On constructibility and unconstructibility of LTS operators from other LTS operators

Acta Informatica manuscript No.
(will be inserted by the editor)

On Constructibility and Unconstructibility of LTS Operators from Other LTS Operators

Antti Valmari

Received: date / Accepted: date

Abstract An LTS operator can be constructed from a set of LTS operators up to an equivalence if and only if there is an LTS expression that only contains operators from the set and whose result is equivalent to the result of the operator. In this publication this idea is made precise in the context where each LTS has an alphabet of its own and the operators may depend on the alphabets. Then the extent to which LTS operators are constructible is studied. Most, if not all, established LTS operators have the property that each trace of the result arises from the execution of no more than one trace of each of its argument LTSs, and similarly for infinite traces. All LTS operators that have this property and satisfy some other rather weak regularity properties can be constructed from parallel composition and hiding up to the equivalence that compares the alphabets, traces, and infinite traces of the LTSs. Furthermore, a collection of other miscellaneous constructibility and unconstructibility results is presented.

Keywords Process algebra · Labelled transition system · Abstract semantics

Mathematics Subject Classification (2000) 68Q85

1 Introduction

In logic, it is customary to define "$\neg$" and "$\wedge$" as fundamental operators, and introduce "$\vee$" as a derived operator via $P \vee Q :\Leftrightarrow \neg(\neg P \wedge \neg Q)$. In process algebras, it would be easier to bring results from some language such as CCS [15] to another language with different operators such as CSP [17], if the operators of the latter were constructible from the operators of the former in some suitable sense.

Since [6], there has been a large body of research on this problem. Two recent sources that both clarify the setting of the problem and cite a lot of earlier work are [9, 11]. Much of the work has paid great attention to constructs via which the operators of the language are defined, such as Structural Operational Semantics (SOS) rules [1]. For instance, a large natural class of operators was defined in [18]

Tampere University of Technology
Department of Mathematics

by specifying a particular SOS rule format, and it was shown that any language that uses only them is obtainable up to bisimilarity using the specific language introduced in [3]. In the latter, the only constant operator denotes a process that can do nothing (we call it Stop), and cyclic behaviour is obtained via recursion.

In this publication we investigate the idea and limits of constructibility in a less syntax-oriented fashion. We pay no attention to the mechanism via which the operators are defined. Instead, for us an operator is simply any "well-behaving" function that inputs zero or more labelled transition systems (LTSs) and outputs an LTS. That is, we work almost exclusively on what can be thought of as a common semantic domain of many process-algebraic languages, and thus need not talk much about the languages themselves. To emphasize this, we use the term *LTS operator*. In particular, we do not discuss recursion.

There is, however, a complication that causes us to use one language-oriented concept. When constructing functions or expressions that express an operator in terms of some other operators up to some equivalence, it is often assumed that "fresh" or "new" action names are available, that is, action names that do not occur in any of the arguments to which the operator is to be applied. To present an example, let τ denote the invisible action and $L \setminus A$ the hiding operator. Assume that the action prefix operator $a; L$ is available under the restriction that a is visible. Now $\tau; L$ can be constructed as $(a_{\mathsf{new}}; L) \setminus \{a_{\mathsf{new}}\}$, if we may assume that a_{new} does not occur in L.

While it is customary to assume a global set of visible actions that all expressions in the language use, the present author has preferred to give each LTS L an alphabet $\Sigma(L)$ of its own. In this setting, a_{new} may be obtained by looking at $\Sigma(L)$ and constructing something that is not there. However, this implies that the construction of $\tau; L$ depends on the alphabet of L (but not on other aspects of L). To make this work when LTS operators are composed together, we will define a notion of an LTS expression over a sequence of LTS variables in an obvious way. It is the language-oriented concept that we will use.

That a function from n LTSs to an LTS is "well-behaving" means that the alphabet of the result depends only on the alphabets of the arguments, and that if any of the arguments is replaced by a bisimilar one, then the result remains bisimilar with the original. This is a mild well-formedness condition on LTS operators. One of our main results makes the further assumption that the operators are "(infinite-)trace-nice", but also this assumption is formulated at the semantic level.

The inspiration for this work came from many sources. When finding the weakest deadlock-preserving congruence [20], it turned out that the interrupt operator "$\triangleright$" of the Basic Lotos language [5] is somehow different from the other operators of Basic Lotos in that the alphabet, stable failures, and initial stability suffice for the other operators, but "$\triangleright$" also requires the traces. This implies that "$\triangleright$" cannot be constructed from the other operators, if the alphabet, stable failures, and initial stability equivalence is used. On the other hand, it is easy to build "$\triangleright$" if only the alphabet and traces are used. So the notion of constructibility is relative to the equivalence that is used.

Much of the earlier work has considered constructibility up to isomorphism, bisimilarity, or some rather strong abstract equivalence, such as branching or weak bisimilarity. This publication focuses on some rather weak abstract equivalences. Those who want to apply process-algebraic results to the verification of hardware

or software (the present author included) are mainly interested in abstract equivalences. As argued in [19] and evidenced by certain verification methods (see [21], for instance), the weaker an equivalence is, the more room it leaves for developing verification algorithms that run sufficiently fast in practice (as long as it is strong enough to preserve the properties of interest). However, even then, isomorphism and especially bisimilarity are extreme useful tools. If an operator is constructible up to bisimilarity, then it is constructible up to virtually every abstract equivalence of interest.

Process algebras contain many different parallel composition operators. However, intuitively, all of them specify synchronization patterns, where some LTSs participate a joint transition via some visible action names, the remaining LTSs do not participate, and the result has some action name which may also be invisible. The development in [2] was explicitly based on such synchronization, but without the name for the resulting joint transition. The name was added in [13]. It was also showed that any synchronization pattern can be constructed up to isomorphism from ordinary parallel composition, relational renaming, and hiding.

The goal of [23] was to find all congruences that are at most as strong as the *CFFD-equivalence* [24]. The latter compares the alphabets, stable failures, divergence traces, and infinite traces of the LTSs. Outside process algebras, the Linear Temporal Logic of [14] is perhaps the most widely used formalism for specifying correctness properties of concurrent systems. CFFD-equivalence is interesting, because it is the weakest congruence that preserves both deadlocks and all "stuttering-insensitive" properties that can be expressed in that logic [12, 22]. Stuttering-insensitivity is, in essence, the Linear Temporal Logic version of the idea that τ cannot be directly observed nor synchronized with.

The publication [23] used parallel composition, relational renaming, hiding, and action prefix. The first three operators were included because of their significance in modelling systems as compositions of LTSs: they are necessary and sufficient for modelling all synchronization patterns. (Hiding and relational renaming are necessary, because they themselves are synchronization patterns with only one argument LTS.) The excuse for including action prefix was that it is in basically all process algebras. However, the reason for including it was that it was used in the proofs. Indeed, with it, it was possible to prove that if a congruence makes any distinctions between LTSs, then it never equates two LTSs which have different alphabets. Without action prefix, this nice result does not hold. As a consequence, action prefix is important in itself, and not CFFD-constructible from the other three operators.

In [17, Chapter 9], the constructibility of operators was investigated in the context of the CSP language. All "CSP-like" operators can be constructed from traditional CSP operators and a more recent operator called *throw*, using any of the standard semantics for CSP. An implementation of the construction was reported in [7]. However, the results do not apply to every widely used process operator, because CSP lacks the "initial stability" issue caused by the choice (and interrupt) operator of most other languages.

In Sections 2, 3, and 5 we recall the well-known concepts used in this publication: labelled transition systems, bisimilarity, most common LTS operators, and some abstract equivalences on LTSs. We also justify our choices of their details.

In the absence of recursion, typical process-algebraic languages only yield acyclic behaviour, because the only constant operator in them is Stop. This leads

to the question: in the absence of recursion, could more interesting results be obtained, if more constant operators (or another constant operator) were used? Section 4 gives a partial answer to this question. It shows that each finite-state LTS is constructible up to bisimilarity from parallel composition, relational renaming, hiding, and an LTS with two states and four transitions. That is, this small set of operators whose only somewhat non-standard operator is relational renaming, suffices for constructing *any* finite behaviour from (copies of) a single *very small* LTS.

In Section 6, constructibility up to an equivalence is defined formally in the LTS setting, and illustrated with several examples. A collection of results about the constructibility and unconstructibility of action prefix, choice, interrupt, and throw is presented in Section 7. Section 8 contains a proof that every LTS operator that satisfies some rather mild assumptions is constructible up to the alphabet, traces, and infinite traces from just parallel composition and hiding. The conclusions are in Section 9.

2 Labelled Transition Systems

In this section and Sections 3 and 5 we recall many widely known concepts that will be used in this publication. The details of some of them have varied in the literature. Where appropriate, we will comment on our choice of the details. In particular, we will make explicit some principles that are obeyed when building the definitions.

Let A be any set. The sets of *finite* and *infinite sequences* of elements of A are denoted with A^* and A^ω, respectively. Let $\sigma \in A^* \cup A^\omega$ and $\rho \in A^* \cup A^\omega$. That σ is a prefix of ρ is denoted with $\sigma \sqsubseteq \rho$, and proper prefix is denoted with $\sigma \sqsubset \rho$. The *empty sequence* is denoted with ε. That is, if $\sigma \in A^*$ and $\xi \in A^\omega$, then $\varepsilon \sqsubseteq \varepsilon\sigma = \sigma\varepsilon = \sigma$ and $\varepsilon \sqsubseteq \varepsilon\xi = \xi$, but $\xi\varepsilon$ is undefined, because ξ has no right end. The *invisible action* is denoted with τ. We have $\tau \neq \varepsilon$. An *alphabet* is any set that contains neither ε nor τ.

A *labelled transition system*, abbreviated *LTS*, is a quadruple $(S, \Sigma, \Delta, \hat{s})$ such that Σ is an alphabet, $\Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S$, and $\hat{s} \in S$. The elements of S, Σ, and Δ are called *states*, *visible actions*, and *transitions*, respectively, and $\hat{s}$ is the *initial state*. If (s, a, s') is a transition, then s is its *tail*, a is its *label*, and s' is its *head*. A transition is (in)visible if and only if its label is (in)visible. We adopt the convention that unless otherwise stated, the components of L_1 are S_1, Σ_1, Δ_1, and $\hat{s}_1$, and similarly with L_2, L', and so on. Furthermore, unless otherwise stated, the alphabet of an LTS presented as a picture is the set of the non-τ labels of the transitions in the picture.

An LTS represents the behaviour of some system or subsystem. The behaviour starts at the initial state and consists of moving along the transitions. If the label a of a transition is visible (that is, $a \neq \tau$), then the environment of the LTS sees an occurrence of a when the LTS moves along the transition. Furthermore, we will see later that if $a \neq \tau$, the environment may even be able to temporarily or permanently refuse a, that is, prevent the LTS from moving along any transition labelled by a.

The treatment of the alphabets of LTSs varies significantly in the literature. In our case, the alphabet can be thought of as the set of the wires via which

the LTS communicates with its environment. We will see later that an LTS can prevent its neighbour LTSs from executing those, and only those, visible actions that are listed in its alphabet. This makes the alphabet essential. Consequently, all equivalences between LTSs defined in this publication check that the LTSs have the same alphabet. We have already mentioned in Section 1, and will briefly mention again in Section 9, an advantage of this approach.

Furthermore, we consider the alphabet as a structural entity that does not depend on the behaviours of LTSs. So we adopt the principle that when a system or subsystem is built as a composition of one or more LTSs, the alphabet of the result depends only on the alphabets, and does not depend on the states and transitions and initial states, of the constituent LTSs. This idea is formalized by the notion defined below.

Definition 1 Let f be an n-ary function from LTSs to LTSs. We say that f is *alphabet-consistent*, if and only if there is an n-ary function f_Σ from alphabets to alphabets such that for every L_1, ..., L_n and L, if $L = f(L_1, \ldots, L_n)$, then $\Sigma = f_\Sigma(\Sigma_1, \ldots, \Sigma_n)$.

Let us continue defining background concepts. Let L be an LTS, $s \in S$, $s' \in S$, and $a_i \in \Sigma \cup \{\tau\}$ for $1 \leq i$. Then $s -a_1 a_2 \cdots a_i \rightarrow s'$ is defined by (1) $s -\varepsilon \rightarrow s$ and (2) $s -a_1 a_2 \cdots a_{i+1} \rightarrow s'$ if and only if there is $s'' \in S$ such that $s -a_1 a_2 \cdots a_i \rightarrow s''$ and $(s'', a_{i+1}, s') \in \Delta$. That is, there is a path from s to s' such that the sequence of labels along the path is $a_1 a_2 \cdots a_i$. A similar case with an infinite path is denoted with $s -a_1 a_2 \cdots \rightarrow$. The notation $s -a_1 a_2 \cdots a_i \rightarrow$ means that there is some s' such that $s -a_1 a_2 \cdots a_i \rightarrow s'$.

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

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

Let "$\approx$" be an equivalence relation on LTSs.[1] We say that an n-ary function f from LTSs to LTSs *respects* "$\approx$" if and only if "$\approx$" is a congruence with respect to f, that is, for every L_1, L_1', ..., L_n, L_n', if $L_1 \approx L_1'$, ..., and $L_n \approx L_n'$, then $f(L_1, \ldots, L_n) \approx f(L_1', \ldots, L_n')$. We say that "$\approx$" *preserves* a property of LTSs if and only if for every L and L', if $L \approx L'$, then L and L' have the same value of the property. For instance, "$\approx$" preserves the alphabet if and only if for every L and L', $L \approx L'$ implies $\Sigma = \Sigma'$.

Often in mathematics, for technical reasons, a definition contains information that is irrelevant for the ultimately intended concept. In the case of LTSs, the names of states are such information. Drawings that represent LTSs usually lack them. However, the formal definition uses names, because otherwise it would be too difficult to define transitions. Often in mathematics, isomorphism is used to abstract away from this kind of superfluous information. That is, the ultimately intended concept is the equivalence classes induced by isomorphism. Often in mathematics, the same name is used for the originally defined and the ultimately intended concept.

[1] For any binary relation symbol $\propto$, by "$\propto$" we refer to the set of pairs that specifies the relation, while $\propto$ is used in the claims that the relation holds. That is, "$\propto$" $= \{(x, y) \mid x \propto y\}$. The benefit of this convention is illustrated by comparing $< \cup = = \leq$ to "$<$" $\cup$ "$=$" $=$ "$\leq$". More generally, we use " and " whenever we believe that it helps to avoid confusion.

In the case of LTSs, especially when the emphasis is on their behaviours, it is common to use an equivalence known as bisimilarity instead of isomorphism. Unlike isomorphism, bisimilarity may equate two LTSs that have a different number of states. Consistently with the above discussion on alphabets, our variant of bisimilarity is the following.

Definition 2 LTSs L_1 and L_2 are *bisimilar*, denoted with $L_1 \equiv L_2$, if and only if there is a relation "$\sim$" $\subseteq S_1 \times S_2$ such that

1. $\Sigma_1 = \Sigma_2$,
2. $\hat{s}_1 \sim \hat{s}_2$, and
3. for every $s_1 \in S_1$, $s_2 \in S_2$, $s_1' \in S_1$, $s_2' \in S_2$, and $a \in \Sigma_1 \cup \{\tau\}$ such that $s_1 \sim s_2$,
   (a) if $(s_1, a, s_1') \in \Delta_1$, then there is an s' such that $s_1' \sim s'$ and $(s_2, a, s') \in \Delta_2$, and
   (b) if $(s_2, a, s_2') \in \Delta_2$, then there is an s' such that $s' \sim s_2'$ and $(s_1, a, s') \in \Delta_1$.

The relation "$\sim$" is a *bisimulation*.

To summarize, our object of study is not LTSs as such, but their bisimilarity equivalence classes. For this to work, LTS operators will be defined such that they respect "$\equiv$", and the LTS properties that we discuss will be defined such that "$\equiv$" preserves them.

