Active Learning of Decomposable Systems

Citation for published version (APA):

Document license:
Unspecified

DOI:
10.1145/3372020.3391560

Document status and date:
Published: 07/10/2020

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

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: 25. Jun. 2021
Active Learning of Decomposable Systems

Omar al Duhaiby
o.z.alzuhaibi@tue.nl
Eindhoven University of Technology
Eindhoven, The Netherlands

Jan Friso Groote
j.f.groote@tue.nl
Eindhoven University of Technology
Eindhoven, The Netherlands

ABSTRACT

Active automata learning is a technique for querying black box systems and modelling their behaviour. In this paper, we aim to apply active learning in parts. We formalise the conditions on systems—with a decomposable set of actions—that make learning in parts possible. The systems are themselves decomposable through non-intersecting subsets of actions. Learning these subsystems/components requires less time and resources. We prove that the technique works for both two components as well as an arbitrary number of components. We illustrate the usefulness of this technique through a classical example and through a real example from the industry.

CCS CONCEPTS

• Computing methodologies → Model development and analysis; • Theory of computation → Formal languages and automata theory; • Active learning; • Software and its engineering → Model-driven software engineering.

KEYWORDS

decomposition, active learning

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FormalISE '20, October 7–8, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7071-4/20/05...$15.00
https://doi.org/10.1145/3372020.3391560

1 INTRODUCTION

Active automata learning [17] is a technique meant for discovering the behaviour of a black box system through querying it with different words composed out of its alphabet. Angluin [4] specified the first active learning algorithm. Ever since, adaptions [9, 12], implementations [5, 11] and successful applications [1, 15, 16] have been following.

The Problem. A fundamental problem with active learning is that it is very difficult to learn large models. Typically, learning models of tens of thousands of states or more is exceptionally resource intensive. That is because the number of queries required to learn a model grows quadratically with the number of states [10]. Moreover, active learning can only be applied if there are roughly 100 inputs or less [17]. That is why we are interested in decomposing the system under learning over inputs, as well as outputs, and learning it in parts.

Our Solution. We learn the behaviour of large systems in parts. We divide the external actions of the system into (1) a set of actions globally-shared between components, and (2) multiple disjoint subsets each pertaining to one component. We then proceed as if learning the behaviour of the entire system while restricted to the actions of one of the subsets in addition to global actions which are never restricted. This is effectively learning only the corresponding component. We learn different parts the same way and combine the learned parts using synchronous products into the overall learned behaviour.

We formulate conditions on the behaviour of the system to guarantee that the overall learned behaviour is exactly the same as the behaviour of the original system. We prove that the learnt parts constitute the whole behaviour and we argue against possible weakenings of the conditions with counterexamples.

Related Work. A recent related work was done by Moerman [13] where he employed parallel learners each observing a subset of the output alphabet. Such a technique is possible when the system under learning is decomposable over output. The formalism used in [13] was a Moore machine whereas this paper uses a Labelled Transition System (LTS) where inputs and outputs are not discerned but are simply labels/actions.

Moerman [13] brilliantly adapts an L*-based algorithm to learn multiple Moore machines in parallel and he proves correctness and termination for his algorithm. Our work, on the other hand, arrives at a general theorem for the requirements of learning decomposable systems in parts.

Moreover, as has Moerman, we use the Rivest and Shappire n-bit register machine example [14] to demonstrate the usefulness of partial learning and its applicability. We go further to prove that the n-bit register machine adheres to our theorem.

Motivation. This work was motivated by a real case example in the industry [3] based on which we pursued the possibility of applying the active learning technique in parts. In [2], we studied the decomposability of systems based on input partitioning and have proven the validity of certain decompositions. This work continues the study on learning in parts by setting and proving constraints on systems under learning.

Contribution. We formalised the conditions on LTSs that enable learning in parts and we prove that the whole behaviour can be learned. We verified our theoretical result on an example system from the industry.
Outline. The outline of this paper is as follows. Section 2 introduces the preliminaries. Section 3 provides the conditions to learn LTs in parts and proves that they work. Section 4 generalises the theorem for learning n components in parts. Section 5 demonstrates the technique through a classical illustrative example. Section 6 argues, through counterexamples, against possible weakenings of the conditions. Section 7 details a more challenging example from the industry. Finally, Section 8 proposes future work and Section 9 concludes the paper.

2 PRELIMINARIES

In this section, we present the preliminaries of labelled transition systems, the synchronous product, and bisimulation relations, aided by [7]. In addition, we define a few concepts central to our conditions for learning in parts. We start with the definition of a labelled transition system (LTS).

