.) This does not hold when dropping the side condition of strong convergence. Take $A =$  and $B =$  with $acts(A) = \emptyset$ and $acts(B) = int(B) = \{\tau\}$. Then $A \equiv_F B$, yet $A \not\sqsubseteq_Q B$ (and $A \not\sqsubseteq^{Pr}_{\text{must}} B$).

Even restricted to finitely branching and strongly convergent I/O automata, $A \sqsubseteq_Q B$ does not imply $A \sqsubseteq_F B$. This is illustrated by [12, Examples 5.1 and 5.2].

## 7 Must testing based on fairness

As explained in Section 3, the notion of must testing is naturally parametrised by the choice of a completeness criterion. As I/O automata are already equipped with a completeness criteria, namely the notion of fairness from Def. 2, the most appropriate form of must testing for I/O automata takes this concept of fairness as its parameter, rather than the default completeness criterion used in Section 6.

A problem in properly defining a must-testing preorder $\sqsubseteq^{F}_{\text{must}}$ involves the definition of the operator [ ] employed in Def. 6. In the context of standard automata, this operator reclassifies all its external actions, except for the success action $w$, as internal. When applied to I/O automaton $A$, it is not a priori clear how to define *part*$([A])$, for this is a partition of the set of locally-controlled actions into tasks, and when changing an input action into a locally-controlled action, one lacks guidance on which task to allocate it to. This was a not a problem in Section 6, as there the must-testing preorder $\sqsubseteq^{Pr}_{\text{must}}$ depends in no way on *part*.

Below I inventorise various solutions to this problem, which gives rise to three possible definitions of $\sqsubseteq^{F}_{\text{must}}$. Then I show in Section 9 that all three resulting preorders coincide, so that it doesn't matter on which of the definitions one settles. Moreover, these preorders all turn out to coincide with the fair preorder $\sqsubseteq_F$ that comes with I/O automata.

My first (and default) solution is to simply drop the operator [ ] from Def. 6:

**Definition 9** An I/O automaton $A$ *must pass* a test $T$ *fairly*—$A$ $\textbf{must}^F$ $T$—if each fair execution of $T \| A$ is successful. Write $A \sqsubseteq^{F}_{\text{must}} B$ if $in(A) = in(B) \wedge out(A) = out(B)$ and $A$ $\textbf{must}^F$ $T$ implies $B$ $\textbf{must}^F$ $T$ for each test $T$ that is strongly compatible with $A$ and $B$.

This is a plausible approach, as none of the testing preorders discussed in Sections 3–6 would change at all were the operator [ ] dropped from Def. 6. This is the case because the set of executions, successful executions and complete executions of an automaton $A$ is independent of the status (input, output or internal) of the actions of $A$.

The above begs the question why I bothered to employ the operator $[\ \ ]$ in Def. 6 in the first place. The main reason is that the theory of testing [2] was developed in the context of CCS, where each synchronisation of an action from a test with one from a tested process yields an internal action $\tau$. Def. 6 recreates this theory using the operator $\|$ from CSP [8] and I/O automata [9], but as here synchronised actions are not internal, they have to be made internal to obtain the same effect. A second reason concerns the argument used towards the end of Section 3 for not parametrising notions of testing with a set $B$ of actions that can be blocked; this argument hinges on all relevant actions being internal.

My second solution is to restrict the set of allowed tests $T$ for comparing I/O automata $A$ and $B$ to those for which $in(T\|A) = in(T\|B) = \emptyset$. This is the case iff $in(T) \subseteq out(A)$ and $in(A) \subseteq out(T)$. In that case $[T\|A]$ and $[T\|B]$ are trivial to define, as the set of locally-controlled actions stays the same. Moreover, it makes no difference whether this operator is included in the definition of **must** or not, as the set of fair executions of a process is not affected by a reclassification of output actions as internal actions.