Isomorphism between LTSs is defined like bisimilarity with the additional requirement that "$\sim$" is a bijection between S_1 and S_2. Isomorphism implies bisimilarity, but not necessarily vice versa.

As has already been mentioned, an LTS represents the behaviour of some (sub)system, starting at the initial state and progressing along the transitions. As a consequence, only the reachable part of any LTS is important. The following lemma formalizes this idea.

Lemma 1 *If L is an LTS and L^r is its reachable part, then $L \equiv L^r$.*

Proof By construction, L and L^r have the same alphabet and the same initial state. If s is reachable and $(s, a, s') \in \Delta$, then also s' is reachable. It is easy to check from these that the relation "$\sim$" $= \{(s, s) \mid s \in S^r\}$ is a bisimulation. □

3 LTS Operators and LTS Expressions

The notion of LTS operator is essential for this publication. In this section we first define the notion, then recall the definitions of many well-known LTS operators, and finally define LTS expressions. The definition below extends the usual notion of an operator with two conditions that enforce two principles mentioned in Section 2.

Definition 3 An n-ary *non-parametric LTS operator* is a function that inputs n LTSs, yields an LTS, respects "$\equiv$", and is alphabet-consistent. An n-ary *LTS operator* is a function that inputs zero or more alphabets, action names, and/or other entities that are not LTSs, and yields an n-ary non-parametric LTS operator. An *LTS constant family* is a function that inputs zero or more alphabets and yields an LTS.

Each LTS is a 0-ary function on LTSs. It is trivially alphabet-consistent and respects every equivalence on LTSs, including "$\equiv$". So it qualifies as a 0-ary non-parametric LTS operator. As a consequence, each LTS constant family is a 0-ary LTS operator.

To simplify the construction of disjoint unions, we introduce notation for attaching an index to a symbol or to every element of a set. In the definition, $i \in \mathbb{N}$, a is any symbol, and A is any set.

- $a^{[i]} = (a, i)$, and
- $A^{[i]} = \{a^{[i]} \mid a \in A\}$.

To avoid irrelevant technical problems, we assume that for every a and i, $\tau \neq a^{[i]} \neq \varepsilon$.

Next we define some particular LTS operators. The definitions are non-standard in that they are written in terms of LTSs. However, it is easy to see that they yield essentially the same operators as the standard SOS definitions yield. It is obvious from the definitions that the operators are alphabet-consistent. (More precisely, for each choice of their parameters, the resulting non-parametric LTS operators are alphabet-consistent.) The proofs that the operators respect "$\equiv$" are long and dull, so we skip them. For most of the operators, it is widely known that they respect "$\equiv$".

Definition 4 Let L, L_1, and L_2 be LTSs, A be an alphabet, a be such that $a \neq \varepsilon$, ϕ be a partial function such that $\phi(\tau)$ is undefined or τ, and Φ be a set of pairs such that for every $(a, b) \in \Phi$ we have $\tau \neq a \neq \varepsilon$ and $\tau \neq b \neq \varepsilon$.

stop $\mathsf{Stop}_A = (\{0\}, A, \emptyset, 0)$ is the LTS with one state, no transitions, and the alphabet A. Because different but bisimilar LTSs are considered as different representations of the same abstract object, the name of the only state is not important. Here the name "0" is used. We have $\mathsf{Stop}_\emptyset = \;\searrow\!\circ$.

τ-loop $\tau\text{-}\mathsf{loop}_A = (\{0\}, A, \{(0, \tau, 0)\}, 0)$ is the LTS with one state, a τ-transition from the only state to itself, and the alphabet A. We have $\tau\text{-}\mathsf{loop}_\emptyset = \;\searrow\!\circlearrowleft\tau$.

hiding [17,5] $L \setminus A = (S, \Sigma', \Delta', \hat{s})$, where $\Sigma' = \Sigma \setminus A$ and $\Delta' = \{(s, a, s') \in \Delta \mid a \notin A\} \cup \{(s, \tau, s') \mid \exists a \in A : (s, a, s') \in \Delta\}$. That is, labels of transitions that are in A are replaced by τ and removed from the alphabet. Other labels of transitions are not affected. (The requirement $\tau \notin A$ could be removed, because it is easy to check from the definition that then $L \setminus (A \cup \{\tau\}) = L \setminus A$.)

(functional) renaming $\phi(L) = (S, \Sigma', \Delta', \hat{s})$, where $\phi_\Sigma(a) = \phi(a)$ if $\phi(a)$ is defined, $\phi_\Sigma(a) = a$ if $\phi(a)$ is not defined and $a \in \Sigma \cup \{\tau\}$, $\Sigma' = \{\phi_\Sigma(a) \mid a \in \Sigma\}$, and $\Delta' = \{(s, \phi_\Sigma(a), s') \mid (s, a, s') \in \Delta\}$. That is, ϕ renames each visible action to precisely one visible action. For convenience, ϕ has been defined such that actions that are mapped to themselves need not be specified. The next operator is a generalization of ϕ.

relational renaming [17] $L\Phi = (S, \Sigma', \Delta', \hat{s})$, where $\Sigma' = \{b \mid \exists a \in \Sigma : (a, b) \in \Phi\} \cup \{a \in \Sigma \mid \neg\exists b : (a, b) \in \Phi\}$ and $\Delta' = \{(s, b, s') \mid \exists a : (s, a, s') \in \Delta \wedge (a, b) \in \Phi\} \cup \{(s, a, s') \in \Delta \mid \neg\exists b : (a, b) \in \Phi)\}$. That is, Φ renames visible actions to visible actions. A visible action may be renamed to more than one visible action. In that case, the transitions labelled by that action are duplicated as needed. If Φ specifies no new names for an action, the transitions labelled by it remain unchanged. In particular, τ-transitions remain unchanged. The alphabet of the result consists of the unchanged and renamed original visible

actions. Pairs in Φ whose first component is not in Σ have no effect. This design makes it simple to specify the intended changes without causing accidental removal of the transitions that are not intended to change.

parallel composition $L_1 \| L_2$ is the reachable part of $(S, \Sigma, \Delta, \hat{s})$, where $S = S_1 \times S_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $\hat{s} = (\hat{s}_1, \hat{s}_2)$, and $((s_1, s_2), a, (s'_1, s'_2)) \in \Delta$ if and only if
- $(s_1, a, s'_1) \in \Delta_1$, $s'_2 = s_2 \in S_2$, and $a \notin \Sigma_2$,
- $(s_2, a, s'_2) \in \Delta_2$, $s'_1 = s_1 \in S_1$, and $a \notin \Sigma_1$, or
- $(s_1, a, s'_1) \in \Delta_1$, $(s_2, a, s'_2) \in \Delta_2$, and $a \in \Sigma_1 \cap \Sigma_2$.

That is, a parallel composition models co-operation of its component LTSs. If a belongs to the alphabets of both components, then an a-transition of the parallel composition consists of simultaneous a-transitions of both components. If a belongs to the alphabet of one but not the other component, then that component may make an a-transition while the other component stays in its current state. Also each τ-transition of the parallel composition consists of one component making a τ-transition without the other participating. The result of the parallel composition is pruned by only taking the reachable part. This convention helps the presentation of examples by often significantly reducing the sizes of parallel compositions, while by Lemma 1 it does not affect the interesting properties of LTSs.

action prefix [5,15,17] $a; L = (S', \Sigma', \Delta', \hat{s}')$, where $\hat{s}' \notin S$ (for instance, $\hat{s}' = S$), $S' = S \cup \{\hat{s}'\}$, $\Sigma' = \Sigma$ if $a = \tau$ and otherwise $\Sigma' = \Sigma \cup \{a\}$, and $\Delta' = \Delta \cup \{(\hat{s}', a, \hat{s})\}$. That is, $a; L$ adds an a-transition to the front of L, adding a to the alphabet if necessary. Obviously $\tau; L \equiv (a; L) \setminus \{a\}$, if a is chosen such that $a \notin \Sigma \cup \{\tau, \varepsilon\}$ (for instance, $a = \Sigma \cup \{\tau, \varepsilon\}$).

choice [15,5] $L_1 + L_2 = (S, \Sigma, \Delta, \hat{s})$, where $S = \{0^{[0]}\} \cup S_1^{[1]} \cup S_2^{[2]}$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $\Delta = \Delta'_1 \cup \Delta'_2 \cup \{(\hat{s}, a, s^{[1]}) \mid (\hat{s}_1, a, s) \in \Delta_1\} \cup \{(\hat{s}, a, s^{[2]}) \mid (\hat{s}_2, a, s) \in \Delta_2\}$, $\Delta'_i = \{(s^{[i]}, a, s'^{[i]}) \mid (s, a, s') \in \Delta_i\}$ for $i \in \{1, 2\}$, and $\hat{s} = 0^{[0]}$. That is, the result consists of disjoint instances of L_1 and L_2, augmented with a new initial state and a copy of each initial transition of L_1 and L_2 that starts at the new initial state. Any first transition of $L_1 + L_2$ thus mimics either a first transition of L_1 or of L_2, after which $L_1 + L_2$ continues like the LTS whose first transition was mimicked. Every first transition of L_1 and L_2 can be mimicked.

interrupt [5] $L_1 \rhd L_2 = (S, \Sigma, \Delta, \hat{s})$, where $S = S_1^{[1]} \cup S_2^{[2]}$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $\Delta = \Delta'_1 \cup \Delta'_2 \cup \{(s^{[1]}, a, s'^{[2]}) \mid s \in S_1 \wedge (\hat{s}_2, a, s') \in \Delta_2\}$, Δ'_1 and Δ'_2 are like above, and $\hat{s} = \hat{s}_1^{[1]}$. That is, $L_1 \rhd L_2$ behaves first like L_1, but may at any instant of time start behaving like L_2. If the initial state of L_2 is the tail of only visible transitions, then the environment can prevent $L_1 \rhd L_2$ from becoming L_2.

throw [17] $L_1 \,\Theta_A\, L_2 = (S, \Sigma, \Delta, \hat{s})$, where $S = S_1^{[1]} \cup S_2^{[2]}$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $\Delta = \Delta_{1,1} \cup \Delta_{1,2} \cup \Delta'_2$, $\Delta_{1,1} = \{(s^{[1]}, a, s'^{[1]}) \mid (s, a, s') \in \Delta_1 \wedge a \notin A\}$, $\Delta_{1,2} = \{(s^{[1]}, a, \hat{s}_2^{[2]}) \mid \exists s' : (s, a, s') \in \Delta_1 \wedge a \in A\}$, Δ'_2 is like above, and $\hat{s} = \hat{s}_1^{[1]}$. That is, $L_1 \,\Theta_A\, L_2$ behaves like L_1 as long as L_1 has not executed any action from A. From then on $L_1 \,\Theta_A\, L_2$ behaves like L_2. If $\tau \neq a \neq \varepsilon$, then $a; L \equiv \diagdown\!\circ\!\xrightarrow{a}\!\circ\!\Theta_{\{a\}} L$. In this sense, "$\Theta$" is a generalization of ";".

In the above definition, only "$\|$", "$+$", and "$\rhd$" are non-parametric LTS operators. For instance, $L \setminus A$ has the parameter A.

The definition of "$\|$" makes the alphabets important.

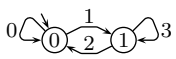

Fig. 1 The LTS Stem.

It is easy to check that $(L_1 \parallel L_2) \parallel L_3$ is isomorphic with and thus bisimilar with $L_1 \parallel (L_2 \parallel L_3)$, so we can treat "$\parallel$" as associative and just write $L_1 \parallel L_2 \parallel L_3$. For a similar reason "$\parallel$" can be treated as commutative, and "$+$" can be treated as associative and commutative. To further (but not fully) reduce the need for parentheses, we declare that the binary operators "$\parallel$", "$+$", "$\rhd$", and "Θ" have lower precedence than the unary operators "$\backslash$", "Φ", and ";". So $a; L_1 \parallel L_2 \setminus A\Phi$ means the same as $(a; L_1) \parallel ((L_2 \setminus A)\Phi)$, where the correct parsing of $L_2 \setminus A\Phi$ arises from the fact that $(A\Phi)$ does not mean anything.

An *LTS variable* is a symbol that ranges over LTSs. We have been using L, L', L_1, and so on as LTS variables. Stop_A is not an LTS variable but an LTS constant family, because, for each A, it denotes a particular fixed LTS.

Definition 5 Let $L_1, \ldots, L_n$ be LTS variables. An *LTS expression over* $L_1, \ldots, L_n$ is any L_i $(1 \le i \le n)$ or an application of an m-ary LTS operator to m LTS expressions over $L_1, \ldots, L_n$, where the operator may be a function of the alphabets $\Sigma_1, \ldots, \Sigma_n$ of $L_1, \ldots, L_n$, but may not depend on other aspects of $L_1, \ldots, L_n$.

An LTS expression over $L_1, \ldots, L_n$ represents a function that inputs n LTSs and yields an LTS. For instance, a certain 2-ary function $f(L_1, L_2)$ can be represented by the LTS expression $(L_1 \| L_2) \setminus \Sigma_1 + \tau; L_1$. The next lemma says that the function represented by an LTS expression is an LTS operator, and it inherits the common congruence properties of its ingredients.

Lemma 2 *Every function represented by an LTS expression is alphabet-consistent. If "$\approx$" preserves the alphabet and every LTS operator in the expression respects "$\approx$", then the function respects "$\approx$". In particular, the function respects "$\equiv$".*

Proof Let the expression be $expr(L_1, \ldots, L_n)$. Fix $\Sigma_1, \ldots, \Sigma_n$ and all other parameters of all operators in the expression to arbitrary values. The operators become non-parametric, and the expression becomes a function $expr_{\Sigma_1,\ldots,\Sigma_n}(L_1, \ldots, L_n)$ from subsets of LTSs, where each L_i ranges over the LTSs whose alphabet is Σ_i. Applying the alphabet-consistency of each operator bottom-up yields that the expression as a whole is alphabet-consistent. Because $L_i \approx L_i'$ implies $\Sigma_i = \Sigma_i'$, the application also reveals that if $L_i \approx L_i'$ for $1 \le i \le n$, then $expr(L_1, \ldots, L_n) = expr_{\Sigma_1,\ldots,\Sigma_n}(L_1, \ldots, L_n) \approx expr_{\Sigma_1,\ldots,\Sigma_n}(L_1', \ldots, L_n') = expr(L_1', \ldots, L_n')$.

By Definitions 2 and 3, "$\equiv$" qualifies as "$\approx$". □

4 Constructing All Finite-State LTSs from One

Let Stem be the LTS in Figure 1. This section is devoted to the proof of the following theorem. The idea behind the choice of Stem is that it is the smallest LTS with which the theorem is known to hold.

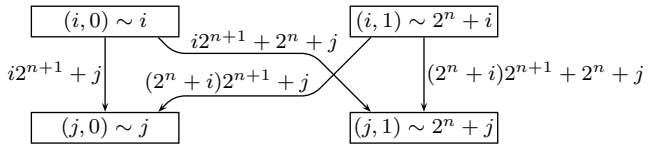

Fig. 2 Illustrating the induction step of the proof of Theorem 1.

Theorem 1 *For every LTS L with a finite number of states, there is an LTS expression expr built only from "$\|$", "Φ", "$\setminus$", and* Stem *such that $L \equiv expr$.*

Proof We prove first by induction that for every $n > 0$, the LTS L_n with $S_n = \{0, 1, \ldots, 2^n - 1\}$, $\Sigma_n = \{0, 1, \ldots, 2^{2n} - 1\}$, $\Delta_n = \{(i, i2^n + j, j) \mid i \in S_n \wedge j \in S_n\}$, and $\hat{s}_n = 0$ can be constructed. This LTS has 2^n states. For any pair of states, it has a transition from the first state to the second with a unique label.