Definition 2.1 (LTS). We define our Labelled Transition System as a four-tuple \((Q, A, \rightarrow, q^0)\) where:

- \(Q\) is a non-empty finite set of states.
- \(A\) is the alphabet, also referred to as the action set.
- \(\rightarrow \subseteq Q \times A \times Q\) is a transition relation.
- \(q^0\) is the initial state.

The language of an alphabet \(A\), the set of all words over \(A\), is denoted by \(A^*.\)

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 \((Q, A, \rightarrow, q^0)\) be an LTS with \(s, s' \in Q\) and \(a \in L\). Then:

- \(s \xrightarrow{a} s'\) if \((s, a, s') \in \rightarrow\).
- \(s \xrightarrow{a}\) if there is an \(s'\) such that \(s \xrightarrow{a} s'\).
- \(s \xrightarrow{a}\) if there is no \(s'\) such that \(s \xrightarrow{a} s'\).

Learning Parts and Composing them, and Global and Local Actions. Learning a system in parts produces components which we need a synchronous product in order to synchronise on the shared actions when computing the total behaviour.

Definition 2.3 (Synchronous Product of two LTs). Let \(G, A_1\) and \(A_2\) be sets of actions with \(G = A_1 \cap A_2\). The synchronous product of two LTs \((Q_1, A_1, \rightarrow_1, r^0) \parallel_G (Q_2, A_2, \rightarrow_2, s^0)\) is the tuple \((Q_1 \times Q_2, A_1 \cup A_2, \rightarrow, (r^0, s^0))\). The transition relation \(\rightarrow \subseteq (Q_1 \times Q_2) \times (A_1 \cup A_2) \times (Q_1 \times Q_2)\) is defined as follows:

\[
\begin{align*}
(s, t) & \xrightarrow{a} (s', t) \quad \text{if } a \in A_1 \land a \notin A_2 \land s \xrightarrow{a} t' \\
(s, t) & \xrightarrow{a} (s', t') \quad \text{if } a \in A_2 \land a \notin A_1 \land t \xrightarrow{a} t' , \text{ and} \\
(s, t) & \xrightarrow{a} (s', t') \quad \text{if } a \in G \land s \xrightarrow{a} s' \land t \xrightarrow{a} t' .
\end{align*}
\]

Next, we define the blocking operator. When applied to an automaton given an action set, it simply removes all transitions that are labelled with an action from the set.

Definition 2.4. (Blocking Operator) Let \(M\) be an LTS \((Q, A, \rightarrow, q^0)\) and let action set \(L \subseteq A\). The blocking operator \(\delta_L\) is defined as

\[
\delta_L(M) = (Q \setminus L, \rightarrow', q^0),\quad \text{where the transition relation } \rightarrow' = \rightarrow \setminus \{(q, a, q') \in Q \times A \times Q \mid a \in L\} = Q \times L \times Q.
\]

Opposite of the blocking operator is the allow operator denoted by \(\nabla_L\). It allows only transitions labelled with actions in the given set \(L\) and blocks all others. The allow operator is only used in the mcRL2 specification in Appendix A.

Definition 2.5. (Allow Operator) Let \(M\) be an LTS \((Q, A, \rightarrow, q^0)\) and let action set \(L \subseteq A\). The allow operator \(\nabla_L\) is defined as

\[
\nabla_L(M) = (Q, L, \rightarrow', q^0),\quad \text{where the transition relation } \rightarrow' = \{ (q, a, q') \in Q \times A \times Q \mid a \in L \} = Q \times L \times Q.
\]

We define semi-confluence, a weaker flavour of strong confluence [8].

Definition 2.6 (Semi-confluence). Let \(M\) be an LTS \((Q, A, \rightarrow, q^0)\) and let action sets \(L_a, L_b \subseteq A\). We say that \(M\) is semi-confluent from \(L_a\) towards \(L_b\), a property written as \(L_a \rightarrow_{L_b}\), if for all states \(s, s_a, s_b \in Q\) and all actions \(a \in L_a\) and \(b \in L_b\), the following holds:

- If \(s \xrightarrow{a} s_a\) and \(s \xrightarrow{b} s_b\), then there is a state \(r \in Q\) and an action sequence \(x \in L_e\) such that \(s_a \xrightarrow{r} r\) and \(s_b \xrightarrow{r} r\).

We define a constraint on LTSs called \(P_{ac}\).

