Fair testing and stubborn sets

Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Fair Testing and Stubborn Sets

Antti Valmari[1], **Walter Vogler**[2]

[1] Department of Mathematics, Tampere University of Technology
P.O. Box 553, FI–33101 Tampere, FINLAND
e-mail: antti.valmari@tut.fi

[2] Institut für Informatik, University of Augsburg
D-86135 Augsburg, GERMANY
e-mail: walter.vogler@informatik.uni-augsburg.de

The date of receipt and acceptance will be inserted by the editor

Abstract. Partial order methods alleviate state explosion by considering only a subset of actions 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*-preserving 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 or not the system may enter a livelock that it cannot leave. We prove that a method that was designed for trace equivalence also preserves fair testing equivalence. We 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.

Key words: partial order methods; stubborn sets; fairness; progress; fair testing equivalence

1 Introduction

State spaces of systems that consist of many parallel components are often huge. Typically many states arise from executing concurrent actions in different orders. The methods discussed in, e.g., [3,5,9–11,13,19,20,25–29,33–37], 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 actions 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.

The differences between stubborn, ample, and persistent set methods were discussed in detail in [37]. We call them *aps set methods*. They are usually called *partial order methods*, but this phrase is also often used in a significantly wider sense, causing ambiguity. The literature on partial order methods is extensive, so it is not possible to provide a survey here. Let us, however, mention two classes of methods that are also called partial order but are fundamentally different from aps set methods.

The basic form of *dynamic partial order methods* (e.g., [1,7,12,22]) is restricted to acyclic systems, but cyclic systems can be dealt with to some extent with trickery. These methods investigate a sufficient collection of executions by firing one transition at a time. They do not compute aps sets. Instead, they recognize potential choice or conflict situations afterwards and, if necessary, later backtrack to investigate the opposite choice. When this works well, most states are encountered only once. This facilitates "stateless" model checking, meaning that states that are not along the current execution path need not be kept in memory. It makes it possible to analyse bigger systems than aps set methods do, but also runs the risk of extremely long analysis times if, against the expectation, same states are encountered repeatedly.

Unfolding methods (e.g., [4,17,22]) are based on building a partial order data structure that represents the behaviour of the system in a true concurrency fashion. Individual states are not represented explicitly. Instead, information on each individual state is distributed over many nodes of the structure. Again, difficulties arise when applying the idea to cyclic systems. Cyclic systems can be handled, but at a potentially high cost.

Aps set, dynamic partial order, and unfolding methods each have their distinct advantages and disadvantages. In the present publication, cyclic behaviour and explicit representation of states are central, making it hard to see how dynamic partial order or unfolding methods could apply.

The phrase "partial order" refers to the intuition that if two executions only differ in the order in which two concurrent actions occur, then they are linearizations of the same, more abstract concurrent execution that does not specify the order. Unfolding methods represent these abstract executions rather directly, while aps set and dynamic partial order methods are often described as trying to represent only one linearization for each abstract execution. This intuition works well with executions that lead to a deadlock, that is, to a state that has no outgoing transitions.

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 occurrences of invisible actions and contain additional occurrences of invisible actions compared to executions that it represents. With branching time properties, thinking in terms of executions is insufficient to start with. Therefore, most aps set methods would better not be called partial order methods, because partial order intuition leads to subtly but dangerously wrong expectations on how the methods work, and thus hampers understanding them.

The more properties a method preserves, the worse are the reduction results that it yields. 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) [25], while at the other end the CTL* logic (excluding the next state operator) and divergence-sensitive branching bisimilarity are preserved [9, 20, 29].

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 progress treat the cycle as a violation against progress. However, this is not always the intention. Therefore, so-called *fairness assumptions* [15] 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 [3]. Furthermore, as pointed out in [5], the most widely used condition for guaranteeing the preservation of linear-time progress (see, e.g., [3, p. 155]) often works in a way that is detrimental to reduction results.

Fair testing equivalence [21], which goes back to Section 3.3.2 in [40], 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 instance, Section 3 presents a system where suitable fairness assumptions are difficult to formulate, but fair testing works well.

Fair testing equivalence implies trace equivalence. As a consequence, it cannot have better reduction methods than trace equivalence. Fair testing equivalence is a branching time notion. So 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 more than 20 years old trace-preserving stubborn set method [27, 29] also preserves fair testing equivalence. This is the first main result of the present study. It means that *no reduction power is lost* compared to trace equivalence.

Most conditions in aps set methods can be enforced with algorithms that only look at the current state of the system under analysis. However, many methods also have a condition that looks at all states in a cycle or terminal strong component of the reduced LTS. Without the additional condition, if some part of the system can execute a cycle without the rest of the system participating, the method may incorrectly terminate after investigating only the behaviour of that part. This is known as the *ignoring problem*, because the method ignores the behaviour of the rest of the system.

Conditions that solve the ignoring problem are particularly difficult to enforce without significant loss of reduction power. In the present study, new observations and theorems lead to the conclusion that when preserving the trace (and fair testing) equivalence, in most cases, the ignoring problem need not be solved at all! This is our second main result.

General background concepts are introduced in Section 2. Section 3 discusses two telecommunication protocols and fairness in their verification. Fair testing equivalence is defined in Section 4. Section 5 introduces stubborn sets. In Section 6 we prove that the trace-preserving stubborn set method also applies to fair testing equivalence, and a modified method to a modified equivalence. Implementation of stubborn sets excluding solutions to the ignoring problem is discussed in Section 7. This material makes this publication self-contained. Section 8 discusses further the ignoring problem and why it is good to avoid strong conditions. The new results on the ignoring problem are presented in Section 9. Section 10 shows some performance measurements, after which a conclusions section ends the study.

A much shorter version of this study appeared as [39]. Compared to it, the novel results on the ignoring problem are new; the method that preserves tree failure equivalence is new; and the discussion of many issues has been made much more extensive, with many small illustrative examples. The old solution to the ignoring problem has been removed.

2 Labelled Transition Systems

In this section we first define labelled transition systems and two operators for composing systems from them. We also define some useful notation. Then we define the well-known trace equivalence and stable failures equivalence, and discuss why the failures must be stable.

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.

In drawings, states are represented as circles, and transitions as labelled arrows from a state to a state. The initial state is distinguished with an arrow whose tail is not adjacent to any state. If the alphabet is not given explicitly, it consists of the labels of transitions other than τ. Figure 1 later in this section shows four LTSs. The alphabets of the first two and last two are $\{a, b\}$ and $\{b\}$, respectively.

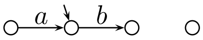
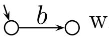
We use the convention that, unless otherwise stated, $L' = (S', \Sigma', \Delta', \hat{s}')$, $L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)$, and so on.

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 $1 \leq i \leq n$. For instance, $\hat{s} -\tau aaab\rightarrow s_4$ in the L in Figure 1. The notation $s -a_1 \cdots a_n\rightarrow$ denotes that there is some 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 L in Figure 1 has $\mathsf{en}(\hat{s}) = \{\tau\}$, $\mathsf{en}(s_2) = \{a, b\}$, $\mathsf{en}(s_3) = \{a\}$, and $\mathsf{en}(s_4) = \emptyset$.

The *reachable part* of L is 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
- $\Delta' \ = \ \{(s, a, s') \in \Delta \mid s \in S'\}$.

For instance, the reachable part of  is  with the alphabet $\{a, b\}$.

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 $L_1 \,||\, \cdots \,||\, L_m$ is well-defined for any positive integer m. Figure 7 in Section 5 shows an example of a parallel composition. For the time being, please read τ_1 and τ_2 as τ in it.

The *hiding* of an action set A in L is denoted with $L \setminus A$. It is $L \setminus A = (S, \Sigma', \Delta', \hat{s})$, where $\Sigma' = \Sigma \setminus A$ 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. Figure 1 shows two examples.

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 $s -a_1 \cdots a_n\rightarrow s'$, of the sequence $a_1 \cdots a_n$, and of the state s. The notation $s =\sigma\Rightarrow$ denotes that there is s' such that $s =\sigma\Rightarrow s'$. The set of traces of L is the set of traces of its initial state, that is,

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

In Figure 1 we have $\mathsf{Tr}(L) = \mathsf{Tr}(L') = \{\varepsilon, a, aa, \ldots, b, ab, aab, \ldots\}$ and $\mathsf{Tr}(L \setminus \{a\}) = \mathsf{Tr}(L' \setminus \{a\}) = \{\varepsilon, b\}$.

The LTSs L_1 and L_2 are *trace equivalent* if and only if $\Sigma_1 = \Sigma_2$ and $\mathsf{Tr}(L_1) = \mathsf{Tr}(L_2)$. The first two LTSs in Figure 1 are trace equivalent, and so are the second two.

An equivalence "$\approx$" is a *congruence* with respect to an operator f if and only if for every L and L', $L \approx L'$ implies $f(L) \approx f(L')$. Trace equivalence is a congruence with respect to "$||$", "$\setminus$", and all other widely used LTS operators. Please see [30] for a proof that all operators in a very wide class can be constructed from " $||$ " and "$\setminus$" up to trace equivalence. Therefore, trace equivalence is a congruence with respect to this class of operators.

A state s is *stable* if and only if $\neg(s -\tau\rightarrow)$. Let $A \subseteq \Sigma$. A state s *refuses* A if and only if $\neg(s =a\Rightarrow)$ for every $a \in A$. A *stable failure* of an LTS L is a pair (σ, A) where $\sigma \in \Sigma^*$ and $A \subseteq \Sigma$ such that there is some stable s that refuses A such that $\hat{s} =\sigma\Rightarrow s$; A is called a *refusal set.* It follows from the definition that if (σ, A) is a stable failure of L, then $\sigma \in \mathsf{Tr}(L)$. Furthermore, σ leads to a deadlock if and only if (σ, Σ) is a stable failure of L. The stable failures of the L and L' in Figure 1 are (a^n, A) and $(a^n b, B)$ where $n \in \mathbb{N}$, $A \subseteq \{b\}$, and $B \subseteq \{a, b\}$. The stable failures of $L \setminus \{a\}$ and $L' \setminus \{a\}$ are $(b, \emptyset)$ and $(b, \{b\})$.

Two LTSs are *stable failure equivalent* if and only if they have the same alphabets and the same stable failures. Stable failure equivalence is a congruence with respect to " $||$ " and "$\setminus$". It is not a congruence with respect to the widely used "choice" operator and the more rare "interrupt" operator, but the equivalence that compares the alphabets, the traces, the stable failures,

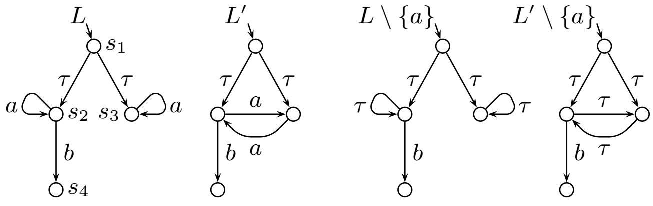


Fig. 1. Ordinary failures suffer from a congruence problem

and the *initial stability* is a congruence with respect to both (L is initially stable if and only if $\hat{s}$ is stable).

Around 1990, some attention was paid to *(ordinary) failures*, defined otherwise like stable failures but not requiring that s is stable. Every stable failure is a failure, but not necessarily vice versa. If (σ, A) is a failure but not a stable failure, then σ is a *divergence trace*, that is, there is s such that $\hat{s} =\!\sigma\!\Rightarrow s -\!\tau\tau\cdots\!\rightarrow$. For instance, $(\varepsilon, \{b\})$ is an ordinary but not a stable failure of the $L \setminus \{a\}$ in Figure 1.

What is more, $(\varepsilon, \{b\})$ is not a failure of $L' \setminus \{a\}$. On the other hand, L and L' have the same alphabet, the same traces, the same failures, and the same divergence traces, and both are initially unstable. This means that unlike stable failures, ordinary failures do not yield a congruence, not even if aided by traces, etc. This is why stable failures became more popular.

Let us relate these failure concepts to the two most well known process-algebraic semantic models. The failures in the failures–divergences equivalence of CSP [23] are neither ordinary nor stable. Instead, they are the union of stable failures with all pairs $(\sigma, A) \in \Sigma^* \times 2^\Sigma$ such that some prefix of σ is a divergence trace. This implies that failures–divergences equivalence does not preserve any information on the behaviour of the system beyond minimal divergence traces, a phenomenon sometimes called *catastrophic divergence*. To see beyond divergence, the equivalence that compares the traces and stable failures was later added to the CSP theory.

Milner's observation equivalence [18] preserves failures and is a congruence with respect to " || " and "\". However, it is too strong for many verification purposes in the sense that it preserves much more information than needed. Because of its strength, although an aps set reduction method for it has been found [29], it is much less powerful than the methods in the present study.

In Section 4 we will find the weakest congruence that preserves ordinary failures. Before that we introduce the example system used in the experiments of this study, because it can be used to illustrate the benefits of the congruence.

3 Self-Synchronizing Alternating Bit Protocol

In this section we introduce the self-synchronizing alternating bit protocol used in the experiments in Section 10.

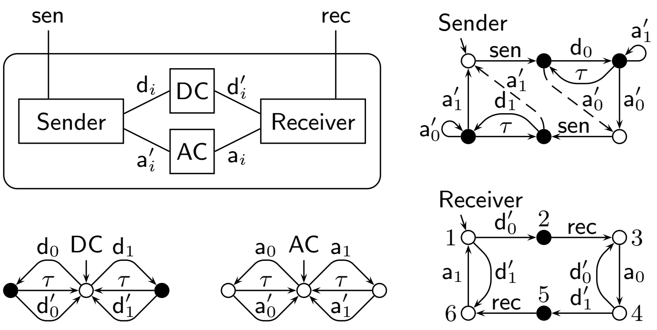


Fig. 2. The architecture and LTSs of the alternating bit protocol

We first introduce the famous original alternating bit protocol, from which our protocol has been developed. In mainstream verification, it is customary to use so-called fairness assumptions [15] to ensure that a system eventually provides service. In this publication we adopt a different approach to fairness. To motivate this decision, we discuss the use of the mainstream fairness assumptions on the original and self-synchronizing alternating bit protocols, pointing out problems.