The base case of the induction holds trivially, because $L_1 = \mathsf{Stem}$. (We start with $n = 1$ instead of $n = 0$, because the constructibility of L_0 is not given in the assumptions of the theorem.)

The induction step is illustrated in Figure 2. It is obtained via $L_{n+1} := L_n\Phi_n \,\|\, L_1\Phi'_n$, where states $(i, 0)$ and $(i, 1)$ of the parallel composition are given the numbers i and $2^n + i$, respectively, and the renaming operators are the following:

- Φ_n maps $i2^n + j$ to the following four values:
  - $i2^{n+1} + j$,
  - $i2^{n+1} + 2^n + j$,
  - $(2^n + i)2^{n+1} + j$, and
  - $(2^n + i)2^{n+1} + 2^n + j$.
- Φ'_n maps, for every $i \in S_n$ and $j \in S_n$,
  - 0 to $i2^{n+1} + j$,
  - 1 to $i2^{n+1} + 2^n + j$,
  - 2 to $(2^n + i)2^{n+1} + j$, and
  - 3 to $(2^n + i)2^{n+1} + 2^n + j$.

Next we show how to construct an LTS that is bisimilar with Stop_Σ. The idea is to put two copies of Stem in parallel such that the labels of one copy have been renamed so that the transitions from the initial states do not synchronize. Then the alphabet of the result is adjusted by hiding the unnecessary action names and converting the remaining action name to the desired set.

$$\mathsf{Stop}_\Sigma \equiv (\ (\mathsf{Stem}\{(0,2),(1,3),(2,0),(3,1)\} \,\|\, \mathsf{Stem}) \setminus \{1,2,3\}\ )\{\ (0,a) \mid a \in \Sigma\ \}$$

Consider any LTS $(S, \Sigma, \Delta, \hat{s})$ with at most 2^n states. It is isomorphic to some LTS of the form $(S', \Sigma, \Delta', 0)$ with $S' = \{0, 1, \ldots, |S| - 1\}$. It, in turn, is bisimilar with

$$((L_n \,\|\, \mathsf{Stop}_A) \setminus A)\Phi \setminus \{\iota\} \,\|\, \mathsf{Stop}_B\ ,$$

where

- ι is a symbol such that $\iota \notin \Sigma \cup \{\varepsilon, \tau\}$ (we may choose $\iota = \Sigma \cup \{\varepsilon, \tau\}$),
- $A = \{i2^n + j \mid \forall a : (i, a, j) \notin \Delta'\}$,
- $\Phi = \{(i2^n + j, a) \mid (i, a, j) \in \Delta' \wedge a \neq \tau\} \cup \{(i2^n + j, \iota) \mid (i, \tau, j) \in \Delta'\}$, and
- $B = \Sigma \setminus \{a \mid \exists i : \exists j : (i, a, j) \in \Delta'\}$.

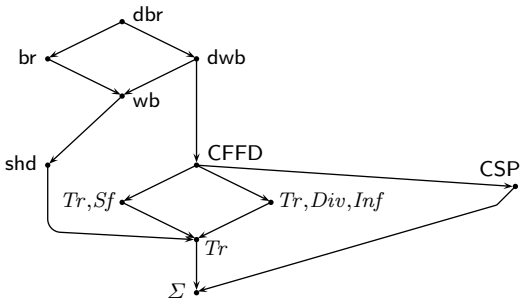

Fig. 3 Implications between some equivalences.

Here $\| \mathsf{Stop}_A$ prunes the transitions of L_n that are not needed, $\setminus A$ removes their labels from the alphabet, $\Phi \setminus \{\iota\}$ takes the desired copies of the remaining transitions, and $\| \mathsf{Stop}_B$ extends the alphabet with the labels in Σ that are not introduced by Φ. □

5 Abstract Equivalences on LTSs

Often the behaviours of LTSs are compared according to some *abstract equivalence*. Unlike bisimilarity, an abstract equivalence pays no attention to occurrences of invisible transitions as such, although it may pay attention to their consequences. In this section we introduce the abstract equivalences used in the rest of this publication.

To help to relate the equivalences to each other, Figure 3 shows implications between many of them. An arrow from "$\approx_1$" to "$\approx_2$" means that for every L and L', $L \approx_1 L'$ implies $L \approx_2 L'$. The bottom node Σ denotes the equivalence that only checks that $\Sigma = \Sigma'$.

The notation $s =\varepsilon\Rightarrow s'$ means that there is a path from s to s' such that each label along the path is τ. Because a single state qualifies as a path of length zero, $s =\varepsilon\Rightarrow s$ holds trivially. If $a \in \Sigma$, then $s =a\Rightarrow s'$ denotes that there are s_0 and s_1 such that $s =\varepsilon\Rightarrow s_0 -a\rightarrow s_1 =\varepsilon\Rightarrow s'$. This notation extends to elements of Σ^* and Σ^ω in the natural way. For instance, $s =a_1a_2\cdots\Rightarrow$ if and only if there are s_1, s_2, ... such that $s =a_1\Rightarrow s_1 =a_2\Rightarrow s_2 =a_3\Rightarrow \cdots$. By $s -\tau^\omega\rightarrow$ we mean that there is an infinite path that starts at s and each label along the path is τ.

We will use the following sets and the following predicate on LTSs. The sets are sometimes called *semantic sets*.

traces
$Tr(L) = \{\sigma \in \Sigma^* \mid \hat{s} =\sigma\Rightarrow\}$,
infinite traces
$Inf(L) = \{\xi \in \Sigma^\omega \mid \hat{s} =\xi\Rightarrow\}$,
initial stability
if $\hat{s} -\tau\rightarrow$, then $Stb(L) = \mathsf{false}$, otherwise $Stb(L) = \mathsf{true}$,
stable failures
$Sf(L) = \{(\sigma, A) \in \Sigma^* \times 2^\Sigma \mid \exists s : \hat{s} =\sigma\Rightarrow s \wedge \forall a \in A \cup \{\tau\} : \neg(s -a\rightarrow)\}$,
divergence traces
$Div(L) = \{\sigma \in \Sigma^* \mid \exists s : \hat{s} =\sigma\Rightarrow s -\tau^\omega\rightarrow\}$, and

minimal divergence traces
$minD(L) = \{\sigma \in Div(L) \mid \neg\exists \rho \in Div(L) : \rho \sqsubset \sigma\}$.

The set A in the definition of $Sf(L)$ is called a *refusal set*. When in s, L *refuses* it.

For every LTS we have

- $\varepsilon \in Tr(L)$, and
- If $\rho \sqsubseteq \sigma \in Tr(L)$, then $\rho \in Tr(L)$.

Trace equivalence, *stable failures equivalence*, and *CFFD-equivalence* without and with initial stability are defined by

- $L \approx_{Tr} L'$ if and only if $\Sigma = \Sigma'$ and $Tr(L) = Tr(L')$,
- $L \approx_{Sf} L'$ if and only if $\Sigma = \Sigma'$ and $Sf(L) = Sf(L')$,
- $L \approx_{\mathsf{CFFD}} L'$ if and only if $\Sigma = \Sigma'$, $Sf(L) = Sf(L')$, $Div(L) = Div(L')$, and $Inf(L) = Inf(L')$ [24], and
- $L \approx_{\mathsf{CFFD},Stb} L'$ if and only if $L \approx_{\mathsf{CFFD}} L'$ and $Stb(L) = Stb(L')$ [24].

In general, if X and Y are functions on LTSs, the notation "$\approx_{X,Y}$" is defined by $L \approx_{X,Y} L'$ if and only if $\Sigma = \Sigma'$, $X(L) = X(L')$, and $Y(L) = Y(L')$. The notation obviously extends to one or more than two functions on LTSs.

Although "$\approx_{\mathsf{CFFD}}$" does not explicitly compare traces, $L \approx_{\mathsf{CFFD}} L'$ implies $Tr(L) = Tr(L')$. This is because $Tr(L) = Div(L) \cup \{\sigma \mid (\sigma, \emptyset) \in Sf(L)\}$.

We will also use weak bisimilarity [15] and (divergence-preserving) branching bisimilarity [10].

Definition 6 LTSs L_1 and L_2 are *weakly bisimilar*, denoted with $L_1 \approx_{\mathsf{wb}} L_2$, if and only if there is a relation "$\sim$" $\subseteq S_1 \times S_2$ such that

1. $\Sigma_1 = \Sigma_2$,
2. $\hat{s}_1 \sim \hat{s}_2$, and
3. for every $s_1 \in S_1$, $s_2 \in S_2$, $s_1' \in S_1$, $s_2' \in S_2$, and $a \in \Sigma_1 \cup \{\varepsilon\}$ such that $s_1 \sim s_2$,
   (a) if $s_1 =a\Rightarrow s_1'$ in L_1, then there is s' such that $s_1' \sim s'$ and $s_2 =a\Rightarrow s'$ in L_2, and
   (b) if $s_2 =a\Rightarrow s_2'$ in L_2, then there is s' such that $s' \sim s_2'$ and $s_1 =a\Rightarrow s'$ in L_1.

The relation "$\sim$" is a *weak bisimulation*. *Branching bisimilarity*, denoted with "$\approx_{\mathsf{br}}$", is defined otherwise similarly, but part 3(a) of the definition is replaced by the following, and its symmetric version replaces 3(b):

if $s_1 -a\rightarrow s_1'$ in L_1, then
- there are s'' and s' such that $s_1 \sim s''$, $s_1' \sim s'$, and $s_2 =\varepsilon\Rightarrow s'' -a\rightarrow s'$ in L_2 or
- $a = \tau$ and $s_1' \sim s_2$.

Divergence-preserving branching bisimilarity, denoted with "$\approx_{\mathsf{dbr}}$", adds the requirement (and its symmetric version) that each infinite sequence of τ-transitions is simulated via "$\sim$" by an infinite sequence of τ-transitions.

Figure 3 shows two equivalences whose precise definitions we will not need: "dwb" and "CSP". "dwb" is obtained from weak bisimilarity by adding the requirement that "$\sim$" may not relate a diverging and non-diverging state to each other. "CSP" refers to the failures-divergences equivalence of Communicating Sequential Processes [17].

Fig. 4 Two "$\approx_{\mathsf{CFFD},Stb}$"-equivalent but not "$\approx_{\mathsf{shd}}$"-equivalent LTSs.

Finally, we will discuss the *fair testing* or *should testing* equivalence of [16]. A *tree failure* of an LTS consists of a trace and a set of finite sequences of visible actions such that after executing the trace, the LTS may be in a state where it *refuses* the set, that is, the LTS cannot execute any sequence in the set to completion. The set is called a *refusal set*. It cannot contain ε, because ε can always be executed by doing nothing. Therefore, the set is a subset of Σ^+, that is, the set of nonempty finite sequences of visible actions.

Comparison of tree failures as such yields an equivalence that, for instance, distinguishes between $E_1 = a;b_1;\searrow\circ + a;b_2;\searrow\circ$ and $E_2 = \tau;a;b_1;\searrow\circ + \tau;a;b_2;\searrow\circ$, since only the latter can refuse $\{ab_2\}$ after ε. Should testing aims at a weaker equivalence. Should testing obtains this goal by using a more complicated comparison.

Definition 7 The set of *tree failures* of an LTS L is

$$Tf(L) \;=\; \{(\sigma, K) \in \Sigma^* \times 2^{\Sigma^+} \mid \exists s : \hat{s} = \sigma \Rightarrow s \wedge \forall \rho \in K : \neg(s = \rho \Rightarrow)\} \;.$$

We define $L_1 \approx_{\mathsf{shd}} L_2$ if and only if $\Sigma_1 = \Sigma_2$ and

- if $(\sigma, K) \in Tf(L_1)$, then either $(\sigma, K) \in Tf(L_2)$ or there are ρ and δ such that $\rho\delta \in K$ and $(\sigma\rho, K') \in Tf(L_2)$, where $K' = \{\pi \mid \rho\pi \in K\}$, and
- the same with the roles of L_1 and L_2 swapped.

We define $L_1 \approx_{\mathsf{shd},Stb} L_2$ if and only if $L_1 \approx_{\mathsf{shd}} L_2$ and $Stb(L_1) = Stb(L_2)$.

The case $(\sigma, K) \in Tf(L_2)$ is needed to make the definition work when $K = \emptyset$. When $K \neq \emptyset$, it arises from the other case by choosing $\rho = \varepsilon$. When applying the definition, we may assume that $\rho \neq \varepsilon$, because the opposite case is covered by $(\sigma, K) \in Tf(L_2)$.

Clearly $Tf(E_1) \subseteq Tf(E_2)$. In the opposite direction, $Tf(E_2) \setminus Tf(E_1) = \{(\varepsilon, \{ab_1\}), (\varepsilon, \{ab_2\})\}$. With $(\varepsilon, \{ab_1\})$, by choosing $\rho = a$ and $\delta = b_1$ we get $(\sigma\rho, K') = (a, \{b_1\}) \in Tf(E_1)$. A similar claim applies to $(\varepsilon, \{ab_2\})$. So $E_1 \approx_{\mathsf{shd}} E_2$.

Obviously $\sigma \in Tr(L)$ if and only if $(\sigma, \emptyset) \in Tf(L)$. From this it is easy to see that if $L_1 \approx_{\mathsf{shd}} L_2$, then $L_1 \approx_{Tr} L_2$. The LTSs in Figure 4 demonstrate that the opposite does not hold. Indeed, they show that not even $L_1 \approx_{\mathsf{CFFD},Stb} L_2$ implies $L_1 \approx_{\mathsf{shd}} L_2$. After ε, the LTS on the left hand side can refuse $a^*b = \{a^n b \mid n \in \mathbb{N}\}$. For any $n \in \mathbb{N}$, we may choose $\rho = a^n$ yielding $K' = a^*b$, or $\rho = a^n b$ yielding $K' = \{\varepsilon\}$. However, the LTS on the right hand side can refuse neither a^*b after a^n nor $\{\varepsilon\}$ after $a^n b$. On the other hand, "$\approx_{\mathsf{shd}}$" and "$\approx_{\mathsf{shd},Stb}$" are insensitive to whether or not a trace is a divergence trace. In particular, $\searrow\circ\xrightarrow{\tau}\circ \approx_{\mathsf{shd},Stb} \searrow\circlearrowleft\tau$. As a consequence, "$\approx_{\mathsf{shd}}$" is incomparable to "$\approx_{Tr,Sf}$", "$\approx_{Tr,Div}$", and "$\approx_{\mathsf{CFFD}}$".

Figure 5 demonstrates that "$\approx_{\mathsf{shd}}$" does not preserve the infinite traces.

It is known that "$\approx_{\mathsf{shd}}$" is the weakest congruence with respect to "$\|$", "ϕ", and "$\backslash$" (and ";" and "Φ") that preserves the property "always in all futures, there

Fig. 5 Two "$\approx_{\mathsf{shd}}$"-equivalent LTSs that have different infinite traces.

is a future where eventually the visible action a is executed". Similarly, "$\approx_{\mathsf{shd},Stb}$" is the weakest congruence, when "+" is added to the list of LTS operators.

Because the renaming operator used in [16] is the functional one, we have to prove here that "Φ" respects "$\approx_{\mathsf{shd}}$".

Lemma 3 *The equivalence "$\approx_{\mathsf{shd}}$" is a congruence with respect to "Φ".*

Proof For any state s, let $Tr(s) = \{\sigma \mid s =\sigma\Rightarrow\}$. For any set V of sequences of visible actions, let $\Phi(V)$ denote the set of the renamings of the elements of V by Φ. If $V \neq \emptyset$ then $\Phi(V) \neq \emptyset$. We use subscript $_\Phi$ to distinguish entities of $L\Phi$ from entities of L. The states of $L\Phi$ and L are the same. Assume that $L \approx_{\mathsf{shd}} L'$.