Definition 2.7 (Pre-access). Let \(M\) be some LTS \((Q, A, \rightarrow, q^0)\) and let \(L \subseteq A\). We say that \(M\) maintains pre-access over \(L\) iff for all \(q \in Q\), \(e \in L\), and \(a \in A \setminus L\), the following holds:

- If \(q \xrightarrow{a} r\) then \(q \xrightarrow{a}\).

Pre-access over a set \(L\) is denoted, primarily in the figures of this paper, by \(P_{L}\). The intuition behind the name is as follows. Following the definition, if state \(q\) did not enable action \(a\) and had no access to a state enabling action \(a\) except through \(L\)-labelled transitions, then it is given pre-access over \(L\) to action \(a\) by directly enabling it at \(q\).

The following lemma extends pre-access over an action set to sequences of actions from that set.

Lemma 2.8. Let \(M\) be some LTS \((Q, A, \rightarrow, q^0)\) that maintains pre-access over \(L\) where \(L \subseteq A\). Then for \(r \in Q\), \(a \in A \setminus L\), and \(\rho \in L^*\), we find:

- If \(r \xrightarrow{\rho} a\) then \(r \xrightarrow{a}\).

Lemma 2.8 is proven by induction on the length of the sequence of actions.

We define the classical notion of Determinism on LTs.

Definition 2.9 (Determinism). An LTS \((Q, A, \rightarrow, q^0)\) is deterministic if for all \(q \in Q\) and \(a \in A\), if \(q \xrightarrow{a} r\) and \(q \xrightarrow{a} s\) then \(r = s\).

Our assumption on learning. The learning process is assumed to be complete, i.e., it discovers the entire behaviour of the target LTS restricted only by the alphabet it uses. We use the blocking operator \(\delta\) from Definition 2.4 to represent such restriction. For example, Figure 6 shows an LTS \(M\) with alphabet \(\{a, b, c\}\). Learning \(M\) in two parts, one with the alphabet \(\{a, c\}\) and the other with alphabet \(\{b, c\}\) results in LTs \(M_1 = \delta_{\{b\}}(M)\) and \(M_2 = \delta_{\{a\}}(M)\) respectively.
A basic explanation of how the constraints work. Semi-confluence, Pre-access and Determinism (Definitions 2.6, 2.7 and 2.9) are the three conditions that enable an LTS to be learnt in parts. To understand this, we explain the setup of the problem. What is given is one set of global actions and multiple sets of local actions each for one component. Each component is learnt through an action set comprising its local actions in addition to the global actions; all other local action sets are blocked (Definition 2.4). Therefore, if one state in the LTS is reachable only through a trace comprised of actions from different local sets, then that state cannot be learnt by any part of the learning process.

The intuition behind these three constraints is to ensure the reachability of every state despite the blockage of sets of actions throughout different parts of the learning. Blocking a set of actions results in partial behaviour, and only actions local to one part of the machine can be blocked at a time. Therefore, if two actions from two different local sets form a sequence of transitions in the original LTS such as \( q \xrightarrow{a} L_1 \xrightarrow{b} r \), then we want to ensure that state \( r \) is reachable by the total behaviour—that is the combination of all partial behaviours. For this to happen, (1) state \( q \) itself must be reachable by the total behaviour, and (2) there must be a trace from \( q \) to \( r \) such that the synchronous product of the two components contains a trace from \( q \) to \( r \). The way the three constraints work in such a case is that, first, pre-access over \( L_1 \) ensures a path to some middle state like \( q \xrightarrow{b \in L_2} x \). Then, semi-confluence ensures a path \( x \xrightarrow{a \in L_1} t \). Third, determinism ensures that state \( t \) is equivalent to \( r \), the desired state to reach.

We define strong bisimilarity, the equivalence relation with which we compare the system under learning to the composition of the learnt parts.

**Definition 2.10** (Strong Bisimilarity). Given an LTS \((Q, A, \rightarrow, q^0)\) and a relation \( R \subseteq Q \times Q \). We call \( R \) a strong bisimulation relation if for all states \( s, t \in Q \) such that \((s, t) \in R\), it holds that:

1. if \( s \xrightarrow{a} s' \), then there is a state \( t' \in Q \) such that \( t \xrightarrow{a} t' \) and \((s', t') \in R\).
2. Symmetrically, if \( t \xrightarrow{a} t' \), then there is a state \( s' \in Q \) such that \( s \xrightarrow{a} s' \) and \((s', t') \in R\).