**Definition 10** Write $A \ {}^{\emptyset}\!\sqsubseteq^{F}_{\text{must}} B$ if $in(A) = in(B) \wedge out(A) = out(B)$ and moreover $A$ $\mathbf{must}^F$ $T$ implies $B$ $\mathbf{must}^F$ $T$ for each test $T$ that is strongly compatible with $A$ and $B$, and for which $in(T\|A) = in(T\|B) = \emptyset$.

A small variation of this idea restricts the set of allowed tests even further, namely to the ones that are complementary to $A$ and $B$, as defined in Def. 7. This yield a fair version of the must-testing preorder employed in [12].

**Definition 11** Write $A \ {}^{\text{cm}}\!\sqsubseteq^{F}_{\text{must}} B$ if $in(A) = in(B) \wedge out(A) = out(B)$ and $A$ $\mathbf{must}^F$ $T$ implies $B$ $\mathbf{must}^F$ $T$ for each $T$ that is complementary to $A$ and $B$.

As a last solution I consider tests $T$ that are not restricted as in Defs. 10 or 11, while looking for elegant ways to define $[T\|A]$ and $[T\|B]$. First of all, note that no generality is lost when restricting to tests $T$ such that $ext(A)(= ext(B)) \subseteq ext(T)$, regardless how the operator $[\ \ ]$ is defined. Namely, employing the first conversion from the proof of Prop. 1, any test $T$ that is strongly compatible with I/O automata $A$ and $B$ can converted into a test $T'$ satisfying this requirement, and such that $T\|A = T'\|A$ and $T\|B = T'\|B$.

An application of $[\ \ ]$ to $T\|A$ consists of reclassifying external actions of $T\|A$ as internal actions. However, since for the definition of the testing preorders it makes no difference whether an action in $T\|A$ is an internal or an output action, one can just as well use an operator $[\ \ ]'$ that merely reclassifies input actions of $T\|A$ as output actions. Note that $in(T\|A) \subseteq in(T)$, using that $ext(A) \subseteq ext(T)$. Let $T^*$ be a result of adapting the test $T$ by reclassifying the actions in $in(T\|A)$ from input actions of $T$ into output actions of $T$; the test $T^*$ is not uniquely defined, as there are various ways to fill in $part(T^*)$.

**Observation 1** Apart from the problematic definition of $part([T\|A]')$, the I/O automaton $[T\|A]'$ is the very same as $T^*\|A$.

In other words, the reclassification of input into output actions can just as well be done on the test, instead of on the composition of test and tested automaton. The advantage of this approach is that the problematic definition of $part([T\|A]')$ is moved to the test as well. Now one can use $T^*\|A$ instead of $[T\|A]'$ in the definition of must testing for any desired definition of $part(T^*)$. This amounts to choosing any test $T^*$ with $in(T^*\|A) = \varnothing$. It makes this solution equivalent to the one of Def. 10.

## 8 Action-based must testing

The theory of testing from [2] employs the success action $w$ merely to mark success states; an execution is successful iff it contains a state in which $w$ is enabled. In [3] this is dubbed *state-based testing*. Segala [11] (in a setting with probabilistic automata) uses another mode of testing, called *action-based* in [3], in which an execution is defined to be successful iff it contains the action $w$.

Although the state-based and action-based may-testing preorders obviously coincide, the state-based and action-based must-testing preorders do not, at least when employing the default completeness criterion. An example showing the difference is given in [3]. It involves two automata $A$ and $B$, which can in fact be seen as I/O automata, such that $A \not\sqsubseteq^{Pr}_{\text{must}} B$, yet $A \overset{\text{ab}}{\equiv}{}^{Pr}_{\text{must}} B$. Here $\overset{\text{ab}}{\equiv}{}^{Pr}_{\text{must}}$ is the action-based version of $\equiv^{Pr}_{\text{must}}$.

