

---

# CORRECT AND EFFICIENT ANTICHAIN ALGORITHMS FOR REFINEMENT CHECKING

MAURICE LAVEAUX, JAN FRISO GROOTE, AND TIM A.C. WILLEMSE

Eindhoven University of Technology. De Groene Loper 5, 5612 AE, Eindhoven, The Netherlands  
*e-mail address:* m.laveaux@tue.nl

Eindhoven University of Technology. De Groene Loper 5, 5612 AE, Eindhoven, The Netherlands  
*e-mail address:* j.f.groote@tue.nl

Eindhoven University of Technology. De Groene Loper 5, 5612 AE, Eindhoven, The Netherlands  
*e-mail address:* t.a.c.willemse@tue.nl

---

**ABSTRACT.** The notion of refinement plays an important role in software engineering. It is the basis of a stepwise development methodology in which the correctness of a system can be established by proving, or computing, that a system refines its specification. Wang *et al.* describe algorithms based on antichains for efficiently deciding trace refinement, 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 additional run time improvements can be obtained by applying divergence-preserving branching bisimulation minimisation.

## 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 [Hoa85, Ros94, Ros10].

This formal view on refinement—as a mathematical relation between a specification and its implementation—has been used successfully in industrial settings [GB16, GBC<sup>+</sup>17], and it has been incorporated in commercial Formal Model-Driven Engineering tools such as *Dezyne* [vBGH<sup>+</sup>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 instance, *trace refinement* [Ros10] only preserves safety properties.

---

*Key words and phrases:* Refinement checking, antichains, algorithms, benchmarks.

The—arguably—most prominent refinement relations for the theory of CSP are the *stable failures refinement* [BKO87, Ros10] and *failures-divergences refinement* [Ros10]. All three refinement relations are implemented in the FDR [GABR14, GABR16] tool for specifying and analysing CSP processes.

Trace refinement, stable failures refinement and failures-divergences refinement are computationally hard problems; deciding whether there is a refinement relation between an implementation and a specification, both represented by CSP processes or labelled transition systems, is PSPACE-hard [KS90]. In practice, however, tools such as FDR are able to work with quite large state spaces. The basic algorithm for deciding a trace refinement, 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 [WSS<sup>+</sup>12] and inspired by successes reported, *e.g.*, in [ACH<sup>+</sup>10, DR10, WDHR06], *antichain* techniques can be exploited to improve on the performance of refinement checking algorithms. Unfortunately, a closer inspection of the results and algorithms in [WSS<sup>+</sup>12] reveals several issues. First, the definitions of stable failures refinement and failures-divergences refinement used in [WSS<sup>+</sup>12] do not match the definitions of [BKO87, Ros10], nor do they seem to match known relations from the literature [vG19].

Second, as we demonstrate in Example 4.3 in this paper, the results [WSS<sup>+</sup>12, Theorems 2 and 3] claiming correctness of their algorithms for deciding both non-standard refinement relations are incorrect. We do note that their algorithm for checking trace refinement is correct, and their algorithm for checking stable failures refinement correctly decides the refinement relation defined by [BKO87, Ros10].

Third, unlike claimed by the authors, the algorithms of [WSS<sup>+</sup>12] violate the antichain property as we demonstrate in Example 4.5. 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  $|Act| \cdot |S|$ , where  $Act$  is the set of actions and  $S$  the set of states of the implementation, as we show in Example 4.4. This factor is even greater, *viz.*  $|Act|^{|S|}$ , 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 [Ros94]; *e.g.*, the conciseness of counterexamples to refinement.

Our contributions are the following. Apart from pointing out the issues in [WSS<sup>+</sup>12], we propose new antichain-based algorithms for deciding trace refinement, stable failures refinement and failures-divergences refinement and we prove their correctness. We compare the performance of the trace refinement algorithm and the stable failures refinement algorithm of [WSS<sup>+</sup>12] 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 [WSS<sup>+</sup>12] generally become quite easy using our algorithm. Finally, we show that divergence-preserving branching bisimulation [vG93, vGLT09] minimisation preserves the desired refinement checking relations and that applying this minimisation as a preprocessing step can yield significant run time improvements.

The current paper is based on [LGW19], but extends it in several respects. First, in addition to stable failures refinement and failures-divergences refinement, the current

exposition also includes a treatment of trace refinement. Second, we include detailed proofs of correctness of the main claims in [LGW19], and we include several results that are needed to support these claims, but which are also of value on their own. Some of the more straightforward proofs have been deferred to the appendix. Third, we expand further on our experimental results. In particular, we have included more performance metrics, to help explain the observed performance improvements of our algorithms over those of [WSS<sup>+</sup>12], and we have included a section on using divergence-preserving branching bisimulation for preprocessing.

**Outline.** In Section 2 the preliminaries of labelled transition systems and the refinement relations are defined. In Section 3 a general procedure for checking refinement relations is described. In Section 4 the antichain-based algorithms of [WSS<sup>+</sup>12] are presented and their issues are described in detail. In Section 5 the improved antichain algorithms are presented and their correctness is shown. Finally, in Section 6 an experimental evaluation is conducted to show the effectiveness of these changes in practice, followed by the evaluation of applying divergence-preserving branching bisimulation minimisation as a preprocessing step.

## 2. PRELIMINARIES

In this section the preliminaries of labelled transition systems and the considered refinement relations are introduced. We follow the standard conventions, notation and definitions of [BR84, Ros10, vG17].

**2.1. Labelled Transition Systems.** Let  $Act$  be a finite set of actions that does not contain the constant  $\tau$ , which models *internal* actions, and let  $Act_\tau$  be equal to  $Act \cup \{\tau\}$ .

**Definition 2.1.** A labelled transition system  $\mathcal{L}$  is a tuple  $(S, \iota, \rightarrow)$  where  $S$  is a set of states;  $\iota \in S$  is an initial state and  $\rightarrow \subseteq S \times Act_\tau \times S$  is a labelled transition relation.

We depict labelled transition systems as edge-labelled directed graphs, where vertices represent states and the labelled edges between vertices represent the transitions. An incoming arrow with no starting state and no action indicates the initial state. We use the initial state to refer to a depicted LTS.

For the remainder of this section, we assume that  $\mathcal{L} = (S, \iota, \rightarrow)$  is an arbitrary LTS. We adopt the following conventions and notation. Typically, we use symbols  $s, t, u$  to denote states,  $U, V$  to denote sets of states and  $a$  to denote actions. A transition  $(s, a, t) \in \rightarrow$  is also written as  $s \xrightarrow{a} t$ . The set of *enabled* actions of state  $s$  is defined as  $\text{enabled}(s) = \{a \in Act_\tau \mid \exists t \in S : s \xrightarrow{a} t\}$ .

A sequence is denoted by concatenation, *i.e.*,  $a_0 a_1 \cdots a_{n-1}$  where  $a_i \in Act_\tau$  for all  $0 \leq i < n$  is a sequence of actions and  $Act_\tau^*$  indicates the set of all finite sequences of actions. We use  $\sigma$  and  $\rho$  to denote a sequence of actions, where  $\rho$  typically does not contain  $\tau$ . The *length* of a sequence, denoted as  $|a_0 a_1 \cdots a_{n-1}|$ , is equal to  $n$ . Finally, we say that any sequence  $a_0 a_1 \cdots a_k$  such that  $k \leq n-1$  is a *prefix* of a sequence  $a_0 a_1 \cdots a_{n-1}$ . A prefix is *strict* whenever its length is strictly smaller than that of the sequence itself.

The transition relation of an LTS is generalised to sequences of actions as follows:  $s \xrightarrow{\epsilon} t$  holds iff  $s = t$ , and  $s \xrightarrow{\sigma a} t$  holds iff there is a state  $u$  such that  $s \xrightarrow{\sigma} u$  and  $u \xrightarrow{a} t$ . The *weak transition* relation  $\Longrightarrow \subseteq S \times Act^* \times S$  is the smallest relation satisfying:

- $s \xRightarrow{\epsilon} s$ , and

- $s \xRightarrow{\epsilon} t$  if  $s \xrightarrow{\tau} t$ , and
- $s \xRightarrow{a} t$  if  $s \xrightarrow{a} t$  for  $a \in Act$ , and
- $s \xRightarrow{\rho\sigma} t$  if there is a state  $u$  such that  $s \xRightarrow{\rho} u$  and  $u \xRightarrow{\sigma} t$ .

**Definition 2.2.** Traces, weak traces and reachable states are defined as follows:

- The *traces* starting in state  $s$  are defined as  $\text{traces}(s) = \{\sigma \in Act_\tau^* \mid \exists t \in S : s \xRightarrow{\sigma} t\}$ . We define  $\text{traces}(\mathcal{L})$  to be  $\text{traces}(\iota)$ .
- The *weak traces* starting in state  $s$  are defined as  $\text{weaktraces}(s) = \{\rho \in Act^* \mid \exists t \in S : s \xRightarrow{\rho} t\}$ . We define  $\text{weaktraces}(\mathcal{L})$  to be  $\text{weaktraces}(\iota)$ .
- the set of states, *reachable* from  $s$  is defined as  $\text{reachable}(s) = \{t \in S \mid \exists \sigma \in Act_\tau^* : s \xRightarrow{\sigma} t\}$ . We define  $\text{reachable}(\mathcal{L})$  to be  $\text{reachable}(\iota)$ .

**Definition 2.3.** Labelled transition system  $\mathcal{L}$  is:

- *deterministic* if and only if for all states  $s, t, u$  and actions  $a \in Act_\tau$  if there are transitions  $s \xrightarrow{a} t$  and  $s \xrightarrow{a} u$  then  $t = u$ .
- *concrete* if it does not contain transitions labelled with  $\tau$ , i.e., for all states  $s$  it holds that  $\tau \notin \text{enabled}(s)$ .
- *universal* if and only if for all states  $s$  it holds that  $\text{enabled}(s) = Act$ .

**Lemma 2.4.** Let  $\mathcal{L}$  be a deterministic LTS. For all sequences  $\sigma \in Act_\tau^*$  and states  $s, t, u$ , if  $s \xRightarrow{\sigma} t$  and  $s \xRightarrow{\sigma} u$  then  $t = u$ .

The models underlying the CSP process algebra [Hoa85, Ros10] build on observations of *weak traces*, *failures* and *divergences*. A weak trace observation records the visible actions that occur when performing an experiment on the system. A failure is a combination of a set of actions that a system observably refuses and a weak trace experiment on the system that leads to the observation of the refusals. A refusal can only be observed when the system has *stabilised*, meaning that it can no longer perform internal behaviour. A *divergence* can be understood as the potential inability of the system to stabilise, which can happen when the system engages in an infinite sequence of  $\tau$ -actions after performing an experiment on the system.

**Definition 2.5** (Refusals). A state  $s$  is *stable*, denoted by  $\text{stable}(s)$ , if and only if  $\tau \notin \text{enabled}(s)$ . For a stable state  $s$ , the *refusals* of  $s$  are defined as  $\text{refusals}(s) = \mathcal{P}(Act \setminus \text{enabled}(s))$ . For a set of states  $U \subseteq S$  its refusals are defined as  $\text{refusals}(U) = \{X \subseteq Act \mid \exists s \in U : \text{stable}(s) \wedge X \in \text{refusals}(s)\}$ .

Formally, a state  $s$  is *diverging*, denoted by the predicate  $s \uparrow$ , if and only if there is an infinite sequence of states  $s \xrightarrow{\tau} s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \dots$ . For a set of states  $U$ , we write  $U \uparrow$ , iff  $s \uparrow$  for some state  $s \in U$ .

**Definition 2.6** (Divergences). The *divergences* of a state  $s$  are defined as  $\text{divergences}(s) = \{\rho\sigma \in Act^* \mid \exists t \in S : (s \xRightarrow{\rho} t \wedge t \uparrow)\}$ . We define  $\text{divergences}(\mathcal{L}) = \text{divergences}(\iota)$ .

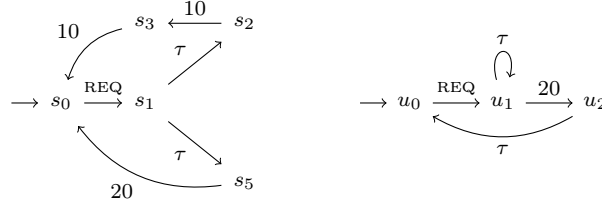
Observe that a divergence is any weak trace that has a prefix  $\rho$  which can reach a diverging state. This is based on the assumption that divergences lead to *chaos*. In theories such as CSP, in which divergences are considered chaotic, chaos obscures all information about the behaviours involving a diverging state; we refer to this as obscuring *post-divergences details*.

**Definition 2.7** (Stable failures). The set of all *stable failures* of a state  $s$  is defined as  $\text{failures}(s) = \{(\rho, X) \in Act^* \times \mathcal{P}(Act) \mid \exists t \in S : (s \xRightarrow{\rho} t \wedge \text{stable}(t) \wedge X \in \text{refusals}(t))\}$ . The

set of failures with post-divergences details *obscured* is defined as  $\text{failures}_\perp(s) = \text{failures}(s) \cup \{(\rho, X) \in \text{Act}^* \times \mathcal{P}(\text{Act}) \mid \rho \in \text{divergences}(s)\}$ .

We illustrate these concepts by means of an example.

**Example 2.8.** Consider the LTSs  $s_0$  and  $u_0$  depicted below.



We observe that states  $s_0, s_2, s_3, s_5$  and  $u_0$  are stable. For each of these states we can determine their refusals, *e.g.*, state  $s_0$  has the refusals  $\{\emptyset, \{10\}, \{20\}, \{10, 20\}\}$  as given by  $\text{refusals}(s_0)$ . Furthermore, we observe that  $\text{REQ } 20$  is a weak trace of  $s_0$  to itself. Consequently, it follows that for example the pairs  $(\text{REQ } 20, \{10\})$  and  $(\text{REQ } 20, \{10, 20\})$  are failures of  $s_0$ . None of the states in  $s_0$  diverge and as such the corresponding set of divergences are empty and both notions of failures coincide. However, for state  $u_1$  we can see that  $u_1 \uparrow$  holds and therefore  $\text{REQ}$ , but also  $\text{REQ } 10$  is a possible divergence of  $u_0$ , *i.e.*,  $\text{REQ } 10 \in \text{divergences}(u_0)$ . This also means that  $(\text{REQ } 10, \{10\}) \in \text{failures}_\perp(u_0)$  is a failure of  $u_0$  with post-divergences details obscured.  $\square$

The three models of CSP building on the different powers of observation are the *weak trace* model, the *stable failures* model and the *failures-divergences* model. The refinement relations, induced by these models, are called *trace refinement*, *stable failures refinement* and *failures-divergences refinement* respectively.

**Definition 2.9** (Refinement). Let  $\mathcal{L}_1$  and  $\mathcal{L}_2$  be two LTSs.

- $\mathcal{L}_1$  is refined by  $\mathcal{L}_2$  in trace semantics, denoted by  $\mathcal{L}_1 \sqsubseteq_{\text{tr}} \mathcal{L}_2$ , if and only if  $\text{weaktraces}(\mathcal{L}_2) \subseteq \text{weaktraces}(\mathcal{L}_1)$ .
- $\mathcal{L}_1$  is refined by  $\mathcal{L}_2$  in stable failures semantics, denoted by  $\mathcal{L}_1 \sqsubseteq_{\text{sfr}} \mathcal{L}_2$ , if and only if  $\text{failures}(\mathcal{L}_2) \subseteq \text{failures}(\mathcal{L}_1)$  and  $\text{weaktraces}(\mathcal{L}_2) \subseteq \text{weaktraces}(\mathcal{L}_1)$ .
- $\mathcal{L}_1$  is refined by  $\mathcal{L}_2$  in failures-divergences semantics, denoted by  $\mathcal{L}_1 \sqsubseteq_{\text{fdr}} \mathcal{L}_2$ , if and only if both  $\text{failures}_\perp(\mathcal{L}_2) \subseteq \text{failures}_\perp(\mathcal{L}_1)$  and  $\text{divergences}(\mathcal{L}_2) \subseteq \text{divergences}(\mathcal{L}_1)$ .

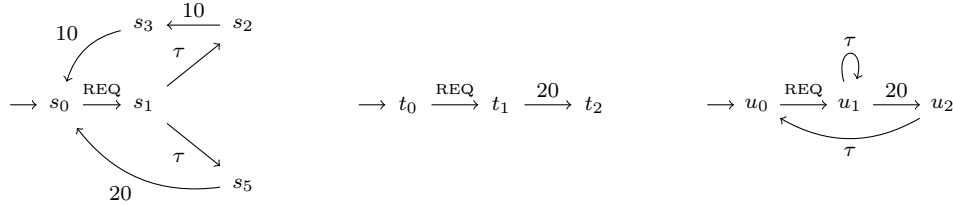
The LTS that is typically refined is referred to as the *specification*, whereas the LTS that refines the specification is referred to as the *implementation*.

**Remark 2.10.** We observe that each refinement relation between LTSs is inverted with respect to the subset relation in its corresponding definition. However, in the setting of CSP where these refinement relations are fundamental and have been extensively studied [BR84, Ros10, vG17], refinement is viewed as an ordering between processes where the process that does not restrict anything, *e.g.*, the process with all failures, is seen as the smallest, least restrictive specification.

**Remark 2.11.** The notions defined above appear in different formulations in [WSS<sup>+</sup>12]. Their definition of stable failures refinement omits the clause for weak trace inclusion, and their definition of failures-divergences refinement replaces  $\text{failures}_\perp$  with  $\text{failures}$ . This yields refinement relations different from the standard ones and neither relation seems to appear in the literature [vG19].

We conclude with a small example, illustrating the uses of, and differences between the various refinement relations.

**Example 2.12.** Consider the LTSs  $s_0$  and  $u_0$  of Example 2.8 again and the LTS  $t_0$  depicted in between. We now consider  $s_0$  to be the specification of a simplified automated teller machine.



In the specification  $s_0$  the user can first request, by action REQ, an amount of twenty from the machine. The machine can then satisfy this request by either choosing to give twenty directly or by presenting two times ten to the user, which might vary depending on availability within the machine. Note that the distinction between user-initiated and response actions is only for the sake of the explanation and is not formally present in the LTS.

An implementation of this specification is *valid* if and only if it refines the specification in the required refinement semantics. Let us consider  $t_0$  as a first implementation of this machine. The weak traces of  $t_0$ , consisting of the set  $\{\epsilon, \text{REQ}, \text{REQ } 20\}$ , are included in the specification and therefore  $s_0 \sqsubseteq_{\text{tr}} t_0$ . Trace refinement is suitable for safety properties; for example the absence of infinite 10 or 20 actions without matching requests can be specified in such semantics. However, trace refinement does not preserve liveness properties. In particular, deadlocks are not preserved by trace refinement. In contrast, stable failures refinement *does* preserve deadlock freedom. For instance, the observation of the failure  $(\text{REQ } 20, \{\text{REQ}, 20, 10\})$  of  $t_0$  leads to the *deadlocked* state  $t_2$ , *i.e.*, a state with no outgoing transitions. This failure is not among the failures that can be observed of state  $s_0$ . Consequently,  $s_0 \not\sqsubseteq_{\text{sfr}} t_0$ .

