Fair Testing and Stubborn Sets

Antti Valmari[1(✉)] and Walter Vogler[2]

[1] Department of Mathematics, Tampere University of Technology,
P.O. Box 553, 33101 Tampere, Finland
antti.valmari@tut.fi
[2] Institut für Informatik, University of Augsburg, 86135 Augsburg, Germany
walter.vogler@informatik.uni-augsburg.de

Abstract. Partial-order methods alleviate state explosion by considering only a subset of transitions in each constructed state. The choice of the subset depends on the properties that the method promises to preserve. Many methods have been developed ranging from deadlock-preserving to CTL*- and divergence-sensitive branching bisimilarity preserving. The less the method preserves, the smaller state spaces it constructs. Fair testing equivalence unifies deadlocks with livelocks that cannot be exited, and ignores the other livelocks. It is the weakest congruence that preserves whether the ability to make progress can be lost. We prove that a method that was designed for trace equivalence also preserves fair testing equivalence. We describe a fast algorithm for computing high-quality subsets of transitions for the method, and demonstrate its effectiveness on a protocol with a connection and data transfer phase. This is the first practical partial-order method that deals with a practical fairness assumption.

Keywords: Partial-order methods · Fairness · Progress · Fair testing equivalence

1 Introduction

State spaces of systems that consist of many parallel components are often huge. Usually many states arise from executing concurrent transitions in different orders. So-called *partial-order methods* [2,3,5–7,10,11,14,16–18,20] try to reduce the number of states by, roughly speaking, studying only some orders that represent all of them. This is achieved by only investigating a subset of transitions in each state. This subset is usually called *ample*, *persistent*, or *stubborn*. In this study we call it *aps*, when the differences between the three do not matter.

This intuition works well only with executions that lead to a deadlock. However, traces and divergence traces, for instance, arise from not necessarily deadlocking executions. With them, to obtain good reduction results, a constructed execution must often lack transitions and contain additional transitions compared to the executions that it represents. With branching-time properties, thinking in terms of executions is insufficient to start with.

© Springer International Publishing Switzerland 2016
D. Bošnački and A. Wijs (Eds.): SPIN 2016, LNCS 9641, pp. 225–243, 2016.
DOI: 10.1007/978-3-319-32582-8_16

As a consequence, a wide range of aps set methods has been developed. The simplest only preserve the deadlocks (that is, the reduced state space has precisely the same deadlocks as the full state space) [14], while at the other end the CTL* logic (excluding the next state operator) and divergence-sensitive branching bisimilarity are preserved [5,11,16]. The more a method preserves, the worse are the reduction results that it yields. The preservation of the promised properties is guaranteed by stating conditions that the aps sets must satisfy. Various algorithms for computing sets that satisfy the conditions have been proposed. In an attempt to improve reduction results, more and more complicated conditions and algorithms have been developed. There is a trade-off between reduction results on the one hand, and simplicity and the time that it takes to compute an aps set on the other hand.

Consider a cycle where the system does not make progress, but there is a path from it to a progress action. As such, traditional methods for proving liveness treat the cycle as a violation against liveness. However, this is not always the intention. Therefore, so-called *fairness assumptions* are often formulated, stating that the execution eventually leaves the cycle. Unfortunately, how to take them into account while retaining good reduction results has always been a problem for aps set methods. For instance, fairness is not mentioned in the partial order reduction chapter of [2]. Furthermore, as pointed out in [3], the most widely used condition for guaranteeing linear-time liveness (see, e.g., [2, p. 155]) often works in a way that is detrimental to reduction results.

Fair testing equivalence [12] always treats this kind of cycles as progress. If there is no path from a cycle to a progress action, then both fair testing equivalence and the traditional methods treat it as non-progress. This makes fair testing equivalence suitable for catching many non-progress errors, without the need to formulate fairness assumptions; for an application, see Sect. 7.

Fair testing equivalence implies trace equivalence. So it cannot have better reduction methods than trace equivalence. Fair testing equivalence is a branching time notion. Therefore, one might have guessed that any method that preserves it would rely on strong conditions, resulting in bad reduction results. Suprisingly, it turned out that a 20 years old trace-preserving stubborn set method [16] also preserves fair testing equivalence. This is the main result of the present paper. It means that *no reduction power is lost* compared to trace equivalence.

Background concepts are introduced in Sect. 2. Sections 3 and 4 present the trace-preserving method and discuss how it can be implemented. This material makes this publication self-contained, but it also contains some improvements over earlier publications. Section 5 discusses further why it is good to avoid strong conditions. The proof that the method also applies to fair testing equivalence is in Sect. 6. Some performance measurements are presented in Sect. 7.

2 Labelled Transition Systems and Equivalences

In this section we first define labelled transition systems and some operators for composing systems from them. We also define some useful notation.

Then we define the well-known trace equivalence and the fair testing equivalence of [12]. We also define tree failure equivalence, because it is a strictly stronger equivalence with a related but much simpler definition.

The symbol τ denotes the *invisible action*. A *labelled transition system* or *LTS* is a tuple $L = (S, \Sigma, \Delta, \hat{s})$ such that $\tau \notin \Sigma$, $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$ and $\hat{s} \in S$. The elements of S, Σ, and Δ are called *states*, *visible actions*, and *transitions*, respectively. The state $\hat{s}$ is the *initial state*. An *action* is a visible action or τ.

We adopt the convention that, unless otherwise stated, $L' = (S', \Sigma', \Delta', \hat{s}')$, $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, and so on.

The empty string is denoted with ε. We have $\varepsilon \neq \tau$ and $\varepsilon \notin \Sigma$.

Let $n \geq 0$, s and s' be states, and $a_1, \ldots, a_n$ be actions. The notation $s - a_1 \cdots a_n \rightarrow s'$ denotes that there are states $s_0, \ldots, s_n$ such that $s = s_0$, $s_n = s'$, and $(s_{i-1}, a_i, s_i) \in \Delta$ for $1 \leq i \leq n$. The notation $s - a_1 \cdots a_n \rightarrow$ denotes that there is s' such that $s - a_1 \cdots a_n \rightarrow s'$. The set of *enabled* actions of s is defined as $\mathsf{en}(s) = \{a \in \Sigma \cup \{\tau\} \mid s - a \rightarrow\}$.

The *reachable part* of L is defined as the LTS $(S', \Sigma, \Delta', \hat{s})$, where

- $S' = \{s \in S \mid \exists \sigma \in (\Sigma \cup \{\tau\})^* : \hat{s} - \sigma \rightarrow s\}$ and
- $\Delta' = \{(s, a, s') \in \Delta \mid s \in S'\}$.

The *parallel composition* of L_1 and L_2 is denoted with $L_1 \,||\, L_2$. It is the reachable part of $(S, \Sigma, \Delta, \hat{s})$, where $S = S_1 \times S_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $\hat{s} = (\hat{s}_1, \hat{s}_2)$, and $((s_1, s_2), a, (s'_1, s'_2)) \in \Delta$ if and only if

- $(s_1, a, s'_1) \in \Delta_1$, $s'_2 = s_2 \in S_2$, and $a \notin \Sigma_2$,
- $(s_2, a, s'_2) \in \Delta_2$, $s'_1 = s_1 \in S_1$, and $a \notin \Sigma_1$, or
- $(s_1, a, s'_1) \in \Delta_1$, $(s_2, a, s'_2) \in \Delta_2$, and $a \in \Sigma_1 \cap \Sigma_2$.

That is, if a belongs to the alphabets of both components, then an a-transition of the parallel composition consists of simultaneous a-transitions of both components. If a belongs to the alphabet of one but not the other component, then that component may make an a-transition while the other component stays in its current state. Also each τ-transition of the parallel composition consists of one component making a τ-transition without the other participating. The result of the parallel composition is pruned by only taking the reachable part.

It is easy to check that $(L_1 \,||\, L_2) \,||\, L_3$ is isomorphic to $L_1 \,||\, (L_2 \,||\, L_3)$. This means that $||$ can be considered associative, and that $L_1 || \cdots || L_n$ is well-defined for any positive integer n.

The *hiding* of an action set A in L is denoted with $L \backslash A$. It is $L \backslash A = (S, \Sigma', \Delta', \hat{s})$, where $\Sigma' = \Sigma \backslash A$ and $\Delta' = \{(s, a, s') \in \Delta \mid a \notin A\} \cup \{(s, \tau, s') \mid \exists a \in A : (s, a, s') \in \Delta\}$. That is, labels of transitions that are in A are replaced by τ and removed from the alphabet. Other labels of transitions are not affected.