Figure 2 shows the *alternating bit protocol* [2] modelled as a system of four LTSs. Its architecture is

$$(\mathsf{Sender}||\mathsf{DC}||\mathsf{AC}||\mathsf{Receiver}) \setminus \{\mathsf{d}_0, \mathsf{d}'_0, \mathsf{d}_1, \mathsf{d}'_1, \mathsf{a}_0, \mathsf{a}'_0, \mathsf{a}_1, \mathsf{a}'_1\} .$$

The actions sen, rec, d_i, and d'_i carry a data value that is not shown in the model. The black states and dashed transitions will be explained later.

The purpose of the protocol is to provide reliable data transmission from the sending site to the receiving site despite the fact that the communication channels between the sites may lose messages. A data value is given for transmission via the action sen. Sender composes a message consisting of the data value and an *alternating bit* that is either 0 or 1. Accordingly, the message is modelled with d_0 or d_1. Sender sends it to Receiver via DC, which models a data channel. DC either loses the message by executing τ, or delivers it to Receiver by executing d'_0 or d'_1. (Please remember that τ-transitions by component LTSs of a parallel composition are not synchronized. So Sender and AC do not participate in the τ-transition of DC.)

If the message has the alternating bit value that Receiver expects, then Receiver delivers its data content via rec, sends an acknowledgement with the same bit value, and then changes its expected bit value. If the message has the wrong bit value, Receiver only sends an acknowledgement. Analogously, the acknowledgement channel AC either loses the acknowledgement or delivers it to Sender.

When Sender receives an acknowledgement with the correct alternating bit value, it changes its own bit value and becomes ready for the transmission of the next data value. If such an acknowledgement does not arrive in time, Sender executes a τ-transition (this is called *timeout*) and re-sends the data message. The length of the

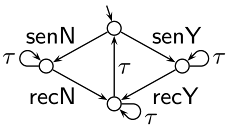


Fig. 3. The service provided by the alternating bit protocol

waiting time is not modelled. If an acknowledgement with the wrong bit value arrives, Sender just consumes it.

In the figure, we have abstracted from the data values. In the model, each black state has a copy for each possible value of the data so that, in case of a re-send, the same value is sent.

If the data message gets through but the acknowledgement is lost or delayed, Sender re-sends a data value that has already been delivered. Thanks to the alternating bit, Receiver recognizes this situation and does not re-deliver the data value. The alternating bit in the acknowledgements prevents Sender from mis-interpreting a delayed acknowledgement of an earlier data message as an acknowledgement of the most recent data message.

Figure 3 shows an LTS that has the same traces, stable failures, and divergence traces as the system in Figure 2, assuming that there are two different data values, N and Y. Altogether, the only way how the protocol may fail is that Sender re-sends the same data message forever. This may happen because of repeated losses of data messages or acknowledgements, or because Sender repeatedly chooses a τ-transition although an a'_i-transition with the correct i is or will become enabled. If no data message gets through, this is seen as a τ-self-loop before the recN- or recY-transition in the figure, and otherwise as the third τ-self-loop. When Sender receives a correct acknowledgement, it becomes ready for a new sen-action. While waiting for it, the protocol may consume the data messages and acknowledgements that are still in transit, but from then on it does nothing until the next sen-action.

Often it is reasonable to assume that although the channels may lose messages, they are not totally broken, and although pre-mature timeouts are possible, Sender is not always too fast. In mainstream verification, these assumptions are modelled with *fairness assumptions* [15]. Full discussion would require more (or different) information on transitions than the labels in the LTS formalism provide. Therefore, we describe the idea in a slightly simplified form that is not technically correct in general, but works in the case of Figure 2.

Fairness is easiest to understand via its negation, unfairness. An infinite execution $\hat{s} \; {-a_1\!\!\rightarrow} \; s_1 \; {-a_2\!\!\rightarrow} \; \ldots$ is *weakly unfair* towards an action a, if and only if there is $n \in \mathbb{N}$ such that when $i \geq n$, $s_i \; {-a\!\!\rightarrow}$ but $a_i \neq a$. That is, from some point on, a is continuously enabled but does not occur. The execution is *strongly unfair* towards a, if and only if there is n such that $s_i \; {-a\!\!\rightarrow}$ for infinitely many i, but $a_i \neq a$ when $i \geq n$. That is, from some point on, a is repeatedly enabled but does not occur. Strong fairness towards a implies weak fairness towards a. In applications, two sets F_w and F_s of actions are chosen, and it is assumed that all executions of the modelled system are weakly fair towards every element of F_w and strongly fair towards every element of F_s. As a consequence, unfair executions of the model are considered to not correspond to "real" executions of the modelled system and thus are not considered as valid counter-examples to a property.

In the case of Figure 2, let $F_\mathsf{w} = F_\mathsf{s} = \{\mathsf{sen}, \mathsf{rec}, \mathsf{d}_0, \ldots, \mathsf{a}'_1\}$. This models the informal assumption that, for instance, although messages may be lost at any time, they are not systematically lost. We now argue that in the presence of the dashed transitions in Figure 2, if an execution contains never-ending re-sendings, then it violates our fairness assumption. This means that it is not considered to correspond to a "real" execution. So in any "real" execution, any message that is sent is eventually delivered.

Assume that at some stage, Sender runs around the d_0-τ-cycle without executing a'_0. Executions of a'_1 may but need not occur. For simplicity, we assume that Receiver is in its initial state (number 1 in the figure); similar arguments apply if it is in some other state. Strong fairness towards d'_0 prevents indefinite losses in DC and forces Receiver to move to state 2. Weak fairness towards rec forces Receiver to continue to state 3. If AC is not in its initial state, strong fairness towards a'_0 and a'_1 guarantees that it eventually moves there, by executing either a'_0 (which we assumed not to happen), a'_1, or τ. (When fairness towards x forces an LTS to leave a state, it does not necessarily mean that the state must be left via an x-transition.) Weak fairness now guarantees that Receiver executes a_0, entering state 4. Then strong fairness towards d'_0 forces Receiver to move again. It returns to state 3, since d'_1 is not available. Thus, a_0 is repeated infinitely many times. As a consequence, a'_0 is enabled infinitely often but not executed, so the execution violates strong fairness.

We now argue that if certain apparently innocent modifications are made to Sender, then our fairness assumption no longer rules out all never-ending re-sendings and thus no longer guarantees that any message that is sent is eventually delivered. Let us use subscripts to indicate the LTS whose τ-transition is executed. First, if the dashed transitions are removed, then the cycle $\mathsf{d}_0\tau_\mathsf{Sender}\mathsf{d}'_0\mathsf{a}_0\tau_\mathsf{AC}$ (where Receiver is in state 4 at the beginning) is fair, because a'_0 is never enabled in it. Second, if the dashed transitions are kept but the solid vertical a'_0- and a'_1-transitions are removed, then the cycle $\mathsf{d}_0\mathsf{d}'_0\mathsf{a}_0\tau_\mathsf{AC}\tau_\mathsf{Sender}$ is fair. So, despite the use of fairness assumptions, these two versions may degenerate to never-ending re-sendings. The last version is observation equivalent [18] to the original version. This illustrates that fairness is not necessarily preserved dur-

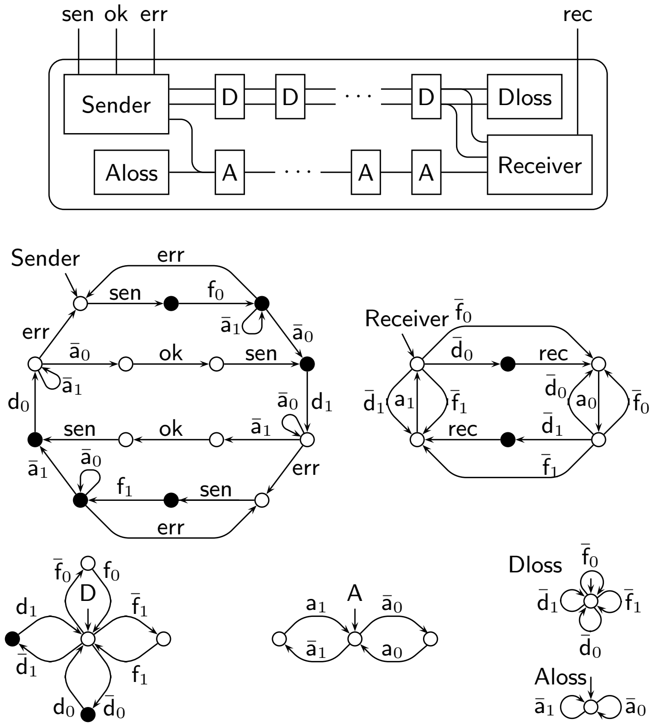


Fig. 4. The example system: architecture, Sender, Receiver, D, A, Dloss, and Aloss. Each sen, rec, d_0, d_1, $\bar{\mathsf{d}}_0$, and $\bar{\mathsf{d}}_1$ carries a parameter that is either N or Y. Each black state corresponds to two states, one for each parameter value. Each $\bar{x}$ synchronizes with x along a line in the architecture picture as explained in the main text. The output of the rightmost D is consumed either by Receiver or Dloss, and similarly with the leftmost A

ing equivalence-preserving reductions, which is a major problem for process-algebraic verification methods.

This example also illustrates that to verify progress properties, the modeller may have to put in details that do not arise naturally from the original system description, such as the dashed transitions in Figure 2. The error scenario in the absence of the dashed transitions and presence of fairness is that always when the acknowledgement is in AC, Sender happens to be in the tail state of the d_0-transition at least until the acknowledgement is lost. Because τ_{Sender} models timeout, an eternally re-sending Sender spends much more time in the head than the tail state of the d_0-transition. This makes the error scenario unrealistic in practice, so one would want a better fairness assumption.

In real life, it is usually unacceptable that a protocol may degenerate to never-ending re-sendings. Instead, the protocol should eventually give up and inform the user about the error situation. It should also be able to recover if the channel is later fixed. In the remainder of this section we design such a protocol by modifying the alternating bit protocol. It is called the *self-synchronizing alternating bit protocol*. To our best knowledge, it was first presented in [38]. It will be used to illustrate one more problem related to traditional fairness. It will also be used in the experiments in Section 10.

The self-synchronizing protocol is shown in Figure 4. To get an infinite family of systems with bigger and bigger LTSs facilitating an interesting series of reduction measurements, we have replaced DC and AC by channels which can store more than one message. The data channel consists of some constant number of copies of *cells* D, together with Dloss. Each cell can hold one message, which it reliably delivers further. A message delivered by the rightmost D is consumed either by Receiver or by Dloss. The latter models the loss of a data message. The acknowledgement channel is modelled in a similar fashion.

The system could be modelled by labelling the actions between Receiver and the first A-cell with a_0^0 and a_1^0, the actions between the first and second A-cell with a_0^1 and a_1^1, and so on, and similarly with the data channel. To avoid the resulting notational complexity, we use overbar notation instead. Excluding sen, ok, err, and rec, synchronization occurs between x in one LTS and $\bar{x}$ in another LTS connected to the former LTS in the architecture picture. For instance, d_0 from Sender synchronizes with $\bar{\mathsf{d}}_0$ from the first D-cell.

We have added two actions to Sender, ok and err. They indicate that it is or is not certain that the message got through. Sender has an upper bound to how many times it tries to send each data message. We denote it with ℓ. To not make Figure 4 too complicated, $\ell = 1$ in it. Sender executes ok after getting an acknowledgement with the correct alternating bit value, and err after getting a timeout after the maximum number of sending attempts.

Assume that Sender has just executed err. If the data message was lost, then Receiver did not change its alternating bit value, but if it did get through (and the acknowledgement was lost or delayed), Receiver did change it. So Sender cannot know which alternating bit value to use for the next data message, so that Receiver will not reject it as a re-send of the previous message.

Because of this, after receiving the next sending request after err, Sender sends a *flush message*, waits for an acknowledgement for it, and only then sends the data message. Receiver acknowledges the flush message without executing rec. When the acknowledgement comes to Sender, 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.

Timeouts and re-sending also apply to flush messages. If ℓ flushing attempts have been made in vain, Sender executes err again. To make the protocol more fault-tolerant, the flush mechanism is used also in connection with the first sending request.

There are two kinds of data items: N and Y. Each black state in Figure 4 exists in two copies, one holding

N and another holding Y. Beyond this, the data items are not shown in the figure, to avoid cluttering 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 $\bar{\mathsf{d}}_1$.

The protocol is expected to provide the following service. 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.

Let us apply typical fairness assumptions to this system. In particular, we assume that neither channel can lose infinitely many messages in a row. These guarantee that if there are infinitely many sen-actions, then infinitely many flush or data messages go through, and infinitely many acknowledgements come back. To avoid the problem that we had with the original alternating bit protocol, let us further assume that Sender actually reads infinitely many of these acknowledgements.

Even these do not guarantee that any data item is ever delivered. This is because they do not prevent the data channel from passing every flush message and losing every data message.

Let us make the stronger assumption that the channels are fair to each kind of messages separately. That is, if both infinitely many flush messages and infinitely many data messages are sent, then both infinitely many flush messages and infinitely many data messages pass through the data channel. Then infinitely many sen-actions do guarantee infinitely many rec-actions. Unfortunately, they still do not guarantee any ok-actions. This is because the flush and data messages use the same ack-messages, so AC has the permission to work such that it passes acknowledgements only when Sender has most recently sent a flush message.

