Correct and Efficient Antichain Algorithms for Refinement Checking

Technical Report

Maurice Laveaux, Jan Friso Groote, and Tim A.C. Willemse
{m.laveaux, j.f.groote, t.a.c.willemse}@tue.nl

Eindhoven University of Technology
De Groene Loper 5, 5612 AE, Eindhoven, The Netherlands

Abstract

Refinement checking plays an important role in system verification. This means that the correctness of the system is established by showing a refinement relation between two models; one for the implementation and one for the specification. In [22], Wang *et al.* describe an algorithm based on antichains for efficiently deciding stable failures refinement and failures-divergences refinement. We identify several issues pertaining to the soundness and performance in these algorithms and propose new, correct, antichain-based algorithms. Using a number of experiments we show that our algorithms outperform the original ones in terms of running time and memory usage. Furthermore, we show that applying divergence-preserving branching bisimulation reduction results in additional run time improvements.

1 Introduction

Refinement is often an integral part of a mature engineering methodology for designing a (software) system in a stepwise manner. It allows one to start from a high-level specification that describes the permitted and desired behaviours of a system and arrive at a detailed implementation that behaves according to this specification. While in many settings, refinement is often used rather informally, it forms the mathematical cornerstone in the theoretical development of the process algebra CSP (Communicating Sequential Processes) by Hoare [9, 13, 14].

This formal view on refinement—as a mathematical relation between a specification and its implementation—has been used successfully in industrial settings [6], and it has been incorporated in commercial Formal Model-Driven Engineering tools such as *Dezyne* [17]. In such settings there are a variety of refinement relations, each with their own properties. In particular, each notion of refinement offers specific guarantees on the (types of) behavioural properties of the specification that carry over to correct implementations. For the theory of CSP, the—arguably—most prominent refinement relations are the *stable failures refinement* [2, 14] and *failures-divergences refinement* [14]. Both are implemented in the FDR [5] tool for specifying and analysing CSP processes.

Both stable failures refinement and failures-divergences refinement are computationally hard problems; deciding whether there is a refinement relation between an implementation and specification, both represented by CSP processes or labelled transition systems, is PSPACE-hard [10]. In practice, however, tools such as FDR are able to work with quite large state spaces. The basic algorithm for deciding a stable failures refinement or a failures-divergences refinement between implementation and specification relies on a *normalisation* of the specification. This normalisation is achieved by a subset construction that is used to obtain a deterministic transition system which represents the specification.

As observed in [22] and inspired by successes reported, *e.g.*, in [1], *antichain* techniques can be exploited to improve on the performance of refinement checking algorithms. Unfortunately, a closer inspection of the results and algorithms in [22], reveals several issues. First, the definitions of stable

failures refinement and failures-divergences refinement used in [22] do not match the definitions of [2, 14], nor do they seem to match known relations from the literature [19].

Second, as we demonstrate in Example 4.5 in this paper, the results [22, Theorems 2 and 3] claiming correctness of their algorithms for deciding both refinement relations are incorrect. We do note that their algorithm for checking stable failures refinement correctly decides the refinement relation defined by [2, 14].

Third, unlike claimed by the authors, the algorithms of [22] violate the antichain property as we demonstrate in Example 4.6. Fourth, their algorithms suffer from severely degraded performance due to sub-optimal decisions made when designing the algorithms, leading to an overhead of a factor $|\Sigma|$, where Σ is the set of events, as we show in Example 4.5. This factor is even greater, *viz.* $|\Sigma|^{|S|}$, where S is the set of states of the implementation, when using a FIFO (first in, first out) queue to realise a breadth-first search strategy instead of the stack used for the depth-first search. Note that there are compelling reasons for using a breadth-first strategy [13]; *e.g.*, the conciseness of counterexamples to refinement.

The contributions of this technical report are as follows. Apart from pointing out the issues in [22], we propose new antichain-based algorithms for deciding stable failures refinement and failures-divergences refinement and we prove their correctness. We compare the performance of the stable failures refinement algorithm of [22] to ours. Due to the flaw in their algorithm for deciding failures-divergences refinement, a comparison of this refinement relation makes little sense. Our results indicate a small improvement in run time performance for practical models when using depth-first search, whereas our experiments using breadth-first search illustrate that decision problems intractable using the algorithm of [22] generally become quite easy using our algorithm. We also show that additional run time improvements can be obtained by applying divergence-preserving branching bisimulation reduction as a preprocessing step.

The rest of this technical report is organized as follows. In Section 2 the preliminaries of labelled transition systems and the refinement relations are defined. In Section 3 a general algorithm for checking refinement relations is described. In Section 4 this algorithm is extended with the antichain technique as shown in [22]. In Section 5 our corrected antichain algorithm is presented together with a proof of correctness. Finally, in Section 6 an experimental evaluation is conducted to show the effectiveness of these changes and the effectiveness of applying divergence-preserving branching bisimulation reduction.

2 Preliminaries

In this section the preliminaries of labelled transition systems, stable failures and failures-divergences refinement checking are presented.

2.1 Labelled transition systems

Let Σ be a finite set of events and let Σ_τ be equal to $\Sigma \cup \{\tau\}$, where τ indicates the *invisible* event. A *labelled transition system*, abbreviated as LTS, is defined as follows.

Definition 2.1. A labelled transition system is a tuple $\mathcal{L} = (S, \mathit{init}, \mathit{Act}, \rightarrow)$ where S is a set of states; $\mathit{init} \in S$ is an initial state; $\mathit{Act} = \Sigma$ or $\mathit{Act} = \Sigma_\tau$ is the set of events and $\rightarrow \subseteq S \times \mathit{Act} \times S$ is a labelled transition relation.

In the context of an LTS $(S, \mathit{init}, \mathit{Act}, \rightarrow)$ we use symbols s, t, u to denote states, U, V to denote sets of states and e to denote events. For states $s, t \in S$ and an event $e \in \mathit{Act}$, a transition $(s, e, t) \in \rightarrow$ is also written as $s \xrightarrow{e} t$.

Definition 2.2. Let $\mathcal{L} = (S, \mathit{init}, \mathit{Act}, \rightarrow)$ be an LTS. For every state $s \in S$ the set of *enabled* events is defined by $\mathsf{enabled}(s) = \{e \in \mathit{Act} \mid \exists t \in S : s \xrightarrow{e} t\}$.

The transition relation of an LTS is generalised to traces in the usual way.

Definition 2.3. Let $\mathcal{L} = (S, \mathit{init}, \mathit{Act}, \rightarrow)$ be an LTS. For all states $s, t \in S$ and sequences $\sigma \in \mathit{Act}^*$ we define $s \xrightarrow{\sigma}\!\!\!\!\rightarrow t$ as the smallest relation satisfying:

- $s \xrightarrow{\epsilon}\!\!\!\!\rightarrow s$, and

- $s \xrightarrow{\sigma e}\!\!\!\!\rightarrow t$ if there is a state $u \in S$ such that $s \xrightarrow{\sigma}\!\!\!\!\rightarrow u$ and $u \xrightarrow{e} t$.

A sequence σ such that $s \xrightarrow{\sigma}\!\!\!\!\rightarrow t$ is called a *trace*. The *traces* starting in s are defined by $\mathsf{traces}(s) = \{\rho \in Act^* \mid \exists t \in S : s \xrightarrow{\rho}\!\!\!\!\rightarrow t\}$. We define $\mathsf{traces}(\mathcal{L})$ to be $\mathsf{traces}(\mathit{init})$.

The *length* of a sequence, denoted as $|e_0 e_1 \cdots e_{n-1}|$, is equal to n. In particular, the length of the empty sequence, denoted as $|\epsilon|$, is zero. For any sequence $e_0 e_1 \cdots e_{n-1}$ we say that any sequence $e_0 e_1 \cdots e_k$ such that $k < n-1$ is a *prefix* of $e_0 e_1 \cdots e_{n-1}$.

A *weak* trace is a trace where invisible events, denoted by τ, are ignored.

Definition 2.4. Let $\mathcal{L} = (S, \mathit{init}, Act, \rightarrow)$ be an LTS. For all states $s, t \in S$, events $e \in \Sigma$ and sequences $\rho \in \Sigma^*$ we define the *weak transition* relation $s \overset{\rho}{\rightsquigarrow} t$ is the smallest relation satisfying:

- $s \overset{\epsilon}{\rightsquigarrow} s$,
- $s \overset{\epsilon}{\rightsquigarrow} t$ if $s \xrightarrow{\tau} t$,
- $s \overset{e}{\rightsquigarrow} t$ if $s \xrightarrow{e} t$, and
- $s \overset{\rho\sigma}{\rightsquigarrow} t$ if there is a state $u \in S$ such that $s \overset{\rho}{\rightsquigarrow} u$ and $u \overset{\sigma}{\rightsquigarrow} t$.

A sequence ρ such that $s \overset{\rho}{\rightsquigarrow} t$ is called a *weak trace*. The weak traces starting in s are defined by $\mathsf{weaktraces}(s) = \{\rho \in \Sigma^* \mid \exists t \in S : s \overset{\rho}{\rightsquigarrow} t\}$. We define $\mathsf{weaktraces}(\mathcal{L})$ to be $\mathsf{weaktraces}(\mathit{init})$.

Definition 2.5. Let $\mathcal{L} = (S, \mathit{init}, Act, \rightarrow)$ be an LTS. For every state $s \in S$ its *reachable* states are defined by $\mathsf{reachable}(s) = \{t \in S \mid \exists \sigma \in Act^* : s \xrightarrow{\sigma}\!\!\!\!\rightarrow t\}$. We define $\mathsf{reachable}(\mathcal{L})$ to be $\mathsf{reachable}(\mathit{init})$.

For labelled transition systems we conclude with the definition of *determinism*.

Definition 2.6. Let $\mathcal{L} = (S, \mathit{init}, Act, \rightarrow)$ be an LTS. We say that $\mathcal{L}$ is *deterministic* if and only if for all states $s, t, t' \in S$ if there are transitions $s \xrightarrow{e} t$ and $s \xrightarrow{e} t'$ then $t = t'$.

2.2 Refinement

The CSP process algebra builds on observations of *failures* and *divergences*. A failure is a set of events that a system observably refuses following an experiment on that system. By assumption, refusals can only be observed whenever the system has *stabilised*. In the operational semantics, described as an LTS, a failure is equivalent to not accepting certain labels in a *stable* state after following a weak trace. We follow the definitions of [3, 21].

Definition 2.7. Let $\mathcal{L} = (S, \mathit{init}, Act, \rightarrow)$ be an LTS. A state $s \in S$ is *stable*, denoted by $\mathsf{stable}(s)$, if and only if $\tau \notin \mathsf{enabled}(s)$.

The inputs that are no longer accepted in a stable state are *refused* by the system.

Definition 2.8. Let $\mathcal{L} = (S, \mathit{init}, Act, \rightarrow)$ be an LTS. For every stable state $s \in S$ its *refusals* are defined by $\mathsf{refusals}(s) = 2^{\Sigma \setminus \mathsf{enabled}(s)}$. For a set of states $U \subseteq S$ its refusals are defined by $\mathsf{refusals}(U) = \{X \subseteq \Sigma \mid s \in U \wedge \mathsf{stable}(s) \wedge X \in \mathsf{refusals}(s)\}$.

We observe that in [22] a different definition was presented where for a stable state $s \in S$ the refusals are defined by $\mathsf{refusals}(s) = \{X \mid \exists s' \in S : (s \overset{\epsilon}{\rightsquigarrow} s' \wedge \mathsf{stable}(s') \wedge X \subseteq \Sigma \setminus \mathsf{enabled}(s'))\}$. This definition is ambiguous in the requirement that s must be stable. If s is required to be a stable state then the only state $s' \in S$ such that $s \overset{\epsilon}{\rightsquigarrow} s'$ is s itself by definition and as such the existential quantification is redundant. If this redundant quantification is removed we obtain Definition 2.8. Otherwise, this definition defines refusals for any state and the existential quantification is required. In [22], for a set of states $U \subseteq S$ the refusals of U, denoted by $\mathsf{refusals}(U)$, are defined as $\{X \mid \exists s \in U : X \in \mathsf{refusals}(s)\}$, which has no requirement on the stability of s. This observation also has an impact on the algorithm as shown in Section 4.

A *divergence* can be understood as the inability of a system to stabilise by having the possibility to perform an infinite sequence of invisible events.

Definition 2.9. Let $\mathcal{L} = (S, init, Act, \rightarrow)$ be an LTS. A state $s \in S$ *diverges*, denoted by $\mathsf{div}(s)$, if and only if there is an infinite sequence of states $s \xrightarrow{\tau} s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots$. A set of states $U \subseteq S$ diverges if and only if it contains a diverging state, defined as $\mathsf{div}(U) = \exists s \in U : \mathsf{div}(s)$.

In the semantics of CSP the ability to diverge is seen as *catastrophic*. This means that all information about behaviour (sequences) after a divergent state is lost. This can be achieved by *flooding*, which means that all failures and divergences past a divergence are added, regardless of whether the LTS contains that behaviour.

Definition 2.10. Let $\mathcal{L} = (S, init, Act, \rightarrow)$ be an LTS. The *divergences* of a state $s \in S$ are defined by $\mathsf{divergences}(s) = \{\rho\sigma \in \Sigma^* \mid \exists t \in S : (s \overset{\rho}{\rightsquigarrow} t \wedge \mathsf{div}(t))\}$. The *minimal* divergences of a state $s \in S$, denoted by $\mathsf{divergences}_\perp(s)$, are the subset of $\mathsf{divergences}(s)$ where for all $\rho \in \mathsf{divergences}_\perp(s)$ there is no prefix of ρ in $\mathsf{divergences}(s)$. We define $\mathsf{divergences}(\mathcal{L})$ to be $\mathsf{divergences}(init)$ and $\mathsf{divergences}_\perp(\mathcal{L})$ to be $\mathsf{divergences}_\perp(init)$.

Definition 2.11. Let $\mathcal{L} = (S, init, Act, \rightarrow)$ be an LTS. The stable failures of $\mathcal{L}$ are defined by $\mathsf{failures}(\mathcal{L}) = \{(\rho, X) \in \Sigma^* \times 2^\Sigma \mid \exists s \in S : (init \overset{\rho}{\rightsquigarrow} s \wedge \mathsf{stable}(s) \wedge X \in \mathsf{refusals}(s))\}$. The set of failures with post-divergence details *obscured* is defined by $\mathsf{failures}_\perp(\mathcal{L}) = \mathsf{failures}(\mathcal{L}) \cup \{(\rho, X) \in \Sigma^* \times 2^\Sigma \mid \rho \in \mathsf{divergences}(\mathcal{L})\}$.

The two models of CSP obtained from these observations are the *stable failures* model and the *failures-divergences* model. The refinement relations, induced by these models, are called *stable failures refinement* and *failures-divergences refinement*. For LTSs these refinement relations are obtained from the failures and divergences extracted from them. The LTS that is refined is referred to as the *specification*, whereas the LTS that refines the specification is referred to as the *implementation*.

Definition 2.12. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be LTSs. Then $\mathcal{L}_1$ refines $\mathcal{L}_2$ in stable failures semantics, denoted by $\mathcal{L}_1 \sqsupseteq_{\mathsf{sfr}} \mathcal{L}_2$, if and only if $\mathsf{failures}(\mathcal{L}_1) \subseteq \mathsf{failures}(\mathcal{L}_2)$ and $\mathsf{weaktraces}(\mathcal{L}_1) \subseteq \mathsf{weaktraces}(\mathcal{L}_2)$.