Let $\sigma \in \Sigma^*$. The notation $s = \sigma \Rightarrow s'$ denotes that there are $a_1, \ldots, a_n$ such that $s - a_1 \cdots a_n \rightarrow s'$ and σ is obtained from $a_1 \cdots a_n$ by leaving out each τ. We say that σ is the *trace* of the path $s - a_1 \cdots a_n \rightarrow s'$. The notation $s = \sigma \Rightarrow$ denotes that there is s' such that $s = \sigma \Rightarrow s'$. The set of *traces* of L is

$$\mathsf{Tr}(L) \;=\; \{\sigma \in \Sigma^* \mid \hat{s} = \sigma \Rightarrow\}.$$

The LTSs L_1 and L_2 are *trace equivalent* if and only if $\Sigma_1 = \Sigma_2$ and $\mathsf{Tr}(L_1) = \mathsf{Tr}(L_2)$.

Let L be an LTS, $K \subseteq \Sigma^+$, and $s \in S$. The state s *refuses* K if and only if for every $\sigma \in K$ we have $\neg(s = \sigma \Rightarrow)$. For example, $\hat{s}$ refuses K if and only if $K \cap \mathsf{Tr}(L) = \emptyset$. Because $s = \varepsilon \Rightarrow$ holds vacuously for every state s, this definition is equivalent to what would be obtained with $K \subseteq \Sigma^*$. The pair $(\sigma, K) \in \Sigma^* \times 2^{\Sigma^+}$ is a *tree failure* of L, if and only if there is $s \in S$ such that $\hat{s} = \sigma \Rightarrow s$ and s refuses K. The set of tree failures of L is denoted with $\mathsf{Tf}(L)$. The LTSs L_1 and L_2 are *tree failure equivalent* if and only if $\Sigma_1 = \Sigma_2$ and $\mathsf{Tf}(L_1) = \mathsf{Tf}(L_2)$.

To define the main equivalence of this publication, we also need the following notation: For $\rho \in \Sigma^*$ and $K \subseteq \Sigma^*$, we write $\rho^{-1}K$ for $\{\pi \mid \rho\pi \in K\}$ and call ρ a *prefix* of K if $\rho^{-1}K \neq \emptyset$.

Definition 1. *The LTSs L_1 and L_2 are* fair testing equivalent *if and only if*

1. $\Sigma_1 = \Sigma_2$,
2. *if $(\sigma, K) \in \mathsf{Tf}(L_1)$, then either $(\sigma, K) \in \mathsf{Tf}(L_2)$ or there is a prefix ρ of K such that $(\sigma\rho, \rho^{-1}K) \in \mathsf{Tf}(L_2)$, and*
3. *Part 2 holds with the roles of L_1 and L_2 swapped.*

If $K \neq \emptyset$, then the first option "$(\sigma, K) \in \mathsf{Tf}(L_2)$" implies the other by letting $\rho = \varepsilon$. Therefore, the "either"-part could equivalently be written as "$K = \emptyset$ and $(\sigma, \emptyset) \in \mathsf{Tf}(L_2)$". The way it has been written makes it easy to see that tree failure equivalence implies fair testing equivalence.

If L_1 and L_2 are fair testing equivalent, then $\sigma \in \mathsf{Tr}(L_1)$ implies by the definitions that $(\sigma, \emptyset) \in \mathsf{Tf}(L_1)$, $(\sigma, \emptyset) \in \mathsf{Tf}(L_2)$, and $\sigma \in \mathsf{Tr}(L_2)$. So fair testing equivalence implies trace equivalence and cannot have better reduction methods.

3 The Trace-Preserving Strong Stubborn Set Method

The trace-preserving strong stubborn set method applies to LTS expressions of the form

$$L = (L_1 \mid\mid \cdots \mid\mid L_m) \backslash A.$$

To discuss the method, it is handy to first give indices to the τ-actions of the L_i. Let $\tau_1, \ldots, \tau_m$ be symbols that are distinct from each other and from all elements of $\Sigma = \Sigma_1 \cup \cdots \cup \Sigma_m$. For $1 \leq i \leq m$, we let $\bar{L}_i = (S_i, \bar{\Sigma}_i, \bar{\Delta}_i, \hat{s}_i)$, where

– $\bar{\Sigma}_i = \Sigma_i \cup \{\tau_i\}$ and
– $\bar{\Delta}_i = \{(s, a, s') \mid a \in \Sigma_i \wedge (s, a, s') \in \Delta_i\} \cup \{(s, \tau_i, s') \mid (s, \tau, s') \in \Delta_i\}$.

The trace-preserving strong stubborn set method computes a reduced version of

$$L' = (\bar{L}_1 \mid\mid \cdots \mid\mid \bar{L}_m) \backslash (A \cup \{\tau_1, \ldots, \tau_m\}).$$

For convenience, we define

– $\bar{L} = \bar{L}_1 \mid\mid \cdots \mid\mid \bar{L}_m$,

- $V = \Sigma \backslash A$ (the set of *visible* actions), and
- $I = (\Sigma \cap A) \cup \{\tau_1, \ldots, \tau_m\}$ (the set of *invisible* actions).

Now we can write $L' = (\bar{L}_1 \mid\mid \cdots \mid\mid \bar{L}_m) \backslash I = \bar{L} \backslash I$.

It is obvious from the definitions that L' is the same LTS as L. The only difference between $\bar{L}$ and $L_1 \mid\mid \cdots \mid\mid L_m$ is that the τ-transitions of the latter are τ_i-transitions of the former, where i reveals the L_i from which the transition originates. The hiding of I makes them τ-transitions again. We have $V \cap I = \emptyset$, $V \cup I = \bar{\Sigma} = \Sigma \cup \{\tau_1, \ldots, \tau_m\}$, and $\bar{L}$ has no τ-transitions at all (although it may have τ_i-transitions). Therefore, when discussing the trace-preserving strong stubborn set method, the elements of V and I are called *visible* and *invisible*, respectively.

The method is based on a function $\mathcal{T}$ that assigns to each $s \in S$ a subset of $\bar{\Sigma}$, called *stubborn set*. Before discussing the definition of $\mathcal{T}$, let us see how it is used. The stubborn set method computes a subset of S called S_r and a subset of Δ called Δ_r. It starts by letting $S_\mathsf{r} = \{\hat{s}\}$ and $\Delta_\mathsf{r} = \emptyset$. For each s that it has put to S_r and for each $a \in \mathcal{T}(s)$, it puts to S_r every s' that satisfies $(s, a, s') \in \bar{\Delta}$ (unless s' is already in S_r). Furthermore, it puts (s, a', s') to Δ_r (even if s' is already in S_r), where $a' = \tau$ if $a \in I$ and $a' = a$ otherwise. The only difference to the computation of L' is that in the latter, every $a \in \bar{\Sigma}$ is used instead of every $a \in \mathcal{T}(s)$.

The LTS $L_\mathsf{r} = (S_\mathsf{r}, \Sigma, \Delta_\mathsf{r}, \hat{s})$ is the *reduced* LTS, while $L = L' = (S, \Sigma, \Delta, \hat{s})$ is the *full* LTS. We will refer to concepts in L_r with the prefix "r-", and to L with "f-". For instance, if $s \in S_\mathsf{r}$ and $\neg(s = \sigma \Rightarrow)$ holds in L_r for all $\sigma \in K$, then s is an r-state and s r-refuses K. Because $S_\mathsf{r} \subseteq S$ and $\Delta_\mathsf{r} \subseteq \Delta$, every r-state is also an f-state and every r-trace is an f-trace. We will soon state conditions on $\mathcal{T}$ that guarantee that also the opposite holds, that is, every f-trace is an r-trace.

Typically many different functions could be used as $\mathcal{T}$, and the choice between them involves trade-offs. For example, a function may be easy and fast to compute, but it may also tend to give worse reduction results (that is, bigger S_r and Δ_r) than another more complex function. Therefore, we will not specify a unique function $\mathcal{T}$. Instead, in the remainder of this section we will only give four conditions that it must satisfy, and in the next section we will discuss how a reasonably good $\mathcal{T}$ is computed quickly.