Two states \( s \) and \( t \) are strongly bisimilar, denoted \( s \approx t \) iff there is a strong bisimulation relation \( R \) such that \((s, t) \in R\). Two LTSs \( P \) and \( Q \) are strongly bisimilar, denoted \( P \approx Q \), iff their initial states are.

### 3 CONDITIONS OF LEARNING TWO COMPONENTS IN PARTS

In this section, we introduce Theorem 3.1 stating the conditions that enable learning in parts and we prove that two separately learnt parts constitute the original behaviour. Even though, Section 4 generalises this proof to \( n \) components, the case for two components proves effective to introduce the concept as well as the technical details of the theorem and the proof more clearly.

Our theorem states that given a system, represented by an LTS, and a certain partitioning of its actions into two sets of local actions and one set of global actions, it can be learnt in two parts if it maintains three properties: Pre-access over each local-action set, semi-confluence, and determinism.

We explained briefly in Section 2 how the three conditions work together to ensure the reachability in the product of every state from the original LTS as well as maintain equivalence. We showed a simple example of a trace \( q \xrightarrow{a \in L_1, b \in L_2} r \) and we showed how state \( r \) is reachable given that state \( q \) is. This is one of three types of sequences possible. Those three are:

1. The sequence given by the aforementioned example trace \( q \xrightarrow{a \in L_1, b \in L_2} r \).
2. A sequence consisting of global actions only. This is reachable in both components because neither blocks global actions.
3. A sequence consisting of local actions from a single component. This is reachable in that particular component and thus reachable in the product.

We claim that, thanks to our conditions, since state \( r \) is reachable, an equivalent of state \( r \) in the product is reachable through sequences from both \( L_1 \) and \( L_2 \) that could also be empty. Using that idea, we defined relation \( R \) in the proof to reflect precisely how a state in the product is generated from states and transitions in the original LTS. We prove that \( R \) is a bisimulation relation and thus prove the equivalence of the original LTS with the product of the learnt parts. Figures 1 and 2 start with the relation and aid in following the steps of the proof.

**Theorem 3.1.** Let \( M \) be some LTS \((Q, A, \rightarrow, q^0)\) and \( L_1, L_2 \) and \( G \) be disjoint subsets of \( A \) such that \( L_1 \cup L_2 \cup G = A \). Let \( M_1 = \delta_{L_1}(M) \) and \( M_2 = \delta_{L_2}(M) \). Then \( M \equiv M_1 \parallel G \parallel M_2 \) if the following three constraints are satisfied:

1. The LTS \( M \) maintains pre-access over \( L_1 \) and over \( L_2 \) (Definition 2.7).
2. The LTS \( M \) is Semi-confluent (Definition 2.6) by: \( G \mathcal{C}_{L_1}, G \mathcal{C}_{L_2}, L_1 \mathcal{C}_{L_1} \) and \( L_2 \mathcal{C}_{L_1} \).
Figure 2: Schematic of Theorem 3.1 proof for case $a \in G$.

(b) With $k \overset{a'}{\rightarrow} i$ and $k \overset{a}{\rightarrow} k'$, semi-confluence $G_{C_{L_1}}$ implies that there is some state $r' \in Q$ and some $\sigma'_1 \in L'_1$ such that $i \overset{a}{\rightarrow} r'$ and $k' \overset{a}{\rightarrow} r'$. Determinism dictates that $r = r' = i'$. Thus, $j' \overset{a'}{\rightarrow} i'$ and $k' \overset{a}{\rightarrow} i'$. We see that $(i', (j', k')) \in R$.

Now assume $(i, k) \overset{a}{\rightarrow} (j', k')$. By definition of $R$, $j \overset{a}{\rightarrow} i$ and $k \overset{a}{\rightarrow} i$. We have three cases:

1. Case $a \in L_1$. Then, $j \overset{a}{\rightarrow} j'$ for some $j' \in Q$ and $k = k'$. Then by semi-confluence $L_1C_{L_2}$, we get $i \overset{a}{\rightarrow} i'$ for some $i' \in Q$ and $j' \overset{a}{\rightarrow} i'$ for some $\sigma'_2 \in L'_2$. For $k$, we have $k \overset{a}{\rightarrow} i \overset{a}{\rightarrow} i'$. Therefore $k \overset{a}{\rightarrow} i'$ for some $\sigma'_1 \in L'_1$. Thus, we get that $(i', (j', k')) \in R$.