So far I have considered only state-based testing preorders on I/O automata. Let $\overset{\text{ab}}{\sqsubseteq}{}^{F}_{\text{must}}$ be the action-based version of $\sqsubseteq^{F}_{\text{must}}$. It is defined as in Def. 9, but using $\mathbf{must}^F_{\text{ab}}$ instead of $\mathbf{must}^F$. Here $A$ $\mathbf{must}^F_{\text{ab}}$ $T$ holds iff each fair trace of $T\|A$ contains the action $w$. Below I will show that when taking the notion of fairness from [9] as completeness criterion, state-based and action-based must testing yields the same result, i.e., $\overset{\text{ab}}{\sqsubseteq}{}^{F}_{\text{must}}$ equals $\sqsubseteq^{F}_{\text{must}}$. In fact, I need this result in my proof that $\sqsubseteq^{F}_{\text{must}}$ coincides with $\sqsubseteq_F$.

## 9 Fair must testing agrees with the fair traces preorder

The following theorem states that the must-testing preorder on I/O automata based on the completeness criterion of fairness that is native to I/O automata, in each of the four forms discussed in Sections 7 and 8, coincides with the standard preorder of I/O automata based on reverse inclusion of fair traces.

**Theorem 3** $A \overset{\text{ab}}{\sqsubseteq}{}^{F}_{\text{must}} B$ iff $A \sqsubseteq^{F}_{\text{must}} B$ iff $A \overset{\varnothing}{\sqsubseteq}{}^{F}_{\text{must}} B$ iff $A \overset{\text{cm}}{\sqsubseteq}{}^{F}_{\text{must}} B$ iff $A \sqsubseteq_F B$.

*Proof.* Suppose $A \sqsubseteq_F B$, i.e., $in(A) = in(B) \wedge out(A) = out(B)$ and $fairtraces(B) \subseteq fairtraces(A)$, and let $T$ be any test that is strongly compatible with $A$ and $B$. Since $\sqsubseteq_F$ is a precongruence for composition (cf. Section 2), $fairtraces(T\|B) \subseteq fintraces(T\|A)$. Since for action-based must testing $C$ $\mathbf{must}^F_{\text{ab}}$ $T$ (for any $C$) iff $w$ occurs in each fair trace $\sigma \in fairtraces(T\|C)$, it follows that $A$ $\mathbf{must}^F_{\text{ab}}$ $T$ implies $B$ $\mathbf{must}^F_{\text{ab}}$ $T$. Thus $A \overset{\text{ab}}{\sqsubseteq}{}^{F}_{\text{must}} B$.

Now suppose $A \;{}^{\text{ab}}\!\sqsubseteq^F_{\text{must}} B$. In order to show that $A \sqsubseteq^F_{\text{must}} B$, suppose that $A$ $\mathbf{must}^F$ $T$, where $T$ is a test that is strongly compatible with $A$ and $B$. Let the test $T^*$ be obtained from $T$ by (i) dropping all transitions $(s, a, s') \in \mathit{steps}(T)$ for $s$ a success state and $a \neq w$, and (ii) adding a loop $(s, a, s)$ for each success state $s$ and $a \in \mathit{in}(T)$. Since for state-based must testing it is irrelevant what happens after encountering a success state, one has

$$C \;\mathbf{must}^F\; T \;\text{ iff }\; C \;\mathbf{must}^F\; T^* \tag{1}$$

for each I/O automaton $C$. Moreover, I claim that for each $C$ one has

$$C \;\mathbf{must}^F\; T^* \;\text{ iff }\; C \;\mathbf{must}^F_{\text{ab}}\; T^*. \tag{2}$$

