Distribution of Behaviour into parallel communicating subsystems

Distribution of Behaviour into Parallel Communicating Subsystems

Omar al Duhaiby Jan Friso Groote
Eindhoven University of Technology
Eindhoven, The Netherlands
o.z.alzuhaibi@tue.nl j.f.groote@tue.nl

The process of decomposing a complex system into simpler subsystems has been of interest to computer scientists over many decades, most recently for the field of distributed computing. In this paper, motivated by the desire to distribute the process of active automata learning onto multiple subsystems, we study the equivalence between a system and the total behaviour of its decomposition which comprises subsystems with communication between them. We show synchronously- and asynchronously-communicating decompositions that maintain branching bisimilarity, and we prove that there is no decomposition operator by our definition that maintains divergence-preserving branching bisimilarity over all LTSs.

1 Introduction

The process of decomposing a complex system into simpler subsystems is the cornerstone of behavioural analysis regardless of where it is applied, to the atom or to the human psyche. Studying the relationship between a complex system and the total behaviour of its decomposition is the subject matter of this paper. However, instead of atoms or human brains, in the field of formal methods, we simply dissect automata. This paper studies how the behaviour of a Labelled Transition System (LTS) can be distributed into a parallel (de)composition of communicating subsystems while maintaining behavioural equivalence.

Motivation This work was motivated by a case study in the industry [2] based on which we pursued the possibility of applying the active model learning technique [1] in parallel. If it were possible at all, then the system under learning must be equivalent to the parallel decomposition on which the learning is distributed.

The primary decomposition theorem by Krohn and Rhodes states that any automaton can be decomposed into a cascaded product of simpler automata such that the automaton is homomorphic to its decomposition [7]. And in 1998, Milner and Moller introduced a semantics of parallel decompositions comprising non-communicating subsystems [8], and they proved that any finite system of behaviour can be decomposed into a unique set of prime parallel non-communicating subsystems. In this paper, we ask ourselves whether any behaviour can be split into *communicating* subsystems, each determined by an action set.

Contribution We define two decompositions of parallel communicating subsystems, one synchronous and the other asynchronous, and we prove that both decompositions maintain branching bisimilarity [4] with the source automaton. We also prove that there is no way of decomposing an automaton (under certain conditions) such that it is divergent-preserving branching-bisimilar [3] to the resulting decomposition.

Outline The outline of this paper is as follows. Section 2 introduces the preliminaries. Section 3 defines and discussed the general decomposition operator on which we base our arguments. Section 4 defines two decompositions of communicating subsystems, one for synchronous communication and the other for asynchronous communication, and proves that each maintains a branching bisimulation relation with the source automaton. Finally, Section 5 contains the proof that there is no way of decomposing an automaton, through our general decomposition operator, such that it maintains divergence preserving branching bisimulation with its decomposition.

Acknowledgement We wish to thank Rick Erkens, Joshua Moerman and Thomas Neele for sharing their knowledge and motivation.

2 Preliminaries

In this section, we present the preliminaries of labelled transition systems, the synchronous product and bisimulation relations, aided by [5]. We start with the definition of a labelled transition system (LTS).

Definition 2.1 (LTS)**.** We define our LTS as a four-tuple $(S, \Sigma, \rightarrow, s_0)$ where:

- S is a non-empty finite set of states.
- Σ is the alphabet, also referred to as the action set.
- $\rightarrow \subseteq S \times \Sigma \times S$ is a transition relation.
- s_0 is the initial state.

We use the notation $x \xrightarrow{a} y$ to express a transition with action a from state x to state y. This and variations of it are formally defined as follows.

Definition 2.2 (Transition Relation)**.** Let $(S, \Sigma, \rightarrow, s_0)$ be an LTS with $s, s' \in S$ and $a \in \Sigma \cup \{\tau\}$, where τ is the internal/unobservable action. Then:

$s \xrightarrow{a} s'$	iff $\langle s, a, s' \rangle \in \rightarrow$.
$s \xrightarrow{a}$	iff there is an s' such that $s \xrightarrow{a} s'$.
$s \not\xrightarrow{a}$	iff there is no s' such that $s \xrightarrow{a} s'$.
$s \xrightarrow{a}^{*} s_n$	iff there are $s_1, s_2, \ldots, s_n \in S$ such that $s \xrightarrow{a} s_1 \xrightarrow{a} s_2 \xrightarrow{a} \cdots \xrightarrow{a} s_n$.
$s \xrightarrow{a}^{\omega}$	iff there are $s_1, s_2, \ldots \in S$ such that $s \xrightarrow{a} s_1$ and for all $i \in \mathbb{N}$, $s_i \xrightarrow{a} s_{i+1}$.

Next, we define complementary actions, i.e., actions on which communicating systems synchronise. Then we define the synchronous product of two automata, and show what role complementary actions play in computing it.

Definition 2.3 (Co-actions)**.** For an arbitrary action a, the action $\overline{a}$ (read as *a bar*) is called its co-action. Also, $\overline{(\overline{a})} = a$. We say that actions a and $\overline{a}$ are *complementary* to each other and we call them a pair of *complementary actions*.

We lift this operator to sets of actions such that $\overline{\Sigma} = \{\overline{a} \mid a \in \Sigma\}$.

Definition 2.4 (Synchronous Product of two LTSs)**.** The synchronous product of two LTSs $(S_1, \Sigma_1, \rightarrow_1, q_0) \times (S_2, \Sigma_2, \rightarrow_2, r_0)$ is the tuple $(S_1 \times S_2, \Sigma_x, \rightarrow_x, (q_0, r_0))$ where $\Sigma_x = (\Sigma_1 \cup \Sigma_2) \setminus \{a, \overline{a} \mid a \in \Sigma_1 \wedge \overline{a} \in \Sigma_2\}$.
The transition relation $\rightarrow_x \subseteq (S_1 \times S_2) \times \Sigma_x \times (S_1 \times S_2)$ is defined as follows:

$$\begin{cases} (s,t) \xrightarrow{a} (s',t) & \text{iff } a \in \Sigma_1 \wedge \overline{a} \notin \Sigma_2 \wedge s \xrightarrow{a}_1 s', \\ (s,t) \xrightarrow{a} (s,t') & \text{iff } a \in \Sigma_2 \wedge \overline{a} \notin \Sigma_1 \wedge t \xrightarrow{a}_2 t', \text{ and} \\ (s,t) \xrightarrow{\tau} (s',t') & \text{iff } a \in \Sigma_1 \wedge \overline{a} \in \Sigma_2 \wedge s \xrightarrow{a}_1 s' \wedge t \xrightarrow{\overline{a}}_2 t', \end{cases}$$

where τ is the unobservable action.

Next, we define two notions of behavioural equivalence.

