Distribution of Behaviour into parallel communicating subsystems

Citation for published version (APA):

Document status and date:
Published: 30/05/2019

Document Version:
Accepted manuscript including changes made at the peer-review stage

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.

Download date: 15. Sep. 2019
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
Parallel Communicating Decompositions

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.
- $\Sigma$ 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 $\tau$ is the internal/unobservable action. Then:

\[
\begin{align*}
&\xrightarrow{a} s' \quad \text{iff } \langle s, a, s' \rangle \in \rightarrow. \\
&\xrightarrow{a} s \quad \text{iff there is an } s' \text{ such that } s \xrightarrow{a} s'. \\
&\xrightarrow{\tau} s \quad \text{iff there is no } s' \text{ such that } s \xrightarrow{a} s'. \\
&\xrightarrow{a} s_n S \quad \text{iff there are } s_1, s_2, \ldots, s_n \in S \text{ such that } s \xrightarrow{a} s_1 \xrightarrow{a} s_2 \cdots \xrightarrow{a} s_n. \\
&\xrightarrow{a} \infty S \quad \text{iff there are } s_1, s_2, \ldots \in S \text{ such that } s \xrightarrow{a} s_1 \text{ and for all } i \in \mathbb{N}, s_i \xrightarrow{a} s_{i+1}.
\end{align*}
\]

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, $(\overline{a}) = a$. 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 \(\Sigma_x = (\Sigma_1 \cup \Sigma_2) \setminus \{a, \overline{a} | a \in \Sigma_1 \wedge \overline{a} \in \Sigma_2\}\).

The transition relation \(\rightarrow_x \subseteq (S_1 \times S_2) \times \Sigma_x \times (S_1 \times S_2)\) is defined as follows:

\[
\begin{align*}
\langle s, t \rangle \xrightarrow{a} \langle s', t \rangle & \quad \text{iff } a \in \Sigma_1 \wedge \overline{a} \not\in \Sigma_2 \wedge s \xrightarrow{a} s', \\
\langle s, t \rangle \xrightarrow{a} \langle s, t' \rangle & \quad \text{iff } a \in \Sigma_2 \wedge \overline{a} \not\in \Sigma_1 \wedge t \xrightarrow{a} t', \text{ and} \\
\langle s, t \rangle \xrightarrow{} \langle s', t' \rangle & \quad \text{iff } a \in \Sigma_1 \wedge \overline{a} \in \Sigma_2 \wedge s \xrightarrow{a} s' \wedge t \xrightarrow{a} t',
\end{align*}
\]

where \(\tau\) is the unobservable action.

Next, we define two notions of behavioural equivalence.

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

1. if \(s \xrightarrow{a} s'\), then:
   - \(a = \tau\) and \(\langle s', t \rangle \in \mathcal{R}\); or
   - \(t \xrightarrow{a} t', \langle s', t' \rangle \in \mathcal{R}\) and \(\langle s', t'' \rangle \in \mathcal{R}\).

2. Symmetrically, if \(t \xrightarrow{a} t'\), then:
   - \(a = \tau\) and \(\langle s, t' \rangle \in \mathcal{R}\); or
   - \(s \xrightarrow{a} s', \langle s', t' \rangle \in \mathcal{R}\) and \(\langle s', t'' \rangle \in \mathcal{R}\).

Two states \(s\) and \(t\) are branching bisimilar, denoted \(s \equiv_b t\) iff there is a branching bisimulation relation \(\mathcal{R}\) such that \(\langle s, t \rangle \in \mathcal{R}\). Two LTSs \(P\) and \(Q\) are branching bisimilar, denoted \(P \equiv_b Q\), iff their initial states are.

A state \(s\) with \(s \xrightarrow{\tau^m} s\) is called divergent. Hence, a state with a \(\tau\) 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 \(\langle S, \Sigma, \rightarrow, s_0 \rangle\) and a relation \(\mathcal{R} \subseteq S \times S\). We call \(\mathcal{R}\) a divergence-preserving branching bisimulation relation iff it is a branching bisimulation relation and for all states \(s, t \in S\) with \(\langle s, t \rangle \in \mathcal{R}\), there is an infinite sequence \(s \xrightarrow{\tau} s_1 \xrightarrow{\tau} s_2 \xrightarrow{\tau} \cdots\) with \(\langle s_i, t \rangle \in \mathcal{R}\) for all \(i > 0\) iff there is an infinite sequence \(t \xrightarrow{\tau} t_1 \xrightarrow{\tau} t_2 \xrightarrow{\tau} \cdots\) and \(\langle s, t_i \rangle \in \mathcal{R}\) for all \(i > 0\).