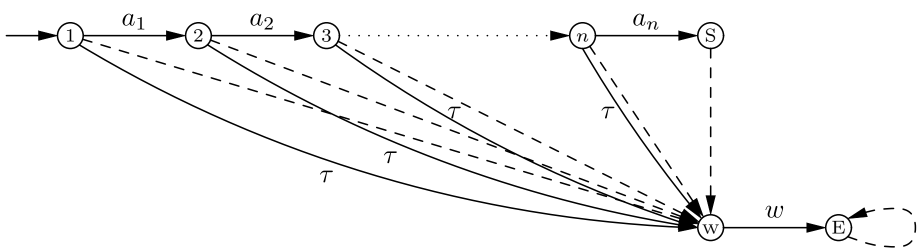
Here "if" is trivial. For "only if", let $\alpha$ be a fair execution of $T^* \| C$, and suppose, towards a contradiction, that $\alpha$ contains a success state $(s, r)$, with $s$ a success state of $T^*$ and $r$ a state of $C$, but does not contain the success action $w$. Let $\alpha'$ be the suffix of $\alpha$ starting with the first occurrence of $(s, r)$. Then all states of $\alpha'$ have the form $(s, r')$, and the action $w$ is enabled in each of these states. Let $\mathcal{T} \in \mathit{part}(T^* \| C)$ be the task containing $w$. Since $w$ is a locally controlled action of $T^*$, by Def. 4 all members of $\mathcal{T}$ must be locally controlled actions of $T^*$. No such action can occur in $\alpha'$. This contradicts the assumption that $\alpha$ is fair (cf. Def. 2), and thereby concludes the proof of (2).

From the assumption $A$ $\mathbf{must}^F$ $T$ one obtains $A$ $\mathbf{must}^F_{\text{ab}}$ $T^*$ by (1) and (2), and $B$ $\mathbf{must}^F_{\text{ab}}$ $T^*$ by the assumption that $A \;{}^{\text{ab}}\!\sqsubseteq^F_{\text{must}} B$. Hence $B$ $\mathbf{must}^F$ $T$ by (2) and (1). Thus $A \sqsubseteq^F_{\text{must}} B$.

That $A \sqsubseteq^F_{\text{must}} B$ implies $A \;{}^{\varnothing}\!\sqsubseteq^F_{\text{must}} B$ is trivial.

That $A \;{}^{\varnothing}\!\sqsubseteq^F_{\text{must}} B$ implies $A \;{}^{\text{cm}}\!\sqsubseteq^F_{\text{must}} B$ is also trivial.

Finally, suppose $A \;{}^{\text{cm}}\!\sqsubseteq^F_{\text{must}} B$. Then $\mathit{in}(A) = \mathit{in}(B) \wedge \mathit{out}(A) = \mathit{out}(B)$. Let $\sigma = a_1 a_2 \dots a_n \in \mathit{fairtraces}(B)$. Let $T$ be the test automaton




with $\mathit{out}(T) := \mathit{in}(A) \uplus \{w\}$, $\mathit{in}(T) := \mathit{out}(A)$ and $\mathit{int}(T) := \{\tau\}$. The dashed arrows are labelled with all input actions of $T$, except for $a_i$ (if $a_i \in \mathit{in}(T)$) for the dashed arrow departing from state $i$. By construction, $T$ is complementary to $A$ and $B$. Now $C$ **must** $T$ (for any $C$) iff $\sigma \notin \mathit{fairtraces}(C)$. Hence $B$ **may not** $T$, and thus $A$ **may not** $T$, and therefore $\sigma \in \mathit{fairtraces}(A)$.

The case that $\sigma = a_1 a_2 \dots \in \mathit{fairtraces}(B)$ is infinite goes likewise, but without the state S in $T$. Hence $A \sqsubseteq_F B$. □

## 10 Reward testing

The reward testing preorder taking the notion of fairness from Def. 2 as underlying completeness criterion can be defined on I/O automata by analogy of Definitions 9, 10 or 11. Here I take the one that follows Def. 9, as it is clearly the strongest, i.e., with its kernel making the most distinctions.

**Definition 12** Write $A \sqsubseteq^F_{\text{reward}} B$ if $in(A) = in(B) \wedge out(A) = out(B)$ and for each reward test $T$ that is strongly compatible with $A$ and $B$ and for each fair execution $\beta$ of $T\|B$ there is a fair execution $\alpha$ of $T\|A$ with $reward(\alpha) \leqslant reward(\beta)$.