Definition 2.13. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be LTSs. Then $\mathcal{L}_1$ refines $\mathcal{L}_2$ in failures-divergences semantics, denoted by $\mathcal{L}_1 \sqsupseteq_{\mathsf{fdr}} \mathcal{L}_2$, if and only if $\mathsf{failures}_\perp(\mathcal{L}_1) \subseteq \mathsf{failures}_\perp(\mathcal{L}_2)$ and $\mathsf{divergences}(\mathcal{L}_1) \subseteq \mathsf{divergences}(\mathcal{L}_2)$.

We remark that we write the implementation LTS on the left and the specification LTS on the right of $\sqsupseteq_{\mathsf{fdr}}$, whereas the literature would define $\mathcal{L}_2 \sqsubseteq_{\mathsf{fdr}} \mathcal{L}_1$. We also remark that both stable failures and failures-divergences refinement are defined differently in [22]. In [22], the definition of stable failures refinement does not require weak trace inclusion, and the definition of failures-divergences refinement uses $\mathsf{failures}$ instead of $\mathsf{failures}_\perp$. These alternative definitions are different from the standard ones presented in the literature [21].

We finish with a small example illustrating the differences between stable failures refinement and failures-divergences refinement.

Example 2.14. Consider the two transition systems depicted below.

For simplicity, we use the initial state when referring to an LTS. Observe that we have $t_0 \sqsupseteq_{\mathsf{fdr}} s_0$, but not $t_0 \sqsupseteq_{\mathsf{sfr}} s_0$. The latter fails because aa is a trace of t_0, but not of s_0; the same goes for the stable failure $(a, \{b\})$ of t_0. The failures-divergences refinement holds because the divergent trace a obfuscates the observations of traces of the form aa^+: since divergence is catastrophic, anything is permitted. We do have $s_0 \sqsupseteq_{\mathsf{sfr}} t_0$ but not $s_0 \sqsupseteq_{\mathsf{fdr}} t_0$. The latter fails because the divergence a of s_0 is not present in t_0. Stable failures refinement holds because all traces and stable failure pairs of s_0 are included in those of t_0; in particular, the instability of state s_1 causes s_1 not to contribute to the stable failures set of s_0. □

2.3 Operations on LTSs

In this section, we define a number of operations on labelled transition systems that are needed for the refinement checking algorithms presented later on. Each algorithm explores a *synchronous product* between the two LTSs on which refinement is checked.

Definition 2.15. Let $\mathcal{L}_1 = (S_1, init_1, Act, \rightarrow_1)$ and $\mathcal{L}_2 = (S_2, init_2, \Sigma, \rightarrow_2)$ be LTSs. The *synchronous product* of $\mathcal{L}_1$ and $\mathcal{L}_2$, denoted by $\mathcal{L}_1 \times \mathcal{L}_2$, is an LTS $\mathcal{L} = (S, init, Act, \rightarrow)$ such that $S = S_1 \times S_2$ and $init = (init_1, init_2)$. The transition relation $\rightarrow$ is the smallest relation that satisfies the following conditions. For all $s_1, t_1 \in S_1$ and $s_2, t_2 \in S_2$:

- If $s_1 \xrightarrow{\tau}_1 t_1$ then $(s_1, s_2) \xrightarrow{\tau} (t_1, s_2)$.
- If $s_1 \xrightarrow{e}_1 t_1$ and $s_2 \xrightarrow{e}_2 t_2$ then $(s_1, s_2) \xrightarrow{e} (t_1, t_2)$.

A non deterministic specification LTS contains multiple states that can be reached by following a weak trace from the initial state. The algorithm can be simplified by only having a single state corresponding to each weak trace. This can be achieved by a *normalization* procedure. The normalisation of an LTS is strongly related to the *determinisation* of an LTS. For determinisation, an LTS can be translated to a trace equivalent deterministic LTS by means of a subset construction. In this construction the states reachable by the same trace are grouped into the same set, which is again reachable by that trace. For stable failures and failures-divergences refinement a different subset construction is used that results in a *normalised* LTS. This normalisation also yields a deterministic LTS, but compared to the typical determinisation it has different properties.

Definition 2.16. Let $\mathcal{L} = (S, init, Act, \rightarrow)$ be an LTS. The normalisation of $\mathcal{L}$, denoted by $\mathsf{norm}_{\mathsf{sfr}}(\mathcal{L})$, is the LTS $\mathcal{L}' = (S', init', \Sigma, \rightarrow')$ where $S' = 2^S$, $init' = \{s \in S \mid init \overset{\epsilon}{\rightsquigarrow} s\}$, and $\rightarrow'$ is defined as for all sets of states $U, V \subseteq S$ and events $e \in \Sigma$ there is a transition $U \xrightarrow{e}' V$ if and only if $V = \{t \in S \mid \exists s \in U : s \overset{e}{\rightsquigarrow} t\}$.

We deliberately permit the empty set to be a state of the normalised LTS. Clearly, a normalised LTS satisfies $\emptyset \xrightarrow{e}' \emptyset$ for all events $e \in \Sigma$. Essentially, a transition to the empty set from a state U indicates that this event cannot be performed from any of the corresponding states of the original LTS in U. For failures-divergences refinement an alternative normalisation that does not contain the weak traces of the original LTS past a minimal divergence. While all states in an LTS in normal form are stable and not diverging, the states of the original LTS comprising a normal form state may not be. To avoid confusion when we wish to reason about the stability and divergences of states U in the LTS $\mathcal{L}$ underlying a normal form LTS, rather than the state of the normal form LTS, we write $\bar{U}$ to indicate we refer to the set of states in $\mathcal{L}$.

Definition 2.17. Given an LTS $\mathcal{L} = (S, init, Act, \rightarrow)$. The normalisation of $\mathcal{L}$, denoted by $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L})$, is the LTS $\mathcal{L}' = (S', init', \Sigma, \rightarrow')$ where $S' = 2^S$, $init' = \{s \in S \mid init \overset{\epsilon}{\rightsquigarrow} s\}$, and $\rightarrow'$ is defined as for all sets of states $U, V \subseteq S$ and events $e \in \Sigma$ there is a transition $U \xrightarrow{e}' V$ if and only if $\neg \mathsf{div}(\bar{U})$ and $V = \{t \in S \mid \exists s \in U : s \overset{e}{\rightsquigarrow} t\}$.

2.4 Properties

Next, we provide a number of characteristic properties of the definitions given above that are required in further proofs. We start with several standard properties of the synchronous product.

Lemma 2.18. Let $\mathcal{L}_1 = (S_1, init_1, Act, \rightarrow_1)$ and $\mathcal{L}_2 = (S_2, init_2, \Sigma, \rightarrow_2)$ be LTSs and let $\mathcal{L} = (S, init, Act, \rightarrow)$ be equal to $\mathcal{L}_1 \times \mathcal{L}_2$. For all states $(s_1, s_2), (t_1, t_2) \in S$ there is a weak transition $(s_1, s_2) \overset{\epsilon}{\rightsquigarrow} (t_1, t_2)$ if and only if $s_1 \overset{\epsilon}{\rightsquigarrow}_1 t_1$ and $s_2 = t_2$.

Proof. The proof can be obtained from induction on the length of all traces in τ^*. Extending this path is a trivial application of the τ-case of the synchronous product. □

Lemma 2.19. Let $\mathcal{L}_1 = (S_1, init_1, Act, \rightarrow_1)$ and $\mathcal{L}_2 = (S_2, init_2, \Sigma, \rightarrow_2)$ be LTSs and let $\mathcal{L} = (S, init, Act, \rightarrow)$ be equal to $\mathcal{L}_1 \times \mathcal{L}_2$. For all states $(s_1, s_2), (t_1, t_2) \in S$ and events $e \in \Sigma$ it holds that $(s_1, s_2) \overset{e}{\rightsquigarrow} (t_1, t_2)$ if and only if $s_1 \overset{e}{\rightsquigarrow}_1 t_1$ and $s_2 \xrightarrow{e}_2 t_2$.

Proof. Take arbitrary states $(s_1, s_2), (t_1, t_2) \in S$ and event $e \in \Sigma$.

$\Longrightarrow$). Assume that $(s_1, s_2) \overset{e}{\rightsquigarrow} (t_1, t_2)$ holds. As such there are transitions $(s_1, s_2) \overset{\epsilon}{\rightsquigarrow} (s_1', s_2)$, $(s_1', s_2) \overset{e}{\rightarrow} (t_1', t_2)$ and $(t_1', t_2) \overset{\epsilon}{\rightsquigarrow} (t_1, t_2)$ from the definition of a weak transition and Lemma 2.18. From the definition of the synchronous product and existence of $(s_1', s_2) \overset{e}{\rightarrow} (t_1', t_2)$ there are transitions $s_1' \overset{e}{\rightarrow}_1 t_1'$ and $s_2 \overset{e}{\rightarrow}_2 t_2$. From the existence of $(s_1, s_2) \overset{\epsilon}{\rightsquigarrow} (s_1', s_2)$ and $(t_1', t_2) \overset{\epsilon}{\rightsquigarrow} (t_1, t_2)$ and by Lemma 2.18 there are weak transitions $s_1 \overset{\epsilon}{\rightsquigarrow}_1 s_1'$ and $t_1' \overset{\epsilon}{\rightsquigarrow}_1 t_1$. Finally, from the definition of weak transition follows $s_1 \overset{e}{\rightsquigarrow}_1 t_1$.

$\Longleftarrow$) Similar. □

These properties can now be extended to sequences of events.

Lemma 2.20. Let $\mathcal{L}_1 = (S_1, init_1, Act, \rightarrow_1)$ and $\mathcal{L}_2 = (S_2, init_2, \Sigma, \rightarrow_2)$ be LTSs and let $\mathcal{L} = (S, init, Act, \rightarrow)$ be equal to $\mathcal{L}_1 \times \mathcal{L}_2$. For all states $s_1, t_1 \in S_1$ and $s_2, t_2 \in S_2$ and all sequences $\rho \in \Sigma^*$ it holds that $(s_1, s_2) \overset{\rho}{\rightsquigarrow} (t_1, t_2)$ if and only if $s_1 \overset{\rho}{\rightsquigarrow}_1 t_1$ and $s_2 \overset{\rho}{\twoheadrightarrow}_2 t_2$.

Proof. Proof by induction on the length of all sequences in Σ^*. The induction hypothesis is that for all states $s_1, t_1 \in S_1$ and $s_2, t_2 \in S_2$ and sequences $\sigma \in \Sigma^*$ of length k it holds that $(s_1, s_2) \overset{\sigma}{\rightsquigarrow} (t_1, t_2)$ iff $s_1 \overset{\sigma}{\rightsquigarrow}_1 t_1$ and $s_2 \overset{\sigma}{\twoheadrightarrow}_2 t_2$.

Base case, the empty sequence ϵ of length zero. Take arbitrary states $s_1, t_1 \in S_1$ and $s_2, t_2 \in S_2$ such that $(s_1, s_2) \overset{\epsilon}{\rightsquigarrow} (t_1, t_2)$. By Lemma 2.18 there is a weak transition $(s_1, s_2) \overset{\epsilon}{\rightsquigarrow} (t_1, t_2)$ if and only if $s_1 \overset{\epsilon}{\rightsquigarrow}_1 t_1$ and $s_2 = t_2$.

Inductive step. Suppose that the induction hypothesis holds for all sequences $\rho \in \Sigma^*$ of length i.

$\Longrightarrow$) Take arbitrary states $s_1, u_1 \in S_1$ and $s_2, u_2 \in S_2$ and event $e \in \Sigma$ such that $(s_1, s_2) \overset{\rho e}{\rightsquigarrow} (u_1, u_2)$. By the definition of a weak transition and the product there are states $t_1 \in S_1$ and $t_2 \in S_2$ such that there is a state $(t_1, t_2) \in S$ where $(s_1, s_2) \overset{\rho}{\rightsquigarrow} (t_1, t_2)$ and $(t_1, t_2) \overset{e}{\rightsquigarrow} (u_1, u_2)$ hold. By the induction hypothesis we know that $s_1 \overset{\rho}{\rightsquigarrow}_1 t_1$ and $s_2 \overset{\rho}{\twoheadrightarrow}_2 t_2$. From Lemma 2.19 we know that $t_1 \overset{e}{\rightsquigarrow}_1 u_1$ and $t_2 \overset{e}{\rightarrow}_2 u_2$. From the definitions of traces and weak traces follows that $s_1 \overset{\rho e}{\rightsquigarrow}_1 u_1$ and $s_2 \overset{\rho e}{\twoheadrightarrow}_2 u_2$.

$\Longleftarrow$) Similar. □

Next, we show a number of characteristic properties of the normalised LTS resulting from $\mathsf{norm}_{\mathsf{fdr}}$.

Lemma 2.21. Let $\mathcal{L}$ be an LTS and let $\mathcal{L}' = (S', init', \Sigma, \rightarrow')$ be equal to $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L})$. For all sequences $\sigma \in \Sigma^*$ and states $U, V, W \in S'$ if $U \overset{\sigma}{\twoheadrightarrow}' V$ and $U \overset{\sigma}{\twoheadrightarrow}' W$ then $V = W$.

Proof. For every event $e \in \Sigma$ and all states $U \in S'$ there can not be $V, W \in S'$ such that $U \overset{e}{\rightarrow}' V$ and $U \overset{e}{\rightarrow}' W$ but $V \neq W$ by definition of the transition relation. Therefore $\mathcal{L}'$ is deterministic. Now, the proof follows from induction on the length of all traces of $\mathcal{L}'$. □

Lemma 2.22. Let $\mathcal{L} = (S, init, Act, \rightarrow)$ be an LTS and let $\mathcal{L}' = (S', init', \Sigma, \rightarrow')$ be equal to $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L})$. For all sequences $\rho \in \Sigma^*$ and states $U \in S'$ such that $init' \overset{\rho}{\twoheadrightarrow}' U$ it holds that $\forall s \in \bar{U} : init \overset{\rho}{\rightsquigarrow} s$.

Proof. Proof by induction on the length of all sequences in Σ^*. The induction hypothesis is that for all sequences $\sigma \in \Sigma^*$ of length k for all states $U \in S'$ such that $init' \overset{\sigma}{\twoheadrightarrow}' U$ it holds that $\forall s \in \bar{U} : init \overset{\sigma}{\rightsquigarrow} s$.

Base case, the empty sequence has a length of zero. By definition only $init' \overset{\epsilon}{\twoheadrightarrow}' \overline{init}'$. The state $init'$ is equal to $\{s \mid init \overset{\epsilon}{\rightsquigarrow} s\}$ by definition of normalisation. Thus for every state $s \in \overline{init}'$ there is a weak transition $init \overset{\epsilon}{\rightsquigarrow} s$ by definition.

