Stubborn Sets, Frozen Actions, and Fair Testing

Antti Valmari [C]
Faculty of Information Technology, University of Jyväskylä
P.O. Box 35 (Ag C416.2), FI-40014 University of Jyväskylä, Jyväskylä, Finland
antti.valmari@jyu.fi

Walter Vogler
Institut für Informatik, University of Augsburg
D-86135 Augsburg, Germany
walter.vogler@informatik.uni-augsburg.de

Abstract. Many partial order methods use some special condition for ensuring that the analysis is not terminated prematurely. In the case of stubborn set methods for safety properties, implementation of the condition is usually based on recognizing the terminal strong components of the reduced state space and, if necessary, expanding the stubborn sets used in their roots. In an earlier study it was pointed out that if the system may execute a cycle consisting of only invisible actions and that cycle is concurrent with the rest of the system in a non-obvious way, then the method may be fooled to construct all states of the full parallel composition. This problem is solved in this study by a method that "freezes" the actions in the cycle. The new method also preserves fair testing equivalence, making it usable for the verification of many progress properties.

Keywords: partial order methods, stubborn sets, safety properties, ignoring problem, fair testing

1. Introduction

Ample set [1, 2, 3], persistent set [4, 5], and stubborn set [6, 7] methods, or *aps* set methods in brief, alleviate state explosion by only firing a subset of enabled actions in each constructed state. Statically

[C]Corresponding author

available information on generalized concurrency and causal dependency between actions is exploited to choose the subsets so that correct answers to analysis questions are obtained. Also the class of analysis questions affects the choice of the subsets. In general, the smaller the class is, the weaker conditions the subsets must satisfy, the better are the chances of finding legal subsets with only few enabled actions, and the better are the reduction results.

In this study we focus on *safety properties*, that is, properties whose counter-examples are finite sequences of actions. We use process-algebraic parlance and thus talk about preserving trace equivalence, but that only means that counter-examples to all safety properties are preserved and fake counter-examples are not introduced. (As almost always in partial order reduction, the set of so-called visible actions must first be chosen appropriately.)

What is important, however, is that the method also preserves so-called fair testing equivalence [8, 9], and this came for free – we have added nothing to the method towards this end. It is widely known that only verifying safety properties often falls short of sufficiently verifying a system, because it fails to detect the errors where the system does not do something that it is expected to do. Therefore, many partial order methods for so-called *liveness properties* have been developed. Unfortunately, they suffer from some problems. Also fair testing equivalence is capable of revealing does-not-do errors. Where its verdict differs from that by liveness, it is sometimes fair testing and not liveness that more appropriately corresponds to the real-life requirement [10]. Therefore, fair testing equivalence is a good addition to the toolbox and deserves at least the attention that has been given to it in the present study. This issue will be discussed in some detail in the conclusions section.

Ample, persistent, and stubborn sets are based on the same overall idea, but differ significantly at a more detailed level. They also differ in the mathematical language used to develop the methods and prove them correct. The differences are discussed extensively in [11, 12].

Excluding the earliest publications, aps set methods are usually described using abstract conditions. Theorems on the correctness of the methods rely on these conditions, instead of on information about how the sets are actually constructed. Then zero or more algorithms are described and proven correct that yield sets that obey the conditions in question. Usually more than one set satisfies the conditions. In particular, usually the set of all (enabled) actions satisfies them. To obtain good reduction results, the algorithms prefer sets with few enabled actions.

As illustrated in [11, 12], in most cases, ample and persistent set conditions are more straightforward and perhaps easier to understand but have less potential for state space reduction than stubborn set conditions. In most cases, ample and persistent set algorithms are simpler than stubborn set algorithms but take more enabled actions to the sets. An important difference is that ample and persistent sets were defined as sets of enabled actions, while stubborn sets may also contain disabled actions. This implies, among other things, that condition 1 of Theorem 8.1 of the present study cannot be expressed naturally in ample and persistent set terminology (even in the absence of the new ideas in the present study). Indeed, the algorithm described after the theorem has only been used with stubborn sets. A similar comment holds on the condition **V** in Section 3, which is provably better than the corresponding ample set condition [11, 12].

Almost all aps set methods need a condition to solve the *ignoring problem* illustrated in Sections 2 and 4. The best conditions that are known to solve the ignoring problem in the case of safety properties are implemented based on recognizing the terminal strong components of the reduced state space [13,

14]. Recently, there has been significant advances in them [15, 16, 17, 11, 12]. Perhaps ironically, when writing [10], it turned out that excluding a somewhat pathological situation, no such condition is needed in the end; and in the pathological situation, even its recently improved forms suffer from a problem. The goal of this study is to illustrate this background and solve the remaining problem.

Ample and persistent set methods do not use terminal strong component conditions, probably because of the following reason. A well-known example (e.g., [11, Fig. 5]) demonstrates that terminal strong component conditions do not necessarily suffice for infinite counter-examples. As a consequence, when the goal is to preserve liveness properties, a stricter condition called the *cycle condition* is usually used. It has been described in [1] and elsewhere, together with a concrete implementation. The cycle condition does not make the terminal strong component conditions useless, because it is much stronger than the latter and thus has less potential for good reduction results. Furthermore, a drawback has been found in its most widely known implementation [18]. It may make it investigate the full state space in a situation that should be easy for aps set methods, that is, when processes do not interact at all [11, 12].

It is perhaps appropriate to mention at this point that while the present submission was being reviewed, an error was published concerning how ample sets are commonly combined to on-the-fly verification [19]. A small counter-example demonstrates that the widely used SPIN model checking tool [20] suffers from the error. Furthermore, a distinct error has been detected by Thomas Neele in how stubborn sets are usually applied to linear temporal logic properties [21]. This error manifests itself only when pushing the limits of aps set methods beyond known algorithms, so it has little practical significance. More detail will be given in Section 3. In both cases, two correct ideas were combined without investigating their interplay carefully enough. Fortunately, neither of these errors affects the present publication.

In this study we introduce a new notion of *frozen actions*. A frozen action is treated as if it does not exist at all. An action is frozen only when it is certain that it is irrelevant for all counter-examples to any safety property. Not all irrelevant actions are frozen, because recent stubborn set methods are quite good in avoiding irrelevant actions in the first place. As will be proven in Section 4, a large class of systems never needs freezing of actions. Unfortunately, as will be illustrated with Fig. 3, the remaining systems may be able to fool earlier stubborn set methods to investigate the same irrelevant actions again and again in subsequent states, with detrimental consequences to reduction. The method in the present study detects this when it has been fooled for the first time. It freezes the corresponding irrelevant actions, and thus does not fall in the same trap again in subsequent states. (It may, however, be fooled again elsewhere in the reduced state space, that is, in states that are not reachable from where the actions were frozen. If that happens, it again freezes actions.)

Unfortunately, the freezing of actions makes it necessary to develop stubborn set theory anew from the beginning. Until now, it sufficed to talk about actions inside and outside a stubborn set. In the present study, there are three kinds of actions: frozen, warm, and others. The stubborn set is the union of the sets of frozen and warm actions. This makes it difficult to discuss earlier work in sufficient detail, because there is a risk of confusion between the old and new notions of stubborn sets. For this reason, even where the purpose is to illustrate or comment on earlier work, almost all definitions, theorems, etc., are presented in the new framework. The few exceptions are clearly marked.

Section 2 introduces the necessary background concepts, including fair testing equivalence. It also

informally explains the intuition behind stubborn sets in general, not yet using frozen actions. In Section 3, the stubborn set method in the presence of frozen actions is developed up to but not including conditions for solving the ignoring problem. The history, intuition, advantages, and disadvantages of such conditions for safety properties is discussed with many examples in Section 4. Also the version used in the present study is presented. It is called **SF**. Section 5 consists of the proofs that the new conditions guarantee that trace quivalence and fair testing equivalence are preserved.

The validity of **SF** is ensured by constructing the reduced state space in a certain order and investigating its terminal strong components. An algorithm for this is presented and proven correct in Section 6. Freezing of actions takes place in this algorithm. Section 7 is dedicated to the proof that frozen actions have the properties that we claim they have. Until Section 8, stubborn and warm sets have only been referred to via abstract conditions. This section presents and proves correct an algorithm that constructs sets with the promised properties. The algorithm is old, but has been adapted to the presence of frozen actions. The conclusions are in Section 9.

Frozen actions resemble the div-sets in [16], but also have important differences. In [16], the goal was to improve the performance when preserving divergenge traces (a notion relevant to liveness errors), and actions could be explicitly thawed when creating new states. The present study deals with safety properties and does not explicitly thaw actions when creating new states. (In both [16] and the present study, implicit thawing is possible in the following sense: action a is frozen in state s, and firing some other action in s leads to a state s' that has been constructed earlier and where a is not frozen.)

This study is based on [17], but is rather a new study than a variant of [17]. This is obvious already by comparing their lengths. In [17], the proofs were sketchy, the result only applied to trace equivalence, and nondeterministic actions were dealt with using an inelegant trick that was designed directly for the formalism in Section 8. This study presents detailed proofs, extends the result to fair testing equivalence, treats nondeterministic actions as first-class citizens, and presents many examples that illustrate the problems in earlier methods that motivated and were solved by the present study.

2. Background

Throughout most of this study, we will only need the following notions to describe systems under model checking: *visible actions*, *invisible actions*, and *state space*. Formal definitions will be presented below, but let us first discuss the intuition.

An *action* is the name of a transition. *Visible actions* are those actions that are directly relevant for the property under verification. For instance, in the case of mutual exclusion, there may be four visible actions: enter_1, leave_1, enter_2, and leave_2, denoting entering and leaving the critical section by clients 1 and 2. A *trace* is the sequence of visible actions arising from a finite (not necessarily complete) execution. (This notion is not the same as Mazurkiewicz traces [22].) We expect all traces of the mutual exclusion system to be prefixes of sequences of the form $(\mathsf{enter}_1\ \mathsf{leave}_1 \mid \mathsf{enter}_2\ \mathsf{leave}_2)^*$. In particular, if the system has a trace ending with $\mathsf{enter}_1\ \mathsf{enter}_2$ or $\mathsf{enter}_2\ \mathsf{enter}_1$, then mutual exclusion is violated. Alternatively, the system may be equipped with an extra piece of code that detects the simultaneous presence of both clients in their critical sections, and then executes a visible action named error. In this case, error may be the only visible action.

Invisible actions are those actions that are not visible. It is customary in process algebras to only use one name τ for invisible actions, but our method will need more names. How to apply our method in the traditional process-algebraic setting will be explained later in this section and in Section 8.

If it is not clear whether an action should be visible or invisible, then it should be declared visible. Having too many visible actions does not endanger the correctness of our method. However, it may be detrimental to reduction results.

Actions, state spaces, and executions. The sets of invisible and visible actions are denoted with I and V, respectively. We have $I \cap V = \emptyset$. The *state space* is a tuple $(S, I, V, \Delta, \hat{s})$, where S is the set of *states*, $\hat{s} \in S$ is the *initial state*, and $\Delta \subseteq S \times (I \cup V) \times S$ is the set of *transitions*. It is thus an edge-labelled directed graph with a distinguished state $\hat{s}$. We do not assume that all states are reachable from $\hat{s}$ (in the familiar sense defined below), because usually the set of reachable states is not known before verification. Typically there are variables, program counters, etc., each of which has a set of possible values; and the set of states is the Cartesian product of these sets.

A *deadlock* is a state that has no outgoing transitions. An action a is *enabled* in state s if and only if there is s' such that $(s, a, s') \in \Delta$; it is *disabled* otherwise. The *set of enabled actions in* s is denoted with $\mathsf{en}(s)$. Therefore, s is a deadlock if and only if $\mathsf{en}(s) = \emptyset$.

A state is *stable* if and only if it has no outgoing transition that is labelled with an invisible action.

By $s_0 \ -a_1\rightarrow \ s_1 \ -a_2\rightarrow \ \ldots \ -a_n\rightarrow \ s_n$ we mean that for every $1 \leq i \leq n$, $(s_{i-1}, a_i, s_i) \in \Delta$. We call it a *path* or an *execution that starts at* s_0. If we use the word "execution" without mentioning the start state, we mean a path that starts at $\hat{s}$. We may also use the prefix "finite", if there is risk of confusion with infinite executions. An *infinite* execution is an infinite path that starts at the given state or the initial state, if the start state is not mentioned. An execution is *complete* if and only if it is infinite or ends in a deadlock. By $s \ -a_1 \cdots a_n\rightarrow \ s'$ we mean that there are $s_0, \ldots, s_n$ such that $s = s_0$, $s_0 \ -a_1\rightarrow \ \ldots \ -a_n\rightarrow \ s_n$, and $s_n = s'$. If there is s' such that $s \ -a_1 \cdots a_n\rightarrow \ s'$, we may also write $s \ -a_1 \cdots a_n\rightarrow$. Infinite paths are denoted with $s_0 \ -a_1\rightarrow \ s_1 \ -a_2\rightarrow \ \ldots$ or $s_0 \ -a_1a_2 \cdots\rightarrow$.

By $|\sigma|$ we denote the length of the string σ. The empty string is denoted with ε, that is, $|\varepsilon| = 0$. Therefore, for every state s we have $s \ -\varepsilon\rightarrow \ s$. To avoid confusion, we declare $\varepsilon \notin I \cup V$.

An action a is *deterministic* if and only if for all states s, s_1, and s_2 such that $s \ -a\rightarrow \ s_1$ and $s \ -a\rightarrow \ s_2$ we have $s_1 = s_2$. The assumption that all actions are deterministic would simplify the present study a lot. However, we will not make that assumption, because typically it does not hold in process algebras.

State s is *reachable from* s'' if and only if there is a finite path that starts at s'' and ends at s. State s is *reachable* if and only if it is reachable from $\hat{s}$. Transition $s \ -a\rightarrow \ s'$ is reachable (from s'') if and only if s is. Occurrence of a is reachable (from s'') if and only if some s is reachable (from s'') such that $s \ -a\rightarrow$. The *reachable part* of the state space $(S, I, V, \Delta, \hat{s})$ is $(S', I, V, \Delta', \hat{s})$, where S' and Δ' are the sets of the reachable states and reachable transitions. Usually it is not explicitly known before verification. Instead, $\hat{s}$ is explicitly known and there is a rule that, given an explicitly known state s and an action a, produces all s' such that $s \ -a\rightarrow \ s'$. (The "$||$" later in this section is an example.) The reachable part can be constructed by letting initially $S' = \{\hat{s}\}$ and $\Delta' = \emptyset$. A reachable state $s \in S'$ is *expanded* by finding all a and s' such that $s \ -a\rightarrow \ s'$, inserting the triples (s, a, s') to Δ', and inserting those of their s' to S' that are not already there. The algorithm expands once each state that is inserted

to S'. To do so, it keeps track of the states that have been inserted to S' but not yet expanded. The state s is the *parent* of s', if and only if s' was first found by firing a transition $s\ {-a\!\rightarrow}\ s'$.

A *strong component* of a directed graph is a maximal set of vertices such that for any two vertices u and v in the set, v is reachable from u. (Then also u is reachable from v.) Strong components constitute a partitioning of the set of vertices. A strong component C is *terminal* if and only if for any $u \in C$ and any v that is reachable from u, also $v \in C$.

Parallel state spaces. Stubborn set ideas are not strictly tied to any particular formalism, but, to present examples, it is useful to have some formalism. We consider systems of the form $L_1 || \cdots || L_N$, where $L_1, \ldots, L_N$ are state spaces $(S_i, I_i, V_i, \Delta_i, \hat{s}_i)$. To keep it clear whether an action is visible or invisible, we assume that for every $1 \le i \le N$ and $1 \le j \le N$, $I_i \cap V_j = \emptyset$. The states of the system are of the form $s = (s_1, \ldots, s_N)$. The system executes an action a such that every L_i that has $a \in I_i \cup V_i$ executes a, and the remaining L_i stand still. More formally, we define $L_1 \,||\, \cdots \,||\, L_N$ as the reachable part of $(S, I, V, \Delta, \hat{s})$, where $S = S_1 \times \cdots \times S_N$, $I = I_1 \cup \cdots \cup I_N$, $V = V_1 \cup \cdots \cup V_N$, $\hat{s} = (\hat{s}_1, \ldots, \hat{s}_N)$, and $(s_1, \ldots, s_N)\ {-a\!\rightarrow}\ (s'_1, \ldots, s'_N)$ (where $-a\!\rightarrow$ represents Δ) if and only if for every $1 \le i \le N$, either $s'_i = s_i$ and $a \notin I_i \cup V_i$, or $s_i\ {-a\!\rightarrow}\ s'_i$ (where $-a\!\rightarrow$ represents Δ_i).

We adopt the convention that if an action does not appear in a drawing representing L_i, then, unless otherwise mentioned, it is not in $I_i \cup V_i$. Furthermore, in drawings, unless otherwise mentioned, a and b are visible; and u, v, and so on are invisible.