When taking progress as underlying completeness criterion, reward testing is stronger than must testing; the opening page of [6] shows an example where reward testing makes useful distinctions that are missed by may as well as must testing. When moving to fairness as the underlying completeness criterion, must testing no longer misses that example, and in fact must testing becomes equally strong as reward testing. In order to show this, I will use the following notation.

**Definition 13** Let $A_1$ and $A_2$ be two strongly compatible I/O automata. A state $\vec{s}$ of $A_1\|A_2$ is a pair $(\vec{s}[1], \vec{s}[2])$ with $\vec{s}[k] \in states(A_k)$ for $k = 1, 2$. Let $\alpha = \vec{s}_0, a_1, \vec{s}_1, a_2, \ldots$ be an execution of $A_1\|A_2$. The projection $\alpha[k]$ of $\alpha$ to the $k^{\text{th}}$ component $A_k$, for $k = 1, 2$, is obtained from $\alpha$ by deleting "$, a_i, \vec{s}_i$" whenever $a_i \notin acts(A_k)$, and replacing the remaining pairs $\vec{s}_i$ by $\vec{s}_i[k]$.

Moreover, if $\sigma$ is a sequence of external actions of $A_1\|A_2$, then $\sigma\upharpoonright A_k$ is what is left of $\sigma$ after removing all actions outside $acts(A_k)$.

Note that if $\sigma = trace(\alpha)$, for $\alpha$ an execution of $A_1\|A_2$, then $\sigma\upharpoonright A_k = trace(\alpha[k])$. Moreover, if $\alpha$ is an execution of $T\|A$, were $T$ is a test and $A$ a tested automaton, then all rewards of the actions in $\alpha$ are inherited from the ones in $\alpha[1]$, so that

$$reward(\alpha) = reward(\alpha[1]). \tag{3}$$

**Theorem 4** $A \sqsubseteq^F_{\text{reward}} B$ iff $A \sqsubseteq^F_{\text{must}} B$ iff $A \sqsubseteq_F B$.

*Proof.* That $A \sqsubseteq^F_{\text{reward}} B$ implies $A \sqsubseteq^F_{\text{must}} B$ has been shown in [6, Thm. 7] and is also justified in Section 3.

That $A \sqsubseteq^F_{\text{must}} B$ implies $A \sqsubseteq_F B$ has been demonstrated by Thm. 3.

Suppose $A \sqsubseteq_F B$, i.e., $in(A) = in(B) \wedge out(A) = out(B)$ and $fairtraces(B) \subseteq fairtraces(A)$, and let $T$ be any test that is strongly compatible with $A$ and $B$. Let $\beta$ be a fair execution of $T\|B$. By [9, Prop. 4], $\beta[1]$ is a fair execution of $T$, and $\beta[2]$ is a fair execution of $B$. Since $A \sqsubseteq_F B$, automaton $A$ has a fair execution $\gamma$ with $trace(\gamma) = trace(\beta[2])$. Let $\sigma := trace(\beta)$. Then $\sigma$ is a sequence of external actions of $T\|A$ such that $\sigma\upharpoonright T = trace(\beta[1])$ and $\sigma\upharpoonright A = \sigma\upharpoonright B = trace(\beta[2]) = trace(\gamma)$. By [9, Prop. 5], there exists a fair execution $\alpha$ of $T\|A$ such that $trace(\alpha) = \sigma$, $\alpha[1] = \beta[1]$ and $\alpha[2] = \gamma$. By (3) one has $reward(\alpha) = reward(\alpha[1]) = reward(\beta[1]) = reward(\beta)$. Thus $A \sqsubseteq^F_{\text{reward}} B$. □

## 11 Conclusion

When adapting the concept of a complete execution, which plays a central rôle in the definition of must testing, to the weakly fair executions of I/O automata, must testing turns out to characterise exactly the fair preorder on I/O automata. Moreover, reward testing, which under the default notion of a complete execution is much more discriminating than must testing, in this setting has the same distinguishing power. Interesting venues for future investigation include extending these connections to timed and probabilistic settings.