Inductive case. Suppose that the induction hypothesis holds for all sequences $\rho \in \Sigma^*$ of length i. Take an arbitrary state $V \in S'$ and event $e \in \Sigma$ such that $init' \overset{\rho e}{\twoheadrightarrow}' V$. By definition of a transition there is a state $U \in S'$ such that $init' \overset{\rho}{\twoheadrightarrow}' U$ and $U \overset{e}{\rightarrow}' V$. By definition of normalisation there is a transition $U \overset{e}{\rightarrow}' V$ if and only if $V = \{s \mid \exists t \in U : t \overset{e}{\rightsquigarrow} s\}$ and $\neg\mathsf{div}(\bar{U})$. For all states $s \in \bar{V}$ there is a state $t \in \bar{U}$ such that $t \overset{e}{\rightsquigarrow} s$ by definition. By the induction hypothesis it holds that for all $t \in \bar{U}$ there is a weak transition $init \overset{\rho}{\rightsquigarrow} t$. Finally, by definition of a weak transition it holds that $init \overset{\rho e}{\rightsquigarrow} s$. □

Lemma 2.23. Let $\mathcal{L} = (S, init, Act, \rightarrow)$ be an LTS and $\mathcal{L}' = (S', init', \Sigma, \rightarrow')$ the result of $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L})$. For all sequences $\rho \in \Sigma^*$ such that $\rho \notin \mathsf{divergences}(\mathcal{L})$ or $\rho \in \mathsf{divergences}_\perp(\mathcal{L})$ and for all states $s \in S$ such that $init \overset{\rho}{\rightsquigarrow} s$ there is a state $U \in S'$ such that $s \in \bar{U}$ and $init \overset{\rho}{\twoheadrightarrow}' U$.

Proof. Proof by induction on the length of all sequences that are not divergences or minimal divergences. The induction hypothesis is that for all sequences $\sigma \in \Sigma^*$ that satisfy $\sigma \notin \mathsf{divergences}(\mathcal{L})$ or $\sigma \in \mathsf{divergences}_\perp(\rho)$ of length k that for all states $s \in S$ if $\mathit{init} \overset{\sigma}{\rightsquigarrow} s$ then there is a state $U \in S'$ such that $s \in \bar{U}$ and $\mathit{init}' \overset{\sigma}{\twoheadrightarrow}' U$.

Base case. The empty trace ϵ of length zero satisfies $\epsilon \notin \mathsf{divergences}(\mathcal{L})$ or $\epsilon \in \mathsf{divergences}_\perp(\rho)$ by definition. We know that if $\mathit{init} \overset{\epsilon}{\rightsquigarrow} s$ then $s \in \overline{\mathit{init}'}$, because init' is defined as $\{s \mid \mathit{init} \overset{\epsilon}{\rightsquigarrow} s\}$ in the normalisation. We also know that $\mathit{init}' \overset{\epsilon}{\twoheadrightarrow} \mathit{init}'$ by definition.

Inductive step. Suppose that the induction hypothesis holds for sequences $\rho \in \Sigma^*$ of length i that are not divergences or minimal divergences. Take an arbitrary state $t \in S$ and event $e \in \Sigma$ such that $\mathit{init} \overset{\rho e}{\rightsquigarrow} t$ and $\rho e \notin \mathsf{divergences}(\mathcal{L})$ or $\rho e \in \mathsf{divergences}_\perp(\mathcal{L})$. By definition of a weak transition there is a state $s \in S$ such that $\mathit{init} \overset{\rho}{\rightsquigarrow} s$ and $s \overset{e}{\rightsquigarrow} t$. By the induction hypothesis there is a state $U \in S'$ such that $s \in \bar{U}$ and $\mathit{init} \overset{\rho}{\twoheadrightarrow}' U$. If $\rho e \notin \mathsf{divergences}(\mathcal{L})$ then $\rho \notin \mathsf{divergences}(\mathcal{L})$ by definition of divergences and whenever $\rho e \in \mathsf{divergences}_\perp(\mathcal{L})$ then $\rho \notin \mathsf{divergences}_\perp(\mathcal{L})$ by definition of a minimal divergence. In both cases $\neg\mathsf{div}(\bar{U})$ holds by Lemma 2.22 and the definition of divergences. Let V be equal to $\{t \mid \exists s \in U : s \overset{e}{\rightsquigarrow} t\}$ such that $U \overset{e}{\rightarrow}' V$ by definition of the normalisation. From the definition of a transition follows that $\mathit{init} \overset{\rho e}{\twoheadrightarrow}' V$. Finally, $t \in \bar{V}$ follows from the existence of a weak transition $s \overset{e}{\rightsquigarrow} t$. □

Lemma 2.24. Let $\mathcal{L}$ be an LTS and let $\mathcal{L}' = (S', \mathit{init}', \Sigma, \rightarrow')$ be equal to $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L})$. For all sequences $\rho \in \Sigma^*$ and states $U \in S'$ it holds that if $\mathit{init}' \overset{\rho}{\twoheadrightarrow}' U$ and $\neg\mathsf{div}(\bar{U})$ then $\rho \notin \mathsf{divergences}(\mathcal{L})$.

Proof. Proof by induction on the length of all sequences in Σ^*. The induction hypothesis is that for all sequences $\sigma \in \Sigma^*$ of length k and states $U \in S'$ if $\mathit{init}' \overset{\sigma}{\twoheadrightarrow}' U$ and $\neg\mathsf{div}(\bar{U})$ then $\sigma \notin \mathsf{divergences}(\mathcal{L})$.

Base case. The empty trace ϵ of length zero. By definition, only $\mathit{init}' \overset{\epsilon}{\twoheadrightarrow}' \mathit{init}'$. We know that $\mathit{init}' = \{t \in S \mid \mathit{init} \overset{\epsilon}{\rightsquigarrow} t\}$ by the definition of normalisation. If $\neg\mathsf{div}(\overline{\mathit{init}'})$ then for all $t \in S$ such that $\mathit{init} \overset{\epsilon}{\rightsquigarrow} t$ it holds that $\neg\mathsf{div}(t)$ by definition of diverges and by construction. Thus $\epsilon \notin \mathsf{divergences}(\mathcal{L})$ by definition of divergences.

Inductive step. Assume that the induction hypothesis holds for all sequence $\rho \in \Sigma^*$ of length i. Take an arbitrary state $V \in S'$ and event $e \in \Sigma$ such that $\mathit{init}' \overset{\rho e}{\twoheadrightarrow}' V$ and $\neg\mathsf{div}(\bar{V})$. There is a state $U \in S'$ such that $\mathit{init}' \overset{\rho}{\twoheadrightarrow}' U$ and $U \overset{e}{\rightarrow}' V$ by definition. From the existence of $U \overset{e}{\rightarrow}' V$ follows that $\neg\mathsf{div}(\bar{U})$ by definition of the normalisation. From the induction hypothesis follows that $\rho \notin \mathsf{divergences}(\mathcal{L})$. From lemmas 2.21 and 2.23 it follows for any state $t \in S$ such that $\mathit{init} \overset{\rho e}{\rightsquigarrow} t$ that $t \in \bar{V}$ and thus from $\neg\mathsf{div}(\bar{V})$ follows that $\neg\mathsf{div}(t)$ by definition of diverges. Therefore, $\rho e \notin \mathsf{divergences}(\mathcal{L})$ by definition of divergences. □

Lemma 2.25. Let $\mathcal{L}$ be an LTS and let $\mathcal{L}' = (S', \mathit{init}', \Sigma, \rightarrow')$ be equal to $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L})$. For all sequences $\rho \in \Sigma^*$ it holds that $\rho \notin \mathsf{divergences}(\mathcal{L}) \cup \mathsf{weaktraces}(\mathcal{L})$ if and only if $\mathit{init}' \overset{\rho}{\twoheadrightarrow}' \emptyset$.

Proof. For the empty trace ϵ of length zero. By definition $\epsilon \notin \mathsf{weaktraces}(\mathcal{L})$ if and only if init' is empty, as init' is equal to $\{s \in S \mid \mathit{init} \overset{\epsilon}{\rightsquigarrow} s\}$ in the normalisation and only $\mathit{init}' \overset{\epsilon}{\twoheadrightarrow} \mathit{init}'$ by definition.

$\Longrightarrow$). Proof by induction on the length of all sequences in $\rho \in \Sigma^*$. The induction hypothesis is that for all sequences $\sigma \in \Sigma^*$ of length k if $\sigma \notin \mathsf{divergences}(\mathcal{L}) \cup \mathsf{weaktraces}(\mathcal{L})$ then $\mathit{init}' \overset{\sigma}{\twoheadrightarrow}' \emptyset$.

The base case was already established, so for the inductive step. Suppose that the induction hypothesis holds for all sequences $\rho \in \Sigma^*$ of length i. Assume an arbitrary event $e \in \Sigma$ such that $\rho e \notin \mathsf{divergences}(\mathcal{L}) \cup \mathsf{weaktraces}(\mathcal{L})$. From $\rho e \notin \mathsf{weaktraces}(\mathcal{L})$ follows that there is no state $t \in S$ such that $\mathit{init} \overset{\rho e}{\rightsquigarrow} t$ by definition. From $\rho e \notin \mathsf{divergences}(\mathcal{L})$ follows that $\rho \notin \mathsf{divergences}(\mathcal{L})$ by definition. Now there are two cases to distinguish.

Case $\rho \notin \mathsf{weaktraces}(\mathcal{L})$. From the induction hypothesis follows that $\mathit{init}' \overset{\rho}{\twoheadrightarrow}' \emptyset$ and $\emptyset \overset{e}{\rightarrow}' \emptyset$ by definition. Thus $\mathit{init}' \overset{\rho e}{\twoheadrightarrow}' \emptyset$ by definition of a transition.

Case $\rho \in \mathsf{weaktraces}(\mathcal{L})$. By definition there is a state $s \in S$ such that $\mathit{init} \overset{\rho}{\rightsquigarrow} s$. From Lemma 2.23 and 2.21 there is a unique state $U \in S'$ where for all $s \in S$ such that $\mathit{init} \overset{\rho}{\rightsquigarrow} s$ it holds that $s \in \bar{U}$ and $\mathit{init}' \overset{\rho}{\twoheadrightarrow}' U$. From the observation that $\rho e \notin \mathsf{weaktraces}(\mathcal{L})$ and $\rho \in \mathsf{weaktraces}(\mathcal{L})$ and the definition of a weak transition follows that for all $s \in S$ where $\mathit{init} \overset{\rho}{\rightsquigarrow} s$ there can not be a state $t \in S$ such that $s \overset{e}{\rightsquigarrow} t$. As such $U \overset{e}{\rightarrow}' \emptyset$ by the definition of normalisation.

$\Longleftarrow$). Take an arbitrary trace $\rho e \in \Sigma^*$ such that $init' \stackrel{\rho e}{\Rightarrow}' \emptyset$. By definition there is a state $U \in S'$ such that $init' \stackrel{\rho}{\Rightarrow}' U$ and $U \stackrel{e}{\rightarrow}' \emptyset$. From the existence of $U \stackrel{e}{\rightarrow}' \emptyset$ follows that $\neg\mathsf{div}(\bar{U})$ by definition of the normalisation. From Lemma 2.24 follows that $\rho \notin \mathsf{divergences}(\mathcal{L})$. From Lemma 2.21 and 2.23 follows that for every state $s \in S$ such that $init \stackrel{\rho}{\rightsquigarrow} s$ that $s \in \bar{U}$. From the definition of normalisation follows that for every $s \in \bar{U}$ there is no state $t \in S$ such that $s \stackrel{e}{\rightsquigarrow} t$. Therefore $init \stackrel{\rho e}{\rightsquigarrow} t$ does not exists by definition of a weak transition and thus $\rho e \notin \mathsf{weaktraces}(\mathcal{L})$. □

For a normalized LTS resulting from $\mathsf{norm}_{\mathsf{sfr}}$ we remark that Lemma 2.23 holds for any sequence and Lemma 2.25 holds for any weak trace $\rho \notin \mathsf{weaktraces}(\mathcal{L})$. We use 2.23' and 2.25' when applying these lemmas to an LTS resulting from $\mathsf{norm}_{\mathsf{sfr}}$.

3 Refinement Checking

First, we present the decision procedure that the antichain-based algorithms use to decide whether a refinement exists between an implementation LTS and a specification LTS.

The number of failures and divergences of a given LTS can be infinite and as such it is not viable to compute them directly. However, in [13] an algorithm is presented to decide the existence of a refinement relation. The first step of the algorithm is to mark all diverging states in both LTSs. Then, the algorithm explores the state pairs of the synchronised product between the implementation and the normalisation of the specification. For each of the explored pairs the algorithm locally decides whether the refinement relation is violated. The pair for which the refinement is violated is referred to as a *witness*. This witness is different for stable failures and failures-divergences refinement. For stable failures the state space of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{sfr}}(\mathcal{L}_2)$ is explored for the following so-called *SF-witness*:

Definition 3.1. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be LTSs. A state (s, U) of the product $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{sfr}}(\mathcal{L}_2)$ is called an *SF-witness* if and only if at least one of the following conditions hold:

- $U = \emptyset$.
- $\mathsf{stable}(s)$ and $\mathsf{refusals}(s) \subsetneq \mathsf{refusals}(\bar{U})$.

For failures-divergences refinement the state space of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ is explored for an *FDR-witness*, which has additional conditions regarding divergences.

Definition 3.2. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be LTSs. A state (s, U) of the product $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ is called an *FDR-witness* if and only if $\neg\mathsf{div}(\bar{U})$ and at least one of the following conditions hold:

- $U = \emptyset$.
- $\mathsf{stable}(s)$ and $\mathsf{refusals}(s) \subsetneq \mathsf{refusals}(\bar{U})$.
- $\mathsf{div}(s)$.

The following theorem is the reason for the ability to locally decide the existence of a refinement relation by searching for such an FDR-witness.

Theorem 3.3. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be LTSs. Then $\mathcal{L}_1 \sqsupseteq_{\mathsf{fdr}} \mathcal{L}_2$ if and only if there is no FDR-witness reachable in $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$.

Proof. Let $\mathcal{L}_1 = (S_1, init_1, Act_1, \rightarrow_1)$ and $\mathcal{L}_2 = (S_2, init_2, Act_2, \rightarrow_2)$ be LTSs and let $\mathcal{L} = (S, init, Act, \rightarrow)$ be equal to $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$.

$\Longrightarrow$). As $\mathcal{L}_1 \sqsupseteq_{\mathsf{fdr}} \mathcal{L}_2$ holds, it holds that $\mathsf{failures}_\perp(\mathcal{L}_1) \subseteq \mathsf{failures}_\perp(\mathcal{L}_2)$ and $\mathsf{divergences}(\mathcal{L}_1) \subseteq \mathsf{divergences}(\mathcal{L}_2)$. Towards a contradiction, assume there is a pair (s, U) in $\mathsf{reachable}(\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2))$ that is an FDR-witness. By definition of reachable we can pick a weak trace $\rho \in \mathsf{weaktraces}(\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2))$ such that $init \stackrel{\rho}{\rightsquigarrow} (s, U)$. From the assumption that (s, U) is an FDR-witness follows that $\neg\mathsf{div}(\bar{U})$ and so $\rho \notin \mathsf{divergences}(\mathcal{L}_2)$ by Lemma 2.24. By Lemmas 2.20 and 2.22 it holds that $init_1 \stackrel{\rho}{\rightsquigarrow}_1 s$ and for all $u \in \bar{U}$ it holds that $init_2 \stackrel{\rho}{\rightsquigarrow}_2 u$. From the assumption that (s, U) is an FDR-witness follows that $\neg\mathsf{div}(\bar{U})$ and from Lemma 2.24 follows that $\rho \notin \mathsf{divergences}(\mathcal{L}_2)$. For (s, U) to be an FDR witness there are three cases:

Case $U = \emptyset \wedge \neg\mathsf{div}(s)$. From $\neg\mathsf{div}(s)$ follows that there is a state $t \in S_1$ such that $init_1 \overset{\rho}{\rightsquigarrow} t$ and $\mathsf{stable}(t)$. So, there is a failure $(\rho, X) \in \mathsf{failures}_\perp(\mathcal{L}_1)$, for some $X \subseteq \Sigma$. By Lemma 2.25 it holds that the weak trace ρ ending in the empty set is not a weak trace of $\mathcal{L}_2$. Together with $\rho \notin \mathsf{divergences}(\mathcal{L}_2)$ follows, for all possible refusal sets $X \subseteq \Sigma$, that $(\rho, X) \notin \mathsf{failures}_\perp(\mathcal{L}_2)$ which leads to a contradiction with the assumption that $\mathsf{failures}_\perp(\mathcal{L}_1) \subseteq \mathsf{failures}_\perp(\mathcal{L}_2)$.