Definition 2.5 (Branching bisimulation)**.** Given an LTS $(S, \Sigma, \rightarrow, s_0)$ and a relation $\mathscr{R} \subseteq S \times S$. We call $\mathscr{R}$ a branching bisimulation relation iff for all states $s, t \in S$ such that $\langle s, t \rangle \in \mathscr{R}$, it holds that:

1. if $s \xrightarrow{a} s'$, then:
   - $a = \tau$ and $\langle s', t \rangle \in \mathscr{R}$; or
   - $t \xrightarrow{\tau}^* t' \xrightarrow{a} t''$, $\langle s, t' \rangle \in \mathscr{R}$ and $\langle s', t'' \rangle \in \mathscr{R}$.
2. Symmetrically, if $t \xrightarrow{a} t'$, then:
   - $a = \tau$ and $\langle s, t' \rangle \in \mathscr{R}$; or
   - $s \xrightarrow{\tau}^* s' \xrightarrow{a} s''$, $\langle s', t \rangle \in \mathscr{R}$ and $\langle s'', t' \rangle \in \mathscr{R}$.

Two states s and t are branching *bisimilar*, denoted $s \leftrightarrow_b t$ iff there is a branching bisimulation relation $\mathscr{R}$ such that $\langle s, t \rangle \in \mathscr{R}$. Two LTSs P and Q are branching bisimilar, denoted $P \leftrightarrow_b Q$, iff their initial states are.

A state s with $s \xrightarrow{\tau}^{\omega}$ is called *divergent*. Hence, a state with a τ loop is also called divergent. Branching bisimulation does not preserve divergence, i.e., a divergent state can be branching bisimilar to a non-divergent one. Therefore, a stronger equivalence relation, namely divergence-preserving branching bisimulation, is defined below.

Definition 2.6 (Divergence-preserving branching bisimulation)**.** Given an LTS $(S, \Sigma, \rightarrow, s_0)$ and a relation $\mathscr{R} \subseteq S \times S$. We call $\mathscr{R}$ a divergence-preserving branching bisimulation relation iff it is a branching bisimulation relation and for all states $s, t \in S$ with $\langle s, t \rangle \in \mathscr{R}$, there is an infinite sequence $s \xrightarrow{\tau} s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau}$ with $\langle s_i, t \rangle \in \mathscr{R}$ for all $i > 0$ iff there is an infinite sequence $t \xrightarrow{\tau} t_1 \xrightarrow{\tau} t_2 \xrightarrow{\tau}$ and $\langle s, t_i \rangle \in \mathscr{R}$ for all $i > 0$.

Two states s and t are divergence-preserving branching bisimilar, denoted $s \leftrightarrow_{db} t$ iff there is a divergence-preserving branching bisimulation relation $\mathscr{R}$ such that $\langle s, t \rangle \in \mathscr{R}$. Two LTSs P and Q are divergence-preserving branching bisimilar, denoted $P \leftrightarrow_{db} Q$, iff their initial states are.

3 The Decomposition Operation

We define a decomposition operation in general to be a function transforming a single LTS, given two disjoint actions sets, into two LTSs.

Definition 3.1 (General Decomposition Operation)**.** Given an LTS M with alphabet Σ and given two alphabets Σ_1, Σ_2 such that $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$, we call G a *general decomposition operator* iff $G(M, \Sigma_1, \Sigma_2) = (M_1, M_2)$ such that M_1 has alphabet Σ_{M_1} with $\Sigma_1 \subseteq \Sigma_{M_1}$ and $\Sigma_{M_1} \cap \Sigma_2 = \emptyset$, and likewise, M_2 has alphabet Σ_{M_2} with $\Sigma_2 \subseteq \Sigma_{M_2}$ and $\Sigma_{M_2} \cap \Sigma_1 = \emptyset$.

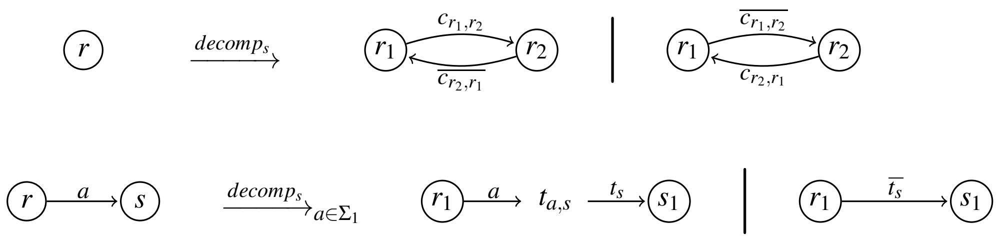

Figure 1: The two patterns that delineate the operator $decomp_s$ (Definition 4.1).

We refer to a method of decomposing automata as a *decomposition operation* whereas the result of such transformation is called a *decomposition*. A decomposition comprises two or more automata. This transformation is depicted in Figure 2. Throughout the paper, we compare LTSs to the synchronous product of the decomposition, and if a a certain bisimulation relation holds between these two, then we say that the operation *maintains* that relation.

Recursive decomposition. Note that in Definition 3.1, the alphabets over which an automaton is decomposed can be empty. This means that the operation can be recursively applied, by decomposing the resulting subsystems, infinitely many times.

4 Branching Bisimilar Decompositions

In this section, we define two decomposition operations that are designed to maintain branching bisimilarity, and we actually prove that they do. The first one ($decomp_s$) decomposes into synchronously communicating subsystems while the second ($decomp_a$) decomposes into asynchronously communicating ones.

4.1 Decomposing into Synchronous Subsystems

We define the decomposition of synchronous subsystems, summarised in Figure 1 in two patterns; the top dictates the decomposition of every state in the source LTS while the bottom dictates the decomposition of every transition. An omitted third pattern is symmetric to the second such that the transition's label simply belongs to the second subsystem rather than the first.

Definition 4.1 (Synchronous Decomposition Operation)**.** Given an LTS $M = (S,\Sigma,\rightarrow,q)$ and two alphabets Σ_1,Σ_2 such that $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$, then we can decompose M over Σ_1 and Σ_2 by applying the following operation:

$decomp_s(M,\Sigma_1,\Sigma_2) = (M_1,M_2)$ where:

1. $M_1 = (S_C \cup S_{T_1},\Sigma_1 \cup \Sigma_{S_1},\rightarrow_1,(q,1))$
   $M_2 = (S_C \cup S_{T_2},\Sigma_2 \cup \Sigma_{S_2},\rightarrow_2,(q,1))$.
2. For every state s in S, we introduce two states $(s,1),(s,2) \in S_C$:

$$S_{C_1} = \{(s,1) \mid s \in S\} \qquad S_{C_2} = \{(s,2) \mid s \in S\} \qquad S_C = S_{C_1} \cup S_{C_2} \tag{1}$$

Notation. Tuple-states of the form (s,i) such as $(s,1)$ and $(s,2)$ are shortened to s_i. Therefore, it is to be held throughout the paper that s_i is derived from s rather than it being a completely unrelated symbol to s.