The second implementation that we consider is  $u_0$ . The self-loop above state  $u_1$  might indicate that the machine uses (repeated) polling via a potentially unstable connection to determine whether the user's bank account permits the requested withdrawal. Note that  $u_1$  is not a stable state; all failures are thus of the shape  $(\text{REQ } 20 \text{ REQ } 20 \dots, \{10, 20\})$  and these are permitted observations for the given specification  $s_0$ . Therefore, this implementation is a valid stable failures refinement of the given specification, *i.e.*,  $s_0 \sqsubseteq_{\text{sfr}} u_0$ . The divergences  $\text{REQ } \rho$  for sequences  $\rho \in \text{Act}^*$  cause this implementation not to be valid under failures-divergences refinement, *i.e.*,  $s_0 \not\sqsubseteq_{\text{fdr}} u_0$ . From the perspective of the user, it is indeed questionable whether  $u_0$  constitutes a proper implementation, as she may perceive a divergence of the system as a deadlock.  $\square$

Note that stable failures refinement is a stronger relation than trace refinement; *i.e.*, whenever  $\sqsubseteq_{\text{sfr}}$  holds then also necessarily  $\sqsubseteq_{\text{tr}}$  holds. This does not hold the other way around as already shown in Example 2.12 where  $s_0 \sqsubseteq_{\text{tr}} t_0$  and  $s_0 \not\sqsubseteq_{\text{sfr}} t_0$ . Furthermore,  $\sqsubseteq_{\text{fdr}}$  is incomparable to  $\sqsubseteq_{\text{tr}}$ . In the preceding example, we have  $u_0 \sqsubseteq_{\text{fdr}} s_0$ , but  $u_0 \not\sqsubseteq_{\text{tr}} s_0$  because  $\text{REQ } 10 \in \text{weaktraces}(s_0)$  and  $\text{REQ } 10 \notin \text{weaktraces}(u_0)$ . But we also have  $s_0 \sqsubseteq_{\text{tr}} t_0$  and  $s_0 \not\sqsubseteq_{\text{fdr}} t_0$ , where the latter fails because  $(\text{REQ } 20, \{\text{REQ}, 20, 10\}) \in \text{failures}_{\perp}(t_0)$ , but  $(\text{REQ } 20, \{\text{REQ}, 20, 10\}) \notin \text{failures}_{\perp}(s_0)$ . Similarly,  $\sqsubseteq_{\text{fdr}}$  is incomparable to  $\sqsubseteq_{\text{sfr}}$ . For instance, we have  $s_0 \sqsubseteq_{\text{sfr}} u_0$  and  $s_0 \not\sqsubseteq_{\text{fdr}} u_0$  in Example 2.12, but also  $u_0 \sqsubseteq_{\text{fdr}} s_0$  and  $u_0 \not\sqsubseteq_{\text{sfr}} s_0$ , where for the latter we observe that  $(\text{REQ}, \{10\}) \in \text{failures}(s_0)$  but  $(\text{REQ}, \{10\}) \notin \text{failures}(u_0)$ .

### 3. REFINEMENT CHECKING

In general, the set of weak traces, failures and divergences of an LTS can be infinite. Therefore, checking inclusion of these sets directly is not viable. In [Ros94, Ros10], an algorithm to decide refinement between two labelled transition systems is sketched. As a preprocessing step to this algorithm, all diverging states in both LTSs are marked. The algorithm then relies on exploring the product of a *normal form* representation of the specification, *i.e.*, the LTS that is to be refined, and the implementation.

For each state in this product it checks whether it can locally decide non-refinement of the implementation state with the normal form state. A state for which non-refinement holds is referred to as a *witness*. Following [Ros10, WSS<sup>+</sup>12] and specifically the terminology of [Ros94], we formalise the product between LTSs that is explored by the procedure.

**Definition 3.1** (Product). Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs. The *product* of  $\mathcal{L}_1$  and  $\mathcal{L}_2$ , denoted by  $\mathcal{L}_1 \times \mathcal{L}_2$ , is an LTS  $(S, \iota, \rightarrow)$  such that  $S = S_1 \times S_2$  and  $\iota = (\iota_1, \iota_2)$ . The transition relation  $\rightarrow$  is the smallest relation such that for all  $s_1, t_1 \in S_1$ , and  $s_2, t_2 \in S_2$  and  $a \in Act$ :

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

The proposition below relates the behaviours of two LTSs to the behaviours of their product.

**Proposition 3.2.** Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs and let  $\mathcal{L}_1 \times \mathcal{L}_2 = (S, \iota, \rightarrow)$ . For all states  $(s_1, t_1), (s_2, t_2) \in S$  and all sequences  $\rho \in Act^*$  it holds that  $(s_1, s_2) \xRightarrow{\rho} (t_1, t_2)$  if and only if  $s_1 \xRightarrow{\rho}_1 t_1$  and  $s_2 \xRightarrow{\rho}_2 t_2$ .

*Proof.* First, we can show that the statement holds for the empty sequence by induction on the length of sequences in  $\tau^*$ . Using this we can prove the statement for all sequences in  $Act^*$  by induction on their length using the previous result in the base case.  $\square$

The normal form LTS of a given LTS is obtained using a typical subset construction as is common when determinising a transition system. A difference between determinisation and normalisation is that the former yields a transition system that preserves and reflects the set of weak traces of the given LTS. This is not the case for the latter, which may add weak traces not present in the original LTS. We first introduce the normal form LTS of a given LTS that is adequate for reducing the trace refinement and stable failures decision problems to a reachability problem.

**Definition 3.3** (Normal form). Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS. The normal form of  $\mathcal{L}$  is the LTS  $\text{norm}(\mathcal{L}) = (S', \iota', \rightarrow')$ , where  $S' = \mathcal{P}(S)$ ,  $\iota' = \{s \in S' \mid \iota \xRightarrow{\epsilon} s\}$  and  $\rightarrow'$  is defined as  $U \xrightarrow{a}' V$  if and only if  $V = \{t \in S' \mid \exists s \in U : s \xRightarrow{a} t\}$  for all sets of states  $U, V \subseteq S'$  and actions  $a \in Act$ .

Notice that  $\emptyset$  is a state in a normal form LTS. Furthermore, the normal form LTS is *deterministic*, *concrete* and *universal*.

Since a normal form LTS is concrete, all of its states are stable. The states of the original LTS comprising a normal form state may not be stable, however. When we need to reason about the stability and refusals of the set of states  $U$  in the LTS  $\mathcal{L}$  underlying a normal form LTS, rather than the state  $U$  of the normal form LTS, we therefore write  $\llbracket U \rrbracket_{\mathcal{L}}$  whenever we wish to stress that we refer to the set of states in  $\mathcal{L}$  that comprise  $U$ .

The three lemmas stated below relate the set of weak traces of an LTS  $\mathcal{L}$  to the set of traces of  $\text{norm}(\mathcal{L})$ .

**Lemma 3.4.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  and states  $U \in S'$  such that  $\iota' \xrightarrow{\rho} U$ , it holds that  $\iota \xRightarrow{\rho} s$  for all  $s \in \llbracket U \rrbracket_{\mathcal{L}}$ .*

**Lemma 3.5.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  and for all states  $s \in S$  such that  $\iota \xRightarrow{\rho} s$ , there is a state  $U \in S'$  such that  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  and  $\iota' \xrightarrow{\rho} U$ .*

The lemma below clarifies the role of the state  $\emptyset$  in  $\text{norm}(\mathcal{L})$ .

**Lemma 3.6.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  it holds that  $\rho \notin \text{weaktraces}(\mathcal{L})$  if and only if  $\iota' \xrightarrow{\rho} \emptyset$ .*

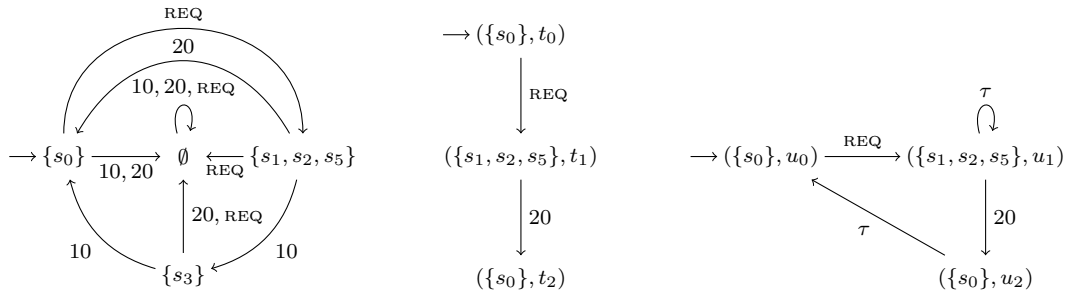
The structure explored by the refinement checking procedure of [Ros94, Ros10] for two LTSs  $\mathcal{L}_1$  and  $\mathcal{L}_2$  is the product  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$  in case of trace refinement and stable failures refinement. For these structures the related witnesses, where the reachability of such a witness indicates non-refinement, are then as follows:

**Definition 3.7** (Witness). Let  $\mathcal{L}_1$  and  $\mathcal{L}_2$  be LTSs. A state  $(U, s)$  of product  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ :

- is called a *TR-witness* if and only if  $U = \emptyset$ .
- is called an *SF-witness* if and only if at least one of the following conditions hold:
  - $U = \emptyset$ .
  - $\text{stable}(s)$  and  $\text{refusals}(s) \not\subseteq \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ .

We illustrate the notion of a witness, and in particular the relation between the reachability of a witness and the (violation of) the corresponding refinement relation by means of a small example.

**Example 3.8.** Consider the specification  $s_0$  and the two implementations  $t_0$  and  $u_0$  as presented in Example 2.12 again. In the figure below the (reachable part of the) normal form LTS of  $s_0$  is depicted on the left, the product  $\text{norm}(s_0) \times t_0$  is shown in the middle and the product  $\text{norm}(s_0) \times u_0$  is shown on the right.



We observe that both  $\text{norm}(s_0) \times t_0$  and  $\text{norm}(s_0) \times u_0$  contain no TR-witnesses. In Example 2.12 we had already established that both  $s_0 \sqsubseteq_{\text{tr}} t_0$  and  $s_0 \sqsubseteq_{\text{tr}} u_0$ . For the product  $\text{norm}(s_0) \times t_0$ , the state  $(\{s_0\}, t_2)$  is reachable and an SF-witness since  $\text{stable}(t_2)$  and  $\{10, 20, \text{REQ}\} \in \text{refusals}(t_2)$  but  $\{10, 20, \text{REQ}\} \notin \text{refusals}(\{s_0\})$ . Intuitively, the product encodes that  $\text{REQ } 20$  is a weak trace in both LTSs  $\text{norm}(s_0)$  and  $t_2$  that reaches  $\{s_0\}$  and  $t_2$  respectively. Similarly, the normal form relates this weak trace to the reachability of  $s_0$  from



$s_0$ . Therefore, this witness indicates that  $(\text{REQ } 20, \{10, 20, \text{REQ}\})$  is a failure of  $t_0$ , but not a failure of  $s_0$ , which establishes a violation of stable failures refinement. Finally, we can observe that  $\text{norm}(s_0) \times u_0$  contains no SF-witnesses.  $\square$

The following lemmas formalise that trace refinement can be decided by checking reachability of a TR-witness in the product  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ . Note that this result, and the related result for stable failures refinement, was already established in the literature; for instance, the relation between an SF-witness and the corresponding refinement relation can be found in [Ros94]. However, the definitions in that paper are not explicit and the proof of this correspondence is only sketched. Therefore, we here provide detailed proofs of these results.

**Lemma 3.9.** *Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs. If  $\mathcal{L}_1 \sqsubseteq_{\text{tr}} \mathcal{L}_2$  holds then no TR-witness is reachable in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ .*

*Proof.* Suppose that  $\mathcal{L}_1 \sqsubseteq_{\text{tr}} \mathcal{L}_2$  holds, which means that  $\text{weaktraces}(\mathcal{L}_2) \subseteq \text{weaktraces}(\mathcal{L}_1)$ . Now assume that there is a reachable TR-witness  $(\emptyset, s)$  in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ . We show that this leads to a contradiction. As the pair  $(\emptyset, s)$  is reachable there is a weak trace  $\rho \in \text{Act}^*$  such that  $\iota \xRightarrow{\rho} (\emptyset, s)$  where  $\iota$  is the initial state of  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ . From Proposition 3.2 it follows that  $\iota_2 \xRightarrow{\rho}_2 s$  and  $\emptyset$  is reachable by following  $\rho$  in  $\text{norm}(\mathcal{L}_1)$ . Therefore,  $\rho \in \text{weaktraces}(\mathcal{L}_2)$  and from Lemma 3.6 it follows that  $\rho \notin \text{weaktraces}(\mathcal{L}_1)$ . This contradicts our assumption that  $\text{weaktraces}(\mathcal{L}_2) \subseteq \text{weaktraces}(\mathcal{L}_1)$ . Hence, no TR-witness is reachable in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ .  $\square$

**Lemma 3.10.** *Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs. If no TR-witness is reachable in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$  then  $\mathcal{L}_1 \sqsubseteq_{\text{tr}} \mathcal{L}_2$ .*

*Proof.* Suppose that no TR-witness is reachable in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ . Again, we prove this by contradiction. Assume that  $\mathcal{L}_1 \not\sqsubseteq_{\text{tr}} \mathcal{L}_2$  holds. This means that  $\text{weaktraces}(\mathcal{L}_2) \not\subseteq \text{weaktraces}(\mathcal{L}_1)$ . Pick a weak trace  $\rho \in \text{weaktraces}(\mathcal{L}_2)$  such that  $\rho \notin \text{weaktraces}(\mathcal{L}_1)$ . Then there is a state  $s \in S_2$  for which  $\iota_2 \xRightarrow{\rho}_2 s$ . By Lemma 3.6 it holds that  $\rho$  leads to the empty set in  $\text{norm}(\mathcal{L}_1)$ . By Proposition 3.2 the pair  $(\emptyset, s)$  is then a reachable TR-witness in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ . Contradiction.  $\square$

**Theorem 3.11.** *Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs. Then  $\mathcal{L}_1 \sqsubseteq_{\text{tr}} \mathcal{L}_2$  holds if and only if no TR-witness is reachable in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ .*

*Proof.* This follows directly from Lemmas 3.9 and 3.10.  $\square$

Next, we formalise the relation between stable failures refinement and the reachability of an SF-witness. In the proofs for the next two lemmas, we exploit Theorem 3.11 and the fact that stable failures refinement is stronger than trace refinement.

**Lemma 3.12.** *Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs. If  $\mathcal{L}_1 \sqsubseteq_{\text{sfr}} \mathcal{L}_2$  holds then no SF-witness is reachable in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ .*

*Proof.* Suppose that  $\mathcal{L}_1 \sqsubseteq_{\text{sfr}} \mathcal{L}_2$  holds. Therefore, both  $\text{failures}(\mathcal{L}_2) \subseteq \text{failures}(\mathcal{L}_1)$  and  $\text{weaktraces}(\mathcal{L}_2) \subseteq \text{weaktraces}(\mathcal{L}_1)$ . Now assume that there is a reachable SF-witness  $(U, s)$  in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ . We show that this leads to a contradiction. For  $(U, s)$  to be an SF-witness it holds that  $U = \emptyset$  or both  $\text{stable}(s)$  and  $\text{refusals}(s) \not\subseteq \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ . However, since  $\text{weaktraces}(\mathcal{L}_2) \subseteq \text{weaktraces}(\mathcal{L}_1)$  it follows that  $\mathcal{L}_1 \sqsubseteq_{\text{tr}} \mathcal{L}_2$  and, hence, by Theorem 3.11, no TR-witness is reachable in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ . Consequently,  $U \neq \emptyset$ , and therefore it must be the case that  $\text{stable}(s)$  and  $\text{refusals}(s) \not\subseteq \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$  hold.

As the pair  $(U, s)$  is reachable there is a weak trace  $\rho \in Act^*$  such that  $\iota \xRightarrow{\rho} (U, s)$  where  $\iota$  is the initial state of  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ . From Proposition 3.2 it follows that  $\iota_2 \xRightarrow{\rho}_2 s$  and  $U$  is reachable by following  $\rho$  in  $\text{norm}(\mathcal{L}_1)$ . Since  $\text{stable}(s)$  and  $\iota_2 \xRightarrow{\rho}_2 s$ , it follows that there must be a failure  $(\rho, X) \in \text{failures}(\mathcal{L}_2)$  where  $s$  can stably refuse  $X \in \text{refusals}(s)$ , but  $X \notin \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ . Let  $(\rho, X)$  be such. By Lemmas 2.4 and 3.5 it follows for all states  $t \in S_1$  where  $\iota_1 \xRightarrow{\rho}_1 t$  that  $t \in \llbracket U \rrbracket_{\mathcal{L}_1}$ . For each stable  $t$  it holds that  $X \notin \text{refusals}(t)$ , because  $X \notin \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ . Therefore, we conclude that  $(\rho, X) \notin \text{failures}(\mathcal{L}_1)$ , which contradicts  $\text{failures}(\mathcal{L}_2) \subseteq \text{failures}(\mathcal{L}_1)$ . We conclude that the state  $(U, s)$  cannot be an SF-witness.  $\square$

**Lemma 3.13.** *Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs. If no SF-witness is reachable in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$  then  $\mathcal{L}_1 \sqsubseteq_{\text{sfr}} \mathcal{L}_2$  holds.*

*Proof.* Suppose that no SF-witness is reachable in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ . Towards a contradiction, assume that  $\mathcal{L}_1 \not\sqsubseteq_{\text{sfr}} \mathcal{L}_2$ . By definition of the stable failures refinement this means that  $\text{failures}(\mathcal{L}_2) \not\subseteq \text{failures}(\mathcal{L}_1)$  or  $\text{weaktraces}(\mathcal{L}_2) \not\subseteq \text{weaktraces}(\mathcal{L}_1)$ . If  $\text{weaktraces}(\mathcal{L}_2) \not\subseteq \text{weaktraces}(\mathcal{L}_1)$  then  $\mathcal{L}_1 \not\sqsubseteq_{\text{tr}} \mathcal{L}_2$  and so, by Theorem 3.11, there must be a reachable TR-witness  $(\emptyset, s)$ , and, therefore, also a reachable SF-witness. Contradiction.