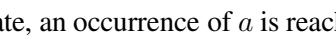
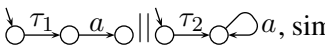
The formalism is, in essence, the same as parallel composition of labelled transition systems in process algebras, with different notation for invisible actions. To make the reading of examples easier, we adopt the convention that τ_i denotes an invisible action executed solely by L_i, that is, $\tau_i \in I_i$ and $\tau_i \notin I_j$ when $j \ne i$. This makes every τ_i behave similarly to the τ in process algebras, the only difference being notational: the subscript records the component that executed the τ. Furthermore, we may write $(L_1 \,||\, \cdots \,||\, L_N) \setminus H$, where H lists the remaining invisible actions, that is, $H = (I_1 \cup \cdots \cup I_N) \setminus \{\tau_1, \ldots, \tau_N\}$. Then "$\setminus H$" can be thought of as the familiar hiding operator in process algebras. For more details on this formalism, please see [14, 16, 15, 10].

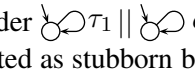
Although stubborn set theory does not rely on the notion of concurrent actions, such a notion is useful in examples and when discussing the intuition. Therefore, we define that *a and b are concurrent* if and only if $\{a, b\} \subseteq I \cup V$ but there is no $1 \le i \le N$ such that $\{a, b\} \subseteq I_i \cup V_i$.

Stubborn set basics. What is explained in this subsection, applies to traditional stubborn set methods. The new idea of frozen actions of the present study will make things more complicated.

Reachable parts of state spaces are often huge. Stubborn set methods compute a *reduced state space* $(S_\mathsf{r}, I, V, \Delta_\mathsf{r}, \hat{s})$ similarly to the computation of the reachable part, but only expand a subset of actions in each state. That is, in each found state $s \in S_\mathsf{r}$, a *stubborn set* $\mathsf{stubb}(s) \subseteq I \cup V$ is computed, and only the transitions $s\ {-a\!\rightarrow}\ s'$ and their end states s' are inserted to Δ_r and S_r that have $a \in \mathsf{stubb}(s)$. For convenience, we use the prefix r- or the subscript $_\mathsf{r}$ to indicate that an entity belongs to the reduced state space. For instance, an r-path is a path in the reduced state space. Obviously $\mathsf{en}_\mathsf{r}(s) \subseteq \mathsf{en}(s)$ but not necessarily vice versa, and every r-path is a path but not necessarily vice versa. We have $\mathsf{en}_\mathsf{r}(s) = \mathsf{en}(s) \cap \mathsf{stubb}(s)$. By definition, $V_\mathsf{r} = V$, $I_\mathsf{r} = I$, and $\hat{s}_\mathsf{r} = \hat{s}$.

The goal is to obtain a smaller state space that can be used for the verification of certain classes of properties. More precisely, the reduced state space contains a counter-example to the property if and only if also the reachable part contains one. In the present study we deal with *stuttering-insensitive safety properties*. A safety property is a property whose counter-examples can always be expressed in terms of finite executions. Stuttering-insensivity means that the number of invisible actions before the first, after the last, and between any two visible actions does not matter. Most, if not all, properties of interest in verification are stuttering-insensitive.

Stubborn set methods exploit the fact that if a and b are concurrent and both are enabled, then they can be executed in either order and the result is the same. Assume that the goal is to check whether from every reachable state, an occurrence of a is reachable. Consider  . It executes first τ_1 or τ_2. Then it executes the other one of τ_1 and τ_2, and then a forever. Typical stubborn set methods fire either only τ_1 or only τ_2 in its initial state, saving one state and two transitions. The above-mentioned property holds both on the full and on the reduced state space. With  , similar saving is obtained. In this case, the property fails on both state spaces, because the system deadlocks.

Now consider  . Let $\hat{s}$ denote its initial state. Because τ_1 and a are concurrent, the set $\{\tau_1\}$ is treated as stubborn by most stubborn set methods. The choice $\mathsf{stubb}(\hat{s}) = \{\tau_1\}$ constructs the r-transition $\hat{s} \ -\tau_1\rightarrow_{\mathsf{r}} \ \hat{s}$, after which all encountered states have been investigated. If the analysis stops here, then a was never fired and the property fails on the reduced state space, although it holds on the full state space. This is known as the *ignoring problem* [13, 18]. To solve it, many stubborn set methods have a special requirement, with names such as "cycle condition" and "terminal strong component condition". We will discuss this issue extensively in Section 4. For that purpose, let *tsr-component* stand for "terminal strong component in the reduced state space".

Trace and fair testing equivalences. If α is a string and A is a set, then let $\alpha - A$ denote the result of the removal of all elements of A from α. That is, $\varepsilon - A = \varepsilon$; and if $a_1 \cdots a_n - A = \sigma$, then $aa_1 \cdots a_n - A = \sigma$ if $a \in A$ and $a\sigma$ if $a \notin A$. The *trace* of a finite sequence of actions $a_1 \cdots a_n$ is $a_1 \cdots a_n - I$, that is, it is obtained by removing the invisible actions. By $s \ =\sigma\Rightarrow \ s'$ we mean that there is ρ such that σ is its trace (that is, $\sigma = \rho - I$) and $s \ -\rho\rightarrow \ s'$. The notation $s \ =\sigma\Rightarrow$ denotes that there is some s' such that $s \ =\sigma\Rightarrow \ s'$. A sequence σ is a trace of state s if and only if $s \ =\sigma\Rightarrow$. The set of traces of a state space L is

$$\mathsf{Tr}(L) \ = \ \{\sigma \mid \hat{s} \ =\sigma\Rightarrow\} \quad .$$

Two state spaces $L_1 = (S_1, I_1, V_1, \Delta_1, \hat{s}_1)$ and $L_2 = (S_2, I_2, V_2, \Delta_2, \hat{s}_2)$ are *trace equivalent* if and only if $V_1 = V_2$ and $\mathsf{Tr}(L_1) = \mathsf{Tr}(L_2)$. The reason for the condition $V_1 = V_2$ is the fact that the set of visible actions plays an important role in the definition of refusals and tree failures presented later in this subsection. Trace equivalence only depends on the reachable parts of the state spaces. To verify a stuttering-insensitive safety property it suffices to know the set of traces of the system. This means that if the reduced state space is trace equivalent to the full state space, then it can be used in the verification of all stuttering-insensitive safety properties. Of course, this nice fact is subject to choosing the set of visible actions so that the property can be expressed in terms of them.

The set of stuttering-insensitive safety properties covers a very big class of properties, but also suffers from one serious drawback: it does not suffice for verifying that the system does something

good, it only suffices for verifying that the system does not do anything bad. Indeed, a system that never executes any visible actions has $\{\varepsilon\}$ as its set of traces, and, therefore, violates no non-trivial safety property. (It does violate the property that says that the system must not have the trace ε, but this is a useless property, because every system violates it.)

Fortunately, the method developed in this study preserves also so-called *fair testing equivalence* [8, 9] – as a welcome side-effect. This implies that it preserves properties of the form "in all futures always, there is a future where eventually a occurs". As a matter of fact, fair testing equivalence is the coarsest equivalence that preserves this property and has the property that if a sub-system is replaced by a fair testing equivalent one, then the system as a whole remains fair testing equivalent [9]. Being the coarsest means having the biggest possible equivalence classes and thus maximizing the possibilities of reducing the state space. This is an advantage in compositional approaches to verification.

Fair testing equivalence can also be thought of as preserving progress properties under the assumption that for every state s that is encountered infinitely many times in an infinite execution, every outgoing transition of s is taken infinitely many times in the execution. Intuitively this means that if we are happy with the assumption that in every choice situation that repeats infinitely often, every possibility is eventually tried, then a full range of stuttering-insensitive progress properties can be verified with our method. This is a somewhat weaker notion of progress than traditionally used in linear temporal logic [23], but is easier to use because the user need not formulate any so-called weak or strong fairness assumptions. Definitely it is much better than no progress verification at all.

The downside is that the definition of fair testing equivalence is complicated. A state s *refuses* a sequence σ of visible actions if and only if s does not have σ as a trace. A state s refuses $K \subseteq V^*$ if and only if s refuses every element of K. The sequence ε cannot be refused, but the set $\emptyset$ can. A *tree failure* is a pair $(\sigma, K) \in V^* \times 2^{V^*}$ such that there is s for which $\hat{s} =\sigma\Rightarrow s$ and s refuses K. If $K \subseteq V^*$ and $\pi \in V^*$, we define $\pi^{-1}K = \{\rho \mid \pi\rho \in K\}$. We say that π is a *prefix* of K if and only if $\pi^{-1}K \neq \emptyset$. Two systems L_1 and L_2 are fair testing equivalent if and only if the following hold:

1. $V_1 = V_2$.
2. For every tree failure (σ, K) of L_1, either (σ, K) is also a tree failure of L_2, or there is a prefix π of K such that $(\sigma\pi, \pi^{-1}K)$ is a tree failure of L_2.
3. For every tree failure (σ, K) of L_2, either (σ, K) is also a tree failure of L_1, or there is a prefix π of K such that $(\sigma\pi, \pi^{-1}K)$ is a tree failure of L_1.

If $K \neq \emptyset$, then (σ, K) can be represented in the form $(\sigma\pi, \pi^{-1}K)$ where π is a prefix of K, by choosing $\pi = \varepsilon$. However, this does not work if $K = \emptyset$, because then ε is not a prefix of K. This is why parts 2 and 3 have the either-part.

A system has the trace σ if and only if $(\sigma, \emptyset)$ is its tree failure. As a consequence, fair testing equivalence implies trace equivalence.

3. Local Conditions

Intuition of earlier methods. To understand the motivation of the definitions later in this section, let us first discuss the intuition behind earlier stubborn set methods for trace equivalence, fair testing

equivalence, and, indeed, most stubborn set methods.

Assume that $s_0 \in S_\mathsf{r}$ and $s_0 =\sigma\Rightarrow$. We want $s_0 =\sigma\Rightarrow_\mathsf{r}$. If $\sigma = \varepsilon$, then obviously $s_0 =\sigma\Rightarrow_\mathsf{r} s_0$. From now on we assume $\sigma \neq \varepsilon$. There is a path $s_0 \ -a_1\rightarrow\ s_1 \ -a_2\rightarrow\ \ldots\ -a_n\rightarrow\ s_n$ such that $n > 0$ and $\sigma = a_1 \cdots a_n - I$. Because we do not want to construct the full state space, we might have $a_1 \notin \mathsf{en}_\mathsf{r}(s_0)$. Earlier stubborn set methods deal with this problem with the following strategy. The cases have been numbered in reverse order, to match the numbering of conditions that will be presented later.

2. The set $\mathsf{en}_\mathsf{r}(s_0)$ may contain an invisible action a that is concurrent with $a_1, \ldots, a_n$ or commutes with them for some other reason (for instance, a_1 may read from and a write to a fifo that is neither empty nor full in s_0, and they do not access other variables in common). Then there are s'_0 and s'_n such that $s_0 \ -a\rightarrow\ s'_0$, $s_n \ -a\rightarrow\ s'_n$, and $s'_0 \ -a_1 \cdots a_n\rightarrow\ s'_n$. We have $s_0 =\varepsilon\Rightarrow_\mathsf{r} s'_0 =\sigma\Rightarrow s'_n$ and $s_n =\varepsilon\Rightarrow s'_n$. So we have constructed an invisible transition in the reduced state space such that after it, the counter-example is still available. If the stubborn set method can guarantee $s'_0 =\sigma\Rightarrow_\mathsf{r}$ (which it can), then we get $s_0 =\sigma\Rightarrow_\mathsf{r}$. However, the path $s'_0 \ -a_1 \cdots a_n\rightarrow\ s'_n$ is of the same length as the original path $s_0 \ -a_1 \cdots a_n\rightarrow\ s_n$, so we are not any closer to the goal than we were originally.

1. Stubborn set construction algorithms are designed so that if (but not necessarily only if) the previous case does not apply, then for some $1 \leq i \leq n$, $a_i \in \mathsf{en}_\mathsf{r}(s_0)$ and a_i commutes with $a_1, \ldots, a_{i-1}$. (How this is obtained will be discussed in Section 8.) In this case there is s'_0 such that $s_0 \ -a_i\rightarrow_\mathsf{r}\ s'_0 \ -a_1 \cdots a_{i-1}\rightarrow\ s_i \ -a_{i+1} \cdots a_n\rightarrow\ s_n$. If $a_i \in I$, then $s_0 =\varepsilon\Rightarrow_\mathsf{r} s'_0 =\sigma\Rightarrow\ s_n$. Stubborn set construction algorithms may be designed to also guarantee that if $a_i \in V$, then a_1, $\ldots$, a_{i-1} are invisible. Then $s_0 =a_i\Rightarrow_\mathsf{r} s'_0 =\varepsilon\Rightarrow\ s_i =\rho\Rightarrow\ s_n$, where ρ is the string such that $\sigma = a_i\rho$. So in both cases we would get $s_0 =\sigma\Rightarrow_\mathsf{r}$, if we could guarantee $s'_0 =\sigma'\Rightarrow_\mathsf{r}$, where $\sigma' = \sigma$ if $a_i \in I$ and $\sigma' = \rho$ otherwise. What is more, the path that yields $s'_0 =\sigma'\Rightarrow$ is shorter than the original path $s_0 \ -a_1 \cdots a_n\rightarrow$. We say that this case *consumed* a_i.

That is, in each r-state, either case 1 or case 2 is available. Case 2 brings us neither closer to nor further from the goal $s_0 =\sigma\Rightarrow_\mathsf{r}$, while case 1 brings us closer to it. As a consequence, if we somehow ensure that case 1 always eventually applies, we get an induction proof that $s_0 =\sigma\Rightarrow_\mathsf{r}$. Unfortunately, ensuring this is far from trivial. The possibility of having an infinite sequence of case 2 without case 1 is how the ignoring problem that was mentioned in Section 2 emerges in proofs. This problem will be solved in Sections 4 and 6.

To prove that the reduced state space is fair testing equivalent to the full state space, proving just $s_0 =\sigma\Rightarrow_\mathsf{r} s'$ for some s' does not suffice. In the proof, we will also need $s_n =\varepsilon\Rightarrow s'$. Fortunately, the above construction has this property, because in case 1 $s'_0 =\sigma'\Rightarrow\ s_n$, and in case 2 $s_n \ -a\rightarrow\ s'_n$ with $a \in I$.

Frozen actions. In the new stubborn set method of the present study, each r-state s has an associated set $\mathsf{frozen}(s)$ of *frozen actions*. We will define stubborn sets such that for every $s \in S_\mathsf{r}$, $\mathsf{frozen}(s) \subseteq \mathsf{stubb}(s)$, and define $\mathsf{warm}(s) = \mathsf{stubb}(s) \setminus \mathsf{frozen}(s)$. Only the elements in $\mathsf{warm}(s)$ are used for constructing transitions in the reduced state space. That is, frozen actions are not taken into account

when constructing reduced state spaces. However, the sets $\mathsf{stubb}(s)$ and $\mathsf{frozen}(s)$ may grow, as will be described in Section 6. (They may not shrink.) If $\mathsf{frozen}(s)$ grows, then a new $\mathsf{warm}(s)$ is immediately computed, yielding a new $\mathsf{stubb}(s)$. Therefore, $\mathsf{frozen}(s) \subseteq \mathsf{stubb}(s)$ always holds, although the values of $\mathsf{stubb}(s)$ and $\mathsf{frozen}(s)$ may change. Note that, after they have changed, the reduced state space may contain some transitions $s\ {-a\!\rightarrow_{\mathsf{r}}}\ s'$ such that $a \in \mathsf{frozen}(s)$. In such a situation, a was warm at some earlier time. At the end of the construction of the reduced state space, $\mathsf{warm}(s) \cap \mathsf{en}(s) \subseteq \mathsf{en}_{\mathsf{r}}(s) \subseteq \mathsf{stubb}(s) \cap \mathsf{en}(s)$ holds for every $s \in S_{\mathsf{r}}$.

The set of frozen actions of the initial state $\hat{s}$ is originally empty. When a new r-state $s' \neq \hat{s}$ is constructed, it inherits the currently frozen actions of its parent state. However, this does not necessarily imply $\mathsf{frozen}(s) \subseteq \mathsf{frozen}(s')$ for every $s\ {-a\!\rightarrow_{\mathsf{r}}}\ s'$, because s' may have been first found via some other transition or may be $\hat{s}$, and because $\mathsf{frozen}(s)$ may grow afterwards.

We will prove in Section 7 that $\mathsf{frozen}(s)$ (both the final one and every earlier version) has the following two properties. As we will see later in detail, they imply that if $s \in S_{\mathsf{r}}$ and $s\ {=\sigma\Rightarrow}\ s'$, frozen actions are not needed to obtain an r-path $s\ {=\sigma\Rightarrow_{\mathsf{r}}}\ s''$ such that $s'\ {=\varepsilon\Rightarrow}\ s''$. **F1** says that for every path that starts at s, there is a path with the same start state and sequence of unfrozen (in s) actions, that has no frozen (in s) actions. Furthermore, the end state of the latter path can be reached from the end state of the former path via frozen (in s) actions. **F2** says that frozen visible actions cannot become enabled.

F1 If $s \in S_{\mathsf{r}}$, $s\ {-a_1 \cdots a_n\!\rightarrow}\ s'$, and $b_1 \cdots b_m = a_1 \cdots a_n - \mathsf{frozen}(s)$, then there are $s'' \in S$ and $\gamma \in \mathsf{frozen}(s)^*$ such that $s\ {-b_1 \cdots b_m\!\rightarrow}\ s''$ and $s'\ {-\gamma\!\rightarrow}\ s''$.