A function from states to subsets of $\bar{\Sigma}$ qualifies as $\mathcal{T}$ if and only if for every $s \in S_\mathsf{r}$ it satisfies the four conditions below (the first two are illustrated in Fig. 1):

Fig. 1. Illustrating D1 (left) and D2 (right). The solid states and transition sequences are assumed to exist and the condition promises the existence of the dashed ones. The yellow (grey in black/white print) part is in the reduced LTS, the rest is not necessarily

D1 If $a \in \mathcal{T}(s)$, $a_1, \ldots, a_n$ are not in $\mathcal{T}(s)$, and $s -a_1 \cdots a_n a\rightarrow s'_n$, then $s - aa_1 \cdots a_n \rightarrow s'_n$.

D2 If $a \in \mathcal{T}(s)$, $a_1, \ldots, a_n$ are not in $\mathcal{T}(s)$, $s -a\rightarrow s'$, and $s -a_1 \cdots a_n \rightarrow s_n$, then there is s'_n such that $s' -a_1 \cdots a_n \rightarrow s'_n$ and $s_n -a\rightarrow s'_n$.

V If $\mathcal{T}(s) \cap V \cap \mathsf{en}(s) \neq \emptyset$, then $V \subseteq \mathcal{T}(s)$.

S For each $a \in V$ there is an r-state s_a and an r-path from s to s_a such that $a \in \mathcal{T}(s_a)$.

Intuitively, D1 says two things. First, it says that a sequence of actions that are not in the current stubborn set ($a_1 \cdots a_n$ in the definition) cannot enable an action that is in the current stubborn set (a in the definition). That is, disabled actions in a stubborn set remain disabled while actions outside the set occur. Second, together with D2 it says that the enabled actions inside the current stubborn set are in a certain kind of a commutativity relation with enabled sequences of outside actions. In theories where actions are deterministic (that is, for every s, s_1, s_2, and a, $s -a\rightarrow s_1$ and $s -a\rightarrow s_2$ imply $s_1 = s_2$), the then-part of D2 is usually written simply as $s_n -a\rightarrow$. It, D1, and determinism imply our current version of D2. However, we do not assume that actions are deterministic.

Certain partial-order semantic models of concurrency use a so-called independence relation [9]. Unlike in the present study, actions are assumed to be deterministic. If a_1 and a_2 are independent, then (1) if $s -a_1\rightarrow s_1$ and $s -a_2\rightarrow s_2$, then there is an s' such that $s_1 -a_2\rightarrow s'$ and $s_2 -a_1\rightarrow s'$; (2) if $s -a_1 a_2\rightarrow$ then $s -a_2\rightarrow$; and (3) if $s -a_2 a_1\rightarrow$ then $s -a_1\rightarrow$. It is often claimed that ample, persistent, and stubborn set methods rely on an independence relation. This is why they are classified as "partial-order methods". In reality, they rely on various strictly weaker relations. For instance, even if determinism is assumed, D1 and D2 do not imply independence of a_1 from a, because they fail to yield (3).

The names D1 and D2 reflect the fact that together with a third condition called D0, they guarantee that the reduced LTS has precisely the same terminal states – also known as deadlocks – as the full LTS. D0 is not needed in the present method, because now the purpose is not to preserve deadlocks but traces. However, because we do not yet have an implementation that has been optimized to the present method, in our experiments in Sect. 7 we used a tool that relies on D0 and implements it. Therefore, we present its definition:

D0 If $\mathsf{en}(s) \neq \emptyset$, then $\mathcal{T}(s) \cap \mathsf{en}(s) \neq \emptyset$.

That is, if s is not a deadlock, then $\mathcal{T}(s)$ contains an enabled action. We skip the (actually simple) proof that D0, D1, and D2 guarantee that deadlocks are preserved, see [16].

The condition V says that if the stubborn set contains an enabled visible action, then it contains all visible actions (also disabled ones). It guarantees that the reduction preserves the ordering of visible actions, in a sense that will become clear in the proof of Lemma 3.

The function $\mathcal{T}_\emptyset$ that always returns the empty set satisfies D1, D2, and V. Its use as $\mathcal{T}$ would result in a reduced LTS that has one state and no transitions.

It is thus obvious that D1, D2, and V alone do not guarantee that the reduced LTS has the same traces as the full LTS.

The condition S forces the method to investigate, intuitively speaking, everything that is relevant for the preservation of the traces. It does that by guaranteeing that every visible action is taken into account, not necessarily in the current state but necessarily in a state that is r-reachable from the current state. Taking always all visible actions into account in the current state would make the reduction results much worse. The name is S because, historically, a similar condition was first used to guarantee the preservation of what is called safety properties in the linear temporal logic framework. Again, the details of how S does its job will become clear in the proof of Lemma 3.

If $V = \emptyset$, then $\mathcal{T}_\emptyset$ satisfies also S. Indeed, then $\mathsf{Tr}(L) = \{\varepsilon\} = \mathsf{Tr}(L_\mathsf{r})$ even if L_r is the one-state LTS that has no transitions. That is, if $V = \emptyset$, then $\mathcal{T}_\emptyset$ satisfies the definition and yields ideal reduction results.

No matter what V is, the function $\mathcal{T}(s) = \bar{\Sigma}$ always satisfies D1, D2, V, and S. However, it does not yield any reduction. The problem of computing sets that satisfy D1, D2, V, and S and do yield reduction will be discussed in Sect. 4. In Sect. 6 we prove that D1, D2, V, and S guarantee that the reduced LTS is fair testing equivalent (and thus also trace equivalent) to the full LTS.

4 On Computing Trace-Preserving Stubborn Sets

To make the abstract theory in the remainder of this publication more concrete, we present in this section one new good way of computing sets that satisfy D1, D2, V, and S. It is based on earlier ideas but has been fine-tuned for the present situation. We emphasize that it is not the only good way. Other possibilities have been discussed in [17,20], among others.

Because the expression under analysis is of the form $(\bar{L}_1 \mid\mid \cdots \mid\mid \bar{L}_m) \backslash I$, its states are of the form $(s_1, \ldots, s_m)$, where $s_i \in L_i$ for each $1 \leq i \leq m$. We employ the notation $\mathsf{en}_i(s_i) = \{a \mid \exists s_i' : (s_i, a, s_i') \in \Delta_i\}$, that is, the set of actions that are enabled in s_i in L_i. We have $\tau \notin \mathsf{en}_i(s_i) \subseteq \bar{\Sigma}_i = \Sigma_i \cup \{\tau_i\}$. Furthermore, if $a \notin \mathsf{en}(s)$, then there is at least one i such that $a \in \bar{\Sigma}_i$ and $a \notin \mathsf{en}_i(s_i)$. Let $\mathsf{dis}(s, a)$ denote the smallest such i.

We start by presenting a sufficient condition for D1 and D2 that does not refer to other states than the current.

Theorem 2. *Assume that the following hold for $s = (s_1, \ldots, s_m)$ and for every $a \in \mathcal{T}(s)$:*

1. *If $a \notin \mathsf{en}(s)$, then there is i such that $a \in \bar{\Sigma}_i$ and $a \notin \mathsf{en}_i(s_i) \subseteq \mathcal{T}(s)$.*
2. *If $a \in \mathsf{en}(s)$, then for every i such that $a \in \bar{\Sigma}_i$ we have $\mathsf{en}_i(s_i) \subseteq \mathcal{T}(s)$.*

Then $\mathcal{T}(s)$ satisfies D1 and D2.

Proof. Let $a_1 \notin \mathcal{T}(s)$, ..., $a_n \notin \mathcal{T}(s)$.

Let first $a \notin \mathsf{en}(s)$. Obviously $s -a\rightarrow$ does not hold, so D2 is vacuously true. We prove now that D1 is as well. By assumption 1, there is i such that L_i

disables a and $\mathsf{en}_i(s_i) \subseteq \mathcal{T}(s)$. To enable a, it is necessary that L_i changes its state, which requires that some action in $\mathsf{en}_i(s_i)$ occurs. These are all in $\mathcal{T}(s)$ and thus distinct from $a_1, \ldots, a_n$. So $s - a_1 \cdots a_n a \rightarrow$ cannot hold.