3. The set of c-actions is defined as follows:

$$\Sigma_C = \{c_{s_1,s_2}, c_{s_2,s_1} \mid s_1 \in S_{C_1}, s_2 \in S_{C_2}\} \tag{2}$$

4. The sets of t actions and t states are defined as follows:

$$\begin{aligned} \Sigma_{T_1} &= \{t_{s_1} \mid s_1 \in S_C\} & \Sigma_{T_2} &= \{t_{s_2} \mid s_2 \in S_C\} \\ S_{T_1} &= \{t_{a,s_1} \mid a \in \Sigma_1, s_1 \in S_{C_1}\} & S_{T_2} &= \{t_{a,s_2} \mid a \in \Sigma_2, s_2 \in S_{C_2}\} \end{aligned} \tag{3}$$

5. The complete sets of actions of M_1 and M_2 are respectively defined as:

$$\Sigma_{S_1} = \Sigma_{T_1} \cup \Sigma_C \cup \overline{\Sigma_{T_2}} \qquad \Sigma_{S_2} = \Sigma_{T_2} \cup \overline{\Sigma_C} \cup \overline{\Sigma_{T_1}} \tag{4}$$

6. The transition relations $\rightarrow_i \subseteq (S_C \cup S_{T_i}) \times (\Sigma_i \cup \Sigma_{S_i}) \times (S_C \cup S_{T_i})$ are defined as follows. For $i, j \in \{1,2\}$ and $i \neq j$, $\rightarrow_i$ is the minimal relation satisfying the following:

   (a) For all $s \in S$ and for all $c_{s_i,s_j}, c_{s_j,s_i} \in \Sigma_C$:

$$s_i \xrightarrow{c_{s_i,s_j}}_i s_j \qquad\qquad s_i \xrightarrow{\overline{c_{s_i,s_j}}}_j s_j \tag{5}$$

   (b) For all $s, s' \in S$, and all $a \in \Sigma_i$, if $s \xrightarrow{a} s'$, then:

$$\begin{aligned} s_i \xrightarrow{a}_i t_{a,s'_i} &\xrightarrow{t_{s'_i}}_i s'_i \\ s_i &\xrightarrow{\overline{t_{s'_i}}}_j s'_i \end{aligned} \tag{6}$$

Two classes of actions are introduced, c-actions and t-actions. The c-actions come in pairs, and they resemble passing a control token between M_1 and M_2. For instance, looking at Figure 2, when, at some state $r \in S$ for which a pair of states $r_1, r_2 \in S_C$ exists in both M_1 and M_2, and control is to be passed from M_1 to M_2, then a pair of complementary c actions synchronises, namely, actions c_{r_1,r_2} and $\overline{c_{r_1,r_2}}$, to produce a synchronous transition in both machines from r_1 to r_2. Likewise, actions c_{r_2,r_1} and $\overline{c_{r_2,r_1}}$ synchronise to pass control in the opposite direction from M_2 to M_1.

The t-actions are introduced to synchronise transitions occurring in one machine with the other. In addition, they require the introduction of t-states. Observe Figure 2 where an a_1 transition occurs in M_1. The aim is the transition $r_1 \xrightarrow{a_1} s_1$, but in order to synchronise this with M_2, we introduce a middle state $t_{a_1,s_1} \in S_{T_1}$ from which the only possible transition is $t_{a_1,s_1} \xrightarrow{t_{s_1}} s_1$ which synchronises with the transition $r_1 \xrightarrow{\overline{t_{s_1}}} s_1$ in M_2.

The operation ($decomp_s$) can be summarised by two patterns shown in Figure 1; the top pattern applies to each state and the bottom one applies to each transition.

Computing the Synchronous Product. For a decomposition (M_1, M_2) by Definition 4.1, the synchronous product $M_x = M_1 \times M_2$ is the LTS $(S_x, \Sigma_1 \cup \Sigma_2, \rightarrow_x, (q_1, q_1))$, where:

$$\begin{aligned} S_x = S_1 \times S_2 &= (S_C \cup S_{T_1}) \times (S_C \cup S_{T_2}) \\ &= (S_C \times S_C) \cup (S_{T_1} \times S_{T_2}) \\ &\quad \cup (S_{T_1} \times S_C) \cup (S_C \times S_{T_2}) \end{aligned} \tag{7}$$

with $\Sigma_{S_1}, \Sigma_{S_2}, S_{T_1}, S_{T_2}$ being sets introduced by $decomp_s$. The transition relation $\rightarrow_x$ is defined as follows for $i, j \in \{1,2\}$ and $i \neq j$:

1. if $s \xrightarrow{a} s'$ and $a \in \Sigma_i$ then by (6) there is a state $t_{a,s'_i} \in S_{T_i}$ and a pair of complementary actions $t_{s'_i}, \overline{t_{s'_i}} \in \Sigma_{S_i}$ such that:

$$(s_i, s_i) \xrightarrow{a}_x s_a \xrightarrow{\tau}_x (s'_i, s'_i), \tag{8}$$

$$\text{where } s_a = \begin{cases} (t_{a,s'_i}, s_i) & \text{if } i = 1, \\ (s_i, t_{a,s'_i}) & \text{if } i = 2. \end{cases}$$

2. For all $s \in S$, there exist $c_{s_i,s_j}, c_{s_j,s_i} \in \Sigma_C$ such that, by (5), $s_i \xrightarrow{c_{s_i,s_j}}_i s_j$ and $s_i \xrightarrow{\overline{c_{s_i,s_j}}}_j s_j$, and thus:

$$(s_i, s_i) \xrightarrow{\tau}_x (s_j, s_j) \tag{9}$$

4.2 Proof that the synchronous decomposition operation maintains branching bisimulation

In this subsection, we show an application of $decomp_s$ (Definition 4.1) to a sample LTS, we demonstrate that $decomp_s$ maintains branching bisimilarity, and then we prove that branching bisimilarity is maintained through any and all applications of $decomp_s$.

Figure 2 shows the LTS at the left side and its decomposition at the right side. The two patterns shown in Figure 1 can be applied directly to this LTS. The top pattern applies twice, once per state, and the bottom pattern applies three times, once per transition.

Next, we compute the synchronous product and form one LTS shown at the right of Figure 3. The nodes are divided into two equivalence classes, top and bottom. The states in the top class are branching bisimilar to state r whereas the states in the bottom one are branching bisimilar to state s.

The following proves the branching bisimilarity and thus proves that there is a way of decomposing an LTS such that branching bisimilarity is maintained.

Theorem 4.2. Given an LTS $M = (S, \Sigma, \rightarrow, s_0)$ and two alphabets Σ_1, Σ_2 such that $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$, and given an LTS $M_x = M_1 \times M_2$ where $(M_1, M_2) = decomp_s(M)$ by Definition 4.1, then $M \leftrightarrow_b M_x$.

Proof. Let $M_1 = (S_C \cup S_{T_1}, \Sigma_1 \cup \Sigma_{S_1}, \rightarrow_1, q_1)$ and $M_2 = (S_C \cup S_{T_2}, \Sigma_2 \cup \Sigma_{S_2}, \rightarrow_2, q_2)$.