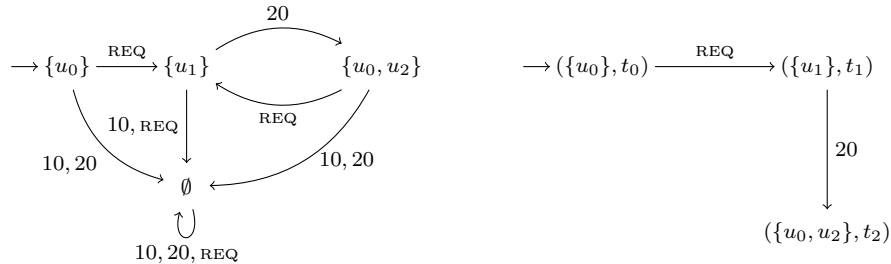
Therefore  $\text{failures}(\mathcal{L}_2) \not\subseteq \text{failures}(\mathcal{L}_1)$  and  $\text{weaktraces}(\mathcal{L}_2) \subseteq \text{weaktraces}(\mathcal{L}_1)$ . Pick a failure  $(\rho, X) \in \text{failures}(\mathcal{L}_2)$  such that  $(\rho, X) \notin \text{failures}(\mathcal{L}_1)$ . Since  $(\rho, X) \in \text{failures}(\mathcal{L}_2)$ , there is a stable state  $s \in S_2$  such that  $\iota_2 \xRightarrow{\rho}_2 s$  and  $X \in \text{refusals}(s)$ . Since  $(\rho, X) \in \text{failures}(\mathcal{L}_2)$ , also  $\rho \in \text{weaktraces}(\mathcal{L}_1)$ , and therefore  $\rho \in \text{weaktraces}(\mathcal{L}_1)$ . By Lemmas 2.4 and 3.5 weak trace  $\rho$  leads to a unique state  $U$  in  $\text{norm}(\mathcal{L}_1)$  such that for all states  $t \in S_1$  with  $\iota_1 \xRightarrow{\rho}_1 t$  it holds that  $t \in \llbracket U \rrbracket_{\mathcal{L}_1}$ . For each stable  $t$  it holds that  $X \notin \text{refusals}(t)$ , because  $(\rho, X) \notin \text{failures}(\mathcal{L}_1)$ . Therefore, by Lemma 3.4 it follows that  $X \notin \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ . Hence,  $\text{refusals}(s) \not\subseteq \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ . By Proposition 3.2 the pair  $(U, s)$  is reachable and it is an SF-witness by definition. Contradiction.  $\square$

**Theorem 3.14.** *Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs. Then  $\mathcal{L}_1 \sqsubseteq_{\text{sfr}} \mathcal{L}_2$  holds if and only if no SF-witness is reachable in  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ .*

*Proof.* This follows directly from Lemmas 3.12 and 3.13.  $\square$

One may be inclined to believe that the normal form LTS can also be used to reduce the failures-divergences refinement decision problem to a reachability problem. This is, however, not the case as the following example illustrates.

**Example 3.15.** Reconsider the LTSs  $t_0$  and  $u_0$  of Example 2.12. Note that  $t_0$  is a correct failures-divergences refinement of  $u_0$ , i.e.,  $u_0 \sqsubseteq_{\text{fdr}} t_0$ . The divergences  $\text{REQ } \rho$ , for  $\rho \in Act^*$ , of  $u_0$  result in specification  $u_0$  permitting all these sequences.



Consider the normal form  $\text{norm}(u_0)$  shown above on the left. The pair  $(\{u_0, u_2\}, t_2)$  in the product  $\text{norm}(u_0) \times t_0$  shown on the right, is thus problematic for the analysis of failures-divergence refinement, as the reachability of this pair might incorrectly indicate a violation

of  $u_0 \sqsubseteq_{\text{fdr}} t_0$ . In turn, this suggests that in the reachability analysis of the product, states beyond those reached via a trace constituting a divergence should not be considered candidate witnesses.  $\square$

Our solution is to modify the construction of the normal form LTS for failures-divergences refinement as follows.

**Definition 3.16** (Failures-divergences normal form). Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS. The failures-divergences normal form of  $\mathcal{L}$  is the LTS  $\text{norm}_{\text{fdr}}(\mathcal{L}) = (S', \iota', \rightarrow')$ , where  $S' = \mathcal{P}(S)$ ,  $\iota' = \{s \in S \mid \iota \xRightarrow{\epsilon} s\}$  and  $\rightarrow'$  is defined as  $U \xrightarrow{a'} V$  if and only if  $\neg(\exists s \in U : s \uparrow)$  and  $V = \{t \in S \mid \exists s \in U : s \xRightarrow{a} t\}$  for all sets of states  $U, V \subseteq S$  and actions  $a \in \text{Act}$ .

Notice that  $\text{norm}_{\text{fdr}}(\mathcal{L})$  yields a subgraph of  $\text{norm}(\mathcal{L})$ . As a result, several properties that we established for  $\text{norm}$  carry over to  $\text{norm}_{\text{fdr}}$ . For instance,  $\text{norm}_{\text{fdr}}$  yields LTSs that are deterministic and concrete. However, contrary to LTSs obtained via  $\text{norm}$ , LTSs obtained via  $\text{norm}_{\text{fdr}}$  are not guaranteed to be universal. In particular, a weak trace  $\rho \in \text{weaktraces}(\mathcal{L})$  is not guaranteed to be preserved in  $\text{norm}_{\text{fdr}}(\mathcal{L})$  if it is a divergence. Consequently, Lemma 3.6, which is essential for Theorems 3.11 and 3.14, no longer holds in its full generality. We show, however, that for failures-divergence refinement a slightly different relation between an LTS and its normal form is sufficient for establishing a theorem that is similar in spirit to the aforementioned theorems.

**Lemma 3.17.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}_{\text{fdr}}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  and states  $U \in S'$  such that  $\iota' \xrightarrow{\rho} U$  it holds that  $\iota \xRightarrow{\rho} s$  for all  $s \in \llbracket U \rrbracket_{\mathcal{L}}$ .*

*Proof.* Along the same lines as the proof of Lemma 3.4.  $\square$

We mentioned that divergences are not necessarily preserved (as traces) by the normalisation. In fact, we can be more specific: only *minimal* divergences are preserved in the normal form LTS. The *minimal* divergences of a state  $s \in S$ , denoted by  $\text{divergences}_{\min}(s)$ , is the largest subset of  $\text{divergences}(s)$  containing all  $\rho \in \text{divergences}(s)$  for which there is no strict prefix of  $\rho$  in  $\text{divergences}(s)$ . For an LTS  $\mathcal{L} = (S, \iota, \rightarrow)$  we define  $\text{divergences}_{\min}(\mathcal{L})$  to be  $\text{divergences}_{\min}(\iota)$ .

**Lemma 3.18.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}_{\text{fdr}}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  such that either  $\rho \notin \text{divergences}(\mathcal{L})$  or  $\rho \in \text{divergences}_{\min}(\mathcal{L})$  and for all states  $s \in S$  such that  $\iota \xRightarrow{\rho} s$  there is a state  $U \in S'$  such that  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  and  $\iota' \xrightarrow{\rho} U$ .*

**Lemma 3.19.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}_{\text{fdr}}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  and states  $U \in S'$  it holds that if  $\iota' \xrightarrow{\rho} U$  and not  $\llbracket U \rrbracket_{\mathcal{L}} \uparrow$  then  $\rho \notin \text{divergences}(\mathcal{L})$ .*

**Lemma 3.20.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}_{\text{fdr}}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  it holds that  $\rho \notin (\text{divergences}(\mathcal{L}) \cup \text{weaktraces}(\mathcal{L}))$  if and only if  $\iota' \xrightarrow{\rho} \emptyset$ .*

For failures-divergences refinement the state space of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  is explored for a witness, where reachability of such a witness also indicates non-refinement. This witness is defined as follows:

**Definition 3.21** (Failures-divergences witness). Let  $\mathcal{L}_1$  and  $\mathcal{L}_2$  be two LTSs. A state  $(U, s)$  of the product  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  is called an *FD-witness* if and only if  $\llbracket U \rrbracket_{\mathcal{L}_1} \uparrow$  does not hold and at least one of the following conditions hold:

- $U = \emptyset$ .
- $\text{stable}(s)$  and  $\text{refusals}(s) \not\subseteq \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ .
- $s \uparrow$ .

We next formalise the correspondence between failures-divergences refinement and the reachability of an FDR-witness. In the proof of the following theorem we cannot easily use Theorem 3.11 because  $\sqsubseteq_{\text{fdr}}$  is incomparable with both  $\sqsubseteq_{\text{tr}}$  and  $\sqsubseteq_{\text{sfr}}$ .

**Lemma 3.22.** *Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs. If  $\mathcal{L}_1 \sqsubseteq_{\text{fdr}} \mathcal{L}_2$  holds then no FD-witness is reachable in  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$ .*

*Proof.* Assume that  $\mathcal{L}_1 \sqsubseteq_{\text{fdr}} \mathcal{L}_2$ . We then have that  $\text{failures}_{\perp}(\mathcal{L}_2) \subseteq \text{failures}_{\perp}(\mathcal{L}_1)$  and  $\text{divergences}(\mathcal{L}_2) \subseteq \text{divergences}(\mathcal{L}_1)$ . Towards a contradiction, assume there is an FD-witness  $(U, s)$  in  $\text{reachable}(\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2)$ . Let  $\iota$  be the initial state of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$ , and let  $\rho \in \text{weaktraces}(\iota)$  be such that  $\iota \xRightarrow{\rho} (U, s)$ . From the assumption that  $(U, s)$  is an FD-witness it follows that  $\text{not } \llbracket U \rrbracket_{\mathcal{L}_1} \uparrow$  and from Lemma 3.19 it follows that  $\rho \notin \text{divergences}(\mathcal{L}_1)$ . By Proposition 3.2 it holds that  $\iota_2 \xRightarrow{\rho}_2 s$  and  $U$  is reachable by following  $\rho$  in  $\text{norm}_{\text{fdr}}(\mathcal{L}_1)$ . Moreover, for  $(U, s)$  to be an FD-witness, at least one of the following must also hold:  $U = \emptyset$ ,  $\text{stable}(s)$  and  $\text{refusals}(s) \not\subseteq \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ , or  $s \uparrow$ . We therefore distinguish these three cases:

- Case  $U = \emptyset$ . We can assume that  $s \uparrow$  does not hold, as this is handled by another case. Then it follows that there is a state  $t \in S_2$  such that  $\iota_2 \xRightarrow{\rho}_2 s \xRightarrow{\epsilon}_2 t$  and  $\text{stable}(t)$ . Let  $t$  be such. Consequently,  $(\rho, X) \in \text{failures}_{\perp}(\mathcal{L}_2)$  for some  $X \in \text{refusals}(t)$ . By Lemma 3.20 it holds that the weak trace  $\rho$  reaching the empty set in  $\text{norm}_{\text{fdr}}(\mathcal{L}_1)$  is not a weak trace of  $\mathcal{L}_1$ . Together with  $\rho \notin \text{divergences}(\mathcal{L}_1)$  it follows, for all possible refusal sets  $Y \subseteq \text{Act}$ , that  $(\rho, Y) \notin \text{failures}_{\perp}(\mathcal{L}_1)$ , and so, in particular,  $(\rho, X) \in \text{failures}_{\perp}(\mathcal{L}_1)$  which contradicts our assumption that  $\text{failures}_{\perp}(\mathcal{L}_2) \subseteq \text{failures}_{\perp}(\mathcal{L}_1)$ .
- Case  $\text{stable}(s)$  and  $\text{refusals}(s) \not\subseteq \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ . From  $\text{stable}(s)$  and the reachability of state  $s$  it follows that  $\text{failures}_{\perp}(\mathcal{L}_2)$  is not empty. Pick a failure  $(\rho, X) \in \text{failures}_{\perp}(\mathcal{L}_2)$  where  $s$  can stably refuse  $X \in \text{refusals}(s)$ , but  $X \notin \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ . By Lemmas 2.4 and 3.18 it follows for all states  $t \in S_1$  where  $\iota_1 \xRightarrow{\rho}_1 t$  that  $t \in \llbracket U \rrbracket_{\mathcal{L}_1}$ . For each stable  $t$  it holds that  $X \notin \text{refusals}(t)$ , because  $X \notin \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ . Due to the previous case, we may assume that  $U \neq \emptyset$ . Then from  $\rho \notin \text{divergences}(\mathcal{L}_1)$  and  $U \neq \emptyset$  it follows that  $(\rho, X) \notin \text{failures}_{\perp}(\mathcal{L}_1)$ , which leads to a contradiction with the assumption that  $\text{failures}_{\perp}(\mathcal{L}_2) \subseteq \text{failures}_{\perp}(\mathcal{L}_1)$ .
- Case  $s \uparrow$ . Since  $\iota_2 \xRightarrow{\rho}_2 s$  and  $s \uparrow$ , also  $\rho \in \text{divergences}(\mathcal{L}_2)$ . However, by  $\rho \notin \text{divergences}(\mathcal{L}_1)$  this contradicts the assumption that  $\text{divergences}(\mathcal{L}_2) \subseteq \text{divergences}(\mathcal{L}_1)$ .  $\square$

**Lemma 3.23.** *Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs. If no FD-witness is reachable in  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  then  $\mathcal{L}_1 \sqsubseteq_{\text{fdr}} \mathcal{L}_2$  holds.*

*Proof.* Assume that no FD-witness is reachable in  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$ . Again, we prove this by contradiction. Assume that  $\mathcal{L}_1 \not\sqsubseteq_{\text{fdr}} \mathcal{L}_2$ . By definition of failures-divergences refinement this means that  $\text{failures}_{\perp}(\mathcal{L}_2) \not\subseteq \text{failures}_{\perp}(\mathcal{L}_1)$  or  $\text{divergences}(\mathcal{L}_2) \not\subseteq \text{divergences}(\mathcal{L}_1)$ . Hence, there are two cases to consider:

- Case  $\text{divergences}(\mathcal{L}_2) \not\subseteq \text{divergences}(\mathcal{L}_1)$ . Pick a diverging weak trace  $\rho \in \text{divergences}(\mathcal{L}_2)$  such that  $\rho \notin \text{divergences}(\mathcal{L}_1)$ . In this case there is a prefix of  $\rho$ , which we call  $\sigma$ , that leads to a diverging state  $s \in S_2$  such that  $\iota_2 \xRightarrow{\sigma}_2 s$ . However, by the assumption that  $\rho \notin \text{divergences}(\mathcal{L}_1)$  we know that all states  $t \in S_1$  reached by following  $\sigma$  are not diverging.

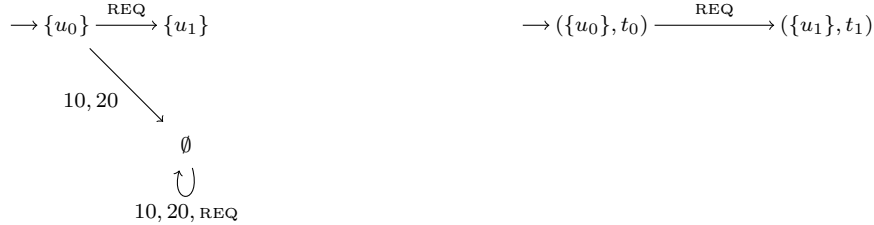
By Lemma 3.17 we know that all  $t \in U$  can be reached by following  $\sigma$ . Therefore state pair  $(U, s)$  is an FD-witness, because  $s \uparrow$  but not  $\llbracket U \rrbracket_{\mathcal{L}_1} \uparrow$ . Contradiction.

- Case  $\text{failures}_\perp(\mathcal{L}_2) \not\subseteq \text{failures}_\perp(\mathcal{L}_1)$ . By the previous case, we may, moreover, assume that  $\text{divergences}(\mathcal{L}_2) \subseteq \text{divergences}(\mathcal{L}_1)$ . Pick any failure  $(\rho, X) \in \text{failures}_\perp(\mathcal{L}_2)$  such that  $(\rho, X) \notin \text{failures}_\perp(\mathcal{L}_1)$ . Observe that  $\rho \notin \text{divergences}(\mathcal{L}_1)$  (and as such  $\rho \notin \text{divergences}(\mathcal{L}_2)$ ) as otherwise no such  $(\rho, X)$  exists by definition of  $\text{failures}_\perp(\mathcal{L}_1)$ . Since  $(\rho, X) \in \text{failures}_\perp(\mathcal{L}_2)$ , there is a stable state  $s \in S_2$  such that  $\iota_2 \xRightarrow{\rho}_2 s$  and  $X \in \text{refusals}(s)$ . We distinguish whether weak trace  $\rho$  is among the weak traces of  $\mathcal{L}_1$  or not:
  - Case  $\rho \notin \text{weaktraces}(\mathcal{L}_1)$ . By Lemma 3.20 this means that  $\rho$  is a trace leading to the empty set in  $\text{norm}_{\text{fdr}}(\mathcal{L}_1)$ . By Proposition 3.2, a pair  $(\emptyset, s)$  is then reachable in  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$ . But then that pair is a reachable FD-witness. Contradiction.
  - Case  $\rho \in \text{weaktraces}(\mathcal{L}_1)$ . Recall that  $(\rho, X) \notin \text{failures}_\perp(\mathcal{L}_1)$  by assumption. By Lemmas 2.4 and 3.18 there is a unique state  $V$  of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1)$  reachable via weak trace  $\rho$  such that for all  $t \in S_1$  where  $\iota_1 \xRightarrow{\rho}_1 t$ , it holds that  $t \in \llbracket V \rrbracket_{\mathcal{L}_1}$ . For each stable state  $t$  it holds that  $X \notin \text{refusals}(t)$ , because  $(\rho, X) \in \text{failures}_\perp(\mathcal{L}_1)$ . Therefore, by Lemma 3.17 it follows that  $X \notin \text{refusals}(\llbracket V \rrbracket_{\mathcal{L}_1})$ . Therefore  $\text{refusals}(s) \not\subseteq \text{refusals}(\llbracket V \rrbracket_{\mathcal{L}_1})$ . By Proposition 3.2 the pair  $(V, s)$  is reachable and it is an FD-witness by definition. Contradiction.  $\square$

**Theorem 3.24.** *Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs. Then  $\mathcal{L}_1 \sqsubseteq_{\text{fdr}} \mathcal{L}_2$  holds if and only if no FD-witness is reachable in  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$ .*

*Proof.* This follows directly from Lemmas 3.22 and 3.23.  $\square$

**Example 3.25.** Consider the LTSs  $t_0$  and  $u_0$  of Example 2.12 once more. As we noted in Example 3.15,  $t_0$  is a correct failures-divergences refinement of  $u_0$ . In the figure below the normal form  $\text{norm}_{\text{fdr}}(u_0)$  is shown on the left and the product  $\text{norm}_{\text{fdr}}(u_0) \times t_0$  is shown on the right.