Case $U \neq \emptyset \wedge \mathsf{stable}(s) \wedge \mathsf{refusals}(s) \subsetneq \mathsf{refusals}(\bar{U})$. Pick a failure $(\rho, X) \in \mathsf{failures}_\perp(\mathcal{L}_1)$ where s can stably refuse $X \in \mathsf{refusals}(s)$, but $X \notin \mathsf{refusals}(\bar{U})$. By Lemma 2.21 there is no state U' of the normalised LTS such that $U \neq U'$ that can be reached by following weak trace ρ. Together with $\rho \notin \mathsf{divergences}(\mathcal{L}_2)$ follows that $(\rho, X) \notin \mathsf{failures}_\perp(\mathcal{L}_2)$, which leads to a contradiction with the assumption that $\mathsf{failures}_\perp(\mathcal{L}_1) \subseteq \mathsf{failures}_\perp(\mathcal{L}_2)$.

Case $\mathsf{div}(s)$. We know that $\rho \in \mathsf{divergences}(\mathcal{L}_1)$ by definition of divergences. However, by $\rho \notin \mathsf{divergences}(\mathcal{L}_2)$ this leads to a contradiction with the assumption that $\mathsf{divergences}(\mathcal{L}_1) \subseteq \mathsf{divergences}(\mathcal{L}_2)$.

$\Longleftarrow$). Assume that no FDR-witness is reachable in $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$. Again, we prove this by contradiction. Assume that $\mathcal{L}_1 \not\sqsupseteq_{\mathsf{fdr}} \mathcal{L}_2$. By definition of failures-divergences refinement this means that $\mathsf{failures}_\perp(\mathcal{L}_1) \subsetneq \mathsf{failures}_\perp(\mathcal{L}_2)$ or $\mathsf{divergences}(\mathcal{L}_1) \subsetneq \mathsf{divergences}(\mathcal{L}_2)$. Now, there are two cases to consider.

- Case $\mathsf{failures}_\perp(\mathcal{L}_1) \subsetneq \mathsf{failures}_\perp(\mathcal{L}_2)$ and $\mathsf{divergences}(\mathcal{L}_1) \subseteq \mathsf{divergences}(\mathcal{L}_2)$. Pick any $(\rho, X) \in \mathsf{failures}_\perp(\mathcal{L}_1)$ such that $\rho \notin \mathsf{failures}_\perp(\mathcal{L}_2)$. Observe that $\rho \notin \mathsf{divergences}(\mathcal{L}_1) \cup \mathsf{divergences}(\mathcal{L}_2)$ as otherwise no such failure (ρ, X) exists by the flooding of $\mathsf{failures}_\perp(\mathcal{L}_2)$. There is a stable state $s \in S_1$ such that $init_1 \overset{\rho}{\rightsquigarrow} s$ and $X \in \mathsf{refusals}(s)$ by definition of a failure. There are two cases for the weak trace ρ to distinguish.

  Case $\rho \notin \mathsf{weaktraces}(\mathcal{L}_2)$. By Lemma 2.25 this means that ρ is a trace leading to the empty set in $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$. By Lemma 2.20 there is a pair $(s, \emptyset)$ such that $init \overset{\rho}{\rightsquigarrow} (s, \emptyset)$ and that pair is a reachable FDR-witness by definition thus leading to the contradiction that there is no reachable FDR-witness.

  Case $\rho \in \mathsf{weaktraces}(\mathcal{L}_2)$. Failure $(\rho, X) \notin \mathsf{failures}_\perp(\mathcal{L}_2)$ by assumption. By Lemmas 2.21 and 2.23 there is a unique state V of $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ reachable via weak trace ρ such that for all $t \in S_2$ where $init_2 \overset{\rho}{\rightsquigarrow}_2 t$ holds that $t \in \bar{V}$. Thus $X \notin \mathsf{refusals}(t)$ by definition of a failure and $X \notin \mathsf{refusals}(\bar{V})$ by definition of union. Therefore $\mathsf{refusals}(s) \subsetneq \mathsf{refusals}(\bar{V})$. By Lemma 2.20 the pair (s, V) is reachable and it is an FDR-witness by definition thus leading to a contradiction with the assumption that there is no reachable FDR-witness.

- Case $\mathsf{divergences}(\mathcal{L}_1) \subsetneq \mathsf{divergences}(\mathcal{L}_2)$. Pick a diverging weak trace $\rho \in \mathsf{divergences}(\mathcal{L}_1)$ such that $\rho \notin \mathsf{divergences}(\mathcal{L}_2)$. In this case there is a prefix of ρ, which we call σ, that leads to a diverging state $init \overset{\sigma}{\rightsquigarrow} s$ by definition of divergences. However, by the assumption that $\rho \notin \mathsf{divergences}(\mathcal{L}_2)$ we know that all states $t \in S_2$ reached by following σ are not diverging. Again, by Lemmas 2.21 and 2.23 the weak trace σ in $\mathcal{L}_2$ results in a unique state U of $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$. Therefore state pair (s, U) is an FDR-witness, because $\mathsf{div}(s)$ but $\neg\mathsf{div}(\bar{U})$. Thus leading to the contradiction that there is no reachable FDR-witness.

□

For stable failures refinement we state a similar theorem that relates the SF-witness and stable failures refinement.

Theorem 3.4. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be LTSs. Then $\mathcal{L}_1 \sqsupseteq_{\mathsf{fdr}} \mathcal{L}_2$ if and only if there is no SF-witness reachable in $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{sfr}}(\mathcal{L}_2)$.

Proof. Let $\mathcal{L}_1 = (S_1, init_1, Act, \rightarrow_1)$ and $\mathcal{L}_2 = (S_2, init_2, Act_2, \rightarrow_2)$ be LTSs and let $\mathcal{L} = (S, init, Act, \rightarrow)$ be equal to $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{sfr}}(\mathcal{L}_2)$.

$\Longrightarrow$) We know that $\mathcal{L}_1 \sqsupseteq_{\mathsf{sfr}} \mathcal{L}_2$ holds. Thus $\mathsf{failures}(\mathcal{L}_1) \subseteq \mathsf{failures}(\mathcal{L}_2)$ and $\mathsf{weaktraces}(\mathcal{L}_1) \subseteq \mathsf{weaktraces}(\mathcal{L}_2)$. We assume there is a pair (s, U) in $\mathsf{reachable}(\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{sfr}}(\mathcal{L}_2))$ that is an SF-witness and show that this leads to a contradiction. As the pair (s, U) is reachable there is a weak trace $\rho \in \mathsf{weaktraces}(\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{sfr}}(\mathcal{L}_2))$ such that $init \overset{\rho}{\rightsquigarrow} (s, U)$. By Lemmas 2.20 and 2.22 it holds that $init_1 \overset{\rho}{\rightsquigarrow}_1 s$ and for all $u \in \bar{U}$ it holds that $init_2 \overset{\rho}{\rightsquigarrow}_2 u$. For (s, U) to be an SF-witness there are two cases:

Case $U = \emptyset$. By Lemma 2.25' it holds that the weak trace ρ ending in the empty set is not a weak trace of $\mathcal{L}_2$. Therefore, $\rho \notin \mathsf{weaktraces}(\mathcal{L}_2)$, which contradicts with the assumption that $\mathsf{weaktraces}(\mathcal{L}_1) \subseteq \mathsf{weaktraces}(\mathcal{L}_2)$.

Case $U \neq \emptyset \wedge \mathsf{stable}(s) \wedge \mathsf{refusals}(s) \subsetneq \mathsf{refusals}(\bar{U})$. Pick a failure $(\rho, X) \in \mathsf{failures}(\mathcal{L}_1)$ where s can stably refuse $X \in \mathsf{refusals}(s)$, but $X \notin \mathsf{refusals}(\bar{U})$. By Lemma 2.21 there is no other state U' of the normalised LTS such that $U \neq U'$ that can be reached by following weak trace ρ. So, $(\rho, X) \notin \mathsf{failures}(\mathcal{L}_2)$, which leads to a contradiction with the observation that $\mathsf{failures}(\mathcal{L}_1) \subseteq \mathsf{failures}(\mathcal{L}_2)$.

$\Longleftarrow$) No SF-witness is reachable in $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{sfr}}(\mathcal{L}_2)$. Again, we prove this by contradiction. Assume that $\mathcal{L}_1 \not\sqsupseteq_{\mathsf{sfr}} \mathcal{L}_2$. By definition of the stable failures refinement this means that $\mathsf{failures}(\mathcal{L}_1) \subsetneq \mathsf{failures}(\mathcal{L}_2)$ or $\mathsf{weaktraces}(\mathcal{L}_1) \subsetneq \mathsf{weaktraces}(\mathcal{L}_2)$. Now, there are two cases to consider.

- Case $\mathsf{weaktraces}(\mathcal{L}_1) \subsetneq \mathsf{weaktraces}(\mathcal{L}_2)$. Pick a weak trace $\rho \in \mathsf{weaktraces}(\mathcal{L}_1)$ such that $\rho \notin \mathsf{weaktraces}(\mathcal{L}_2)$. So, there is a state $s \in S_1$ such that $init_1 \overset{\rho}{\rightsquigarrow}_1 s$. By Lemma 2.25' it holds that ρ leads to the empty set in $\mathsf{norm}_{\mathsf{sfr}}(\mathcal{L}_2)$. By Lemma 2.20 the pair $(s, \emptyset)$ is a reachable SF-witness by definition. This leads to a contradiction that there is no reachable SF-witness.

- Case $\mathsf{failures}(\mathcal{L}_1) \subsetneq \mathsf{failures}(\mathcal{L}_2)$. Pick a failure $(\rho, X) \in \mathsf{failures}(\mathcal{L}_1)$ such that $(\rho, X) \notin \mathsf{failures}(\mathcal{L}_2)$. There is a stable state such that $init_1 \overset{\rho}{\rightsquigarrow}_1 s$ and $X \in \mathsf{refusals}(s)$ by definition of a failure.

  Case $\rho \notin \mathsf{weaktraces}(\mathcal{L}_2)$. Then $\mathsf{weaktraces}(\mathcal{L}_1) \subsetneq \mathsf{weaktraces}(\mathcal{L}_2)$ by definition of failure, which leads to the same contradiction as in the $\mathsf{weaktraces}(\mathcal{L}_1) \subsetneq \mathsf{weaktraces}(\mathcal{L}_2)$ case.

  Case $\rho \in \mathsf{weaktraces}(\mathcal{L}_2)$. Failure $(\rho, X) \notin \mathsf{failures}(\mathcal{L}_2)$ by assumption. By Lemmas 2.21 and 2.23' weak trace ρ leads to a unique state U in $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ such that for all $t \in S_2$ where $init_2 \overset{\rho}{\rightsquigarrow}_2 t$ it holds that $t \in \bar{U}$. For all $t \in \bar{U}$ it holds that $X \notin \mathsf{refusals}(t)$ by definition of a failure, and as such $X \notin \mathsf{refusals}(\bar{U})$ by definition of union. Therefore, $\mathsf{refusals}(s) \subsetneq \mathsf{refusals}(\bar{U})$. By Lemma 2.20 the pair (s, U) is reachable and it is an SF-witness by definition. Thus leading to a contradiction with the assumption that there is no reachable FDR-witness.

□

4 Antichain Algorithms for Refinement Checking

The previously explained decision procedure requires the whole state space of the synchronous product to be explored when there is no witness. In that case, the subset construction of the specification LTS dominates the theoretical worst-case run time of refinement checking. As observed in [22] this can be improved by exploiting an *antichain* approach to potentially reduce the state space of the normalised specification. An antichain is a subset of incomparable elements from a partially ordered set.

Definition 4.1. Let X be a set and $\leq \subseteq X \times X$ a reflexive, antisymmetric and transitive relation. Then $(X, \leq)$ is a *partially ordered set*.

Definition 4.2. Let $(X, \leq)$ be a partially ordered set. A set $\mathcal{A} \subseteq X$ is an *antichain* iff for all $x, y \in \mathcal{A}$ with $x \neq y$ it holds that $x \not\leq y$ and $y \not\leq x$.

An antichain $\mathcal{A}$ over a partially ordered set $(X, \leq)$ supports two operations. The operation $\Subset$ checks whether a given element is included within $\mathcal{A}$. Let $x \in X$ be an element. We denote $x \Subset \mathcal{A}$ if and only if there is an element $y \in \mathcal{A}$ such that $y \leq x$. This can also be extended to a set of elements. Let $U \subseteq X$ be a set of elements. Then $U \Subset^{\forall} \mathcal{A}$ if and only if for all $y \in U$ it holds that $y \Subset \mathcal{A}$. The operation $\Cup$ inserts a state pair into $\mathcal{A}$ and removes all larger pairs. Formally $\mathcal{A} \Cup x$ results in the set $\{(y \mid y \in \mathcal{A} \wedge x \not\leq y\} \cup \{x\}$. Note that this operation only results in an antichain whenever $x \notin \mathcal{A}$.

As [22] suggests the state space of the synchronised product induces a partially ordered set as follows. For pairs of states (s, U) and (t, V) of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ we define an ordering such that $(s, U) \leq (t, V)$ if and only if $s = t$ and $U \subseteq V$. A fundamental property for the correctness of the antichain-based refinement checking is expressed in the following lemma. We note that this property holds due to the allowance of the empty set in the normalised LTS.

Lemma 4.3. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ be two LTSs. For all states $(s, U), (s, V)$ of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ satisfying $(s, U) \leq (s, V)$ and events $e \in \Sigma_\tau$ such that $(s, V) \xrightarrow{e} (t, V')$ there is a state (t, U') such that $(s, U) \xrightarrow{e} (t, U')$ and $(t, U') \leq (t, V')$.

From the state pair ordering we can observe that smaller state pairs lead to larger refusal sets, by observing that the refusals of a set are defined as the union of all stable states. Similarly, if a set of states of the specification does not diverge, then any subset of this set does not diverge either. The proof of this property is in Section 5.

We remark that in [22, Proposition 1] a similar lemma was stated where for states $(s, V) \leq (s, U)$ the traces can be mimicked and preserve $(t, V') \leq (t, U')$. This property is correct for $\mathsf{norm}_{\mathsf{sfr}}(\mathcal{L}_2)$, but not for $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ as $\mathsf{div}(\bar{U})$ does not necessarily imply $\mathsf{div}(\bar{V})$ whenever $U \subseteq V$. We use our reformulated Lemma 4.3 to prove the correctness of the antichain algorithm.

The idea for the antichain-based algorithm is that the normalisation of the specification and the synchronisation is computed on-the-fly. The *antichain* is then used to keep track of the already explored subset of the combined state space resulting by recording the smallest state pairs already explored. If a larger state pair is encountered during the exploration, further exploration is unnecessary, thereby pruning the state space.