Define a relation $\mathscr{R} \subseteq S \times ((S_C \cup S_{T_1}) \times (S_C \cup S_{T_2}))$ with $\mathscr{R} = \{\langle s, (s_n, s_n)\rangle, \langle r', (t_{a,r'_n}, r_n)\rangle, \langle r', (r_n, t_{a,r'_n})\rangle \mid s, r, r' \in S, n \in \{1,2\}, a \in \Sigma, r \xrightarrow{a} r'\}$. We prove that $\mathscr{R}$ is a branching bisimulation relation through the following cases:

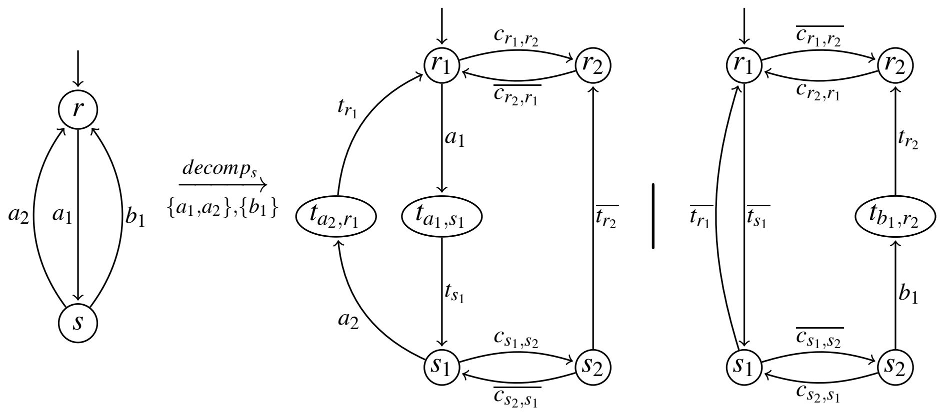

Figure 2: Example of synchronous decomposition operation of Definition 4.1.

1. Consider a pair $\langle s, s_x \rangle = \langle s, (s_n, s_n) \rangle$ where $n \in \{1,2\}$.
   (a) Assume $s \xrightarrow{a} s'$. Then we have two cases:
      i. $a \in \Sigma_1$. Then, by (8), $s_x \xrightarrow{a}_x (t_{a,s'_n}, s_n)$. We see that $\langle s', (t_{a,s'_n}, s_n) \rangle \in \mathscr{R}$.
      ii. $a \in \Sigma_2$. Then, by (8), $s_x \xrightarrow{a}_x (s_n, t_{a,s'_n})$. We see that $\langle s', (s_n, t_{a,s'_n}) \rangle \in \mathscr{R}$.
   (b) Assume $s_x \xrightarrow{a}_x s'_x$. Then we have the following three cases:
      i. $a \in \Sigma_1 \wedge \overline{a} \notin \Sigma_2$, then this transition is only possible, by definition, through the transition $s \xrightarrow{a} s'$ for some s' such that $s'_x \overset{(8)}{=} (t_{a,s'}, s_1)$. We see that $\langle s', s'_x \rangle \in \mathscr{R}$.
      ii. $a \in \Sigma_2 \wedge \overline{a} \notin \Sigma_1$. This is a symmetric case where $s \xrightarrow{a} s'$ and $s'_x \overset{(8)}{=} (s_2, t_{a,s'})$. We see that $\langle s', s'_x \rangle \in \mathscr{R}$.
      iii. $a \in \Sigma_1 \wedge \overline{a} \in \Sigma_2$, then the only transition possible is the τ transition of (9). Then $s'_x = (s_m, s_m)$ where $m \in \{1,2\}$ and $m \neq n$. We see that $\langle s, s'_x \rangle \in \mathscr{R}$.
2. Consider a pair $\langle r, r_x \rangle = \langle s', (t_{a,s'_n}, s_n) \rangle$ where $n \in \{1,2\}$, $a \in \Sigma$ and $s \xrightarrow{a} s'$.
   (a) Assume $r \xrightarrow{a} r'$. Then we show that $r_x \xrightarrow{\tau}_x r'_x$ and $r'_x \xrightarrow{a}_x r''_x$ and $\langle r, r'_x \rangle \in \mathscr{R}$ and $\langle r', r''_x \rangle \in \mathscr{R}$. We do this for $a \in \Sigma_1$. The case for $a \in \Sigma_2$ is symmetric.
      i. $r'_x \overset{(8)}{=} (s'_2, s'_2)$. We see that $\langle r, r'_x \rangle \in \mathscr{R}$.
      ii. Since $r = s'$ and $r \xrightarrow{a} r'$, then by (8), there exists a state r''_x such that $r'_x \xrightarrow{a}_x r''_x$, and $r''_x = (t_{a,s''_2}, s'_2)$, where $s'' = r'$. We see that $\langle r', r''_x \rangle \in \mathscr{R}$.
   (b) Assume $r_x \xrightarrow{a}_x r'_x$. By the definition of $\rightarrow_x$, it is only possible that a is a τ action and that $n = 1$. Thus, $r'_x \overset{(8)}{=} (s'_2, s'_2)$. We see that $\langle r', r''_x \rangle \in \mathscr{R}$.
3. Consider a pair $\langle r, r_x \rangle = \langle s', (s_n, t_{a,s'_n}) \rangle$ where $n \in \{1,2\}$, $a \in \Sigma$ and $s \xrightarrow{a} s'$. This case is symmetric to Case 2.

$\square$

Corollary 4.3. It follows from Theorem 4.2 that there is a universal way of decomposing an LTS M using a general synchronous decomposition operator (Definition 3.1) such that M is branching-bisimilar to the synchronous product of its decomposition.

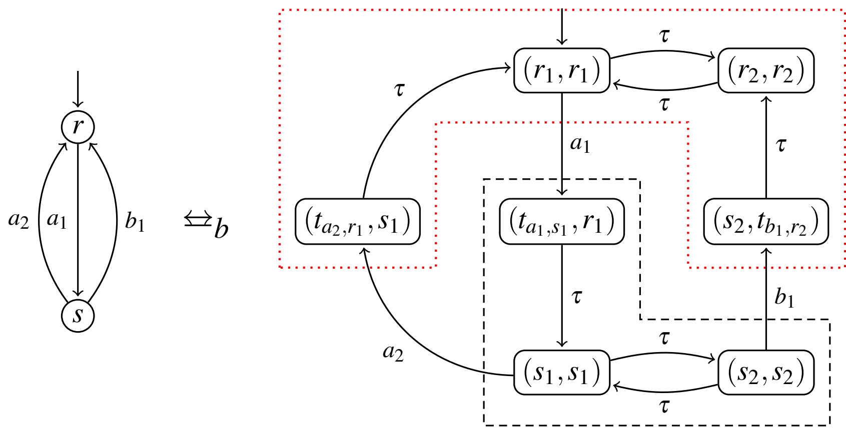

Figure 3: Showing branching bisimulation on the example of Figure 2.