F2 If $s \in S_{\mathsf{r}}$, $s\ {-a_1 \cdots a_n\!\rightarrow}\ s'$, and $a_n \in \mathsf{frozen}(s)$, then $a_n \in I$.

These conditions were designed so that they can be implemented as will be discussed in Section 6, and that they yield the following lemma.

Lemma 3.1. Assume **F1** and **F2**. If $s \in S_{\mathsf{r}}$ and $s\ {=\sigma\Rightarrow}\ z$, then there is z' such that $s\ {=\sigma\Rightarrow}\ z'$, $z\ {=\varepsilon\Rightarrow}\ z'$, and the path that yields $s\ {=\sigma\Rightarrow}\ z'$ contains no actions from $\mathsf{frozen}(s)$ and is either shorter than or the same path as the path that yields $s\ {=\sigma\Rightarrow}\ z$.

Proof:
If $s\ {=\sigma\Rightarrow}\ z$ contains no frozen actions, then the claim holds trivially with $z' = z$. Otherwise **F1** gives most of the claim. We still have to show that the elements of γ and $\{a_1, \ldots, a_n\} \setminus \{b_1, \ldots, b_m\}$ are invisible. This follows from **F2**, because $s\ {-a_1 \cdots a_n\!\rightarrow}\ s'\ {-\gamma\!\rightarrow}$. □

We say that the actions in $\mathsf{frozen}(s)$ were *consumed* from the action sequence $a_1 \cdots a_n$ of the path $s\ {=\sigma\Rightarrow}\ z$. We now have two notions of consuming actions. They will be used in induction proofs, to demonstrate that excluding the base case of the induction, given a path with certain properties, a shorter path with the mentioned properties exists.

New local stubborn set conditions. Traditional stubborn set methods rely on conditions called **D1** and **D2** that facilitate reasoning of the kind in cases 1 and 2 of the above strategy. They describe

how generalized concurrency and independency relations are exploited. We replace them by slightly more complicated conditions **D1F** and **D2F** that take frozen actions into account. There also is a third condition **D3F** that will be used in the correctness proof of the algorithm that guarantees **F1** and **F2**. The implementation of **D1F**, **D2F**, and **D3F** will be discussed in Section 8. In all of the following conditions, we assume that

$$\mathsf{stubb}(s_0) \subseteq I \cup V,\ a \in \mathsf{warm}(s_0), \text{ and } a_1, \ldots, a_n \text{ are not in } \mathsf{stubb}(s_0).$$

D1F If $s_0 -a_1\rightarrow s_1 -a_2\rightarrow \ldots -a_n\rightarrow s_n -a\rightarrow s'_n$, then there are $s'_0, \ldots, s'_{n-1}$ such that $s'_0 -a_1\rightarrow s'_1 -a_2\rightarrow \ldots -a_n\rightarrow s'_n$ and for $0 \leq i < n$ we have $s_i -a\rightarrow s'_i$.

D2F If $s_0 -a_1\rightarrow s_1 -a_2\rightarrow \ldots -a_n\rightarrow s_n$ and $s_0 -a\rightarrow s'_0$, then there are $s'_1, \ldots, s'_n$ such that $s'_0 -a_1\rightarrow s'_1 -a_2\rightarrow \ldots -a_n\rightarrow s'_n$ and for $1 \leq i \leq n$ we have $s_i -a\rightarrow s'_i$.

D3F If $s_0 -a_1 \cdots a_n\rightarrow s_n -a\rightarrow s'_n$, $s_0 -a_1 \cdots a_n\rightarrow z_n -a\rightarrow z'_n$, and $s_n \neq z_n$, then $s'_n \neq z'_n$.

If all actions are deterministic, then **D3F** can be forgotten since it holds vacuously, and **D1F** and **D2F** can be replaced by much simpler conditions given in the following theorem:

Theorem 3.2. If all actions are deterministic, then the following conditions imply **D1F** and **D2F**.

- If $s_0 -a_1 \cdots a_n a\rightarrow s'_n$, then $s_0 -a a_1 \cdots a_n\rightarrow s'_n$.
- If $s_0 -a_1 \cdots a_n\rightarrow s_n$ and $s_0 -a\rightarrow$, then $s_n -a\rightarrow$.

Proof:
Assume the if-part of **D2F**. For each $0 \leq i \leq n$, an application of the second condition to $s_0 -a_1 \cdots a_i\rightarrow s_i$ yields an s''_i such that $s_i -a\rightarrow s''_i$. Then the first condition yields $s_0 -a a_1 \cdots a_i\rightarrow s''_i$. Let $z_n = s''_n$. An application of the first condition to $s_0 -a_1 \cdots a_n a\rightarrow z_n$ yields $z_0, \ldots, z_{n-1}$ such that $s_0 -a\rightarrow z_0 -a_1\rightarrow z_1 -a_2\rightarrow \ldots -a_n\rightarrow z_n$, implying $s_0 -a a_1 \cdots a_i\rightarrow z_i$. Because both s''_i and z_i are reachable from s_0 via $a a_1 \cdots a_i$, by determinism $s''_i = z_i$ and $z_0 = s'_0$. Therefore, the then-part of **D2F** holds.

Now assume the if-part of **D1F**. An application of the first condition to $s_0 -a_1 \cdots a_n a\rightarrow s'_n$ yields $s_0 -a\rightarrow$, returning the situation to the previous case. By determinism, the s'_n provided by **D2F** is the same state as the s'_n assumed by **D1F**. □

The traditional **D1** and **D2** are obtained by choosing $\mathsf{frozen}(s_0) = \emptyset$ and dropping the requirement $s_i -a\rightarrow s'_i$ for $0 < i < n$. In Section 7 an example will be presented that illustrates the necessity of this requirement in the presence of frozen actions. That this requirement is needed is not a big drawback, because the previous theorem tells that if all actions are deterministic, then the issue does not make a difference; and the standard approach to constructing stubborn sets of nondeterministic actions (Theorem 8.1) automatically guarantees $s_i -a\rightarrow s'_i$ for $0 < i < n$.

Thomas Neele observed recently that $s_i -a\rightarrow s'_i$ for $0 < i < n$ is also needed when stubborn sets are applied to linear temporal logic [21]. There is a set of atomic propositions and a mapping that, for each state, tells which atomic propositions hold and which do not hold on the state. When moving

from a state to the next, zero or more atomic propositions change their truth values. Methods for linear temporal logic aim at preserving the sequences of these changes, except that empty sets of changes play the role of the invisible actions and may be added or removed at will during the reduction. Neele pointed out that if $s_i \ -a\rightarrow \ s'_i$ for $0 < i < n$ is not assumed, then the sequence of changes caused by a pair a_1a_2 of visible actions may depend on whether the pair is executed before or after an invisible action a. He constructed a counter-example together with Valmari.

The distinction between invisible and visible actions is taken care of by a condition that says that if the stubborn set contains an unfrozen enabled visible action, then it must contain all visible actions.

VF If $\mathsf{warm}(s) \cap \mathsf{en}(s) \cap V \neq \emptyset$, then $V \subseteq \mathsf{stubb}(s)$.

The traditional **V** says that if $\mathsf{stubb}(s) \cap \mathsf{en}(s) \cap V \neq \emptyset$, then $V \subseteq \mathsf{stubb}(s)$. (Please see [11, 12] for why **V** has potential for better reduction results than the corresponding condition **C2** in ample set theory.)

All the conditions discussed above are satisfied by the choice $\mathsf{warm}(s) = \emptyset$ (that is, $\mathsf{stubb}(s) = \mathsf{frozen}(s)$). However, using $\emptyset$ as $\mathsf{warm}(\hat{s})$ results in a reduced state space that consists of one state and no transitions. Although it sometimes does preserve trace and fair testing equivalences, usually it does not. We will introduce in Section 4 a condition called **SF** that solves this problem. We will also discuss another, simpler condition that often but not always obtains the same goal. It is called **D0VF**.

D0VF For every $s \in S_{\mathsf{r}}$, either $\mathsf{warm}(s) \cap \mathsf{en}(s) \neq \emptyset$ or $V \subseteq \mathsf{stubb}(s)$.

We will later prove that if $V \subseteq \mathsf{stubb}(s)$ and $\mathsf{warm}(s) \cap \mathsf{en}(s) = \emptyset$, then occurrences of visible actions cannot be reached from s. Therefore, **D0VF** ensures that either s has outgoing transitions in the reduced state space, or the future of s is certainly irrelevant from the point of view of trace and fair testing equivalences.

We call **D0VF**, **D1F**, **D2F**, **D3F**, and **VF** *local conditions*, because they are of the form $\forall s \in S_{\mathsf{r}} : \varphi(s)$, where $\varphi(s)$ refers to only one r-state. The condition **SF** that we will introduce in the next section is not local.

The next lemma says that there always is a set that satisfies these conditions.

Lemma 3.3. The set $\mathsf{stubb}(s) = I \cup V$ satisfies **D0VF**, **D1F**, **D2F**, **D3F**, and **VF** in s.

Proof:
D1F, **D2F**, and **D3F** hold, because the set from which their $a_1, \ldots, a_n$ must be picked is empty, forcing $n = 0$. The then-part of **VF** and latter part of **D0VF** hold trivially. □

A subset of the conditions allows us to prove also the following two lemmas.

Lemma 3.4. Assume **F1**, **F2**, **D1F**, **D2F**, and **VF**. Assume that $\sigma \neq \varepsilon$, $s_0 \ =\sigma\Rightarrow \ z_0$, $s_0 \ -b_1\rightarrow_{\mathsf{r}} \ s_1 \ -b_2\rightarrow_{\mathsf{r}} \ldots \ -b_m\rightarrow_{\mathsf{r}} \ s_m$, and $V \subseteq \mathsf{stubb}(s_m)$. Then there is i such that $0 \leq i \leq m$, $\{b_1, \ldots, b_i\} \subseteq I$, and there are s'_i, z'_i, b, and σ' such that $s_i \ -b\rightarrow_{\mathsf{r}} \ s'_i \ =\sigma'\Rightarrow \ z'_i$, $z_0 \ =\varepsilon\Rightarrow \ z'_i$, and the path that yields $s'_i \ =\sigma'\Rightarrow \ z'_i$ is shorter than the path that yields $s_0 \ =\sigma\Rightarrow \ z_0$, where either $b \in I$ and $\sigma' = \sigma$ or $b \in V$ and $b\sigma' = \sigma$.

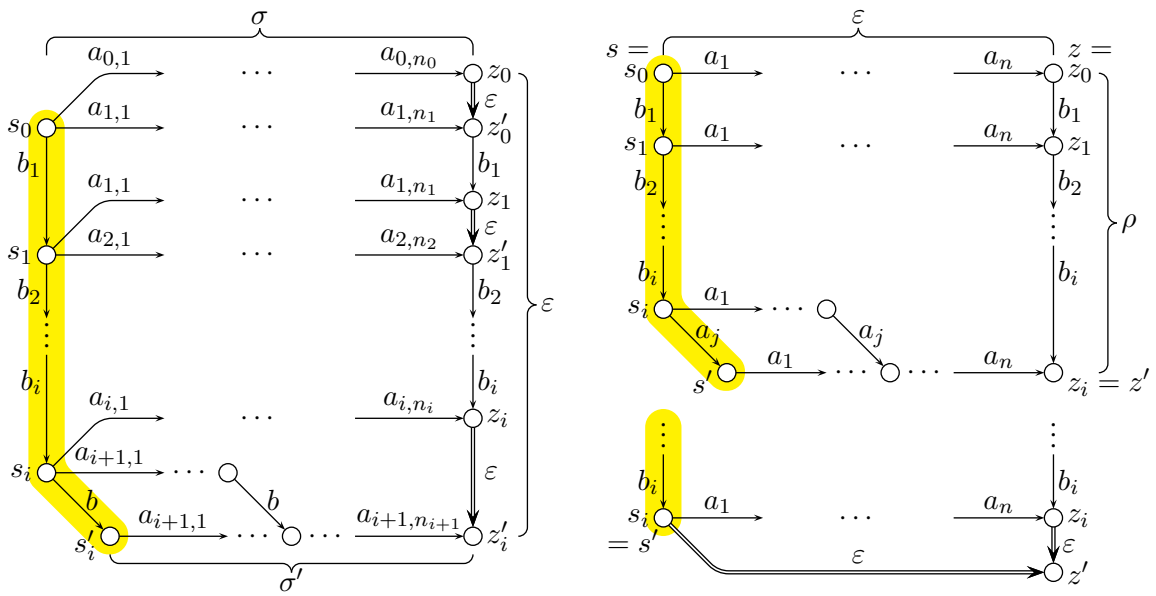


Figure 1. (left) Illustrating Lemma 3.4 and its proof. **D2F** is applied to $s_0 \ -b_1\rightarrow_{\mathsf{r}} \ s_1$ and **D1F** is applied to $s_i \ -a_{i+1,1}\cdots a_{i+1,n_{i+1}}\rightarrow \ z'_i$ (right) Illustrating Lemma 3.5 and its proof. **D1F** is applied on top and Lemma 3.1 is applied at bottom. States and transitions on yellow background are in the reduced state space

Proof:

Let $s_0 \ -a_{0,1}\cdots a_{0,n_0}\rightarrow \ z_0$ be the path causing $s_0 \ =\sigma\Rightarrow \ z_0$. The proof applies induction along the path $s_0 \ -b_1\rightarrow_{\mathsf{r}} \ s_1 \ -b_2\rightarrow_{\mathsf{r}} \ \ldots \ -b_m\rightarrow_{\mathsf{r}} \ s_m$ as long as possible. At each step we have $b_1\cdots b_i \in I^*$, $z_0 \ =\varepsilon\Rightarrow \ z_1 \ =\varepsilon\Rightarrow \ \ldots \ =\varepsilon\Rightarrow \ z_i$, and $s_i \ -a_{i,1}\cdots a_{i,n_i}\rightarrow \ z_i$, where $n_i \leq n_{i-1} \leq \ldots \leq n_0$ and $a_{i,1}\cdots a_{i,n_i} - I = \sigma$.

If $i < m$, we use the values of $\mathsf{warm}(s_i)$ and $\mathsf{frozen}(s_i)$ at the time when $s_i \ -b_{i+1}\rightarrow_{\mathsf{r}} \ s_{i+1}$ was constructed, and otherwise we use the final values of $\mathsf{warm}(s_m)$ and $\mathsf{frozen}(s_m)$. By Lemma 3.1 there are z'_i and a path $s_i \ -a_{i+1,1}\cdots a_{i+1,n_{i+1}}\rightarrow \ z'_i$ such that $n_{i+1} \leq n_i$, $a_{i+1,1}\cdots a_{i+1,n_{i+1}} - I = \sigma$, $z_i \ =\varepsilon\Rightarrow \ z'_i$, and $\{a_{i+1,1},\ldots,a_{i+1,n_{i+1}}\} \cap \mathsf{frozen}(s_i) = \emptyset$.

Let first $\{a_{i+1,1},\ldots,a_{i+1,n_{i+1}}\} \cap \mathsf{warm}(s_i) = \emptyset$. Because none of the $a_{i+1,j}$ is in $\mathsf{frozen}(s_i)$, none of them is in $\mathsf{stubb}(s_i)$. By $\sigma \neq \varepsilon$ at least one of them is visible, so $V \not\subseteq \mathsf{stubb}(s_i)$, implying $i < m$. Therefore, **D2F** can be applied to $s_i \ -b_{i+1}\rightarrow_{\mathsf{r}} \ s_{i+1}$, yielding z_{i+1} such that $z'_i \ -b_{i+1}\rightarrow \ z_{i+1}$ and $s_{i+1} \ -a_{i+1,1}\cdots a_{i+1,n_{i+1}}\rightarrow \ z_{i+1}$. **VF** and $V \not\subseteq \mathsf{stubb}(s_i)$ imply $b_{i+1} \notin V$. So $z_i \ =\varepsilon\Rightarrow \ z'_i \ =\varepsilon\Rightarrow \ z_{i+1}$. We have taken a step along the path $s_0 \ -b_1\rightarrow_{\mathsf{r}} \ s_1 \ -b_2\rightarrow_{\mathsf{r}} \ \ldots \ -b_m\rightarrow_{\mathsf{r}} \ s_m$.