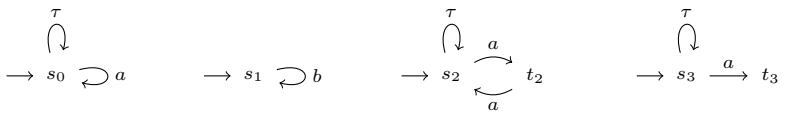
First, we present the algorithm that was introduced in [22]. The algorithm takes an implementation LTS $\mathcal{L}_1$ and a specification LTS $\mathcal{L}_2$. A *working* stack is used to keep track of state pairs that still have to be explored. When a new state pair is discovered it is not added back to *working* if it is already recorded in the *antichain*. We remark that the pseudocode for both stable failures and failures-divergences refinement checking have been combined and can be selected by means of a boolean flag CheckDiv.

Algorithm 1 The erroneous refinement checking algorithm as presented in [22]. For LTSs $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ the algorithm is claimed to return *true* iff $\mathcal{L}_1$ refines $\mathcal{L}_2$. This algorithm decides stable failures refinement if CheckDiv is false and failures-divergences refinement otherwise.

```
1:  procedure REFINES(L1, L2, CheckDiv)
2:      let working be a stack containing a pair (init_1, {s ∈ S_2 | init_2 ⇝^ε_2 s})
3:      let antichain ← ∅
4:      while working ≠ ∅ do
5:          pop (impl, spec) from working
6:          antichain ← antichain ⩊ (impl, spec)
7:          if CheckDiv ∧ div(impl) then
8:              if ¬div(spec) then
9:                  return false
10:         else
11:             if refusals(impl) ⊊ refusals(spec) then
12:                 return false
13:             for impl -e->_1 impl' do
14:                 if e = τ then
15:                     spec' ← spec
16:                 else
17:                     spec' ← {s' ∈ S_2 | ∃s ∈ spec : s ⇝^e_2 s'}
18:                 if spec' = ∅ then
19:                     return false
20:                 if (impl', spec') ∉ antichain then
21:                     push (impl', spec') into working
22:     return true
```

We note that this algorithm correctly decides stable failures refinement as presented in Definition 2.12, but it fails to correctly decide failures-divergences refinement defined in Definition 2.13. Moreover, the algorithm also fails to decide the non-standard stable failures and failures-divergences relations presented in [22, Definitions 4] and [22, Definitions 5] respectively. All three issues are illustrated by the example below.

Example 4.4. Consider the four transition systems depicted below.



First, we observe that the algorithm correctly detects that $s_0 \sqsupseteq_{\mathsf{sfr}} s_1$ does not hold, which follows from the fact that $\mathsf{weaktraces}(s_0) \subsetneq \mathsf{weaktraces}(s_1)$. However, since s_0 is not stable, we have $\mathsf{failures}(s_0) = \emptyset$ and hence $\mathsf{failures}(s_0) \subseteq \mathsf{failures}(s_1)$. This implies that the stable failures defined in [22, Definition 4] should hold, which means that the result of Algorithm 1 is not conform the definition and as such [22, Theorem 2] is wrong. Next, observe that we have $s_1 \sqsupseteq_{\mathsf{fdr}} s_0$, since the divergence of s_0 results in chaotic behaviour of s_0, thus any system refines this system. However, Algorithm 1, returns *false* when CheckDiv holds, which follows from $\mathsf{failures}(s_1) \subsetneq \mathsf{failures}(s_0)$. Next, observe that the algorithm returns *true* when checking for $s_3 \sqsupseteq_{\mathsf{fdr}} s_2$. The reason is that for the pair $(\{s_2\}, s_3)$, it detects that state s_3 diverges and concludes that since also the normal form state of the specification $\{s_2\}$ diverges, it can terminate the iteration and returns *true*. This is a consequence of splitting the divergence tests over two **if**-statements in lines 7 and 8. According to the failures-divergences refinement of [22], however, the algorithm should return *false*, since $\mathsf{failures}(s_3) \subseteq \mathsf{failures}(s_2)$ fails to hold: we have $(a, \{a\}) \in \mathsf{failures}(s_3)$ but not $(a, \{a\}) \in \mathsf{failures}(s_2)$. □

The problem with checking failures-divergences refinement as defined in Definition 2.13 is that whenever CheckDiv is true the algorithm does *not* explore the state space of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ correctly. Therefore if CheckDiv is true the resulting witness does not necessarily satisfy $\neg\mathsf{div}(\overline{spec})$ and thus might not be a valid FDR-witness.

We note that the algorithm explores the synchronous product of the specification and implementation in a depth-first, on-the-fly manner. However, for the purpose of generating counterexamples it would be favourable to use a breadth-first search to ensure conciseness of the resulting trace leading to a witness.

Algorithm 1 can easily be changed to a breadth-first variant by using a FIFO (First in, first out) queue instead of a stack as the data structure of *working*. However, the breadth-first variant of the above algorithm suffers from an enormous performance degradation. A number of performance problems can be identified in the original algorithm, which are also present (albeit less pronounced in practice) in the depth-first exploration:

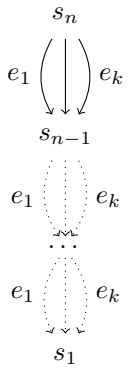
1. On line 11 the refusal check is also performed on unstable states, which results in a potential unnecessary and expensive search for stable states.

2. Adding the pair $(impl, spec)$ that is taken from *working* to *antichain* at that moment results in redundant (even duplicate) pairs being added to *working*. Namely all successors of $(impl, spec)$ might be added if they did not already occur in *antichain* before.

3. Contrary to the explicit claim in [22, Section 2.2] the variable *antichain* is not guaranteed to be an antichain.

The first problem can easily be identified as unnecessary overhead and is also wrong according to our corrected definition of refusals. However, the second and third problems are more difficult to see and we construct two examples to explain them in more detail. Example 4.5 illustrates the second problem.

Example 4.5. Let $\mathcal{L}_n^k$ be a class of labelled transitions systems $(S, \rightarrow, Act, init)$ where $S = \{s_1, \ldots, s_n\}$, $Act = \{e_1, \ldots, e_k\}$, transitions $s_i \xrightarrow{e_j} s_{i-1}$ for all $1 \leq j \leq k$, $1 < i \leq n$ and $init = s_n$. These transition systems can be depicted as follows:



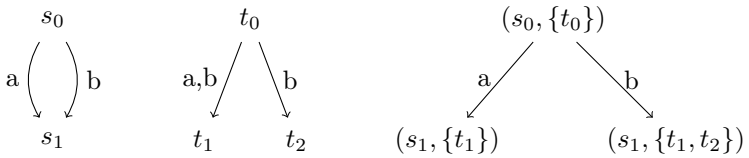
Each labelled transition system in this class has n states and $k * (n-1)$ transitions. The refinement relation that is checked is $\mathcal{L}_n^k \sqsupseteq_{\mathsf{sfr}} \mathcal{L}_n^k$. Note that this class of LTSs is completely deterministic and they contain no τ-transitions. The state pairs that are added to *antichain* are $(s_i, \{s_i\})$ for all $1 \leq i \leq n$.

Algorithm 1 exploring depth-first will add the state reachable via a single step once for every event, because $(s_{i+1}, \{s_{i+1}\})$ is only added to *antichain* after $(s_i, \{s_i\})$ has finished exploring its outgoing transitions. This occurs in every state, because the state reached via such a transition was not visited before, resulting in a maximum *working* stack size of $\mathcal{O}(n*k)$ entries. Now every state is part of *antichain*, so every state taken from *working* will visit all of its successors and conclude that they are already part of the *antichain*. This results in a total of $\mathcal{O}(n * k^2)$ *antichain* inclusion checks.

The breadth-first variant of Algorithm 1 requires even more *antichain* inclusion tests. Similarly to the depth-first variant it also adds the state reachable via a single step once for every event for the same reason. However, now $(s_{i+1}, \{s_{i+1}\})$ is only added to the *antichain* after all k copies of $(s_i, \{s_i\})$ are taken from the *working*, which is now a FIFO queue. Therefore each element in *working* adds k elements before it is added to *antichain*, resulting in a maximum queue size of $\mathcal{O}(k^n)$ at state $(s_1, \{s_1\})$. This results in $\mathcal{O}(k^{n+1})$ *antichain* inclusion checks.

Next, we present an example where the *antichain* is not guaranteed to remain an antichain as an illustration of the third problem.

Example 4.6. Consider the two left-most labelled transition systems depicted below, along with the (normalisation) of the synchronous product on the right.



Algorithm 1 starts with *working* containing $(s_0, \{t_0\})$ and an empty *antichain*. When exploring $(s_0, \{t_0\})$ the reached states $(s_1, \{t_1\})$ and $(s_1, \{t_1, t_2\})$ are not part of the *antichain* $\{(s_0, \{t_0\})\}$. So these pairs will be added to the *working* queue. The pair $(s_1, \{t_1\})$ from the *working* stack is the next to be considered and it will be added to the *antichain* resulting in $\{(s_0, \{t_0\}), (s_1, \{t_1\})\}$. In the next iteration the pair $(s_1, \{t_1, t_2\})$ is popped from *working* and will be added to *antichain*, resulting in the set $\{(s_0, \{t_0\}), (s_1, \{t_1\}), (s_1, \{t_1, t_2\})\}$. Clearly, the resulting *antichain* is no longer an antichain, as $(s_1, \{t_1\}) \leq (s_1, \{t_1, t_2\})$.

5 A Correct and Improved Antichain Algorithm

Our corrected algorithm addresses the identified issues with a number of changes. The first performance problem can be solved by only performing the refusal check whenever the implementation state is stable and by computing the refusals according to Definition 2.8.

Solving the second performance problem is more involved. The essential observation here is that in order for the information in *antichain* to be most effective, states of the synchronous product must

be added to *antichain* as soon as these are discovered, even if these have not yet been fully explored. Effectively we maintain the invariant that *working* $\Subset^{\forall}$ *antichain*. This can be achieved by adding the initial state pair and all explored pairs to *antichain* upon discovery. In turn, this also resolves the third issue, as only state pairs that are not included in *antichain* get added.

Finally, the on-the-fly normalisation when searching for an FDR-witness was changed such that it only continues with the current state pair whenever *spec* does not diverge, effectively following Definition 2.17. This is achieved by exchanging the conditions of these if-statements for the divergence checks. Algorithm 2 contains the corrected algorithm for checking stable failures and failures-divergences refinement.

Algorithm 2 The corrected refinement checking algorithm. For LTSs $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ the algorithm returns *true* iff $\mathcal{L}_1$ refines $\mathcal{L}_2$. This algorithm decides stable failures refinement if CheckDiv is false and failures-divergences refinement otherwise.

```
1:  procedure REFINES_NEW(L1, L2, CheckDiv)
2:      let working be a queue containing a pair (init1, {s ∈ S2 | init2 ⇝ε_2 s})
3:      let antichain ← ∅ ⩏ (init1, {s ∈ S2 | init2 ⇝ε_2 s})
4:      while working ≠ ∅ do
5:          pop (impl, spec) from working
6:          if ¬CheckDiv ∨ ¬div(spec) then
7:              if CheckDiv ∧ div(impl) then
8:                  return false
9:              else
10:                 if stable(impl) ∧ refusals(impl) ⊊ refusals(spec) then
11:                     return false
12:                 for impl →e_1 impl' do
13:                     if e = τ then
14:                         spec' ← spec
15:                     else
16:                         spec' ← {s' ∈ S2 | ∃s ∈ spec : s ⇝e_2 s'}
17:                     if spec' = ∅ then
18:                         return false
19:                     if (impl', spec') ∉ antichain then
20:                         antichain ← antichain ⩏ (impl', spec')
21:                         push (impl', spec') into working
22:     return true
```

Consider Example 4.5 again, but now checking stable failures refinement using Algorithm 2. The depth-first variant of this algorithm only adds the successor state to the *working* stack once, because for every other outgoing transition it was already part of *antichain* when it is discovered. This results in a maximum *working* stack size of at most $\mathcal{O}(1)$ entries. For each state and each successor *antichain* inclusion is checked once, resulting in $\mathcal{O}(n * k)$ checks. This is an improvement compared to the depth-first variant of Algorithm 1 of a factor $n * k$ in the maximum *working* stack size and a factor k in the number of inclusion checks. For the breadth-first variant the bounds are exactly the same, *i.e.*, maximum $\mathcal{O}(1)$ *working* queue size and $\mathcal{O}(n * k)$ number of inclusion checks. This is an improvement compared to the breadth-first variant of Algorithm 1 of a factor k^n in the *working* queue size and a factor k^n/n in the number of *antichain* inclusion checks.

The correctness of Algorithm 2 is captured by the following theorem, which is repeated at the end of this section with an explicit proof.

Theorem 5.1. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ be two LTSs.

- REFINES$_{\text{NEW}}(\mathcal{L}_1, \mathcal{L}_2,$ false) returns true if and only if $\mathcal{L}_1 \sqsupseteq_{\mathsf{sfr}} \mathcal{L}_2$.
- REFINES$_{\text{NEW}}(\mathcal{L}_1, \mathcal{L}_2,$ true) returns true if and only if $\mathcal{L}_1 \sqsupseteq_{\mathsf{fdr}} \mathcal{L}_2$.

We focus on the proof of correctness with respect to failures-divergences refinement; for stable failures refinement the proof is virtually the same. As the corrected algorithm fundamentally differs from

Algorithm 1, we cannot reuse arguments for the proof of correctness presented in [22], which are based on invariants that do not hold in our case.

First we show termination of Algorithm 2. An important observation of *antichain* is that adding elements to it does not affect the $\Subset$-inclusion of other elements, *i.e.*, pairs that are $\Subset$-included remain included after any insertion.

Lemma 5.2. Let $\mathcal{A}$ be an antichain of a partially ordered set $(Z, \leq)$. For any elements $X, U \in Z$ such that $U \Subset \mathcal{A}$ and $X \notin \mathcal{A}$ the statement $U \Subset (\mathcal{A} \Cup X)$ holds.

Proof. Assume arbitrary elements $U, X \in Z$ such that $U \Subset \mathcal{A}$ and $X \notin \mathcal{A}$. Recall that the definition of $\mathcal{A} \Cup X$ results in an antichain $\{V \mid V \in \mathcal{A} \wedge X \not\leq V\} \cup \{X\}$, because $X \notin \mathcal{A}$ by assumption.

Case $X \leq U$. Then $U \Subset (\mathcal{A} \Cup X)$ follows from the fact that $X \in \mathcal{A} \Cup X$.

Case $X \not\leq U$. There is an element $V \in \mathcal{A}$ such that $V \leq U$ by assumption that $U \Subset \mathcal{A}$. Because $X \not\leq U$ and $V \leq U$ we also know that $X \not\leq V$. Consequently, $V \in (\mathcal{A} \Cup X)$ and thus also $U \Subset (\mathcal{A} \Cup X)$. □

Next, we show that all processed pairs do not get added back to the *working* stack. We define a ghost variable *Done* that contains the set of state pairs that have been processed, *i.e.*, a pair was taken from *working* and the outer while loop has finished one iteration. In Appendix A the exact introduction of variable *Done* into Algorithm 2 is shown. Now, we first prove that all states in *Done* and *working* are reachable.

Lemma 5.3. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ be two LTSs and CheckDiv is true. The invariant $Done \cup working \subseteq \mathsf{reachable}(\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2))$ holds in the outer while loop (lines 4-21) of Algorithm 2.