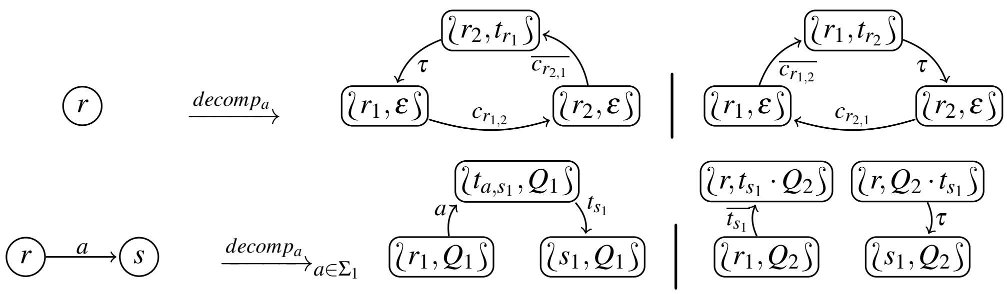

Figure 4: The two patterns that delineate the $decomp_a$ operator (Definition 4.5).

4.3 Decomposing into Asynchronous Subsystems

We define a new decomposition operation ($decomp_a$) such that the communication between subsystems is asynchronous. We assign each subsystem a queue that stores received messages until they are consumed. An action of sending such a message does synchronise, however, with the queue of the opposite side receiving it. The operation $decomp_a$ is summarised in Figure 4.

Definition 4.4 (LTS with Queue)**.** A queue is an ordered-list of actions. An LTS with a queue is a transition system of the shape $(S \times Q, \Sigma, \rightarrow, s_0)$. A state in $S \times Q$ holds the contents of the queue Q and is written as $\wr s, Q \wr$.

Elements in a queue are concatenated using the $\cdot$ operator. Appending an element m to the back of a queue Q produces the queue $m \cdot Q$, while $Q \cdot m$ represents the queue with m in the front. The symbol ε represents the empty queue.

Definition 4.5 (Decomposing into asynchronous subsystems)**.** Given an LTS $M = (S, \Sigma, \rightarrow, r_0)$ and two alphabets Σ_1, Σ_2 such that $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$, then we can decompose M over Σ_1 and Σ_2 by applying the following operation:

$decomp_a(M, \Sigma_1, \Sigma_2) = (M_1, M_2)$ where, for $i, j \in \{1, 2\}$ and $i \neq j$, M_i is an LTS with a queue (Definition 4.4) defined as follows:

1. $M_i = ((S_C \cup S_{T_i}, Q_i), \Sigma_i \cup \Sigma_{S_i}, \rightarrow_i, r_1)$

2. For every state in S, we introduce a pair of states $s_1, s_2 \in S_C$, a pair of c-actions, and a pair of t-actions:

$$\begin{aligned} S_{C_i} &= \{s_i \mid s \in S\} & S_C &= S_{C_1} \cup S_{C_2} \\ \Sigma_{C_{i,j}} &= \{c_{s_{i,j}} \mid s \in S\} & \Sigma_{T_i} &= \{t_{s_i} \mid s \in S\} \end{aligned} \tag{10}$$

3. Sets of t-states are defined as follows:

$$S_{T_i} = \{t_{a,s_i} \mid a \in \Sigma_i, s_i \in S_{C_i}\} \tag{11}$$

4. Sets of synchronous actions are defined as follows:

$$\Sigma_{S_i} = \Sigma_{T_i} \cup \overline{\Sigma_{T_j}} \cup \Sigma_{C_{i,j}} \cup \overline{\Sigma_{C_{j,i}}} \tag{12}$$

5. The transition relation $\rightarrow_i \subseteq (S_C \cup S_{T_i}) \times Q_i \times (\Sigma_i \cup \Sigma_{S_i}) \times (S_C \cup S_{T_i}) \times Q_i$ is the minimal relation satisfying the following:

   (a) For all $s \in S$:

$$\wr s_i, Q_i \wr \xrightarrow{c_{s_{i,j}}}_i \wr s_j, Q_i \wr \qquad \wr s_i, Q_i \wr \xrightarrow{\overline{c_{s_{i,j}}}}_j \wr s_i, t_{s_j} \cdot Q_i \wr \tag{13}$$

   (b) For all $s, s' \in S$, and all $a \in \Sigma_i$, if $s \xrightarrow{a} s'$, then:

$$\begin{aligned} \wr s_i, Q_i \wr \xrightarrow{a}_i \wr t_{a,s'_i}, Q_i \wr \xrightarrow{t_{s'_i}}_i \wr s'_i, Q_i \wr \\ \wr s_i, Q_i \wr \xrightarrow{\overline{t_{s'_i}}}_j \wr s_i, t_{s'_i} \cdot Q_i \wr \end{aligned} \tag{14}$$

   (c) Consuming an element from the front of a queue is an internal transition of the form:

$$\wr s, Q \cdot t_{s'} \wr \xrightarrow{\tau} \wr s', Q \wr \tag{15}$$

We see in (13) that the two automata synchronise on action $c_{s_{i,j}}$. The effect is a message sent from M_i and received in the queue of M_j. The same occurs in (14). Moreover, this makes sending messages only possible when both machines are in sync, i.e., on the same state s_i.

4.4 Proof that the Asynchronous Decomposition Operation Maintains Branching Bisimulation

In this subsection, similar to Section 4.2, we prove that the asynchronous decomposition operation ($decomp_a$) also maintains branching bisimilarity. Figure 5 shows the result of applying $decomp_a$ to the same example behaviour as in Figure 2. In Figure 6, we compute the synchronous product of the decomposition of Figure 5 and then divide the nodes of the product into two equivalence classes, top and bottom. The states in the top class are branching bisimilar to state r whereas the states in the bottom on are branching bisimilar to state s.

Next, we prove that any LTS decomposed using Definition 4.5 maintains branching bisimulation with its decomposition, thus by proving that there is at least one universal method of decomposing LTSs into asynchronous ones while maintaining branching bisimulation.

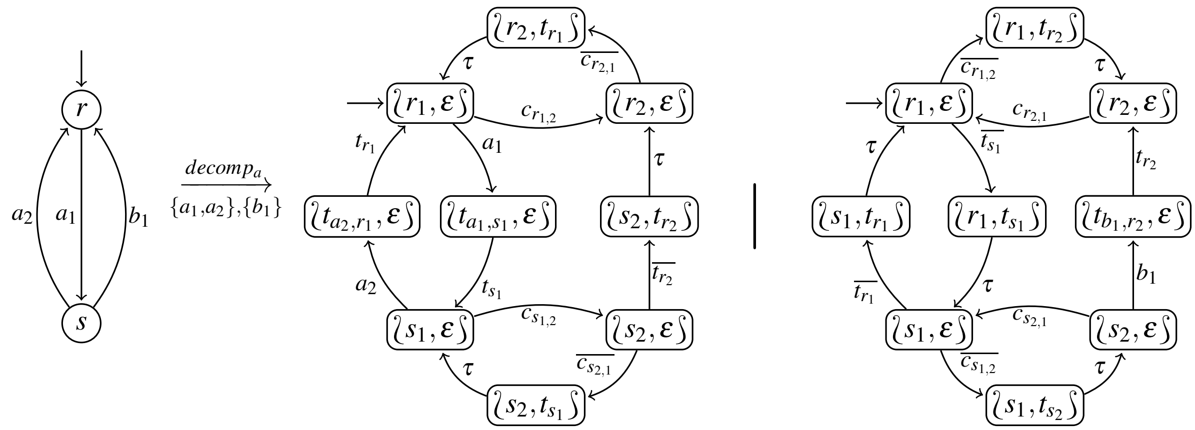

Figure 5: Example of asynchronous decomposition operation (Definition 4.5).