It might be that by introducing separate kinds of acknowledgements for flush and data messages, fairness assumptions could be made to work also for ok. However, this would mean making a non-trivial change to the protocol.

As a consequence, the traditional approach of proving progress that is based on fairness assumptions is difficult to use with the alternating bit protocol, and is not appropriate for the self-synhcronizing protocol. In the next section we introduce an LTS equivalence that offers an alternative approach to fairness that is free from this problem.

4 Fair Testing Equivalence

In this section we define the fair testing equivalence of [21] and discuss its intuition and properties that make it useful for verifying progress properties. As a preliminary step we define and discuss tree failure equivalence, which is a strictly stronger equivalence with a related but much simpler definition.

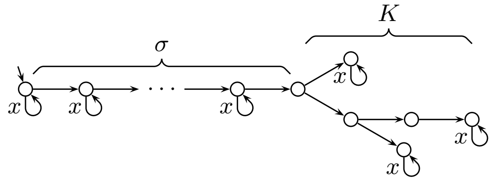


Fig. 5. The LTS L_σ^K

In Section 2, the failures of an LTS L were defined as the pairs (σ, A) where σ is a trace and A is a set of visible actions such that L can execute σ and then refuse A. Refusing A means that L is in a state where it cannot execute any member of A, not even if L is allowed to execute zero or more τ-transitions before the member of A. Not necessarily all possible ways in which L can execute σ lead to the refusal of A, but at least one does. Now we replace A by a set K of non-empty strings of visible actions, obtaining a more general notion of refusal. Refusal of K means that L can execute no member κ of K to completion, not even if L is allowed to also execute invisible actions. The empty string ε is ruled out from K because no state can refuse ε anyway.

Definition 1. Let L be an LTS, $K \subseteq \Sigma^+$, and $s \in S$. The state s *refuses* K if and only if for every $\kappa \in K$ we have $\neg(s =\kappa\Rightarrow)$. 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 $\mathsf{Tf}(L_1) = \mathsf{Tf}(L_2)$.

For example, $\hat{s}$ refuses K if and only if $K \cap \mathsf{Tr}(L) = \emptyset$. Ordinary failures are a special case of tree failures, with the length of each $\kappa \in K$ being one. We have $\sigma \in \mathsf{Tr}(L)$ if and only if $(\sigma, \emptyset) \in \mathsf{Tf}(L)$.

Although, when refusing K, L cannot execute any member κ of K to completion, L can execute ε and may be able to execute some non-empty proper prefixes of κ. To discuss such situations, we define that $\rho \in \Sigma^*$ is a *prefix of* K if and only if there is π such that $\rho\pi \in K$. We write $\rho^{-1}K$ for $\{\pi \mid \rho\pi \in K\}$. So ρ is a prefix of K if and only if $\rho^{-1}K \neq \emptyset$.

Tree failure equivalence preserves ordinary failures and is a congruence with respect to " || " and "\" [21]. However, it is not the weakest such congruence.

To see why this is the case, let us follow the line of thought in typical weakest congruence proofs. Take two LTSs L_1 and L_2 over Σ that are related by such congruence, and consider some $(\sigma, K) \in \mathsf{Tf}(L_1)$. We only discuss the case $K \neq \emptyset$. From this tree failure and some $x \notin \Sigma$, we can construct L_σ^K with alphabet $\Sigma \cup \{x\}$ as illustrated in Figure 5. This LTS consists of a σ-labelled path ending in z_σ, say, and a tree rooted at z_σ. This tree represents K. Formally, there is a state z_ρ for each prefix ρ of σ, and for each ρ of the form $\rho = \sigma\pi$, where π is a prefix of K. For each state $z_{\rho a}$ where $a \in \Sigma$, L_σ^K has the transition $(z_\rho, a, z_{\rho a})$. So $z_\varepsilon \; {-\rho\!\!\rightarrow} \; z_\rho$ holds for every $z_\rho \in S_\sigma^K$. There also is the x-self-loop (z_ρ, x, z_ρ) for every proper prefix of σ, and $(z_{\sigma\kappa}, x, z_{\sigma\kappa})$ for every $\kappa \in K$. The initial state is z_ε.

Since $(\sigma, K) \in \mathsf{Tf}(L_1)$, by executing σ, L_1 can go to a state s_1 from which it cannot execute any member of K to completion. If L_1 does this as a component of $(L_1 \,||\, L_\sigma^K) \setminus \Sigma$, the latter goes to the state $(s_1, z_\sigma) \in S_1 \times S_\sigma^K$. By the structure of L_σ^K, this state does not enable x. The same holds for every state that is reachable from it, because L_1 prevents the execution of any member of K to completion. This means that $(L_1 \,||\, L_\sigma^K) \setminus \Sigma$ has the (ordinary) failure $(\varepsilon, \{x\})$.

This must also hold for $(L_2 \,||\, L_\sigma^K) \setminus \Sigma$, by the choice of L_1 and L_2. It holds if and only if the system can go to a state $(s_2, z) \in S_2 \times S_\sigma^K$ from which it cannot continue to a state that enables x. Here z must be $z_{\sigma\rho}$ for some ρ that is a prefix (but not a member) of K, $\hat{s}_2 \; {=\sigma\rho\!\Rightarrow} \; s_2$, and s_2 must refuse $\rho^{-1}K$. That is, we cannot prove that $(\sigma, K) \in \mathsf{Tf}(L_2)$, we only can prove that $(\sigma\rho, \rho^{-1}K) \in \mathsf{Tf}(L_2)$ for some prefix ρ of K.

This observation motivates the following definition. The either-part in item 2 handles the case $K = \emptyset$, and overlaps with the or-part (with $\rho = \varepsilon$) when $K \neq \emptyset$.

Definition 2. Let L_1 and L_2 be LTSs. They 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.

Fair testing equivalence is a congruence with respect to "$||$", "$\setminus$", and some other operators, and the addition of the comparison of initial stability makes it a congruence also with respect to the choice operator. Our discussion above indicates that it is the weakest congruence that preserves ordinary failures. For the same reason, it is also the weakest congruence that preserves the claim "in all futures always, there is a future where eventually x occurs". Furthermore, 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. As a consequence, it cannot have better reduction methods than trace equivalence.

In [32] it was proven that trace equivalence is the only non-trivial alphabet-preserving congruence with respect to "$||$", "$\setminus$", and (functional) renaming that is strictly weaker than fair testing equivalence. By the trivial alphabet-preserving congruence we mean the one that unifies two LTSs if and only if they have the same alphabets. With also the choice operator, precisely six non-trivial alphabet-preserving and eight non-alphabet-preserving congruences are strictly weaker than fair testing equivalence with initial stability. Four of the six are rather artificial, one is the trace equivalence, and one unifies all unstable LTSs while comparing the initial visible actions of stable LTSs. The latter is the weakest alphabet-preserving congruence that preserves initial stability.

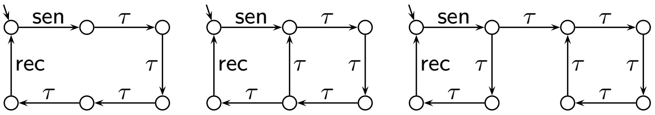


Fig. 6. A progressing, an intermediate, and a non-progressing LTS

Fair testing equivalence is insensitive to divergence in the sense that for every L, $L \,||\,$↘○↺$\,\tau$ is fair testing equivalent to L. This means that if only sen and rec are visible, it abstracts away from the infinite sequences of re-sending that caused problems in Section 3. On the other hand, if the protocol can enter a situation from which rec is no longer possible, fair testing equivalence reveals it as a refusal of $\{$rec$\}$. As a consequence, mainstream fairness assumptions are not needed to verify progress.

The notion of progress that can be verified with fair testing equivalence is strictly weaker than with mainstream fairness assumptions. Both notions treat the first LTS in Figure 6 as guaranteeing progress and the last LTS as not guaranteeing. Fair testing does and the mainstream notion does not treat the second LTS as yielding progress. This means that *if* the mainstream notion of progress is the real goal, then fair testing equivalence may fail to reveal some errors but never gives false alarms. From this perspective, it is thus not perfect, but better than nothing and easy to use, because there is no need to formulate fairness assumptions. On the other hand, sometimes the weaker guarantee of progress provided by fair testing equivalence is sufficient. On finite-state LTSs, it actually corresponds to strong fairness with respect to *transitions* instead of actions [21].

5 Stubborn Sets

Stubborn set methods for process algebras apply to LTS expressions of the form

$$L \;=\; (L_1 \,||\, \cdots \,||\, L_m) \setminus A \;.$$

To discuss them, 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
- $\bar{\Delta}_i \;=\; \{(s, a, s') \mid a \in \Sigma_i \wedge (s, a, s') \in \Delta_i\} \;\cup$ $\{(s, \tau_i, s') \mid (s, \tau, s') \in \Delta_i\}$.

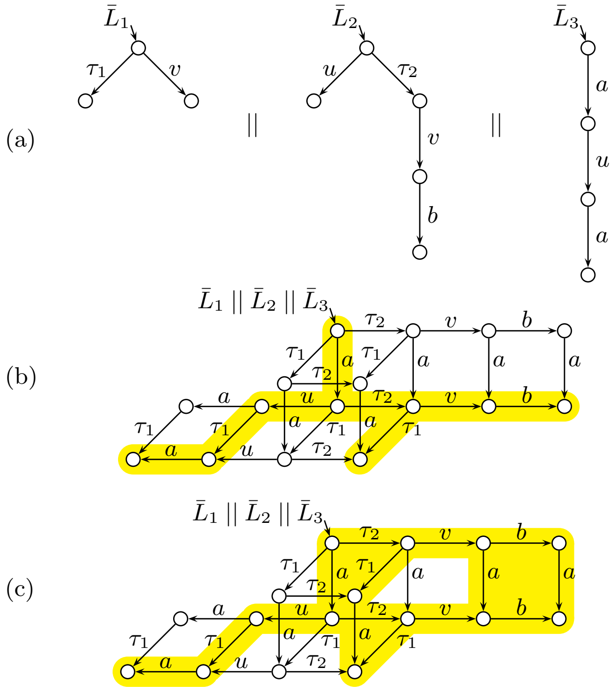

(a)
$\bar{L}_1$
τ_1
v
||
$\bar{L}_2$
u
τ_2
v
b
||
$\bar{L}_3$
a
u
a
(b)
$\bar{L}_1 \mid\mid \bar{L}_2 \mid\mid \bar{L}_3$
(c)
$\bar{L}_1 \mid\mid \bar{L}_2 \mid\mid \bar{L}_3$

Fig. 7. **(a)** A system with $V = \{a, b\}$ and $I = \{u, v, \tau_1, \tau_2\}$

(b) The LTS and a reduced LTS of (a) obeying D0, D1, and D2. The reduced LTS consists of the states and transitions on yellow (grey in black/white print) background, with τ replacing the labels in I. Stubborn sets are computed with $\mathsf{esc}(s, x)$, where x is the leftmost enabled action in (a)

(c) Like (b), but also V is obeyed

The methods compute a reduced version of

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

For convenience, we define

- $\bar{L} \;=\; \bar{L}_1 \mid\mid \cdots \mid\mid \bar{L}_m$,
- $V \;=\; \Sigma \setminus A$ (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 $L' \;=\; (\bar{L}_1 \mid\mid \cdots \mid\mid \bar{L}_m) \setminus I \;=\; \bar{L} \setminus I$.

It follows from the definitions that L' is the same LTS as L. The only difference between $\bar{L}_1 \mid\mid \cdots \mid\mid \bar{L}_m$ 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 stubborn sets, the elements of V and I are called *visible* and *invisible*, respectively.

If L_i has no τ-transitions, then τ_i is unnecessary. To simplify our examples, we often drop such τ_is.

Figure 7 (a) shows an example of $\bar{L}_1 \mid\mid \cdots \mid\mid \bar{L}_m$ with $m = 3$. In it, $L = L' = (\bar{L}_1 \mid\mid \bar{L}_2 \mid\mid \bar{L}_3) \setminus \{u, v, \tau_1, \tau_2\}$. The corresponding $\bar{L} = \bar{L}_1 \mid\mid \bar{L}_2 \mid\mid \bar{L}_3$ is shown in (b). We will refer to this system many times in the sequel, explaining the rest of the figure.

Each stubborn set method uses a function $\mathcal{A}$ that assigns to each $s \in S$ a subset of $\bar{\Sigma}$, called a *stubborn set*. Before discussing the definition of $\mathcal{A}$, let us see how it is used. The 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 in S_r and for each $a \in \mathcal{A}(s)$, it puts in 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') in 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 $L' = (S, \Sigma, \Delta, \hat{s})$ is that in the latter, every $a \in \bar{\Sigma}$ is used instead of every $a \in \mathcal{A}(s)$.

We will also talk about $\bar{\Delta}_\mathsf{r} = \{(s, a, s') \mid s \in S_\mathsf{r} \wedge a \in \mathcal{A}(s) \wedge (s, a, s') \in \bar{\Delta}\}$. It is otherwise like Δ_r, but the labels in I have not yet been hidden.

In Figure 7, the elements of S_r and $\bar{\Delta}_\mathsf{r}$ are drawn on yellow background. To get Δ_r, τ must be replaced for the labels in I. In (b), $\mathcal{A}(\hat{s}) = \{a\}$. In (c), $\mathcal{A}(\hat{s}) = \{a, b, u, \tau_2\}$. There is, however, neither b- nor u-transition from the initial state, because b and u are disabled there. We will see in Section 7 why $u \in \mathcal{A}(\hat{s})$ and, in general, how the $\mathcal{A}(s)$ are obtained. The τ_1-transition from the initial state is not on yellow background, because $\tau_1 \notin \mathcal{A}(\hat{s})$.