Proof. Initially, the pair $(init_1, \{s \in S_2 \mid init_2 \overset{\epsilon}{\rightsquigarrow} s\})$ is reachable by the empty trace because this pair is the initial state of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ by definition. So *working* only consisting of this pair is reachable as well, and *Done* is empty.

Maintenance. For every state $(impl, spec) \in Done \cup working$ there is a trace $\sigma \in Act^*$, such that $init \overset{\sigma}{\twoheadrightarrow} (impl, spec)$ by the definition of reachable. At line 12 the outgoing transition $(impl, e, impl')$ is an element of $\rightarrow_1$. Line 13 corresponds exactly to the first case of the product definition (Def. 2.15). Similarly, line 16 corresponds exactly to the second case of the product definition where $(spec, e, spec')$ is a transition in $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_1)$ because $\neg\mathsf{div}(\overline{spec})$ holds. As such, there is a transition $(impl, spec) \xrightarrow{e} \rightarrow (impl', spec')$ in the product LTS. By definition of a trace and the definition of reachable this means that $working \cup \{(impl', spec')\} \subseteq \mathsf{reachable}(\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2))$. From the observation that $(impl, spec)$ was reachable we can conclude that $Done \cup \{(impl, spec)\}$ is a subset of reachable as well. □

Next, we show that every pair gets visited at most once and then is added to *Done* by a number of observations. We show that *working* contains no duplicates by showing that all pairs in *working* are already part of the *antichain* and thus do not get added again. For the purpose of identifying elements in the stack we define, for a given index i, the notation $working^i$ to represent the ith pair on the stack, or queue respectively. Furthermore we formalize that pairs in *Done* do not get added back to *working* from the observation that they already occur in *antichain* and that therefore *Done* and *working* are disjoint.

Lemma 5.4. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ be two LTSs and CheckDiv is true. The following invariant holds in the outer while loop (lines 4-21) of Algorithm 2.

$$Done \cup working \Subset^{\forall} antichain \wedge \forall i \neq j : working^i \neq working^j \wedge Done \cap working = \emptyset \tag{I}$$

Proof. Initially, the initial pair is both added to *working* and *antichain*, and *Done* is empty.

Maintenance. Let $Done'$ be equal to $Done \cup \{(impl, spec)\}$. By the assumption that Invariant I holds we know at line 5 that $(impl, spec) \Subset antichain$ from the assumption that $working \Subset^{\forall} antichain$. This ensures that $Done' \Subset^{\forall} antichain$ and $(working \setminus \{(impl, spec)\}) \Subset^{\forall} antichain$ holds by definition. Furthermore we know that $(impl, spec) \notin (working \setminus \{(impl, spec)\})$ from $\forall i \neq j : working^i \neq working^j$. From this it follows that $Done' \cap (working \setminus \{(impl, spec)\}) = \emptyset$.

At line 19, from $(impl', spec') \notin antichain$, we know that there is no pair $(s, U) \in antichain$ such that $(s, U) \leq (impl', spec')$. From the observation that $Done' \cup working \Subset^{\forall} antichain$ it holds that if

$(impl', spec') \in Done' \cup working$ then there is a pair $(t, V) \in antichain$ such that $(t, V) \leq (impl', spec')$. However, that leads to a contradiction with the observation that there is no pair $(s, U) \in antichain$ such that $(s, U) \leq (impl', spec')$ thus $(impl', spec') \notin Done' \cup working$. Let $working'$ be equal to $\{(impl', spec')\} \cup working \setminus \{(impl, spec)\})$. From the fact that $(impl', spec') \notin working$ follows that $\forall i \neq j : working'^i \neq working'^j$ holds. At line 20 the $(impl', spec')$ pair is added to *antichain* and Lemma 5.2 ensures that $Done' \cup working' \Subset^\forall antichain$ holds. From the observation that $(impl', spec') \notin Done' \cup working$ and $Done' \cap (working \setminus \{(impl, spec)\}) = \emptyset$ we can also conclude that $Done' \cap working' = \emptyset$ at line 21. □

Observe that Lemma 5.3 implies that all states in *working* are reachable states from $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$. Using Lemma 5.4 we can now prove termination by showing that all pairs will only be processed once and as such Algorithm 2 terminates for finite state, finitely branching labelled transition systems. This is formalized in the following theorem.

Theorem 5.5. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ be two LTSs and CheckDiv is true. If the sets of states S_1 and S_2 are finite then Algorithm 2 terminates.

Proof. The inner for-loop is bounded as the number of outgoing transitions $\rightarrow_1$ is finite. The total number of state pairs in $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ is finite since S_1 and S_2 are finite. From Lemma 5.3 it follows that *Done* is a subset of the reachable state pairs. As $Done \cap working = \emptyset$ by Lemma 5.4 we conclude that *Done* strictly increases with every iteration. So, only a finite number of iterations of the outer for-loop are possible. □

These observations also hold for stable failures refinement checking. The only difference is that in Lemma 5.3 we can observe that $(spec, e, spec')$ is a transition in the LTS resulting from $\mathsf{norm}_{\mathsf{sfr}}(\mathcal{L}_2)$. Note that although we do not discuss the theoretical worst-case complexity of the improved algorithm these observations already give an upper bound on the number of states that can be explored. Especially the absence of duplicates in *working* and the maximisation of *antichain* following from $Done \cup working \Subset^\forall antichain$ do not hold for Algorithm 1.

Now we prove the correctness of the already presented Lemma 4.3 and generalise it to traces.

Lemma 4.3. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ be two LTSs. For all states $(s, U), (s, V)$ of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ satisfying $(s, U) \leq (s, V)$ and events $e \in \Sigma_\tau$ such that $(s, V) \xrightarrow{e} (t, V')$ there is a state (t, U') such that $(s, U) \xrightarrow{e} (t, U')$ and $(t, U') \leq (t, V')$.

Proof. Let $\mathcal{L} = (S, init, Act, \rightarrow)$ be equal to $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ and let $\mathcal{L}' = (S_2', init_2', Act_2', \rightarrow_2')$ be equal to $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$. Take any two state pairs such that $(s, U) \leq (s, V)$. Pick an arbitrary pair $(t, V') \in S$ such that $(s, V) \xrightarrow{e} (t, V')$. Now there are two cases to distinguish.

Case $e = \tau$. By definition of the product a transition $s \xrightarrow{\tau}_1 t$ exists and $V = V'$. So there is also a transition $(s, U) \xrightarrow{\tau} (t, U')$ with $U = U'$. By the assumption that $(s, U) \leq (s, V)$ we know that $(t, U') \leq (t, V')$.

Case $e \neq \tau$. By definition of the product there are transitions $s \xrightarrow{e}_1 t$ and $V \xrightarrow{e}_2' V'$. The normalisation has transitions $V \xrightarrow{e}_2' V'$ if and only if $V' = \{v \in S_2 \mid \exists w \in \bar{V} : w \overset{e}{\rightsquigarrow}_2 v\}$ and $\neg\mathsf{div}(\bar{V})$ by definition. Let U' be equal to $\{v \in S_2 \mid \exists w \in \bar{U} : w \overset{e}{\rightsquigarrow}_2 v\}$. From monotonicity it follows that $\bar{U} \subseteq \bar{V}$ implies that $\overline{U'} \subseteq \overline{V'}$. Furthermore, from $\neg\mathsf{div}(\bar{V})$ follows that $\neg\mathsf{div}(\bar{U})$ by definition. Therefore, $U \xrightarrow{e}_2' U'$ by definition of normalisation and $(s, U) \xrightarrow{e} (t, U')$ is a valid transition in the product with $(t, U') \leq (t, V')$. □

Lemma 5.6. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ be two LTSs. For all states $(s, U), (s, V)$ of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ satisfying $(s, U) \leq (s, V)$ and for every sequence $\sigma \in Act^*$ such that $(s, V) \xrightarrow{\sigma} (t, V')$ then $(s, U) \xrightarrow{\sigma} (t, U')$ and $(t, U') \leq (t, V')$.

Proof. Let $\mathcal{L} = (S, init, Act, \rightarrow)$ be equal to $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$. The proof uses induction on the length of all sequences in Act^*. The induction hypothesis is that for all states $(s, V), (t, V') \in S$ and sequences $\rho \in Act^*$ of length k such that $(s, V) \xrightarrow{\rho} (t, V')$ then for any state $(s, U) \in S$ satisfying $(s, U) \leq (s, V)$ there is a state $(t, U') \in S$ such that $(s, U) \xrightarrow{\rho} (t, U')$ and $(t, U') \leq (t, V')$.

Base case, the empty trace ϵ of length zero. Take two pairs $(s,U),(s,V) \in S$ satisfying $(s,U) \leq (s,V)$. The empty trace can only reach $(s,U) \xRightarrow{\epsilon} (s,U)$, similar for (s,V), and as such $(s,U) \leq (s,V)$ follows by assumption.

Inductive step. Suppose that the induction hypothesis holds for all traces $\sigma \in Act^*$ of length i. Take an arbitrary state (t,V') and event $e \in Act$ such that $(r,V'') \xRightarrow{\sigma e} (t,V')$. By definition of a trace there is a state (s,V) such that $(r,V'') \xRightarrow{\sigma} (s,V)$ and $(s,V) \xrightarrow{e} (t,V')$. Take an arbitrary state $(r,U'') \leq (r,V'')$. By Lemma 4.3 and the existence of $(s,V) \xrightarrow{e} (t,V')$ for any state $(s,U) \leq (s,V)$ there is a $(s,U) \xrightarrow{e} (t,U')$ such that $(t,U') \leq (t,V')$. From the induction hypothesis follows that $(r,U'') \xRightarrow{\sigma} (s,U)$ for some (s,U) such that $(s,U) \leq (s,V)$. Therefore, by definition of a trace $(r,U'') \xRightarrow{\sigma e} (t,U')$, and $(t,U') \leq (t,V')$ was already established. □

Next, we formalise the property that if a state pair is an FDR-witness then any smaller state pair is also an FDR-witness. This can be combined with the previous lemma to obtain the property that if a state pair can reach an FDR-witness then a smaller state pair can also reach an FDR-witness.

Lemma 5.7. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1,2\}$ be two LTSs. For all states $(s,U),(s,V)$ of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ satisfying $(s,U) \leq (s,V)$ it holds that if (s,V) is an FDR-witness then (s,U) is an FDR-witness.

Proof. Take arbitrary state pairs $(s,U),(s,V)$ of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ satisfying $(s,U) \leq (s,V)$. From the definition of an FDR-witness it follows that $\neg\mathsf{div}(\bar{V})$ and one of the following holds: $V = \emptyset$ or $\mathsf{stable}(s) \wedge \mathsf{refusals}(s) \not\subseteq \mathsf{refusals}(\bar{V})$ or $\mathsf{div}(s)$. From the definition of refusals we know that $\mathsf{refusals}(\bar{U}) \subseteq \mathsf{refusals}(\bar{V})$ and by definition of diverges that $\neg\mathsf{div}(\bar{V})$ implies $\neg\mathsf{div}(\bar{U})$. This means that (s,U) is an FDR-witness, because $\neg\mathsf{div}(\bar{U})$ and if $V = \emptyset$ then $U = \emptyset$ or if $\mathsf{stable}(s) \wedge \mathsf{refusals}(s) \not\subseteq \mathsf{refusals}(\bar{V})$ then $\mathsf{stable}(s) \wedge \mathsf{refusals}(s) \not\subseteq \mathsf{refusals}(\bar{U})$ or $\neg\mathsf{div}(s)$. □

Lemma 5.8. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1,2\}$ be two LTSs. For all states $(s,U),(s,V)$ of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ where $(s,U) \leq (s,V)$ and for every sequence $\sigma \in \Sigma_\tau^*$ it holds that if (s,V) can reach an FDR-witness with σ then (s,U) can reach an FDR-witness with σ as well.

Proof. Let $\mathcal{L} = (S, init, Act, \rightarrow)$ be equal to $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$. Take arbitrary states $(s,U),(s,V) \in S$ satisfying $(s,U) \leq (s,V)$. By the definition of reachable pick (t,V') to be an FDR-witness and $\sigma \in \Sigma_\tau^*$ a trace such that $(s,V) \xRightarrow{\sigma} (t,V')$. By Lemma 5.6 there is a pair $(t,U') \leq (t,V')$ such that $(s,U) \xRightarrow{\sigma} (t,U')$. From lemma 5.7 and the assumption that (t,V') is an FDR-witness it follows that state (t,U') must be an FDR-witness as well. Therefore, (t,U') is a reachable FDR-witness with the trace σ starting in (s,U). □

Next, we define a predicate for the existence of an FDR-witness and a function that computes the distance to the closest FDR-witness.

Definition 5.9. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1,2\}$ be two LTSs. For a set of states U of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$, let $\mathsf{FDR}(U)$ be the predicate that is true if and only if there is an FDR-witness in U.

Definition 5.10. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1,2\}$ be two LTSs and let $\mathcal{L} = (S, init, Act, \rightarrow)$ be equal to $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$. For a state $s \in S$, let $\mathsf{Dist}(s) \in \mathbb{N} \cup \{\infty\}$ denote the *shortest* distance from state s to an FDR-witness or ∞ when none are reachable, formally $\mathsf{Dist}(s) = \min\{|\sigma| \mid \sigma \in \mathsf{traces}(\mathcal{L}) \wedge (t,V) \in S \wedge init \xRightarrow{\sigma} (t,V) \wedge \mathsf{FDR}(\{(t,V)\})\}$, where $\min\{\emptyset\}$ is equal to ∞. For a set of states $U \subseteq S$, let $\mathsf{Dist}(U)$ denote the shortest distance among all states in U, formally $\mathsf{Dist}(U) = min\{\mathsf{Dist}(s) \mid s \in U\}$.

The following lemma relates the reachability of an FDR-witness for smaller state pairs to the minimal distance.

Lemma 5.11. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1,2\}$ be two LTSs. For all states $(s,U),(s,V)$ of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ satisfying $(s,U) \leq (s,V)$ it holds that $\mathsf{Dist}((s,U)) \leq \mathsf{Dist}((s,V))$.

Proof. Let $\mathcal{L} = (S, init, Act, \rightarrow)$ be equal to $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$. Take arbitrary states $(s,U),(s,V) \in S$ satisfying $(s,U) \leq (s,V)$. From Lemma 5.8 it follows that if (s,V) can reach an FDR-witness by the shortest trace σ then (s,U) can also reach an FDR-witness with trace σ, which by definition means that $\mathsf{Dist}((s,U)) \leq \mathsf{Dist}((s,V))$. □

The last lemma implies that whenever a pair is removed from the *antichain* due to an insertion, the inserted (smaller) state pair has a shorter or equal distance to its closest FDR-witness. This property can be used to show that the algorithm always gets closer to an FDR-witness during exploration and that pruning parts of the statespace does not remove all existing FDR-witnesses from the reachable states. The latter property is captured by the following lemmas.

Lemma 5.12. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ be two LTSs. For all states (s, U) of $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ it holds that if $\mathsf{div}(\bar{U})$ then $\mathsf{Dist}((s, U))$ is ∞.