Theorem 4.6. Given an LTS $M = (S, \Sigma, \rightarrow, s_0)$ and two alphabets Σ_1, Σ_2 such that $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$, and given an LTS $M_x = M_1 \times M_2$ where $(M_1, M_2) = decomp_a(M)$ by Definition 4.5, then $M \leftrightarrow_b M_x$.

Proof. Let $M_1 = ((S_C \cup S_{T_1}, Q_1), \Sigma_1 \cup \Sigma_{S_1}, \rightarrow_1, r_1)$ and $M_2 = ((S_C \cup S_{T_2}, Q_2), \Sigma_2 \cup \Sigma_{S_2}, \rightarrow_2, r_1)$.
Define a relation $\mathscr{R} \subseteq S \times ((S_C \cup S_{T_1}, Q_1) \times (S_C \cup S_{T_2}, Q_2))$ with $\mathscr{R} =$

$$\begin{aligned}
&\{\langle s, (\lbag s_i, \varepsilon \rbag, \lbag s_i, \varepsilon \rbag)\rangle,\\
&\langle s, (\lbag s_i, t_{s_j} \rbag, \lbag s_j, \varepsilon \rbag)\rangle, \langle s, (\lbag s_j, \varepsilon \rbag, \lbag s_i, t_{s_j} \rbag)\rangle,\\
&\langle s, (t_{a,s_i}, \lbag r_i, \varepsilon \rbag)\rangle, \langle s, (\lbag r_i, \varepsilon \rbag, t_{a,s_i})\rangle,\\
&\langle s, (\lbag s_i, \varepsilon \rbag, \lbag r_i, t_{s_i} \rbag)\rangle, \langle s, (\lbag r_i, t_{s_i} \rbag, \lbag s_i, \varepsilon \rbag)\rangle,\\
&\langle u, (t_{b,s'_i}, \lbag r_i, t_{s_i} \rbag)\rangle, \langle u, (\lbag r_i, t_{s_i} \rbag, t_{b,s'_i})\rangle \mid\\
&r, s, u \in S \text{ and } i, j \in \{1,2\} \text{ where } i \neq j \text{ and } a, b \in \Sigma_i \text{ and } r \xrightarrow{a} s \xrightarrow{b} u\}.
\end{aligned}$$

We prove that $\mathscr{R}$ is a branching bisimulation relation through the following cases:

1. Consider a pair $\langle s, s_x \rangle = \langle s, (\lbag s_i, \varepsilon \rbag, \lbag s_i, \varepsilon \rbag)\rangle$ where $i \in \{1,2\}$.
   (a) Assume $s \xrightarrow{a} s'$. Then if $a \in \Sigma_1$, then $s_x \xrightarrow{a}_1 s'_x$ where $s'_x \overset{(14)}{=} (t_{a,s'_i}, \lbag s_i, \varepsilon \rbag)$ with $i = 1$. Else if $a \in \Sigma_2$ then $s_x \xrightarrow{a}_2 s''_x$ where $s''_x \overset{(14)}{=} (\lbag s_i, \varepsilon \rbag, t_{a,s'_i})$ with $i = 2$. We see that both pairs $\langle s, s'_x \rangle$ and $\langle s, s''_x \rangle$ are in $\mathscr{R}$.
   (b) Assume $s_x \xrightarrow{a}_x s'_x$. Then we have the following three cases:
      i. $a \in \Sigma_1 \wedge \overline{a} \notin \Sigma_2$, then this transition is only possible, by definition, through the transition $s \xrightarrow{a} s'$ for some s' such that $s'_x \overset{(14)}{=} (t_{a,s'}, s_1)$. We see that the pair $\langle s', s'_x \rangle \in \mathscr{R}$ and is covered in case 4.
      ii. $a \in \Sigma_2 \wedge \overline{a} \notin \Sigma_1$. This is a symmetric case where $s \xrightarrow{a} s'$ and $s'_x \overset{(14)}{=} (s_2, t_{a,s'})$. We see that the pair $\langle s', s'_x \rangle \in \mathscr{R}$ and is covered in case 5.
      iii. $a \in \Sigma_1 \wedge \overline{a} \in \Sigma_2$, then the only transition possible is the τ transition of (13). Then either $s'_x = (\lbag s_j, \varepsilon \rbag, \lbag s_i, t_{s_j} \rbag)$ or $s'_x = (\lbag s_i, t_{s_j} \rbag, \lbag s_j, \varepsilon \rbag)$ where $j \in \{1,2\}$ and $j \neq i$. We see that in both possible values of s'_x, the pair $\langle s, s'_x \rangle \in \mathscr{R}$ and is covered in cases 2 and 3.
2. Consider a pair $\langle s, s_x \rangle = \langle s, (\lbag s_i, t_{s_j} \rbag, \lbag s_j, \varepsilon \rbag)\rangle$.

(a) Assume $s \xrightarrow{a} s'$. Then $s_x \xrightarrow{\tau}_x (\wr s_j, \varepsilon \int, \wr s_j, \varepsilon \int)$, and we covered the pair $\langle s, (\wr s_j, \varepsilon \int, \wr s_j, \varepsilon \int)\rangle$ in case 1.

(b) Assume $s_x \xrightarrow{a}_x s'_x$. The the only possible transition in $\rightarrow_x$ is if a is a τ action consuming the queue message t_{s_j} then $s'_x = (\wr s_j, \varepsilon \int, \wr s_j, \varepsilon \int)$ and we covered the pair $\langle s, (\wr s_j, \varepsilon \int, \wr s_j, \varepsilon \int)\rangle$ in case 1.

3. Consider a pair $\langle s, s_x \rangle = \langle s, (\wr s_j, \varepsilon \int, \wr s_i, t_{s_j} \int)\rangle$. Symmetric to case 2.

4. Consider a pair $\langle s, s_x \rangle = \langle s, (t_{a,s_i}, \wr r_i, \varepsilon \int)\rangle$ such that $r \xrightarrow{a} s$.

   (a) Assume $s \xrightarrow{b} s'$. Then $s_x \xrightarrow{\tau}_x (\wr s_i, \varepsilon \int, \wr r_i, t_{s_i} \int)$.