By the assumption, L and L' have the same alphabet Σ. It implies that $L\Phi$ and $L'\Phi$ have the same alphabet.

Assume that $(\sigma_\Phi, K_\Phi) \in Tf(L\Phi)$. By the definition of Tf, there is s such that $\hat{s} =\sigma_\Phi\Rightarrow s$ in $L\Phi$ and $Tr_\Phi(s) \cap K_\Phi = \emptyset$. By the definition of Φ, $Tr_\Phi(s) = \Phi(Tr(s))$, and there is σ such that $\hat{s} =\sigma\Rightarrow s$ in L and $\sigma_\Phi \in \Phi(\{\sigma\})$. Let $K = \{\pi \in \Sigma^* \mid \Phi(\{\pi\}) \cap K_\Phi \neq \emptyset\}$. If $\pi \in K \cap Tr(s)$, then there is $\pi_\Phi \in \Phi(\{\pi\}) \cap K_\Phi \cap Tr_\Phi(s)$, contradicting $Tr_\Phi(s) \cap K_\Phi = \emptyset$. Therefore, $K \cap Tr(s) = \emptyset$ and $(\sigma, K) \in Tf(L)$.

By the definition of "$\approx_{\mathsf{shd}}$", there are ρ and δ such that $\rho = \varepsilon$ or $\rho\delta \in K$, and $(\sigma\rho, K') \in Tf(L')$ where $K' = \{\pi \mid \rho\pi \in K\}$. So L' has a state s' such that $\hat{s}' =\sigma\rho\Rightarrow s'$ in L' and $Tr'(s') \cap K' = \emptyset$.

If $\rho \neq \varepsilon$, then $\rho\delta \in K$. By the construction of K, there are $\rho_\Phi \in \Phi(\{\rho\})$ and $\delta_\Phi \in \Phi(\{\delta\})$ such that $\rho_\Phi\delta_\Phi \in K_\Phi$. If $\rho = \varepsilon$, then let $\rho_\Phi = \varepsilon$. Also then $\rho_\Phi \in \Phi(\{\rho\})$.

Let $K'_\Phi = \{\pi_\Phi \mid \rho_\Phi\pi_\Phi \in K_\Phi\}$. We have $\hat{s}' =\sigma_\Phi\rho_\Phi\Rightarrow s'$ in $L'\Phi$. If $\pi \in Tr'(s')$, then $\pi \notin K'$, yielding $\rho\pi \notin K$ and $\Phi(\{\rho\pi\}) \cap K_\Phi = \emptyset$. In particular, for every $\pi_\Phi \in \Phi(\{\pi\})$ we have $\rho_\Phi\pi_\Phi \notin K_\Phi$ and $\pi_\Phi \notin K'_\Phi$. This implies $Tr'_\Phi(s') \cap K'_\Phi = \emptyset$ and $(\sigma_\Phi\rho_\Phi, K'_\Phi) \in Tf(L'\Phi)$.

This completes the proof of the case $(\sigma_\Phi, K_\Phi) \in Tf(L\Phi)$. The opposite case is symmetric. □

We will show in Section 7 that "$\approx_{\mathsf{shd}}$" is a congruence with respect to "Θ", and "$\approx_{\mathsf{shd},Stb}$" is a congruence with respect to "▷" and "Θ".

6 Constructibility from a Set of LTS Operators up to an Equivalence

In this section we define the central concept studied in this publication, and illustrate it with many examples.

Definition 8 Let "$\approx$" be an equivalence on LTSs, Ω be a set of LTS operators, and $op(L_1, \ldots, L_n)$ be an n-ary LTS operator. We say that op is $\approx$*-constructible* from Ω if and only if there is an LTS expression $expr(L_1, \ldots, L_n)$, composed of only L_1, ..., L_n and elements of Ω, such that for all L_1, ..., L_n, we have $op(L_1, \ldots, L_n) \approx expr(L_1, \ldots, L_n)$.

Example 1 Let A be an alphabet. The operator $L_1 \parallel_A L_2$ is defined as the reachable part of the LTS $(S, \Sigma, \Delta, \hat{s})$, where $S = S_1 \times S_2$, $\Sigma = \Sigma_1 \cup \Sigma_2$, $\hat{s} = (\hat{s}_1, \hat{s}_2)$, and $((s_1, s_2), a, (s_1', s_2')) \in \Delta$ if and only if

- $(s_1, a, s_1') \in \Delta_1$, $s_2' = s_2 \in S_2$, and $a \notin A$,
- $(s_2, a, s_2') \in \Delta_2$, $s_1' = s_1 \in S_1$, and $a \notin A$, or
- $(s_1, a, s_1') \in \Delta_1$, $(s_2, a, s_2') \in \Delta_2$, and $a \in A$.

That is, A determines the actions that L_1 and L_2 must take simultaneously. If $(s_1, a, s_1') \in \Delta_1$, then $a = \tau$ or $a \in \Sigma_1$. If, furthermore, $a \notin \Sigma_1 \cap \Sigma_2$, then $a \notin \Sigma_2$. The same holds with 1 and 2 swapped. A comparison to the definition of "$\parallel$" reveals that $L_1 \parallel_{\Sigma_1 \cap \Sigma_2} L_2$ is the same LTS as $L_1 \parallel L_2$. Therefore, $L_1 \parallel L_2 \equiv L_1 \parallel_{\Sigma_1 \cap \Sigma_2} L_2$. So "$\parallel$" is $\equiv$-constructible from {"$\parallel_{\Sigma_1 \cap \Sigma_2}$"}. □

For simplicity, we often drop the set parentheses and the action and set parameters when saying that an LTS operator is $\approx$-constructible.

Example 2 Let A be an alphabet. The operator $L \setminus_{\mathsf{CCS}} A$ is defined as $L \setminus_{\mathsf{CCS}} A = (S, \Sigma', \Delta', \hat{s})$, where $\Sigma' = \Sigma \setminus A$ and $\Delta' = \{(s, a, s') \in \Delta \mid a \notin A\}$. That is, $L \setminus_{\mathsf{CCS}} A$ is otherwise like $L \setminus A$, but it removes the transitions labelled with elements of A instead of converting them to τ-transitions. The operator captures the essence of the *restriction operator* in CCS. Clearly $L \setminus_{\mathsf{CCS}} A \equiv (L \parallel \mathsf{Stop}_A) \setminus A$. So the unary LTS operator "$\setminus_{\mathsf{CCS}} A$" is $\equiv$-constructible from {"$\parallel$","$\setminus A$","Stop_A"}. In the simpler but less precise language, "$\setminus_{\mathsf{CCS}}$" is $\equiv$-constructible from "$\parallel$", "$\setminus$", and "Stop". □

The following instances of ϕ will prove useful. The first one can be used to convert the alphabets of two or more LTSs so that the resulting alphabets are disjoint. The latter two can be used to convert the alphabets back to the original. The definitions of similar functions with three, four, and more indices are obvious.

- $\lceil L \rceil^{[i]} = \phi(L)$, where $\phi(a) = a^{[i]}$ for every $a \in \Sigma$,
- $\lfloor L \rfloor_{[i]} = \phi(L)$, where $\phi(a^{[i]}) = a$ for every $a \in \Sigma$, and
- $\lfloor L \rfloor_{[i,j]} = \phi(L)$, where $\phi(a^{[i]}) = \phi(a^{[j]}) = a$ for every $a \in \Sigma$.

Example 3 We have $L_1 \parallel_A L_2 \equiv \lfloor \phi_1(L_1) \parallel \phi_2(L_2) \rfloor_{[0,1,2]} \parallel \mathsf{Stop}_{A'}$, where

- $\phi_1(a) = a^{[0]}$ if $a \in A$, and $\phi_1(a) = a^{[1]}$ if $a \in \Sigma_1 \setminus A$,
- $\phi_2(a) = a^{[0]}$ if $a \in A$, and $\phi_2(a) = a^{[2]}$ if $a \in \Sigma_2 \setminus A$, and
- $A' = ((\Sigma_1 \setminus \Sigma_2) \cup (\Sigma_2 \setminus \Sigma_1)) \cap A$.

In this construction, each visible action a of L_i (where $i \in \{1, 2\}$) is converted to either $a^{[0]}$ or to $a^{[i]}$, depending on whether it should synchronize with the opposite LTS L_{3-i}. The operator $\lfloor \rfloor_{[0,1,2]}$ converts the action back to the original. The use of the indices ensures that the visible actions that are not in A are not accidentally in the alphabets of the other components of the parallel composition. If $a \in \Sigma_1 \cap \Sigma_2 \cap A$, then its converted version $a^{[0]}$ is in the alphabets of $\phi_1(L_1)$ and $\phi_2(L_2)$ but not of $\mathsf{Stop}_{A'}$, so it is executed precisely when both L_1 and L_2 execute it. If $a \in \Sigma_i \cap A$ but not in Σ_{3-i}, then $\lfloor \phi_1(L_1) \parallel \phi_2(L_2) \rfloor_{[0,1,2]}$ can execute it such that only L_i moves. However, $\mathsf{Stop}_{A'}$ blocks it. If $a \in \Sigma_i \setminus A$, then $\mathsf{Stop}_{A'}$ does not block it.

So "$\parallel_A$" is $\equiv$-constructible from "$\parallel$", "ϕ", and "Stop". The "$\setminus$" in the formulae are not LTS hiding operators but symbols used for specifying the functions that yield the LTS operators ϕ_1, ϕ_2, and $\mathsf{Stop}_{A'}$ from Σ_1 and Σ_2. □

Fig. 6 The LTSs in Example 4.

It is immediate from the definition that if op is $\approx_1$-constructible from Ω and $L \approx_1 L'$ implies $L \approx_2 L'$ for all LTSs L and L', then op is $\approx_2$-constructible from Ω. As a consequence, $\equiv$-constructibility is a very strong notion of constructibility. It implies all other instances of constructibility discussed in this publication.

On the other hand, $\approx_{Tr}$-constructibility is rather weak. It is implied by most (but not all) instances of constructibility discussed in this publication. The purpose of the next few examples is to present LTS operators that are not even $\approx_{Tr}$-constructible from commonly used LTS operators, progressing from unnatural towards more natural operators.

Example 4 Let L be an LTS. When $s_1 \in S$ and $s_2 \in S$, let $s_1 \sim s_2$ if and only if $(S, \Sigma, \Delta, s_1) \equiv (S, \Sigma, \Delta, s_2)$. The *bisimilarity equivalence class* of $s \in S$ is $[[s]] = \{s' \mid s \sim s'\}$. The *bisimilarity-minimized* version of L is the reachable part of $(S', \Sigma, \Delta', \hat{s}')$, where $S' = \{[[s]] \mid s \in S\}$, $\hat{s}' = [[\hat{s}]]$, and Δ' consists of precisely the triples determined by the rule that if $(s, a, s') \in \Delta$, then $([[s]], a, [[s']]) \in \Delta'$.

Let $op(L)$ first compute the bisimilarity-minimized version of L and then reverse the transitions of the result. Clearly op is alphabet-consistent and respects "$\equiv$". Consider the LTSs in Figure 6. Clearly $op(E_1) \equiv E_1$, but $op(E_2) \equiv \mathsf{Stop}_{\{a\}} \not\approx_{Tr} E_2 \approx_{Tr} E_1 \approx_{Tr} op(E_1)$. We conclude that op does not respect "$\approx_{Tr}$". On the other hand, all LTS operators listed in Definition 4 and thus all expressions $expr(L)$ built from them do respect "$\approx_{Tr}$", yielding $expr(E_1) \approx_{Tr} expr(E_2)$. So $op(E_1) \not\approx_{Tr} expr(E_1)$ or $op(E_2) \not\approx_{Tr} expr(E_2)$, and op is not $\approx_{Tr}$-constructible from the operators listed in Definition 4. □

An LTS operator that does respect "$\approx_{Tr}$" but is not $\approx_{Tr}$-constructible from commonly used LTS operators can be easily constructed by exploiting the fact that the latter are *monotonic* with respect to the traces. That is, if $\Sigma_i = \Sigma'_i$ and $Tr(L_i) \subseteq Tr(L'_i)$ for every L_i and L'_i where $1 \le i \le n$, then $Tr(op(L_1, \ldots, L_n)) \subseteq Tr(op(L'_1, \ldots, L'_n))$. (Non-monotonic operators are sometimes used to model priority, that is, a lower-priority transition can be made only if no higher-priority transition can be made.)

Example 5 Let $op(L) = \xrightarrow{a}\circ$ if $Tr(L) = \{\varepsilon\}$ and otherwise $op(L) = \mathsf{Stop}_{\{a\}}$. We have $Tr(\mathsf{Stop}_{\{a\}}) = \{\varepsilon\} \subseteq \{\varepsilon, a\} = Tr(\xrightarrow{a}\circ)$. Therefore, $Tr(expr(\mathsf{Stop}_{\{a\}})) \subseteq Tr(expr(\xrightarrow{a}\circ))$ for any $expr$ built only using the commonly used LTS operators. On the other hand, $Tr(op(\mathsf{Stop}_{\{a\}})) = Tr(\xrightarrow{a}\circ) = \{\varepsilon, a\} \not\subseteq \{\varepsilon\} = Tr(\mathsf{Stop}_{\{a\}}) = Tr(op(\xrightarrow{a}\circ))$. So $expr(L) \not\approx_{Tr} op(L)$ for $L = \mathsf{Stop}_{\{a\}}$ or for $L = \xrightarrow{a}\circ$. □

The next example presents an LTS operator that does respect "$\approx_{Tr}$", is monotonic, but is not $\approx_{Tr}$-constructible from commonly used LTS operators.

Example 6 If $a^n \in Tr(L)$ for every $n \in \mathbb{N}$, then let $op(L) = \xrightarrow{a}\circ$, and otherwise let $op(L) = \mathsf{Stop}_{\{a\}}$.

Let $E_\omega = \xrightarrow{a}\circ\xrightarrow{a}\circ\xrightarrow{a}\ldots$, and let E_n be similar but end after n a-transitions. Assume that $expr$ is constructed only from commonly used LTS operators. Each

transition of the result of a commonly used LTS operator arises from zero or one transition of each of its arguments. If $expr(L) \approx_{Tr} op(L)$, then $expr(E_\omega)$ may eventually execute a. At that point of time it has only made a finite number of transitions. Let that number be n. Within $expr(E_\omega)$, each instance of E_ω has made at most n transitions. The past behaviour of $expr(L)$ is insensitive to the part of L that has not been executed by any instance of L. Therefore, also $expr(E_n)$ may eventually execute a. However, $op(E_n)$ may not. So $expr(E_n) \not\approx_{Tr} op(E_n)$, and we conclude that *expr* does not exist. □

The previous example was based on using an unbounded number of traces of L to decide whether $op(L)$ has a certain trace. In contrast, each operator in Definition 4 uses at most one trace of each of its argument LTSs for producing a trace of an application of the operator. Intuitively, that is a reasonable property of LTS operators. We will study the consequences of this idea in Section 8.

The next three examples present a case that is strictly between "$\equiv$" and "$\approx_{Tr}$".

Example 7 Let $\Phi = \{(1,a) \mid a \in \Sigma_1^{[1]}\} \cup \{(2,a) \mid a \in \Sigma_2^{[2]}\}$. Let $C = 1 \circlearrowright \xrightarrow{2} \circlearrowleft 2$, except that if $\Sigma_1 = \emptyset$ then drop the 1-transition from C and 1 from its alphabet, and similarly with 2. Let $expr(L_1, L_2) = \lfloor C\Phi \parallel \lceil L_1 \rceil^{[1]} \parallel \lceil L_2 \rceil^{[2]} \rfloor_{[1,2]}$. The alphabet of $C\Phi$ is $\Sigma_1^{[1]} \cup \Sigma_2^{[2]}$ and the alphabet of $expr(L_1, L_2)$ is $\Sigma_1 \cup \Sigma_2$. We call C the "control LTS", because its state determines whether L_1, L_2, or both can make visible transitions in $expr(L_1, L_2)$. As long as C is in its initial state, both L_1 and L_2 can. When L_2 makes a visible transition, C moves to its rightmost state. From then on, only L_2 can make visible transitions.