Two states \(s\) and \(t\) are divergence-preserving branching bisimilar, denoted \(s \equiv_{db} t\) iff there is a divergence-preserving branching bisimulation relation \(\mathcal{R}\) such that \(\langle s, t \rangle \in \mathcal{R}\). Two LTSs \(P\) and \(Q\) are divergence-preserving branching bisimilar, denoted \(P \equiv_{db} Q\), 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 \(\Sigma\) and given two alphabets \(\Sigma_1, \Sigma_2\) 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 \(\Sigma_{M_1}\) with \(\Sigma_1 \subseteq \Sigma_{M_1}\) and \(\Sigma_{M_1} \cap \Sigma_2 = \emptyset\), and likewise, \(M_2\) has alphabet \(\Sigma_{M_2}\) with \(\Sigma_2 \subseteq \Sigma_{M_2}\) and \(\Sigma_{M_2} \cap \Sigma_1 = \emptyset\).
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 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 \( \text{decomp}_s \) decomposes into synchronously communicating subsystems while the second \( \text{decomp}_a \) 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 \( \Sigma_1, \Sigma_2 \) such that \( \Sigma = \Sigma_1 \cup \Sigma_2 \) and \( \Sigma_1 \cap \Sigma_2 = \emptyset \), then we can decompose \( M \) over \( \Sigma_1 \) and \( \Sigma_2 \) by applying the following operation:

\[
\text{decomp}_s(M, \Sigma_1, \Sigma_2) = (M_1, M_2)
\]

where:

1. \( M_1 = (S_C \cup S_{T_1}, \Sigma_1 \cup \Sigma_{S_1}, \rightarrow_1, (q, 1)) \)
2. \( M_2 = (S_C \cup S_{T_2}, \Sigma_2 \cup \Sigma_{S_2}, \rightarrow_2, (q, 1)) \).

2. For every state \( s \) in \( S \), we introduce two states \( (s, 1), (s, 2) \in S_C \):

\[
S_{C_1} = \{(s, 1) \mid s \in S\} \quad S_{C_2} = \{(s, 2) \mid s \in S\} \quad S_C = S_{C_1} \cup S_{C_2}
\]
Notation. Tuple-states of the form \((s, t)\) 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:

\[
\Sigma_C = \{c_{s_1, s_2}, c_{s_2, s_1} \mid s_1 \in S_{C_1}, s_2 \in S_{C_2}\}
\]  

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

\[
\Sigma_{T_1} = \{t_{s_1} \mid s_1 \in S_C\} \quad \Sigma_{T_2} = \{t_{s_2} \mid s_2 \in S_C\}
\]

\[
S_{T_1} = \{t_{a, s_1} \mid a \in \Sigma_1, s_1 \in S_{C_1}\} \quad S_{T_2} = \{t_{a, s_2} \mid a \in \Sigma_2, s_2 \in S_{C_2}\}
\]  

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

\[
\Sigma_{S_1} = \Sigma_{T_1} \cup \Sigma_C \cup \Sigma_{T_2}\]

\[
\Sigma_{S_2} = \Sigma_{T_2} \cup \Sigma_C \cup \Sigma_{T_1}\]

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 \(c_{s_i, s_j}, c_{s_j, s_i} \in \Sigma_C\):

\[
s_i \xrightarrow{c_{s_i, s_j}} t_i s_j \quad s_i \xrightarrow{c_{s_j, s_i}} t_j s_j
\]  

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

\[
s_i \xrightarrow{a} t_{a, s'} \xrightarrow{t_i} s_i'
\]  

\[
\xrightarrow{t_j}
\]

\[
s_i \xrightarrow{t_j} s_j'
\]

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 \(c_{r_2, r_1}\), to produce a synchronous transition in both machines from \(r_1\) to \(r_2\). Likewise, actions \(c_{r_2, r_1}\) and \(c_{r_1, r_2}\) 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 \(t_{a_1, s_1} \xrightarrow{t_1} s_1\) which synchronises with the transition \(r_1 \xrightarrow{t_1} s_1\) in \(M_2\).