Let now $a \in \mathsf{en}(s)$. Our next goal is to show that there are no $1 \leq k \leq n$ and $1 \leq j \leq m$ such that both $a \in \bar{\Sigma}_j$ and $a_k \in \bar{\Sigma}_j$. To derive a contradiction, consider a counterexample where k has the smallest possible value. So none of $a_1, \ldots, a_{k-1}$ is in $\bar{\Sigma}_j$. If $s - a_1 \cdots a_n \rightarrow$, then there is s' such that $s - a_1 \cdots a_{k-1} \rightarrow s' - a_k \rightarrow$. Obviously $a_k \in \mathsf{en}_j(s'_j)$. This implies $a_k \in \mathsf{en}_j(s_j)$, because L_j does not move between s and s' since none of $a_1, \ldots, a_{k-1}$ is in $\bar{\Sigma}_j$. By assumption 2, $\mathsf{en}_j(s_j) \subseteq \mathcal{T}(s)$. This contradicts $a_k \notin \mathcal{T}(s)$.

This means that the L_j that participate in a are disjoint from the L_j that participate in $a_1 \cdots a_n$. From this D1 and D2 follow by well-known properties of the parallel composition operator. □

Theorem 2 makes it easy to represent a sufficient condition for D1 and D2 as a directed graph that depends on the current state s. The set of the vertices of the graph is $\bar{\Sigma}$. There is an edge from $a \in \bar{\Sigma}$ to $b \in \bar{\Sigma}$, denoted with $a \rightsquigarrow b$, if and only if either $a \notin \mathsf{en}(s)$ and $b \in \mathsf{en}_i(s_i)$ where $i = \mathsf{dis}(s, a)$, or $a \in \mathsf{en}(s)$ and there is i such that $a \in \bar{\Sigma}_i$ and $b \in \mathsf{en}_i(s_i)$. By the construction, if $\mathcal{T}(s)$ is closed under the graph (that is, for every a and b, if $a \in \mathcal{T}(s)$ and $a \rightsquigarrow b$, then $b \in \mathcal{T}(s)$), then $\mathcal{T}(s)$ satisfies D1 and D2.

It is not necessary for correctness to use the smallest i, when more than one L_i disables a. The choice to use the smallest i was made to obtain a fast algorithm. An alternative algorithm (called *deletion algorithm* in [17]) is known that exploits the freedom to choose any i that disables a. It has the potential to yield smaller reduced LTSs than the algorithm described in this section. On the other hand, it consumes more time per constructed state.

Furthermore, the condition in Theorem 2 is not the weakest possible, as shown by the following useful observation: Assume a writes to a finite-capacity fifo L_f, $\bar{a}$ reads from it, and they have no other L_i in common; although $\bar{\Sigma}_f$ links them, we need not declare $a \rightsquigarrow \bar{a}$ when a is enabled, and we need not declare $\bar{a} \rightsquigarrow a$ when $\bar{a}$ is enabled, since they commute if both are enabled. Trying to make the condition as weak as possible would have made it very hard to read.

It is trivial to also take the condition V into account in the graph representation of the stubborn set computation problem. It suffices to add the edge $a \rightsquigarrow b$ from each $a \in V \cap \mathsf{en}(s)$ to each $b \in V$.

Let "$\rightsquigarrow^*$" denote the reflexive transitive closure of "$\rightsquigarrow$". By the definitions, if $a \in \bar{\Sigma}$, then $\{b \mid a \rightsquigarrow^* b\}$ satisfies D1, D2, and V. We denote it with $\mathsf{clsr}(a)$. It can be computed quickly with well-known elementary graph search algorithms.

However, we can do better. The better algorithm is denoted with $\mathsf{esc}(a)$, for "enabled strong component". Applied at some state s, it uses a as the starting point of a depth-first search in ($\bar{\Sigma}$, "$\rightsquigarrow$"). During the search, the strong components (i.e., the maximal strongly connected subgraphs) of ($\bar{\Sigma}$, "$\rightsquigarrow$") are recognized using Tarjan's algorithm [4,13]. It recognizes each strong component at the time of backtracking from it. When $\mathsf{esc}(a)$ finds a strong component C that contains an action enabled at s, it stops and returns C as the result; note

that a might not be in C. In principle, the result should also contain actions that are reachable from C but are not in C. However, they are all disabled, so leaving them out does not change L_{r}, which we are really interested in. If $\mathsf{esc}(a)$ does not find such a strong component, it returns $\emptyset$.

Obviously $\mathsf{esc}(a) \subseteq \mathsf{clsr}(a)$. So $\mathsf{esc}(a)$ has potential for better reduction results. Tarjan's algorithm adds very little overhead to depth-first search. Therefore, $\mathsf{esc}(a)$ is never much slower than $\mathsf{clsr}(a)$. On the other hand, it may happen that $\mathsf{esc}(a)$ finds a suitable strong component early on, in which case it is much faster than $\mathsf{clsr}(a)$.

To discuss the implementation of S, let $V = \{a_1, \ldots, a_{|V|}\}$. Let $\mathrm{S}(s, i)$ denote that there is an s_i and an r-path from s to s_i such that $a_i \in \mathcal{T}(s_i)$. Our algorithm constructs L_{r} in depth-first order. The *root* of a strong component C of L_{r} is the state in C that was found first. Our algorithm recognizes the roots with Tarjan's algorithm. In each root s_C, it enforces $\mathrm{S}(s_C, i)$ for each $1 \leq i \leq n$ in a manner which is discussed below. This suffices, because if $\mathrm{S}(s, i)$ holds for one state in a strong component, then it clearly holds for every state in the component.

Each state s has an attribute ν such that if $\nu > |V|$, then $\mathrm{S}(s, i)$ is known to hold for $a_1, \ldots, a_{\nu - |V|}$. When a state is processed for the first time, its ν value is set to 1 and $\mathsf{esc}(a_1)$ is used as its stubborn set. When the algorithm is about to backtrack from a root s_C, it checks its ν value. The algorithm actually backtracks from a root only when $\nu = 2|V|$. Otherwise it increments ν by one. Then it extends $\mathcal{T}(s_C)$ by $\mathsf{esc}(a_\nu)$ if $\nu \leq |V|$, and by $\mathsf{clsr}(a_{\nu - |V|})$ if $|V| < \nu \leq 2|V|$. The extension may introduce new outgoing transitions for s_C, and s_C may cease from being a root. If s_C remains a root, then its ν eventually grows to $2|V|$ and S holds for s_C. The purpose of making $\mathcal{T}(s_C)$ grow in steps with esc-sets first is to obtain as small a stubborn set as possible, if s_C ceases from being a root.

During the depth-first search, information on ν-values is backward propagated and the maximum is kept. This way, if s_C ceases from being a root, the new root benefits from the work done at s_C. Furthermore, non-terminal strong components automatically get $\nu = 2|V|$. To exploit situations where $V \subseteq \mathcal{T}(s)$ by condition V, if a visible action is in $\mathsf{en}(s) \cap \mathcal{T}(s)$, then the algorithm makes the ν value of s be $2|V|$.

Unfortunately, we do not yet have an implementation of this algorithm. Therefore, in our experiments in Sect. 7 we used a trick. A system is *always may-terminating* if and only if, from every reachable state, the system is able to reach a deadlock. For each deadlock s, we can pretend that $\mathcal{T}(s) = \bar{\Sigma}$ and thus that $V \subseteq \mathcal{T}(s)$, because $\mathcal{T}(s)$ contains no enabled actions no matter how we choose it. This implies that S holds automatically for always may-terminating systems. In [18] it was proven that if, instead of S, the condition D0 is used, then it is easy to check from the reduced LTS whether the system is always may-terminating. So we will use the following new approach in Sect. 7:

1. Try to make the system always may-terminating.
2. Construct L'_{r} obeying D0, D1, D2, and V.
3. If L'_{r} is always may-terminating, then extract a reduced LTS for the original system from L'_{r} as will be described in Sect. 7. Otherwise, go back to 1.

Stubborn sets obeying D0, D1, D2, and V can be computed by, in each state that is not a deadlock, choosing an enabled a and computing $\mathsf{esc}(a)$.

5 On the Performance of Various Conditions

The goal of aps set methods is to alleviate the state explosion problem. Therefore, reducing the size of the state space is a main issue. However, if the reduction introduces too much additional work per preserved state, then time is not saved. So the cost of computing the aps set is important. Also the software engineering issue plays a role. Little is known on the practical performance of ideas that have the biggest theoretical reduction potential, because they are complicated to implement, so few experiments have been made. For instance, first big experiments on weak stubborn sets [17] and the deletion algorithm [17] appeared in [8].