Otherwise $\{a_{i+1,1},\ldots,a_{i+1,n_{i+1}}\} \cap \mathsf{warm}(s_i) \neq \emptyset$. This happens at the latest when $i = m$. We let j be the smallest such that $a_{i+1,j} \in \mathsf{warm}(s_i)$ and choose $b = a_{i+1,j}$. Then **D1F** yields an s'_i such that

$$s_i \ -b\rightarrow_{\mathsf{r}} \ s'_i \ -a_{i+1,1}\cdots a_{i+1,j-1}a_{i+1,j+1}\cdots a_{i+1,n_{i+1}}\rightarrow \ z'_i \ .$$

Let $\sigma' = a_{i+1,1}\cdots a_{i+1,j-1}a_{i+1,j+1}\cdots a_{i+1,n_{i+1}} - I$. If $b \in I$ we have $\sigma' = \sigma$. Otherwise **VF** implies $V \subseteq \mathsf{stubb}(s_i) = \mathsf{warm}(s_i) \cup \mathsf{frozen}(s_i)$, implying $a_{i+1,1}\cdots a_{i+1,j-1} \in I^*$, because by the choice of

j, these actions are not in $\mathsf{warm}(s_i)$, and we already saw that they are not in $\mathsf{frozen}(s_i)$ either. In this case, $b \in V$ and $b\sigma' = \sigma$. Everything that was promised in the claim has been justified above. □

Lemma 3.5. Assume **F1**, **F2**, **D1F**, and **D2F**. Assume that $s \in S_\mathsf{r}$, $s =\varepsilon\Rightarrow z$, and z refuses K. Either s r-refuses K, or there are s', z', and a prefix ρ of K such that $s =\rho\Rightarrow_\mathsf{r} s' =\varepsilon\Rightarrow z'$, $z =\rho\Rightarrow z'$, and the path that yields $s' =\varepsilon\Rightarrow z'$ is shorter than the path that yields $s =\varepsilon\Rightarrow z$.

Proof:
Let $s_0 = s$, $z_0 = z$, and $s_0 -a_1 \cdots a_n\rightarrow z_0$ be the path that yields $s =\varepsilon\Rightarrow z$. If s does not r-refuse K, then there is a $\kappa \in K$ such that $s =\kappa\Rightarrow_\mathsf{r}$. Let $s = s_0 -b_1\rightarrow_\mathsf{r} \ldots -b_m\rightarrow_\mathsf{r} s_m$ be the corresponding r-path. We apply **D2F** along this path for $i = 0$, $i = 1$, ... as long as $i < m$ and $\{a_1, \ldots, a_n\} \cap \mathsf{stubb}(s_i) = \emptyset$. **D2F** is applied to $s_i -a_1 \cdots a_n\rightarrow z_i$ and $s_i -b_{i+1}\rightarrow_\mathsf{r} s_{i+1}$, and it yields a z_{i+1} such that $s_{i+1} -a_1 \cdots a_n\rightarrow z_{i+1}$ and $z_i -b_{i+1}\rightarrow z_{i+1}$. Because z_0 refuses κ, we cannot have $z_0 -b_1 \cdots b_m\rightarrow z_m$. Therefore, for some $0 \le i < m$, $\{a_1, \ldots, a_n\} \cap \mathsf{stubb}(s_i) \neq \emptyset$.

If $\{a_1, \ldots, a_n\} \cap \mathsf{frozen}(s_i) = \emptyset$, then let $z' = z_i$. Because some a_j is in $\mathsf{stubb}(s_i)$ and no a_j is in $\mathsf{frozen}(s_i)$, some a_j is in $\mathsf{warm}(s_i)$. An application of **D1F** using the smallest such j yields an s' such that $s_i -a_j\rightarrow_\mathsf{r} s' =\varepsilon\Rightarrow z'$. This reasoning step consumes a_j.

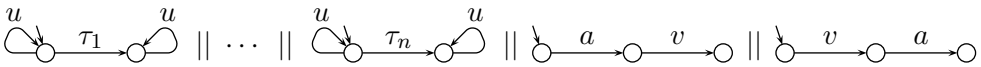
If $\{a_1, \ldots, a_n\} \cap \mathsf{frozen}(s_i) \neq \emptyset$, then let $s' = s_i$. Lemma 3.1 yields a z' such that $s_i =\varepsilon\Rightarrow z'$ and $z_i =\varepsilon\Rightarrow z'$. This reasoning step consumes at least one frozen action.

In both cases, $s -b_1 \cdots b_i\rightarrow_\mathsf{r} s_i =\varepsilon\Rightarrow_\mathsf{r} s'$, $z -b_1 \cdots b_i\rightarrow z_i =\varepsilon\Rightarrow z'$, the path that yields $s' =\varepsilon\Rightarrow z'$ is of length less than n, and the choice $\rho = b_1 \cdots b_i - I$ gives the claim. □

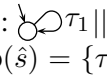
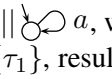
4. Driving Force

We pointed out in the previous section that a condition is needed that forces the stubborn set method to fire actions whenever it is necessary for verification. Such a condition may be nick-named "driving force". In this section we discuss the properties of various such conditions, including **D0VF**.

Early stubborn set methods contained the condition **D0** saying that if $s \in S_\mathsf{r}$ and $\mathsf{en}(s) \neq \emptyset$, then $\mathsf{stubb}(s) \cap \mathsf{en}(s) \neq \emptyset$. In the presence of frozen actions, the corresponding condition would say that if $s \in S_\mathsf{r}$ and $\mathsf{en}(s) \not\subseteq \mathsf{frozen}(s)$, then $\mathsf{warm}(s) \cap \mathsf{en}(s) \neq \emptyset$. This condition does, however, force the firing of actions also when it is easy to see that it is unnecessary, making the reduced state space grow. For instance, consider the following system with $V = \{a\}$. The last component blocks a and the second last blocks v, so a (and v) cannot ever occur. The algorithms presented in this study detect this and fire nothing in the initial state, resulting in a reduced state space that consists of one state and zero transitions. (In this case, it is a correct reduced state space.) **D0**, on the other hand, forces something to be fired in the initial state. Because no τ_i is concurrent with u, a typical stubborn set implementation that obeys **D0** would fire all enabled actions in each r-state, resulting in 2^n states and $(n + 2)2^{n-1}$ transitions.



What is worse, although **D0** forces firing some action, it does not necessarily force ever firing any action that makes progress towards $s =\sigma\Rightarrow_\mathsf{r}$ when $s \in S_\mathsf{r}$ and $s =\sigma\Rightarrow$. We have already seen an

example of this:  $||$  , with $V = \{a\}$. The conditions that have been presented this far allow choosing $\mathsf{stubb}(\hat{s}) = \{\tau_1\}$, resulting in a reduced state space that does not preserve the trace a. This is an instance of the ignoring problem.

These problems could be easily solved by, instead of **D0**, requiring $V \subseteq \mathsf{stubb}(s)$ for every $s \in S_\mathsf{r}$. In the example above with $\tau_1, \ldots, \tau_n$, we could choose $\mathsf{stubb}(\hat{s}) = \{a, v\}$, and thus fire nothing in the initial state. In general, assume that $s =a\Rightarrow$. We have $s -a_1 \cdots a_n\rightarrow$ where $a_n = a \in V$ and no other a_i is in V. By Lemma 3.1 we can assume that none of the a_i is in $\mathsf{frozen}(s)$. If $V \subseteq \mathsf{stubb}(s)$, then $a_n \in \mathsf{warm}(s)$. So there is a smallest $1 \leq i \leq n$ such that $a_i \in \mathsf{warm}(s)$. By **D1F**, there is s' such that $s -a_i\rightarrow_\mathsf{r} s' -a_1 \cdots a_{i-1}a_{i+1} \cdots a_n\rightarrow$. So, in each r-state, progress towards $s =a\Rightarrow_\mathsf{r}$ is certainly made.

Unfortunately, this very property makes the reduction results bad. Assume that **D1F** always applies, that is, assume that for each r-state s and each trace σ of s, there is an r-transition that makes progress towards $s =\sigma\Rightarrow_\mathsf{r}$. Consider

where $V = \{a, b\}$. In each r-state where τ_1 or a is enabled, the method has to fire τ_1 or a to make progress towards $s =a\Rightarrow_\mathsf{r}$, and in each r-state where τ_2 or b is enabled, the method has to fire τ_2 or b to make progress towards $s =b\Rightarrow_\mathsf{r}$. This results in constructing the full state space. Therefore, we do not ensure that $V \subseteq \mathsf{stubb}(s)$ for every $s \in S_\mathsf{r}$.

Earlier terminal strong component conditions. The first better solution to the ignoring problem was a method that guarantees that for every $s \in S_\mathsf{r}$ and $a \in \mathsf{en}(s)$, there are s_a and an r-path from s to it such that $a \in \mathsf{stubb}(s_a)$ [13]. A result resembling Lemma 3.4 or 3.5 was proven that said that if $s -\rho a\rightarrow$ and we start following the r-path from s to s_a, then either **D1** applies at some point along the path, consuming an action from ρa; or s_a is reached. In the latter case, **D1** applies at s_a, consuming an action from ρa.

To implement the condition, it suffices to focus on tsr-components, because from every r-state, a tsr-component is reachable. Using Tarjan's algorithm [24] similarly to Section 6, the method guarantees that for every tsr-component C and for every a that is enabled in some state of C, C contains a state s_a with $a \in \mathsf{stubb}(s_a)$. When Tarjan's algorithm has completed a tsr-component C, the method checks whether any a is ignored in C. If a is ignored, then by **D2** and **D1** it is enabled in every state of C but r-occurs in none of them. To check whether a r-occurs in none of them, it suffices to know C and Δ_r. If a is ignored, the stubborn set of the current state is extended to also contain some ignored action.

The drawback of this solution is that similarly to **D0**, it may fire obviously unnecessary actions.

An alternative is to guarantee that for every $s \in S_\mathsf{r}$ and $a \in V$, there are s_a and an r-path from s to it such that $a \in \mathsf{stubb}(s_a)$ [14]. This method does not fire obviously unnecessary actions in the same sense as the method in [13]. If the tsr-component happens to contain an r-occurrence $s -a\rightarrow_\mathsf{r}$ of a visible action a, then by **V**, $V \subseteq \mathsf{stubb}(s)$. So the remaining problem is, what to do with tsr-components that do not contain r-occurrences of visible actions. A technically simple possibility is to choose one state s in it and ensure $V \subseteq \mathsf{stubb}(s)$. Requiring $V \subseteq \mathsf{stubb}(s)$ for one r-state in each tsr-component is much better than requiring it for every $s \in S_\mathsf{r}$, but, even if $|V| = 1$, it still

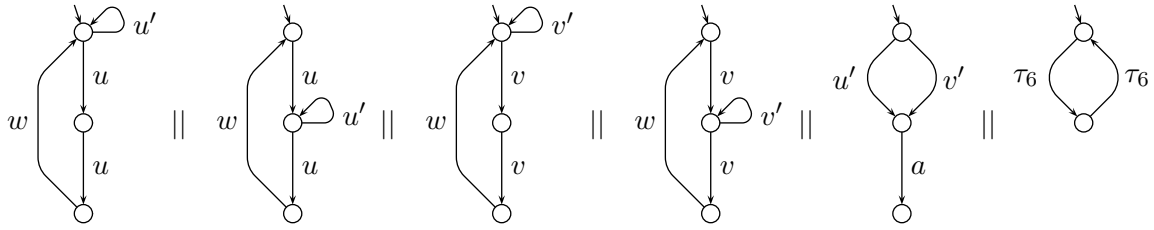


Figure 2. An example where the methods in both [13] and [14] yield unnecessarily bad reduction; $V = \{a\}$

sometimes fires concurrent actions in the same state, while the method in [13] fires them in different states, yielding better reduction.

Figure 2 shows an example. In it, to enable a, it is necessary and sufficient to fire either u' or v'. Synchronization has been designed so that u' and v' cannot ever occur. We assume below that the algorithm for constructing stubborn sets is good but not miraculous; for instance, the one in Section 8 would do. The methods in both [13] and [14] would first fire, for instance, the cycle $\hat{s} \ -uuvvw\rightarrow_{\mathsf{r}} \hat{s}$. Then the method in [13] would detect that $\mathsf{en}(\hat{s}) = \{u, v, \tau_6\}$ and both u and v occur in the cycle, but τ_6 does not. So it would fire τ_6 in $\hat{s}$. In the resulting state it would either fire τ_6 again, leading back to $\hat{s}$; or fire u or v, constructing even more r-states. In the former case, it would terminate, because now the reduced state space consists of precisely one tsr-component; both u, v, and τ_6 r-occur in it; and no other actions are enabled in $\hat{s}$.

A good implementation of the method in [14] would never fire τ_6. On the other hand, because no visible action occurs in the cycle $\hat{s} \ -uuvvw\rightarrow_{\mathsf{r}} \hat{s}$, it would try to ensure that $a \in \mathsf{stubb}(\hat{s})$. Most stubborn set algorithms would not realize that u' and v' are permanently disabled. Therefore, they would put also u' and v' into $\mathsf{stubb}(\hat{s})$. Then they would put u into $\mathsf{stubb}(\hat{s})$, because u' is disabled by the second component, and this fact will remain valid until the second component executes u. They would put also v into $\mathsf{stubb}(\hat{s})$, because v' is disabled by the fourth component, and this fact will remain valid until the fourth component executes v. So they would fire both u and v in $\hat{s}$.

Two decades later, the good reduction properties of the above solutions were combined by introducing a complicated method that in one state of the tsr-component chooses a set of enabled actions such that each of them should be fired in some state of the tsr-component, but the states need not be the same [11, 12]. In the case of Fig. 2, this method would choose $\{u, v\}$ in $\hat{s}$, construct (for instance) $\hat{s} \ -uuvvw\rightarrow_{\mathsf{r}} \hat{s}$, detect that both u and v occur in it, and terminate.

All visible or at least one enabled. Recently it turned out that a simple condition solves the ignoring problem in many, although not all, cases [10, 16]. This condition is what **D0VF** reduces to in the absence of frozen actions. The next theorem characterizes a subset of systems such that **D0VF** suffices to guarantee that all traces are preserved. The theorem can replace Lemma 5.1 in the next section, yielding a proof that also fair testing equivalence is preserved.

Theorem 4.1. Assume that **D0VF**, **D1F**, **D2F**, **VF**, **F1**, and **F2** are obeyed. At the end of the construction of the reduced state space, if s is an r-state, z is stable, and $s =\sigma\Rightarrow z$, then there is an r-path

$s =\sigma\Rightarrow_{\mathsf{r}} s'$ that is at most as long as the path that yields $s =\sigma\Rightarrow z$. Furthermore, s' is r-stable and $s' =\varepsilon\Rightarrow z$.

Proof:
Consider all paths of the form $s -b_1 \cdots b_m\rightarrow_{\mathsf{r}} s' -a_1 \cdots a_n\rightarrow z$, where the trace of $b_1 \cdots b_m a_1 \cdots a_n$ is σ and $m + n$ is at most the length of the path that yields $s =\sigma\Rightarrow z$. At least one such path exists, because we may choose $m = 0$. We prove that any such path with a minimal n has the promised properties.

To prove that none of the a_i is in $\mathsf{frozen}(s')$, we apply Lemma 3.1. It provides a z' such that $s' =\rho\Rightarrow z'$ and $z =\varepsilon\Rightarrow z'$, where ρ is the trace of $a_1 \cdots a_n$. Because z is stable we have $z' = z$. If any of the a_i is in $\mathsf{frozen}(s')$, then the path $s' =\rho\Rightarrow z'$ is of smaller length than n, contradicting the choice of n.

If any a_i is in $\mathsf{warm}(s')$, then **D1F** is applicable to the first such a_i, yielding an s'' such that $s' -a_i\rightarrow_{\mathsf{r}} s'' -a_1 \cdots a_{i-1}a_{i+1} \cdots a_n\rightarrow z$. The trace is not changed because of the following. If $a_i \in I$, this is obvious. If $a_i \in V$, then $V \subseteq \mathsf{stubb}(s')$ by **VF**; so $a_1, \ldots, a_{i-1}$ are not in V since they are not in $\mathsf{stubb}(s')$ by the choice of i. Also this contradicts the choice of n.

So none of the a_i is in $\mathsf{stubb}(s')$. If s' is not r-stable, then there is an $a \in I$ such that $s' -a\rightarrow_{\mathsf{r}}$. When that transition was constructed, **D2F** yielded $z -a\rightarrow_{\mathsf{r}}$, contradicting the stability of z. So s' is r-stable.

Therefore, if $s' -a\rightarrow_{\mathsf{r}}$, then $a \in V$. By **VF** $V \subseteq \mathsf{stubb}(s')$ held when that transition was constructed, after which $\mathsf{stubb}(s')$ has not shrunk. If s' has no outgoing r-transitions, then $V \subseteq \mathsf{stubb}(s')$ by **D0VF**.

Because $V \subseteq \mathsf{stubb}(s')$ and none of the a_i is in it, the a_i are invisible. As a consequence, the trace of $a_1 \cdots a_n$ is ε and the trace of $b_1 \cdots b_m$ is σ. □

This means that every trace leading to a stable state is preserved, even if the simple condition **D0VF** is used instead of the complicated tsr-component conditions discussed above. That is, if the system has the property that from every reachable state, a stable state is reachable, then the complicated conditions are not needed. Furthermore, thanks to the following theorem, the user need not know in advance that the system has this property.

Theorem 4.2. Assume that **D0VF**, **D1F**, **D2F**, **VF**, **F1**, and **F2** are obeyed. At the end of the construction of the reduced state space, if s is an r-state and has the trace σ in the full but not in the reduced state space, then some prefix of σ that s has in the reduced state space leads to an r-state s' such that all r-states that are r-reachable from s' are r-unstable and have no visible outgoing r-transitions.