Note that the pair  $(\{u_0, u_2\}, t_2)$ , which was reachable in the product  $\text{norm}(u_0) \times t_0$ , is no longer reachable in the product  $\text{norm}_{\text{fdr}}(u_0) \times t_0$ . In fact, no FD-witness is reachable in the latter product, thus confirming that indeed  $u_0 \sqsubseteq_{\text{fdr}} t_0$ .  $\square$

#### 4. ANTICHAIN ALGORITHMS FOR REFINEMENT CHECKING

Notice that Theorems 3.11, 3.14 and 3.24 provide the basis for straightforward algorithms for deciding trace refinement, stable failures refinement and failures-divergences refinement: one can explore the product of the normalised specification and the implementation, looking for a witness *on-the-fly*. In these algorithms, the normalisation of the specification LTS dominates the theoretical worst-case run time complexity of the algorithms. While refinement checking itself is a PSPACE-hard problem, in practice, the problem can often be solved quite effectively. Nevertheless, as observed in [WSS<sup>+</sup>12], antichains provide room for improvement

by potentially reducing the number of states of the normal form LTS of the specification that must be checked.

An antichain is a set  $\mathcal{A} \subseteq X$  of a partially ordered set  $(X, \leq)$  in which all distinct  $x, y \in \mathcal{A}$  are incomparable: neither  $x \leq y$  nor  $y \leq x$ . Given a partially ordered set  $(X, \leq)$  and an antichain  $\mathcal{A}$ , the membership test, denoted by  $\in$ , checks whether  $\mathcal{A}$  ‘contains’ an element  $x$ ; that is,  $x \in \mathcal{A}$  holds true if and only if there is some  $y \in \mathcal{A}$  such that  $y \leq x$ . We write  $Y \in^\forall \mathcal{A}$  iff  $y \in \mathcal{A}$  for all  $y \in Y$ . Antichain  $\mathcal{A}$  can be extended by inserting an element  $x \in X$ , denoted  $\mathcal{A} \uplus x$ , which is defined as the set  $\{y \mid y = x \vee (y \in \mathcal{A} \wedge x \not\leq y)\}$ . Note that this operation only yields an antichain whenever  $x \notin \mathcal{A}$ .

As [WSS<sup>+</sup>12, ACH<sup>+</sup>10] suggest, the state space of the product  $(S, \iota, \rightarrow)$  between a normal form of LTS  $\mathcal{L}_1$  and the LTS  $\mathcal{L}_2$  induces a partially ordered set as follows. For  $(U, s), (V, t) \in S$ , define  $(U, s) \leq (V, t)$  iff  $s = t$  and  $\llbracket U \rrbracket_{\mathcal{L}_1} \subseteq \llbracket V \rrbracket_{\mathcal{L}_1}$ . Then the set  $(S, \leq)$  is a partially ordered set. The fundamental property underlying the reason why an antichain approach to refinement checking works is expressed by the following claim (which we repeat as Proposition 5.8, and prove in Section 5), stating that the traces of any state  $(V, s)$  in the product can be executed from all states smaller than  $(V, s)$ . Notice that this property relies on the fact that the empty set is included as a state in the normal form LTS.

**Claim 4.1.** *For all states  $(U, s), (V, s)$  of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  satisfying  $(U, s) \leq (V, s)$  and for every sequence  $\sigma \in \text{Act}_\tau^*$  such that  $(V, s) \xrightarrow{\sigma} (V', t)$  there is a state  $(U', t)$  such that  $(U, s) \xrightarrow{\sigma} (U', t)$  and  $(U', t) \leq (V', t)$ .*

The main idea of the antichain algorithms is now as follows: the set of states of the product that have been explored are recorded in an antichain rather than a set. Whenever a new state of the product is found that is already included in the antichain (w.r.t. the membership test  $\in$ ), further exploration of that state is unnecessary, thereby pruning the state space of the product. While the proposition stated above suggests this is sound for trace refinement, it is not immediate that doing so is also sound for refusals and divergences.

Based on the above informal reasoning, [WSS<sup>+</sup>12] presents antichain algorithms that intend to check for trace refinement, stable failures refinement and failures-divergences refinement. Before we discuss these algorithms in more detail, see Algorithms 1-3, we here present their pseudocode for the sake of completeness; in the remainder of this paper, we refer to these as the *original* algorithms.

**Remark 4.2.** For the implementation of refusals in Algorithms 2 and 3 we followed the definition of **refusals** provided in [WSS<sup>+</sup>12]. This definition differs subtly from Definition 2.5, by defining, for any (not necessarily stable) state  $s$ ,  $\text{refusals}(s) = \{X \mid \exists s' \in S : (s \xrightarrow{\epsilon} s' \wedge \text{stable}(s') \wedge X \subseteq \text{Act} \setminus \text{enabled}(s'))\}$  and  $\text{refusals}(U) = \{X \mid \exists s \in U : X \in \text{refusals}(s)\}$  for  $U \subseteq S$ .

---

**Algorithm 1** Antichain-based trace refinement algorithm presented in [WSS<sup>+</sup>12]. The algorithm returns *true* if and only if  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  is refined by  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  in trace semantics.

---

```

1: procedure REFINES-TRACE( $\mathcal{L}_1, \mathcal{L}_2$ )
2:   let working be a stack containing a pair  $(\{s \in S_1 \mid \iota_1 \xRightarrow{\epsilon}_1 s\}, \iota_2)$ 
3:   let antichain :=  $\emptyset$ 
4:   while working  $\neq \emptyset$  do
5:     pop (spec, impl) from working
6:     antichain := antichain  $\uplus$  (spec, impl)
7:     for impl  $\xrightarrow{a}_2$  impl' do
8:       if  $a = \tau$  then
9:         spec' := spec
10:      else
11:        spec' :=  $\{s' \in S_1 \mid \exists s \in \text{spec} : s \xRightarrow{a}_1 s'\}$ 
12:      if spec' =  $\emptyset$  then
13:        return false
14:      if (spec', impl')  $\notin$  antichain then
15:        push (spec', impl') into working
16:   return true

```

---



---

**Algorithm 2** Antichain-based stable failures refinement algorithm presented in [WSS<sup>+</sup>12]. The algorithm returns *true* if and only if  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  is refined by  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  in stable failures semantics.

---

```

1: procedure REFINES-STABLE-FAILURES( $\mathcal{L}_1, \mathcal{L}_2$ )
2:   let working be a stack containing a pair  $(\{s \in S_1 \mid \iota_1 \xRightarrow{\epsilon}_1 s\}, \iota_2)$ 
3:   let antichain :=  $\emptyset$ 
4:   while working  $\neq \emptyset$  do
5:     pop (spec, impl) from working
6:     antichain := antichain  $\uplus$  (spec, impl)
7:     if refusals(impl)  $\not\subseteq$  refusals(spec) then
8:       return false
9:     for impl  $\xrightarrow{a}_2$  impl' do
10:      if  $a = \tau$  then
11:        spec' := spec
12:      else
13:        spec' :=  $\{s' \in S_1 \mid \exists s \in \text{spec} : s \xRightarrow{a}_1 s'\}$ 
14:      if spec' =  $\emptyset$  then
15:        return false
16:      if (spec', impl')  $\notin$  antichain then
17:        push (spec', impl') into working
18:   return true

```

---

---

**Algorithm 3** Antichain-based failures-divergences refinement algorithm presented in [WSS<sup>+</sup>12]. The algorithm is claimed to return *true* iff  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  is refined by  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  in failures-divergences semantics.

---

```

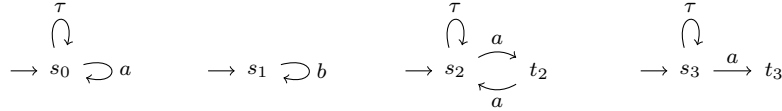
1: procedure REFINES-FAILURES-DIVERGENCES( $\mathcal{L}_1, \mathcal{L}_2$ )
2:   let working be a stack containing a pair  $(\{s \in S_1 \mid \iota_1 \xRightarrow{\epsilon}_1 s\}, \iota_2)$ 
3:   let antichain :=  $\emptyset$ 
4:   while working  $\neq \emptyset$  do
5:     pop (spec, impl) from working
6:     antichain := antichain  $\uplus$  (spec, impl)
7:     if impl  $\uparrow$  then
8:       if not spec  $\uparrow$  then
9:         return false
10:    else
11:      if refusals(impl)  $\not\subseteq$  refusals(spec) then
12:        return false
13:      for impl  $\xrightarrow{a}_2$  impl' do
14:        if  $a = \tau$  then
15:          spec' := spec
16:        else
17:          spec' :=  $\{s' \in S_1 \mid \exists s \in \text{spec} : s \xRightarrow{a}_1 s'\}$ 
18:        if spec' =  $\emptyset$  then
19:          return false
20:        if (spec', impl')  $\notin$  antichain then
21:          push (spec', impl') into working
22:  return true

```

---

Let us stress that Algorithm 1 correctly decides trace refinement and Algorithm 2 correctly decides stable failures refinement. However, Algorithm 3 fails to correctly decide failures-divergences refinement. Moreover, it is interesting to note that Algorithms 2 and 3 fail to decide the *non-standard* relations used in [WSS<sup>+</sup>12], see also the discussion in Remark 2.11. All three issues are illustrated by the example below.

**Example 4.3.** Consider the four transition systems depicted below.



Let us first observe that Algorithm 2 correctly decides that  $s_1 \sqsubseteq_{\text{sfr}} s_0$  does not hold, which follows from a violation of  $\text{weaktraces}(s_0) \subseteq \text{weaktraces}(s_1)$ . Next, observe that we have  $s_0 \sqsubseteq_{\text{fdr}} s_1$ , since the divergence of the root state  $s_0$  implies chaotic behaviour of  $s_0$  and, hence, any system refines such a system. It is not hard to see, however, that Algorithm 3 returns *false*, wrongly concluding that  $s_0 \not\sqsubseteq_{\text{fdr}} s_1$ .

With respect to the non-standard refinement relations defined in [WSS<sup>+</sup>12], see also Remark 2.11, we observe the following. Since  $s_0$  is not stable, we have  $\text{failures}(s_0) = \emptyset$  and hence  $\text{failures}(s_0) \subseteq \text{failures}(s_1)$ . Consequently, stable failures refinement as defined in [WSS<sup>+</sup>12] should hold, but as we already concluded above, the algorithm returns *false* when checking for  $s_1 \sqsubseteq_{\text{sfr}} s_0$ . Next, observe that the algorithm returns *true* when checking for  $s_2 \sqsubseteq_{\text{fdr}} s_3$ . 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 return *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 [WSS<sup>+</sup>12], however, the algorithm should return *false*, since  $\text{failures}(s_3) \subseteq \text{failures}(s_2)$  fails to hold: we have  $(a, \{a\}) \in \text{failures}(s_3)$  but not  $(a, \{a\}) \in \text{failures}(s_2)$ .  $\square$

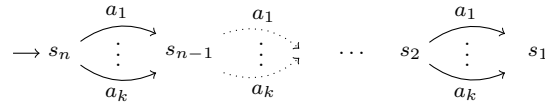
Notice that each algorithm explores the product between the normal form of a specification, and an implementation in a depth-first, on-the-fly manner. While depth-first search is typically used for detecting divergences, [Ros94] states a number of reasons for running a refinement check in a breadth-first manner. Indeed, a compelling argument in favour of using a breadth-first search is conciseness of the counterexample in case of a non-refinement.

Each algorithm can be made to run in a breadth-first fashion simply by using a FIFO *queue* rather than a stack as the data structure for *working*. However, our implementations of these algorithms suffer from severely degraded performance. The performance degradation can be traced back to the following three additional problems in the original algorithms, which also are present (albeit less pronounced in practice) when utilising a depth-first exploration:

- (1) The refusal check on line 7 of Algorithm 2 (and line 11 of Algorithm 3) is also performed for unstable states, which, combined with the definition of *refusals* in [WSS<sup>+</sup>12] (see also Remark 4.2), results in a repeated, potentially expensive, search for stable states;
- (2) In all three algorithms, duplicate pairs might be added to *working* since *working* is filled with all successors of  $(\text{spec}, \text{impl})$  that fail the *antichain* membership test, regardless of whether these pairs are already scheduled for exploration, *i.e.*, included in *working*, or not;
- (3) In all three algorithms, contrary to the explicit claim in [WSS<sup>+</sup>12, Section 2.2] the variable *antichain* is not guaranteed to be an antichain.

The first problem is readily seen to lead to undesirable overhead. The second and third problem are more subtle. We first illustrate the second problem on Algorithm 1: the following example shows a case where the algorithm stores an excessive number of pairs in *working*. Note that the two other algorithms suffer from the same phenomenon.

**Example 4.4.** Consider the family of LTSs  $\mathcal{L}_n^k = (S_n, \iota_n, \rightarrow_n)$  with states  $S_n = \{s_1, \dots, s_n\}$ , transitions  $s_i \xrightarrow{a_j}_n s_{i-1}$  for all  $1 \leq j \leq k$ ,  $1 < i \leq n$  and  $\iota_n = s_n$ ; see also the transition system depicted below. Note that each LTS that belongs to this family is completely deterministic and concrete.



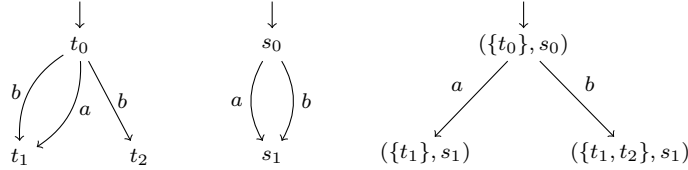
Each labelled transition system in this class has  $n$  states and  $k \cdot (n - 1)$  transitions. Suppose one checks for trace refinement between an implementation and specification both of which are given by  $\mathcal{L}_n^k$ ; *i.e.*, we test for  $\mathcal{L}_n^k \sqsubseteq_{\text{tr}} \mathcal{L}_n^k$ .

Using a depth-first search, Algorithm 1 will add the state reachable via a single step once for every action, 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. Hence, *working* contains exactly  $i \cdot (k - 1) + 1$  pairs at the end of the  $i$ -th iteration, resulting in a maximum *working* stack size of  $\mathcal{O}(n \cdot k)$  entries. At the end of the  $n$ -th iteration *antichain* contains all reachable pairs of the product, *i.e.*, *antichain* is equal to  $(\{s_i\}, s_i)$  for all  $1 \leq i \leq n$ . Emptying *working* after the  $n$ -th iteration involves  $k$  antichain membership tests per entry. Consequently,  $\mathcal{O}(n \cdot k^2)$  antichain membership tests are required to check  $\mathcal{L}_n^k \sqsubseteq_{\text{tr}} \mathcal{L}_n^k$ .

The breadth-first variant of Algorithm 1 also adds the state reachable via a single step once for every action for the same reason as the depth-first variant. 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 FIFO queue *working*. Therefore each entry in *working* adds  $k$  elements before it is added to *antichain*, resulting in a maximum queue size of  $\mathcal{O}(k^n)$  at state  $(\{s_1\}, s_1)$ . Emptying *working* results in  $\mathcal{O}(k^{n+1})$  antichain membership tests.  $\square$

Finally, the example below illustrates the third problem of the algorithms, *viz.*, the violation of the antichain property. We again illustrate the problem on the most basic of all three algorithms, *viz.*, Algorithm 1. Note that this violation does not influence the result of antichain membership tests, but it can have an effect on the size of the antichain which in turn leads to overhead.

**Example 4.5.** Consider the two left-most labelled transition systems depicted below, along with the (normal form) product (the LTS on the right).



Algorithm 1 starts with *working* containing pair  $(\{t_0\}, s_0)$  and *antichain* =  $\emptyset$ . Inside the loop, the pair  $(\{t_0\}, s_0)$  is popped from *working* and added to *antichain*. The successors of the pair  $(\{t_0\}, s_0)$  are the pairs  $(\{t_1\}, s_1)$  and  $(\{t_1, t_2\}, s_1)$ . Since *antichain* contains neither of these, both successors are added to *working* in line 15. Next, popping  $(\{t_1\}, s_1)$  from *working* and adding this pair to *antichain* results in *antichain* consisting of the set  $\{(\{t_0\}, s_0), (\{t_1\}, s_1)\}$ . In the final iteration of the algorithm, the pair  $(\{t_1, t_2\}, s_1)$  is popped from *working* and added to *antichain*, resulting in the set  $\{(\{t_0\}, s_0), (\{t_1\}, s_1), (\{t_1, t_2\}, s_1)\}$ . Clearly, since  $(\{t_1\}, s_1) \leq (\{t_1, t_2\}, s_1)$ , the set *antichain* no longer is a proper antichain.  $\square$

## 5. CORRECT AND IMPROVED ANTICHAIN ALGORITHMS

We first focus on solving the performance problems of Algorithms 1 and 2. Subsequently, we discuss the additional modifications that are required for Algorithm 3 to correctly decide failures-divergences refinement.

The first performance problem that we identified, *viz.*, the computational overhead induced by checking for refusal inclusion in non-stable states (which does not occur when checking for a TR-witness), can be solved in a rather straightforward manner: we only perform the check to compare the refusals of the implementation and the normal form state of the specification in case the implementation state is stable. Doing so avoids a potentially expensive search for stable states.

The second and third performance problems we identified can be solved by rearranging the computations that are conducted; these modifications are more involved. The essential observation here is that in order for the information in *antichain* to be most effective, states of the product must be added to *antichain* as soon as these are discovered, even if these have not yet been fully explored. This is achieved by maintaining, as an invariant, that  $\text{working} \subseteq^\forall \text{antichain}$  holds true; the states in *working* then, intuitively, constitute the

*frontier* of the exploration. We achieve this by initialising *working* and *antichain* to consist of exactly the initial state of the product, and by extending *antichain* with all (not already discovered) successors for the state  $(spec, impl)$  that is popped from *working*. As a side effect, this also resolves the third issue, as now both *working* and *antichain* are only extended with states that have not yet been discovered, *i.e.*, for which the membership test in *antichain* fails, and for which insertion of such states does not invalidate the antichain property.

The modifications we discussed above yield improved algorithms for deciding trace refinement and stable failures refinement, see the pseudocode of Algorithms 4 and 5. We postpone the discussion of their correctness until after discussing the modifications required to Algorithm 3 and its proof of correctness. The example we present below illustrates the impact of our changes.

---

**Algorithm 4** The improved trace refinement checking algorithm. The algorithm returns *true* iff  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  is refined by  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  in trace semantics.