Often a state has more than one aps set. Let T_1 and T_2 be two of them and let $\mathcal{E}(T_1)$ and $\mathcal{E}(T_2)$ be the sets of enabled transitions in T_1 and T_2. It is obvious that if the goal is to preserve deadlocks and if $\mathcal{E}(T_1) \subseteq \mathcal{E}(T_2)$, then T_1 can lead to better but cannot lead to worse reduction results than T_2. We are not aware of any significant result on the question which should be chosen, T_1 or T_2, if both are aps, $\mathcal{E}(T_1) \not\subseteq \mathcal{E}(T_2)$, and $\mathcal{E}(T_2) \not\subseteq \mathcal{E}(T_1)$. Let us call it the *non-subset choice problem.* Already [15] gave an example where always choosing the set with the smallest number of enabled transitions does not yield the best reduction result.

We now demonstrate that the order in which the components of a system are given to a tool can have a tremendous effect on the running time and the size of the reduced state space. Assume that $L_1 \mid\mid \cdots \mid\mid L_m$ has deadlocks. Consider $L_1 \mid\mid \cdots \mid\mid L_m \mid\mid L_{m+1}$, where $L_{m+1} = \circlearrowleft \tau$. This extended system has no deadlocks. If the deadlock-preserving stubborn set method always investigates L_{m+1} last, then it finds the deadlocks of the original system in the original fashion, finds that L_{m+1} is enabled in them, and eventually concludes that the system has no deadlocks. So it does approximately the same amount of work as it does with $L_1 \mid\mid \cdots \mid\mid L_m$. If, in the initial state $\hat{s}$, the method happens to investigate L_{m+1} first, it finds a τ-loop $\hat{s} -\tau\rightarrow \hat{s}$. D0, D1, and D2 do not tell it to investigate anything else. So it stops extremely quickly, after constructing only one state.

For this and other reasons, measurements are not as reliable for comparing different methods as we would like them to be.

Technically, optimal sets could be defined as those (not necessarily aps) sets of enabled transitions that yield the smallest reduced state space that preserves the deadlocks. Unfortunately, it was shown in [20] that finding subsets of transitions of a 1-safe Petri net that are optimal in this sense is at least as hard as testing whether the net has a deadlock. Another similar result was proven in [2, p. 154]. Therefore, without additional assumptions, optimal sets are too hard to find.

This negative result assumes that optimality is defined with respect to all possible ways of obtaining information on the behaviour of the system. Indeed, optimal sets can be found by first constructing and investigating the full state space. Of course, aps set methods do not do so, because constructing the full

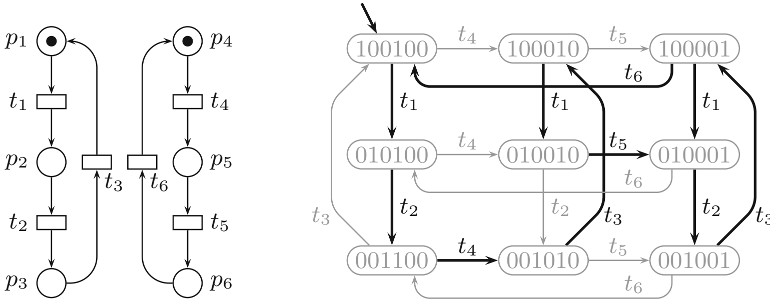

Fig. 2. Transitions are tried in the order of their indices until one is found that does not close a cycle. If such a transition is not found, then all transitions are taken

state space is what they try to avoid. In [20], a way of obtaining information was defined such that most (but not all) deadlock-preserving aps set methods conform to it. Using non-trivial model-theoretic reasoning, it was proven in [20] that, in the case of 1-safe Petri nets, the best possible (not necessarily aps) sets that can be obtained in this context are of the form $\mathcal{E}(T_s)$, where T_s is stubborn. In this restricted but nevertheless meaningful sense, stubborn sets are optimal.

The situation is much more complicated when preserving other properties than deadlocks. We only discuss one difficulty. Instead of S, [2, p. 155] assumes that the reduced state space is constructed in depth-first order and tells to choose an aps set that does not close a cycle if possible, and otherwise use all enabled transitions. Figure 2 shows an example where this condition leads to the construction of all reachable states, although the processes do not interact at all. The condition S is less vulnerable to but not totally free from this kind of difficulties. Far too little is known on this problem area.

The approach in Sects. 4 and 7 that does not use S is entirely free of this difficulty. This is one reason why it seems very promising.

In general, it is reasonable to try to find as weak conditions as possible in place of D1, V, S, and so on, because the weaker a condition is, the more potential it has for good reduction results. Because of the non-subset choice problem and other similar problems, it is not certain that the potential can be exploited in practice. However, if the best set is ruled out already by the choice of the condition, then it is certain that it cannot be exploited.

For instance, instead of V, [2, p. 149] requires that if $\mathcal{T}(s) \cap V \cap \mathsf{en}(s) \neq \emptyset$, then $\mathcal{T}(s)$ must contain all enabled transitions. This condition is strictly stronger than V and thus has less potential for reduction. Furthermore, the algorithm in Sect. 4 can exploit the additional potential of V at least to some extent.

This also illustrates why stubborn sets are defined such that they may contain disabled transitions. The part $V \subseteq \mathcal{T}(s)$ in the definition of condition V could not be formulated easily, or perhaps not at all, if $\mathcal{T}(s)$ cannot contain disabled transitions. The following example reveals both that $V \cap \mathsf{en}(s) \subseteq \mathcal{T}(s)$ fails

(it loses the trace b) and that V yields better reduction than the condition in [2] ($\{a, b, u, \tau_2\}$ is stubborn, satisfies V, but $\tau_3 \in \mathsf{en}(s) \not\subseteq \{a, b, u, \tau_2\}$):

$$(\downarrow\!\circ \xrightarrow{a} \circ \xrightarrow{u} \circ \xrightarrow{a} \circ \;||\; \circ \xleftarrow{u} \downarrow\!\circ \xrightarrow{\tau_2} \circ \xrightarrow{v} \circ \xrightarrow{b} \circ \;||\; \circ \xleftarrow{\tau_3} \downarrow\!\circ \xrightarrow{v} \circ) \setminus \{u, v\}$$

6 The Fair Testing Equivalence Preservation Theorem

In this section we assume that $L_\mathsf{r} = (S_\mathsf{r}, \Sigma, \Delta_\mathsf{r}, \hat{s})$ has been constructed with the trace-preserving strong stubborn set method, that is, obeying D1, D2, V, and S. We show that L_r is fair testing equivalent to L, where $L = (S, \Sigma, \Delta, \hat{s})$ denotes the corresponding full LTS, based on a series of lemmata. Lemma 4 shows that a trace leaving S_r can be found inside S_r, and Lemma 3 treats a step for this. Similarly, Lemmas 5 and 6 show how to transfer a refusal set in a suitable way.

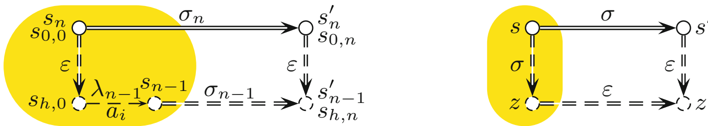

Fig. 3. Illustrating Lemma 3 (left) and Lemma 4 (right)

Lemma 3. *Assume that* $n \in \mathbb{N}$, $s_n \in S_\mathsf{r}$, $s'_n \in S$, $\varepsilon \neq \sigma_n \in V^*$, *and there is an f-path of length* n *from* s_n *to* s'_n *such that its trace is* σ_n. *There are* $s_{n-1} \in S_\mathsf{r}$, $s'_{n-1} \in S$, $\lambda_{n-1} \in V \cup \{\varepsilon\}$, *and* $\sigma_{n-1} \in V^*$ *such that* $\lambda_{n-1}\sigma_{n-1} = \sigma_n$, $s_n = \lambda_{n-1} \Rightarrow s_{n-1}$ *in* L_r, $s'_n = \varepsilon \Rightarrow s'_{n-1}$ *in* L, *and there is an f-path of length* $n-1$ *from* s_{n-1} *to* s'_{n-1} *such that its trace is* σ_{n-1}.