2. Case $a \in L_2$. Then, $j \overset{a}{\rightarrow} k'$ for some $k' \in Q$ and $j = j'$. Then by semi-confluence $L_1C_{L_2}$, we get $i \overset{a}{\rightarrow} i'$ for some $i' \in Q$ and $k' \overset{a}{\rightarrow} i'$ for some $\sigma'_1 \in L'_1$. For $j$, we have $j \overset{a}{\rightarrow} i \overset{a}{\rightarrow} i'$. Therefore $j \overset{a}{\rightarrow} i'$ for some $\sigma'_2 \in L'_2$. Then we see that $(i', (j', k')) \in R$.

3. Case $a \in G$. Then, $j \overset{a}{\rightarrow} j'$ and $k \overset{a}{\rightarrow} k'$ for some $j', k' \in Q$. Then by confluence, $i \overset{a}{\rightarrow} i'$ for some $i' \in Q$ and $j' \overset{a}{\rightarrow} i'$ for some $\sigma'_1 \in L'_1$. We see that $(i', (j', k')) \in R$.

Note that $q^0$ and $(q^0, q^0)$, the initial states of $M$ and $M_1 || G M_2$, respectively, are also related by $R$. Therefore, $M \equiv M_1 || G M_2$. □

4 CONDITIONS OF LEARNING $n$ COMPONENTS IN PARTS

In this section, we generalise the conditions from Section 3 for learning two components in parts to ones that enable learning $n$ components. The theorem and proof are of the same essence as Theorem 3.1.

Theorem 4.1. Let $M$ be some LTS $(Q, A, \rightarrow, q^0)$ and $L_1, \ldots, L_n$ and $G$ be disjoint subsets of $A$ such that $L_1 \cup \ldots \cup L_n \cup G = A$. Let $M_i = \nabla_{L_i}(M)$ where $\nabla$ is the allow operator. Let $M_X = (Q_X, A \rightarrow_X, q^0_X)$ be the parallel composition of all $M_i$ synchronising on $G$. Thus $M_X = M_1 || G \ldots || G M_n$. Then $M \equiv M_X$ if the following three constraints are satisfied:

1. The LTS $M$ maintains pre-access over each $L_i$ (Definition 2.7).
2. The LTS $M$ is Semi-confluent (Definition 2.6) by: $G_{C_{L_1}}$ and $L_iC_{L_2}$ for all $i, j \in \{1, \ldots, n\}$ and $i \neq j$.
3. The LTS $M$ is Deterministic: for all $q \in Q$ and $a \in A$, if $q \overset{a}{\rightarrow} r$ and $q \overset{a}{\rightarrow} s$ then $r = s$.

Proof. Consider the following relation:
$R = \{(j, (k_1, \ldots, k_n)) \mid k_i \overset{a}{\rightarrow} j \text{ where } a \in (\bigcup_{m=1, m \neq i}^n L_m)^* \text{ for all } i = \{1, \ldots, n\}\}$.

We prove that $R$ is a strong bisimulation relation.

1. Assume $j \overset{a}{\rightarrow} j'$.
The way the register machine can be learned using the presented
Active Learning of Decomposable Systems FormaliSE ’20, October 7–8, 2020, Seoul, Republic of Korea
transitioning to three different states. In general, a state in this LTS
that the array of bits can hold in addition to the position of the
pointer is at the rightmost bit moves it to the leftmost bit in one
action when the

5.2 The Learning Setup
Active learning algorithms typically perform $O(m^2)$ membership
queries, where $m$ is the number of states in the target system [13].
The $n$-bit register machine consists of $n \cdot 2^n$ states. This means that
the register machine requires $O((n \cdot 2^n)^2) = O(n^2 \cdot 2^{2n})$ queries. On
the other hand, the total number of states of all the individual
parts is $n \cdot (2n)$. When learning these parts by exploiting Theorem 4.1, the
number of queries becomes $O(n \cdot (2n)^2) = O(n^3)$ queries, similar to
the estimate of [13].