---

```

1: procedure REFINES-TRACENEW( $\mathcal{L}_1, \mathcal{L}_2$ )
2:   let working be a stack containing a pair  $(\{s \in S_1 \mid \iota_1 \xrightarrow{\epsilon}_1 s\}, \iota_2)$ 
3:   let antichain :=  $\emptyset \uplus (\{s \in S_1 \mid \iota_1 \xrightarrow{\epsilon}_1 s\}, \iota_2)$ 
4:   while working  $\neq \emptyset$  do
5:     pop  $(spec, impl)$  from working
6:     for  $impl \xrightarrow{a}_2 impl'$  do
7:       if  $a = \tau$  then
8:          $spec' := spec$ 
9:       else
10:         $spec' := \{s' \in S_1 \mid \exists s \in spec : s \xrightarrow{a}_1 s'\}$ 
11:       if  $spec' = \emptyset$  then
12:         return false
13:       if  $(spec', impl') \notin antichain$  then
14:          $antichain := antichain \uplus (spec', impl')$ 
15:         push  $(spec', impl')$  into working
16:   return true

```

---

**Example 5.1.** Consider Example 4.4 again, but now using Algorithm 4 to check for trace refinement. The depth-first variant of this algorithm only adds a successor state to the *working* stack once, because for every other outgoing transition it will already be 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* membership is tested once, resulting in  $\mathcal{O}(n \cdot k)$  *antichain* membership tests. This is an improvement compared to the depth-first variant of Algorithm 4 of a factor  $n \cdot k$  in the maximum *working* stack size and a factor  $k$  in the number of *antichain* membership tests. The bounds for the breadth-first variant are identical to the bounds for the depth-first variant, *i.e.*, maximum  $\mathcal{O}(1)$  *working* queue size and  $\mathcal{O}(n \cdot k)$  number of *antichain* membership tests. Compared to the breadth-first variant of Algorithm 1, this is an improvement of a factor  $k^n$  in the *working* queue size and a factor  $k^n/n$  in the number of *antichain* membership tests.  $\square$

We next focus on the soundness problem of Algorithm 3. The source of the incorrectness of this algorithm can be traced back to the fact that it (partially) explores the state space of  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ , rather than  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$ . As illustrated by Example 3.15, this causes the algorithm to consider states in the product that should not be considered, thus potentially arriving at a wrong verdict. The fix to this problem is simple yet subtle, requiring a swap

---

**Algorithm 5** The improved stable failures refinement checking algorithm. The algorithm returns *true* iff  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  is refined by  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  in stable failures semantics.

---

```

1: procedure REFINES-STABLE-FAILURESNEW( $\mathcal{L}_1, \mathcal{L}_2$ )
2:   let working be a stack containing a pair  $(\{s \in S_1 \mid \iota_1 \xRightarrow{\epsilon}_1 s\}, \iota_2)$ 
3:   let antichain :=  $\emptyset \uplus (\{s \in S_1 \mid \iota_1 \xRightarrow{\epsilon}_1 s\}, \iota_2)$ 
4:   while working  $\neq \emptyset$  do
5:     pop (spec, impl) from working
6:     if  $\text{stable}(\text{impl}) \wedge \text{refusals}(\text{impl}) \not\subseteq \text{refusals}(\text{spec})$  then
7:       return false
8:     for  $\text{impl} \xrightarrow{a}_2 \text{impl}'$  do
9:       if  $a = \tau$  then
10:        spec' := spec
11:       else
12:        spec' :=  $\{s' \in S_1 \mid \exists s \in \text{spec} : s \xRightarrow{a}_1 s'\}$ 
13:       if spec' =  $\emptyset$  then
14:        return false
15:       if  $(\text{spec}', \text{impl}') \notin \text{antichain}$  then
16:        antichain := antichain  $\uplus (\text{spec}', \text{impl}')$ 
17:        push (spec', impl') into working
18:   return true

```

---



---

**Algorithm 6** The corrected failures-divergences refinement checking algorithm. The algorithm returns *true* iff  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  is refined by  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  in failures-divergences semantics.

---

```

1: procedure REFINES-FAILURES-DIVERGENCESNEW( $\mathcal{L}_1, \mathcal{L}_2$ )
2:   let working be a stack containing a pair  $(\{s \in S_1 \mid \iota_1 \xRightarrow{\epsilon}_1 s\}, \iota_2)$ 
3:   let antichain :=  $\emptyset \uplus (\{s \in S_1 \mid \iota_1 \xRightarrow{\epsilon}_1 s\}, \iota_2)$ 
4:   { done :=  $\emptyset$  }
5:   while working  $\neq \emptyset$  do
6:     pop (spec, impl) from working
7:     if not spec↑ then
8:       if impl↑ then
9:        return false
10:    else
11:      if  $\text{stable}(\text{impl}) \wedge \text{refusals}(\text{impl}) \not\subseteq \text{refusals}(\text{spec})$  then
12:        return false
13:      for  $\text{impl} \xrightarrow{a}_2 \text{impl}'$  do
14:        if  $a = \tau$  then
15:          spec' := spec
16:        else
17:          spec' :=  $\{s' \in S_1 \mid \exists s \in \text{spec} : s \xRightarrow{a}_1 s'\}$ 
18:        if spec' =  $\emptyset$  then
19:          return false
20:        if  $(\text{spec}', \text{impl}') \notin \text{antichain}$  then
21:          antichain := antichain  $\uplus (\text{spec}', \text{impl}')$ 
22:          push (spec', impl') into working
23:    { done :=  $\{(\text{spec}, \text{impl})\} \cup \text{done}$  }
24:   return true

```

---

of the divergence tests on lines 7 and 8, and making the further exploration of the state (*spec*, *impl*) conditional on the specification not diverging.

As all three algorithms presented in this section fundamentally differ (some even in the relations that they compute) from the original ones, we cannot reuse arguments for the proof of correctness presented in [WSS<sup>+</sup>12], which are based on invariants that do not hold in our case, and which rely on definitions, some of which are incomparable to ours. The correctness of our improved algorithms is claimed by the following theorem, which we repeat at the end of this section with an explicit proof.

**Theorem 5.2.** *Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs.*

- *REFINES-TRACE<sub>NEW</sub>( $\mathcal{L}_1, \mathcal{L}_2$ ) returns true if and only if  $\mathcal{L}_1 \sqsubseteq_{\text{tr}} \mathcal{L}_2$ .*
- *REFINES-STABLE-FAILURES<sub>NEW</sub>( $\mathcal{L}_1, \mathcal{L}_2$ ) returns true if and only if  $\mathcal{L}_1 \sqsubseteq_{\text{sfr}} \mathcal{L}_2$ .*
- *REFINES-FAILURES-DIVERGENCES<sub>NEW</sub>( $\mathcal{L}_1, \mathcal{L}_2$ ) returns true if and only if  $\mathcal{L}_1 \sqsubseteq_{\text{fdr}} \mathcal{L}_2$ .*

For the remainder of this section we fix two LTSs  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$ . We focus on the proof of correctness of Algorithm 6; the correctness proofs for Algorithm 4 for deciding trace refinement and Algorithm 5 for deciding stable failures refinement proceed along the same lines.

First we show termination of Algorithm 6. To reason about the states that have been processed, we have introduced a ghost variable *done* which is initialised as the empty set (see line 4) and each pair (*spec*, *impl*) that is popped from *working* at line 6 is added to *done* (line 23). For termination of the algorithm, we argue that every state in the product gets visited, and is added to *done*, at most once. A crucial observation in our reasoning is the following property of an antichain: adding elements to an antichain does not affect the membership test of elements already included. This is formalised by the lemma below.

**Lemma 5.3.** *Let  $(Z, \leq)$  be a partially ordered set and  $\mathcal{A} \subseteq Z$  an antichain. For all elements  $x, y \in Z$  if  $x \in \mathcal{A}$  and  $y \notin \mathcal{A}$  then  $x \in (\mathcal{A} \uplus y)$  holds.*

*Proof.* Assume arbitrary elements  $x, y \in Z$  such that  $x \in \mathcal{A}$  and  $y \notin \mathcal{A}$ . Recall that the definition of  $\mathcal{A} \uplus y$  results in an antichain  $\{z \mid z = y \vee (z \in \mathcal{A} \wedge y \not\leq z)\}$ , because  $y \notin \mathcal{A}$  by assumption. Consider the following two cases:

- Case  $y \leq x$ . Then  $x \in (\mathcal{A} \uplus y)$  follows from the fact that  $y \in (\mathcal{A} \uplus y)$ .
- Case  $y \not\leq x$ . There is an element  $z \in \mathcal{A}$  such that  $z \leq x$  by assumption that  $x \in \mathcal{A}$ . Because  $y \not\leq x$  and  $z \leq x$  we also know that  $y \not\leq z$ . Consequently,  $z \in (\mathcal{A} \uplus y)$  and thus also  $x \in (\mathcal{A} \uplus y)$ .  $\square$

Next, we prove that *done* and *working* are disjoint, which implies that pairs present in *done* (which is a set that is easily seen to only grow) are not added to *working* again. Showing this property to be true requires two additional observations, *viz.*, (1) pairs in *working* and *done* are contained in *antichain*, and (2), *working* contains only *unique* pairs, thus representing a proper set (and, by abuse of notation, we will treat it as such). For the purpose of identifying elements in *working* we define, for a given index  $i$ , the notation  $\text{working}^i$  to represent the  $i$ th pair on the stack. Now we can describe that all elements in *working* are unique by showing that  $\forall i \neq j : \text{working}^i \neq \text{working}^j$  holds true. The lemma below formalises these insights.

**Lemma 5.4.** *The following invariant holds in the while loop (lines 5-23) of Algorithm 6:*

$$(\text{done} \cup \text{working}) \in^{\forall} \text{antichain} \wedge (\forall i \neq j : \text{working}^i \neq \text{working}^j) \wedge (\text{done} \cap \text{working}) = \emptyset \quad (\text{I})$$

*Proof.* Initially, the initial pair is added to both *working* and *antichain*, and *done* is empty, so the invariant holds trivially upon entry of the while loop.

Maintenance. At line 6 we know that  $(spec, impl) \in antichain$  from  $working \in^\forall antichain$ . Therefore, it holds that  $(done \cup \{(spec, impl)\}) \in^\forall antichain$  and  $(working \setminus \{(spec, impl)\}) \in^\forall antichain$ . Furthermore, from  $\forall i \neq j : working^i \neq working^j$  it follows that  $(spec, impl) \notin (working \setminus \{(spec, impl)\})$ . Upon executing line 6 we may therefore conclude that  $((done \cup \{(spec, impl)\}) \cap (working \setminus \{(spec, impl)\})) = \emptyset$ .

Next, notice that as a result of condition  $(spec', impl') \notin antichain$  on line 20, we have  $(spec', impl') \notin (done \cup working)$ . Let  $working' = \{(spec', impl')\} \cup (working \setminus \{(spec, impl)\})$  and  $done' = \{(spec, impl)\} \cup done$ . From the fact that  $(spec', impl') \notin working$  it follows that  $\forall i \neq j : working'^i \neq working'^j$  holds true. At line 21 the  $(spec', impl')$  pair is added to  $antichain$  and Lemma 5.3 ensures that  $(done' \cup working') \in^\forall (antichain \uplus (spec', impl'))$  holds. Finally, from  $((done \cup \{(spec, impl)\}) \cap (working \setminus \{(spec, impl)\})) = \emptyset$  we can also conclude that  $(done' \cap working') = \emptyset$ .  $\square$

Finally, we need to show that the elements in  $done$  and  $working$  are bounded by the state space of the product  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$ .

**Lemma 5.5.** *The invariant  $(done \cup working) \subseteq \text{reachable}(\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2)$  holds in the while loop (lines 5-23) of Algorithm 6.*

*Proof.* Initially, the pair  $(\{s \in S_1 \mid \iota_1 \xRightarrow{\epsilon} s\}, \iota_2)$  is reachable by the empty trace because this pair is the initial state of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  by definition. Therefore,  $working$ , which only consists of this pair, contains pairs that are reachable as well. Moreover,  $done$  is empty, so the invariant holds upon entry of the while loop.

Maintenance. Let  $(spec, impl)$  be a pair that is popped from  $working$  and assume that  $\llbracket spec \rrbracket_{\mathcal{L}_1} \uparrow$  does not hold. Note that, by our invariant, there is a trace  $\sigma \in Act_\tau^*$ , such that  $\iota \xrightarrow{\sigma} (spec, impl)$ . At line 13 the outgoing transition  $(impl, a, impl')$  is an element of  $\rightarrow_2$ . Line 14 corresponds exactly to the first case of the product definition (Def. 3.1). Similarly, line 16 corresponds exactly to the second case of the product definition where  $(spec, a, spec')$  is a transition in  $\text{norm}_{\text{fdr}}(\mathcal{L}_1)$  because  $\llbracket spec \rrbracket_{\mathcal{L}_1} \uparrow$  does not hold. As such, there is a transition  $(spec, impl) \xrightarrow{a} (spec', impl')$  in the product LTS. By definition of a trace and the definition of reachable this means that  $(working \cup \{(spec', impl')\}) \subseteq \text{reachable}(\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2)$ . From the observation that  $(spec, impl)$  was reachable we can conclude that  $(done \cup \{(spec, impl)\})$  is a subset of the reachable states as well.  $\square$

**Theorem 5.6.** *Algorithm 6 terminates for finite state, finitely branching LTSs.*

*Proof.* The inner for-loop is bounded as the number of outgoing transitions  $\rightarrow_2$  is finite. The total number of state pairs in  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  is finite since  $S_1$  and  $S_2$  are finite. From Lemma 5.5 it follows that  $done$  is a subset of the reachable state pairs. Furthermore, 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 while loop are possible.  $\square$

Note that these observations 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  $(done \cup working) \in^\forall antichain$  do not hold for Algorithm 1, 2 and 3, as already observed in Example 4.4.

The remainder of this section is dedicated to proving the partial correctness of Algorithm 6, *viz.*, that when it terminates, the algorithm correctly decides failures-divergences refinement. We first revisit the claim we made in Section 4; before we restate and prove this claim, we prove a simplified version thereof in the next lemma.

**Lemma 5.7.** *For all states  $(U, s), (V, s)$  of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  satisfying  $(U, s) \leq (V, s)$  and actions  $a \in \text{Act}_\tau$  such that  $(V, s) \xrightarrow{a} (V', t)$  there is a state  $(U', t)$  such that  $(U, s) \xrightarrow{a} (U', t)$  and  $(U', t) \leq (V', t)$ .*

*Proof.* Let  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2 = (S, \iota, \rightarrow)$  and let  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) = (S'_1, \iota'_1, \rightarrow'_1)$ . Take any two state pairs such that  $(U, s) \leq (V, s)$ . Pick an arbitrary pair  $(V', t) \in S$  and action  $a \in \text{Act}_\tau$  such that  $(V, s) \xrightarrow{a} (V', t)$ . Now there are two cases to distinguish:

- Case  $a = \tau$ . Then a transition  $s \xrightarrow{\tau}_2 t$  exists and  $V = V'$ . Therefore, there is also a transition  $(U, s) \xrightarrow{\tau} (U', t)$  and  $U = U'$ . By the assumption that  $(U, s) \leq (V, s)$  we know that  $(U', t) \leq (V', t)$ .
- Case  $a \neq \tau$ . Then there are transitions  $V \xrightarrow{a}_1 V'$  and  $s \xrightarrow{a}_2 t$ . The normalisation has, by definition, transition  $V \xrightarrow{a}_1 V'$  if and only if  $V' = \{v' \in S_1 \mid \exists v \in \llbracket V \rrbracket_{\mathcal{L}_1} : v \xrightarrow{a}_1 v'\}$  and not  $\llbracket V \rrbracket_{\mathcal{L}_1} \uparrow$ . Let  $U'$  be equal to  $\{u' \in S_1 \mid \exists u \in \llbracket U \rrbracket_{\mathcal{L}_1} : u \xrightarrow{a}_1 u'\}$ . From  $\llbracket U \rrbracket_{\mathcal{L}_1} \subseteq \llbracket V \rrbracket_{\mathcal{L}_1}$  it follows that  $\llbracket U' \rrbracket_{\mathcal{L}_1} \subseteq \llbracket V' \rrbracket_{\mathcal{L}_1}$ . Furthermore, as  $\llbracket V \rrbracket_{\mathcal{L}_1} \uparrow$  does not hold it follows that not  $\llbracket U \rrbracket_{\mathcal{L}_1} \uparrow$ . Therefore,  $U \xrightarrow{a}_1 U'$  exists and  $(U, s) \xrightarrow{a} (U', t)$  is a transition in the product with  $(U', t) \leq (V', t)$ .  $\square$

We are now in a position to formally prove the claim that we made in Section 4. For convenience, we repeat the claim as a proposition below.

**Proposition 5.8.** *For all states  $(U, s), (V, s)$  of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  satisfying  $(U, s) \leq (V, s)$  and for every sequence  $\sigma \in \text{Act}_\tau^*$  such that  $(V, s) \xrightarrow{\sigma} (V', t)$  there is a state  $(U', t)$  such that  $(U, s) \xrightarrow{\sigma} (U', t)$  and  $(U', t) \leq (V', t)$ .*

*Proof.* Let  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2 = (S, \iota, \rightarrow)$ . The proof is by induction on the length of sequences in  $\text{Act}_\tau^*$ .

Base case. Take two pairs  $(U, s), (V, s) \in S$  satisfying  $(U, s) \leq (V, s)$ . The empty trace can only reach  $(U, s) \xrightarrow{\epsilon} (U, s)$ ; similarly, we have  $(V, s) \xrightarrow{\epsilon} (V, s)$ . Therefore,  $(U, s) \leq (V, s)$  follows by assumption.

Inductive step. Suppose that the statement holds for all sequences in  $\text{Act}_\tau^*$  of length  $i$  and take a sequence  $\sigma \in \text{Act}_\tau^*$  of length  $i$ . Take arbitrary states  $(V, s), (V'', r) \in S$  and action  $a \in \text{Act}_\tau$  such that  $(V, s) \xrightarrow{\sigma a} (V'', r)$ . Then there is a state  $(V', t) \in S$  such that  $(V, s) \xrightarrow{\sigma} (V', t)$  and  $(V', t) \xrightarrow{a} (V'', r)$ . From the induction hypothesis it follows that for all  $(U, s) \leq (V, s)$  there is a state  $(U', t) \leq (V', t)$  such that  $(U, s) \xrightarrow{\sigma} (U', t)$ . By Lemma 5.7 and the existence of  $(V', t) \xrightarrow{a} (V'', r)$  there is a state  $(U'', r) \leq (V'', r)$  such that  $(U', t) \xrightarrow{a} (U'', r)$ . We thus conclude that  $(U, s) \xrightarrow{\sigma a} (U'', r)$ .  $\square$

The correctness arguments of Algorithm 6 furthermore require a lemma showing the anti-monotonicity of FD-witnesses. Such a result is needed because the antichain algorithms may explore only part of the reachable state space of a product. The anti-monotonicity property helps to show, however, that the part that is explored contains all relevant information.