Proof. Let $s_{0,0} = s_n$ and $s_{0,n} = s'_n$. Let the f-path of length n be $s_{0,0} - a_1 \cdots a_n \rightarrow s_{0,n}$. Because $\sigma_n \neq \varepsilon$, there is a smallest v such that $1 \leq v \leq n$ and $a_v \in V$. By S, there are $k \in \mathbb{N}$, $s_{1,0}, \ldots, s_{k,0}$, and $b_1, \ldots, b_k$ such that $a_v \in \mathcal{T}(s_{k,0})$ and $s_{0,0} - b_1 \rightarrow s_{1,0} - b_2 \rightarrow \ldots - b_k \rightarrow s_{k,0}$ in L_r. Let h be the smallest natural number such that $\{a_1, \ldots, a_n\} \cap \mathcal{T}(s_{h,0}) \neq \emptyset$. Because $a_v \in \mathcal{T}(s_{k,0})$, we have $0 \leq h \leq k$. By h applications of D2 at $s_{0,0}, \ldots, s_{h-1,0}$, there are $s_{1,n}, \ldots, s_{h,n}$ such that $s_{i,0} - a_1 \cdots a_n \rightarrow s_{i,n}$ in L for $1 \leq i \leq h$ and $s_{0,n} - b_1 \rightarrow s_{1,n} - b_2 \rightarrow \ldots - b_h \rightarrow s_{h,n}$ in L. If $b_i \in V$ for some $1 \leq i \leq h$, then $V \subseteq \mathcal{T}(s_{i-1,0})$ by V. It yields $a_v \in \mathcal{T}(s_{i-1,0})$, which contradicts the choice of h. As a consequence, $s_{0,0} = \varepsilon \Rightarrow s_{h,0}$ in L_r and $s_{0,n} = \varepsilon \Rightarrow s_{h,n}$ in L.

Because $\{a_1, \ldots, a_n\} \cap \mathcal{T}(s_{h,0}) \neq \emptyset$, there is a smallest i such that $1 \leq i \leq n$ and $a_i \in \mathcal{T}(s_{h,0})$. By D1 at $s_{h,0}$, there is s_{n-1} such that $s_{h,0} - a_i \rightarrow s_{n-1}$ in L_r and $s_{n-1} - a_1 \cdots a_{i-1}a_{i+1} \cdots a_n \rightarrow s_{h,n}$ in L. We choose $s'_{n-1} = s_{h,n}$ and let σ_{n-1} be the trace of $a_1 \cdots a_{i-1}a_{i+1} \cdots a_n$. If $a_i \notin V$, then we choose $\lambda_{n-1} = \varepsilon$, yielding $\lambda_{n-1}\sigma_{n-1} = \sigma_n$. If $a_i \in V$, then $V \subseteq \mathcal{T}(s_{h,0})$ by V, so none of $a_1, \ldots, a_{i-1}$ is in V,

and by choosing $\lambda_{n-1} = a_i$ we obtain $\lambda_{n-1}\sigma_{n-1} = \sigma_n$. That $s_n = \lambda_{n-1} \Rightarrow s_{n-1}$ in L_r follows from $s_{0,0} = \varepsilon \Rightarrow s_{h,0} - a_i \rightarrow s_{n-1}$ in L_r. The rest of the claim is obtained by replacing s'_n for $s_{0,n}$ and s'_{n-1} for $s_{h,n}$ in already proven facts. □

Lemma 4. *Let $n \in \mathbb{N}$. Assume that $s \in S_\mathsf{r}$, $s' \in S$, $\sigma \in V^*$, and $s = \sigma \Rightarrow s'$ in L due to an f-path of length n. Then there are $z \in S_\mathsf{r}$ and $z' \in S$ such that $s = \sigma \Rightarrow z$ in L_r, $z = \varepsilon \Rightarrow z'$ in L, and $s' = \varepsilon \Rightarrow z'$ in L.*

Proof. The proof is by induction on n. We start with the observation that, in case $\sigma = \varepsilon$, the claim holds with choosing $z = s$ and $z' = s'$. This settles the base case $n = 0$ and a subcase of the induction step, and it leaves us with the case $n > 0$ and $\sigma \neq \varepsilon$.

We apply Lemma 3 and get $s_1 \in S_\mathsf{r}$, $s'_1 \in S$, $\sigma_1 \in V^*$, and $\lambda_1 \in V \cup \{\varepsilon\}$ such that $\lambda_1\sigma_1 = \sigma$, $s = \lambda_1 \Rightarrow s_1$ in L_r, and $s' = \varepsilon \Rightarrow s'_1$ in L. Furthermore, $s_1 = \sigma_1 \Rightarrow s'_1$ in L due to an f-path of length $n - 1$, for which the lemma holds; hence, there are $z \in S_\mathsf{r}$ and $z' \in S$ such that $s_1 = \sigma_1 \Rightarrow z$ in L_r, $z = \varepsilon \Rightarrow z'$ in L, and $s'_1 = \varepsilon \Rightarrow z'$ in L. Together, these also give $s = \lambda_1 \Rightarrow s_1 = \sigma_1 \Rightarrow z$ in L_r and $s' = \varepsilon \Rightarrow s'_1 = \varepsilon \Rightarrow z'$ in L, so we are done. □

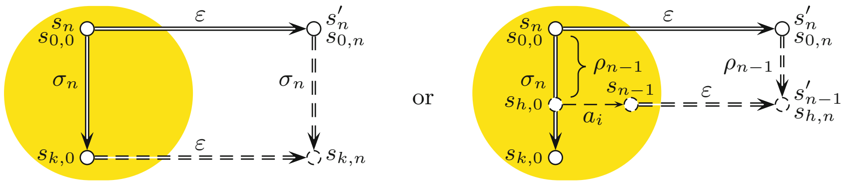

Fig. 4. Illustrating Lemma 5; a_i is invisible

Lemma 5. *Assume that $n \in \mathbb{N}$, $s_n \in S_\mathsf{r}$, $s'_n \in S$, $\sigma_n \in V^*$, $s_n = \sigma_n \Rightarrow$ in L_r, and there is an f-path of length n from s_n to s'_n such that its trace is ε. Either $s'_n = \sigma_n \Rightarrow$ in L or there are $s_{n-1} \in S_\mathsf{r}$, $s'_{n-1} \in S$, and ρ_{n-1} such that ρ_{n-1} is a prefix of σ_n, $s_n = \rho_{n-1} \Rightarrow s_{n-1}$ in L_r, $s'_n = \rho_{n-1} \Rightarrow s'_{n-1}$ in L, and there is an f-path of length $n - 1$ from s_{n-1} to s'_{n-1} such that its trace is ε.*

Proof. Let $s_{0,0} = s_n$ and $s_{0,n} = s'_n$. Let the f-path of length n be $s_{0,0} - a_1 \cdots a_n \rightarrow s_{0,n}$; obviously, the a_i are invisible. By the assumption, there is a path $s_{0,0} - b_1 \rightarrow s_{1,0} - b_2 \rightarrow \ldots - b_k \rightarrow s_{k,0}$ in L_r such that its trace is σ_n.

If $\{a_1, \ldots, a_n\} \cap \mathcal{T}(s_{i,0}) = \emptyset$ for $0 \leq i < k$, then k applications of D2 yield $s_{1,n}, \ldots, s_{k,n}$ such that $s_{0,n} - b_1 \rightarrow s_{1,n} - b_2 \rightarrow \ldots - b_k \rightarrow s_{k,n}$ in L. This implies $s'_n = \sigma_n \Rightarrow$ in L.