6 DETERMINING THE CONSTRAINTS
The constraints imposed by Theorem 3.1 on the LTSs are quite strong.
Many attempts to weaken these conditions have been carried out
throughout this research. Many invalidated the theorem, but some
honed the conditions into the current form. In this section, we show
a number of counterexamples each weakening one of the three

(a) Case $a \in L_i$. By definition of $R$, we see that $k_1 \xrightarrow{a}^* j$ where
$s_0$ contains no actions from $L_i$. By pre-access over all $L_m$
where $1 \leq m \leq n$ and $m \neq i$, we have $k_1 \xrightarrow{a} \sigma_i$, and then
by semi-confluence rules $L_iC_{L_m}$, we have $k_1 \xrightarrow{a}^* \sigma_i$,
and $\sigma_i$ is in the same language as $s_0$. Determinism dictates
that $r = j'$. Then we see that $(j', (k_1, k_2, \ldots, k_n)) \in R$
because all states $k_i$ where $t \in \{1, \ldots, n\}$ do reach $j'$
without actions from $L_i$.

(b) Case $a \in G$. By pre-access over all $L_i$, we get $k_1 \xrightarrow{a} k'_1$
for $k'_1 \in Q$. Thus, $(k_1, \ldots, k_n) \xrightarrow{a} (k'_1, \ldots, k_n)$
because of synchronisation on $G$ actions. Then by semi-confluence
$GCL_i$, we get $k'_1 \xrightarrow{a'_i} r_i$ where $a'_i$ is in the same language
as $s_0$. Determinism dictates that $r_i = j'$ for all $i$. Then we see that
$(j', (k'_1, \ldots, k'_n)) \in R$.

(2) Assume $(k_1, \ldots, k_n) \xrightarrow{a} f$.
(a) Case $a \in L_i$. Then $f = (k_1, \ldots, k'_i, \ldots, k_n)$. By confluence,
f $\xrightarrow{a'_i} j'$ where $j \xrightarrow{a} j'$. We see that $(j', f) \in R$.
(b) Case $a \in G$. Then $f = (k'_1, \ldots, k'_n)$ and $j \xrightarrow{a} j'$. We see that
$(j', f) \in R$.

5 RIVEST-SCHAPIRE n-BIT REGISTER MACHINE
Here, we describe the $n$-bit register machine from [14] and we show
that Section 3 applies to it.

5.1 Description of the n-bit Register Machine
The machine consists of a register of $n$ bits and one pointer that
points to one bit at a time. Three global actions control this machine:
$G = \{r, l, f\}$. Action $r$ moves the pointer one bit to the right, $l$ to the
left, and $f$ flips the value of the bit under the pointer. The pointer
wraps around the register. So performing an $r$ action when the
pointer is at the rightmost bit moves it to the leftmost bit in one
step. The same is true for action $l$ in the opposite way. We have $2n$
local actions in the set $L$, two actions per bit denoted by $i_0$ and $i_1$
indicating that the value of the $i$th bit is 0 or 1 respectively.

In the LTS, we have one state for each permutation of values
that the array of bits can hold in addition to the position of the
pointer. That means that there are $n \cdot 2^n$ states.

At every state of the LTS, $n$ actions from $L$ are enabled, each indicating
the value of one bit. All $L$ transitions are self-loops because
they do not change the value of any bit nor the pointer.

Moreover, at every state, all three actions $r, l$ and $f$ are enabled,
transitioning to three different states. In general, a state in this LTS
looks like state $t$ in Figure 3.

5.2 The Learning Setup
The way the register machine can be learned using the presented
technique is by first finding a partitioning of local actions that
constitute the non-communicating parts. We introduced the set of
all local actions $L$ which includes two actions per bit $i_0$ and $i_1$. This
can be partitioned into $n$ sets $L_i$ for $i \in \{1, \ldots, n\}$. This way, bit 1
has local action set $L_1 = \{1_0, 1_1\}$; bit 2 has $L_2 = \{2_0, 2_1\}$, etc. Thus, we define the local action sets as $L_i = \{i_0, i_1\}$.

5.3 Conformity to Theorem 4.1
This LTS conforms to our constraints from Theorem 4.1:

(1) Pre-access: over each $L_i$ is satisfied. That is because all $L$
actions make self-loops. Therefore, for $i \in \{1, \ldots, n\}$, any
state $q$ with $q \xrightarrow{f} q$ also has $q \xrightarrow{f}$.

(2) Semi-confluence. For all $j, k \in \{1, \ldots, n\}$, the rule $L_jCL_k$
is satisfied because all $L$ actions make self-loops. Rule $GCL_i$ is
also satisfied; to check this, consider all actions in $G$. For $r$ and
$l$, as demonstrated in Figure 4 (left), the property $(r,l)CL_i$
is satisfied because $r$ and $l$ transitions do not change the values
of registers and thus do not change which of the local ($L$) actions are enabled. For action $f$, as shown in Figure 4 (right) given that
$q \in L_j$, the rule $(f,G)CL_i$ satisfies some state $x$ with $q \xrightarrow{f} x$ and $\beta \xrightarrow{f} x$ where $\beta \in L_i$. In this case, the rule
$(f,G)CL_i$ is satisfied by considering state $x = x$ and sequence
$\beta = i_1 \cdot b$. Alternatively, it is satisfiable by considering $\beta$
to be the empty string $\varepsilon$.