## References


1. E. Brinksma, A. Rensink & W. Vogler (1995): *Fair Testing*. In I. Lee & S.A. Smolka, editors: Proceedings 6th International Conference on *Concurrency Theory*, (CONCUR'95), Philadelphia, PA, USA, August 1995, LNCS 962, Springer, pp. 313–327, doi:10.1007/3-540-60218-6_23.
2. R. De Nicola & M. Hennessy (1984): *Testing equivalences for processes*. *Theoretical Computer Science* 34, pp. 83–133, doi:10.1016/0304-3975(84)90113-0.
3. Y. Deng, R.J. van Glabbeek, M. Hennessy & C.C. Morgan (2008): *Characterising Testing Preorders for Finite Probabilistic Processes*. *Logical Methods in Computer Science* 4(4):4, doi:10.2168/LMCS-4(4:4)2008.
4. R.J. van Glabbeek: *Just testing*. Available at http://theory.stanford.edu/~rvg/abstracts.html#160.
5. R.J. van Glabbeek (2019): *Justness: A Completeness Criterion for Capturing Liveness Properties (extended abstract)*. In M. Bojańczyk & A. Simpson, editors: Proceedings 22st International Conference on *Foundations of Software Science and Computation Structures* (FoSSaCS'19); held as part of the *European Joint Conferences on Theory and Practice of Software* (ETAPS'19), Prague, Czech Republic, April 2019, LNCS 11425, Springer, pp. 505–522, doi:10.1007/978-3-030-17127-8_29.
6. R.J. van Glabbeek (2019): *Reward Testing Equivalences for Processes*. In M. Boreale, F. Corradini, M. Loreti & R. Pugliese, editors: *Models, Languages, and Tools for Concurrent and Distributed Programming*, Essays Dedicated to Rocco De Nicola on the Occasion of His 65th Birthday, LNCS 11665, Springer, pp. 45–70, doi:10.1007/978-3-030-21485-2_5.
7. R.J. van Glabbeek & P. Höfner (2019): *Progress, Justness and Fairness*. *ACM Computing Surveys* 52(4):69, doi:10.1145/3329125.
8. C.A.R. Hoare (1985): *Communicating Sequential Processes*. Prentice Hall.
9. N.A. Lynch & M.R. Tuttle (1989): *An Introduction to Input/Output Automata*. *CWI Quarterly*, 2(3), pp. 219–246. Available at http://groups.csail.mit.edu/tds/papers/Lynch/CWI89.pdf.
10. V. Natarajan & R. Cleaveland (1995): *Divergence and Fair Testing*. In Z. Fülöp & F. Gécseg, editors: Proceedings 22nd International Colloquium on *Automata, Languages and Programming* (ICALP'95), Szeged, Hungary, July 1995, LNCS 944, Springer, pp. 648–659, doi:10.1007/3-540-60084-1_112.
11. R. Segala (1996): *Testing Probabilistic Automata*. In U. Montanari & V. Sassone, editors: Proceedings of the 7th International Conference on *Concurrency Theory*, CONCUR'96, Pisa, Italy, August 1996, LNCS 1119, Springer, pp. 299–314, doi:10.1007/3-540-61604-7_62.

12. R. Segala (1997): *Quiescence, Fairness, Testing, and the Notion of Implementation.* *Information and Computation* 138(2), pp. 194–210, doi:10.1006/inco.1997.2652.
13. F.W. Vaandrager (1991): *On the Relationship Between Process Algebra and Input/Output Automata.* In: *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, IEEE Computer Society, pp. 387–398, doi:10.1109/LICS.1991.151662.