The behaviour of $expr(L_1, L_2)$ is similar to the behaviour of $L_1 \rhd L_2$, with two differences. First, the change from the behaviour of L_1 to the behaviour of L_2 takes place when executing a visible transition, while in $L_1 \rhd L_2$ it can also take place during a τ-transition. Second, L_2 may be capable of making invisible transitions also before and L_1 also after the change. So we have $Tr(expr(L_1, L_2)) = Tr(L_1 \rhd L_2) = \{\sigma\rho \mid \sigma \in Tr(L_1) \wedge \rho \in Tr(L_2)\}$ and $Inf(expr(L_1, L_2)) = Inf(L_1 \rhd L_2) = Inf(L_1) \cup \{\sigma\xi \mid \sigma \in Tr(L_1) \wedge \xi \in Inf(L_2)\}$, but not necessarily $Sf(expr(L_1, L_2)) = Sf(L_1 \rhd L_2)$ or $Div(expr(L_1, L_2)) = Div(L_1 \rhd L_2)$. For instance, if $L_1 = \circlearrowleft \tau$ and $L_2 = \xrightarrow{a}$, then $Div(expr(L_1, L_2)) = \{\varepsilon, a\}$ and $Sf(expr(L_1, L_2)) = \emptyset$, but $Div(L_1 \rhd L_2) = \{\varepsilon\}$ and $Sf(L_1 \rhd L_2) = \{(a, \emptyset), (a, \{a\})\}$.

So $expr(L_1, L_2) \approx_{Tr,Inf} L_1 \rhd L_2$, and we conclude that $L_1 \rhd L_2$ is $\approx_{Tr,Inf}$-constructible from "$\parallel$", "Φ", $1 \circlearrowright \xrightarrow{2} \circlearrowleft 2$, $\xrightarrow{2} \circlearrowleft 2$, $1 \circlearrowright$, and $\circlearrowright$. Theorem 4 will say that it is $\approx_{Tr,Inf,minD,Stb}$-constructible. □

The failure of the above construction to yield $Sf(expr(L_1, L_2)) = Sf(L_1 \rhd L_2)$ does not prove that $L_1 \rhd L_2$ is not $\approx_{Sf}$-constructible from "$\parallel$", "Φ", and LTS constant families. However, the next simple lemma facilitates proving it.

Lemma 4 *Assume that "$\approx'$" preserves the alphabet and is a congruence with respect to all LTS operators in Ω, but not with respect to $op(L_1, \ldots, L_n)$. Assume that for every L and L', $L \approx L'$ implies $L \approx' L'$. Then $op(L_1, \ldots, L_n)$ is not $\approx$-constructible from Ω.*

Proof To derive a contradiction, assume that $op(L_1, \ldots, L_n)$ is $\approx$-constructible from Ω. By Definition 8 and Lemma 2 there is an LTS expression *expr* such that if $L_1 \approx' L_1'$, …, $L_n \approx' L_n'$, then $op(L_1, \ldots, L_n) \approx expr(L_1, \ldots, L_n) \approx' expr(L_1', \ldots, L_n') \approx op(L_1', \ldots, L_n')$. So *op* respects "$\approx'$". □

Let us present three examples of the use of Lemma 4. The first just recalls the well-known fact that to obtain a congruence with respect to "+" and "$\rhd$", the initial stability *Stb* is often needed. The second shows that "$\rhd$" does but "+" does not have more to it, in the sense that prefixing the arguments makes the latter but not the former $\equiv$-constructible. The third illustrates that unconstructibility can sometimes be proven by inventing an equivalence that has suitable congruence properties.

Example 8 It is known that "$\approx_{Sf}$" is a congruence with respect to "$\|$", "Φ", "$\backslash$", and ";", but not with respect to "+", "$\rhd$", and "Θ". (For instance, $\xrightarrow{a} + 0$ and $\xrightarrow{a} \rhd 0$ do not but $\xrightarrow{a} + \xrightarrow{\tau}$ and $\xrightarrow{a} \rhd \xrightarrow{\tau}$ do have the stable failure $(\varepsilon, \{a\})$. Furthermore, $\tau\text{-}\mathsf{loop}_{\{a\}}\,\Theta_{\{a\}}\,0$ does but $\tau\text{-}\mathsf{loop}_{\{a\}}\,\Theta_{\{a\}}\,0$ does not have the stable failure $(a, \emptyset)$.) By Lemma 4, for any "$\approx$" that preserves at least the alphabet and stable failures, "+", "$\rhd$", and "Θ" are not $\approx$-constructible from "$\|$", "Φ", "$\backslash$", ";", and LTS constant families. □

Example 9 If $\tau \neq a_1 \neq \varepsilon$ and $\tau \neq a_2 \neq \varepsilon$, then

$$a_1; L_1 + a_2; L_2 \;\equiv\; \lfloor \xleftarrow{a_1^{[1]}} \xrightarrow{a_2^{[2]}} \| \; a_1^{[1]}; \lceil L_1 \rceil^{[3]} \;\|\; a_2^{[2]}; \lceil L_2 \rceil^{[4]} \rfloor_{[1,2,3,4]} \; .$$

If a_1, a_2, or both are τ, the construction is changed by hiding the resulting $\tau^{[1]}$, $\tau^{[2]}$, or both after the parallel composition.

As a consequence, the *prefixed choice* $op(L_1, L_2) = a_1; L_1 + a_2; L_2$ is $\equiv$-constructible from "$\|$", "ϕ", "$\backslash$", ";", and constant LTSs.

However, the same does not hold for "$\rhd$". Let $L_1 \overline{\rhd} L_2$ be defined as $\tau; L_1 \rhd \tau; L_2$. Although $\tau \xrightarrow{a} \tau$ and $\tau\text{-}\mathsf{loop}_{\{a\}}$ have the same alphabet $\{a\}$, the same set of stable failures $\emptyset$, and the same initial stability false, $\tau \xrightarrow{a} \tau \,\overline{\rhd}\, 0$ has but $\tau\text{-}\mathsf{loop}_{\{a\}} \,\overline{\rhd}\, 0$ does not have the stable failure $(a, \emptyset)$. So, although "$\approx_{Sf,Stb}$" is a congruence with respect to "$\|$", "Φ", "$\backslash$", ";", "+", and LTS constant families, it is not with respect to "$\overline{\rhd}$". We conclude that "$\overline{\rhd}$" and "$\rhd$" are not $\approx_{Sf,Stb}$-constructible from "$\|$", "Φ", "$\backslash$", ";", "+", and LTS constant families. The latter claim follows from the fact that if "$\rhd$" were constructible, then, by its definition, also "$\overline{\rhd}$" would be. □

Example 10 Let $L \approx' L'$ if and only if

- $\Sigma = \Sigma'$, and
- $Sf(L) = Sf(L')$ or $\varepsilon \in Div(L) \cap Div(L')$.

Let $\Sigma = \Sigma'$. Assume first that $Sf(L) = Sf(L')$. Then $L \approx_{Sf} L'$. We have $L \setminus A \approx_{Sf} L' \setminus A$, yielding $L \setminus A \approx' L' \setminus A$. For a similar reason, $L\Phi \approx' L'\Phi$ and $L \| M \approx' L' \| M$. Assume then that $\varepsilon \in Div(L) \cap Div(L')$. Then clearly $\varepsilon \in Div(L \setminus A) \cap Div(L' \setminus A)$, $\varepsilon \in Div(L\Phi) \cap Div(L'\Phi)$, and $\varepsilon \in Div(L \parallel M) \cap Div(L' \parallel M)$. So "$\|$", "$\Phi$", "$\backslash$", and LTS constant families respect "$\approx'$".

Obviously $\tau \xrightarrow{\tau} \approx' \tau$. However, $a; \tau \xrightarrow{\tau} \not\approx' a; \tau$, because neither one has ε as a divergence trace, and $(a, \emptyset)$ is a stable failure of the former but not the latter.

We conclude that if "$\approx$" preserves the alphabet, stable failures and minimal divergence traces, then ";" is not $\approx$-constructible from "$\|$", "Φ", "$\backslash$", and LTS constant families. In particular, ";" is not $\approx_{\mathsf{CFFD}}$-constructible from "$\|$", "Φ", "$\backslash$", and LTS constant families. □

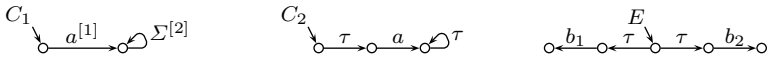

Fig. 7 LTS constant families used in Theorem 2. C_1 has the loop transition for each element of $\Sigma^{[2]}$.

7 Constructibility of Action Prefix, Choice, Interrupt, and Throw

In this section we present, in a summarized form, many results that concern the extent to which ";", "+", "$\rhd$", and "Θ" are constructible.

In Example 8 we recalled that mimicking the initial stable failures of $L_1 + L_2$ and $L_1 \rhd L_2$ with other commonly used LTS operators is impossible. We now introduce an LTS operator that largely solves this problem and thus, in some sense, captures the essence of the initial stability issue.

Definition 9 Let $\tau \neq a \neq \varepsilon$. The *stabilizing prefix* is defined as

$$a\breve{;}\, L \;=\; \begin{cases} (S, \Sigma \cup \{a\}, \Delta, \hat{s}) & \text{, if } Stb(L) = \mathsf{true} \\ a; L & \text{, otherwise.} \end{cases}$$

That is, if L is initially stable, then $a\breve{;}\, L$ adds a to its alphabet (if it is not already there) but causes no other changes. Otherwise $a\breve{;}\, L$ adds an a-transition to the front of L. In both cases, $a\breve{;}\, L$ is initially stable.

Both "$\approx_{Stb}$", "$\approx_{Tr,Stb}$", "$\approx_{Inf,Stb}$", "$\approx_{Sf,Stb}$", "$\approx_{Div,Stb}$", and "$\approx_{\mathsf{shd},Stb}$" are congruences with respect to "$\breve{;}$".

It is the time to present our results on the constructibility of ";". In Example 7, four different LTSs were used, depending on whether $\Sigma_1 = \emptyset$ and whether $\Sigma_2 = \emptyset$. Then "Φ" was used to adapt one of the constant LTSs to the alphabets in question. To avoid this complexity, we introduce a simple drawing convention that allows drawing some LTS constant families. If the label of a transition in a drawing is a set of actions like in Figure 7 left, then the LTS has a transition for each member of the set. In particular, if the label is the empty set, then there is no transition.

Theorem 2 *Let $\tau \neq a \neq \varepsilon$, and let C_1 and C_2 be like in Figure 7. The LTS operator $a; L$*

- *is $\approx_{Tr,Inf}$- and $\approx_{\mathsf{shd}}$-constructible from "$\|$", "ϕ", and C_1,*
- *is $\approx_{Tr,Inf,Sf}$-constructible from "$\|$", "ϕ", "+", C_1, and C_2,*
- *is $\approx_{\mathsf{dbr}}$-constructible from "$\|$", "$\breve{;}$", and $\overset{\tau}{\rightarrow}\circ$,*
- *is $\equiv$-constructible from "Θ", and*
- *is neither $\approx_{minD}$-, $\approx_{Tr,Stb}$-, $\approx_{\mathsf{shd},Stb}$-, nor $\approx_{\mathsf{wb}}$-constructible from "$\|$", "Φ", "$\setminus$", "+", "$\rhd$", and LTS constant families.*

Proof Consider $C_1 \,\|\, \lceil L \rceil^{[2]}$. Its alphabet is $\{a^{[1]}\} \cup \Sigma^{[2]}$. In it, C_1 may execute $a^{[1]}$ once at any instant of time. The visible transitions of $\lceil L \rceil^{[2]}$ are blocked until then, but not blocked afterwards. So the behaviour of $C_1 \,\|\, \lceil L \rceil^{[2]}$ is otherwise the same as the behaviour of $a^{[1]}; \lceil L \rceil^{[2]}$, except that in the former, $\lceil L \rceil^{[2]}$ may do any of its initial sequences of τ-transitions "prematurely", that is, before $a^{[1]}$ is executed. This does not affect the traces and infinite traces. They are $\{\varepsilon\} \cup \{a^{[1]}\sigma^{[2]} \mid \sigma \in Tr(L)\}$ and $\{a^{[1]}\xi^{[2]} \mid \xi \in Inf(L)\}$ for both. Let $expr(L) = \lfloor C_1 \,\|\, \lceil L \rceil^{[2]} \rfloor_{[1,2]}$. We conclude that $a; L \approx_{Tr,Inf} expr(L)$.

Regarding "$\approx_{\mathsf{shd}}$", for each nonempty trace σ, $expr(L)$ can precisely simulate the execution of σ in $a;L$, and $a;L$ can simulate the execution of σ in $expr(L)$ such that the places of some τ-transitions may have changed. However, both executions lead to the same state of L, from which on both expressions continue like L. Therefore, when $\sigma \neq \varepsilon$, $(\sigma, K) \in Tf(a;L)$ if and only if $(\sigma, K) \in Tf(expr(L))$.

The case $\sigma = \varepsilon$ remains. Let $(\varepsilon, K) \in Tf(a;L)$. When executing ε, $a;L$ does not move. Because $Tr(a;L) = Tr(expr(L))$, $(\varepsilon, K) \in Tf(expr(L))$ is seen true by letting also $expr(L)$ stay in its initial state. The only non-trivial case is $(\varepsilon, K) \in Tf(expr(L)) \setminus Tf(a;L)$. If it arises, then $expr(L)$ has made at least one τ-transition, taking it to some state s. Then there is some $a\delta \in K$ such that $a\delta \in Tr(a;L)$ but not $s =a\delta\Rightarrow$ in $expr(L)$. We may choose $\rho = a$ in Definition 7, yielding $K' = \{\pi \mid a\pi \in K\}$. By letting $a;L$ make those τ-transitions of L after a that $expr(L)$ made without executing a, we see that $(a, K') \in Tf(a;L)$.

To prove the second claim, let $expr(L) = \lfloor C_1 \parallel \lceil L \rceil^{[2]} \rfloor_{[1,2]} + C_2$. The part "$+\ C_2$" introduces no infinite traces. It introduces the traces ε and a which also are traces of $\lfloor C_1 \parallel \lceil L \rceil^{[2]} \rfloor_{[1,2]}$. By the previous result, $a;L \approx_{Tr,Inf} expr(L)$. Both $a;L$ and $\lfloor C_1 \parallel \lceil L \rceil^{[2]} \rfloor_{[1,2]}$ have precisely the following stable failures whose trace is not ε:

- $\{(a\sigma, A) \mid (\sigma, A) \in Sf(L)\}$, if $a \in \Sigma$, and
- $\{(a\sigma, A) \mid (\sigma, A \setminus \{a\}) \in Sf(L)\}$, if $a \notin \Sigma$.

Regarding stable failures of the form (ε, A), in the case of $a;L$ we have $A \subseteq \Sigma\setminus\{a\}$. The same holds for $\lfloor C_1 \parallel \lceil L \rceil^{[2]} \rfloor_{[1,2]}$ if $(\varepsilon, \emptyset) \in Sf(L)$, otherwise $\lfloor C_1 \parallel \lceil L \rceil^{[2]} \rfloor_{[1,2]}$ lacks such stable failures. However, the part "$+\ C_2$" introduces them but no other stable failures. Therefore, $a;L \approx_{Sf} expr(L)$.