**Lemma 5.9.** *For all states  $(U, s), (V, s)$  of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  satisfying  $(U, s) \leq (V, s)$  it holds that if  $(V, s)$  is an FD-witness then  $(U, s)$  is an FD-witness.*

*Proof.* Take arbitrary states  $(U, s), (V, s)$  of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  satisfying  $(U, s) \leq (V, s)$  and let  $(V, s)$  be an FD-witness. It follows that  $\llbracket V \rrbracket_{\mathcal{L}_1} \uparrow$  does not hold and one of the following holds:  $V = \emptyset$  or  $\text{stable}(s) \wedge \text{refusals}(s) \not\subseteq \text{refusals}(\llbracket V \rrbracket_{\mathcal{L}_1})$  or  $s \uparrow$ . By monotonicity,  $\llbracket U \rrbracket_{\mathcal{L}_1} \subseteq \llbracket V \rrbracket_{\mathcal{L}_1}$  implies  $\text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1}) \subseteq \text{refusals}(\llbracket V \rrbracket_{\mathcal{L}_1})$ , and not  $\llbracket V \rrbracket_{\mathcal{L}_1} \uparrow$  implies not  $\llbracket U \rrbracket_{\mathcal{L}_1} \uparrow$ . Now,

$(U, s)$  is an FD-witness, because  $\llbracket U \rrbracket_{\mathcal{L}_1} \uparrow$  does not hold and if  $V = \emptyset$  then  $U = \emptyset$ , or if  $\text{stable}(s) \wedge \text{refusals}(s) \not\subseteq \text{refusals}(\llbracket V \rrbracket_{\mathcal{L}_1})$  then  $\text{stable}(s) \wedge \text{refusals}(s) \not\subseteq \text{refusals}(\llbracket U \rrbracket_{\mathcal{L}_1})$ , or  $s \uparrow$ .  $\square$

**Corollary 5.10.** *For all states  $(U, s), (V, s)$  of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  where  $(U, s) \leq (V, s)$  and for every sequence  $\sigma \in \text{Act}_\tau^*$  it holds that if  $(V, s)$  can reach an FD-witness with  $\sigma$  then  $(U, s)$  can reach an FD-witness with  $\sigma$  as well.*

*Proof.* Let  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2 = (S, \iota, \rightarrow)$ . Take arbitrary states  $(U, s), (V, s) \in S$  satisfying  $(U, s) \leq (V, s)$ . Let  $(V', t)$  be an FD-witness and  $\sigma \in \text{Act}_\tau^*$  a trace such that  $(V, s) \xrightarrow{\sigma} (V', t)$ . By Lemma 5.8 there is a pair  $(U', t) \leq (V', t)$  such that  $(U, s) \xrightarrow{\sigma} (U', t)$ . From Lemma 5.9 it follows that state  $(U', t)$  is an FD-witness.  $\square$

For a set of states  $S'$  of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$ , let  $\text{FDR}(S')$  be the predicate that is true if and only if  $S'$  contains an FD-witness. For a state  $s$  in the product, we define the *distance* to a set of states  $S'$  of the product as the *shortest* distance from state  $s$  to a state in  $S'$ . If  $S'$  is unreachable, the distance is set to infinity. Formally,  $\text{Dist}_{S'}(s) = \min\{|\sigma| \mid \sigma \in \text{traces}(\mathcal{L}) \wedge t \in S' \wedge s \xrightarrow{\sigma} t\}$ , where  $\min\{\emptyset\}$  is defined as  $\infty$ . For a set of states  $S''$ , let  $\text{Dist}_{S'}(S'')$  denote the shortest distance among all states in  $S''$ , formally  $\text{Dist}_{S'}(S'') = \min\{\text{Dist}_{S'}(s) \mid s \in S''\}$ . We denote the set of all reachable FD-witnesses in the product  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  by  $\mathcal{F}$ .

**Lemma 5.11.** *For all states  $(U, s), (V, s)$  of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  satisfying  $(U, s) \leq (V, s)$  it holds that  $\text{Dist}_{\mathcal{F}}((U, s)) \leq \text{Dist}_{\mathcal{F}}((V, s))$ .*

*Proof.* Let  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2 = (S, \iota, \rightarrow)$ . Take arbitrary states  $(U, s), (V, s) \in S$  satisfying  $(U, s) \leq (V, s)$ . From Corollary 5.10 it follows that if  $(V, s)$  can reach an FD-witness by the shortest trace  $\sigma$  then  $(U, s)$  can also reach an FD-witness with trace  $\sigma$ , which by definition means that  $\text{Dist}_{\mathcal{F}}((U, s)) \leq \text{Dist}_{\mathcal{F}}((V, s))$ .  $\square$

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

**Lemma 5.12.** *For all states  $(U, s)$  of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  it holds that if  $\llbracket U \rrbracket_{\mathcal{L}_1} \uparrow$  then  $\text{Dist}_{\mathcal{F}}((U, s))$  is  $\infty$ .*

*Proof.* Let  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2 = (S, \iota, \rightarrow)$  and let  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) = (S'_1, \iota'_1, \rightarrow'_1)$ . Take an arbitrary state  $(U, s) \in S$  such that  $\llbracket U \rrbracket_{\mathcal{L}_1} \uparrow$ . For any action  $a \in \text{Act}_\tau$  and state  $V \in S'_1$  there is no transition  $U \xrightarrow{a}_1 V$  by definition of  $\text{norm}_{\text{fdr}}$ . Consequently, from  $(U, s)$ , only  $\tau$ -transitions due to  $\mathcal{L}_2$  can be taken. As a result, by definition of the product and Lemma 3.2, for any state  $(V, t) \in S$  such that  $(U, s) \xrightarrow{\epsilon} (V, t)$  it holds that  $U = V$ . Thus, any reachable state  $(V, t)$  also satisfies  $\llbracket V \rrbracket_{\mathcal{L}_1} \uparrow$  and therefore cannot be an FD-witness. Hence,  $\text{Dist}_{\mathcal{F}}((U, s))$  is  $\infty$ .  $\square$

**Lemma 5.13.** *If  $\text{FDR}(\text{reachable}(\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2))$  is true then invariant II holds for every iteration of the while loop (lines 5-23) of Algorithm 6:*

$$\text{Dist}_{\mathcal{F}}(\text{done}) > \text{Dist}_{\mathcal{F}}(\text{working}) \wedge \text{Dist}_{\mathcal{F}}(\text{working}) = \text{Dist}_{\mathcal{F}}(\text{antichain}) \quad (\text{II})$$



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

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

**Maintenance.** Assume that *working* is not empty and that  $\text{Dist}_{\mathcal{F}}(\text{done}) > \text{Dist}_{\mathcal{F}}(\text{working})$  and  $\text{Dist}_{\mathcal{F}}(\text{working}) = \text{Dist}_{\mathcal{F}}(\text{antichain})$  hold. At line 6 a pair  $(\text{spec}, \text{impl})$  is taken from *working*, so *working*, which by invariant I represents a set, becomes equal to  $\text{working} \setminus \{(\text{spec}, \text{impl})\}$ . Let  $\text{done}' = \text{done} \cup \{(\text{spec}, \text{impl})\}$  and let  $N = \text{Dist}_{\mathcal{F}}((\text{spec}, \text{impl}))$ . There are three cases to distinguish.

- Case  $N > \text{Dist}_{\mathcal{F}}(\text{working} \cup \{(\text{spec}, \text{impl})\})$ . Removing  $(\text{spec}, \text{impl})$  from *working* did not change its distance, so  $\text{Dist}_{\mathcal{F}}(\text{working}) = \text{Dist}_{\mathcal{F}}(\text{working} \cup \{(\text{spec}, \text{impl})\})$ . Because  $N > \text{Dist}_{\mathcal{F}}(\text{working})$ , adding this pair to *done* results in  $\text{Dist}_{\mathcal{F}}(\text{working}) < \text{Dist}_{\mathcal{F}}(\text{done}') \leq \text{Dist}_{\mathcal{F}}(\text{done})$ . Consider the outgoing transitions  $(\text{impl}, a, \text{impl}') \in \rightarrow_2$  at line 13. The resulting pairs  $(\text{spec}', \text{impl}')$  must have a distance of at least  $\text{Dist}_{\mathcal{F}}(\text{working})$ , because  $N - 1 \geq \text{Dist}_{\mathcal{F}}(\text{working})$ . Let  $\text{working}' = \text{working} \cup \{(\text{spec}', \text{impl}')\}$ . Then  $\text{Dist}_{\mathcal{F}}(\text{working}) = \text{Dist}_{\mathcal{F}}(\text{working}')$ . Let *antichain'* be *antichain* if  $(\text{spec}', \text{impl}')$  was not inserted and let it be  $\text{antichain} \uplus (\text{spec}', \text{impl}')$  otherwise. By the invariant it follows that  $N - 1 \geq \text{Dist}_{\mathcal{F}}(\text{antichain})$  and so by Lemma 5.11 if  $(\text{spec}', \text{impl}')$  is inserted into *antichain* its distance will also not change. Therefore,  $\text{Dist}_{\mathcal{F}}(\text{done}') > \text{Dist}_{\mathcal{F}}(\text{working}') \wedge \text{Dist}_{\mathcal{F}}(\text{working}') = \text{Dist}_{\mathcal{F}}(\text{antichain}')$ .
- Case  $0 < N \leq \text{Dist}_{\mathcal{F}}(\text{working} \cup \{(\text{spec}, \text{impl})\})$ . Observe that  $N$  must be equal to  $\text{Dist}_{\mathcal{F}}(\text{working} \cup \{(\text{spec}, \text{impl})\})$ . From Lemma 5.12 it follows that  $\llbracket \text{spec} \rrbracket_{\mathcal{L}_1} \uparrow$  does not hold and so the successors of  $(\text{spec}, \text{impl})$  are explored. Invariant II holds upon termination of the inner for-loop at lines 13 to 22. This follows from an invariant for the inner for-loop, which we state next.

Let *antichain'* be equal to the value of variable *antichain* after line 13 at each iteration and *working'* be equal to the value of variable *working*. Furthermore, let  $T$  be the set of successors of  $(\text{spec}, \text{impl})$ , due to the transitions emanating from *impl*, and let *done'* be the successors that have been processed, i.e., *done'* is initially empty and  $(\text{spec}', \text{impl}')$  is inserted into it after line 17. It can be shown, using Lemma 5.11, that the following is an invariant for the inner for-loop:

$$\begin{aligned} &(\text{Dist}_{\mathcal{F}}(T \setminus \text{done}') < N \vee \text{Dist}_{\mathcal{F}}(\text{working}') < N) \\ &\wedge \text{Dist}_{\mathcal{F}}(\text{working}') = \text{Dist}_{\mathcal{F}}(\text{antichain}') \end{aligned}$$

Upon termination we conclude that  $(T \setminus \text{done}') = \emptyset$ . It then follows that  $\text{Dist}_{\mathcal{F}}(T \setminus \text{done}') = \infty$ . As a consequence, we find that  $\text{Dist}_{\mathcal{F}}(\text{working}') < N$  and therefore  $\text{Dist}_{\mathcal{F}}(\text{working}') < \text{Dist}_{\mathcal{F}}(\text{done} \cup \{(\text{spec}, \text{impl})\})$ .

- Case  $N = 0$ . The state  $(\text{spec}, \text{impl})$  is checked for the FD-witness conditions and the algorithm terminates.  $\square$

We conclude with the following result, which underlies the correctness of Algorithm 6.

**Theorem 5.14.** *Algorithm 6 returns false if and only if an FD-witness is reachable in the product  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$ .*

*Proof.*  $\implies$ ) Assume that Algorithm 6 returns false. This occurs when the current pair  $(spec, impl)$  satisfies the conditions of an FD-witness, as shown in lines 7, 8, 11 and 18 of Algorithm 6. All pairs taken from *working* are reachable according to Lemma 5.5, so this FD-witness is also reachable.

$\impliedby$ ) Assume that an FD-witness is reachable in the product of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$ , i.e.,  $\mathcal{F} \neq \emptyset$ . Then invariant II of Lemma 5.13 holds:

$$\text{Dist}_{\mathcal{F}}(done) > \text{Dist}_{\mathcal{F}}(working) \wedge \text{Dist}_{\mathcal{F}}(working) = \text{Dist}_{\mathcal{F}}(antichain)$$

Towards a contradiction, assume that Algorithm 6 returns true. The algorithm returns true if and only if *working* is empty, which means that  $\text{Dist}_{\mathcal{F}}(working) = \text{Dist}_{\mathcal{F}}(\emptyset) = \infty$ . The initial state  $\iota$  of  $\text{norm}_{\text{fdr}}(\mathcal{L}_1) \times \mathcal{L}_2$  is equal to  $(\{s \in S_1 \mid \iota_1 \xrightarrow{\epsilon} s\}, \iota_2)$  and can reach an FD-witness by assumption. Therefore,  $\text{Dist}_{\mathcal{F}}(\iota) < \infty$ . Initially  $\iota$  was inserted into *antichain* so by Lemma 5.3 follows that  $\iota \in antichain$  and from Lemma 5.11 it follows that  $\text{Dist}_{\mathcal{F}}(antichain) < \infty$ . Contradiction, so we conclude that if Algorithm 6 terminates then it returns false. Since termination is shown in Theorem 5.6 we establish that the algorithm returns false.  $\square$

We here note that analogues of Theorem 5.14 for Algorithms 4 and 5 can be proved along the same lines. In particular, invariants I and II, fundamental in proving termination, and proved in Lemmas 5.4 and 5.5, can be shown to hold for both algorithms using the same arguments (where, of course, the counterpart of invariant II relies on a distance to the set of TR-witnesses or SF-witnesses). Proposition 5.8 also holds for the product  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ , and Lemma 5.9 and Corollary 5.10 hold for TR-witnesses and SF-witnesses in the product  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ . Without going into these details, we here claim the correctness for Algorithms 4 and 5.

**Theorem 5.15.** *Algorithm 4 returns false if and only if a TR-witness is reachable in the product  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ . Algorithm 5 returns false if and only if an SF-witness is reachable in the product  $\text{norm}(\mathcal{L}_1) \times \mathcal{L}_2$ .*

We finish with restating the formal claim of correctness of all three improved algorithms.

**Theorem 5.2.** *Let  $\mathcal{L}_1 = (S_1, \iota_1, \rightarrow_1)$  and  $\mathcal{L}_2 = (S_2, \iota_2, \rightarrow_2)$  be two LTSs.*

- $\text{REFINES-TRACE}_{\text{NEW}}(\mathcal{L}_1, \mathcal{L}_2)$  returns true if and only if  $\mathcal{L}_1 \sqsubseteq_{\text{tr}} \mathcal{L}_2$ .
- $\text{REFINES-STABLE-FAILURES}_{\text{NEW}}(\mathcal{L}_1, \mathcal{L}_2)$  returns true if and only if  $\mathcal{L}_1 \sqsubseteq_{\text{sfr}} \mathcal{L}_2$ .
- $\text{REFINES-FAILURES-DIVERGENCES}_{\text{NEW}}(\mathcal{L}_1, \mathcal{L}_2)$  returns true if and only if  $\mathcal{L}_1 \sqsubseteq_{\text{fdr}} \mathcal{L}_2$ .

*Proof.* From Theorem 5.14 we can conclude that Algorithm 6 returns false if and only if an FD-witness is reachable. By Theorem 3.24 an FD-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 trace and stable failures refinement.  $\square$

## 6. EXPERIMENTAL VALIDATION

We have conducted several experiments to compare the run time of the various algorithms to show that solving the identified issues actually improves the run time performance in practice. For this purpose we have implemented a depth-first and breadth-first variant for each of the original algorithms (Algorithms 1, 2 and 3) and improved algorithms (Algorithms 4, 5 and 6) in a branch of the mCRL2<sup>1</sup> toolset [BGK<sup>+</sup>19] as part of the *ltscompare* tool, which is

<sup>1</sup>[www.mcrl2.org](http://www.mcrl2.org)

implemented in C++. As the name of the tool suggests it can be used to check for various preorder and equivalence relations between labelled transition systems.

The data structures used in these implementations compute most concepts, *e.g.*, the antichain membership test and insertion, in the same way. However, the implementations of Algorithms 5 and 6 perform the check at line 6, or line 11 respectively, according to the definition of *refusals* we presented in Definition 2.5, whereas the implementations of Algorithms 2 and 3 compute the refusal check with an additional local search, according to the definition given in [WSS<sup>+</sup>12], see also Remark 4.2.

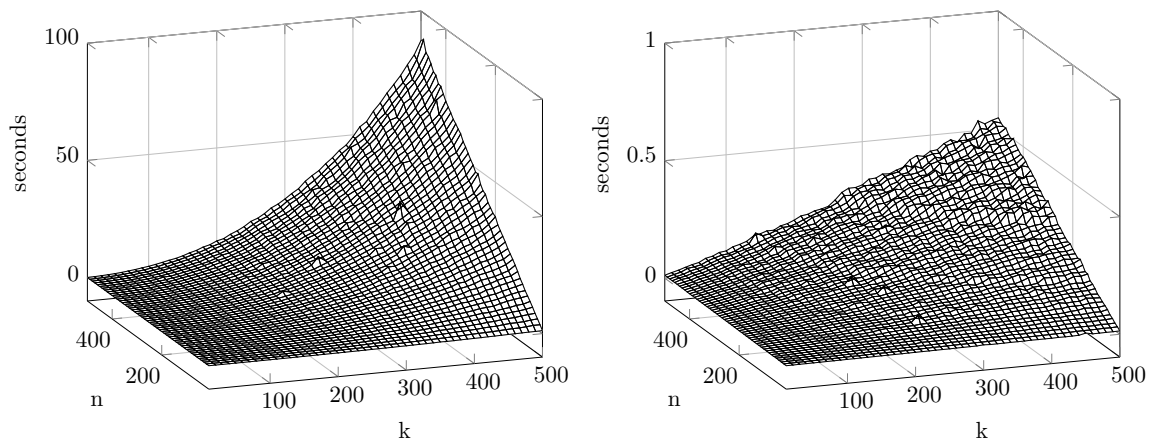
We first revisit Example 4.4 in Section 6.1, illustrating that the performance overhead we predict for the original algorithm for checking trace refinement also manifests itself in practice. In Section 6.2, we then analyse the performance of the algorithms on practical examples consisting of a model of an industrial system and models of concurrent data structures. Finally, in Section 6.3, we analyse the effect of using a cheap state space minimisation algorithm on the total run time of the algorithms.

All experiments and measurements have been performed on a machine with an Intel Core i7-7700HQ CPU 2.80Ghz and a 16GiB memory limit imposed by `ulimit -Sv 16777216`. The source modifications and experiments can be obtained from the downloadable package [Lav19].

**6.1. Experiment I: Example 4.4.** We have used our implementations of Algorithms 1 and 4 to measure the run time (in seconds) for checking the trace refinement  $\mathcal{L}_n^k \sqsubseteq_{\text{tr}} \mathcal{L}_n^k$ , for all combinations of parameters  $n, k \in \{10, 20, \dots, 500\}$ , as described in Example 4.4. The results of these measurements are shown as a 3D-plot in Figure 1.