   (b) Assume $s_x \xrightarrow{b}_x s'_x$. The only possible transition in $\rightarrow_x$ is if b is a τ action resulting from the synchronisation of the two transitions $t_{a,s_i} \xrightarrow{t_{s_i}}_i \wr s_i, \varepsilon \int$ and $\wr r_i, \varepsilon \int \xrightarrow{\overline{t_{s_i}}}_j \wr r_i, t_{s_i} \int$. Then, in the product, $s_x \xrightarrow{\tau}_x (\wr s_i, \varepsilon \int, \wr r_i, t_{s_i} \int)$; and the pair $\langle s, (\wr s_i, \varepsilon \int, \wr r_i, t_{s_i} \int)\rangle \in \mathscr{R}$ and is covered in case 6.

5. Consider a pair $\langle s, s_x \rangle = \langle s, (\wr r_i, \varepsilon \int, t_{a,s_i})\rangle$. Symmetric to case Case 4.

6. Consider a pair $\langle s, s_x \rangle = \langle s, (\wr s_i, \varepsilon \int, \wr r_i, t_{s_i} \int)\rangle$ such that $r \xrightarrow{a} s$.

   (a) Assume $s \xrightarrow{b} s'$. Then because of the queue-consuming transition $\wr r_i, t_{s_i} \int \xrightarrow{\tau}_j \wr s_i, \varepsilon \int$, then $s_x \xrightarrow{\tau}_x (\wr s_i, \varepsilon \int, \wr s_i, \varepsilon \int)$.
   The pair $\langle s, (\wr s_i, \varepsilon \int, \wr s_i, \varepsilon \int)\rangle$ is covered in case 1.

   (b) Assume $s_x \xrightarrow{b}_x s'_x$, then there are two possible values for b:

      i. Action b is a queue-consuming τ, then $s_x \xrightarrow{\tau}_x (\wr s_i, \varepsilon \int, \wr s_i, \varepsilon \int)$; and the pair $\langle s, (\wr s_i, \varepsilon \int, \wr s_i, \varepsilon \int)\rangle$ is covered in case 1.

      ii. $b \in \Sigma_i$, then $s_x \xrightarrow{b}_x (t_{b,s'_i}, \wr r_i, t_{s_i} \int)$ such that $s \xrightarrow{b} s'$; and the pair $\langle s', (t_{b,s'_i}, \wr r_i, t_{s_i} \int)\rangle \in \mathscr{R}$.

7. Consider a pair $\langle s, s_x \rangle = \langle s, \wr r_i, t_{s_i} \int, \wr s_i, \varepsilon \int)\rangle$. Symmetric to case 6.

8. Consider a pair $\langle s, s_x \rangle = \langle s, (t_{a,s_i}, \wr \wr p_i, \varepsilon \int, t_{r_i} \int)\rangle$ such that $p \xrightarrow{b} r \xrightarrow{a} s$. Then $s_x \xrightarrow{\tau} (t_{a,s_i}, \wr r_i, \varepsilon \int)$; and the pair $\langle s, (t_{a,s_i}, \wr r_i, \varepsilon \int)\rangle$ is covered in case 4.

9. Consider a pair $\langle s, s_x \rangle = \langle s, (\wr p_i, t_{r_i} \int, t_{a,s_i})\rangle$ such that $p \xrightarrow{b} r \xrightarrow{a} s$. This is symmetric to case 8.

$\square$

Corollary 4.7. It follows from Theorem 4.6 that there is a universal way of decomposing an LTS M using a general asynchronous decomposition operator (Definition 3.1) such that M is branching-bisimilar to the synchronous product of its decomposition.

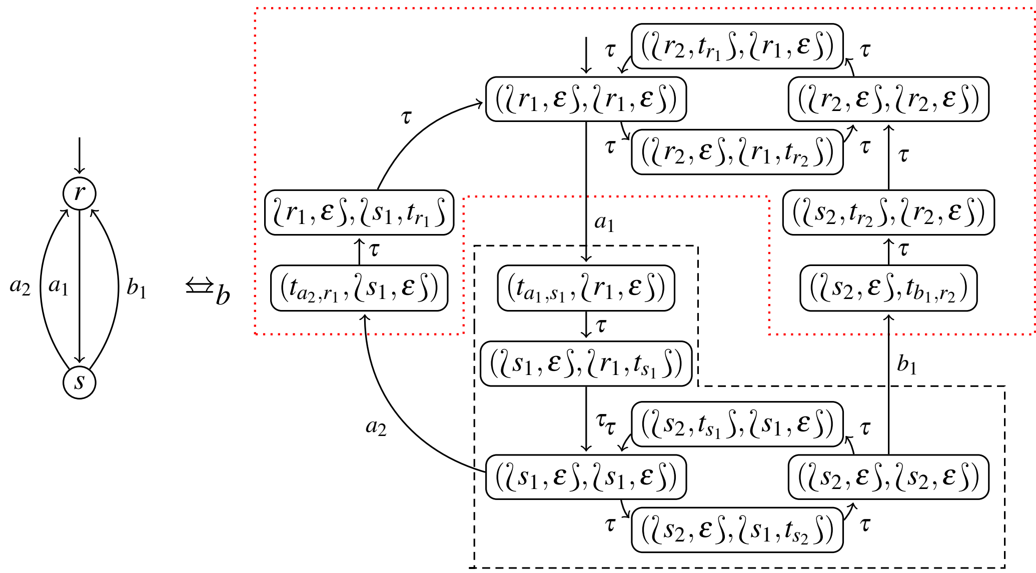

Figure 6: Showing branching bisimulation following Figure 5.

5 Proof that no decomposition operation maintains $\leftrightarroweq_{db}$

In this section, we prove that there is no way of decomposing an LTS such that it is divergence-preserving branching-bisimilar to the synchronous product of its decomposition.

We define confluence based on [6].

Definition 5.1 (Confluence)**.** An LTS $(S, \Sigma_1 \cup \Sigma_2, \rightarrow, s_0)$ is called confluent over Σ_1 and Σ_2 iff for all states $s, s_a, s_b \in S$ and for all $a \in \Sigma_1$ and $b \in \Sigma_2$, if $s \xrightarrow{a} s_a$ and $s \xrightarrow{b} s_b$, then there is a state s_c such that $s_b \xrightarrow{a} s_c$ and $s_a \xrightarrow{b} s_c$.

Lemma 5.2. Any LTS M that is the synchronous product (Definition 2.4) of two LTSs M_1 and M_2 whose action sets are Σ_1 and Σ_2 respectively, is confluent over two sets $\Sigma_1 \setminus \overline{\Sigma_2}$ and $\Sigma_2 \setminus \overline{\Sigma_1}$.