To prove the third claim, consider $expr(L) = a;(L \parallel \xrightarrow{\tau})$. It first makes an a-transition. Then it has two copies of L and a τ-transition from each state of the first copy to the corresponding state in the second copy. Let these states be called $(s,1)$ and $(s,2)$, where s is a state of L. Let "$\sim$" relate the initial state of $a;L$ to the initial state of $expr(L)$, and let it relate each $s \in S$ to both $(s,1)$ and $(s,2)$. It is easy to check that this relation is a divergence-preserving branching bisimulation.

The fourth claim was proven already when introducing "Θ": $a;L \equiv \xrightarrow{a}\Theta_{\{a\}}L$.

From now on, let $expr(L)$ be an expression built only from the operators listed in the last claim. Consider first "$\approx_{minD}$". If $expr(L)$ does not refer to L, then $expr(\xrightarrow{}) = expr(\xrightarrow{}\tau)$. However, $a;\xrightarrow{} \not\approx_{minD} a;\xrightarrow{}\tau$, so either $expr(\xrightarrow{}) \not\approx_{minD} a;\xrightarrow{}$ or $expr(\xrightarrow{}\tau) \not\approx_{minD} a;\xrightarrow{}\tau$. So $expr(L)$ fails to mimick $a;L$. The case remains where $expr(L)$ uses L. Excluding LTS constants, the operators listed in the claim have the property that if any of their arguments has ε as a divergence trace, then the result of the application of the operator also has ε as a divergence trace. So $\varepsilon \in minD(expr(\xrightarrow{}\tau))$. However, $\varepsilon \notin minD(a;\xrightarrow{}\tau)$, so $expr$ fails again.

Every operator listed in the last claim has the property that if any of its arguments is initially unstable, then also the result of the application of the operator is initially unstable. On the other hand, $a;L$ is initially stable. Therefore, if $expr(L)$ uses L, then $Stb(expr(\xrightarrow{}\tau)) \neq Stb(a;\xrightarrow{}\tau)$. If $expr(L)$ does not use L, then obviously $Tr(expr(L)) = Tr(a;L)$ cannot hold for every L. So $a;L$ is $\approx_{Tr,Stb}$-unconstructible. Because "$\approx_{\mathsf{shd},Stb}$" implies "$\approx_{Tr,Stb}$", $a;L$ is $\approx_{\mathsf{shd},Stb}$-unconstructible.

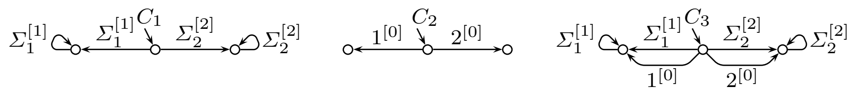

Fig. 8 LTS constant families used in Theorem 3.

To derive a contradiction with the $\approx_{\mathsf{wb}}$-claim, assume that $expr(L) \approx_{\mathsf{wb}} a; L$, and let E be like in Figure 7. The LTS $a; E$ has the trace ab_1. Also $expr(E)$ must have it. It must arise from the execution of the b_1-branch of at least one instance of E, because otherwise also $expr(\tau; \mathsf{Stop}_{\{b_1\}} + \tau; b_2; \searrow\!\!\circ)$ would have it, which would be incorrect. However, the listed operators cannot prevent all instances of E from executing their τ-transitions of the b_2-branch before anything else has happened. So, before yielding a, $expr(E)$ may commit to not yield b_1. As a consequence, $expr(E) \not\approx_{\mathsf{wb}} a; E$. ☐

The next theorem discusses the constructibility of the choice operator.

Theorem 3 *Let C_1, C_2, and C_3 be like in Figure 8. The LTS operator $L_1 + L_2$*

- *is $\approx_{Tr,Inf,minD,Stb}$-constructible from "$\|$", "$\phi$", and C_1,*
- *is $\approx_{Tr,Inf,Div}$-constructible from "$\|$", "ϕ", "$\backslash$", "$;$", and C_2,*
- *is $\approx_{\mathsf{CFFD},Stb}$- and $\approx_{\mathsf{shd},Stb}$-constructible from "$\|$", "$\phi$", "$\backslash$", "$\overset{\curlyvee}{;}$", and C_3,*
- *is not $\approx_{Div}$-constructible from "$\|$", "Φ", "$\backslash$", and LTS constant families,*
- *is not $\approx_{Sf}$-constructible from "$\|$", "Φ", "$\backslash$", "$;$", and LTS constant families, and*
- *is neither $\approx_{Tr,Sf}$- nor $\approx_{\mathsf{shd}}$-constructible from "$\|$", "Φ", "$\backslash$", "$;$", "Θ", and LTS constant families.*

Proof To prove the first claim, let $expr(L_1, L_2) = \lfloor C_1 \parallel \lceil L_1 \rceil^{[1]} \parallel \lceil L_2 \rceil^{[2]} \rfloor_{[1,2]}$. The alphabet of $expr(L_1, L_2)$ is $\Sigma_1 \cup \Sigma_2$. The first visible action of $expr(L_1, L_2)$ either arises from L_1 and blocks the visible actions of L_2, or the other way round. So $expr(L_1, L_2) \approx_{Tr,Inf} L_1 + L_2$. Because C_1 does not introduce τ-transitions to $expr(L_1, L_2)$, we have $Stb(expr(L_1, L_2))$ if and only if $Stb(L_1)$ and $Stb(L_2)$. If $\varepsilon \in Div(L_1)$ or $\varepsilon \in Div(L_2)$, then $minD(expr(L_1, L_2)) = minD(L_1 + L_2) = \{\varepsilon\}$. Otherwise $Div(expr(L_1, L_2)) = Div(L_1) \cup Div(L_2) = Div(L_1 + L_2)$, implying $minD(expr(L_1, L_2)) = minD(L_1 + L_2)$. So $expr(L_1, L_2) \approx_{minD} L_1 + L_2$.

The second claim follows from the expression

$$expr(L_1, L_2) \;=\; \lfloor (C_2 \parallel 1^{[0]}; \lceil L_1 \rceil^{[1]} \parallel 2^{[0]}; \lceil L_2 \rceil^{[2]}) \setminus \{1^{[0]}, 2^{[0]}\} \rfloor_{[1,2]} \;.$$

The proof of the third claim uses

$$expr(L_1, L_2) \;=\; \lfloor (C_3 \parallel 1^{[0]} \overset{\curlyvee}{;} \lceil L_1 \rceil^{[1]} \parallel 2^{[0]} \overset{\curlyvee}{;} \lceil L_2 \rceil^{[2]}) \setminus \{1^{[0]}, 2^{[0]}\} \rfloor_{[1,2]} \;.$$

For instance, if L_1 is initially stable but L_2 is not, then $expr(L_1, L_2)$ has two possibilities. It may execute a visible transition of L_1, causing C_3 to move to its leftmost state. From then on its behaviour is the same as that of L_1 after the same transition. Alternatively, it executes the τ-transition that arises from $2^{[0]}$, causing C_3 to move to its rightmost state. From then on its behaviour is the same as that of L_2 starting at its initial state. In general,

- if $Stb(L_1)$ and $Stb(L_2)$, then $expr(L_1, L_2) \equiv L_1 + L_2$,

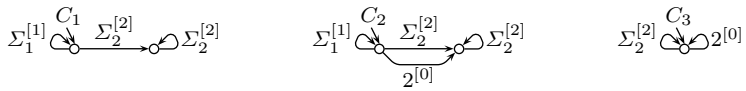

Fig. 9 LTS constant families used in Theorem 4.

– if $Stb(L_1)$ and $\neg Stb(L_2)$, then $expr(L_1, L_2) \equiv L_1 + \tau; L_2$,
– if $\neg Stb(L_1)$ and $Stb(L_2)$, then $expr(L_1, L_2) \equiv \tau; L_1 + L_2$, and
– if $\neg Stb(L_1)$ and $\neg Stb(L_2)$, then $expr(L_1, L_2) \equiv \tau; L_1 + \tau; L_2$.

The additional "τ;"s in the above expressions do not affect the traces, infinite traces, and divergence traces of the result. They do not affect the initial stability and stable failures either, because they only occur when the prefixed L_1 or L_2 is unstable. So in all cases, $expr(L_1, L_2) \approx_{\mathsf{CFFD},Stb} L_1 + L_2$.

The additional "τ;"s do not affect those tree failures whose trace is not ε. Furthermore, they do not affect those tree failures where none of them is executed or where at least one τ-transition of L_1 or L_2 is executed. In the remaining case, a tree failure with the same trace and at least the same refusal set is obtained by letting L_1 or L_2 execute an initial τ-transition. So in all cases, $expr(L_1, L_2) \approx_{\mathsf{shd},Stb} L_1 + L_2$. This construction does not apply to "$\approx_{\mathsf{wb}}$", because for it, the "τ;"s are significant.

The fourth claim follows from the observation that if $L_1 = \tau$, then any $expr(L_1, L_2)$ that uses L_1 and is only built from the listed LTS operators diverges in every state. Therefore, it cannot precisely mimick the divergence traces of $\tau + a\,a\,\tau$. If $expr(\tau, a\,a\,\tau)$ has the divergence trace aa that it should have, then it also has a state where it has executed a but not aa. Because it diverges in every state, a is its divergence trace, which is incorrect.

The last two claims follow from Lemma 4, because the operators listed in them do but "+" does not respect the equivalence. Clearly $\approx \tau$ but $+ a \not\approx \tau + a$ when "$\approx$" is "$\approx_{Sf}$", "$\approx_{Tr,Sf}$", or "$\approx_{\mathsf{shd}}$". □

The next theorem focuses on the interrupt operator.

Theorem 4 *Let C_1, C_2, and C_3 be like in Figure 9. The LTS operator $L_1 \rhd L_2$*

– *is $\approx_{Tr,Inf,minD,Stb}$-constructible from "$\|$", "$\phi$", and C_1,*
– *is $\approx_{\mathsf{shd},Stb}$-constructible from "$\|$", "$\phi$", "$\backslash$", "$\mathring{;}$", and C_2,*
– *is $\approx_{\mathsf{CFFD},Stb}$-constructible from "$\|$", "$\phi$", "$\backslash$", "$\Theta$", "$\mathring{;}$", and C_3,*
– *is not $\approx_{Sf}$-constructible from "$\|$", "Φ", "$\backslash$", "$;$", and LTS constant families,*
– *is neither $\approx_{Tr,Sf}$- nor $\approx_{\mathsf{shd}}$-constructible from "$\|$", "Φ", "$\backslash$", "$;$", "Θ", and LTS constant families, and*
– *is neither $\approx_{Sf,Stb}$- nor $\approx_{Div}$-constructible from "$\|$", "Φ", "$\backslash$", "$;$", "$+$", "$\mathring{;}$", and LTS constant families.*

Proof Consider the expression $expr(L_1, L_2) = \lfloor C_1 \parallel \lceil L_1 \rceil^{[1]} \parallel \lceil L_2 \rceil^{[2]} \rfloor_{[1,2]}$. That $expr(L_1, L_2) \approx_{Tr,Inf} L_1 \rhd L_2$ was proven in Example 7. The $C\Phi$ there is the same LTS as C_1.

If L_1 and L_2 are initially stable, then both $expr(L_1, L_2)$ and $L_1 \rhd L_2$ are initially stable. Otherwise both are initially unstable.

The expression can diverge when $L_1 \rhd L_2$ diverges by mimicking precisely the contributions of L_1 and L_2 to the execution. We still have to prove that

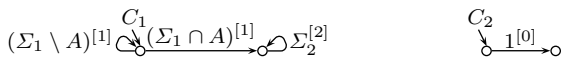

Fig. 10 LTS constant families used in Theorem 5.

$expr(L_1, L_2)$ does not diverge when it should not. The expression diverges precisely when L_1 or L_2 diverges. If L_1 diverges, then either also $L_1 \rhd L_2$ diverges or L_2 has started in $L_1 \rhd L_2$. In the latter case, the trace has the divergence trace of L_1 as a prefix, so minimal divergence traces are not affected. If L_2 diverges, then either also $L_1 \rhd L_2$ diverges or L_2 has not started in $L_1 \rhd L_2$. In the latter case, $\varepsilon \in minD(L_2)$, implying that $minD(expr(L_1, L_2)) = minD(L_1 \rhd L_2) = \{\varepsilon\}$.

The second claim is proven similarly to the $\approx_{\mathsf{shd},Stb}$-constructibility claim in Theorem 3, using the expression

$$expr(L_1, L_2) \;=\; \lfloor (C_2 \parallel \lceil L_1 \rceil^{[1]} \parallel 2^{[0]} \mathbin{\breve{;}} \lceil L_2 \rceil^{[2]}) \setminus \{2^{[0]}\} \rfloor_{[1,2]} \;.$$

This proof does not apply to "$\approx_{\mathsf{CFFD},Stb}$", because if L_2 starts when L_1 has executed a divergence trace, then $L_1 \rhd L_2$ does but $expr(L_1, L_2)$ does not stop L_1 from diverging. However, the following expression works for "$\approx_{\mathsf{CFFD},Stb}$", giving the third claim:

$$expr(L_1, L_2) \;=\; \lfloor (((\lceil L_1 \rceil^{[1]} \parallel C_3) \,\Theta_{\Sigma^{[2]} \cup \{2^{[0]}\}}\, C_3) \parallel 2^{[0]} \mathbin{\breve{;}} \lceil L_2 \rceil^{[2]}) \setminus \{2^{[0]}\} \rfloor_{[1,2]} \;.$$

The fourth claim was proven in Example 8. The fifth claim follows from Lemma 4, because "$\approx_{Tr,Sf}$" and "$\approx_{\mathsf{shd}}$" are congruences with respect to the listed operators but not with respect to "$\rhd$". Excluding "$\breve{;}$", the first part of the last claim was proven in Example 9, but the proof there applies also to "$\breve{;}$".

The remaining claim follows, if we prove that $op(L) = L \rhd$  is not constructible. To derive a contradiction, assume that $expr(L)$ has been built only using the listed operators, and, for every L, $expr(L) \approx_{Div} op(L)$. Let $E =$  We have $abb \in Div(op(E)) = Div(expr(E))$ and $ab \notin Div(op(E))$. When some instance of L has started diverging within $expr(L)$, the listed operators cannot stop the divergence. Therefore, when producing the divergence trace abb, $expr(L)$ must not let any instance of L start before it has produced the trace abb, to prevent $ab \in Div(expr(E))$. So $expr(L)$ first produces abb and then lets at least one instance of L start. This implies that also $expr(\tau\text{-}\mathsf{loop}_{\{a\}})$ yields abb and diverges, although $abb \notin Div(op(\tau\text{-}\mathsf{loop}_{\{a\}}))$. □

Finally, we discuss the throw operator. Because it is a generalization of $a; L$ with $a \neq \tau$, the results and proofs have a lot in common with Theorem 2. However, the possibility that L_1 diverges makes a significant difference.