Let $\bar{L}_\mathsf{r} = (S_\mathsf{r}, \bar{\Sigma}, \bar{\Delta}_\mathsf{r}, \hat{s})$. The LTS

$$L_\mathsf{r} \;=\; (S_\mathsf{r}, \Sigma, \Delta_\mathsf{r}, \hat{s}) \;=\; \bar{L}_\mathsf{r} \setminus I$$

is the *reduced* LTS, while

$$L \;=\; L' \;=\; (S, \Sigma, \Delta, \hat{s})$$

is the *full* LTS. Because L_r is easy to obtain from $\bar{L}_\mathsf{r}$ and the labels of transitions are more informative in $\bar{L}_\mathsf{r}$, when we say that we show a picture of a reduced LTS, we sometimes actually show $\bar{L}_\mathsf{r}$. We will refer to concepts in L_r and $\bar{L}_\mathsf{r}$ with the prefix "r-", and to L and $\bar{L}$ 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. Furthermore, if an r-state f-refuses a set, then it also r-refuses the set.

In Figure 7 (b), b and ba are f-traces but not r-traces. In Figure 7 (c), the f-traces and r-traces are the same. We will soon state conditions on $\mathcal{A}$ that guarantee that every f-trace of an r-state is also its r-trace.

Typically many different functions could be used as $\mathcal{A}$, 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 $\mathcal{A}$.

Instead, we now give four conditions such that if $\mathcal{A}(s)$ satisfies them in every $s \in S_\mathsf{r}$, then the traces of the system are preserved. The efficient computation of sets that satisfy the conditions and yield good reduction is a difficult problem. We will deal with it in two steps in Sections 7 and 9. The first two conditions are illustrated in Figure 8.

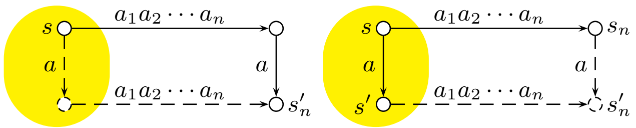


Fig. 8. Illustrating D1 (left) and D2 (right). The solid states and transition sequences are assumed to exist and the condition promises the existence of the dashed ones. The yellow part is in the reduced LTS, the rest is not necessarily

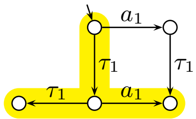


Fig. 9. An example with $\mathcal{A}(\hat{s}) = \{\tau_1\}$ but τ_1 and a_1 are not independent, although D1 and D2 hold

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

D2 If $a \in \mathcal{A}(s)$, $s \;-a_1 \cdots a_n\!\rightarrow\; s_n$, $a_1, \ldots, a_n$ are not in $\mathcal{A}(s)$, and $s \;-a\!\rightarrow\; s'$, 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 $\mathcal{A}(s) \cap V \cap \mathsf{en}(s) \neq \emptyset$, then $V \subseteq \mathcal{A}(s)$.

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

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 [16]. Unlike in the present study, actions are assumed to be deterministic. If a_1 and a_2 are independent, then in every state s (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). Figure 9 demonstrates that because the definition of independence refers to all states, an enabled action inside and outside the stubborn set need not satisfy part (1) of independence. Please see [36,37] for a more extensive discussion on this issue.

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 deadlocks as the full LTS. Indeed, all deadlocks in Figure 7 are on yellow background. D0 is not needed in the method that is the main topic of the present study, because its purpose is not to preserve deadlocks but traces. However, various issues are much easier to explain in the presence of D0 instead of SV. Therefore, we now present D0 and then a result that uses it. We will later replace SV for D0.

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

That is, if s is not a deadlock, then $\mathcal{A}(s)$ contains an enabled action.

To understand how D0, D1, and D2 do their job, we now prove that they guarantee that deadlocks are preserved, although the proof can be found in [29] and elsewhere. We also illustrate how V acts in proofs. It says that if the stubborn set contains an enabled visible action, then it contains all visible actions (also disabled ones). It makes the reduction preserve the ordering of visible actions. Together with D0, D1, and D2 it guarantees that traces that lead to deadlocks are preserved.

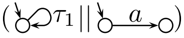
Theorem 1. *Assume D0, D1, and D2. Assume that $n \in \mathbb{N}$, $s_n \in S_\mathsf{r}$, $s' \in S$ is a deadlock, and $s_n \;-a_1 \cdots a_n\!\rightarrow\; s'$ in L. Then there is a permutation $b_1 \cdots b_n$ of $a_1 \cdots a_n$ such that $s_n \;-b_1 \cdots b_n\!\rightarrow\; s'$ in L_r. If also V is assumed, then $b_1 \cdots b_n$ has the same trace as $a_1 \cdots a_n$.*

Proof. To prove the first claim, we use induction on n. The base case $n = 0$ is obvious, since then $s' = s_n$.

If $n > 0$, then $s_n \;-a_1\!\rightarrow$. By D0, there is $a \in \mathcal{A}(s_n)$ such that $s_n \;-a\!\rightarrow$. If none of a_1, …, a_n is in $\mathcal{A}(s_n)$, then $s' \;-a\!\rightarrow$ by D2, contradicting the fact that s' is a deadlock. So there is a smallest i such that $1 \leq i \leq n$ and $a_i \in \mathcal{A}(s_n)$. There is some s'_n such that $s_n \;-a_1 \cdots a_i\!\rightarrow\; s'_n \;-a_{i+1} \cdots a_n\!\rightarrow\; s'$ in L. By D1, there is $s_{n-1} \in S_\mathsf{r}$ such that $s_n \;-a_i\!\rightarrow\; s_{n-1}$ in L_r and $s_{n-1} \;-a_1 \cdots a_{i-1}\!\rightarrow\; s'_n$ in L. Since $s_{n-1} \;-a_1 \cdots a_{i-1}\!\rightarrow\; s'_n \;-a_{i+1} \cdots a_n\!\rightarrow\; s'$ in L, the induction hypothesis yields $s_{n-1} \;-b_2 \cdots b_n\!\rightarrow\; s'$ in L_r for some permutation $b_2 \cdots b_n$ of $a_1 \cdots a_{i-1} a_{i+1} \cdots a_n$. By choosing $b_1 = a_i$ we have the claim.

To get the second claim, it suffices to prove that $s_n \;-a_i a_1 \cdots a_{i-1} a_{i+1} \cdots a_n\!\rightarrow\; s'$ has the same trace as $s_n \;-a_1 \cdots a_n\!\rightarrow\; s'$. This is obvious if $a_i \in I$. If $a_i \in V$, then a_i is an enabled visible action in $\mathcal{A}(s_n)$. By V, all visible actions are in $\mathcal{A}(s_n)$. So a_1, …, a_{i-1} are invisible. As a consequence, again $s_n \;-a_i a_1 \cdots a_{i-1} a_{i+1} \cdots a_n\!\rightarrow\; s'$ has the same trace as $s_n \;-a_1 \cdots a_n\!\rightarrow\; s'$. □

The system in Figure 7 (a) has the traces ε, a, aa, ab, b, and ba. Indeed, the reduced LTS in (c) has them all. However, there was a cost: the reduced LTS in (c) is much bigger than in (b). We can see from the figure that ba cannot be preserved unless $\tau_2 \in \mathcal{A}(\hat{s})$, and aa cannot be preserved unless a or τ_1 is in $\mathcal{A}(\hat{s})$. We will return to this example in Section 7.

The function $\mathcal{A}_\emptyset$ that always returns the empty set satisfies D1, D2, and V. Its use as $\mathcal{A}$ 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 deadlocks as the full LTS. Adding D0 solves the problem for deadlocks and deadlock traces, but not for all traces, because $\{\tau_1\}$ satisfies D0, D1, D2, and V in the initial state of () $\setminus \{\tau_1\}$. Then a reduced LTS is obtained whose only transition is a τ-transition from the initial state to itself, losing the trace a. This is an example of the ignoring problem mentioned in Section 1.

The condition SV 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. In Section 6 we prove that D1, D2, V, and SV guarantee that the reduced LTS is fair testing equivalent (and thus also trace equivalent) to the full LTS. The details of how SV does its job will become clear in the proof of Lemma 2.

The name is SV because, historically, a similar condition was first used to guarantee the preservation of what is called safety properties in the linear temporal logic framework [15]. So "S" refers to safety and "V" refers to the use of the notion of visibility. Recently, an improvement of SV has been found [33,35,37], but its definition is so complicated that we skip it in the present study.

If $V = \emptyset$, then $\mathcal{A}_\emptyset$ satisfies also SV. 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 $\mathcal{A}_\emptyset$ is trace-preserving and yields ideal reduction results.

No matter what V is, the function $\mathcal{A}(s) = \bar{\Sigma}$ always satisfies D1, D2, V, and SV (and D0). However, it does not yield any reduction.

6 The Fair Testing and Tree Failure Equivalence Preservation Theorems

In most of this section we assume that $L_\mathsf{r} = (S_\mathsf{r}, \Sigma, \Delta_\mathsf{r}, \hat{s})$ has been constructed with the trace-preserving stubborn set method in Section 5, that is, obeying D1, D2, V, and SV. We first 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. Then we demonstrate that the method does not necessarily preserve tree failure equivalence, but it can be made to preserve it by replacing a stronger condition for V.

We first prove a result that will afterwards be used to prove the induction step in an induction proof. The induction proof will show that if an r-state has a non-empty f-trace, then the state has the trace also as an r-trace. By the definition of traces, the f-trace arises from some path in L. Lemma 2 tells that there is a path in L that starts at the same state, has the same trace, whose first part is in L_r, and the rest of whom is shorter than the original path. The lemma and its proof are illustrated in Figure 10 left.

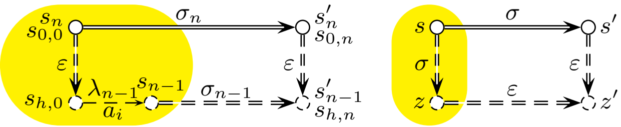


Fig. 10. Illustrating Lemma 2 (left) and Lemma 3 (right)

Lemma 2. *Assume D1, D2, V, and SV. 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. Then 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 v such that $1 \leq v \leq n$ and $a_v \in V$. By SV, there are $k \in \mathbb{N}$, $s_{1,0}, \ldots, s_{k,0}$, and $b_1, \ldots, b_k$ such that $a_v \in \mathcal{A}(s_{k,0})$ 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 $\{a_1, \ldots, a_n\} \cap \mathcal{A}(s_{h,0}) \neq \emptyset$. We have $0 \leq h \leq k$ because $a_v \in \mathcal{A}(s_{k,0})$. 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 $V \subseteq \mathcal{A}(s_{i-1,0})$ by V. It yields $a_v \in \mathcal{A}(s_{i-1,0})$, which contradicts the choice of h. So $\{b_1, \ldots, b_h\} \subseteq I$. 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 $\{a_1, \ldots, a_n\} \cap \mathcal{A}(s_{h,0}) \neq \emptyset$, there is a smallest i such that $1 \leq i \leq n$ and $a_i \in \mathcal{A}(s_{h,0})$. 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. Let σ_{n-1} be the trace of $a_1 \cdots a_{i-1}a_{i+1} \cdots a_n$. We choose $s'_{n-1} = s_{h,n}$. 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 $V \subseteq \mathcal{A}(s_{h,0})$ 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. □

We now show that if an r-state has an (empty or non-empty) f-trace, then the state has the trace also as an r-trace. What is more, there is an f-state that is f-reachable via invisible actions from the end of both the original f-path and the r-path. This latter property will be used afterwards to prove that the refusal sets after

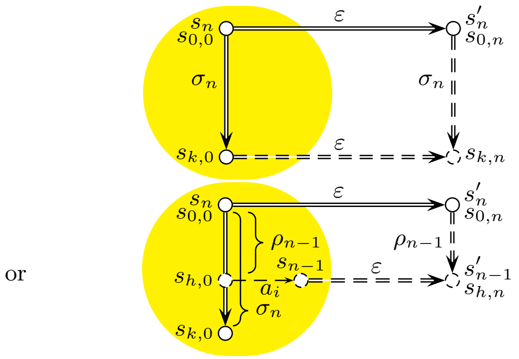


Fig. 11. Illustrating Lemma 4; a_i is invisible

the trace in L_r are what they should be. The lemma is illustrated in Figure 10 right.

Lemma 3. *Assume D1, D2, V, and SV. Assume that $n \in \mathbb{N}$, $s \in S_\mathsf{r}$, $s' \in S$, $\sigma \in V^*$, 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 2 and get $s_{n-1} \in S_\mathsf{r}$, $s'_{n-1} \in S$, $\lambda_{n-1} \in V \cup \{\varepsilon\}$, and $\sigma_{n-1} \in V^*$ such that $\lambda_{n-1}\sigma_{n-1} = \sigma$, $s =\lambda_{n-1}\Rightarrow s_{n-1}$ in L_r, and $s' =\varepsilon\Rightarrow s'_{n-1}$ in L. Furthermore, $s_{n-1} =\sigma_{n-1}\Rightarrow s'_{n-1}$ in L due to an f-path of length $n-1$, for which the lemma holds by the induction assupmtion. Hence, there are $z \in S_\mathsf{r}$ and $z' \in S$ such that $s_{n-1} =\sigma_{n-1}\Rightarrow z$ in L_r, $z =\varepsilon\Rightarrow z'$ in L, and $s'_{n-1} =\varepsilon\Rightarrow z'$ in L. These give $s =\lambda_{n-1}\Rightarrow s_{n-1} =\sigma_{n-1}\Rightarrow z$ in L_r and $s' =\varepsilon\Rightarrow s'_{n-1} =\varepsilon\Rightarrow z'$ in L, so we are done. □

The next two lemmata deal with refusal sets analogously to how Lemmas 2 and 3 dealt with traces. Lemma 4 has two cases, both of which and whose proofs are illustrated in Figure 11.