Proof:
There are $s' \in S_{\mathsf{r}}$ and some prefix ρa of σ such that $s =\rho\Rightarrow_{\mathsf{r}} s' =a\Rightarrow$ but not $s' =a\Rightarrow_{\mathsf{r}}$. We choose s' such that the path $s' -a_1 \cdots a_n\rightarrow$ that yields $s' =a\Rightarrow$ is as short as possible. So $a_n = a$. None of the a_j is in $\mathsf{frozen}(s')$, because otherwise Lemma 3.1 would provide a shorter path. None of the a_j is in $\mathsf{warm}(s')$, because otherwise, using the smallest possible value of j, **D1F** would provide either $s' -a_n\rightarrow_{\mathsf{r}}$, yielding $s' =a\Rightarrow_{\mathsf{r}}$; or $s' -a_j\rightarrow_{\mathsf{r}} z =a\Rightarrow$, where $z =a\Rightarrow$ arises from a shorter path than $s' =a\Rightarrow$. So no a_j is in $\mathsf{stubb}(s')$.

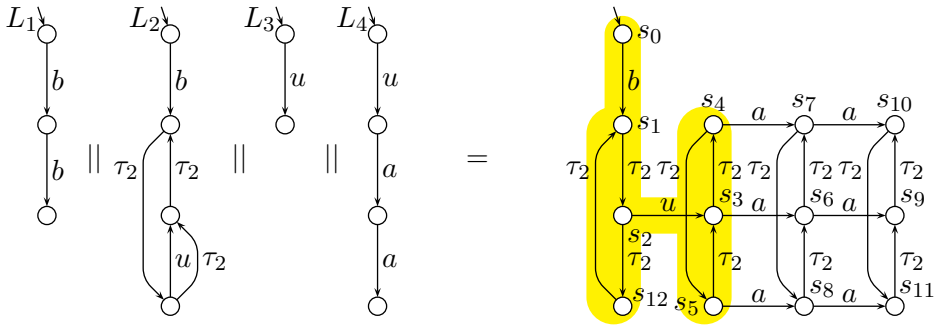


Figure 3. A system with $V = \{a, b\}$ where after s_3, every state has an invisible transition [10]

Because $a_n = a \in V$, we have $V \not\subseteq \mathsf{stubb}(s')$. From it **D0VF** implies that $\mathsf{warm}(s') \cap \mathsf{en}(s') \neq \emptyset$ and **VF** implies that $\mathsf{warm}(s') \cap \mathsf{en}(s') \cap V = \emptyset$. So s' has outgoing r-transitions, but none of them is labelled with a visible action. Let $s' \ -b\rightarrow_{\mathsf{r}} \ s''$ be any of them. An application of **D2F** to it yields $s'' \ -a_1 \cdots a_n\rightarrow$, that is, an as short as possible path that yields $s'' \ =a\Rightarrow$. Clearly $s \ =\rho\Rightarrow_{\mathsf{r}} \ s''$. Therefore, the reasoning applies also to s'' and, by induction, to every r-state that is r-reachable from s'. □

As a consequence, if the reduced state space obeys **D0VF**, **D1F**, **D2F**, **VF**, **F1**, and **F2**, then either the trace and fair testing equivalences are preserved, or the reduced state space exhibits pathological behaviour: it contains a state from which neither a stable state nor any visible activity is reachable. This pathological property can be detected from the reduced state space with well-known linear-time algorithms, by performing a graph search using the edges in reverse and using the deadlocks and tail states of visible transitions as the starting points.

That the reduced state space has this property does not prove that traces are lost. However, by Theorem 4.1, it does prove that the original system cannot reach a stable state from the states in question. If this is considered as sufficient reason for declaring the original system incorrect, then the freezing technique presented in this study is not needed. Instead, the easily implementable conditions **D0V**, **D1**, **D2**, and **V** suffice. (The proofs of Theorems 4.1 and 4.2 do not use the property that **D1F** and **D2F** with empty frozen sets add to **D1** and **D2**, that is, $s_i \ -a\rightarrow \ s'_i$ holds also when $1 \leq i < n$.)

A nasty example. We may, however, want to preserve the traces even when the system can no longer reach a stable state. Then, assuming that actions are never frozen, we run the risk of the problem that is illustrated with Fig. 3. The actions u and τ_2 are invisible. Originally only b is enabled, then only τ_2, and then only u and τ_2. After firing $s_2 \ -\tau_2\rightarrow_{\mathsf{r}} \ s_{12} \ -\tau_2\rightarrow_{\mathsf{r}} \ s_1$, the algorithm backtracks to s_2. After firing $s_2 \ -u\rightarrow_{\mathsf{r}} \ s_3$, the τ_2-cycle of L_2 and the aa-sequence of L_4 are concurrent, and b and u are permanently disabled.

Assume that in s_3, the construction of stubborn sets is started with a. Because a is enabled and visible, **VF** forces to also take b into $\mathsf{warm}(s_3)$, if a is taken. However, b is disabled because of L_2. If the stubborn set construction algorithm is good enough, it detects that b is permanently disabled, chooses $\mathsf{stubb}(s_3) = \{a, b\}$, and only fires a in s_3. However, detecting that an action is permanently

disabled is **PSPACE**-hard in general. So it is not realistic to assume that a stubborn set construction algorithm can always detect that an action is permanently disabled. For the sake of an example, we assume that the algorithm in Section 8 is used without any advanced features. It fails to detect that b is permanently disabled.

Because b is disabled by L_2 and only by it, the algorithm in Section 8 focuses on what L_2 can do next, that is, τ_2. Because τ_2 is enabled, invisible, not synchronized to by any other component, and its start state has no other outgoing transitions, $\{\tau_2\}$ qualifies as $\mathsf{warm}(s_3)$. Also $\{a, b, \tau_2\}$ qualifies as $\mathsf{warm}(s_3)$, but $\{\tau_2\}$ has fewer enabled actions, so the algorithm chooses $\mathsf{warm}(s_3) = \{\tau_2\}$. That is, the method constructs the r-transition $s_3 \,{-}\tau_2{\to_\mathsf{r}}\, s_4$. In the resulting state s_4 the situation is similar to s_3, so the r-transition $s_4 \,{-}\tau_2{\to_\mathsf{r}}\, s_5$ is constructed.

The situation is only slightly more complicated in s_5. Because u is an alternative for τ_2, the algorithm takes also u into $\mathsf{stubb}(s_5)$. The algorithm detects that u is disabled by L_3 which is in a deadlock, so it does not continue analysis further from u. Thus the method constructs (only) the r-transition $s_5 \,{-}\tau_2{\to_\mathsf{r}}\, s_3$. In conclusion, the cycle $s_3 \,{-}\tau_2{\to_\mathsf{r}}\, s_4 \,{-}\tau_2{\to_\mathsf{r}}\, s_5 \,{-}\tau_2{\to_\mathsf{r}}\, s_3$ is constructed.

At this point, simple methods backtrack from s_5, s_4, and s_3, losing the traces ba and baa. In agreement with Theorem 4.1, every state that is reachable from s_3 (that is, $s_3, \ldots, s_{11}$) is unstable. In harmony with Theorem 4.2, every r-state that is r-reachable from s_3 (that is, s_3, s_4, and s_5) is unstable and no occurrences of visible actions are r-reachable from s_3.

Instead of giving up in s_3, earlier methods based on tsr-components fire a in it, constructing $s_3 \,{-}a{\to_\mathsf{r}}\, s_6$. The same behaviour repeats in s_6 and finally in s_9 (except that a is found disabled in s_9). The method constructed all states of the full state space, although the $\tau_2\tau_2\tau_2$-cycle and aa-sequence are concurrent.

In conclusion, recovery based on tsr-components is actually seldom needed, but when it is needed, then the method is likely to fire the same unnecessary actions again and again (in this case τ_2), making the reduced state space grow. The idea of frozen actions was introduced to fix this problem [17]. That publication made the idea work for deterministic actions. Furthermore, because of lack of space, the proofs were sketchy. In the present study we make the idea work also with nondeterministic actions, and present detailed proofs.

Tsr-condition with frozen actions. The following condition is used in the present study. Unlike the very similar condition mentioned above, it does not suffer from the problem of firing in the same state actions that could otherwise be fired in different states. The condition puts all of them in $\mathsf{stubb}(s')$, but thanks to freezing, this is not the same thing as firing them all in s'. The implementation of the condition will be discussed in Section 6, and the condition will be used in Section 5 to prove that trace and fair testing equivalences are preserved.

SF At the end of the construction of the reduced state space, for every $s \in S_\mathsf{r}$ there is $s' \in S_\mathsf{r}$ such that $s =\varepsilon\Rightarrow_\mathsf{r} s'$ and $V \subseteq \mathsf{stubb}(s')$ holds with the final value of $\mathsf{stubb}(s')$.

To see that **SF** implies **D0VF**, assume that s obeys **SF** but has no outgoing r-transitions. Then the s' can only be s itself. So $V \subseteq \mathsf{stubb}(s)$.

Similarly to **D0VF**, **SF** does not necessarily force firing any enabled unfrozen actions, and this can only happen if firing any would be futile. This is because **SF** allows $\mathsf{en}_\mathsf{r}(s') = \emptyset$ even if $\mathsf{en}(s') \not\subseteq$

$\mathsf{frozen}(s')$, provided that $V \subseteq \mathsf{stubb}(s')$. In that case, no occurrences of visible actions are reachable from s'. To see this, let $s' =a\Rightarrow$ where $a \in V$. The elements of $\mathsf{frozen}(s')$ can be consumed with Lemma 3.1, resulting in $s' -a_1 \cdots a_n a\rightarrow$ with $a \in \mathsf{warm}(s')$. Then **D1F** can be applied either to a or to some a_i, implying $\mathsf{en}_\mathsf{r}(s') \neq \emptyset$.

5. Preservation of Trace and Fair Testing Equivalences

Throughout this section, we assume that **F1**, **F2**, **D1F**, **D2F**, **VF**, and **SF** are obeyed, and the situation is at the end of the construction of the reduced state space. (**D3F** is not needed. **D0VF** is not mentioned, because it follows from **SF**.) Based on these assumptions, we prove that trace equivalence and fair testing equivalence are preserved.

Lemma 5.1. If $s \in S_\mathsf{r}$ and $s =\sigma\Rightarrow z$, then there are s' and z' such that $s =\sigma\Rightarrow_\mathsf{r} s' =\varepsilon\Rightarrow z'$ and $z =\varepsilon\Rightarrow z'$.

Proof:
We prove the claim by induction on the length of the path that yields $s =\sigma\Rightarrow z$.

In the base case the length is 0, so $\sigma = \varepsilon$ and $s = z$. The claim is obtained by choosing $s' = s = z'$.

Assume now that the path is of positive length. If $\sigma = \varepsilon$, the claim can be obtained by choosing $s' = s$ and $z' = z$. Otherwise $\sigma \neq \varepsilon$. By **SF** there is s_V such that $s =\varepsilon\Rightarrow_\mathsf{r} s_V$ and $V \subseteq \mathsf{stubb}(s_V)$. We apply Lemma 3.4 to this r-path with $s_0 = s =\sigma\Rightarrow z = z_0$. The lemma yields i, s'_i, z'_i, b, and σ'. Let $\beta = b$ if $b \in V$ and $\beta = \varepsilon$ otherwise, so that $\sigma = \beta\sigma'$. Let $s' = s'_i$ and $z' = z'_i$. By the lemma, there is an s_i such that $s =\varepsilon\Rightarrow_\mathsf{r} s_i =\beta\Rightarrow_\mathsf{r} s' =\sigma'\Rightarrow z'$ and $z =\varepsilon\Rightarrow z'$. Furthermore, the path that yields $s' =\sigma'\Rightarrow z'$ is shorter than the path that yields $s =\sigma\Rightarrow z$. By induction, there are s'' and z'' such that $s' =\sigma'\Rightarrow_\mathsf{r} s'' =\varepsilon\Rightarrow z''$ and $z' =\varepsilon\Rightarrow z''$. We have $s =\beta\sigma'\Rightarrow_\mathsf{r} s''$ and $z =\varepsilon\Rightarrow z''$, giving the claim. □

Theorem 5.2. For every $s \in S_\mathsf{r}$ and $\sigma \in V^*$, $s =\sigma\Rightarrow$ if and only if $s =\sigma\Rightarrow_\mathsf{r}$.

Proof:
One direction follows from Lemma 5.1, and the other direction from the fact that every transition of the reduced state space is also a transition of the full state space. □

Lemma 5.3. If $s \in S_\mathsf{r}$, $s =\varepsilon\Rightarrow z$, and z refuses K, then s r-refuses K or there are s' and π such that π is a prefix of K, $s =\pi\Rightarrow_\mathsf{r} s'$, and s' r-refuses $\pi^{-1}K$.

Proof:
We prove the claim by induction on the length of the path that yields $s =\varepsilon\Rightarrow z$.

In the base case the length is 0, so $s = z$. If $s =\kappa\Rightarrow_\mathsf{r}$ for some $\kappa \in K$, then $z = s =\kappa\Rightarrow$, contradicting z refuses K. Therefore, s r-refuses K.

Assume now that the path is of positive length. If s does not r-refuse K, then by Lemma 3.5 there are s', z', and a prefix π of K such that $s =\pi\Rightarrow_\mathsf{r} s' =\varepsilon\Rightarrow z'$, $z =\pi\Rightarrow z'$, and the path that yields $s' =\varepsilon\Rightarrow z'$ is shorter than the path that yields $s =\varepsilon\Rightarrow z$. Because z refuses K, z' refuses $\pi^{-1}K$.

By induction, either s' r-refuses $\pi^{-1}K$; or there are s'' and μ such that $s' =\mu\Rightarrow_{\mathsf{r}} s''$, μ is a prefix of $\pi^{-1}K$, and s'' r-refuses $\mu^{-1}(\pi^{-1}K) = (\pi\mu)^{-1}K$. So we have the claim with s'' and $\pi\mu$. □

Theorem 5.4. The reduced state space is fair testing equivalent to the full state space.

Proof:
Assume that (σ, K) is an r-tree failure. There is s such that $\hat{s} =\sigma\Rightarrow_{\mathsf{r}} s$ and s r-refuses K. Clearly $\hat{s} =\sigma\Rightarrow s$. By Theorem 5.2, s refuses K. So (σ, K) is a tree failure.

Assume that (σ, K) is a tree failure. There is z such that $\hat{s} =\sigma\Rightarrow z$ and z refuses K. By Lemma 5.1 there are s' and z' such that $\hat{s} =\sigma\Rightarrow_{\mathsf{r}} s' =\varepsilon\Rightarrow z'$ and $z =\varepsilon\Rightarrow z'$. Also z' refuses K. By Lemma 5.3 either s' r-refuses K, implying that (σ, K) is an r-tree failure; or there are s'' and π such that π is a prefix of K, $s' =\pi\Rightarrow_{\mathsf{r}} s''$ and s'' r-refuses $\pi^{-1}K$. So $(\sigma\pi, \pi^{-1}K)$ is an r-tree failure. □

6. Implementation of SF and Freezing

In this section we present and prove correct an algorithm that guarantees **SF**, assuming that all stubborn sets obey **D0VF** and **VF**. The algorithm also contains the freezing of actions. We will prove in Section 7 that **F2** and a strengthened version of **F1** are invariant properties of the algorithm. (There we will also assume **D1F**, **D2F**, and **D3F**.) In principle, we should first prove this fact and only after that prove **SF**. However, both proofs require that the algorithm has been described, and not much is needed on top of the description to prove **SF**. Therefore, we present the material in "wrong" order. This is correct, because we do not appeal to **F1** and **F2** in this section.

The algorithm is based on the following observation.

Lemma 6.1. At the end of the construction of the reduced state space, **SF** holds if and only if every tsr-component contains a state s such that $V \subseteq \mathsf{stubb}(s)$.

Proof:
Assume that every tsr-component C contains a state s_C such that $V \subseteq \mathsf{stubb}(s_C)$. Let $s \in S_{\mathsf{r}}$. Some tsr-component C is r-reachable from s and some state s_C in it has $V \subseteq \mathsf{stubb}(s_C)$. If the r-path from s to s_C contains only invisible actions, then s_C plays the role of s' of **SF** for s. Otherwise the path contains an r-transition $s' -a\rightarrow_{\mathsf{r}} s''$ such that $a \in V$. When it was constructed, we had $a \in \mathsf{warm}(s') \cap \mathsf{en}(s')$. At that time, **VF** implied $V \subseteq \mathsf{stubb}(s')$. Because $\mathsf{stubb}(s')$ does not shrink and V does not change at all, $V \subseteq \mathsf{stubb}(s')$ has held since then. So s' plays the role of s' of **SF** for s.