Theorem 5 *Let A be an alphabet, let C_1 and C_2 be like in Figure 10, and let C_1' be C_1 with also $0^{[0]}$ in its alphabet. The LTS operator $L_1 \,\Theta_A\, L_2$*

- *is $\approx_{Tr,Inf}$- and $\approx_{\mathsf{shd}}$-constructible from "$\parallel$", "ϕ", and C_1,*
- *is $\approx_{\mathsf{shd},Stb}$- and $\approx_{\mathsf{br}}$-constructible from "$\parallel$", "Φ", "$\setminus$", "$;$", and C_1',*
- *is $\approx_{Tr,Inf,Sf,Stb}$-constructible from "$\parallel$", "Φ", "$\setminus$", "$;$", "$\rhd$", C_1', and C_2,*
- *is not $\approx_{Sf}$-constructible from "$\parallel$", "Φ", "$\setminus$", "$;$", "$+$", "$\breve{;}$", and LTS constant families, and*
- *is not $\approx_{minD}$-constructible from "$\parallel$", "Φ", "$\setminus$", "$;$", "$+$", "$\rhd$", "$\breve{;}$", and LTS constant families.*

Proof The proof of the first claim uses $expr(L_1, L_2) = \lfloor C_1 \parallel \lceil L_1 \rceil^{[1]} \parallel \lceil L_2 \rceil^{[2]} \rfloor_{[1,2]}$. It simulates $L_1 \mathbin{\Theta_A} L_2$ otherwise faithfully, except that L_1 may make invisible transitions also after and L_2 before C_1 moves from left to right. These invisible transitions do not affect "$\approx_{Tr,Inf}$". The remnant invisible transitions of L_1 do not affect the sequences of visible actions, so they do not affect "$\approx_{\mathsf{shd}}$". The premature invisible transitions of L_2 may introduce tree failures (σ, K) that $L_1 \mathbin{\Theta_A} L_2$ lacks. This happens only when $\sigma \in Tr(L_1) \cap (\Sigma_1 \setminus A)^*$ and for each s_1 such that $\hat{s} =\sigma\Rightarrow s_1$ in L_1, K contains some $\rho a \delta$ such that $a \in A$, ρ does not contain elements of A, L_1 can execute ρa from s_1, and L_2 can execute δ from its initial state but not from its current state s_2 in $expr(L_1, L_2)$. However, then $(\sigma\rho a, K') \in Tf(L_1 \mathbin{\Theta_A} L_2)$, where $K' = \{\pi \mid \rho a \pi \in K\}$ and L_2 moves to s_2 after L_1 has executed a in $L_1 \mathbin{\Theta_A} L_2$.

To obtain the second claim, the construction is modified such that L_2 cannot execute τ-transitions prematurely. The expression

$$expr(L_1, L_2) \;=\; \lfloor (C_1' \parallel \lceil L_1 \rceil^{[1]} \parallel (0^{[0]}; \lceil L_2 \rceil^{[2]})\Phi) \setminus \{0^{[0]}\} \rfloor_{[1,2]}$$

suffices, where $\Phi = \{(0^{[0]}, a) \mid a \in (\Sigma_1 \cap A)^{[1]}\}$. If $\Sigma_1 \cap A \neq \emptyset$, then $0^{[0]}$ synchronizes with the first action in A executed by L_1. The $0^{[0]}$ in the alphabet of C_1' is superfluous and hidden. If $\Sigma_1 \cap A = \emptyset$, then $0^{[0]}$ survives Φ but is blocked by C_1'.

The third claim requires that L_1 is switched off when L_2 starts, so that $expr(\xrightarrow{a}\tau, \circ)$ has the stable failure $(a, \emptyset)$ of $\xrightarrow{a}\tau \mathbin{\Theta_{\{a\}}} \circ$. This is obtained with

$$expr(L_1, L_2) \;=\; \lfloor (C_1' \parallel (\lceil L_1 \rceil^{[1]} \rhd C_2) \parallel (0^{[0]}; 1^{[0]}; \lceil L_2 \rceil^{[2]})\Phi) \setminus \{0^{[0]}, 1^{[0]}\} \rfloor_{[1,2]} \;.$$

We have $(a, \emptyset) \in Sf(\xrightarrow{a}\tau \mathbin{\Theta_{\{a\}}} \circ)$. To get $(a, \emptyset) \in Sf(expr(\xrightarrow{a}\tau, \circ))$ without getting $(a, \emptyset) \in Sf(expr(\mathsf{Stop}_{\{a\}}, \circ))$, $expr(L_1, L_2)$ must execute the a of at least one instance of $\xrightarrow{a}\tau$ before yielding the stable failure $(a, \emptyset)$. The execution of a resolves all choice operators in whose scope the instance is, so afterwards it is not in the scope of any choice operator. None of the remaining operators listed in the fourth claim can stop $\xrightarrow{a}\tau$ from diverging. So the stable failure $(a, \emptyset)$ is not produced, and we conclude that the fourth claim holds.

The last claim follows similarly using $minD(\xrightarrow{a}\tau \mathbin{\Theta_{\{a\}}} \xrightarrow{b}\tau) = \{ab\}$. To get $ab \in Div(expr(L_1, L_2))$ when $L_1 = \xrightarrow{a}\tau$ and $L_2 = \xrightarrow{b}\tau$ without getting it when $L_1 = \mathsf{Stop}_{\{a\}}$ or $L_2 = \mathsf{Stop}_{\{b\}}$, $expr(L_1, L_2)$ must execute both the a and the b before yielding it. However, if $expr(L_1, L_2)$ executes them one at a time, it diverges already when it has executed one of them. If it executes them simultaneously, then it incorrectly yields $ab \in Div(expr(\xrightarrow{a}\tau, \xrightarrow{b}\xrightarrow{b}\tau))$. With "$\rhd$" it is possible to switch the premature divergence off, but only after it has affected the divergence traces of the result. □

The previous two theorems yield immediately the following corollary.

Corollary 1 *The equivalence "$\approx_{\mathsf{shd}}$" is a congruence with respect to "Θ". The equivalence "$\approx_{\mathsf{shd},Stb}$" is a congruence with respect to "$\rhd$" and "Θ".*

Proof The operators from which "$\rhd$" and "Θ" are built in the theorems have the promised congruence properties. □

8 Universality of "$\|$" and "$\backslash$" up to (Infinite) Trace Equivalence

In Example 6 we saw that if an LTS operator may test an unbounded number of traces for producing a trace, then the operator may be unconstructible from commonly used LTS operators. Therefore, we now restrict ourselves to LTS operators that, to yield a trace, may test at most one trace of each of its arguments. We prove that *all* such LTS operators are $\approx_{Tr}$-constructible from "$\|$" and "$\backslash$". A natural assumption on infinite traces extends the result to $\approx_{Tr,Inf}$-constructibility.

The assumptions are not unduly restrictive. For instance, all LTS operators in Definition 4 obey them. Furthermore, our result indirectly covers the case where the operator may test a fixed finite number of traces. This is because, for instance, a unary LTS operator $op_1(L)$ that tests at most three traces of its argument could be built as a ternary LTS operator $op_3(L_1, L_2, L_3)$ and then declaring that $op_1(L) = op_3(L, L, L)$.

Before formalizing the assumptions, let us present a lemma that we will need.

Lemma 5 *The operator "ϕ" is $\approx_{Tr,Inf}$-constructible from "$\|$", "$\backslash$", and two LTS constant families.*

Proof Extend ϕ to a total function on Σ by letting $\phi(a) = a$ if $a \in \Sigma$ and $\phi(a)$ was undefined. Let $\phi(\Sigma) = \{\phi(a) \mid a \in \Sigma\}$, $X = \Sigma \cup \phi(\Sigma)$, and $\Sigma' = \{(a, X) \mid a \in \Sigma\}$. The goal of this choice of Σ' is to ensure that $\Sigma' \cap \Sigma = \Sigma' \cap \phi(\Sigma) = \emptyset$. By definition, $(a, X) = \{\{a\}, \{a, X\}\}$. We have $X \in \{a, X\} \in (a, X)$. Thus $(a, X) \in X$ would violate the axiom that all sets are well-founded.

For each $a \in \Sigma$, let $\phi_1(a) = (a, X)$ and $\phi_2(a, X) = \phi(a)$. Let C be the LTS with $\hat{s}_C \notin \Sigma$, $S_C = \Sigma \cup \{\hat{s}_C\}$, $\Sigma_C = \Sigma \cup \Sigma'$, and $\Delta_C = \{(\hat{s}_C, a, a) \mid a \in \Sigma\} \cup \{(a, (a, X), \hat{s}_C) \mid a \in \Sigma\}$. We have $(L \,\|\, C) \setminus \Sigma \approx_{Tr,Inf} \phi_1(L)$. The operator ϕ_2 can be implemented similarly. Then $\phi(L)$ is obtained as $\phi_2(\phi_1(L))$. □

Informally, our trace assumption is that, to yield a trace, an LTS operator may test at most one trace of each of its arguments. More precisely, for each Σ_1, ..., Σ_n, $Tr(op(L_1, \ldots, L_n))$ is determined by a set of tuples of the form $\langle \sigma_1, \ldots, \sigma_n; \sigma \rangle$, each tuple meaning that if $\sigma_i \in Tr(L_i)$ for $1 \leq i \leq n$, then $\sigma \in Tr(op(L_1, \ldots, L_n))$. This motivates the following definition.

Definition 10 An LTS operator $op(L_1, \ldots, L_n)$ is *trace-nice*, if and only if for each Σ_1, ..., Σ_n there is a set $\mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$ of tuples of the form $\langle \sigma_1, \ldots, \sigma_n; \sigma \rangle$, where op_Σ is the function that yields the alphabet of $op(L_1, \ldots, L_n)$ from Σ_1, ..., Σ_n, $\sigma \in op_\Sigma(\Sigma_1, \ldots, \Sigma_n)^*$, and $\sigma_i \in \Sigma_i^*$ for $1 \leq i \leq n$, such that

$$Tr(op(L_1, \ldots, L_n)) \;=\; \{\, \sigma \mid \forall i; 1 \leq i \leq n : \exists \sigma_i \in Tr(L_i) : \langle \sigma_1, \ldots, \sigma_n; \sigma \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n} \,\} .$$

Our corresponding definition for infinite traces requires that each infinite trace of the result of the operator arises step by step from traces and infinite traces of the arguments of the operator.

Definition 11 An LTS operator $op(L_1, \ldots, L_n)$ is *infinite-trace-nice*, if and only if it is trace-nice and

$$Inf(op(L_1, \ldots, L_n)) \;=\; \{\, a_1 a_2 \cdots \mid \forall i; 1 \leq i \leq n : \exists \sigma_i \in Tr(L_i) \cup Inf(L_i) : \forall j : \exists \sigma_{i,j} : \langle \sigma_{1,j}, \ldots, \sigma_{n,j}; a_1 \cdots a_j \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n} \wedge \sigma_{i,0} \sqsubseteq \sigma_{i,1} \sqsubseteq \sigma_{i,2} \sqsubseteq \cdots \sqsubseteq \sigma_i \,\} .$$

The set $\mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$ has some properties that we will exploit.

Lemma 6 *Assume that $\mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$ is like in Definition 10. We have*

- $\langle \varepsilon, \ldots, \varepsilon; \varepsilon \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$, *and*
- *if $\langle \sigma_1, \ldots, \sigma_n; \sigma \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$ and $\sigma' \sqsubseteq \sigma$, then there are σ'_i for $1 \leq i \leq n$ such that $\sigma'_i \sqsubseteq \sigma_i$ and $\langle \sigma'_1, \ldots, \sigma'_n; \sigma' \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$.*

Proof Because ε is a trace of every LTS, we have $\varepsilon \in Tr(op(\mathsf{Stop}_{\Sigma_1}, \ldots, \mathsf{Stop}_{\Sigma_n}))$. By Definition 10, for $1 \leq i \leq n$ there are $\sigma_i \in Tr(\mathsf{Stop}_{\Sigma_i}) = \{\varepsilon\}$ such that $\langle \sigma_1, \ldots, \sigma_n; \varepsilon \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$. This implies the first claim.

To prove the second claim, let L_i be the LTS that executes the trace σ_i and does nothing else. We have $\sigma \in Tr(op(L_1, \ldots, L_n))$. Because each prefix of any trace is a trace, also $\sigma' \in Tr(op(L_1, \ldots, L_n))$. So there are $\sigma'_i \in Tr(L_i)$ such that $\langle \sigma'_1, \ldots, \sigma'_n; \sigma' \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$. We have $\sigma'_i \sqsubseteq \sigma_i$, because L_i has no other traces. □

We write $\langle \sigma'_1, \ldots, \sigma'_n; \sigma' \rangle \preceq \langle \sigma_1, \ldots, \sigma_n; \sigma \rangle$ if and only if $\sigma' = \sigma$ and $\sigma'_i \sqsubseteq \sigma_i$ for $1 \leq i \leq n$. We are ready to present the main result of this section.

Theorem 6 *Let $op(L_1, \ldots, L_n)$ be an n-ary trace-nice LTS operator. It is $\approx_{Tr}$-constructible from "$\|$", "$\setminus$", and $2n + 3$ LTS constant families. If the operator is also infinite-trace-nice, then it is $\approx_{Tr,Inf}$-constructible from the listed operators and LTS constant families.*

Proof Let $\Sigma = op_\Sigma(\Sigma_1, \ldots, \Sigma_n)$, that is, the alphabet of $op(L_1, \ldots, L_n)$. Let

$$expr(L_1, \ldots, L_n) \;=\; \lfloor (C \parallel \lceil L_1 \rceil^{[1]} \parallel \cdots \parallel \lceil L_n \rceil^{[n]}) \setminus (\Sigma_1^{[1]} \cup \cdots \cup \Sigma_n^{[n]}) \rfloor_{[0]} \;,$$

where C is described soon. This expression uses $n+1$ instances of ϕ. By Lemma 5, they can be constructed from "$\|$", "$\setminus$", and $2n + 2$ LTS constant families.

The alphabet of C is $\Sigma_1^{[1]} \cup \cdots \cup \Sigma_n^{[n]} \cup \Sigma^{[0]}$. Each element of $\mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$ is a state of C. If $a \in \Sigma$ and $\langle \sigma_1, \ldots, \sigma_n; \sigma a \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$, then $\langle \sigma_1, \ldots, \sigma_n; \sigma \rangle$ is a state of C, and C has a transition labelled with $a^{[0]}$ from it to $\langle \sigma_1, \ldots, \sigma_n; \sigma a \rangle$. If s_1 and s_3 are states of C and $s_1 \preceq s_2 \preceq s_3$, then s_2 is a state of C. If $\langle \sigma'_1, \ldots, \sigma'_n; \sigma \rangle$ and $\langle \sigma_1, \ldots, \sigma_n; \sigma \rangle$ are states of C, $1 \leq i \leq n$, $a_i \in \Sigma_i$, $\sigma'_i a = \sigma_i$, and $\sigma'_j = \sigma_j$ when $1 \leq j \leq n$ and $j \neq i$, then C has a transition labelled with $a^{[i]}$ from $\langle \sigma'_1, \ldots, \sigma'_n; \sigma \rangle$ to $\langle \sigma_1, \ldots, \sigma_n; \sigma \rangle$. There are no other states and transitions in C. The initial state is $\hat{s} = \langle \varepsilon, \ldots, \varepsilon; \varepsilon \rangle$.

By Lemma 6, if $k > 0$ and $s_k = \langle \sigma_{1,k}, \ldots, \sigma_{n,k}; a_1 a_2 \cdots a_k \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$, then $\mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$ contains an $s_{k-1} = \langle \sigma_{1,k-1}, \ldots, \sigma_{n,k-1}; a_1 a_2 \cdots a_{k-1} \rangle$ such that $s_{k-1} \preceq \langle \sigma_{1,k}, \ldots, \sigma_{n,k}; a_1 a_2 \cdots a_{k-1} \rangle$. By the construction, s_{k-1} and s_k are states of C, and C has a path from s_{k-1} to s_k, where first $\sigma_{i,k-1}^{[i]}$ are extended to $\sigma_{i,k}^{[i]}$ in an arbitrary order, and then $(a_1 a_2 \cdots a_{k-1})^{[0]}$ is extended to $(a_1 a_2 \cdots a_{k-1} a_k)^{[0]}$. Furthermore, we may choose $s_0 = \langle \varepsilon, \ldots, \varepsilon; \varepsilon \rangle = \hat{s}$.

By Definition 10 and the construction, if $\sigma \in Tr(op(L_1, \ldots, L_n))$, then C has a state $s = \langle \sigma_1, \ldots, \sigma_n; \sigma \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$ for some $\sigma_1 \in Tr(L_1)$, …, $\sigma_n \in Tr(L_n)$. By the above discussion, C has a path from $\hat{s}$ to s such that $expr(L_1, \ldots, L_n)$ can execute it, yielding the trace σ. Thus $Tr(expr(L_1, \ldots, L_n)) \supseteq Tr(op(L_1, \ldots, L_n))$.