The operation \((decomp)\) 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 \[\text{Definition 4.1}\] 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:

\[
S_x = S_1 \times S_2 = (S_C \cup S_{T_1}) \times (S_C \cup S_{T_2})
\]

\[
= (S_C \times S_C) \cup (S_{T_1} \times S_{T_1})
\]

\[
\cup (S_{T_1} \times S_C) \cup (S_C \times S_{T_2})
\]

(7)

with \(\Sigma_{S_1}, \Sigma_{S_2}, \Sigma_{T_1}, \Sigma_{T_2}\) being sets introduced by \(\text{decomp}_x\). 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 \(t_{a,s'} \in S_{T_i}\) and a pair of complementary actions \(\tau_j, \tau_j' \in \Sigma_{T_j}\) such that:

\[
(s_i, s_j) \xrightarrow{a} s_a \xrightarrow{\tau_x} (s'_i, s'_j),
\]

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

(8)

2. For all \(s \in S\), there exist \(c_i, s_j, c_j, s_i \in \Sigma_C\) such that, by \(5\), \(s_i \xrightarrow{c_i} s_j, \tau_j\) and \(s_i \xrightarrow{c_i} \tau_j s_j, \tau_j\), and thus:

\[
(s_i, s_j) \xrightarrow{\tau_x} (s_j, s_j)
\]

(9)

4.2 Proof that the synchronous decomposition operation maintains branching bisimulation

In this subsection, we show an application of \(\text{decomp}_x\) \[\text{Definition 4.1}\] to a sample LTS, we demonstrate that \(\text{decomp}_x\) maintains branching bisimilarity, and then we prove that branching bisimilarity is maintained through any and all applications of \(\text{decomp}_x\).

[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 \(\Sigma_1, \Sigma_2\) 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 \((M_1, M_2) = \text{decomp}_x(M)\) by \[\text{Definition 4.1}\], then \(M \equiv_b M_x\).

**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 \(R \subseteq S \times ((S_C \cup S_{T_1}) \times (S_C \cup S_{T_2}))\) with \(R = \{(s, (s_n, n)), (r', (t_n, t_n'))\} \mid s, r', r \in S, n \in \{1, 2\}, a \in \Sigma, r \xrightarrow{a} r'\}.\) We prove that \(R\) is a branching bisimulation relation through the following cases:
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 decompa operator (Definition 4.5).

4.3 Decomposing into Asynchronous Subsystems

We define a new decomposition operation (decompa) 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 decompa 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 \(s, Q\).

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 \(\varepsilon\) represents the empty queue.

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

\[
\text{decompa}(M, \Sigma_1, \Sigma_2) = (M_1, M_2)
\]

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. \(M_i = (S_C \cup S_{Ti}, Q_i, \Sigma_i \cup \Sigma_{Si}, \rightarrow_i, r_i)\)
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:

\[
S_C = \{ s \mid s \in S \} \quad \Sigma_C = \{ c_{s,i,j} \mid s \in S \} \quad \Sigma_T = \{ t_s \mid s \in S \}
\]

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

\[
S_T = \{ t_{a,s} \mid a \in \Sigma_i, s \in S_C \}
\]

4. Sets of synchronous actions are defined as follows:

\[
\Sigma_S = \Sigma_T \cup \Sigma_T \cup \Sigma_C \cup \Sigma_C
\]

5. The transition relation \( \rightarrow_i \subseteq (S_C \cup S_T) \times Q_i \times (\Sigma_i \cup \Sigma_S) \times (S_C \cup S_T) \times Q_i \) is the minimal relation satisfying the following:

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

\[
\hat{s}_i, Q_i \xrightarrow{c_{s,i,j}} \hat{s}_j, Q_i
\]

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

\[
\hat{s}_i, Q_i \xrightarrow{a} \hat{t}_{a,s'}, Q_i \xrightarrow{t'_{s,i,j}} \hat{s}_j, Q_i
\]

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

\[
\hat{s}, Q \cdot t_s \xrightarrow{\tau} \hat{s}', Q
\]

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 \((\text{decomp}_a)\) also maintains branching bisimilarity. Figure 5 shows the result of applying \(\text{decomp}_a\) 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.
Theorem 4.6. Given an LTS $M = (S, \Sigma, \rightarrow, s_0)$ and two alphabets $\Sigma_1, \Sigma_2$ such that $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$, and given an LTS $M_2 = M_1 \times M_2$ where $(M_1, M_2) = \text{decomp}_a(M)$ by [Definition 4.5], then $M \cong_b M_x$.

Proof. Let $M_1 = (((S_C \cup S_{T_1}, Q_1), \Sigma_1) \cup \Sigma_{S_1}, \rightarrow, r_1)$ and $M_2 = (((S_C \cup S_{T_2}, Q_2), \Sigma_2) \cup \Sigma_{S_2}, \rightarrow, r_1)$.