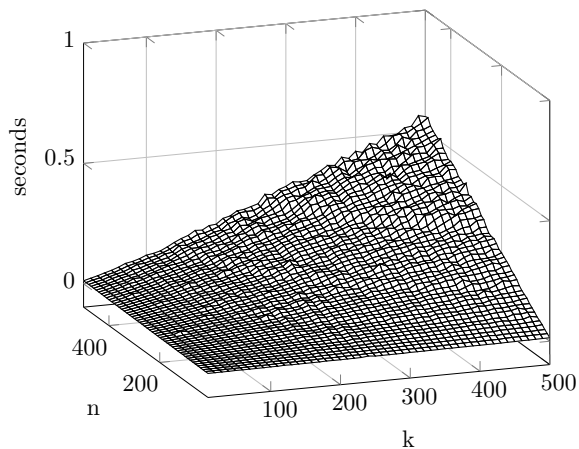
These plots show a quadratic growth of Algorithm 1 in the parameter  $k$  and a linear growth in the parameter  $n$ . For Algorithm 4 the asymptotic growth is linear in both  $k$  and  $n$ . These observed growths coincide with the analysis that was presented in Example 4.4 for Algorithm 1 and on page 19 for Algorithm 4. Note that the scale of the vertical axes of both plots, 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 4. As there is no difference in the data structures the difference in run time is entirely due to the different way of inspecting and extending *working* and *antichain*.

Figure 1: The run time results for Example 4.4 using the depth-first variant of Algorithm 1 on the left and our Algorithm 4 on the right.



The breadth-first variant of Algorithm 1 was unable to complete the smallest, *i.e.*,  $n = k = 10$ , case within the given memory limit. However, as shown in Figure 2 the run time performance of the breadth-first variant of the improved algorithm is almost equivalent to its depth-first variant.

Figure 2: The run time for Example 4.4 using the breadth-first variant of Algorithm 4.



**6.2. Experiment II: Practical Examples.** The experiments that we consider are taken from two sources. First, a model of an industrial system that first exposed the performance issues in practice of a control system modelled in the Dezyne language [vBGH<sup>+</sup>17]. This example is of a more traditional flavour, in which the specification is an abstract description of the behaviours at the external interface of a control system, and the implementation is a detailed model that interacts with underlying services to implement the expected interface. For reasons of confidentiality, the industrial model cannot be made available.

Second, we consider several *linearisability tests* of concurrent data structures. These models have been taken from [Pav18], and consist of six implementations of concurrent data types that, when *trace refining* their specifications, are guaranteed to be linearisable. As in [WSS<sup>+</sup>12], we approximate trace refinement by the stronger stable failures refinement. For these models, the implementation and specification pairs are based on the same descriptions; the difference between the two is that the specification uses a simple construct to guarantee that each method of the concurrent data structure executes atomically. This significantly reduces the non-determinism and the number of transitions in the specification models.

In Table 1 the origin of each model, 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.

Model	Ref.	states spec	trans. spec	$\sqsubseteq_{\text{sfr}}$	states impl	trans. impl
Coarse set	[HS08]	50 488	64 729	True	55 444	145 043
Fine-grained set	[HS08]	3 720	3 305	True	5 077	9 006
Lazy set	[HS08]	3 565	3 980	True	24 496	41 431
Optimistic set	[HS08]	25 435	28 154	True	234 332	389 344
Non-blocking queue	[SHC00]	1 248	1 473	False	3 030	5 799
Treiber stack	[Tre86]	87 389	124 740	True	205 634	564 862
Industrial	-	24	45	True	24 551	45 447

The run time measurements of both Algorithms 2 and 5 with both the depth-first and breadth-first variants is shown in Table 2. The run times that we report are the averages obtained from five consecutive runs.

Table 2: Run time comparison between Algorithm 2 and Algorithm 5 using depth-first (df) and breadth-first (bf) exploration.

Model	Alg. 2 df (s)	Alg. 2 bf (s)	Alg. 5 df (s)	Alg. 5 bf (s)
Coarse set	9.15	†	8.61	9.06
Fine-grained set	0.37	†	0.32	0.46
Lazy set	1.19	†	1.02	1.26
Optimistic set	16.96	†	14.13	22.67
Non-blocking queue	0.03	0.17	0.02	0.09
Treiber stack	148.39	†	137.52	352.59
Industrial	1.36	296.29	0.15	0.17

Here, we observe that the depth-first variant of both algorithms perform similarly with a small run time advantage for Algorithm 5. However, for the breadth-first variants our algorithm is able to complete all experiments, whereas Algorithm 2 reaches the memory limit, indicated by †, in five cases and only completes two cases successfully.

To gain more insight into the performance differences between both algorithms we repeat the experiments and report a number of performance metrics. The reported metrics are the maximum *working* size and the number of *antichain* membership test that fail (misses), succeed (hits) and the maximum *antichain* size during the exploration. We report the maximum size instead of its size upon termination as these do not necessarily coincide,

because inserting an element can evict one or more pairs in *antichain*. The following two tables (Tables 3 and 4) show the discussed metrics for the depth-first variant of both algorithms.

Table 3: Performance metrics for the depth-first variant of Algorithm 2.

Model	<i>working</i> max	<i>antichain</i> hits	<i>antichain</i> misses	<i>antichain</i> max
Coarse set	96	93 330	58 438	55 444
Fine-grained set	60	5 786	7 575	5 077
Lazy set	61	21 184	30 771	24 496
Optimistic set	96	234 692	354 068	238 726
Non-blocking queue	52	548	672	591
Treiber stack	101	1 238 727	756 692	234 118
Industrial	74	36 544	43 419	43 091

Table 4: Performance metrics for the depth-first variant of Algorithm 5.

Model	<i>working</i> max	<i>antichain</i> hits	<i>antichain</i> misses	<i>antichain</i> max
Coarse set	96	93 330	58 438	55 444
Fine-grained set	60	5 786	7 575	5 077
Lazy set	61	21 184	30 771	24 496
Optimistic set	96	234 692	354 068	238 728
Non-blocking queue	43	520	641	634
Treiber stack	101	1 238 727	756 692	234 119
Industrial	69	36 369	43 090	43 091

We observe in Tables 3 and 4 that only for the industrial and non-blocking queue models the performance metrics are different. An explanation for this is that because the antichain membership test is delayed (in Algorithm 2), more pairs are added to *working* and these additional pairs increase the number of *antichain* checks. In all other cases, the difference in run time can only be the result of the different refusal computation implementation, as the number of *antichain* operations is the same.

The following two tables (Tables 5 and 6) show the obtained performance metrics for the breadth-first variants of both algorithms. In the experiments where the refinement checking terminates early, due to reaching the memory limit, we report the last observed measurements.

Table 5: Performance indicators for the breadth-first variant of Algorithm 2.

Model	<i>working</i> max	<i>antichain</i> hits	<i>antichain</i> misses	<i>antichain</i> max
Coarse set	4 710 289	13 870	7 807 403	3 629
Fine-grained set	6 604 516	180 669	15 547 890	1 900
Lazy set	6 726 497	130 523	14 852 835	4 306
Optimistic set	6 366 524	38 649	14 238 042	4 439
Non-blocking queue	6 262	3 078	14 560	274
Treiber stack	5 829 902	76 114	8 340 606	4 811
Industrial	549 263	5 459 028	12 888 388	43 091

Table 6: Performance indicators for the breadth-first variant of Algorithm 5.

Model	<i>working</i> max	<i>antichain</i> hits	<i>antichain</i> misses	<i>antichain</i> max
Coarse set	3 411	96 167	60 332	55 444
Fine-grained set	434	7 192	9 657	5 077
Lazy set	1 748	24 340	35 192	24 496
Optimistic set	15 209	292 525	434 218	234 352
Non-blocking queue	338	3 426	4 032	2 675
Treiber stack	139 218	2 411 614	1 523 830	214 795
Industrial	2 243	36 369	43 090	43 091

From these results it is clear to see that for the breadth-first variant of Algorithm 2, delaying the *antichain* insertion of discovered state pairs results in an enormous overhead. The size of the *antichain* remains quite small, which causes many discovered pairs to fail the antichain membership test. As each pair that fails the membership test is added to *working*, it causes the *working* queue to grow rapidly, until it reaches the memory limit. On the other hand, for Algorithm 5 we can observe that the number of successful (and unsuccessful) *antichain* membership test is quite similar to its depth-first variant. There can be some differences between these variants as the pairs are discovered in a different order. The increase of the *working* size has the same reason as for ordinary breadth-first search, which depends on the out degree of the visited pairs.

To verify that the difference in performance of the depth-first variants is due to the changes of the refusal computation we have implemented another variant of Algorithm 2 with the stability check of *impl* added. The run time impact of this change for both depth-first and breadth-first variants of Algorithm 2 is shown in Table 7.

Table 7: Run time results for Algorithm 2 with the stability check of *impl*.

Model	Alg. 2 df (s)	Alg. 2 bf (s)
Coarse set	9.59	†
Fine-grained set	0.36	†
Lazy set	1.12	†
Optimistic set	15.85	†
Non-blocking queue	0.03	†
Treiber stack	156.67	†
Industrial	0.17	26.32

As expected, the run time for this alternative depth-first variant closely matches the run time of the depth-first variant of the improved algorithm. The alternative breadth-first variant of Algorithm 2 is still not able to complete most experiments, but the industrial case has improved quite significantly. However, the non-blocking queue experiment now reaches the set memory limit. For this we provide the following explanation. Note that in case of a failing refinement the exploration stops when a suitable (SF-)witness has been found, which must exist as stable failures refinement does not hold. Recall that the computation of refusals for (possibly) unstable states as defined in [WSS<sup>+</sup>12], see Remark 4.2, has been implemented using a separate local search for stable states. We think that in the previous case such an SF-witness was found for an unstable state using this local search. However, in the alternative

version the algorithm continues the exploration of the LTSs when encountering an unstable state (in  $\mathcal{L}_2$ ), which causes the *working* queue to reach the memory limit.

Finally, we repeat the same experiments while checking for failures-divergences refinement. This has only been done for Algorithm 6 as the original algorithm for failures-divergences refinement is incorrect. The run time measurements and the expected result of the failures-divergences refinement check are presented in Table 8.

Table 8: The run time results for checking failures-divergences refinement using Algorithm 6.

Model	Alg. 6 df (s)	Alg. 6 bf (s)	$\sqsubseteq_{\text{fdr}}$
Coarse set	8.68	9.29	True
Fine-grained set	0.33	0.48	True
Lazy set	1.04	1.33	True
Optimistic set	14.55	23.81	True
Non-blocking queue	0.08	0.1	True
Treiber stack	140.7	363.34	True
Industrial	0.05	0.05	False

The run time results of Table 8 show that deciding failures-divergences refinement has a similar performance to deciding stable failures.

**6.3. State Space Minimisation as Preprocessing.** The size of the transition systems has a major impact on the practical run time of the refinement checking algorithms we studied, as can also be seen from, *e.g.*, Tables 1 and 2. Note that this is particularly true of the size of the specification LTS, whose normal form can be exponentially larger than the specification itself. As an alternative to the pruning achieved using antichains, reducing the size of the specification as a preprocessing step to checking for refinement may therefore be an effective tool in improving on the practical run time of these algorithms. Of course, it is desirable that the computational overhead of the reduction remains minimal. One possibility is to minimise transition systems using one of the many equivalence relations available for labelled transition systems, see, *e.g.* [vG93, BGR16]. When choosing such an equivalence it is important that it has the property that, apart from an appealing run time complexity, the observations, *i.e.*, **weaktraces**, **failures** and **divergences**, that are extracted from equivalent states are the same.

*Strong* bisimilarity is known to preserve the **weaktraces**, **failures** and **divergences** observations of equivalent states. However, a more substantial state space reduction can often be achieved by considering equivalences that treat the special action  $\tau$  as invisible, as the given LTSs often contain  $\tau$ -transitions. In [Ros10], Roscoe suggests to use a variant of weak bisimulation, *viz.*, *divergence respecting weak bisimulation*, to minimise a transition system. For *divergence respecting weak* bisimulation it is known that it is a suitable abstraction; see the following theorem [Ros10, Theorem 9.2].

**Theorem 6.1.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS. For two states  $s, t \in S$  that are divergence respecting weak bisimilar it holds that  $\text{weaktraces}(s) = \text{weaktraces}(t)$ ,  $\text{divergences}(s) = \text{divergences}(t)$ ,  $\text{failures}(s) = \text{failures}(t)$ . Hence, also  $\text{failures}_\perp(s) = \text{failures}_\perp(t)$ .*

From a computational point of view, however, (divergence respecting) weak bisimulation is not particularly promising. For instance, the best known algorithm [RT08] for computing



weak bisimulation has a worst-case time complexity of  $\mathcal{O}(m \cdot n)$ , where  $n$  is the number of states and  $m$  the number of transitions. Such run time complexities are non-negligible and may result in an undesirable overhead.

In practice, divergence-respecting weak bisimulation often coincides with *divergence preserving branching bisimulation*<sup>2</sup> and it has the far more appealing worst-case run time complexity of  $\mathcal{O}(m \cdot \log n)$  [JGKW20], which is equivalent to the run time complexity of computing strong bisimulation [PT87]. Divergence-preserving branching bisimulation is stronger than divergence respecting weak bisimulation, *i.e.*, two states that are divergence-preserving branching bisimilar are also divergence respecting weak bisimilar. Divergence-preserving branching bisimilarity is, by Theorem 6.1, therefore a suitable abstraction.

The algorithm that decides divergence-preserving branching bisimulation equivalence between two states can also be adapted to a minimisation procedure with the same  $\mathcal{O}(m \cdot \log n)$  time complexity. We have made the preprocessing step of minimisation modulo divergence-preserving branching bisimulation available as an option in our tool. In Table 9 the number of states and transitions of each model of Section 6.2, *after* minimisation modulo divergence-preserving branching bisimulation, and whether the specification and implementation LTSs are divergence-preserving branching bisimilar is shown.

Table 9: The number of states and transitions after diverging preserving branching bisimulation minimisation and whether the LTSs are equivalent in divergence-preserving branching bisimulation semantics denoted by  $\Leftrightarrow_{db}$ .

Model	states spec	trans. spec	$\Leftrightarrow_{db}$	states impl	trans. impl
Coarse set	1 089	3 618	True	1 089	3 618
Fine-grained set	92	210	True	92	210
Lazy set	92	210	True	92	210
Optimistic set	170	410	True	170	410
Non-blocking queue	119	274	False	163	378
Treiber stack	7 988	26 070	True	7 988	26 070
Industrial	24	45	False	4 626	14 380

Observe that for most of the models, the minimised implementation and specification LTSs are of equal size; indeed, in those cases the implementation and specification are divergence-preserving branching bisimulation equivalent, so no further stable failures refinement check would be needed.

One option would therefore be to apply the minimisation to both implementation and specification LTSs. This approach turns out to be beneficial for the Treiber stack example, obtaining a run time of 3 seconds to determine stable failures refinement. The approach is not beneficial for the other examples. Moreover, minimising the implementation might even be less effective in case the refinement relation between specification and implementation does not hold, in which case the refinement check will probably quickly determine this fact.

We therefore measure the effect of using minimised specifications, but unmodified implementations. The run time measurements of checking stable failures refinement using

<sup>2</sup>Also in case of our benchmarks, both relations yield only a minimal difference in size for all models, with the exception of the industrial implementation model. For that model, however, the costs of computing the weak-bisimulation reduction far exceeds the costs of performing the refinement check. Perhaps this could be partially mitigated by more efficient (divergence respecting) weak bisimulation algorithms, but that has not been further investigated.

Algorithms 2 and 5 using the minimised specification LTS is shown in Table 10. The time that it takes to compute the divergence-preserving branching bisimulation minimisation is presented in the last column and the other measurements are the run time of the algorithm including preprocessing.

Table 10: Run time comparison between the original algorithm (Algorithm 2) and the improved algorithm (Algorithm 5) using depth-first (df) and breadth-first (bf) exploration where the specification is reduced modulo divergence-preserving branching bisimulation.

Model	Alg. 2 df (s)	Alg. 2 bf (s)	Alg. 5 df (s)	Alg. 5 bf (s)	Reduction (s)
Coarse set	0.74	†	0.69	0.69	0.10
Fine-grained set	0.04	†	0.04	0.04	0.01
Lazy set	0.21	†	0.15	0.15	0.01
Optimistic set	2.52	†	1.59	1.57	0.04
Non-blocking queue	0.02	0.04	0.02	0.02	0.01
Treiber stack	8.19	†	6.61	11.71	0.24
Industrial	1.38	293.1	0.16	0.17	0.01

Comparing these results with Table 2 shows that reducing the specification modulo divergence-preserving branching bisimulation can indeed substantially improve the performance of the antichain-based algorithms. In particular, it never degrades the performance of our algorithms as the preprocessing time is negligible. For failures-divergences refinement the results, using Algorithm 6, are similar, as is shown in Table 11.

Table 11: The run time results for checking failures-divergences refinement using Algorithm 6 where the specification is reduced modulo divergence-preserving branching bisimulation.

Model	Alg. 6 df (s)	Alg. 6 bf (s)
Coarse set	0.75	0.7
Fine-grained set	0.04	0.04
Lazy set	0.15	0.15
Optimistic set	1.61	1.7
Non-blocking queue	0.02	0.02
Treiber stack	6.76	12.13
Industrial	0.05	0.06

## 7. CONCLUSION

Our study of the antichain-based algorithms for deciding trace refinement, stable failures refinement and failures-divergences refinement presented in [WSS<sup>+</sup>12] revealed that the failures-divergences refinement algorithm is incorrect. All three algorithms perform suboptimally when implemented using a depth-first search strategy and poorly when implemented using a breadth-first search strategy. Furthermore, all three algorithms 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 trace refinement, 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 [BGK<sup>+</sup>19] and is currently used as the backbone in the commercial F-MDE toolset Dezyne; see also [vBGH<sup>+</sup>17].

#### ACKNOWLEDGEMENT

This work is part of the TOP Grants research programme with project number 612.001.751 (AVVA), which is (partly) financed by the Dutch Research Council (NWO). We also would like to thank the anonymous reviewer for their effort and constructive feedback.

#### REFERENCES

- [ACH<sup>+</sup>10] P. A. Abdulla, Y.-F. 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.
- [BGK<sup>+</sup>19] O. Bunte, J. F. Groote, J. J. A. Keiren, M. Laveaux, T. Neele, E. P. de Vink, J. W. Wesselink, A. Wijs, and T. A. C. Willemse. The mCRL2 toolset for analysing concurrent systems - improvements in expressivity and usability. In T. Vojnar and L. Zhang, editors, *TACAS 2019*, volume 11428 of *LNCS*, pages 21–39. Springer, 2019.
- [BGR16] A. Boulgakov, T. Gibson-Robinson, and A. W. Roscoe. Computing maximal weak and other bisimulations. *Formal Asp. Comput.*, 28(3):381–407, 2016.
- [BKO87] J. A. Bergstra, J. W. Klop, and E.-R. 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.
- [BR84] S. D. Brookes and A. W. Roscoe. An improved failures model for communicating processes. In S. D. Brookes, A. W. Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency 1984*, volume 197 of *LNCS*, pages 281–305. Springer, 1984.
- [DR10] L. Doyen and J.-F. Raskin. Antichain algorithms for finite automata. In *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 2–22. Springer, 2010.
- [GABR14] 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.
- [GABR16] T. Gibson-Robinson, P. J. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3: a parallel refinement checker for CSP. *STTT*, 18(2):149–167, 2016.
- [GB16] A. O. Gomes and A. Butterfield. Modelling the haemodialysis machine with circus. In M. J. Butler, K.-D. Schewe, A. Mashkoor, and M. Biró, editors, *ABZ 2016*, volume 9675 of *LNCS*, pages 409–424. Springer, 2016.
- [GBC<sup>+</sup>17] T. Gibson-Robinson, G. H. Broadfoot, G. Carvalho, P. J. Hopcroft, G. Lowe, S. Nogueira, C. O’Halloran, and A. Sampaio. FDR: from theory to industrial application. In *Concurrency, Security, and Puzzles*, volume 10160 of *Lecture Notes in Computer Science*, pages 65–87. Springer, 2017.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
- [HS08] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.
- [JGKW20] D. N. Jansen, J. Friso Groote, J. J. A. Keiren, and A. Wijs. An  $O(m \log n)$  algorithm for branching bisimilarity on labelled transition systems. In *TACAS (2)*, volume 12079 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2020.
- [KS90] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes, and three problems of equivalence. *Inf. Comput.*, 86(1):43–68, 1990.