Proof. Let $\mathcal{L} = (S, init, Act, \rightarrow)$ be equal to $\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ and let $\mathcal{L}' = (S'_2, init'_2, Act'_2, \rightarrow'_2)$ be equal to $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$. Take an arbitrary state $(s, U) \in S$ such that $\mathsf{div}(\bar{U})$. For any event $e \in \Sigma$ and state $V \in S'_2$ there is no transition $U \xrightarrow{e}'_2 V$ by definition of the normalization. Therefore, by definition of the product and Lemma 2.18, for any state $(t, V) \in S$ such that $(s, U) \overset{\epsilon}{\rightsquigarrow} (t, V)$ it holds that $U = V$. Thus, any reachable state (t, V) also satisfies $\mathsf{div}(\bar{V})$ and as such can not be an FDR-witness, meaning that $\mathsf{Dist}((s, U))$ is ∞. □

Lemma 5.13. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ be two LTSs and $\mathsf{CheckDiv}$ is true. If $\mathsf{FDR}(\mathsf{reachable}(\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)))$ holds then Invariant II holds for every iteration of the while loop at line 4 of Algorithm 2:

$$\mathsf{Dist}(Done) > \mathsf{Dist}(working) \wedge \mathsf{Dist}(working) = \mathsf{Dist}(antichain) \tag{II}$$

Proof. Assume that $\mathsf{FDR}(\mathsf{reachable}(\mathcal{L}_1 \times \mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)))$ holds, so there is a reachable FDR-witness.

Initialisation. *Done* is empty, so $\mathsf{Dist}(Done) = \mathsf{Dist}(\emptyset) = \infty$. For *working*, which at this point only contains the initial state, the witness is reachable and so $\mathsf{Dist}(working) < \infty$. The initial state is also added to *antichain*. Thus $\mathsf{Dist}(Done) > \mathsf{Dist}(working) \wedge \mathsf{Dist}(working) = \mathsf{Dist}(antichain)$.

Maintenance. Assume that $working \neq \emptyset$ and $\mathsf{Dist}(Done) > \mathsf{Dist}(working) \wedge \mathsf{Dist}(working) = \mathsf{Dist}(antichain)$. At line 5 a pair $(impl, spec)$ is taken from *working*, so *working* becomes equal to $working \setminus (impl, spec)$. Let $Done'$ be equal to $Done \cup \{(impl, spec)\}$ and let N be equal to $\mathsf{Dist}((impl, spec))$. There are three cases to distinguish.

Case $N > \mathsf{Dist}(working \cup \{(impl, spec)\})$. Removing $(impl, spec)$ from *working* did not change its distance, so $\mathsf{Dist}(working) = \mathsf{Dist}(working \cup \{(impl, spec)\})$. Adding this pair to *Done*, because $N > \mathsf{Dist}(working)$, results in $\mathsf{Dist}(working) < \mathsf{Dist}(Done') \leq \mathsf{Dist}(Done)$. Consider the outgoing transitions at line 12. These $(impl', spec')$ pairs must have a distance of at least $\mathsf{Dist}(working)$, because $N - 1 \geq \mathsf{Dist}(working)$. Let $working'$ be equal to $working \cup \{(impl', spec')\}$. Adding any of them to *working* at line 21 does not change its minimal distance. Let $antichain'$ be *antichain* if $(impl', spec')$ was not inserted and $antichain \Cup (impl', spec')$ otherwise. By the invariant follows that $N - 1 \geq \mathsf{Dist}(antichain)$ and so by Lemma 5.11 if $(impl', spec')$ is inserted into *antichain* its distance will not change. Therefore, $\mathsf{Dist}(Done') > \mathsf{Dist}(working') \wedge \mathsf{Dist}(working') = \mathsf{Dist}(antichain')$.

Case $0 < N \leq \mathsf{Dist}(working \cup \{(impl, spec)\})$. Note that $N = \mathsf{Dist}(working \cup \{(impl, spec)\})$. From Lemma 5.12 follows that $\neg\mathsf{div}(\overline{spec})$ and so the successors of $(impl, spec)$ are explored. As $0 < N$, there must be some successors $(impl', spec')$ at line 12 such that $\mathsf{Dist}((impl', spec')) < N$. Towards a contradiction, assume that one of these successors $(impl', spec')$ is included in *antichain*, *i.e.*, $(impl', spec') \Subset antichain$. In that case, $\mathsf{Dist}((impl', spec')) \geq \mathsf{Dist}(antichain)$ by Lemma 5.11. Consequently, it follows that $\mathsf{Dist}((impl', spec')) \geq \mathsf{Dist}(working \cup \{(impl, spec)\}) = N$, which contradicts with $\mathsf{Dist}((impl', spec')) < N$. Therefore, none of them should already be included in *antichain*. Let $working'$ be equal to $working \cup \{(impl', spec')\}$. For some successor $(impl', spec') \notin antichain$ that is added to *working* at line 21, we observe that $\mathsf{Dist}((impl', spec')) < \mathsf{Dist}((impl, spec))$ and as such $\mathsf{Dist}(working') < \mathsf{Dist}(working \cup \{(impl, spec)\})$. Therefore also $\mathsf{Dist}(working') < \mathsf{Dist}(Done \cup \{(impl, spec)\})$. Finally, $\mathsf{Dist}(antichain \Cup \{(impl', spec')\}) = \mathsf{Dist}(working')$ follows from $\mathsf{Dist}(impl', spec') < \mathsf{Dist}(antichain)$ and Lemma 5.13.

Case $N = 0$. The state $(impl, spec)$ is checked for the FDR-witness conditions and the algorithm terminates. □

Now we can conclude the proof of correctness for Algorithm 2.

Theorem 5.14. Algorithm 2 returns false if and only if an FDR-witness is reachable in the product of $\mathcal{L}_1$ and $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$.

Proof. $\Longrightarrow$) Assume that Algorithm 2 returns false. This only occurs when the current $(impl, spec)$ pair satisfies the conditions of an FDR-witness, as shown in lines 6, 7, 10 and 17 of Algorithm 2. All pairs taken from *working* are reachable according to Lemma 5.3, so this FDR-witness is also reachable.

$\Longleftarrow$) Assume that an FDR-witness is reachable in the product of $\mathcal{L}_1$ and $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$. Towards a contradiction, assume that Algorithm 2 returns true. The Invariant II of Lemma 5.13 is equal to $\mathsf{Dist}(Done) > \mathsf{Dist}(working) \wedge \mathsf{Dist}(working) = \mathsf{Dist}(antichain)$. The algorithm returns true if and only if *working* is empty, which means that $\mathsf{Dist}(working) = \mathsf{Dist}(\emptyset) = \infty$. The initial state *init* of $\mathcal{L}_1$ and $\mathsf{norm}_{\mathsf{fdr}}(\mathcal{L}_2)$ is equal to $(init_1, \{s \in S_2 \mid init_2 \stackrel{\epsilon}{\rightsquigarrow}_2 s\})$ and can reach an FDR-witness by assumption. Therefore, $\mathsf{Dist}(init) < \infty$. Initially *init* was inserted into *antichain* so by Lemma 5.2 follows that $init \Subset antichain$ and from Lemma 5.11 follows that $\mathsf{Dist}(antichain) < \infty$, which leads to a contradiction. □

Theorem 5.1. Let $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ be two LTSs.

- $\textsc{refines}_{\textsc{new}}(\mathcal{L}_1, \mathcal{L}_2, \text{false})$ returns true if and only if $\mathcal{L}_1 \sqsupseteq_{\mathsf{sfr}} \mathcal{L}_2$.
- $\textsc{refines}_{\textsc{new}}(\mathcal{L}_1, \mathcal{L}_2, \text{true})$ returns true if and only if $\mathcal{L}_1 \sqsupseteq_{\mathsf{fdr}} \mathcal{L}_2$.

Proof. From Theorem 5.14 we can conclude that Algorithm 2 returns false if and only if an FDR-witness is reachable. By Theorem 3.3 an FDR-witness is only reachable if and only if $\mathcal{L}_1$ does not refine $\mathcal{L}_2$ in failures-divergences semantics. Virtually the same arguments apply for stable failures refinement. □

6 Experimental Validation

We have conducted several experiments to compare both algorithms to show that solving the indicated performance problems actually improves the run time of antichain-based refinement checking.

We have implemented a depth-first and breadth-first variant of Algorithm 1 and Algorithm 2 in a branch of the mCRL2[1] toolset [4] as part of the *ltscompare* tool. Both implementations use the same data structures and compute most concepts, *e.g.*, the antichain inclusion and insertion, in the same way. The implementation of Algorithm 2 performs the check at line 10 according to the correct refusal definition presented in Definition 2.8, whereas, the implementation of Algorithm 1 computes the refusal check according to the definition given in [22]. Algorithm 1 applies $\mathsf{refusals}$ to any, possibly unstable, implementation and specification states. Therefore, for Algorithm 1 we have implemented the refusals computation for any state, omitting the requirement that the given state is stable.

A downloadable package [11] was created that contains the source code and the benchmark files. In the following measurements, the *ltscompare* tool has been built using Clang 7.0.0 in the release configuration. The measurements have been performed on a machine with the following hardware specification:

- Intel Core i7-7700HQ CPU 2.80GHz
- Memory: 16GiB limit imposed by `ulimit -Sv 16777216`

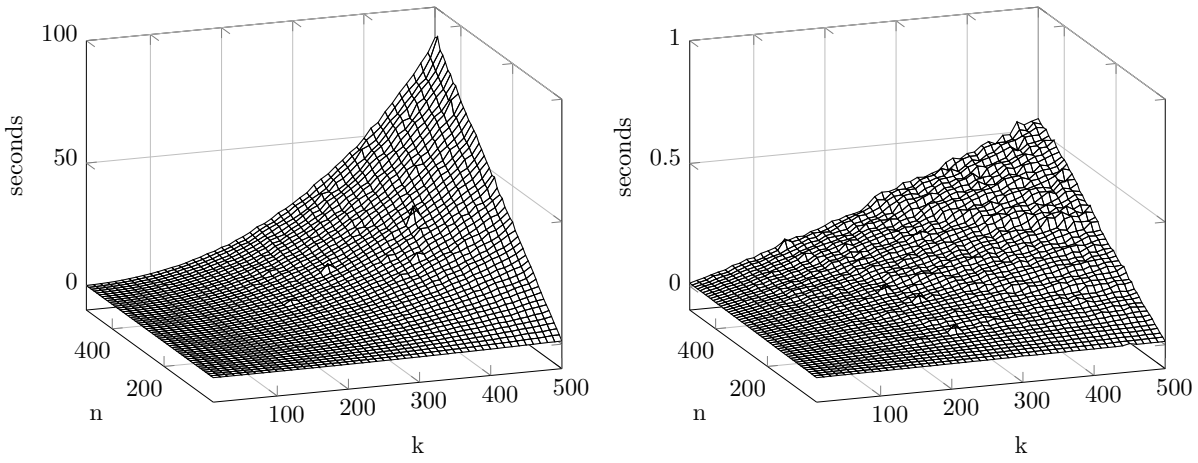
The input models are taken from three sources. First, Example 4.5 is benchmarked for various input combinations to show the asymptotic behaviour in practice. Second, six mCRL2 specifications were obtained from a master thesis that investigates the correctness of various concurrent data structures modelled in mCRL2 [12]. In that thesis weak trace equivalence was used to show the *linearisability* of these data structures. This is similar to the benchmarks related to linearisability that were performed in [22]. For benchmarking purposes we check the stronger stable failures and failures-divergences refinement relations between the implementation and specification pairs. Finally, a single industrial model, a control system modelled in the Dezyne language [17], is also included in the benchmarks as it was the first instance that exposed the performance issues. However, for confidentiality reasons, this model is not made publicly available.

[1] www.mcrl2.org

6.1 Benchmarking Example 4.5

Example 4.5 has been benchmarked for all combinations of parameters $n, k \in \{10, 20, \ldots, 500\}$ checking for the stable failures refinement $\mathcal{L}_n^k \sqsupseteq_{\mathsf{sfr}} \mathcal{L}_n^k$. Figure 1 shows the run time in seconds in a three dimensional plot for all combinations of n and k using the depth-first variants of both algorithms.

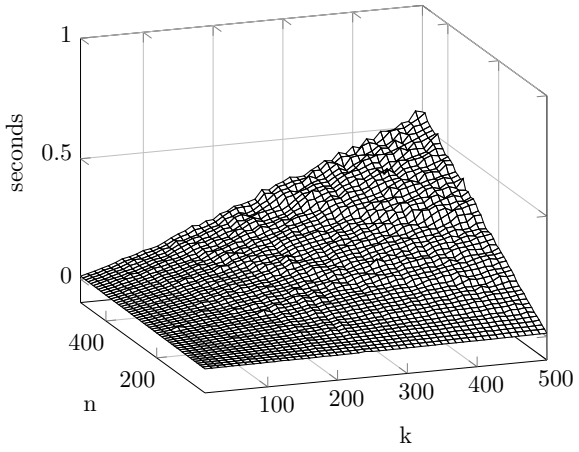
Figure 1: The run time results for Example 4.5 using the depth-first variant of Algorithm 1 on the left and our Algorithm 2 on the right.



These plots show a quadratic growth of Algorithm 1 in the parameter k and a linear growth in the parameter n. For Algorithm 2 the asymptotic growth is linear in both k and n. These observed growths match the analysis that was presented in Example 4.5 for Algorithm 1 and on page 14 for Algorithm 2. Note that the scale of the vertical axis, displaying the run time, differs by two orders of magnitude and the highest runtime (for the $n = 500$ and $k = 500$ case) of Algorithm 1 is a factor 170 higher than that of Algorithm 1. We observe that due to the absence of τ-transitions, there is no performance difference in the computation of refusals in both algorithms. Consequently, the difference in performance is entirely due to the different way of inspecting and extending *working* and *antichain*.

The breadth-first variant of Algorithm 1 was unable to complete the smallest $(10, 10)$ case within the given memory limit. However, for the corrected algorithm breadth-first the run time performance is almost equivalent to its depth-first variant, as shown in Figure 2.

Figure 2: The run time for Example 4.5 using the breadh-first variant of Algorithm 2.



6.2 Benchmarking Practical Examples

The next set of experiments consists of more typical refinement checking questions, assessing whether the behaviour of the implementation are indeed allowed by the corresponding specification in stable failures semantics. In Table 1 the origin of each benchmark, the number of states and transitions of each implementation and specification LTS, and whether the stable failures refinement relation holds is shown.

Table 1: The number of states and transitions in each benchmark.

benchmark	ref.	states impl	trans. impl	states spec	trans. spec	$\sqsupseteq_{\mathsf{sfr}}$
coarseSet	[8]	55444	145043	50488	64729	True
fineGrainedSet	[8]	5077	9006	3720	3305	True
industrial	-	24551	45447	24	45	True
lazySet	[8]	24496	41431	3565	3980	True
optimisticSet	[8]	234332	389344	25435	28154	True
pracNonBlock	[15]	3030	5799	1248	1473	False
treiberStack	[16]	205634	564862	87389	124740	True

The run time results, in seconds, for both algorithms to decide the existence of a refinement relation can be found in Table 2. We use † to indicate that the algorithm failed to complete within the given 16GiB limit. No experiment failed to complete within a time limit of 10 minutes. Each measurement shown is the average of five different runs.

Table 2: Run time comparison between Algorithm 1 and Algorithm 2 using depth-first (df) and breadth-first (bf) exploration.

benchmark	Alg. 1 df (s)	Alg. 1 bf (s)	Alg. 2 df (s)	Alg. 2 bf (s)
coarseSet	9.15	†	8.61	9.06
fineGrainedSet	0.37	†	0.32	0.46
industrial	1.36	296.29	0.15	0.17
lazySet	1.19	†	1.02	1.26
optimisticSet	16.96	†	14.13	22.67
pracNonBlock	0.03	0.17	0.02	0.09
treiberStack	148.39	†	137.52	352.59

Here, we observe that for the depth-first variant both algorithms perform similarly with a small performance advantage for Algorithm 2. However, for the breadth-first variants our algorithm is able to complete all benchmarks, whereas, Algorithm 1f is only able to complete two within the given memory limit.