Lemma 4. *Assume D1 and D2. Assume that $n \in \mathbb{N}$, $s_n \in S_\mathsf{r}$, $\sigma_n \in V^*$, $s_n =\sigma_n\Rightarrow$ in L_r, $s'_n \in S$, and there is an f-path of length n from s_n to s'_n such that its trace is ε. Then 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 $\{a_1, \ldots, a_n\} \cap \mathcal{A}(s_{i,0}) = \emptyset$ 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 $\{a_1, \ldots, a_n\} \cap \mathcal{A}(s_{h,0}) \neq \emptyset$. There also is a smallest i such that $1 \leq i \leq n$ and $a_i \in \mathcal{A}(s_{h,0})$. 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 5. *Assume D1 and D2. Assume that $n \in \mathbb{N}$, $K \subseteq V^*$, $\kappa \in K$, $z \in S_\mathsf{r}$, $z' \in S$, and $z =\varepsilon\Rightarrow z'$ due to an f-path of length n. Assume further that z' f-refuses K and $z =\kappa\Rightarrow$ 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 $n = 0$ yields $z' = z$, implying $z' =\kappa\Rightarrow$ in L_r and, consequently, in L. On the other hand, $\kappa \in K$ and z' f-refuses K. These are in contradiction. So the assumptions cannot all hold when $n = 0$, making the case $n = 0$ hold vacuously.

To prove the induction step, we assume the assumptions in the lemma for n and that the lemma holds for $n-1$. We apply Lemma 4 to z, z', and κ. In the first case, we would again have the impossible $z' =\kappa\Rightarrow$. So only the second case can hold. According to it, we have a $z_{n-1} \in S_\mathsf{r}$, $z'_{n-1} \in S$, and prefix ρ of κ and thus of K such that $z =\rho\Rightarrow z_{n-1}$ in L_r, $z' =\rho\Rightarrow z'_{n-1}$ in L, and $z_{n-1} =\varepsilon\Rightarrow z'_{n-1}$ due to an f-path of length $n-1$.

Since z' f-refuses K, z'_{n-1} must f-refuse $\rho^{-1}K$. If z_{n-1} r-refuses $\rho^{-1}K$, we can choose $s = z_{n-1}$ and $\pi = \rho$, and we are done. Otherwise, we can apply the induction hypothesis to $z_{n-1} =\varepsilon\Rightarrow z'_{n-1}$ and $\rho^{-1}K$. This results in an $s \in S_\mathsf{r}$ and a prefix π' of $\rho^{-1}K$ such that $z_{n-1} =\pi'\Rightarrow s$ 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 after choosing $\pi = \rho\pi'$ we are done. □

We are ready to prove our first main theorem.

Theorem 6. *If L_r obeys D1, D2, V, and SV, then it is fair testing equivalent to L.*

Proof. Part 1 of Definition 2 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 3, $s =\rho\Rightarrow$ also in L_r. This implies that s f-refuses K and that (σ, K) is a tree failure of L. In conclusion, part 2 of Definition 2 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 3 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.

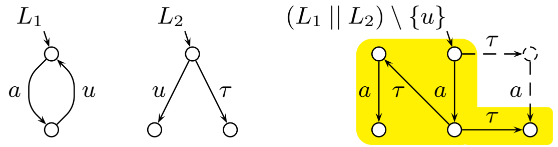


Fig. 12. A counterexample to the preservation of all tree failures. In $(L_1||L_2)\setminus\{u\}$, the solid states and transitions are in the reduced and the dashed ones only in the full LTS

Either z r-refuses K and we are done, or we apply Lemma 5, 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 2 also holds. □

We now discuss a counterexample that shows that the method does not preserve tree failure equivalence. Consider $(L_1 \,||\, L_2) \setminus \{u\}$, where L_1 and L_2 are shown in Figure 12 left and middle. Initially three sets are stubborn: $\{a\}$, $\{a, u\}$, and $\{a, u, \tau_2\}$. If $\{a\}$ is chosen, then the LTS is obtained that is shown with solid arrows on the right in Figure 12. The full LTS also contains the dashed arrows. The full LTS has the tree failure $(\varepsilon, \{aa\})$ that the reduced LTS lacks.

We conclude this section by proving that if V is replaced by the stronger condition C2 from [3, p. 149], then the method does preserve tree failure equivalence.

C2 If $\mathcal{A}(s) \cap V \cap \mathsf{en}(s) \neq \emptyset$, then $\mathcal{A}(s) = \bar{\Sigma}$.

C2 says that if the aps set contains an enabled visible action, then it must contain all actions. (To be precise, [3] only requires that the aps set contains all enabled actions. However, that yields the same reduced LTS.)

Lemma 7. *Assume D1, D2, and C2. Assume that $n \in \mathbb{N}$, $K \subseteq V^*$, $s_n \in S_\mathsf{r}$, $s'_n \in S$, and there is an f-path of length n from s_n to s'_n such that its trace is ε. Assume further that s'_n f-refuses K. Then there exists $s \in S_\mathsf{r}$ such that $s_n =\varepsilon\Rightarrow s$ in L_r and s r-refuses K.*

Proof. We use induction on n. The base case $n = 0$ is obvious, because then $s'_n = s_n$ and f-refusal guarantees r-refusal.

In the induction step, $n > 0$. If s_n r-refuses K, then we can choose $s = s_n$ and we are done. Otherwise, there is $\kappa \in K$ such that $s_n =\kappa\Rightarrow$ in L_r. Because s'_n f-refuses K, $\kappa \neq \varepsilon$. Let b be the first action of κ, and let $s_{0,0} = s_n$. So there is a path $s_{0,0} -b_1 \cdots b_k\rightarrow s_{k,0}$ in L_r such that $b_k = b \in V$ and $b_i \in I$ when $1 \leq i < k$. Let $s_n -a_1 \cdots a_n\rightarrow s'_n$ be the path in L of length n. The a_i are invisible, because the trace of the path is ε. Because $n > 0$, a_1 exists. By C2, $\mathcal{A}(s_{k-1,0}) = \bar{\Sigma}$. So there is a smallest h such that $0 \leq h < k$ and $\{a_1, \ldots, a_n\} \cap \mathcal{A}(s_{h,0}) \neq \emptyset$.

Similarly to the last paragraph of the proof of Lemma 4, h applications of D2 followed by an application of D1 yield an i, s_{n-1}, and s'_{n-1} such that $s_n -b_1 \cdots b_h a_i\rightarrow s_{n-1}$ in L_r, $s_{n-1} -a_1 \cdots a_{i-1} a_{i+1} \cdots a_n\rightarrow s'_{n-1}$ in L, and $s'_n -b_1 \cdots b_h\rightarrow s'_{n-1}$ in L. Because $h < k$, the b_j are invisible. So $s_n =\varepsilon\Rightarrow s_{n-1}$ in L_r, there is an f-path of length $n-1$ from s_{n-1} to s'_{n-1} with the trace ε, and $s'_n =\varepsilon\Rightarrow s'_{n-1}$ in L. Because s'_n f-refuses K, also s'_{n-1} f-refuses K. By the induction hypothesis, there is $s \in S_\mathsf{r}$ such that $s_{n-1} =\varepsilon\Rightarrow s$ in L_r and s r-refuses K. Because $s_n =\varepsilon\Rightarrow s_{n-1}$ in L_r was already shown, $s_n =\varepsilon\Rightarrow s$ in L_r and we have the claim. □

Theorem 8. *If L_r obeys D1, D2, C2, and SV, then it is tree failure equivalent to L.*

Proof. It is immediate from the construction that $\Sigma_1 = \Sigma_2$. The second part of the proof of Theorem 6 actually shows that if (σ, K) is a tree failure of L_r, then it also is a tree failure of L. Because C2 implies V, this fact applies also here.

It remains to be proven that if (σ, K) is a tree failure of L, then it also is a tree failure of L_r. So we assume that there is s' such that $\hat{s} =\sigma\Rightarrow s'$ in L and s' f-refuses K. By Lemma 3 there are $z \in S_\mathsf{r}$ and $z' \in S$ such that $\hat{s} =\sigma\Rightarrow z$ in L_r, $z =\varepsilon\Rightarrow z'$ in L, and $s' =\varepsilon\Rightarrow z'$ in L. Since s' f-refuses K, also z' f-refuses K. By Lemma 7, there is $s \in S_\mathsf{r}$ such that $z =\varepsilon\Rightarrow s$ in L_r and s r-refuses K. Clearly $\hat{s} =\sigma\Rightarrow s$ in L_r via z, so we are done. □

7 On Computing Stubborn Sets

The computation of stubborn sets that satisfy D1, D2, V, and SV and yield good reduction consists of two quite different tasks. First, sets that satisfy D1, D2, V, and an additional property called D0V are computed based on information on only the current state. Second, based on information on more than one state, algorithms that perform the first task are executed in a coordinated manner to enforce SV. How to do this has been described in [29,33,37,39], among others. However, as we will argue in Section 9, SV often holds automatically. Section 9 also introduces D0V. In this section we discuss one algorithm for the first task using, for simplicity, D0 instead of D0V. We emphasize that it is not the only good algorithm. Other possibilities have been discussed in [28, 36], among others.

Because the expression under analysis is of the form $(\bar{L}_1 \,||\, \cdots \,||\, \bar{L}_m) \setminus I$, its states are of the form $(s_1, \ldots, s_m)$, where $s_i \in S_i$ for each $1 \leq i \leq m$. We employ the notation $\mathsf{en}_i(s_i) = \{a \mid \exists s'_i : (s_i, a, s'_i) \in \bar{\Delta}_i\}$, that is, the set of actions that are enabled in s_i in $\bar{L}_i$. 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. For instance, in Figure 7, $\mathsf{en}_1(\hat{s}_1) = \{v, \tau_1\}$, $\mathsf{en}_2(\hat{s}_2) = \{u, \tau_2\}$, $\mathsf{en}_3(\hat{s}_3) = \{a\}$, $\mathsf{dis}(\hat{s}, u) = 3$, and $\mathsf{dis}(\hat{s}, v) = 2$.

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

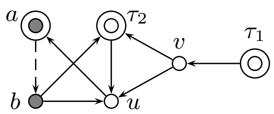


Fig. 13. The "$\rightsquigarrow_{\hat{s}}$"-graph of the system in Figure 7

Theorem 9. *Assume that the following hold for $s = (s_1, \ldots, s_m)$ and for every $a \in \mathcal{A}(s)$:*

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

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

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

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 condition 1, there is i such that $\bar{L}_i$ disables a and $\mathsf{en}_i(s_i) \subseteq \mathcal{A}(s)$. To enable a, it is necessary that $\bar{L}_i$ changes its state, which requires that some action in $\mathsf{en}_i(s_i)$ occurs. These are all in $\mathcal{A}(s)$ and thus distinct from a_1, …, 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 $1 \leq j \leq m$ and $1 \leq k \leq n$ 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 $\bar{L}_j$ does not move between s and s' since none of $a_1, \ldots, a_{k-1}$ is in $\bar{\Sigma}_j$. By condition 2, $\mathsf{en}_j(s_j) \subseteq \mathcal{A}(s)$. This contradicts $a_k \notin \mathcal{A}(s)$.

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

Theorem 9 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 $a \rightsquigarrow_s b$, if and only if $a \neq b$ and 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 $b \in \mathsf{en}_i(s_i)$. The graph for the initial state of Figure 7 (b) is shown in Figure 13 without the dashed edge. Enabled actions are shown with double circles and visible actions with grey circles.

By the construction, if $\mathcal{A}(s)$ is closed under the graph (that is, for every a and b, if $a \in \mathcal{A}(s)$ and $a \rightsquigarrow_s b$, then $b \in \mathcal{A}(s)$), then $\mathcal{A}(s)$ satisfies D1 and D2. To also satisfy D0, it is necessary and sufficient that either there are no enabled actions, or the closed set contains an enabled action. Many closed sets can be found in Figure 13, including $\emptyset$, $\{a\}$, $\{a, u\}$, and $\{a, u, \tau_2\}$. D0 rules out $\emptyset$ and accepts the rest. Of the rest, $\{a\}$ and $\{a, u\}$ are equally good and better than the others, because their only enabled action is a, while the others also have at least τ_2. This is why $\mathcal{A}(\hat{s}) = \{a\}$ in Figure 7 (b).

It is not necessary for correctness to use the smallest i, when more than one $\bar{L}_i$ disables a. The choice to use the smallest i was made to obtain a fast algorithm and to avoid confusion in the examples. An alternative algorithm (called *deletion algorithm* in [28]) 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 9 and the corresponding "$\rightsquigarrow_s$"-relation are not the weakest possible to achieve D1 and D2. To see this, we now discuss two scenarios.

Assume that w writes to a finite-capacity fifo $\bar{L}_f$, r reads from it, and they have no other $\bar{L}_i$ in common. Although $\bar{\Sigma}_f$ links them, we need not declare $w \rightsquigarrow_s r$ when w is enabled, and we need not declare $r \rightsquigarrow_s w$ when r is enabled, since they commute if both are enabled. We will exploit this in Section 10. (On the other hand, when the fifo is empty, it may be that $\mathsf{dis}(s, r) = f$, yielding $r \rightsquigarrow_s w$. That is, to enable reading from a fifo that is currently empty, it is necessary to write something to it. Similarly, when the fifo is full, $w \rightsquigarrow_s r$ may hold, because to enable writing, one has to make room in the fifo.) At the LTS level, it is laborious to detect if any of the $\bar{L}_i$ models a fifo, while at a higher level of description, it has usually been mentioned explicitly.