Proof. Consider the synchronous product $(S_1 \times S_2, \Sigma_x, \rightarrow_x, (q_0, r_0))$ from Definition 2.4 and some actions $a \in \Sigma_1 \setminus \overline{\Sigma_2}$ and $b \in \Sigma_2 \setminus \overline{\Sigma_1}$. Then $a \neq \overline{b}$. Consider some states $s, s' \in S_1$, $t, t' \in S_2$, and $s_a \in S_1 \times S_2$. We know that if $(s,t) \xrightarrow{a} s_a$ then that is due to a transition $s \xrightarrow{a} s'$ and that makes $s_a = (s',t)$, and that given a transition $t \xrightarrow{b} t'$, then a transition $(s',t) \xrightarrow{b} (s',t')$ is possible. Similarly, if $(s,t) \xrightarrow{b} (s,t')$ then $(s,t') \xrightarrow{a} (s',t')$. Therefore, the defined synchronous product is confluent. □

Figure 7 (centre) shows a simple LTS P. Concretely, it is defined as $(\{p,r,s\}, \Sigma_1 \cup \Sigma_2, \rightarrow, p)$ with alphabets $\Sigma_1 = \{a\}$ and $\Sigma_2 = \{b\}$ and transitions $p \xrightarrow{a} r$ and $p \xrightarrow{b} s$. In the following lemma and theorem, we prove that no way of decomposing P maintains divergence-preserving branching bisimulation.

Lemma 5.3. Given the LTS P (Figure 7, centre) with action set $\Sigma_1 \cup \Sigma_2$, let P_1 and P_2 be two LTSs with action sets Σ_{P_1} and Σ_{P_2} respectively, and with $\Sigma_1 \subseteq \Sigma_{P_1}$ and $\Sigma_2 \subseteq \Sigma_{P_2}$ and $\Sigma_1 \cap \Sigma_2 = \emptyset = \Sigma_1 \cap \Sigma_{P_2} = \Sigma_{P_1} \cap \Sigma_2$. Let P_x be the synchronous product $P_1 \times P_2$ by Definition 2.4. Then $P \not\leftrightarroweq_{db} P_x$.

Proof. We prove this lemma by contradiction. Assume that $P \leftrightarroweq_{db} P_x$, and let p_x be the initial state of P_x, then $p \leftrightarroweq_{db} p_x$. As p is not divergent and cannot do a τ-transition, it holds that only finite sequences of

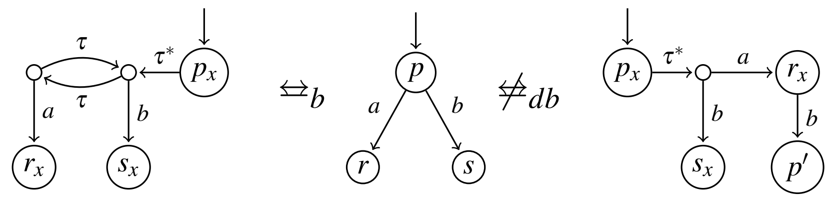

Figure 7: Illustration for Lemma 5.3.

τ's are possible from p_x. This can be seen as follows. If $p_x \xrightarrow{\tau} p_1 \xrightarrow{\tau} p_2 \xrightarrow{\tau} \cdots$, then $p \leftrightarrow_b p_i$ for all $i > 0$. Hence, p_x is divergent. But this is not possible because p is not divergent. So, p_x takes a finite number of τ steps to reach some state p'_x where $p'_x \not\xrightarrow{\tau}$.

Since it must be that $p \leftrightarrow_{db} p'_x$, and since $p \xrightarrow{a} r$ and $p \xrightarrow{b} s$ where $a \in \Sigma_1$ and $b \in \Sigma_2$, then there are two states r_x and s_x such that $p'_x \xrightarrow{a} r_x$ and $p'_x \xrightarrow{b} s_x$, and $r \leftrightarrow_{db} r_x$ and $s \leftrightarrow_{db} s_x$.

Now because $a \in \Sigma_1 \setminus \overline{\Sigma_2}$ and $b \in \Sigma_2 \setminus \overline{\Sigma_1}$, then P_x is confluent over these two sets, then there must exist a state p''_x such that $r_x \xrightarrow{b} p''_x$. However, $r \not\xrightarrow{b}$. Therefore, $r \not\leftrightarrow_{db} r_x$. Contradiction. Therefore $P \not\leftrightarrow_{db} P_x$. □

The proof is illustrated in Figure 7 showing that divergence-preserving branching bisimulation ($\leftrightarrow_{db}$) does not hold when decomposing the LTS P due to the confluence property of decompositions. On the other hand (literally the other hand of the same figure), branching bisimulation holds when decomposing LTS P. The reason it holds under $\leftrightarrow_b$, but not under $\leftrightarrow_{db}$ is that the former admits infinite τ cycles, i.e. divergence, which, as demonstrated here in right side of the figure, avoids the premise of confluence altogether.

Theorem 5.4. There is no decomposition operation that maintains divergence-preserving branching bisimulation ($\leftrightarrow_{db}$) for all LTSs.

Proof. We prove this theorem by contradiction. Assume that there is a decomposition operation that maintains $\leftrightarrow_{db}$ for all LTSs. Then it must do so for any arbitrary LTS P. But since Lemma 5.3 proves that no LTS maintains $\leftrightarrow_{db}$ for one such LTS P, i.e the one in Figure 7, then there is no decomposition operation that maintains $\leftrightarrow_{db}$ for all LTSs. □

6 Interpretation

One way to understand this fundamental result is that if the subsystems of the decomposition must communicate, then there is no escape from introducing divergence in order to maintain equivalence over any and all decompositions of LTSs. Moreover, if we look at systems while not observing divergences, it is impossible to recognise the internal structure by looking at the actions on the outside. We interpret this as a reason why model learning based on the internal structure is not possible.

Furthermore, divergence, in an industrial context, is undesired due to the requirement of fairness, i.e., one subsystem seizing unfair control over the total behaviour of the system through infinite looping. This means that if some decomposition is found to maintain fairness, then that is guaranteed not to be the case universally over all contexts and all LTSs.

References

[1] Dana Angluin (1987): *Learning regular sets from queries and counterexamples*. Information and Computation 75(2), pp. 87–106.

[2] Omar al Duhaiby, Arjan Mooij, Hans van Wezep & Jan Friso Groote (2018): *Pitfalls in Applying Model Learning to Industrial Legacy Software*. In Tiziana Margaria & Bernhard Steffen, editors: Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice, Lecture Notes in Computer Science 11247, Springer, Cham, pp. 121–138.

[3] Rob J. van Glabbeek, Bas Luttik & Nikola Trcka (2008): *Branching Bisimilarity with Explicit Divergence*. *CoRR* abs/0812.3068. Available at `http://arxiv.org/abs/0812.3068`.

[4] Rob J. van Glabbeek & W. Peter Weijland (1996): *Branching Time and Abstraction in Bisimulation Semantics*. *J. ACM* 43(3), pp. 555–600.

[5] Jan Friso Groote & Mohammad Reza Mousavi (2014): *Modeling and Analysis of Communicating Systems*. The MIT Press.

[6] Jan Friso Groote & MPA Sellink (1996): *Confluence for process verification*. Theoretical Computer Science 170(1-2), pp. 47–81.

[7] Kenneth Krohn & John Rhodes (1965): *Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines*. Transactions of the American Mathematical Society 116, pp. 450–464.

[8] Robin Milner & Faron Moller (1993): *Unique decomposition of processes*. Theoretical Computer Science 107(2), pp. 357–363.