(3) It is Deterministic.

5.4 2-bit Register Demonstration
Figure 5 (top left) shows the 2-bit register LTS. Learning that system
through a partial alphabet, one time blocking bit-2 actions $\{2_0, 2_1\}$
and another blocking bit-1 actions $\{1_0, 1_1\}$, produces the LTSs
at the bottom left and bottom right respectively. The synchronous
product of these two, shown at the top right, is strongly bisimilar
to the original LTS.

We can see how relation $R$ from Theorem 3.1 plays out in the
equivalence shown in Figure 5. For example, the figure shows that
state 7 is strongly bisimilar to state 5 (3). If $R$ captures this equivalence,
then we expect to see a path 5 $\xrightarrow{a_0} \rightarrow 7$ with $a_0 \in \{(2_0, 2_1) \cup G\}$
and a path 3 $\xrightarrow{a_1} \rightarrow 7$ with $a_1 \in \{(1_0, 1_1) \cup G\}$. This is trivially true in
this example because all paths between different states comprise
G actions alone.

5.5 Conclusion of Learning the Rivest-Schapire Example
Active learning algorithms typically perform $O(m^2)$ membership
queries, where $m$ is the number of states in the target system [13].

The $n$-bit register machine consists of $n \cdot 2^n$ states. This means that
the register machine requires $O((n \cdot 2^n)^2) = O(n^2 \cdot 2^{2n})$ queries. On
the other hand, the total number of states of all the individual
parts is $n \cdot (2n)$. When learning these parts by exploiting Theorem 4.1, the
number of queries becomes $O(n \cdot (2n)^2) = O(n^3)$ queries, similar to
the estimate of [13].
We start with an LTS $M$ register machine. Two parts with action sets $\{a, c\}$ and $\{b, c\}$ produces LTSs $M_1$ and $M_2$ respectively. However, the synchronous product $M_1 \parallel_G M_2$ is not strongly bisimilar to $M$. We explain two possible violations of semi-confluence properties through this example that cause inequivalence.

The first semi-confluence property is $G \subseteq L_j$, for $j \in \{1, \ldots, n\}$. Observe in Figure 6 the transitions $3 \xrightarrow{a} 5$ and $3 \xrightarrow{a} 6$. Semi-confluence $\{c\} C(a)$ requires transition $6 \xrightarrow{\epsilon} 5$ which is not in $M$. This results in breaking the equivalence between $M$ and the synchronous product of the two parts $M_1 \parallel \{c\} M_2$ in the following way.

LTS $M_1$ and $M_2$, at states $4$ and $3$ respectively, both synchronise on a transition $c$ toward state $5$. This produces the transition $(4, 3) \xrightarrow{c} (5, 5)$ in the synchronous product which breaks the bisimulation between states $6$ and $(4, 3)$ and thus between the two LTSs.

The second semi-confluence property is $L_i \subseteq C(j)$, for $j, k \in \{1, \ldots, n\}$ which, in this example, translates to $\{a\} C(b)$. State $6$ in $M$ (Figure 6) maintains that property. Assuming that state $6$ is unnecessary and removing it from $M$, the synchronous product would still construct confluent behaviour and would still form state $(4, 3)$. So without state $6$, the equivalence breaks.

## 6.2 Pre-access

By pre-access, we require that if any state in an LTS enables a sequence $e \xrightarrow{a} L_i$ for $i \in \{1, \ldots, n\}$, then it also enables $\xrightarrow{a}$. There are two cases for where $a$ belongs: (1) $a \in L_j$ where $j \in \{1, \ldots, n\}$ but $j \neq i$; and (2) $a \in G$. We give a counterexample for each case.

**Case 1:** Let $e \in L_1$ and $a \in L_2$. Consider the LTS $\xrightarrow{e} a$. Then, $M_1$, learning with blocking $L_2$, is $\xrightarrow{e}$ and $M_2$, learning with blocking $L_1$, cannot perform the $a$ action because it cannot access the state that enables it. Therefore, if a sequence in $L_i$ is enabled only after a sequence in $L_j$, then the equivalence breaks.