Assume now that some tsr-component C does not have such a state. Then its states violate **SF**. □

As a consequence, **SF** can be implemented by recognizing the tsr-components, and ensuring that each of them contains a state s such that $V \subseteq \mathsf{stubb}(s)$. Fig. 4 shows both how this can be done and how actions are frozen. The algorithm is based on the well-known recursive method for constructing $(S_{\mathsf{r}}, I, V, \Delta_{\mathsf{r}}, \hat{s})$ in depth-first order. It is initially called with DFS$(\hat{s}, \emptyset)$. To recognize tsr-components efficiently, Tarjan's algorithm [24] is applied on top of the depth-first search, although its details are not shown in the figure. (A good practical improvement to Tarjan's algorithm has been presented in [25]. It was re-invented and slightly modified in [26].) The *root* of a strong r-component is the

DFS(s, old_frozen)

```
1  S_r := S_r ∪ {s}
2  frozen(s) := old_frozen
3  done := false
4  while ¬ done do
5      warm(s) := compute_stubborn(s, frozen(s))
6      for a ∈ warm(s) ∩ en(s) do
7          for s' such that s −a→ s' do
8              Δ_r := Δ_r ∪ {(s, a, s')}
9              if s' ∉ S_r then DFS(s', frozen(s))
10     if   warm(s) ∩ en(s) = ∅
11         ∨  s is not the root of a terminal strong component of (S_r, I, V, Δ_r, ŝ)
12         ∨  ∃s' ∈ R_r(s) : ∃a ∈ V: s' −a→_r
13     then done := true
14     else frozen(s) := ⋃_{z∈R_r(s)} stubb(z)
```

Figure 4. Implementation of **SF** and freezing. In the figure, $\mathcal{R}_{\mathsf{r}}(s) = \{s' \mid s' \text{ is r-reachable from } s\}$

r-state in it that depth-first search found first and thus backtracks from last. The algorithm performs special activity towards ensuring **SF** only when the current state s is the root of a tsr-component.

In the algorithm, the parameter s is a pointer to or index number of a global state. The parameter old_frozen is a pointer to or index number of a set of actions. The algorithm does not change the values of any of its parameters. The variables S_{r} and Δ_{r} are global.

When an r-state s is constructed, it inherits the current frozen set of its parent state. That it obeys **F1** and **F2** in s will be proven in Section 7. The algorithm enters the **while**-loop. In each iteration, it computes a new warm set. By Lemma 3.3, the set of all actions, that is, $I \cup V$, satisfies the conditions listed towards the beginning of this section. However, $I \cup V$ is usually the worst possible from the point of view of reduction results. How a usually better set is found will be described in Section 8.

The algorithm executes all enabled actions in the warm set in all possible ways, storing the resulting r-transitions and entering those of the resulting r-states that have not yet been entered. After the warm set has been processed, several things are possible. The options have been designed so that when the algorithm backtracks from any tsr-component C, there is $s' \in C$ such that $V \subseteq \mathsf{stubb}(s')$.

If s is not the root of a tsr-component, then backtracking from it does not mean backtracking from a tsr-component. The algorithm treats s as ready and backtracks from it. Otherwise, if the tsr-component contains an r-occurrence $s' \ -a\rightarrow_{\mathsf{r}}$ of a visible action a, then, similarly to the proof of Lemma 6.1, $V \subseteq \mathsf{stubb}(s')$ by **VF**. Again, the algorithm backtracks from s. If $\mathsf{warm}(s)$ does not contain enabled actions, then $V \subseteq \mathsf{stubb}(s)$, because $\mathsf{warm}(s)$ obeys **D0VF** for s. Also this is sufficient justification for backtracking.

In the remaining case, on line 14 the algorithm freezes all actions in the stubborn sets of every state in the tsr-component, resulting in a new frozen set. That it obeys **F1** and **F2** will be proven in

Section 7. Then the algorithm starts a new iteration of the **while**-loop. This may result in firing new actions in s. The strong r-component that contains s may grow and may cease from being terminal, and s may cease from being its root. Fortunately, for the reason discussed soon, Tarjan's algorithm is not confused. Eventually the algorithm is back on line 10. It checks whether s is still the root of a tsr-component. If it is, the algorithm checks whether the tsr-component now contains an occurrence of a visible action, or whether the new warm set lacks enabled actions. If neither of these holds and s is still the root of a tsr-component, the algorithm again freezes actions and computes a warm set.

That is, the algorithm does not backtrack from s until either s is no longer the root of a tsr-component, or it is but the component contains an s' such that $V \subseteq \mathsf{stubb}(s')$. As a consequence, the algorithm does not backtrack from a tsr-component until the component contains an s' such that $V \subseteq \mathsf{stubb}(s')$. On the other hand, the algorithm does backtrack from each r-state and thus from each tsr-component for the following reason. Each $\mathsf{frozen}(s)$ is bigger than the previous $\mathsf{frozen}(s)$, because by line 14 it has the previous one as a subset, by line 10 it is not computed unless it contains a new action, and the only assignments to $\mathsf{frozen}(s)$ are those on lines 2 and 14. Therefore, the **while**-loop will terminate at the latest when $\mathsf{frozen}(s) = I \cup V$.

In conclusion, the algorithm terminates, and then every tsr-component contains a state s' such that $V \subseteq \mathsf{stubb}(s')$. So **SF** holds by Lemma 6.1. We have proven the following.

Theorem 6.2. **SF** holds at the end of the execution of the algorithm in Fig. 4.

The algorithm tries to compute new warm sets for s one at a time, instead of establishing $V \subseteq \mathsf{stubb}(s)$ early on. This is to benefit from the possibility that something else than $V \subseteq \mathsf{stubb}(s)$ stops $\mathsf{stubb}(s)$ from growing, so that fewer actions would be fired in s. We saw in Section 4 that unnecessary $V \subseteq \mathsf{stubb}(s)$ can be detrimental to reduction results.

Interaction with Tarjan's algorithm. Tarjan's algorithm recognizes strong components in depth-first order, with deepest first. It maintains a *component stack* that consists of the states that have been found but whose strong component has not yet been completed. A state is pushed to this stack when it is entered during depth-first search. Each state in the component stack has two numbers called *index* and *low-link*. The index does not change. The index of any state other than $\hat{s}$ is bigger than the index of its parent. When s has not yet been backtracked from, its low-link is the smallest index of any state in the component stack to which there is a path from s via the edges that Tarjan's algorithm has so far traversed. After investigating all outgoing edges of s, Tarjan's algorithm tests whether s is the root of a strong component by testing whether its low-link is not smaller than its index. If the answer is yes, Tarjan's algorithm marks all states of the component as fully processed and pops them from the component stack.

To facilitate testing whether or not the component is terminal, a bit may be backward-propagated during the traversal that tells whether a state in another strong component is known to be reachable from the current state. In the case of the root (but not necessarily all other states of the same component) this bit tells reliably whether the component is actually terminal.

To implement line 11 in Fig. 4, these tests are performed outside Tarjan's algorithm, before informing Tarjan's algorithm whether or not the current state has more outgoing edges. The data structures

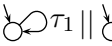
of Tarjan's algorithm are read, but this does not affect Tarjan's algorithm, because they are not modified. If more outgoing edges are added by lines 14 and 4 to 9, from the point of view of Tarjan's algorithm the situation is not different from what it would have been if the added edges had always been there. Therefore, Tarjan's algorithm operates on the final reduced state space, unaffected by the fact that some edges were inserted just before it would have become aware of their non-existence, and the insertion was based on the contents of its data structures.

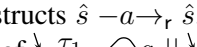
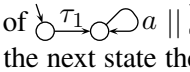
Examples. In all these examples, a and b are visible and u, v, and so on are not.

Consider again Fig. 3. We already saw that the algorithm constructs the cycle $s_3 \ -\tau_2\rightarrow_{\mathsf{r}} \ s_4 \ -\tau_2\rightarrow_{\mathsf{r}} \ s_5 \ -\tau_2\rightarrow_{\mathsf{r}} \ s_3$. When it has done so and is about to backtrack from s_3, it freezes u and τ_2, and computes a new $\mathsf{warm}(s_3)$. This computation returns $\{a, b\}$, because now a is the only unfrozen enabled action, and by **D0VF**, either such an action must be returned or the result must contain V as a subset. So the algorithm constructs $s_3 \ -a\rightarrow_{\mathsf{r}} \ s_6$. In s_6 it again only fires a, for the same reason. In the next state s_9 $compute\_stubborn$ again returns $\{a, b\}$ but the **while**-loop fires nothing, because now both a and b are disabled. **D0VF** holds because $V \subseteq \mathsf{warm}(s_9) \cup \mathsf{frozen}(s_9) = \{a, b\} \cup \{\tau_2, u\}$.

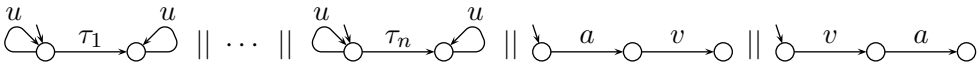
The example also illustrates that freezing an action prematurely may cause an erroneous result. If τ_2 is frozen already in the initial state, the trace ba is lost.

In general, what happens with an example may depend on the details of the implementation of $compute\_stubborn$. The implementation in Section 8 is good in the sense used in this subsection.

Consider  . A good implementation of $compute\_stubborn$ returns initially $\{a\}$, resulting in only constructing $\hat{s} \ -a\rightarrow_{\mathsf{r}} \ \hat{s}$. Then the algorithm terminates. A bad implementation may return initially $\{\tau_1\}$. If that happens, the algorithm in Fig. 4 constructs $\hat{s} \ -\tau_1\rightarrow_{\mathsf{r}} \ \hat{s}$, then freezes τ_1, and finally constructs $\hat{s} \ -a\rightarrow_{\mathsf{r}} \ \hat{s}$.

In the case of  and  , a good implementation fires first one and in the next state the other of τ_1 and τ_2, after which it fires a. With the following system, it constructs $\hat{s} \ -\sigma\rightarrow_{\mathsf{r}} \ s$ where σ is some permutation of $\tau_1\tau_1\tau_2\tau_2$, and then fires both a and b in s.

A good implementation fires nothing in the initial state of the following system, because $\{a, v\}$ satisfies $V \subseteq \{a, v\}$, **D1F**, **D2F**, **D3F**, and **VF**:



In the case of Fig. 2, a good implementation first constructs $\hat{s} \ -\sigma w\rightarrow_{\mathsf{r}} \ \hat{s}$ where σ is a permutation of $uuvv$. Then it freezes u, v, w, u', and v', tries the warm set $\{a\}$ in vain, and terminates. A bad implementation might first construct $\hat{s} \ -\tau_6\tau_6\rightarrow_{\mathsf{r}} \ \hat{s}$, then freeze τ_6, and then continue like the good implementation.

7. Validity of F1 and F2

In this section we assume that the stubborn sets obey **D1F**, **D2F**, and **D3F**. We prove that both the original $\mathsf{frozen}(s)$ and its later versions satisfy **F1** and **F2**. We first prove that the following strength-

ened variant of **F1** is an invariant property of the algorithm, that is, every operation that modifies any $\mathsf{frozen}(s)$ keeps it valid, assuming it was valid before the operation:

F1' If $s_0 \in S_\mathsf{r}$, $s_0 \ -a_1 \cdots a_n\rightarrow \ s'_0 \ -b_1 \cdots b_m\rightarrow \ s''_0$, and $c_1 \cdots c_k = b_1 \cdots b_m - \mathsf{frozen}(s_0)$, where none of $a_1, \ldots, a_n$ is in $\mathsf{frozen}(s_0)$, then there are z_0 and $\gamma \in \mathsf{frozen}(s_0)^*$ such that $s'_0 \ -c_1 \cdots c_k\rightarrow z_0$ and $s''_0 \ -\gamma\rightarrow z_0$.

We first deal with the initial frozen sets of newly created r-states, that is, those assigned on line 2. The initial call DFS$(\hat{s}, \emptyset)$ makes $\mathsf{frozen}(\hat{s}) = \emptyset$, so **F1'** clearly holds with $z_0 = s''_0$ and $\gamma = \varepsilon$. The initial frozen sets of the remaining r-states are dealt with by the following lemma.

Lemma 7.1. The call DFS$(s', \mathsf{frozen}(s))$ on line 9 does not invalidate **F1'**.

Proof:
Let $s_0 = s'$. The call makes $\mathsf{frozen}(s_0) = \mathsf{frozen}(s)$ and does not modify any other frozen set. The call was made after constructing $s \ -a\rightarrow_\mathsf{r} \ s'$. Because this transition was constructed, we had $a \notin \mathsf{frozen}(s)$. Because **F1'** held for $\mathsf{frozen}(s)$, we can apply it to $s \ -a\rightarrow \ s' \ -a_1 \cdots a_n\rightarrow \ s'_0 \ -b_1 \cdots b_m\rightarrow \ s''_0$. Doing so immediately gives z_0 and γ with the promised properties. □

The only remaining place where any frozen set changes is line 14. It is much more difficult to deal with.

Lemma 7.2. Line 14 does not invalidate **F1'**.

Proof:
Let the current state be denoted with s_0, and assume that the execution is currently immediately before the assignment on line 14. This fixes the meaning of all sets whose contents may change. Because the algorithm is on line 14, s_0 is the root of a tsr-component C. Let $\mathsf{frozen}'(s_0) = \bigcup_{s\in C} \mathsf{stubb}(s)$, that is, $\mathsf{frozen}'(s_0)$ denotes the value that the frozen set of s_0 will get due to line 14. We have $\mathsf{frozen}(s_0) \subseteq \mathsf{stubb}(s_0) \subseteq \mathsf{frozen}'(s_0)$. We may assume that for every $s \in S_\mathsf{r}$, **F1'** currently holds for $\mathsf{frozen}(s)$, and we must show that it also holds for $\mathsf{frozen}'(s_0)$. That is, we assume the if-part of **F1'** using $\mathsf{frozen}'(s_0)$, and prove its then-part. Let $\beta_0 = b_1 \cdots b_m$. If $\beta_0 - \mathsf{frozen}'(s_0) = \beta_0 - \mathsf{frozen}(s_0)$, then the claim follows immediately. So from now on we assume that $\beta_0 - \mathsf{frozen}'(s_0) \neq \beta_0 - \mathsf{frozen}(s_0)$.

Please see Fig. 5. Our next goal is to demonstrate the existence of an r-path $s_0 \ -u_1\rightarrow_\mathsf{r} \ s_1 \ -u_2\rightarrow_\mathsf{r} \ \ldots \ -u_\ell\rightarrow_\mathsf{r} \ s_\ell$, a path $s'_0 \ -u_1\rightarrow \ s'_1 \ -u_2\rightarrow \ \ldots \ -u_\ell\rightarrow \ s'_\ell$, states $s''_1, \ldots, s''_\ell$, and strings $\beta_1, \gamma_1, \ldots, \beta_\ell, \gamma_\ell$ such that $s_\ell = s_0$, $\beta_\ell = \beta_0 - \mathsf{frozen}'(s_0)$, and, for $1 \leq i \leq \ell$, $s_i \ -a_1 \cdots a_n\rightarrow \ s'_i \ -\beta_i\rightarrow \ s''_i$, $s''_{i-1} \ -\gamma_i\rightarrow \ s''_i$, and $\gamma_i \in \mathsf{frozen}'(s_0)^*$. In terms of Fig. 5, γ_i is either γ'_i (then $z_i = s''_i$) or $\gamma'_i u_i$.

Because $s_0 \in C$ and C is a tsr-component, each s_i belongs to C, implying $\mathsf{stubb}(s_i) \subseteq \mathsf{frozen}'(s_0)$. By the if-part of **F1'**, none of $a_1, \ldots, a_n$ is in $\mathsf{frozen}'(s_0)$.

The path $s_0 \ -a_1 \cdots a_n\rightarrow \ s'_0 \ -\beta_0\rightarrow \ s''_0$ was given in the if-part of **F1'**. Assume that the paths have been constructed up to $s_{i-1} \ -a_1 \cdots a_n\rightarrow \ s'_{i-1} \ -\beta_{i-1}\rightarrow \ s''_{i-1}$. Because all frozen sets satisfied **F1'** beforehand, there are z_i, $\beta'_i = \beta_{i-1} - \mathsf{frozen}(s_{i-1})$, and $\gamma'_i \in \mathsf{frozen}(s_{i-1})^* \subseteq \mathsf{stubb}(s_{i-1})^* \subseteq \mathsf{frozen}'(s_0)^*$ such that $s'_{i-1} \ -\beta'_i\rightarrow \ z_i$ and $s''_{i-1} \ -\gamma'_i\rightarrow \ z_i$. There are now four cases.