On the other hand, each state of C is of the form $\langle \sigma_1, \ldots, \sigma_n; \sigma \rangle$, where $\langle \sigma'_1, \ldots, \sigma'_n; \sigma \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$ for some $\sigma'_i \sqsubseteq \sigma_i$. It can only be reached such that

each L_i executes σ_i, and reaching it yields the trace σ. This is correct, because if $\sigma_i \in Tr(L_i)$, then also $\sigma_i' \in Tr(L_i)$. We conclude that $Tr(expr(L_1, \ldots, L_n)) \subseteq Tr(op(L_1, \ldots, L_n))$.

From now on, we assume that the operator is infinite-trace-nice.

If $expr(L_1, \ldots, L_n)$ can execute $a_1 a_2 \cdots$, then, for $j \geq 0$, the state immediately after the execution of a_j ($\hat{s}$ for $j = 0$) is of the form $\langle \sigma_{1,j}, \ldots, \sigma_{n,j}; a_1 \cdots a_j \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$. Let $1 \leq i \leq n$. Because the trace of L_i cannot become shorter when the execution proceeds further, we have $\sigma_{i,0} \sqsubseteq \sigma_{i,1} \sqsubseteq \sigma_{i,2} \sqsubseteq \cdots$. Let σ_i be the limit of $\sigma_{i,0}$, $\sigma_{i,1}$, ... The execution of $a_1 a_2 \cdots$ by $expr(L_1, \ldots, L_n)$ implies the execution of σ_i by L_i even if σ_i is infinite, so $\sigma_i \in Tr(L_i) \cup Inf(L_i)$. By Definition 11, $Inf(expr(L_1, \ldots, L_n)) \subseteq Inf(op(L_1, \ldots, L_n))$.

Finally, assume that $a_1 a_2 \cdots \in Inf(op(L_1, \ldots, L_n))$. For each $j > 0$, Definition 11 yields an $s_j = \langle \sigma_{1,j}, \ldots, \sigma_{n,j}; a_1 a_2 \cdots a_j \rangle \in \mathcal{T}_{\Sigma_1,\ldots,\Sigma_n}$ with certain properties. We may choose $s_0 = \langle \varepsilon, \ldots, \varepsilon; \varepsilon \rangle$. The construction guarantees the existence in C of a path from each s_j to s_{j+1} with precisely the labels that extend each $\sigma_{i,j}^{[i]}$ to $\sigma_{i,j+1}^{[i]}$ and $(a_1 a_2 \cdots a_j)^{[0]}$ to $(a_1 a_2 \cdots a_j a_{j+1})^{[0]}$. Together they constitute an infinite path that starts at $\hat{s}$, can be executed by $expr(L_1, \ldots, L_n)$ if each L_i can execute σ_i, and causes $expr(L_1, \ldots, L_n)$ to yield the infinite trace $a_1 a_2 \cdots$. So $Inf(expr(L_1, \ldots, L_n)) \supseteq Inf(op(L_1, \ldots, L_n))$. □

Instead of "$\|$" and "$\backslash$", the constructions in the proofs of Lemma 5 and Theorem 6 can be made using just the operator "$|\backslash|$" defined by $L_1 | \backslash | L_2 = (L_1 \| L_2) \backslash \{\Sigma_1 \cap \Sigma_2\}$. That is, it suffices to use the binary parallel composition operator that hides the actions that have been used for synchronization. This implies that also the CCS parallel composition and restriction operators suffice.

Let us conclude this section by speculating on the possibilities of obtaining a universality theorem for some stronger equivalence than "$\approx_{Tr,Inf}$".

Consider the simulation of $L_1 \| L_2$ with the construction used above, with $L_1 = L_2 = \overset{a}{\circ\!\!\longrightarrow\!\!\circ}$. While L_1 and L_2 execute their a-transitions synchronously in $L_1 \| L_2$, they execute them one at a time in $expr(L_1, L_2)$. With Φ, the construction could be made more faithful in this respect. The alphabet of C could contain a label for each necessary combination of $a_1 \in \Sigma_1 \cup \{\varepsilon\}$, ..., $a_n \in \Sigma_n \cup \{\varepsilon\}$, and $a \in \Sigma \cup \{\tau\}$, where those L_i whose a_i is ε do not participate, the remaining L_i execute their a_i synchronously, and a is the resulting action of $expr(L_1, \ldots, L_n)$. The elements of Σ_i would be transformed to the required labels of C with an instance of Φ.

Unfortunately, as such, this more faithful construction does not seem to give much benefit. Theorems 2 to 5 present many results that some LTS operator is not $\approx_{Tr,Sf}$-, not $\approx_{minD}$-, or not $\approx_{\mathsf{shd}}$-constructible from "$\|$", "Φ", and "$\backslash$". Therefore, to obtain a universal $\approx_{Tr,Sf}$-, $\approx_{minD}$-, or $\approx_{\mathsf{shd}}$-constructibility result, one or more additional LTS operators are needed.

To be interesting, the result should prove the constructibility of all LTS operators in a natural class. For instance, the operator "$;$" made "$+$" and "$\triangleright$" $\approx_{\mathsf{shd}}$- and $\approx_{\mathsf{shd},Stb}$-constructible. However, it can only be used at the front of an LTS. Therefore, it does not suffice for the operators "$\oplus$" and "$+_\circ$" defined by the following SOS rules:

$$\frac{L_1 \xrightarrow{\tau} L_1'}{L_1 \oplus L_2 \xrightarrow{\tau} L_1' +_\circ L_2} \qquad \frac{L_2 \xrightarrow{\tau} L_2'}{L_1 \oplus L_2 \xrightarrow{\tau} L_2' +_\circ L_1} \qquad \frac{L_1 \xrightarrow{\tau} L_1'}{L_1 +_\circ L_2 \xrightarrow{\tau} L_1' +_\circ L_2}$$

$$\frac{L_1 \mathrel{-a\!\!\rightarrow} L_1',\ a \neq \tau}{L_1 \oplus L_2 \mathrel{-a\!\!\rightarrow} L_1' \oplus L_2} \quad \frac{L_2 \mathrel{-a\!\!\rightarrow} L_2',\ a \neq \tau}{L_1 \oplus L_2 \mathrel{-a\!\!\rightarrow} L_1 \oplus L_2'} \quad \frac{L_1 \mathrel{-a\!\!\rightarrow} L_1',\ a \neq \tau}{L_1 +_\circ L_2 \mathrel{-a\!\!\rightarrow} L_1' \oplus L_2}\ .$$

We have $\searrow\!\circ\!\xrightarrow{a}\!\circ\!\xrightarrow{\tau}\!\circ \oplus \searrow\!\circ\!\xrightarrow{b}\!\circ$ and $\searrow\!\circ\!\xrightarrow{a}\!\circ\!\xrightarrow{\tau}\!\circ +_\circ \searrow\!\circ\!\xrightarrow{b}\!\circ$ may but $\searrow\!\circ\!\xrightarrow{a}\!\circ \oplus \searrow\!\circ\!\xrightarrow{b}\!\circ$ and $\searrow\!\circ\!\xrightarrow{a}\!\circ +_\circ \searrow\!\circ\!\xrightarrow{b}\!\circ$ may not deadlock after the trace a. So, even in the absence of divergences, "$\oplus$" and "$+_\circ$" do not respect any familiar abstract equivalence that preserves deadlocks. As a consequence, they might be deemed unnatural and thus irrelevant as a counterexample. However, the largely similar fact that $\searrow\!\circ\!\xrightarrow{\tau}\!\circ + \searrow\!\circ\!\xrightarrow{b}\!\circ \not\approx \searrow\!\circ + \searrow\!\circ\!\xrightarrow{b}\!\circ$ when "$\approx$" is "$\approx_{\mathsf{shd}}$", "$\approx_{\mathsf{wb}}$", "$\approx_{\mathsf{br}}$", or "$\approx_{Tr,Sf}$" has not led to the rejection of the "$+$" operator (except in CSP, where it and "$\rhd$" have been replaced by operators that are not sensitive to the initial stability issue). So the class of operators should contain "$+$" (and "$\rhd$"), but not "$\oplus$" and "$+_\circ$". It seems difficult to define such a class that would widely be considered as natural. One possibility could be to start with the set of operators that respect branching bisimilarity [4,8].

Leaving "$+$" and "$\rhd$" out helps. Indeed, as was mentioned in Section 1, a class of "CSP-like" operators has been defined. All operators in it are $\approx$-constructible from the classical CSP operators and "Θ", where "$\approx$" is any of the common CSP equivalences [17,7].

9 Conclusions

We proved that at the level of the alphabet, traces, and infinite traces, things are simple: every LTS operator that satisfies some reasonable mild assumptions is constructible from just "$\|$" and "$\backslash$". We demonstrated with examples and miscellaneous results that the simplicity is lost when moving to familiar equivalences that preserve more than "$\approx_{Tr,Inf}$".

The simplicity depends a bit on our convention that each LTS has its own alphabet. Lemma 5 says that the functional renaming operator "ϕ" is $\approx_{Tr,Inf}$-constructible from "$\|$" and "$\backslash$". Its proof relies on the availability of actions that are not in the alphabets of L and $\phi(L)$. If there were a global alphabet that is common to all LTSs, then L could have a transition for each action in it, and the construction in the proof would not have worked. We feel that failure of Lemma 5 because of running out of action names would have been an artifact.

If an equivalence is a congruence with respect to some LTS operators but not with respect to yet another LTS operator, then the latter is not constructible from the former up to the equivalence in question and all strictly stronger equivalences (Lemma 4). This often gives a simple way of proving that an LTS operator is not $\approx$-constructible, when "$\approx$" is some rather strong equivalence, such as "$\approx_{\mathsf{CFFD}}$" or weak bisimilarity. However, often unconstructibility arises already with a rather weak equivalence. Knowing that action prefix is not $\approx_{minD}$-constructible from the five operators listed in Theorem 2 is more than knowing that it is not $\approx_{\mathsf{CFFD}}$-constructible. To obtain such stronger unconstructibility results, many proofs in this publication relied on ad-hoc arguments.

Roughly speaking, LTS operators that are sensitive to the initial stability issue are not constructible from LTS operators that are not, when the equivalence preserves any kind of failures (this includes "$\approx_{\mathsf{shd}}$") or sufficient information on the branching structure (this includes weak bisimilarity). To facilitate seeing beyond the initial stability issue, the *stabilizing prefix* operator "$\tilde{;}$" was defined. It made the choice operator both $\approx_{\mathsf{CFFD}}$- and $\approx_{\mathsf{shd}}$-constructible and the interrupt operator

$\approx_{\mathsf{shd}}$- but not $\approx_{\mathsf{CFFD}}$-constructible. CFFD-construction of the interrupt operator requires that when its first argument is interrupted, it is switched off so strongly that it no longer can cause a divergence. This was obtained by also using the CSP throw operator "Θ".

When modelling some systems, LTS operators would be useful with which an LTS could be switched off and later switched on so that it continues from where it was. The operators "Θ" and "$\rhd$" allow switching off, but not switching on again. Unfortunately, switching on introduces a generalized version of the initial stability issue, as was illustrated in Section 8.

Acknowledgements

I thank Walter Vogler for the helpful discussions on "$\approx_{\mathsf{shd}}$" and other topics that we have had every now and then. I also thank the anonymous reviewers for their valuable comments, written in a very helpful way, on related work and other issues.

References

1. Aceto, L., Fokkink, W.J., Verhoef, C.: Structural Operational Semantics. Bergstra, J.A., Ponse, A., Smolka, S.A. (eds.): Handbook of Process Algebra, Chapter 3, Elsevier, The Netherlands (2001) 197–292
2. Arnold, A.: Finite Transition Systems. Prentice-Hall, Englewood Cliffs, NJ, USA (1994)
3. Austry, D., Boudol, G.: Algèbre de Processus et Synchronisation. Theoretical Computer Science 30 (1984) 91–131
4. Bloom, B.: Structural Operational Semantics for Weak Bisimulations. Theoretical Computer Science 146(1&2) (1995) 25–68
5. Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14 (1987) 25–59
6. Boudol, G.: Notes on Algebraic Calculi of Processes. Apt, K. (ed.): Logics and Models of Concurrent Systems, NATO ASI Series F13, Springer (1985) 261–303
7. Gibson-Robinson, T.: Efficient Simulation of CSP-Like Languages. Welch, P.H., Barnes, F.R.M., Broenink, J.F., Chalmers, K., Pedersen, J.B., Sampson, A.T. (eds.): Communicating Process Architectures 2013. Open Channel Publishing Ltd., Bicester (2013) 185–204
8. van Glabbeek, R.: On Cool Congruence Formats for Weak Bisimulations. Theoretical Computer Science 412(28) (2011) 3283–3302
9. van Glabbeek, R.: Musings on Encodings and Expressiveness. Luttik, B., Reniers, M.A. (eds.): Proc. Combined 19th International Workshop on Expressiveness in Concurrency and 9th Workshop on Structured Operational Semantics, Electronic Proceedings in Theoretical Computer Science 89 (2012) 81–98
10. van Glabbeek, R., Weijland, W.: Branching Time and Abstraction in Bisimulation Semantics (Extended Abstract). Ritter, G. (ed.): Proc. IFIP International Conference on Information Processing '89, North-Holland (1989) 613–618
11. Gorla, D.: Towards a Unified Approach to Encodability and Separation Results for Process Calculi. Information and Computation 208(9) (2010) 1031–1053
12. Kaivola, R., Valmari, A.: The Weakest Compositional Semantic Equivalence Preserving Nexttime-less Linear Temporal Logic. Cleaveland, R. (ed.): CONCUR '92, Third International Conference on Concurrency Theory, Lecture Notes in Computer Science 630 (1992) 207–221
13. Karsisto, K.: A New Parallel Composition Operator for Verification Tools. Dr.Tech. Thesis, Tampere University of Technology Publications 420, Tampere, Finland (2003)
14. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems, Volume I: Specification. Springer-Verlag (1992)
15. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs, NJ, USA (1989)

16. Rensink, A., Vogler, W.: Fair Testing. Information and Computation 205 (2007) 125–198
17. Roscoe, A.W.: Understanding Concurrent Systems. Springer, Heidelberg, Germany (2010)
18. de Simone, R.: Higher-Level Synchronising Devices in Meije-SCCS. Theoretical Computer Science 37 (1985) 245–267
19. Valmari, A.: Failure-Based Equivalences Are Faster Than Many Believe. Desel, J. (ed.): Structures in Concurrency Theory 1995, Springer-Verlag Workshops in Computing series (1995) 326–340
20. Valmari, A.: The Weakest Deadlock-Preserving Congruence. Information Processing Letters 53 (1995) 341–346
21. Valmari, A.: Stubborn Set Methods for Process Algebras. Peled, D.A., Pratt, V.R., Holzmann, G.J. (eds.): Partial Order Methods in Verification: DIMACS Workshop, DIMACS Series in Discrete Mathematics and Theoretical Computer Science Vol. 29, American Mathematical Society (1997) 213–231
22. Valmari, A.: A Chaos-Free Failures Divergences Semantics with Applications to Verification. Davies, J., Roscoe, B., Woodcock, J. (eds.): Millennial Perspectives in Computer Science, Proceedings of the 1999 Oxford–Microsoft Symposium in Honour of sir Tony Hoare, Palgrave (2000) 365–382
23. Valmari, A.: All Linear-Time Congruences for Familiar Operators. Logical Methods in Computer Science 9 (2013) 11:1–34
24. Valmari, A., Tienari, M.: Compositional Failure-Based Semantic Models for Basic LOTOS. Formal Aspects of Computing 7(4) (1995) 440–468