As another scenario, if $i = \mathsf{dis}(s, a)$ and there is no path in $\bar{L}_i$ from s_i to any s'_i with $a \in \mathsf{en}_i(s'_i)$, then a is permanently disabled, so $a \rightsquigarrow_s b$ need not be declared for any $b \in \mathsf{en}_i(s_i)$ – and, indeed, for any b at all. Consider the state in Figure 7 that is reachable by executing $a\tau_2$. In it, u is permanently disabled by $\bar{L}_2$. However, the condition in Theorem 9 unnecessarily declares $u \rightsquigarrow_s v$. Because systems are often meant to repeat the same activity forever (for instance, the protocols in Section 3 are not meant to deliver two messages and then stop, but are meant to deliver messages as long as there are any to deliver), LTSs are often strongly connected, so testing this condition at the LTS level would often be wasted work. Testing it becomes worthwile if higher-level knowledge tells that the LTS is not strongly connected.

These scenarios illustrate that the more knowledge there is about the system at a level higher than the LTS, the more optimizations tend to be available. As a consequence, trying to make the condition in Theorem 9 as weak as possible would not succeed, and it would make it very hard to read. On the other hand, D1, D2, and so on have been made very weak, with the hope that when new optimizations are found, it would suffice to prove that they guarantee D1, D2, and so on, instead of having to prove anew all correctness theorems, such as Theorems 6 and 8 in the present publication.

It is trivial to also take V into account in the graph representation of the stubborn set computation problem. It suffices to add the edge $a \leadsto_s b$ from each $a \in V \cap \mathsf{en}(s)$ to each $b \in V \setminus \{a\}$. This extended edge set is what we denote by "$\leadsto_s$" henceforth. In the case of Figure 13, the dashed edge $a \leadsto_{\hat{s}} b$ is added. Then there are four closed sets: $\emptyset$, $\{a, b, u, \tau_2\}$, $\{a, b, u, v, \tau_2\}$, and $\{a, b, u, v, \tau_1, \tau_2\}$. Again, D0 rules out $\emptyset$. All the other three have at least two enabled actions, a and τ_2. The last one is inferior, because it also has the enabled τ_1. In Figure 7 (c), $\{a, b, u, \tau_2\}$ was chosen, but $\{a, b, u, v, \tau_2\}$ could have been chosen as well.

At this point we emphasize that the graph $(\bar{\Sigma}, \text{"}\leadsto_s\text{"})$ need not be represented as an explicit data structure. So its use does not entail the cost of constructing it for every $s \in S_{\mathsf{r}}$. Instead, there is a piece of program that, given a and having access to s and every $\bar{L}_i$, returns, one by one, the b such that $a \leadsto_s b$. If a is enabled, it finds every i such that $a \in \bar{\Sigma}_i$, finds each s_i, and returns the labels of the output transitions of s_i in $\bar{L}_i$ except a itself. (As a matter of fact, it is harmless to return a itself. So it is unnecessary to write additional code to protect against returning a itself.) If a is disabled, it computes $i = \mathsf{dis}(s, a)$, finds s_i, and continues like when a is enabled.

Let "$\leadsto_s^*$" denote the reflexive transitive closure of "$\leadsto_s$". By the definitions, if $a \in \bar{\Sigma}$, then $\{b \mid a \leadsto_s^* b\}$ satisfies D1 and D2 (and V, if "$\leadsto_s$" respects it). We denote it with $\mathsf{clsr}(s, a)$. If $A \subseteq \bar{\Sigma}$, by $\mathsf{clsr}(s, A)$ we mean $\bigcup_{a \in A} \mathsf{clsr}(s, a)$. Both can be computed quickly with well-known elementary graph search algorithms.

D0 may be enforced by choosing an enabled a and computing $\mathcal{A}(s) = \mathsf{clsr}(s, a)$. Unfortunately, the quality of the result is vulnerable to the choice of a. In Figure 13, $\mathsf{clsr}(\hat{s}, \tau_1)$ yields a worse result than $\mathsf{clsr}(\hat{s}, a)$.

Therefore, we employ an algorithm called $\mathsf{esc}(s, a)$, for "enabled strong component". Applied at some state s, it uses a as the starting point of a depth-first search in $(\bar{\Sigma}, \text{"}\leadsto_s\text{"})$. During the search, the strong components (that is, the maximal strongly connected subgraphs) of $(\bar{\Sigma}, \text{"}\leadsto_s\text{"})$ are recognized using Tarjan's algorithm [6, 8, 24]. It recognizes each strong component at the time of backtracking from it. When $\mathsf{esc}(s, a)$ finds a strong component C that contains an action enabled at s, it stops and returns $\mathsf{clsr}(s, C)$ as the result; note that a might not be in C. If $\mathsf{esc}(s, a)$ does not find such a strong component, it returns $\mathsf{clsr}(s, a)$.

Obviously $\mathsf{esc}(s, a) \subseteq \mathsf{clsr}(s, a)$. So $\mathsf{esc}(s, a)$ has potential for better reduction results. By the properties of Tarjan's algorithm, all enabled actions of $\mathsf{clsr}(s, C)$ are in C. Therefore, $\mathsf{esc}(s, a)$ is optimal in the sense that no closed proper subset of $\mathsf{esc}(s, a)$ contains enabled actions. In Figure 13, no matter what x is, $\mathsf{esc}(\hat{s}, x)$ returns $\{a\}$ in the absence and $\{a, b, u, \tau_2\}$ in the presence of the dashed edge. In contrast, $\mathsf{clsr}(\hat{s}, b) = \{a, b, u, \tau_2\}$ in the absence and $\mathsf{clsr}(\hat{s}, \tau_1) = \{a, b, u, v, \tau_1, \tau_2\}$ in the presence of the dashed edge.

Tarjan's algorithm adds very little overhead to depth-first search, in particular if the optimizations in [6, 8] are exploited. Therefore, $\mathsf{esc}(s, a)$ is never much slower than $\mathsf{clsr}(s, a)$. On the other hand, $\mathsf{esc}(s, a)$ might find a suitable strong component early on, in which case it is much faster than $\mathsf{clsr}(s, a)$.

The publicly available ASSET tool [31] contains an implementation of this algorithm, with the modification that instead of returning any set, it executes each enabled action that it finds (that is, finds the s' such that $s -a\rightarrow s'$ and puts (s, a, s') in $\bar{\Delta}_{\mathsf{r}}$), and terminates when the C described above has been processed. It returns a truth value telling whether it executed at least one action.

8 On the Performance of Various Conditions

In this section we discuss some problems in the use of aps sets that have not received as much attention in the literature as they deserve. In addition to pointing out potential topics for future research, they motivate the approach in Section 9.

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 [28] and the deletion algorithm [28] appeared in [13].

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 actions 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 [26] gave an example where always choosing the set with the smallest number of enabled actions does not yield the best reduction result. Recently it has been observed that always choosing a singleton stubborn set if one is available does not necessarily guarantee best reduction results [37].

Figure 14 illustrates that the order in which the component LTSs of a system are given to a tool can have a tremendous effect on the running time and the size of the reduced LTS. In it, the stubborn sets are computed with the $\mathsf{esc}(s, a)$-algorithm, picking the starting points

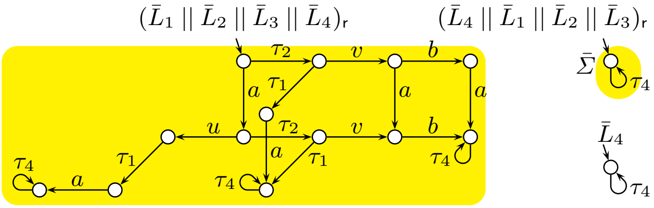

$(\bar{L}_1 \,||\, \bar{L}_2 \,||\, \bar{L}_3 \,||\, \bar{L}_4)_\mathsf{r}$
$(\bar{L}_4 \,||\, \bar{L}_1 \,||\, \bar{L}_2 \,||\, \bar{L}_3)_\mathsf{r}$
τ_2
v
b
a
τ_1
a
a
u
τ_2
v
b
τ_4
τ_1
a
τ_1
τ_4
a
τ_4
τ_4
$\bar{\Sigma}$
τ_4
$\bar{L}_4$
τ_4

Fig. 14. Two reduced LTSs, where $\bar{L}_1$, $\bar{L}_2$, and $\bar{L}_3$ are from Figure 7, and $\bar{L}_4$ is shown on bottom right. The alphabet of the LTS on top right is $\bar{\Sigma}$

from the alphabet of the first LTS, then from the alphabet of the second, and so on, until a suitable set is found or the LTSs are exhausted. The method that preserves deadlocks and traces that lead to them is used. That is, D0, D1, D2, and V are obeyed, but not necessarily SV.

In the case of $(\bar{L}_1 \,||\, \bar{L}_2 \,||\, \bar{L}_3 \,||\, \bar{L}_4)_\mathsf{r}$, the algorithm finds the same stubborn sets as in Figure 7 (c), except when Figure 7 (c) deadlocks. In the latter states, no enabled actions are found from $\bar{\Sigma}_1$, $\bar{\Sigma}_2$, and $\bar{\Sigma}_3$, so the algorithm proceeds to $\bar{\Sigma}_4 = \{\tau_4\}$. Therefore, it tries τ_4 and finds that it introduces a self-loop to the state. The algorithm thus correctly finds out that the system has no deadlocks.

With $(\bar{L}_4 \,||\, \bar{L}_1 \,||\, \bar{L}_2 \,||\, \bar{L}_3)_\mathsf{r}$, the algorithm tries $\bar{\Sigma}_4$ first. Therefore, it uses $\{\tau_4\}$ as the stubborn set already in the initial state, and thus constructs the self-loop $\hat{s} \xrightarrow{\tau_4} \hat{s}$. D0, D1, D2, and V do not tell it to investigate anything else. So it stops extremely quickly, after constructing only one state and one transition. Again, the algorithm correctly finds out that the system has no deadlocks.

It is clear that any parallel composition could be in the place of $\bar{L}_1 \,||\, \bar{L}_2 \,||\, \bar{L}_3$ in the above example. Furthermore, similar examples can be constructed also in the presence of SV. For instance, one input order could lure the algorithm into investigating a huge sub-LTS that has no occurrences of visible actions, while it is avoided with another input order. These examples mean that the very same method and verification program can give dramatically different results on the very same example system, if the system is written in two different orders. A similar observation on dynamic partial order redcution has been made in [14]. For this and other reasons, measurements are not as reliable for comparing different methods as we would like.

Technically, optimal sets could be defined, for instance, as those (not necessarily aps) sets of enabled actions that yield the smallest reduced state space that preserves the deadlocks. Unfortunately, it was shown in [36] 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 [3, 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 state space is what they try to avoid. In [36], 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 [36] 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 $\mathcal{E}(\mathsf{esc}(s,a))$ for some a. (Unfortunately, we do not know which a to choose.) In this restricted but nevertheless meaningful sense, the $\mathsf{esc}(s,a)$-algorithm is optimal.

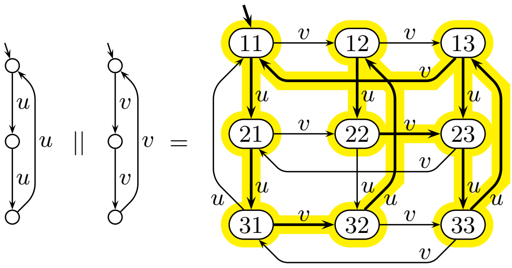


Fig. 15. Actions are tried in the order u, v until one is found that does not close a cycle. If such an action is not found, then all actions are taken

The situation is much more complicated when preserving other properties than deadlocks. We only discuss one difficulty. Instead of SV, [3, 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 actions. In [5] it was demonstrated that this condition performs badly on a variant of the dining philosophers' system. Figure 15 shows an example where this condition causes the method to zigzag through the LTS such that all reachable states are constructed, although the component LTSs do not interact at all. SV is less vulnerable to but not totally free from this kind of difficulties. Far too little is known on this problem area.

In general, it is reasonable to try to find as weak conditions as possible in place of D1, V, SV, 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, [3, p. 149] requires the condition C2 that we met in Section 6. It is strictly stronger than V and thus has less potential for reduction. Furthermore, the algorithm in Section 7 can exploit the additional potential of V at least to some extent. In Figure 7 (c), $\tau_1 \in \mathsf{en}(\hat{s})$ and $\tau_1 \notin \mathcal{A}(\hat{s})$. On the other hand, C2 would force to take τ_1 into the set, because

$a \in \mathcal{A}(\hat{s}) \cap V \cap \mathsf{en}(\hat{s})$. So V yields better reduction than C2 in this example.

This also illustrates why stubborn sets are defined such that they may contain disabled actions. The part $V \subseteq \mathcal{A}(s)$ in the definition of V could not be formulated easily, or perhaps not at all, if $\mathcal{A}(s)$ cannot contain disabled actions. For instance, $V \cap \mathsf{en}(s) \subseteq \mathcal{A}(s)$ fails, because it lets to choose $\mathcal{A}(\hat{s}) = \{a\}$ and thus loses the trace b in Figure 7 (c).

9 The SV Condition Often Holds Automatically

In this section we make the novel observation that if the stubborn sets are computed using the visible actions as the starting points, then either SV holds or the system and its reduced LTS exhibit pathological behaviour. This means that the implementations of SV presented in [29, 33, 37, 39], among others, are usually not needed. Therefore, we do not present any in this study, but, of course, they can be used if considered necessary.

The observation continues a line of research started in [34], whose conference version was published in 2015. We say that a system is *always may-terminating* if and only it from every reachable state, it can continue to a deadlock. The main result of [34] was that it is easy to check from the reduced LTS whether the system is always may-terminating, and if it is, then SV holds automatically. This means that when we want the system be always may-terminating, then we need not implement SV (even if we do not know whether it indeed is always may-terminating). We could apply this idea to our example protocols by adding a sending client to the models that in its initial state chooses between entering a deadlock or executing senN or senY and then returning to the initial state. In this section we obtain results of similar spirit, with weaker conditions in the place of may-termination.