Otherwise, there is a smallest h such that $0 \leq h < k$ and $\{a_1, \ldots, a_n\} \cap \mathcal{T}(s_{h,0}) \neq \emptyset$. There also is a smallest i such that $1 \leq i \leq n$ and $a_i \in \mathcal{T}(s_{h,0})$. Applying D2 h times yields $s_{1,n}, \ldots, s_{h,n}$ such that $s_{0,n} - b_1 \rightarrow \ldots - b_h \rightarrow s_{h,n}$ in L and $s_{h,0} - a_1 \cdots a_n \rightarrow s_{h,n}$ in L. By D1 there is s_{n-1} such that $s_{h,0} - a_i \rightarrow s_{n-1}$ in L_r and $s_{n-1} - a_1 \cdots a_{i-1}a_{i+1} \cdots a_n \rightarrow s_{h,n}$ in L. The claim follows by choosing $s'_{n-1} = s_{h,n}$ and letting ρ_{n-1} be the trace of $s_{0,0} - b_1 \cdots b_h \rightarrow s_{h,0}$. □

Lemma 6. *Let $n \in \mathbb{N}$. Assume $K \subseteq V^*$, $\rho \in K$, $z \in S_\mathsf{r}$, $z' \in S$, and $z = \varepsilon \Rightarrow z'$ due to an f-path of length n; assume further that z' f-refuses K and $z = \rho \Rightarrow$ in L_r. Then there exist $s \in S_\mathsf{r}$ and a prefix π of K such that $z = \pi \Rightarrow s$ in L_r and s r-refuses $\pi^{-1}K$.*

Proof. The proof is by induction on n. The case $n = 0$ holds vacuously, since it would imply $z' = \rho \Rightarrow$, contradicting $\rho \in K$.

So we assume the lemma to hold for $n - 1$, and also the assumptions in the lemma for n. We apply Lemma 5 to z, z', and ρ. In the first case, we would again have the impossible $z' = \rho \Rightarrow$. So according to the second case, we have a z_1, z_1', and prefix ρ' of ρ and thus of K with $z = \rho' \Rightarrow z_1$ in L_r, $z' = \rho' \Rightarrow z_1'$ in L, and $z_1 = \varepsilon \Rightarrow z_1'$ due to an f-path of length $n - 1$.

Since z' f-refuses K, z_1' must f-refuse $\rho'^{-1}K$. If z_1 r-refuses $\rho'^{-1}K$, we are done. Otherwise, we can apply the induction hypothesis to $z_1 = \varepsilon \Rightarrow z_1'$ and $\rho'^{-1}K$. This results in an $s \in S_\mathsf{r}$ and a prefix π' of $\rho'^{-1}K$ such that $z_1 = \pi' \Rightarrow s$ in L_r and s r-refuses $\pi'^{-1}\rho'^{-1}K = (\rho'\pi')^{-1}K$. We also have that $\rho'\pi'$ is a prefix of K and $z = \rho'\pi' \Rightarrow s$ in L_r, so we are done. □

Theorem 7. *The LTS L_r is fair testing equivalent to L.*

Proof. Part 1 of Definition 1 is immediate from the construction.

Let (σ, K) be a tree failure of L_r. That is, there is $s \in S_\mathsf{r}$ such that $\hat{s} = \sigma \Rightarrow s$ in L_r and s r-refuses K. Consider any $\rho \in V^*$ such that $s = \rho \Rightarrow$ in L. By Lemma 4, $s = \rho \Rightarrow$ also in L_r. This implies that s refuses K in L and that (σ, K) is a tree failure of L. In conclusion, Part 2 of Definition 1 holds.

Let (σ, K) be a tree failure of L. That is, there is $s' \in S$ such that $\hat{s} = \sigma \Rightarrow s'$ in L and s' f-refuses K. By Lemma 4 there are $z \in S_\mathsf{r}$ and $z' \in S$ such that $\hat{s} = \sigma \Rightarrow z$ in L_r, $s' = \varepsilon \Rightarrow z'$ in L, and $z = \varepsilon \Rightarrow z'$ in L. Since s' f-refuses K, also z' f-refuses K.

Either z r-refuses K and we are done, or we apply Lemma 6, giving us an $s \in S_\mathsf{r}$ and a prefix π of K such that $z = \pi \Rightarrow s$ in L_r and s r-refuses $\pi^{-1}K$. Hence, $(\sigma\pi, \pi^{-1}K) \in \mathsf{Tf}(L_\mathsf{r})$ and Part 3 of Definition 1 also holds. □

Let us conclude this section with a counterexample that shows that the method does not preserve tree failure equivalence.

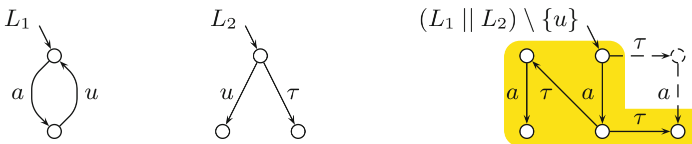

Fig. 5. A counterexample to the preservation of all tree failures. In $(L_1 \,||\, L_2) \setminus \{u\}$, the solid states and transitions are in the reduced and the dashed ones only in the full LTS

Consider $(L_1 \,||\, L_2) \setminus \{u\}$, where L_1 and L_2 are shown in Fig. 5 left and middle. Initially two sets are stubborn: $\{a\}$ and $\{a, u, \tau_2\}$. If $\{a\}$ is chosen, then the LTS

is obtained that is shown with solid arrows on the right in Fig. 5. The full LTS also contains the dashed arrows. The full LTS has the tree failure $(\varepsilon, \{aa\})$ that the reduced LTS lacks.

7 Example

Figure 6 shows the example system used in the measurements in this section. It is a variant of the alternating bit protocol [1]. Its purpose is to deliver data items from a sending client to a receiving client via unreliable channels that may lose messages at any time. There are two kinds of data items: N and Y. To avoid cluttering Fig. 6, the data items are not shown in it. In reality, instead of sen, there are the actions senN and senY, and similarly with rec, d_0, $\bar{\mathsf{d}}_0$, d_1, and $\bar{\mathsf{d}}_1$.

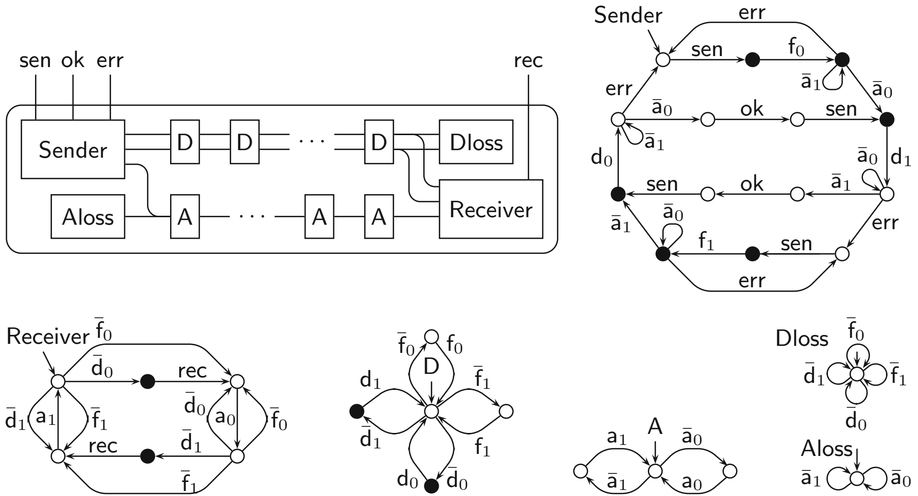

Fig. 6. The example system: architecture, Sender, Receiver, D, A, Dloss, and Aloss. Each sen, rec, d_0, d_1, $\bar{\mathsf{d}}_0$, and $\bar{\mathsf{d}}_1$ carries a parameter that is either N or Y. Each black state corresponds to two states, one for each parameter value. Each $\bar{x}$ synchronizes with x along a line in the architecture picture. The output of the rightmost D is consumed either by Receiver or Dloss, and similarly with the leftmost A

Because messages may be lost in the data channel, the alternating bit protocol has a timeout mechanism. For each message that it receives, the receiver sends back an acknowledgement message. After sending any message, the sender waits for the acknowledgement for a while. If it does not arrive in time, then the sender re-sends the message. To prevent the sender and receiver from being fooled by outdated messages, the messages carry a number that is 0 or 1.

The alternating bit protocol is impractical in that if either channel is totally broken, then the sender sends without limit in vain, so the protocol diverges.

The variant in Fig. 6 avoids this problem. For each sen action, Sender tries sending at most a fixed number of times, which we denote with ℓ. (In the figure, $\ell = 1$ for simplicity.) The protocol is expected to behave as follows. For each sen, it eventually replies with ok or err. If it replies with ok, it has delivered the data item with rec. If it replies with err, delivery is possible but not guaranteed, and it may occur before or after the err. There are no unsolicited or double deliveries. If the channels are not totally broken, the protocol cannot lose the ability to reply with ok.