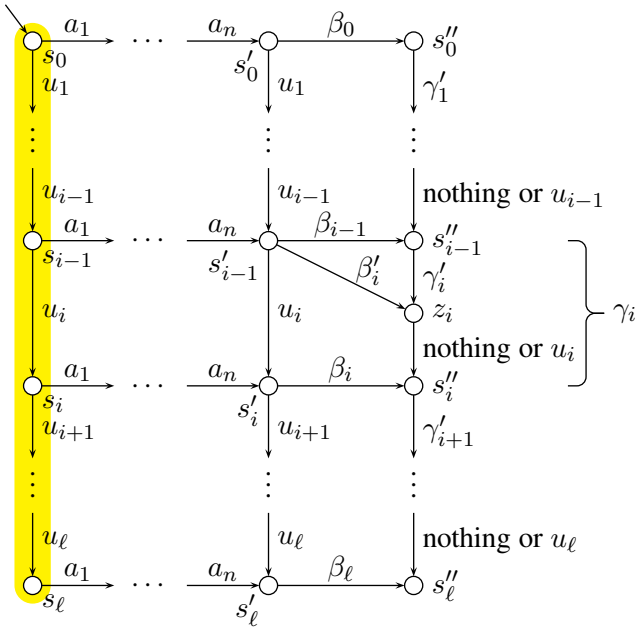


Figure 5. Illustrating the proof of Lemma 7.2

1. If β'_i has an element in common with $\mathsf{warm}(s_{i-1})$, then, by letting u_i be the first such element of β'_i, we get δ_i and ρ_i such that $\beta'_i = \delta_i u_i \rho_i$ and δ_i has no element in common with $\mathsf{warm}(s_{i-1})$. Because β'_i has no element in common with $\mathsf{frozen}(s_{i-1})$, this implies that δ_i has no element in common with $\mathsf{stubb}(s_{i-1})$. Because none of a_j is in $\mathsf{frozen}'(s_0)$ and $\mathsf{stubb}(s_{i-1}) \subseteq \mathsf{frozen}'(s_0)$, none of a_j is in $\mathsf{stubb}(s_{i-1})$. So **D1F** can be applied to $s_{i-1} -a_1 \cdots a_n\rightarrow s'_{i-1} -\delta_i u_i \rho_i\rightarrow z_i$, yielding s_i and s'_i such that $s_{i-1} -u_i\rightarrow_{\mathsf{r}} s_i -a_1 \cdots a_n\rightarrow s'_i -\delta_i \rho_i\rightarrow z_i$ and $s'_{i-1} -u_i\rightarrow s'_i$. We choose $\beta_i = \delta_i \rho_i$, $s''_i = z_i$, and $\gamma_i = \gamma'_i$. Because $|\beta_i| < |\beta'_i| \leq |\beta_{i-1}|$, this case can only arise a finite number of times.

2. If case 1 does not apply, β'_i has no element in common with $\mathsf{warm}(s_{i-1})$. Because β'_i has no element in common with $\mathsf{frozen}(s_{i-1})$, β'_i has no element in common with $\mathsf{stubb}(s_{i-1})$. Assume that β'_i has an element b in common with $\mathsf{frozen}'(s_0)$.

   There is $s_b \in C$ such that $b \in \mathsf{stubb}(s_b)$. Because C is a tsr-component, there is an r-path from s_{i-1} to s_b. We choose b and s_b so that the r-path is as short as possible. We choose u_i and s_i so that $s_{i-1} -u_i\rightarrow_{\mathsf{r}} s_i$ is the first step of this r-path. So $u_i \in \mathsf{stubb}(s_{i-1}) \subseteq \mathsf{frozen}'(s_0)$ and thus u_i does not occur in $a_1 \cdots a_n \beta'_i$. Although u_i may now be frozen, it was warm when $s_{i-1} -u_i\rightarrow_{\mathsf{r}} s_i$ was constructed. Therefore, **D2F** can be applied, yielding s'_i and s''_i such that $s_i -a_1 \cdots a_n\rightarrow s'_i -\beta'_i\rightarrow s''_i$, $s'_{i-1} -u_i\rightarrow s'_i$, and $z_i -u_i\rightarrow s''_i$. We choose $\beta_i = \beta'_i$ and $\gamma_i = \gamma'_i u_i$. So $s''_{i-1} -\gamma_i\rightarrow s''_i$.

   Clearly $|\beta_i| \leq |\beta_{i-1}|$. If $\beta'_i \neq \beta_{i-1}$, then $|\beta_i| < |\beta_{i-1}|$. Otherwise $\beta_i = \beta'_i = \beta_{i-1}$, and b occurs in all of them. The shortest r-path from s_i to s_b is shorter than the shortest r-path from s_{i-1} to

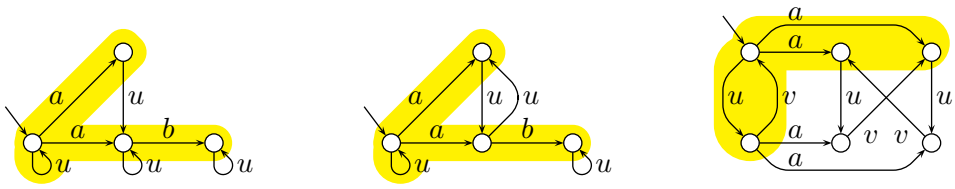


Figure 6. Illustrating the necessity of (left) **D3F** and (middle) $s_i \xrightarrow{-a\rightarrow} s'_i$ for $0 < i < n$ in **D1F** and **D2F**; and (right) that $s'_\ell = s'_0$ does not necessarily hold

s_b. Therefore, this subcase can repeat only a finite number of times in a row. Eventually case 1 or the first subcase of case 2 applies, shortening β_i.

3. If cases 1 and 2 do not apply, β'_i has no element in common with $\mathsf{frozen}'(s_0)$. Because all elements removed from β_{i-1} or β'_i during earlier steps belong to $\mathsf{frozen}(s_{i-1})$ or $\mathsf{warm}(s_{i-1})$ and thus to $\mathsf{frozen}'(s_0)$, we have $\beta'_i = \beta_0 - \mathsf{frozen}'(s_0) = c_1 \cdots c_k$. Because C is a tsr-component, there is an r-path from s_{i-1} to s_0. If $s_{i-1} \neq s_0$, then we choose as short such an r-path as possible, and choose u_i and s_i so that $s_{i-1} \xrightarrow{-u_i\rightarrow_{\mathsf{r}}} s_i$ is the first step of this r-path. Let $\beta_i = \beta'_i$ and $\gamma_i = \gamma'_i u_i$. Similarly to case 2, an application of **D2F** yields s'_i and s''_i such that $s_i \xrightarrow{-a_1 \cdots a_n\rightarrow} s'_i \xrightarrow{-\beta_i\rightarrow} s''_i$, $s'_{i-1} \xrightarrow{-u_i\rightarrow} s'_i$, and $s''_{i-1} \xrightarrow{-\gamma_i\rightarrow} s''_i$. If also $s_i \neq s_0$, we will have $\beta'_{i+1} = \beta_i = \beta'_i = \beta_0 - \mathsf{frozen}'(s_0)$ and thus get back to this case. Furthermore, we may choose $\gamma'_{i+1} = \varepsilon$ and $z_{i+1} = s''_i$.

4. Repetition of case 3 ends when $s_{i-1} = s_0$. Then the construction of the paths is complete and we let $\ell = i - 1$. So $s_\ell = s_0$. Because $\beta'_{\ell+1} = \beta_0 - \mathsf{frozen}'(s_0) \neq \beta_0 - \mathsf{frozen}(s_0) = \beta'_1$, we have $\ell > 0$. Therefore, $\mathsf{frozen}(s_\ell) = \mathsf{frozen}(s_0)$ was removed from β_0 when i was 1, implying $\beta_\ell = \beta'_{\ell+1} = \beta_0 - \mathsf{frozen}'(s_0) = c_1 \cdots c_k$.

If we had $s'_\ell = s'_0$, we would have the claim. Unfortunately, as we will demonstrate later with an example, it does not necessarily hold. So we need a more complicated argument.

We have now gone once around $s_0 \xrightarrow{-u_1 \cdots u_\ell\rightarrow_{\mathsf{r}}} s_0$. We have $u_i \in \mathsf{stubb}(s_{i-1}) \subseteq \mathsf{frozen}'(s_0)$, and $a_1 \cdots a_n c_1 \cdots c_k$ has no elements in common with $\mathsf{frozen}'(s_0)$. Therefore, we can go around $s_0 \xrightarrow{-u_1 \cdots u_\ell\rightarrow_{\mathsf{r}}} s_0$ a second time, then a third time and so on, repeatedly applying **D2F** to yield $s'_{\ell+1}$, $s''_{\ell+1}$, $s'_{\ell+2}$, $s''_{\ell+2}$, …, $s'_{2\ell}$, $s''_{2\ell}$, $s'_{2\ell+1}$, $s''_{2\ell+1}$, and so on. More formally, for every $1 \leq i \leq \ell$ and $j \geq 1$, **D2F** can be applied to $s_{i-1} \xrightarrow{-a_1 \cdots a_n\rightarrow} s'_{j\ell+i-1} \xrightarrow{-c_1 \cdots c_k\rightarrow} s''_{j\ell+i-1}$ and $s_{i-1} \xrightarrow{-u_i\rightarrow_{\mathsf{r}}} s_i$, to obtain $s'_{j\ell+i}$ and $s''_{j\ell+i}$ such that $s_i \xrightarrow{-a_1 \cdots a_n\rightarrow} s'_{j\ell+i} \xrightarrow{-c_1 \cdots c_k\rightarrow} s''_{j\ell+i}$, $s'_{j\ell+i-1} \xrightarrow{-u_i\rightarrow} s'_{j\ell+i}$, and $s''_{j\ell+i-1} \xrightarrow{-u_i\rightarrow} s''_{j\ell+i}$. We have shown $s'_{j\ell} \xrightarrow{-c_1 \cdots c_k\rightarrow} s''_{j\ell}$ for every $j > 0$, but not yet for $j = 0$.

For every $j \geq 0$ we have $s_0 = s_{j\ell} \xrightarrow{-a_1 \cdots a_n\rightarrow} s'_{j\ell}$. Because the reduced state space is finite, there are i and j such that $i < j$ and $s'_{i\ell} = s'_{j\ell}$. Let i be the smallest possible. If $i > 0$, then $s'_{(i-1)\ell} \neq s'_{(j-1)\ell}$ but $s'_{i\ell} = s'_{j\ell}$, contradicting **D3F**. Therefore, $i = 0$. Because $j > i$, we have $s'_{j\ell} \xrightarrow{-c_1 \cdots c_k\rightarrow} s''_{j\ell}$. By $i = 0$ and $s'_{i\ell} = s'_{j\ell}$ we conclude $s'_0 \xrightarrow{-c_1 \cdots c_k\rightarrow} s''_{j\ell}$. We choose $z_0 = s''_{j\ell}$ and $\gamma = \gamma_1 \cdots \gamma_\ell (u_1 \cdots u_\ell)^{j-1}$ and have the then-part of **F1'**. □

Figure 6 left shows an example where **D1F**, **D2F**, **VF**, and **SF** hold, but **D3F** does not and **F1** becomes invalid by the algorithm in Fig. 4. In it, $I = \{u\}$, $V = \{a, b\}$, and originally all frozen sets

are empty. The set $\{u\}$ satisfies **D1F**, **D2F**, and **VF** in the initial state. The algorithm may try it first, resulting in the freezing of u. The algorithm then constructs $\hat{s} \ -a\rightarrow_{\mathsf{r}} \ s$, where s is the topmost state in the figure. This makes $\mathsf{frozen}(s) = \{u\}$. As a consequence, although $s \ -ub\rightarrow$, $s \ =b\Rightarrow_{\mathsf{r}}$ does not hold, making $(a, \{b\})$ a fake tree failure. This illustrates the necessity of an extra condition, which in our case is **D3F**.

F1 becomes invalid also in Fig. 6 middle. The example violates the requirement in **D1F** and **D2F** that $s_i \ -a\rightarrow \ s'_i$ also when $0 < i < n$.

In Fig. 6 right, **D1F**, **D2F**, **D3F**, **VF**, and **SF** hold, and **F1'** is not invalidated. The example demonstrates that in the proof of Lemma 7.2, s'_ℓ may be different from s'_0. The algorithm first executes $\hat{s} \ -uv\rightarrow_{\mathsf{r}} \ \hat{s}$, then freezes u and v, and finally executes $\hat{s} \ -a\rightarrow_{\mathsf{r}}$ in both ways.

Let us now deal with **F2**.

Lemma 7.3. Neither the initial call DFS$(\hat{s}, \emptyset)$, the call on line 9, nor line 14 invalidates **F2**.

Proof:
The initial call makes $\mathsf{frozen}(\hat{s}) = \emptyset$. **F2** holds vacuously, because $\emptyset$ cannot contain a_n.

Now consider the call on line 9 that corresponds to $s \ -a\rightarrow_{\mathsf{r}} \ s'$. If $s' \ -a_1 \cdots a_n\rightarrow$ is a counter-example after line 2, then $s \ -aa_1 \cdots a_n\rightarrow$ was a counter-example beforehand, contradicting the assumption that **F2** held before the call.

Finally consider line 14. Let s be the current state, C be the tsr-component whose root is s, $a \in V$, and $s \ -a_1 \cdots a_n a\rightarrow$. Let us say that $s' \in C$ is *near* a if and only if $s' \ -\alpha a\rightarrow$ and α is as short as possible. At least some $s' \in C$ is near a. Because **F2** held beforehand, $a \notin \mathsf{frozen}(s')$. No element of α is in $\mathsf{frozen}(s')$, because otherwise by **F1**, α would not be a shortest. No element of α is in $\mathsf{warm}(s')$, because otherwise **D1F** would yield an r-transition $s' \ -u\rightarrow_{\mathsf{r}} \ s''$ and an α' such that $s'' \ -\alpha' a\rightarrow$ and $|\alpha'| < |\alpha|$. So no element of α is in $\mathsf{stubb}(s')$. Also $a \notin \mathsf{stubb}(s')$, because otherwise **D1F** would yield $s' \ -a\rightarrow_{\mathsf{r}}$, contradicting line 12.

So **D2F** can be applied to each $s' \ -u\rightarrow_{\mathsf{r}} \ s''$, yielding $s'' \ -\alpha a\rightarrow$. Also s'' is near a. Repeating this argument reveals that every $s \in C$ is near a. So $a \notin \bigcup_{s' \in C} \mathsf{stubb}(s')$ and the algorithm does not freeze a. □

By the above lemmas, and because **F1'** implies **F1**, we have the following.

Theorem 7.4. **F1** and **F2** hold throughout the execution of the algorithm in Fig. 4.

8. Construction of Warm Sets

The only remaining task is to describe how warm sets can be constructed such that together with the frozen sets they satisfy the local conditions **D0VF**, **D1F**, **D2F**, **D3F**, and **VF**. The sets should preferably yield good reduction results. Although the ideas are largely independent of the formalism used for representing systems, for concreteness, we will use the parallel state spaces formalism in Section 2. Let $L = (S, I, V, \Delta, \hat{s}) = L_1 \mid\mid \cdots \mid\mid L_N$.

By $\mathsf{en}_i(s_i)$ we mean the actions that L_i is ready to execute when it is in its local state s_i. That is, $\mathsf{en}_i(s_i) = \{a \mid \exists s'_i : (s_i, a, s'_i) \in \Delta_i\}$. It is the set of the labels of the edges of L_i whose tail is s_i. The

proof of the following theorem has been developed from [15] and earlier work. The key idea is that in case 1, L_i keeps a disabled until an element of $\mathsf{warm}(s)$ occurs, and in case 2, a is concurrent with all actions that are not in $\mathsf{stubb}(s)$.

Theorem 8.1. Assume that the following hold for $s = (s_1, \ldots, s_N)$ and for every $a \in \mathsf{warm}(s)$:

1. If $a \notin \mathsf{en}(s)$, then there is i such that $1 \leq i \leq N$, $a \in I_i \cup V_i$, and $a \notin \mathsf{en}_i(s_i) \subseteq \mathsf{stubb}(s)$.
2. If $a \in \mathsf{en}(s)$, then for every i such that $1 \leq i \leq N$ and $a \in I_i \cup V_i$ we have $\mathsf{en}_i(s_i) \subseteq \mathsf{stubb}(s)$.

Then $\mathsf{warm}(s)$ and $\mathsf{stubb}(s)$ satisfy **D1F**, **D2F**, and **D3F**.

Proof:
Let $a_1 \notin \mathsf{stubb}(s), \ldots, a_n \notin \mathsf{stubb}(s)$.

Let first $a \notin \mathsf{en}(s)$. Obviously $s\ {-a\!\rightarrow}$ does not hold, so **D2F** is vacuously true. We prove now that **D1F** and **D3F** are vacuously true as well. By condition 1, there is i such that L_i disables a and $\mathsf{en}_i(s_i) \subseteq \mathsf{stubb}(s)$. To enable a, it is necessary that L_i changes its state, which requires that some action in $\mathsf{en}_i(s_i)$ occurs. These are all in $\mathsf{stubb}(s)$ and thus distinct from $a_1, \ldots, a_n$. So $s\ {-a_1 \cdots a_n a\!\rightarrow}$ cannot hold.

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

This means that the L_j that participate in a are disjoint from the L_j that participate in $a_1 \cdots a_n$. From this **D1F** and **D2F** follow by well-known properties of the parallel composition operator. Regarding **D3F**, let $s_n = (s_{n,1}, \ldots, s_{n,N})$ and similarly with s'_n, z_n, and z'_n. Then $s_n \neq z_n$ implies $s_{n,j} \neq z_{n,j}$ for some L_j that participates in $a_1 \cdots a_n$. Because L_j does not participate in a, we have $s'_{n,j} = s_{n,j} \neq z_{n,j} = z'_{n,j}$, yielding $s'_n \neq z'_n$. □

Let $b \notin \mathsf{frozen}(s)$. Cases 1 and 2 can be interpreted as spanning rules of the form $a \rightsquigarrow_s b$ for each b in $\mathsf{en}_i(s_i)$, meaning that if $a \in \mathsf{warm}(s)$, then b must be in $\mathsf{stubb}(s)$. We do this only if $b \notin \mathsf{frozen}(s)$, because such a rule is unnecessary if $b \in \mathsf{frozen}(s)$, because $\mathsf{frozen}(s) \subseteq \mathsf{stubb}(s)$ by definition. So $a \rightsquigarrow_s b$ actually says that if $a \in \mathsf{warm}(s)$, then also b must be in $\mathsf{warm}(s)$. In case 1, there may be more than one i that satisfies the condition. Although each choice of such an i is correct, we artificially assumed in the examples earlier in this study that the smallest one is chosen, to avoid ambiguity.