- [Lav19] M. Laveaux. Downloadable sources and benchmarks for the experimental validation. 2019. <https://doi.org/10.5281/zenodo.3449420>.
- [LGW19] M. Laveaux, J. F. Groote, and T. A. C. Willemse. Correct and efficient antichain algorithms for refinement checking. In *FORTE*, volume 11535 of *Lecture Notes in Computer Science*, pages 185–203. Springer, 2019.
- [Pav18] R. Paval. Modeling and verifying concurrent data structures. Master’s thesis, Eindhoven University of Technology, 2018. [https://research.tue.nl/files/93882157/Thesis\\_Roxana\\_Paval.pdf](https://research.tue.nl/files/93882157/Thesis_Roxana_Paval.pdf).
- [PT87] Robert Paige and Robert Endre Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.
- [Ros94] A. W. Roscoe. Model-checking CSP. In A. W. Roscoe, editor, *A Classical Mind: essays in Honour of C. A. R. Hoare*, chapter 21, pages 353–378. Prentice Hall International (UK) Ltd., 1994.
- [Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2010.
- [RT08] F. Ranzato and F. Tapparo. Generalizing the paige-tarjan algorithm by abstract interpretation. *Inf. Comput.*, 206(5):620–651, 2008.
- [SHC00] C.-H. Shann, T.-L. Huang, and C. Chen. A practical nonblocking queue algorithm using compare-and-swap. In *ICPADS 2000*, pages 470–475. IEEE Computer Society, 2000.
- [Tre86] R. K. Treiber. *Systems programming: Coping with parallelism*. International Business Machines Incorporated, Thomas J. Watson Research, 1986.
- [vBGH<sup>+</sup>17] R. van Beusekom, J. F. Groote, P. F. Hoogendijk, R. Howe, J. W. Wesselink, R. Wieringa, and T. A. C. Willemse. Formalising the Dezyne modelling language in mCRL2. In L. Petrucci, C. Secleanu, and A. Cavalcanti, editors, *FMICS-AVoCS 2017*, volume 10471 of *LNCS*, pages 217–233. Springer, 2017.
- [vG93] 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.
- [vG17] 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.
- [vG19] R. J. van Glabbeek, 2019. Personal Communication, 7 January 2019.
- [vGLT09] R. J. van Glabbeek, B. Luttik, and N. Trčka. Branching bisimilarity with explicit divergence. *Fundam. Inform.*, 93(4):371–392, 2009.
- [WDHR06] M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer, 2006.
- [WSS<sup>+</sup>12] 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.

## APPENDIX A. PROOF OF LEMMA 3.4

**Lemma 3.4.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  and states  $U \in S'$  such that  $\iota' \xrightarrow{\rho} U$ , it holds that  $\iota \xRightarrow{\rho} s$  for all  $s \in \llbracket U \rrbracket_{\mathcal{L}}$ .*

*Proof.* We use induction on the length of sequences in  $\text{Act}^*$  to prove the statement.

Base case. First,  $\iota' \xrightarrow{\epsilon} U$  iff  $U = \iota'$  by definition. The state  $\iota'$  is equal to  $\{s \mid \iota \xRightarrow{\epsilon} s\}$  as defined in the normalisation. Hence, for every state  $s \in \llbracket \iota' \rrbracket_{\mathcal{L}}$  we have  $\iota \xRightarrow{\epsilon} s$ .

Inductive case. Pick any sequence  $\rho \in \text{Act}^*$  of length  $i$  and suppose that the statement holds for all sequences in  $\text{Act}^*$  of length  $i$ . Take an arbitrary state  $V \in S'$  and action  $a \in \text{Act}$  such that  $\iota' \xrightarrow{\rho a} V$ . Then there is a state  $U \in S'$  such that  $\iota' \xrightarrow{\rho} U$  and  $U \xrightarrow{a} V$ . By definition of normalisation there is a transition  $U \xrightarrow{a} V$  if and only if  $V = \{t \in S \mid \exists s \in U : s \xrightarrow{a} t\}$ . So for all states  $t \in \llbracket V \rrbracket_{\mathcal{L}}$  there is a state  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  such that  $s \xrightarrow{a} t$ . By the induction hypothesis it holds that for all  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  there is a weak transition  $\iota \xRightarrow{\rho} s$ . But then we may conclude that  $\iota \xRightarrow{\rho a} t$  for all  $t \in \llbracket V \rrbracket_{\mathcal{L}}$ .  $\square$

## APPENDIX B. PROOF OF LEMMA 3.5

**Lemma 3.5.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  and for all states  $s \in S$  such that  $\iota \xRightarrow{\rho} s$ , there is a state  $U \in S'$  such that  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  and  $\iota' \xrightarrow{\rho} U$ .*

*Proof.* We proceed using an induction on the length of sequences in  $\text{Act}^*$ .

Base case. Let  $s \in S$  and suppose  $\iota \xRightarrow{\epsilon} s$ . Then  $s \in \llbracket \iota' \rrbracket_{\mathcal{L}}$ , since  $\iota'$  is defined as  $\{s \in S \mid \iota \xRightarrow{\epsilon} s\}$ . Moreover, we trivially have  $\iota' \xrightarrow{\epsilon} \iota'$ .

Inductive step. Pick any sequence  $\rho \in \text{Act}^*$  of length  $i$  and suppose that the statement holds for all sequences in  $\text{Act}^*$  of length  $i$ . Take an arbitrary state  $t \in S$  and action  $a \in \text{Act}$  such that  $\iota \xRightarrow{\rho a} t$ . Then there is a state  $s \in S$  such that  $\iota \xRightarrow{\rho} s$  and  $s \xrightarrow{a} t$ . Fix such a state  $s \in S$ . From the induction hypothesis it then follows that there is a state  $U \in S'$  such that  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  and  $\iota' \xrightarrow{\rho} U$ . Fix this state  $U \in S'$ . Let  $V$  be equal to  $\{t \in S \mid \exists u \in U : u \xrightarrow{a} t\}$ ; then by definition,  $U \xrightarrow{a} V$ . It follows that  $\iota' \xrightarrow{\rho a} V$ . Finally,  $t \in \llbracket V \rrbracket_{\mathcal{L}}$  follows from  $s \xrightarrow{a} t$  and  $s \in \llbracket U \rrbracket_{\mathcal{L}}$ .  $\square$

## APPENDIX C. PROOF OF LEMMA 3.6

**Lemma 3.6.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  it holds that  $\rho \notin \text{weaktraces}(\mathcal{L})$  if and only if  $\iota' \xrightarrow{\rho} \emptyset$ .*

*Proof.*

$\Rightarrow$ ) We use induction on the length of the sequences in  $\text{Act}^*$ .

Base case. The implication holds vacuously since  $\epsilon \in \text{weaktraces}(\mathcal{L})$ .

Inductive step. Pick a sequence  $\rho \in \text{Act}^*$  of length  $i$  and suppose that the implication holds for all sequences of length  $i$ . Assume an arbitrary action  $a \in \text{Act}$  such that  $\rho a \notin \text{weaktraces}(\mathcal{L})$ . From  $\rho a \notin \text{weaktraces}(\mathcal{L})$  it follows that there is no state  $t \in S$  such that  $\iota \xRightarrow{\rho a} t$ . We distinguish two cases:

- Case  $\rho \notin \text{weaktraces}(\mathcal{L})$ . From the induction hypothesis we obtain  $\iota' \xrightarrow{\rho} \emptyset$  and  $\emptyset \xrightarrow{a} \emptyset$  by definition. We may therefore also conclude  $\iota' \xrightarrow{\rho a} \emptyset$ .
  - Case  $\rho \in \text{weaktraces}(\mathcal{L})$ . Then there is a state  $s \in S$  such that  $\iota \xrightarrow{\rho} s$ . Since  $\mathcal{L}'$  is deterministic there is a unique state  $U \in S'$  such that both  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  and  $\iota' \xrightarrow{\rho} U$  by Lemmas 3.5 and 2.4. For all states  $u \in S$  satisfying  $\iota \xrightarrow{\rho} u$  (which exist as  $\rho \in \text{weaktraces}(\mathcal{L})$ ) there cannot be a state  $t \in S$  such that  $u \xrightarrow{a} t$  by the observation that  $\rho a \notin \text{weaktraces}(\mathcal{L})$ . Therefore,  $U \xrightarrow{a} \emptyset$  and thus also  $\iota' \xrightarrow{\rho a} \emptyset$ .
- $\Leftarrow$ ) Suppose  $\iota' \xrightarrow{\rho} \emptyset$ . Towards a contradiction, assume that  $\rho \in \text{weaktraces}(\mathcal{L})$ . Then there is a state  $s \in S$  such that  $\iota \xrightarrow{\rho} s$ . By Lemma 3.5, there must be some  $U \in S'$  such that  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  and  $\iota' \xrightarrow{\rho} U$ . Since  $\mathcal{L}'$  is deterministic, by Lemma 2.4, we obtain that this state  $U$  must be such that  $U = \emptyset$ .  $\square$

#### APPENDIX D. PROOF OF LEMMA 3.18

**Lemma 3.18.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}_{\text{fdr}}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  such that either  $\rho \notin \text{divergences}(\mathcal{L})$  or  $\rho \in \text{divergences}_{\min}(\mathcal{L})$  and for all states  $s \in S$  such that  $\iota \xrightarrow{\rho} s$  there is a state  $U \in S'$  such that  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  and  $\iota' \xrightarrow{\rho} U$ .*

*Proof.* Proof by induction on the length of sequences that are not divergences, or minimal divergences.

Base case. The empty trace  $\epsilon$  satisfies  $\epsilon \notin \text{divergences}(\mathcal{L})$  or  $\epsilon \in \text{divergences}_{\min}(\mathcal{L})$  by definition. Hence, we must show that for all states  $s \in S$  satisfying  $\iota \xrightarrow{\epsilon} s$  there is a state  $U \in S'$  such that  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  and  $\iota' \xrightarrow{\epsilon} U$ . We know that if  $\iota \xrightarrow{\epsilon} s$  then  $s \in \llbracket \iota' \rrbracket_{\mathcal{L}}$ , because  $\iota'$  is defined as  $\{s \in S \mid \iota \xrightarrow{\epsilon} s\}$  in the normalisation. Finally, we also know that  $\iota' \xrightarrow{\epsilon} \iota'$ .

Inductive step. Suppose that the statement holds for all sequences of length  $i$  that are either not divergences or minimal divergences of  $\mathcal{L}$ . Pick a sequence  $\rho \in \text{Act}^*$  of length  $i$ , an arbitrary state  $t \in S$  and action  $a \in \text{Act}$  such that  $\iota \xrightarrow{\rho a} t$  and  $\rho a \notin \text{divergences}(\mathcal{L})$  or  $\rho a \in \text{divergences}_{\min}(\mathcal{L})$ . Note that whenever  $\rho a \notin \text{divergences}(\mathcal{L})$  or  $\rho a \in \text{divergences}_{\min}(\mathcal{L})$  then  $\rho \notin \text{divergences}(\mathcal{L})$ . From  $\iota \xrightarrow{\rho a} t$  it follows that there is a state  $s \in S$  such that  $\iota \xrightarrow{\rho} s$  and  $s \xrightarrow{a} t$ . By our induction hypothesis it then follows that there is a state  $U \in S'$  such that  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  and  $\iota' \xrightarrow{\rho} U$ . For all states  $u \in \llbracket U \rrbracket_{\mathcal{L}}$  it holds that  $\iota \xrightarrow{\rho} u$  by Lemma 3.17, so by definition of divergences it must be that  $u \uparrow$  does not hold and hence  $\llbracket U \rrbracket_{\mathcal{L}} \uparrow$  does not hold. Let  $V$  be equal to  $\{v \in S \mid \exists u \in U : u \xrightarrow{a} v\}$  such that  $U \xrightarrow{a} V$  by definition of the normalisation. It follows that  $\iota' \xrightarrow{\rho a} V$ . Finally,  $t \in \llbracket V \rrbracket_{\mathcal{L}}$  follows from  $s \xrightarrow{a} t$  and  $s \in \llbracket U \rrbracket_{\mathcal{L}}$ .  $\square$

#### APPENDIX E. PROOF OF LEMMA 3.19

**Lemma 3.19.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}_{\text{fdr}}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  and states  $U \in S'$  it holds that if  $\iota' \xrightarrow{\rho} U$  and not  $\llbracket U \rrbracket_{\mathcal{L}} \uparrow$  then  $\rho \notin \text{divergences}(\mathcal{L})$ .*

*Proof.* Let  $\rho \in \text{Act}^*$  and  $U \in S'$  be such that  $\iota' \xrightarrow{\rho} U$  and not  $\llbracket U \rrbracket_{\mathcal{L}} \uparrow$ . Towards a contradiction, assume that  $\rho \in \text{divergences}(\mathcal{L})$ . We distinguish two cases:

- $\rho \in \text{divergences}_{\min}(\mathcal{L})$ . Let  $t \in S$  be such that  $\iota \xRightarrow{\rho} t$  and  $t \uparrow$ . Note that due to the determinism of  $\text{norm}_{\text{fdr}}(\mathcal{L})$  and Lemma 3.18, for all  $s \in S$  such that  $\iota \xRightarrow{\rho} s$ , we have  $s \in U$ . Hence also  $t \in U$ . But  $t \uparrow$  then implies  $\llbracket U \rrbracket_{\mathcal{L}} \uparrow$ . Contradiction.
- $\rho \notin \text{divergences}_{\min}(\mathcal{L})$ . Then  $\rho = \rho' \rho''$  for some  $\rho' \in \text{divergences}_{\min}(\mathcal{L})$ . Let  $t \in S$  be such that  $\iota \xRightarrow{\rho'} t$  and  $t \uparrow$ . Then, by Lemma 3.18, there must be some  $V \in S'$  such that  $\iota' \xRightarrow{\rho'} V$  and  $t \in V$ . Let  $V$  be such. Since  $t \uparrow$  and  $t \in V$ ,  $V$  has no outgoing transitions in  $\text{norm}_{\text{fdr}}(\mathcal{L})$ . In particular, we cannot have  $V \xRightarrow{\rho''} U$ , and because  $\text{norm}_{\text{fdr}}(\mathcal{L})$  is deterministic, we also cannot have  $\iota' \xRightarrow{\rho} U$ . Contradiction.  $\square$

## APPENDIX F. PROOF OF LEMMA 3.20

**Lemma 3.20.** *Let  $\mathcal{L} = (S, \iota, \rightarrow)$  be an LTS and let  $\text{norm}_{\text{fdr}}(\mathcal{L}) = (S', \iota', \rightarrow')$ . For all sequences  $\rho \in \text{Act}^*$  it holds that  $\rho \notin (\text{divergences}(\mathcal{L}) \cup \text{weaktraces}(\mathcal{L}))$  if and only if  $\iota' \xRightarrow{\rho} \emptyset$ .*

*Proof.*

$\Rightarrow$ ) Proof by induction on the length of sequences in  $\text{Act}^*$ .

Base case. The implication holds vacuously since  $\epsilon \in \text{weaktraces}(\mathcal{L})$ .

Inductive step. Suppose that the statement holds for all sequences  $\text{Act}^*$  of length  $i$ . Pick a sequence  $\rho \in \text{Act}^*$  of length  $i$ . Take an arbitrary action  $a \in \text{Act}$  such that  $\rho a \notin (\text{divergences}(\mathcal{L}) \cup \text{weaktraces}(\mathcal{L}))$ . From  $\rho a \notin \text{weaktraces}(\mathcal{L})$  it follows that there is no state  $t \in S$  such that  $\iota \xRightarrow{\rho a} t$ . From  $\rho a \notin \text{divergences}(\mathcal{L})$  it follows that  $\rho \notin \text{divergences}(\mathcal{L})$ . Now, there are two cases to distinguish:

- Case  $\rho \notin \text{weaktraces}(\mathcal{L})$ . From the induction hypothesis we obtain  $\iota' \xRightarrow{\rho} \emptyset$  and  $\emptyset \xrightarrow{a} \emptyset$  by definition. Thus  $\iota' \xRightarrow{\rho a} \emptyset$ .
- Case  $\rho \in \text{weaktraces}(\mathcal{L})$ . There is a state  $s \in S$  such that  $\iota \xRightarrow{\rho} s$ . Since  $\mathcal{L}'$  is deterministic there is a unique state  $U \in S'$  such that  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  and  $\iota' \xRightarrow{\rho} U$  by Lemma 3.18 and 2.4. We may furthermore conclude that  $\llbracket U \rrbracket_{\mathcal{L}} \uparrow$  does not hold. Because  $\rho a \notin \text{weaktraces}(\mathcal{L})$ , no state  $u \in S$  for which  $\iota \xRightarrow{\rho} u$  satisfies  $u \xrightarrow{a} t$ , for any state  $t$ . Therefore, by definition,  $U \xrightarrow{a} \emptyset$ , and thus  $\iota' \xRightarrow{\rho a} \emptyset$ .

$\Leftarrow$ ) Suppose  $\iota' \xRightarrow{\rho} \emptyset$ . From the observation that  $\llbracket \emptyset \rrbracket_{\mathcal{L}} \uparrow$  does not hold and Lemma 3.19 it follows that  $\rho \notin \text{divergences}(\mathcal{L})$ . Towards a contradiction, assume that  $\rho \in \text{weaktraces}(\mathcal{L})$ . Then there is a state  $s \in S$  such that  $\iota \xRightarrow{\rho} s$ . By Lemma 3.20, there must be some  $U \in S'$  such that  $s \in \llbracket U \rrbracket_{\mathcal{L}}$  and  $\iota' \xRightarrow{\rho} U$ . Since  $\mathcal{L}'$  is deterministic, by Lemma 2.4, we obtain that this state  $U$  must be such that  $U = \emptyset$ . Contradiction.  $\square$