We first investigate the following easily implementable condition that is strictly weaker than SV.

D0V If $\mathsf{en}(s) \neq \emptyset$, then $V \subseteq \mathcal{A}(s)$ or $\mathcal{A}(s) \cap \mathsf{en}(s) \neq \emptyset$.

D0V is strictly weaker than D0 (also in the presence of D1, D2, and V), because it is implied by D0 but allows $\mathcal{A}(s) \cap \mathsf{en}(s) = \emptyset$ when there are enabled actions outside but not inside $\mathsf{clsr}(s, V)$. D0V is implied by SV, because if $\mathcal{A}(s) \cap \mathsf{en}(s) = \emptyset$, then the s_a in SV can only be s itself, so $a \in \mathcal{A}(s)$ for each $a \in V$. D0V combined with D1, D2, and V does not guarantee the preservation of traces, because it allows $\mathcal{A}(\hat{s}) = \{\tau_1\}$ in $\circlearrowleft \tau_1 \,||\, \circlearrowleft a$.

Our strategy is to only use subsets of $\mathsf{clsr}(s, V)$ as stubborn sets, to avoid executing actions that do not contribute towards satisfying SV. If $\mathsf{clsr}(s, V)$ contains no enabled actions, then no actions need to be executed, because $\mathcal{A}(s) = \mathsf{clsr}(s, V)$ implies $V \subseteq \mathcal{A}(s)$, making D0V hold in s and SV hold in every r-state from which s is r-reachable. There may be enabled actions also in this case, but they are invisible and cannot lead to occurrences of visible actions.

D0V with D1, D2, and V can be implemented by computing $\mathsf{esc}(s, a)$ for each $a \in V$ until an enabled action is encountered or V is exhausted, taking the union of the results. We call this *the* $\mathsf{esc}(s, V)$*-algorithm*. In the former case, $\mathsf{esc}(s, V) = \mathsf{esc}(s, a') \cup \bigcup_{a \in V'} \mathsf{clsr}(s, a)$ and $\mathsf{esc}(s, V) \cap \mathsf{en}(s) = \mathsf{esc}(s, a') \cap \mathsf{en}(s)$, where a' is the $a \in V$ that yielded an enabled action, and V' is the set of $a \in V$ that were tried before a'. In the latter case, $\mathsf{esc}(s, a) = \mathsf{clsr}(s, a)$ for each $a \in V$, so $\mathsf{esc}(s, V) = \mathsf{clsr}(s, V)$.

The $\mathsf{esc}(s, V)$-algorithm inputs a single state and returns a stubborn set in it. By the *D0V-algorithm* we mean the algorithm that constructs a reduced LTS by always choosing $\mathcal{A}(s) = \mathsf{esc}(s, V)$. The D0V-algorithm inputs a system description and produces a reduced LTS that satisfies D0V, D1, D2, and V, but not necessarily SV.

We say that an LTS is *always may-stabilizing*, if and only if from every reachable state, a stable state is reachable. It is *always may-progressing*, if and only if from every reachable state, a deadlock or an occurrence of a visible action is reachable. (Calling a deadlock may-progress arises from the fact that it is easy to check from each deadlock state whether it represents intended or erroneous termination. So in the present context, deadlocks can be ignored as a sub-case that has already been solved.)

Lemma 10. *If an LTS is always may-stabilizing, then it is always may-progressing.*

Proof. If L is always may-stabilizing, then from every $s \in S$, a stable s' is reachable. If s' is not a deadlock, then it has an outgoing transition. Its action is visible, because otherwise s' would not be stable. □

The following observation is important.

Theorem 11. *If L_r obeys D0V and V and is always may-progressing or always may-stabilizing, then it obeys SV.*

Proof. To prove the first claim, let L_r be always may-progressing and $s \in S_\mathsf{r}$. By the definition of always may-progressing, at least one of the following two cases applies to some s' that is r-reachable from s. Each case yields $V \subseteq \mathcal{A}(s')$, making SV hold for s.

If $s' -a\rightarrow$ in L_r where $a \in V$, then $V \subseteq \mathcal{A}(s')$ by V.

If s' is an r-deadlock, then D0V yields either $V \subseteq \mathcal{A}(s')$ or that s' is an f-deadlock. In the latter case we can use $\mathcal{A}(s') = \bar{\Sigma}$ without loss of reduction, and thus have $V \subseteq \mathcal{A}(s')$ in any case.

The second claim follows from Lemma 10. □

That L_r is always may-progressing can be checked in linear time with breadth-first or depth-first search, using r-deadlocks and tail states of visible r-transitions as starting points and traversing the r-transitions backwards.

So we can construct a reduced LTS for the system obeying D0V, D1, D2, and V with the D0V-algorithm, and then test whether it is always may-progressing. If it is, we can use it. Let us study some facts that together suggest that always may-progressing reduced LTSs are common.

Lemma 12. *Assume D1 and D2. If the system is always may-stabilizing, then also the reduced LTS is.*

Proof. Let $s_n \in S_\mathsf{r}$, and let $s_n \,{-}a_1 \cdots a_n{\rightarrow}\, s$ be an f-path to an f-stable f-state s. We show by induction on n that there is an r-path from s_n to an r-stable r-state. Because any f-stable r-state is also r-stable by $\Delta_\mathsf{r} \subseteq \Delta$, the base case $n = 0$ holds.

Let now $n > 0$. If s_n is r-stable, the claim holds trivially. Otherwise there is $a \in I \cap \mathsf{en}(s_n) \cap \mathcal{A}(s_n)$. If none of $a_1, \ldots, a_n$ is in $\mathcal{A}(s_n)$, then D2 yields $s \,{-}a{\rightarrow}$, contradicting the f-stability of s. So there is a smallest i such that $1 \leq i \leq n$ and $a_i \in \mathcal{A}(s_n)$. By D1, there is $s_{n-1} \in S_\mathsf{r}$ such that $s_n \,{-}a_i{\rightarrow}\, s_{n-1}$ in L_r and $s_{n-1} \,{-}a_1 \cdots a_{i-1} a_{i+1} \cdots a_n{\rightarrow}\, s$ in L. By the induction assumption, an r-stable r-state is r-reachable from s_{n-1}, and thus from s_n. □

The opposite does not necessarily hold even when adding V, D0, and SV, since $\mathcal{A}(\hat{s}) = \{a\} = V$ satisfies D0, D1, D2, V, and SV on $\circlearrowleft\tau_1 \,||\, \circlearrowleft a$, yielding $L_\mathsf{r} = \circlearrowleft a$.

The system $\circlearrowleft\tau_1 \,||\, \circlearrowleft a$ is always may-progressing, the choice $\mathcal{A}(\hat{s}) = \{\tau_1\}$ is allowed by D0V, D1, D2, and V, and it yields the reduced LTS $\circlearrowleft\tau_1$ that is not always may-progressing. So there is no analogy of Lemma 12 about always may-progressing systems.

Many (but not necessarily all) correct systems have the property that if the system is not given any input (such as sen), then the system eventually stops to wait for input or at least keeps on having the ability to do so. This means that the system is always may-stabilizing. By Lemma 12, its reduced LTS is always may-progressing. As can be seen from Figure 3, the alternating bit protocol is always may-stabilizing. Thanks to the following lemma, we can reason that also the self-synchronizing alternating bit protocol is always may-stabilizing.

Lemma 13. *Consider any LTS. If s is a state from which no stable state is reachable, then s diverges, and so does every state that is reachable from s.*

Proof. Choose $s_0 = s$. Assume we have found $s_0 \,{-}\tau{\rightarrow}\, s_1 \,{-}\tau{\rightarrow}\, s_2 \,{-}\tau{\rightarrow} \cdots {-}\tau{\rightarrow}\, s_n$ in L; this holds for $n = 0$ since then the execution is just s_0. By assumption, there is some $s_n \,{-}\tau{\rightarrow}\, s_{n+1}$. By induction, this constructs a diverging execution. Also by assumption, a state reachable from s cannot reach a stable state, so the second claim follows. □

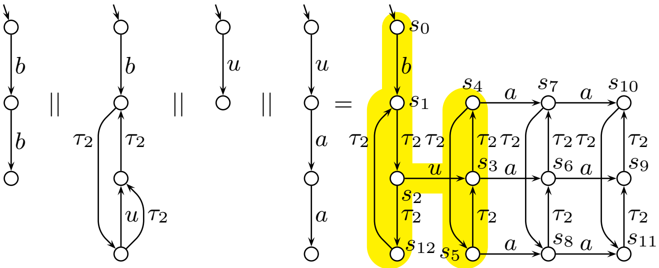


Fig. 16. A system with $V = \{a, b\}$ and $I = \{u, \tau_2\}$ that yields a not always may-progressing reduced LTS

In the self-synchronizing alternating bit protocol, sen, rec, ok, and err are visible. *Of the rest*, let us call $\bar{\mathsf{f}}_0$, $\bar{\mathsf{f}}_1$, and so on *barred*, and f_0, f_1, and so on *unbarred*. In the protocol, the invisible transitions always arise from a barred action synchronizing with its unbarred counterpart. Thus, if we can bound the number of possible unbarred action occurrences in the components after the last visible action, we cannot have a diverging execution. Let n_D and n_A denote the number of D-cells and A-cells.

Let us assume that there is a reachable state of the protocol with a diverging execution. Without the visible transitions, Sender has no cycle with an unbarred transition; along the execution, it can perform at most two unbarred transitions. So let us concentrate on the diverging execution after the last one of those.

In all components, each unbarred transition is followed by a barred or visible one. Thus, the first D-cell can perform at most 1 unbarred action and then blocks, since Sender will not perform such a transition anymore. Similarly, the second D-cell can perform at most 2 unbarred actions, etc., $\mathsf{Receiver}$ at most $n_\mathsf{D} + 1$, the first A-cell at most $n_\mathsf{D} + 2$, etc., and the last A-cell at most $n_\mathsf{D} + n_\mathsf{A} + 1$. By the discussion above, this contradicts our assumption. Hence, the system has no divergence traces.

By $\Delta_\mathsf{r} \subseteq \Delta$, no reduced LTS of the system has divergence traces either. By Lemma 13, L_r will be always may-stabilizing.

Please notice that the user of the D0V-algorithm need not reason like this. It suffices that they run the algorithm and detect that the resulting reduced LTS is always may-progressing.

We now start to investigate the case where the result is not always may-progressing. The reduced LTSs in Figure 14 are not always may-progressing. However, the D0V-algorithm was not used when constructing them. If it is used, then always $\mathcal{A}(s) \subseteq \mathsf{clsr}(s, V)$. Because $\tau_4 \notin \mathsf{clsr}(s, V)$, it yields an LTS that is isomorphic to Figure 7 (c), thus always may-progressing. A similar thing happens with $\circlearrowleft\tau_1 \,||\, \circlearrowleft a$: the D0V-algorithm yields $\circlearrowleft a$ from it.

Figure 16 shows an example where things are not so nice. The D0V-algorithm produces a not always may-progressing (and not trace-preserving) reduced LTS in it. In s_3, the stubborn set computation begins with a.

Because a is visible and enabled, $a \leadsto_{s_3} b$. We have $b \leadsto_{s_3} \tau_2$, because $\mathsf{dis}(s_3, b) = 2$. Furthermore, $\tau_2 \not\leadsto_{s_3} x$ for every action x. So $\mathcal{A}(s_3) = \mathsf{esc}(s_3, V) = \{\tau_2\}$. The algorithm executes $s_3 -\tau_2\rightarrow s_4$. In s_4 we get the same stubborn set for the same reason as in s_3. In the next state s_5 we have $a \leadsto_{s_5} b \leadsto_{s_5} \tau_2$, $b \leadsto_{s_5} u$, and $\tau_2 \leadsto_{s_5} u$. Because $\mathsf{dis}(s_5, u) = 3$, we do not have $u \leadsto_{s_5} x$ for any x. So $\mathcal{A}(s_5) = \mathsf{esc}(s_5, V) = \{u, \tau_2\}$. The action u is disabled, and the execution of τ_2 leads to s_3 that has already been processed. The algorithm terminates ignoring a.

It is obvious with Figure 16 that to preserve the traces, a must be executed in s_3, s_4, or s_5. For the sake of an example, we extend $\mathcal{A}(s_3)$ such that it becomes $\mathsf{clsr}(s_3, a) = \{a, b, \tau_2\}$, and let the D0V-algorithm continue. It constructs the transition $s_3 -a\rightarrow s_6$. It turns out that "$\leadsto_{s_6}$", "$\leadsto_{s_7}$", and "$\leadsto_{s_8}$" are the same as "$\leadsto_{s_3}$", "$\leadsto_{s_4}$", and "$\leadsto_{s_5}$", respectively. Therefore, the algorithm constructs $s_6 -\tau_2\rightarrow s_7 -\tau_2\rightarrow s_8 -\tau_2\rightarrow s_6$ and terminates. If we again intervene and force the algorithm to construct $s_6 -a\rightarrow s_9$, it continues by constructing the last two states. All states of the parallel composition would be constructed.

In this example, the D0V-algorithm computed a terminal strong component only containing occurrences of invisible actions, although an occurrence of a visible action was reachable in the full LTS. Let us call the actions in the stubborn sets of such a component *unproductive*, and the remaining enabled actions *productive*. We also call the component unproductive. In the example, u and τ_2 are unproductive and a is productive. The reduction result was bad, because the occurrence of a did not change the relevant part of the "$\leadsto_s$"-relation, so the enabled unproductive actions were executed again after a.