To gain more insight into the performance differences between both algorithms we repeat the benchmarks and report a number of performance metrics. The metrics are the maximum *working* size and the number of *antichain* checks that fail (misses), succeeded (hits) and the maximum *antichain* size during the exploration. Note that the *antichain* size can decrease by inserting elements, because more than one pair could be removed as a result of an insertion.

The following two tables show the discussed metrics for depth-first variant of both algorithms.

Table 3: Performance indicators for the depth-first variant of Algorithm 1.

benchmark	*working* max	*antichain* hits	*antichain* misses	*antichain* max
coarseSet	96	93330	58438	55444
fineGrainedSet	60	5786	7575	5077
lazySet	61	21184	30771	24496
optimisticSet	96	234692	354068	238726
pracNonBlock	52	548	672	591
treiberStack	101	1238727	756692	234118
industrial	74	36544	43419	43091

Table 4: Performance indicators for the depth-first variant of Algorithm 2.

benchmark	*working* max	*antichain* hits	*antichain* misses	*antichain* max
coarseSet	96	93330	58438	55444
fineGrainedSet	60	5786	7575	5077
lazySet	61	21184	30771	24496
optimisticSet	96	234692	354068	238728
pracNonBlock	43	520	641	634
treiberStack	101	1238727	756692	234119
industrial	69	36369	43090	43091

Here, we can observe that for all experiments except for the industrial and pracNonBlock benchmarks the measurements are identical. This difference can be explained by the different refusal computation used as the refusal definition presented in [22] can search for an FDR-witness in a number of steps bypassing the exploration with *working*. For the other cases, the only difference in performance can be obtained from the different refusal computation, as the *antichain* and *working* operations are computed in the same way.

The following two tables show the metrics for the breadth-first variants of both algorithms.

Table 5: Performance indicators for the breadth-first variant of Algorithm 1.

benchmark	*working* max	*antichain* hits	*antichain* misses	*antichain* max
coarseSet	4710289	13870	7807403	3629
fineGrainedSet	6604516	180669	15547890	1900
lazySet	6726497	130523	14852835	4306
optimisticSet	6366524	38649	14238042	4439
pracNonBlock	6262	3078	14560	274
treiberStack	5829902	76114	8340606	4811
industrial	549263	5459028	12888388	43091

Table 6: Performance indicators for the breadth-first variant of Algorithm 2.

benchmark	*working* max	*antichain* hits	*antichain* misses	*antichain* max
coarseSet	3411	96167	60332	55444
fineGrainedSet	434	7192	9657	5077
lazySet	1748	24340	35192	24496
optimisticSet	15209	292525	434218	234352
pracNonBlock	338	3426	4032	2675
treiberStack	139218	2411614	1523830	214795
industrial	2243	36369	43090	43091

From these results it becomes clear that especially for the breadth-first variant delaying the insertion of state pairs into *antichain* results in many pairs failing the *antichain* check. Each pair that fails the check is added to *working*, which causes it to grow very rapidly. For Algorithm 2 we can observe that the *working* queue is larger than the depth-first variant due to the "width" of the state space. This also explains the slightly worse run time of breadth-first compared to depth-first even for a successful refinement check.

To confirm that the run time improvements are due to the refusal optimisation we have implemented another variant of Algorithm 1 with the stability check of *impl* added, but using the original refusal definition.

Table 7: Run time results for Algorithm 1 with the stability check of *impl*.

benchmark	Alg. 1 df (s)	Alg. 1 bf (s)
coarseSet	9.59	†
fineGrainedSet	0.36	†
industrial	0.17	26.32
lazySet	1.12	†
optimisticSet	15.85	†
pracNonBlock	0.03	†
treiberStack	156.67	†

As expected the run time for the depth-first variant now matches the results of the corrected algorithm more closely. However, for the breadth-first case the industrial run time is greatly reduced, but the benchmark pracNonBlock performs (much) worse. This can be explained due to the fact that in a failing refinement the exploration stops when a suitable (SF-)witness has been found. In this case it is more efficient to directly search the witness in the refusal calculation instead of continuing the search whenever the implementation state was stable, which causes the *working* queue to reach the memory limit.

Finally, we repeat the benchmarks with failures-divergences refinement, but only for Algorithm 2 as the original algorithm was incorrect. The run time results are presented Table 8.

Table 8: The run time results for checking failures-divergences refinement using Algorithm 2.

benchmark	Alg. 2 df (s)	Alg. 2 bf (s)	$\sqsupseteq_{\mathsf{fdr}}$
coarseSet	8.68	9.29	True
fineGrainedSet	0.33	0.48	True
industrial	0.05	0.05	False
lazySet	1.04	1.33	True
optimisticSet	14.55	23.81	True
pracNonBlock	0.08	0.1	True
treiberStack	140.7	363.34	True

This shows that deciding failures-divergences refinement has a similar performance to deciding stable failures. The only exception is that the industrial model now fails the refinement check and as such its run time is dependent on the exploration order.

6.3 Preprocessing

Refinement checking can be sped up by reducing the state spaces of the two transition systems to be compared. It is known that divergence-respecting weak bisimulation is strong enough to preserve stable failures and failures-divergence refinement [14]. Therefore, the implementation and specification LTSs can be reduced modulo divergence-respecting weak bisimulation. We can also use divergence-preserving branching bisimulation [18] as it is stronger than divergence-respecting weak bisimulation [20]. The known algorithms for divergence-preserving branching bisimulation are far more efficient than the ones for divergence-respecting weak bisimulation. The algorithm that we have used to minimize the LTSs modulo divergence-preserving branching bisimulation is presented in [7]. Its complexity is $\mathcal{O}(m(\log|Act| + \log n))$, where m is the number of transitions and n the number of states of the labelled transitions system.

In Table 9 the number of states and transitions of the benchmarks after reduction are shown.

Table 9: The number of states and transitions after diverging preserving branching bisimulation reduction.

benchmark	states impl	trans. impl	states spec	trans. spec
coarseSet	1089	3618	1089	3618
fineGrainedSet	92	210	92	210
industrial	24551	45447	24	45
lazySet	92	210	92	210
optimisticSet	170	410	170	410
pracNonBlock	119	274	163	378
treiberStack	4626	14380	7988	26070

We remark that Example 4.5 will have the same size after reduction using branching bisimulation as it does not contain τ-transitions and as such it has been excluded from these benchmarks.

In the following results the reduction time is both included in the run time and presented separately. For most of the benchmarks the implementation and specification LTSs are divergence preserving branching bisimilar so when we reduce both, stable failures refinement can be calculated in negligible time. Therefore, only the specification LTS is reduced when performing these benchmarks. Another advantage of only reducing the specification LTS is that it is easier to reconstruct the trace to the witness when presenting a counter example.

Table 10: Run time comparison between the original algorithm (Algorithm 1) and the corrected algorithm (Algorithm 2) using depth-first (df) and breadth-first (bf) exploration where the specification is reduced modulo divergence-preserving branching bisimulation.

benchmark	Alg. 1 df (s)	Alg. 1 bf (s)	Alg. 2 df (s)	Alg. 2 bf (s)	Reduction (s)
coarseSet	0.74	†	0.69	0.69	0.34
fineGrainedSet	0.04	†	0.04	0.04	0.02
industrial	1.38	293.1	0.16	0.17	0.01
lazySet	0.21	†	0.15	0.15	0.03
optimisticSet	2.52	†	1.59	1.57	0.15
pracNonBlock	0.02	0.04	0.02	0.02	0.02
treiberStack	8.19	†	6.61	11.71	0.66

Comparing these results with Table 2 shows that by reducing the specification modulo divergence-preserving branching bisimulation can substantially improve the performance of antichain-based algorithms and never harms.

For failures-divergences refinement the results for Algorithm!2) are again similar, which is shown in Table 11.

Table 11: The run time results for checking failures-divergences refinement using Algorithm 2 where the specification is reduced module divergence-preserving branching bisimulation.

benchmark	Alg. 2 df (s)	Alg. 2 bf (s)
coarseSet	0.75	0.7
fineGrainedSet	0.04	0.04
industrial	0.05	0.06
lazySet	0.15	0.15
optimisticSet	1.61	1.7
pracNonBlock	0.02	0.02
treiberStack	6.76	12.13

7 Conclusion

Our study of the antichain-based algorithms for deciding stable failures refinement and failures-divergences refinement presented in [22] revealed that the failures-divergences refinement algorithm is incorrect. Both algorithms perform suboptimally when implemented using a depth-first search strategy and poorly when implemented using a breadth-first search strategy. Furthermore, both violate the claimed antichain property. We propose alternative algorithms for which we have shown correctness and which utilise proper antichains. Our experiments indicate significant performance improvements for deciding stable failures refinement and a performance of deciding failures-divergences refinement that is comparable to deciding stable failures refinement. We also show that preprocessing using divergence-preserving branching bisimulation offers substantial performance benefits. The implementation of our algorithms is available in the open source toolset mCRL2 [4] and is currently used as the backbone in the commercial F-MDE toolset Dezyne; see also [17].

References

[1] P. A. Abdulla, Y. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In J. Esparza and R. Majumdar, editors, *TACAS 2010*, volume 6015 of *LNCS*, pages 158–174. Springer, 2010.

[2] J. A. Bergstra, J. W. Klop, and E. Olderog. Failures without chaos: a new process semantics for fair abstraction. In M. Wirsing, editor, *IFIP TC 2/WG 2.2 1986*, pages 77–104. North-Holland, 1987.

[3] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency 1984*, volume 197 of *LNCS*, pages 281–305. Springer, 1984.

[4] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. An overview of the mCRL2 toolset and its recent advances. In N. Piterman and S. A. Smolka, editors, *TACAS 2013*, volume 7795 of *LNCS*, pages 199–213. Springer, 2013.

[5] T. Gibson-Robinson, P. J. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 - A modern refinement checker for CSP. In E. Ábrahám and K. Havelund, editors, *TACAS 2014*, volume 8413 of *LNCS*, pages 187–201. Springer, 2014.

[6] A. O. Gomes and A. Butterfield. Modelling the haemodialysis machine with circus. In M. J. Butler, K. Schewe, A. Mashkoor, and M. Biró, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016, Linz, Austria, May 23-27, 2016, Proceedings*, volume 9675 of *LNCS*, pages 409–424. Springer, 2016.

[7] J. F. Groote, D. N. Jansen, J. J. A. Keiren, and A. Wijs. An $O(m\log n)$ algorithm for computing stuttering equivalence and branching bisimulation. *ACM Trans. Comput. Log.*, 18(2):13:1–13:34, 2017.

[8] M. Herlihy and N. Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[9] C. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[10] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput.*, 86(1):43–68, 1990.

[11] M. Laveaux. Downloadable sources and benchmarks for the experimental validation. 2019. https://doi.org/10.5281/zenodo.2573095.

[12] R. Paval. Modeling and verifying concurrent data structures. Master's thesis, Eindhoven University of Technology, 2018.

[13] A. Roscoe. Model-checking CSP. In A. Roscoe, editor, *A Classical Mind: essays in Honour of C.A.R. Hoare*, chapter 21, pages 353–378. Prentice Hall International (UK) Ltd., 1994.

[14] A. W. Roscoe. *Understanding Concurrent Systems.* Texts in Computer Science. Springer, 2010.

[15] C. Shann, T. Huang, and C. Chen. A practical nonblocking queue algorithm using compare-and-swap. In *ICPADS 2000*, pages 470–475. IEEE Computer Society, 2000.

[16] R. Treiber. *Systems programming: Coping with parallelism.* International Business Machines Incorporated, Thomas J. Watson Research, 1986.

[17] R. van Beusekom, J. F. Groote, P. F. Hoogendijk, R. Howe, W. Wesselink, R. Wieringa, and T. A. C. Willemse. Formalising the Dezyne modelling language in mCRL2. In L. Petrucci, C. Seceleanu, and A. Cavalcanti, editors, *FMICS-AVoCS 2017*, volume 10471 of *LNCS*, pages 217–233. Springer, 2017.

[18] R. van Glabbeek, B. Luttik, and N. Trcka. Branching bisimilarity with explicit divergence. *Fundam. Inform.*, 93(4):371–392, 2009.

[19] R. J. van Glabbeek. Personal Communication, 7 January 2019.

[20] R. J. van Glabbeek. The linear time - branching time spectrum II. In E. Best, editor, *CONCUR 1993*, volume 715 of *LNCS*, pages 66–81. Springer, 1993.

[21] R. J. van Glabbeek. A branching time model of CSP. In T. Gibson-Robinson, P. J. Hopcroft, and R. Lazic, editors, *Concurrency, Security, and Puzzles - Essays Dedicated to Andrew William Roscoe on the Occasion of His 60th Birthday*, volume 10160 of *LNCS*, pages 272–293. Springer, 2017.

[22] T. Wang, S. Song, J. Sun, Y. Liu, J. S. Dong, X. Wang, and S. Li. More anti-chain based refinement checking. In T. Aoki and K. Taguchi, editors, *ICFEM*, volume 7635 of *LNCS*, pages 364–380. Springer, 2012.

A Introducing ghost variable Done

The same pseudo code as shown in Algorithm 2 with the ghost variable *Done* introduced. This variable indicates that a certain state pair was already processed and is used by a number of lemmas in the proof of correctness.

Algorithm 3 The corrected refinement checking algorithm with *Done* For LTSs $\mathcal{L}_i = (S_i, init_i, Act_i, \rightarrow_i)$ where $i \in \{1, 2\}$ the algorithm returns *true* iff $\mathcal{L}_1$ refines $\mathcal{L}_2$. This algorithm decides stable failures refinement if CheckDiv is false and failures-divergences refinement otherwise. The *Done* variable indicates state pairs that have been fully processed.

1: **procedure** REFINES$_{\text{NEW}}$($\mathcal{L}_1, \mathcal{L}_2$, CheckDiv)
2: let *working* be a queue containing a pair $(init_1, \{s \in S_2 \mid init_2 \overset{\epsilon}{\rightsquigarrow}_2 s\})$
3: let *antichain* $\leftarrow \emptyset \Cup (init_1, \{s \in S_2 \mid init_2 \overset{\epsilon}{\rightsquigarrow}_2 s\})$
4: $\{Done \leftarrow \emptyset\}$
5: **while** *working* $\neq \emptyset$ **do**
6: pop $(impl, spec)$ from *working*
7: **if** $\neg$CheckDiv $\vee \neg$div($spec$) **then**
8: **if** CheckDiv $\wedge$ div($impl$) **then**
9: **return** false
10: **else**
11: **if** stable($impl$) $\wedge$ refusals($impl$) $\subsetneq$ refusals($\overline{spec}$) **then**
12: **return** false
13: **for** $impl \xrightarrow{e}_1 impl'$ **do**
14: **if** $e = \tau$ **then**
15: $spec' \leftarrow spec$
16: **else**
17: $spec' \leftarrow \{s' \in S_2 \mid \exists s \in spec : s \overset{e}{\rightsquigarrow}_2 s'\}$
18: **if** $spec' = \emptyset$ **then**
19: **return** false
20: **if** $(impl', spec') \notin antichain$ **then**
21: $antichain \leftarrow antichain \Cup (impl', spec')$
22: push $(impl', spec')$ into *working*
23: $\{Done \leftarrow Done \cup (impl, spec)\}$
24: **return** true