After err, Sender does not know whether any data message got through and therefore it does not know which bit value Receiver expects next. For this reason, initially and after each err, the protocol performs a connection phase before attempting to deliver data items. It consists of sending a flush message and expecting an acknowledgement with the same bit value. When the acknowledgement comes, it is certain that there are no remnant messages with the opposite bit value in the system, so the use of that value for the next data message is safe. This is true despite the fact that the acknowledgement with the expected bit value may itself be a remnant message.

Assume that neither channel can lose infinitely many messages in a row. This is a typical fairness assumption. It guarantees that if there are infinitely many sen-actions, then infinitely many flush actions go through and infinitely many acknowledgements come back. However, it does not guarantee that any data message ever gets through. To guarantee that, it is necessary to further assume that if the acknowledgement channel delivers infinitely many messages, then eventually the data channel delivers at least one of the next ℓ messages that have been sent via it after the acknowledgement channel delivered a message. This assumption is very unnatural, because it says that the channels must somehow coordinate the losses of messages.

As a consequence, the traditional approach of proving liveness that is based on fairness assumptions is not appropriate for this protocol. On the other hand, fair testing equivalence can be used to prove a weaker but nevertheless useful property: the protocol cannot lose the ability to deliver data items and reply ok. This is why the protocol was chosen for the experiments in this section.

To implement Steps 1 and 3 in Sect. 4, we add to Sender a new state s_d and a transition labelled with t to s_d from each state that has an outgoing transition labelled with senN or senY. Let the resulting LTS be called Sender$'$. Clearly Sender = (Sender$'$ || Block_t)$\backslash\{$t$\}$, where Block_t is the single-state LTS whose alphabet is $\{$t$\}$ and that has no transitions. After computing the reduced LTS L'_r using Sender$'$ and treating t as visible, the final result is obtained as $(L'_\mathsf{r}$ || Block_t)$\backslash\{$t$\}$, which is trivial to compute from L'_r. This is correct, because fair testing equivalence is a congruence.

Table 1 shows analysis results obtained with the ASSET tool [18,19]. ASSET does not input parallel compositions of LTSs, but it allows to mimic their behaviour with C++ code. It also allows to express the "$\leadsto$" relation in C++ and computes stubborn sets with the esc algorithm. Thus it can be used to compute Step 2 of Sect. 4. ASSET verified that each LTS with the t-transitions is

Table 1. Each channel consists of c separate cells. Times are in seconds

c	Full LTS			Full, with t-transitions			Stubborn sets		
	States	Edges	Time	States	Edges	Time	States	Edges	Time
1	380	1068	0.0	440	1254	0.1	372	700	0.0
2	1880	6212	0.0	2224	7360	0.1	1234	1992	0.0
3	9200	34934	0.1	10976	41560	0.1	2986	4382	0.0
4	44000	188710	0.2	52672	224928	0.3	6104	8360	0.1
5	205760	983614	0.5	246656	1173536	0.6	11140	14494	0.1
6	944000	4977246	2.3	1132288	5941760	2.7	18726	23432	0.2
7	4263680	24582270	11.4	5115392	29357952	13.8	29578	35906	0.2
8	19013120	119011454	63.4	22813696	142177792	77.2	44496	52732	0.3
10							90150	103124	0.4
20							946520	1005784	3.6
30							4083190	4238144	18.8
40							11854160	12170204	68.2

Table 2. Each channel is a single reduced LTS

c	Full LTS			Full, with t-transitions			Stubborn sets		
	States	Edges	Time	States	Edges	Time	States	Edges	Time
10	42680	183912	0.3	51128	216300	0.4	16818	29756	0.2
20	287280	1278742	2.1	344568	1502900	2.4	84928	144116	0.7
30	913880	4112572	9.0	1096408	4831900	10.7	236438	391276	2.0
40	2102480	9513402	25.9	2522648	11175300	30.6	503348	819236	4.6
50	4033080	18309232	60.4	4839288	21505100	71.9	917658	1475996	9.3
60							1511368	2409556	17.7
70							2316478	3667916	29.1
80							3364988	5299076	45.9
90							4688898	7351036	70.3
100							6320208	9871796	102.8

indeed always may-terminating. To gain confidence that the modelling with C++ is correct, additional runs were conducted where the ASSET model contained machinery that verified most of the correctness properties listed above, including that the protocol cannot lose the ability to execute ok (except by executing t).

Table 1 shows spectacular reduction results, but one may argue that the model of the channels in Fig. 6 is unduly favourable to stubborn sets. The messages travel through the channels step by step. Without stubborn sets, any combination of empty and full channel slots may be reached, creating an exponential number of states. If a message is ready to move from a cell to the next one, then the corresponding action constitutes a singleton stubborn set.

Therefore, the stubborn set method has the tendency to quickly move messages to the front of the channel, dramatically reducing the number of constructed states.

To not give stubborn sets unfair advantage, another series of experiments was made where the messages are always immediately moved as close to the front of the channel as possible during construction of the full LTS. The fact about fifo queues and the "$\rightsquigarrow$" relation that was mentioned in Sect. 4 is also exploited. The results are shown in Table 2. Although they are less spectacular, they, too, show great benefit by the stubborn set method.

Acknowledgements. We thank the anonymous reviewers for their comments. Unfortunately, space constraints prevented us from implementing some of these.

References

1. Bartlett, K.A., Scantlebury, R.A., Wilkinson, P.T.: A note on reliable full-duplex transmission over half-duplex links. Commun. ACM **12**(5), 260–261 (1969)
2. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking, p. 314. MIT Press, Cambridge (1999)
3. Evangelista, S., Pajault, C.: Solving the ignoring problem for partial order reduction. Softw. Tools Technol. Transf. **12**(2), 155–170 (2010)
4. Eve, J., Kurki-Suonio, R.: On computing the transitive closure of a relation. Acta Inform. **8**(4), 303–314 (1977)
5. Gerth, R., Kuiper, R., Peled, D., Penczek, W.: A partial order approach to branching time logic model checking. In: Proceedings of Third Israel Symposium on the Theory of Computing and Systems, pp. 130–139. IEEE (1995)
6. Godefroid, P.: Using partial orders to improve automatic verification methods. In: Proceedings of CAV 1990, AMS-ACM DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 3, pp. 321–340 (1991)
7. Godefroid, P. (ed.): Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. LNCS, vol. 1032. Springer, Heidelberg (1996)
8. Laarman, A., Pater, E., van de Pol, J., Hansen, H.: Guard-based partial-order reduction. Softw. Tools Technol. Transf. 1–22 (2014)
9. Mazurkiewicz, A.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) Petri Nets 1986. LNCS, vol. 255, pp. 279–324. Springer, Heidelberg (1987)
10. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
11. Peled, D.: Partial order reduction: linear and branching temporal logics and process algebras. In: Proceedings of POMIV 1996, Workshop on Partial Order Methods in Verification, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29, pp. 233–257. American Mathematical Society (1997)
12. Rensink, A., Vogler, W.: Fair testing. Inf. Comput. **205**(2), 125–198 (2007)
13. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
14. Valmari, A.: Error Detection by reduced reachability graph generation. In: Proceedings of the 9th European Workshop on Application and Theory of Petri Nets, pp. 95–122 (1988)

15. Valmari, A.: State space generation: efficiency and practicality. Dr. Techn. thesis, Tampere University of Technology Publications 55, Tampere (1988)
16. Valmari, A.: Stubborn set methods for process algebras. Peled, D.A., Pratt, V.R., Holzmann, G.J. (eds.) Partial Order Methods in Verification: DIMACS Workshop, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 29, pp. 213–231. American Mathematical Society (1997)
17. Valmari, A.: The state explosion problem. In: Reisig, W., Rozenberg, G. (eds.) APN 1998. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998)
18. Valmari, A.: Stop it, and be stubborn! In: Haar, S., Meyer, R. (eds.) 15th International Conference on Application of Concurrency to System Design, pp. 10–19. IEEE Computer Society (2015). doi:10.1109/ACSD.2015.14
19. Valmari, A.: A state space tool for concurrent system models expressed in C++. In: Nummenmaa, J., Sievi-Korte, O., Mäkinen, E. (eds.) SPLST 2015 Symposium on Programming Languages and Software Tools, vol. 1525, pp. 91–105. CEUR Workshop Proceedings (2015)
20. Valmari, A., Hansen, H.: Can stubborn sets be optimal? Fundamenta Informaticae **113**(3–4), 377–397 (2011)