To deal with **VF**, we also add the rule $a \rightsquigarrow_s b$ for every $a \in (\mathsf{en}(s) \cap V) \setminus \mathsf{frozen}(s)$ and $b \in V \setminus \mathsf{frozen}(s)$. (This is equivalent to adding yet another parallel component $(\{\hat{s}\}, \emptyset, V, \Delta, \hat{s})$ with $\Delta = \{(\hat{s}, a, \hat{s}) \mid a \in V\}$.)

Whether or not $a \rightsquigarrow_s b$, usually depends on the state s.

If $A \subseteq I \cup V$, let $\mathsf{clsr}_s(A)$ (the closure of A) denote the smallest set such that $A \setminus \mathsf{frozen}(s) \subseteq \mathsf{clsr}_s(A)$ and for every a and b, if $a \in \mathsf{clsr}_s(A)$ and $a \rightsquigarrow_s b$, then also $b \in \mathsf{clsr}_s(A)$. By the definitions,

$\mathsf{clsr}_s(A)$ contains no frozen actions. Furthermore, it satisfies **D1F**, **D2F**, **D3F**, and **VF** in s in the role of $\mathsf{warm}(s)$ (with, as always, $\mathsf{stubb}(s) = \mathsf{warm}(s) \cup \mathsf{frozen}(s)$).

If $s\ {-a_1 \cdots a_n}\!\!\rightarrow$ where $a_n \in V$ and $\{a_1, \ldots, a_n\} \cap \mathsf{frozen}(s) = \emptyset$, then $a_n \in \mathsf{clsr}_s(V)$. By **D1F**, $\{a_1, \ldots, a_n\} \cap \mathsf{clsr}_s(V) \cap \mathsf{en}(s) \neq \emptyset$. Therefore, to focus efforts on actions that are important for preserving all traces, it is reasonable to only use subsets of $\mathsf{clsr}_s(V)$ as the warm sets. In Section 6 we assumed that $compute\_stubborn(s, \mathsf{frozen}(s))$ returns a warm set $\mathsf{warm}(s)$ that obeys **D0VF**, that is, either $V \subseteq \mathsf{warm}(s) \cup \mathsf{frozen}(s)$ or $\mathsf{warm}(s) \cap \mathsf{en}(s) \neq \emptyset$. If $\mathsf{clsr}_s(V)$ contains no enabled actions, this is achieved by returning $\mathsf{clsr}_s(V)$, because always $V \subseteq \mathsf{clsr}_s(V) \cup \mathsf{frozen}(s)$.

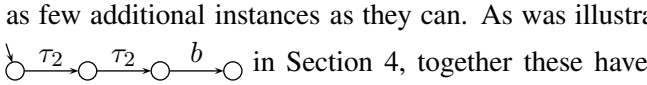
To guarantee **D0VF** in the opposite case, $\mathsf{clsr}_s(\{a\})$ could be returned for an arbitrary $a \in \mathsf{clsr}_s(V) \cap \mathsf{en}(s)$. To reduce the number of enabled actions in the resulting set, Tarjan's algorithm may be used to find a strong component of the graph spanned by "$\rightsquigarrow_s$" such that it contains an enabled action but no other strong component reachable from it contains enabled actions. This has been a standard approach in stubborn sets [7, 14, 11, 12], and a very efficient implementation exists in the tool described in [27] (using a "$\rightsquigarrow_s$"-relation derived from another formalism than the one in the present study). The "$\rightsquigarrow_s$" relation is not presented as an explicit directed graph; instead, it is derived as needed similarly to how we derived it from Theorem 8.1.

When the algorithm in Fig. 4 calls $compute\_stubborn$ again on the same state, the actions in the previous reply have been frozen. Therefore, $\mathsf{clsr}_s(V)$ has become smaller and $compute\_stubborn$ will return a new answer.

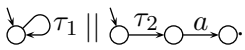
To improve reduction results, one may make "$\rightsquigarrow_s$" smaller as long as the property remains valid that every $\mathsf{clsr}_s(A)$ satisfies **D1F**, **D2F**, **D3F**, and **VF**. (Because of anomalic counter-examples, that this improves reduction is only a heuristic, not a theorem.) For instance, the purpose of the rules introduced by case 1 of Theorem 8.1 is to ensure that if a is put into the warm set, then it cannot become enabled without some action in the warm set occurring first. However, if L_i has no path from s_i to a state s_i' such that $a \in \mathsf{en}_i(s_i')$, then a cannot become enabled at all. This is the case with b and L_2 in Fig. 3, if b has already been executed. In such a situation, no rule of the form $a \rightsquigarrow_s x$ is needed, although case 1 implies that such a rule should be added for every $x \in \mathsf{en}_i(s_i) \setminus \mathsf{frozen}(s)$. If L_i models a fifo queue, a is enabled, one of a and b reads from and the other writes to the fifo, and L_i is their only shared component, then $a \rightsquigarrow_s b$ is not needed, despite case 2.

9. Conclusions

We presented a major improvement to partial order reduction methods for safety properties. Our method also covers a subset of branching time liveness properties including "in all futures always, there is a future where eventually a occurs".

Because of concurrency and other phenomena that make actions commutative (such as writing to and reading from a non-empty non-full fifo), if there is a counter-example to the property under verification, there often are many related counter-examples. To yield correct results, partial order reduction methods preserve at least one of them, and to obtain good reduction, they try to preserve as few additional instances as they can. As was illustrated with the example  in Section 4, together these have the consequence that the preserved counter-

example cannot always be a shortest possible. Instead, the preserved counter-example may contain actions that are irrelevant for it, but were fired because when they were fired, it was not certain that they are not relevant for any counter-example. In the example, firing initially only τ_1 lengthens the shortest preserved counter-example to "b never occurs", firing initially only τ_2 has the same effect to "a never occurs", and firing both would be bad for reduction.

Unless special precaution is taken, this leads to the following possibility. Assume that we want to check that a never occurs. Consider  The method may initially choose to focus on (in this case non-existent) counter-examples for which τ_1 is relevant, postponing the development of the other counter-examples to later r-states. However, the firing of τ_1 brings the system back to an r-state that has already been processed, so in the end τ_2 was investigated in no r-state and all counter-examples were lost. This is the ignoring problem. It has forced the introduction of additional conditions to partial order reduction methods, most importantly the cycle condition for liveness properties and terminal strong component conditions for safety properties. They solve the ignoring problem in the sense of guaranteeing correctness. Unfortunately, there are examples showing that they are sometimes very detrimental to reduction results, please see Section 4 and, e.g., [11, 12]. Little is known about how they affect reduction results in typical cases.

The major new contribution of [17] and the present study is the freezing of actions. Actions are frozen when they have been investigated but they proved irrelevant for all counter-examples and constituted a futile cycle. Thanks to the use of **D0VF**, most of the time (and, by Theorem 4.1, for a major class of systems all of the time) the method succeeds in avoiding such actions in the first place. However, because stubborn set construction algorithms cannot always determine whether or not an action is relevant, and because failure to pick a relevant action may compromise correctness, the methods sometimes pick irrelevant actions to be on the safe side. The example in Fig. 3 illustrates that when this happens and the irrelevant actions introduce a cycle in the reduced state space, the phenomenon tends to repeat in subsequent states, with bad consequences to reduction. The freezing of actions is a powerful solution to this problem. Frozen actions are treated as if they did not exist at all, preventing the repetition of the cycle.

Two decades ago, there were two different terminal strong component conditions. As was illustrated with Fig. 2, each of them suffers from a problem that the other one solves. To combine the good parts of both conditions, a complicated condition was employed in [11, 12] that in the root of a tsr-component selects a set X of enabled actions, each of which must be fired in some state of the tsr-component. To obtain good reduction results, if such an action has been fired elsewhere in the tsr-component, then, if possible, it should not be fired again in the root. The freezing of actions removes from consideration those actions that have been fired in the tsr-component, automatically ensuring that they are not fired again in the root. So the set X need not be implemented and discussed in the correctness proof.

Let us illustrate another problem that has not received sufficient attention. Consider $L_1 || L_2$, where each L_i consists of a cycle of $i+1$ τ_i-transitions preceded by an a-transition, and we want to verify that the system lacks the trace aa. Assuming that the stubborn set construction favours τ_1, the method in the present study constructs $\hat{s}\ {-a\!\rightarrow_{\mathsf{r}}}\ s\ {-\tau_1\tau_1\!\rightarrow_{\mathsf{r}}}\ s$, freezes τ_1, constructs $s\ {-\tau_2\tau_2\tau_2\!\rightarrow_{\mathsf{r}}}\ s$, freezes τ_2, and terminates. However, if the algorithm alternates between favouring τ_1 and τ_2, it constructs $\hat{s}\ {-a\!\rightarrow_{\mathsf{r}}}\ s\ {-\tau_1\tau_2\tau_1\tau_2\tau_1\tau_2\!\rightarrow_{\mathsf{r}}}\ s$, freezes both τ_1 and τ_2, and terminates, resulting in a bigger reduced state

space. In the absence of freezing, if the method favours τ_1, after constructing $\hat{s} \ -a\rightarrow_{\mathsf{r}} \ s \ -\tau_1\tau_1\rightarrow_{\mathsf{r}} \ s$ it would continue by constructing $s \ -\tau_2\rightarrow_{\mathsf{r}} \ s' \ -\tau_1\tau_1\rightarrow_{\mathsf{r}} \ s' \ -\tau_2\rightarrow_{\mathsf{r}} \ s'' \ -\tau_1\tau_1\rightarrow_{\mathsf{r}} \ s'' \ -\tau_2\rightarrow_{\mathsf{r}} \ s$. Freezing of actions prevents this last scenario, but not the previous one. Further research is needed. Our hypothesis is that the method should favour actions that belong to the same component as the action whose firing led to the creation of the current state.

Another apparently natural topic for further research would be to extend the method to linear time liveness properties. They are currently dealt with by the so-called cycle condition. However, this topic might be less fruitful than it seems, because linear time liveness properties tend to rely on so-called fairness assumptions, and, as was discussed in detail in [28], nobody has so far been able to reasonably combine them to partial order reduction. (For instance, fairness is not mentioned in the partial order reduction chapter of [1].)

On the other hand, a big subset of liveness properties is preserved by fair testing equivalence. This is the reason for giving it lots of attention in the present study. It preserves a representative for each counter-example that goes via a state after which the desired activity cannot happen. What it may lose is counter-examples where the only reason for the desired activity not happening is the scheduling of actions. That is, the system infinitely many times chooses an action (like loss of a message, or executing an irrelevant component) that does not make progress towards the desired thing, while an action that does make progress (delivery of the message, executing another component) would have been available. Although linear time liveness with fairness assumptions is the standard approach, we believe that it is less reliable than generally thought and alternative approaches may be worth consideration [29].

Of course, implementing the frozen action method and experimenting with it would be an important topic for future research. Among other things, a data structure for maintaining the frozen sets is needed. It may exploit the fact that only the r-states in the depth-first stack need a frozen set, and the frozen set of an r-state (other than $\hat{s}$) is always a superset of the frozen set of its parent.

Acknowledgements. We thank the anonymous reviewers of both RP '17 and Fundamenta Informaticae for their effort and comments.

References

[1] Clarke EM, Grumberg O, Peled DA. Model checking. MIT Press, 1999. ISBN 978-0-262-03270-4.

[2] Peled DA. All from One, One for All: on Model Checking Using Representatives. In: Courcoubetis C (ed.), Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings, volume 697 of *Lecture Notes in Computer Science*. Springer, 1993 pp. 409–423. doi:10.1007/3-540-56922-7_34.

[3] Peled DA. Partial order reduction: Linear and branching temporal logics and process algebras. In: Peled DA, Pratt VR, Holzmann GJ (eds.), 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*. DIMACS/AMS, 1996 pp. 233–258.

[4] Godefroid P. Using Partial Orders to Improve Automatic Verification Methods. In: Clarke EM, Kurshan RP (eds.), 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*. DIMACS/AMS, 1990 pp. 321–340.

[5] Godefroid P. 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. ISBN 3-540-60761-7. doi:10.1007/3-540-60761-7.

[6] Valmari A. Error Detection by Reduced Reachability Graph Generation. In: Proceedings of the 9th European Workshop on Application and Theory of Petri Nets. 1988 pp. 95–122.

[7] Valmari A. The State Explosion Problem. In: Reisig W, Rozenberg G (eds.), 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*. Springer, 1996 pp. 429–528. doi:10.1007/3-540-65306-6_21.

[8] Vogler W. Modular Construction and Partial Order Semantics of Petri Nets, volume 625 of *Lecture Notes in Computer Science*. Springer, 1992. ISBN 3-540-55767-9. doi:10.1007/3-540-55767-9.

[9] Rensink A, Vogler W. Fair testing. *Inf. Comput.*, 2007. **205**(2):125–198. doi:10.1016/j.ic.2006.06.002.

[10] Valmari A, Vogler W. Fair Testing and Stubborn Sets. *STTT*, 2017. doi:10.1007/s10009-017-0481-2.

[11] Valmari A, Hansen H. Stubborn Set Intuition Explained. In: Cabac L, Kristensen LM, Rölke H (eds.), 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*. CEUR-WS.org, 2016 pp. 213–232.

[12] Valmari A, Hansen H. Stubborn Set Intuition Explained. *T. Petri Nets and Other Models of Concurrency*, 2017. **12**:140–165. doi:10.1007/978-3-662-55862-1_7.

[13] Valmari A. Stubborn sets for reduced state space generation. In: Rozenberg G (ed.), Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings], volume 483 of *Lecture Notes in Computer Science*. Springer, 1989 pp. 491–515. doi:10.1007/3-540-53863-1_36.

[14] Valmari A. Stubborn set methods for process algebras. In: Peled DA, Pratt VR, Holzmann GJ (eds.), 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*. DIMACS/AMS, 1996 pp. 213–232.

[15] Valmari A, Vogler W. Fair Testing and Stubborn Sets. In: Bosnacki D, Wijs A (eds.), 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*. Springer, 2016 pp. 225–243. doi:10.1007/978-3-319-32582-8_16.

[16] Valmari A. More Stubborn Set Methods for Process Algebras. In: Gibson-Robinson T, Hopcroft PJ, Lazic R (eds.), 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*. Springer, 2017 pp. 246–271. doi:10.1007/978-3-319-51046-0_13.

[17] Valmari A. Stubborn Sets with Frozen Actions. In: Hague M, Potapov I (eds.), Reachability Problems, 11th International Workshop, RP 2017, volume 10506 of *Lecture Notes in Computer Science*. 2017 pp. 160–175. doi:10.1007/978-3-319-67089-8_12.

[18] Evangelista S, Pajault C. Solving the ignoring problem for partial order reduction. *STTT*, 2010. **12**(2):155–170. doi:10.1007/s10009-010-0137-y.

[19] Siegel SF. What's Wrong with On-the-Fly Partial Order Reduction. In: Dillig I, Tasiran S (eds.), Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II, volume 11562 of *Lecture Notes in Computer Science*. Springer, 2019 pp. 478–495. doi:10.1007/978-3-030-25543-5_27. URL .

[20] Holzmann GJ. The SPIN Model Checker - primer and reference manual. Addison-Wesley, 2004. ISBN 978-0-321-22862-8.

[21] Neele T, Valmari A, Willemse TAC. The Inconsistent Labelling Problem of Stutter-Preserving Partial-Order Reduction. *CoRR*, 2019. **abs/1910.09829**. , URL .

[22] Mazurkiewicz AW. Trace Theory. In: Brauer W, Reisig W, Rozenberg G (eds.), 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*. Springer, 1986 pp. 279–324. doi:10.1007/3-540-17906-2_30.

[23] Manna Z, Pnueli A. The temporal logic of reactive and concurrent systems - specification. Springer, 1992. ISBN 978-3-540-97664-6.

[24] Tarjan RE. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1972. **1**(2):146–160. doi:10.1137/0201010.

[25] Eve J, Kurki-Suonio R. On Computing the Transitive Closure of a Relation. *Acta Inf.*, 1977. **8**:303–314. doi:10.1007/BF00271339.

[26] Gabow HN. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 2000. **74**(3-4):107–114. doi:10.1016/S0020-0190(00)00051-X.

[27] Valmari A. A State Space Tool for Concurrent System Models Expressed in C++. In: Nummenmaa J, Sievi-Korte O, Mäkinen E (eds.), 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*. CEUR-WS.org, 2015 pp. 91–105.

[28] Valmari A. Stop It, and Be Stubborn! *ACM Trans. Embedded Comput. Syst.*, 2017. **16**(2):46:1–46:26. doi:10.1145/3012279.

[29] Valmari A, Hansen H. Progress Checking for Dummies. In: Howar F, Barnat J (eds.), Proceedings of Formal Methods for Industrial Critical Systems, volume 11119 of *Lecture Notes in Computer Science*. Springer, 2018 pp. 115–130. doi:10.1007/978-3-030-00244-2_8.