If a productive action could enable or disable any of the unproductive actions, by D1 and D2 it would not have been ignored. So the only way an occurrence of a productive action can change the relevant part of the "$\leadsto_s$"-relation is by making $\mathsf{dis}(s, u)$ smaller for some disabled unproductive u. This means that if the D0V-algorithm constructs an unproductive terminal strong component, there is a high risk that if SV is made to hold using only ideas in the earlier publications, there will eventually be many copies of the component, so the reduction results will not be good. This is what happened with Figure 16.

A solution to this problem was found very recently [35]. It is based on "freezing" the unproductive actions, that is, computing the stubborn sets of the subsequent states as if the frozen actions did not exist at all. Also [33] presents a freezing technique, but it solves a different problem and is thus not the same. In the latter, the goal was to preserve divergence traces.

10 An Experiment

In this section we discuss analysis experiments on the self-synchronizing alternating bit protocol, using the ASSET tool [31,34]. ASSET does not input parallel compositions of LTSs, but it allows to mimic their behaviour with C++ code. It also allows to express the "$\leadsto_s$" relation in C++ and computes stubborn sets with the $\mathsf{esc}(s, a)$-algorithm, trying each $a \in \bar{\Sigma}$ as a starting point until an enabled action is encountered or $\bar{\Sigma}$ is exhausted. The result obeys D0, D1, D2, and V.

ASSET contains neither a mechanism for ensuring SV nor an implementation of a test whether the result is always may-progressing. However, it can test whether the result is always may-terminating. We used this test to show that the reduced LTS is always may-progressing, by first adding a visible transition to deadlock to each tail state of sen in Sender. Then ASSET verified that the modified protocol passes this test and that all of its deadlocks are due to the added transitions. This shows that the original protocol can always continue to a state where sen is enabled, and is thus always may-progressing. Furthermore, in Section 9 we obtained the same result manually. So Theorem 11 implies that SV holds.

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 in Section 3, including that the protocol cannot lose the ability to deliver data items and reply ok.

Table 1 shows analysis results obtained with a model that is faithful to Figure 4. The experiments were run on a 2.6 GHz Linux laptop with 7 GiB of memory, which is ample memory for the largest experiment. When $c = 8$, the analysis time was 14.3, 0.03, 24.6, and 0.05 seconds, and when $c = 40$ it was 11.7 and 19.2 seconds. ASSET allows to choose whether the actions are scanned forwards or backwards, when used as the a in $\mathsf{esc}(s, a)$. When this setting was changed, the last line ($c = 40$) became 8 632 152, 8 885 544, 27 003 716, and 28 039 718. This suggests that the effect of the order is non-negligible. The effect was in opposite direction with and without the re-sending.

The table shows spectacular reduction results, but one may argue that the model of the channels in Figure 4 is unduely favourable to stubborn sets. The messages travel through the channels step by step. Without stubborn sets, any combination of empty and full channel cells 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

Table 1. Each channel consists of c separate cells

	only one sending				one sending and one re-sending			
	full LTS		stubborn sets		full LTS		stubborn sets	
c	states	edges	states	edges	states	edges	states	edges
1	380	1 068	312	594	822	2 270	640	1 180
2	1 880	6 212	1 030	1 686	3 780	12 210	1 956	3 126
3	9 200	34 934	2 570	3 792	17 318	64 414	4 826	7 018
4	44 000	188 710	5 360	7 354	78 384	330 448	9 736	13 156
5	205 760	983 614	9 946	12 938	350 322	1 650 074	17 898	23 064
6	944 000	4 977 246	16 972	21 208	1 548 668	8 059 068	29 888	36 986
7	4 263 680	24 582 270	27 182	32 928	6 784 206	38 653 782	47 522	57 214
8	19 013 120	119 011 454	41 420	48 962	29 494 824	182 624 072	71 228	83 668
10			85 856	97 928			144 772	164 442
20			970 176	1 028 858			1 537 912	1 629 892
30			4 346 996	4 506 888			6 676 552	6 921 642
40			12 910 316	13 246 018			19 457 692	19 964 692

Table 2. Each channel is a single reduced LTS

	only one sending				one sending and one re-sending			
	full LTS		stubborn sets		full LTS		stubborn sets	
c	states	edges	states	edges	states	edges	states	edges
10	42 680	183 912	15 628	27 652	64 622	273 188	25 772	45 358
20	287 280	1 278 742	85 968	144 542	395 442	1 732 118	134 502	226 298
30	913 880	4 112 572	251 108	410 832	1 208 662	5 361 048	382 432	627 238
40	2 102 480	9 513 402	551 048	886 522	2 720 282	12 143 978	825 562	1 332 178
50	4 033 080	18 309 232	1 025 788	1 631 612	5 146 302	23 064 908	1 519 892	2 425 118
60	6 885 680	31 328 062	1 715 328	2 706 102	8 702 722	39 107 838	2 521 422	3 990 058
70	10 840 280	49 397 892	2 659 668	4 169 992	13 605 542	61 256 768	3 886 152	6 110 998
80	16 076 880	73 346 722	3 898 808	6 083 282	20 070 762	90 495 698	5 670 082	8 871 938
90	22 775 480	104 002 552	5 472 748	8 505 972	28 314 382	127 808 628	7 929 212	12 356 878
100			7 421 488	11 498 062			10 719 542	16 649 818

always immediately moved as close to the front of the channel as possible during reading from and writing to a channel. The fact about fifo queues and the "$\leadsto_s$" relation that was mentioned in Section 7 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. The running times on the line $c = 90$ were 91.5, 11.6, 115.1, and 16.5 seconds. Reversing the action scanning order converted the line $c = 100$ to 4 774 892, 7 538 066, 18 683 266, and 32 716 878. Again, this is a significant decrease without and significant increase with re-sending.

11 Conclusions

We proved that stubborn sets obeying D1, D2, V, and SV yield reduced LTSs that are fair testing equivalent, but not necessarily tree failure equivalent, to the original system. Tree failure equivalence is obtained using C2 instead of V. To our best knowledge, these are the first powerful aps set methods that deal with any reasonable fairness assumption (here the inherent assumption in the equivalences). The results are surprising, because aps set methods have been not so powerful with branching-time properties, and fair testing equivalence preserves the canonical branching-time property "in all futures always, there is a future where eventually a occurs".

We also gave new insight to situations where SV fails because of the ignoring problem. We provided evidence that such situations tend to be rare but nasty. We presented a condition that can be cheaply checked from the reduced LTS. If it holds, then also SV holds and trace / fair testing / tree failure equivalence holds. If it fails, earlier algorithms that enforce SV can be used. Unfortunately, we found a new performance problem with them. This problem has been solved very recently [35].

We also experimented with the method on the self-synchronizing alternating bit protocol. Significant savings were obtained both in the number of constructed states and in the running time.

Acknowledgements. We thank Henri Hansen and the reviewers of the conference and journal version for their comments.

References

1. Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. Optimal dynamic partial order reduction. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384. ACM, 2014.
2. Keith A. Bartlett, Roger A. Scantlebury, and Peter T. Wilkinson. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM*, 12(5):260–261, 1969.
3. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking.* MIT Press, 1999.
4. Javier Esparza and Keijo Heljanko. *Unfoldings - A Partial-Order Approach to Model Checking.* Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2008.
5. Sami Evangelista and Christophe Pajault. Solving the ignoring problem for partial order reduction. *STTT*, 12(2):155–170, 2010.
6. J. Eve and Reino Kurki-Suonio. On computing the transitive closure of a relation. *Acta Inf.*, 8:303–314, 1977.
7. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005.
8. Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000.
9. Rob Gerth, Ruurd Kuiper, Doron A. Peled, and Wojciech Penczek. A partial order approach to branching time logic model checking. In *Third Israel Symposium on Theory of Computing and Systems, ISTCS 1995, Tel Aviv, Israel, January 4-6, 1995, Proceedings*, pages 130–139. IEEE Computer Society, 1995.
10. Patrice Godefroid. Using partial orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer-Aided Verification, Proceedings of a DIMACS Workshop 1990, New Brunswick, New Jersey, USA, June 18-21, 1990*, volume 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 321–340. DIMACS/AMS, 1990.
11. Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science.* Springer, 1996.
12. Harmen Kastenberg and Arend Rensink. Dynamic partial order reduction using probe sets. In Franck van Breugel and Marsha Chechik, editors, *CONCUR 2008 - Concurrency Theory, 19th International Conference, CONCUR 2008, Toronto, Canada, August 19-22, 2008. Proceedings*, volume 5201 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2008.
13. Alfons Laarman, Elwin Pater, Jaco van de Pol, and Henri Hansen. Guard-based partial-order reduction. *STTT*, 18(4):427–448, 2016.
14. Steven Lauterburg, Rajesh K. Karmani, Darko Marinov, and Gul Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In David S. Rosenblum and Gabriele Taentzer, editors, *Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6013 of *Lecture Notes in Computer Science*, pages 308–322. Springer, 2010.
15. Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems - specification.* Springer, 1992.
16. Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986.
17. Kenneth L. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In Gregor von Bochmann and David K. Probst, editors, *Computer Aided Verification, Fourth International Workshop, CAV '92, Montreal, Canada, June 29 - July 1, 1992, Proceedings*, volume 663 of *Lecture Notes in Computer Science*, pages 164–177. Springer, 1992.
18. Robin Milner. *Communication and concurrency.* PHI Series in computer science. Prentice Hall, 1989.
19. Doron A. Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.
20. Doron A. Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In Doron A. Peled, Vaughan R. Pratt, and Gerard J. Holzmann, editors, *Partial Order Methods in Verification, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, July 24-26, 1996*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 233–258. DIMACS/AMS, 1996.
21. Arend Rensink and Walter Vogler. Fair testing. *Inf. Comput.*, 205(2):125–198, 2007.
22. César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based partial order reduction. In Luca Aceto and David de Frutos-Escrig, editors, *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, volume 42 of *LIPIcs*, pages 456–469. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
23. A. W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2010.
24. Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
25. Antti Valmari. Error detection by reduced reachability graph generation. In *Proceedings of the 9th European Workshop on Application and Theory of Petri Nets*, pages 95–122, 1988.

26. Antti Valmari. State space generation: Efficiency and practicality. Tampere University of Technology Publications 55, 1988. Dr. Techn. Thesis.
27. Antti Valmari. Alleviating state explosion during verification of behavioural equivalence. Technical report, Department of Computer Science, University of Helsinki, Helsinki, Finland, 1992. Report A-1992-4.
28. Antti Valmari. The state explosion problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1996.
29. Antti Valmari. Stubborn set methods for process algebras. In Doron A. Peled, Vaughan R. Pratt, and Gerard J. Holzmann, editors, *Partial Order Methods in Verification, Proceedings of a DIMACS Workshop, Princeton, New Jersey, USA, July 24-26, 1996*, volume 29 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 213–232. DIMACS/AMS, 1996.
30. Antti Valmari. On constructibility and unconstructibility of LTS operators from other LTS operators. *Acta Inf.*, 52(2-3):207–234, 2015.
31. Antti Valmari. A state space tool for concurrent system models expressed in C++. In Jyrki Nummenmaa, Outi Sievi-Korte, and Erkki Mäkinen, editors, *Proceedings of the 14th Symposium on Programming Languages and Software Tools (SPLST'15), Tampere, Finland, October 9-10, 2015.*, volume 1525 of *CEUR Workshop Proceedings*, pages 91–105. CEUR-WS.org, 2015.
32. Antti Valmari. The congruences below fair testing with initial stability. In Jörg Desel and Alex Yakovlev, editors, *16th International Conference on Application of Concurrency to System Design, ACSD 2016, Torun, Poland, June 19-24, 2016*, pages 25–34. IEEE Computer Society, 2016.
33. Antti Valmari. More stubborn set methods for process algebras. In Thomas Gibson-Robinson, Philippa J. Hopcroft, and Ranko Lazic, editors, *Concurrency, Security, and Puzzles - Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday*, volume 10160 of *Lecture Notes in Computer Science*, pages 246–271. Springer, 2017.
34. Antti Valmari. Stop it, and be stubborn! *ACM Trans. Embedded Comput. Syst.*, 16(2):46:1–46:26, 2017.
35. Antti Valmari. Stubborn sets with frozen actions. In Matthew Hague and Igor Potapov, editors, *Reachability Problems, 11th International Workshop, RP 2017*, volume 10506 of *Lecture Notes in Computer Science*, pages 160–175, 2017.
36. Antti Valmari and Henri Hansen. Can stubborn sets be optimal? *Fundam. Inform.*, 113(3-4):377–397, 2011.
37. Antti Valmari and Henri Hansen. Stubborn set intuition explained. In Lawrence Cabac, Lars Michael Kristensen, and Heiko Rölke, editors, *Proceedings of the International Workshop on Petri Nets and Software Engineering 2016, including the International Workshop on Biological Processes & Petri Nets 2016 co-located with the 37th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2016 and the 16th International Conference on Application of Concurrency to System Design ACSD 2016, Toruń, Poland, June 20-21, 2016.*, volume 1591 of *CEUR Workshop Proceedings*, pages 213–232. CEUR-WS.org, 2016.
38. Antti Valmari, Konsta Karsisto, and Manu Setälä. Visualisation of reduced abstracted behaviour as a design tool. In *4th Euromicro Workshop on Parallel and Distributed Processing (PDP '96), January 24-26, 1996, Portugal*, pages 187–195. IEEE Computer Society, 1996.
39. Antti Valmari and Walter Vogler. Fair testing and stubborn sets. In Dragan Bosnacki and Anton Wijs, editors, *Model Checking Software - 23rd International Symposium, SPIN 2016, Co-located with ETAPS 2016, Eindhoven, The Netherlands, April 7-8, 2016, Proceedings*, volume 9641 of *Lecture Notes in Computer Science*, pages 225–243. Springer, 2016.
40. Walter Vogler. *Modular Construction and Partial Order Semantics of Petri Nets*, volume 625 of *Lecture Notes in Computer Science*. Springer, 1992.