Define a relation $\mathcal{R} \subseteq S \times ((S_C \cup S_{T_1}, Q_1) \times (S_C \cup S_{T_2}, Q_2))$ with $\mathcal{R} =$

$$\{ (s, (s, \epsilon, \overline{s}), (s, \epsilon, \overline{s})), (s, (s, \epsilon, \overline{s}), (s, \epsilon, \overline{s})), (s, (s, \epsilon, \overline{s}), (s, \epsilon, \overline{s})), (s, (s, \epsilon, \overline{s}), (s, \epsilon, \overline{s})), (s, (s, \epsilon, \overline{s}), (s, \epsilon, \overline{s})), (s, (s, \epsilon, \overline{s}), (s, \epsilon, \overline{s})), (s, (s, \epsilon, \overline{s}), (s, \epsilon, \overline{s})), (s, (s, \epsilon, \overline{s}), (s, \epsilon, \overline{s})), (s, (s, \epsilon, \overline{s}), (s, \epsilon, \overline{s})) \mid r, s, u \in S \text{ and } i, j \in \{1, 2\} \text{ where } i \neq j \text{ and } a, b \in \Sigma_i \text{ and } r \overset{a}{\rightarrow} s \overset{b}{\rightarrow} u \}.$$  

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

1. Consider a pair $\langle s, s_x \rangle = \langle s, (s, \epsilon, \overline{s}), (s, \epsilon, \overline{s}) \rangle$ where $i \in \{1, 2\}$.
   
   (a) Assume $s \overset{a}{\rightarrow} s'$. Then if $a \in \Sigma_1$, then $s_x \overset{a}{\rightarrow} s'_x$ where $s'_x = (t_{a,s'},(s,\epsilon,\overline{s}))$ with $i = 1$. Else if $a \in \Sigma_2$ then $s_x \overset{a}{\rightarrow} s''_x$ where $s''_x = (t_{a,s''},(s,\epsilon,\overline{s}))$ with $i = 2$. We see that both pairs $\langle s, s'_x \rangle$ and $\langle s, s''_x \rangle$ are in $\mathcal{R}$.

   (b) Assume $s_x \overset{a}{\rightarrow} s'_x$. Then we have the following three cases:
   
   i. $a \in \Sigma_1 \land \overline{a} \notin \Sigma_2$, then this transition is only possible, by definition, through the transition $s' \overset{a}{\rightarrow} s'$ for some $s'$ such that $s'_x = (t_{a,s'},(s,\epsilon,\overline{s}))$. We see that the pair $\langle s', s'_x \rangle \in \mathcal{R}$ and is covered in case 4.

   ii. $a \in \Sigma_2 \land \overline{a} \notin \Sigma_1$. This is a symmetric case where $s \overset{a}{\rightarrow} s'$ and $s'_x = (s_2, t_{a,s'})$. We see that the pair $\langle s', s'_x \rangle \in \mathcal{R}$ and is covered in case 5.

   iii. $a \in \Sigma_1 \land \overline{a} \in \Sigma_2$, then the only transition possible is the $\tau$ transition of (13). Then either $s'_x = (t_{a,s'},(s,\epsilon,\overline{s}))$ or $s'_x = (s_1, t_{a,s'})$ where $j \in \{1, 2\}$ and $j \neq i$. We see that in both possible values of $s'_x$, the pair $\langle s, s'_x \rangle \in \mathcal{R}$ and is covered in cases 2 and 3.

2. Consider a pair $\langle s, s_x \rangle = \langle s, (s, \epsilon, \overline{s}), (s, \epsilon, \overline{s}) \rangle$. 

Figure 5: Example of asynchronous decomposition operation (Definition 4.5).
(a) Assume \( s \xrightarrow{a} s' \). Then \( s_x \xrightarrow{\tau} (\tau, s', \epsilon, \tau, s') \), and we covered the pair \( \langle s, (\tau, s', \epsilon, \tau, s') \rangle \) in case 1.

(b) Assume \( s_x \xrightarrow{a} s'_x \). The only possible transition in \( \rightarrow_x \) is if \( a \) is a \( \tau \) action consuming the queue message \( t_s \), then \( s'_x = (\tau, s', \epsilon, \tau, s') \) and we covered the pair \( \langle s, (\tau, s', \epsilon, \tau, s') \rangle \) in case 1.

3. Consider a pair \( \langle s, s_x \rangle = \langle s, (s', s, t, s_t) \rangle \). Symmetric to case 2.

4. Consider a pair \( \langle s, s_x \rangle = \langle s, (t_a, s, t_r, s_r) \rangle \) such that \( a \rightarrow s \).

(a) Assume \( s \xrightarrow{b} s' \). Then \( s_x \xrightarrow{\tau} (s, s', t, s_t) \).

(b) Assume \( s_x \xrightarrow{b} s'_x \). The only possible transition in \( \rightarrow_x \) is if \( b \) is a \( \tau \) action resulting from the synchronisation of the two transitions \( t_a \xrightarrow{i} (s, t, s_t) \) and \( t_r \xrightarrow{j} (r, t, s_t) \). Then, in the product, \( s_x \xrightarrow{\tau} (s, s', t, s_t) \) and \( s \xrightarrow{b} s' \). Then because of the queue-consuming transition, \( s \rightarrow s \), the only possible transition in \( b \rightarrow s \), then there are two possible values for \( b \):

i. Action \( b \) is a queue-consuming \( \tau \), then \( s \xrightarrow{\tau} (s, s', t, s_t) \) and the pair \( \langle s, (s, s', t, s_t) \rangle \) is covered in case 1.

ii. \( b \in \Sigma_t \), then \( s_x \xrightarrow{b} (t, s', t, s_t) \) such that \( b \rightarrow s' \); and the pair \( \langle s', (t, s', t, s_t) \rangle \) is covered in case 1.

5. Consider a pair \( \langle s, s_x \rangle = \langle s, (s', s, t, s_t) \rangle \). Symmetric to case 4.

6. Consider a pair \( \langle s, s_x \rangle = \langle s, (s, s', t, s_t) \rangle \) such that \( a \rightarrow s \).

(a) Assume \( s \xrightarrow{b} s' \). Then because of the queue-consuming transition \( (r, t, s_t) \xrightarrow{\tau} (s, s', t, s_t) \), then \( s_x \xrightarrow{\tau} (s, s', t, s_t) \).

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

i. Action \( b \) is a queue-consuming \( \tau \), then \( s \xrightarrow{\tau} (s, s', t, s_t) \) and the pair \( \langle s, (s, s', t, s_t) \rangle \) is covered in case 1.

ii. \( b \in \Sigma_t \), then \( s_x \xrightarrow{b} (t, s', s, t, s_t) \) such that \( b \rightarrow s' \); and the pair \( \langle s', (t, s', s, t, s_t) \rangle \) is covered in case 1.

7. Consider a pair \( \langle s, s_x \rangle = \langle s, (r, t, s_t) \rangle \). Symmetric to case 6.

8. Consider a pair \( \langle s, s_x \rangle = \langle s, (t_a, s, t_r, s_r) \rangle \) such that \( a \rightarrow r \rightarrow s \). Then \( s_x \xrightarrow{\tau} (t_a, s, t_r, s_r) \); and the pair \( \langle s, (t_a, s, t_r, s_r) \rangle \) is covered in case 4.

9. Consider a pair \( \langle s, s_x \rangle = \langle s, (t_r, t, s_t) \rangle \) such that \( b \rightarrow a \rightarrow s \). 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.
We prove this lemma by contradiction. Assume that \( P \not\equiv_{db} P_x \). Then \( P \) is not divergent and cannot do a \( \tau \)-transition, it holds that only finite sequences of...
\[\tau's \text{ are possible from } p_x. \text{ This can be seen as follows. If } p_x \xrightarrow{\tau} p_1 \xrightarrow{\tau} p_2 \xrightarrow{\tau} \ldots, \text{ then } p \equiv_b p_i \text{ for all } i > 0.\]

Hence, \(p_x\) is divergent. But this is not possible because \(p\) is not divergent. So, \(p_x\) takes a finite number of \(\tau\) steps to reach some state \(p'_x\) where \(p'_x \not\equiv \tau p_x\).

Since it must be that \(p \equiv_{db} p'_x\), and since \(p \xrightarrow{a} r \text{ and } p \xrightarrow{b} s\) 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 \(r \equiv_{db} r_x\) and \(s \equiv_{db} s_x\).

Now because \(a \in \Sigma_1 \setminus \Sigma_2\) and \(b \in \Sigma_2 \setminus \Sigma_1\), 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, \(r \not\equiv db r_x\). Therefore, \(r \not\equiv db r_x\). Contradiction. Therefore \(P \not\equiv db P_x\). □

The proof is illustrated in Figure 7 showing that divergence-preserving branching bisimulation (\(\equiv_{db}\)) 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 \(\equiv_b\), but not under \(\equiv_{db}\) is that the former admits infinite \(\tau\) 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 (\(\equiv_{db}\)) for all LTSs.

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