**Case 2:** Let $e \in L_i$ for some $i \in \{1, \ldots, n\}$ and $a \in G$. Consider the LTS $\xrightarrow{e} a$. Here, $M_j$, learning with blocking $\{e\}$, has no access to the state that enables the $a$ action. Since $a$ must synchronise in the product of all parts, it must appear in all parts. Since it clearly does not in this example, this breaks the equivalence.
6.3 Determinism

Consider the LTS in Figure 7 with action sets $L_1 = \{a, a_2\}$, $L_2 = \{b\}$, and $G = \emptyset$ which satisfies pre-access and semi-confluence. Note the non-deterministic behaviour at state $k$ on action $a$. Assuming that we can learn non-deterministic behaviour, machine $M_1$ (blocking $b$ actions) can only perform a deterministic $a$ transition. It cannot reach state $k$ to learn the non-deterministic behaviour nor to discover the transition $a_2 \rightarrow b$. Therefore, the product LTS $M_1 ||_G M_2$ is not strongly bisimilar to $M$.

7 AN EXAMPLE FROM THE INDUSTRY

We applied our theorem on the controls of an X-ray machine from Philips Healthcare. In this section, we describe the behaviour of these controls, and we describe how we partitioned actions into sets used to learn the behaviour in parts. More importantly, this section provides an insight on the applicability and practical challenges of applying the theorem.

7.1 An Informal Description of the Behaviour and the Actions

The machine has two x-ray tubes positioned around the bed in two orthogonal axes, frontal and lateral. A pair of joysticks control the precise position of these two tubes. A set of foot pedals power the radiation on and off. A pair of buttons zoom-in/zoom-out the resulting image. And finally, a keyboard is used to program various other parameters. The machine can take single snapshot images, but it can also display a live image with continuous radiation, and it is this latter mode of use that is more intricate and interesting to us.

There are two live image modes: a lower radiation mode called Fluoroscopy, shortened Fluo, and a higher radiation mode called Exposure. The image can be taken through either the frontal tube, the lateral tube, or through an interleaving of both tubes called the Biplane. Each of these makes an image generation channel. So we have three channels: frontal, lateral, and biplane.

7.2 The System under Learning

In this experiment, we chose to simulate the described part of the X-ray machine in the mCRL2 toolset [6]. We wrote a specification in the mCRL2 language, shown in Appendix A, capturing the informal description above. We defined each of the learnt parts by the action set it allows. The details of the action set for each part are given below.

7.3 The Alphabet and its Partitioning

We lay out the complete set of actions that constitute the behaviour, and we divide it into independent subsets of actions, i.e., actions that do not enable or disable other actions. Pedals enable radiation, and some of them override others. Therefore, all pedals need to be in the same subset together with radiation actions. Moreover, the two tubes cannot radiate independently of each other; therefore, no splitting is possible between them. Our first set, therefore, $\Sigma_1$, contains actions press/release(Pedals) and radiate(Tube$_{f/l}$, type, intensity). Using the keyboard takes the machine into ‘programming mode’ (prog mode for short) where various parameters can be programmed such as the intensity of the beam. Prog mode interrupts radiation even while the corresponding pedal is pressed. Only after we exit prog mode is the radiation allowed to resume (typically with new settings). Since entering and exiting prog mode influences actions in $\Sigma_1$, then keyboard actions are inseparable from $\Sigma_1$, thus we include them.

We are left with the zoom buttons and the joysticks which respectively make action sets $\Sigma_2$ and $\Sigma_3$. Note that both zooming on the live generated image and moving and rotating the tubes do not interfere with its radiation.

Global Actions. The set of global actions, which influences the behaviour of all components, includes (On, Off) which turn the X-ray machine on and off respectively. In addition, when the machine starts or stops radiation, a pair of internal actions RadOn and RadOff synchronise across components.

Satisfying the Theorem. As explained, the actions are partitioned such that the conditions in our theorem is satisfied. This is done intuitively through our understanding of the conditions (Section 2). For example, an action from one set blocking (disabling) an action from another set immediately breaks the pre-access condition. That is why we placed all actions that can interrupt radiation or affect the function of the pedals in the same set as the pedals.

Semi-confluence between two actions similarly follows from the concept that neither one enables or disables the other.

As for determinism, the behaviour informally described is clearly deterministic.

7.4 The Learning Setup and the Resulting Automata

After specifying the system and the partitioning of the action set, as shown in Appendix A, we built an automaton, using mCRL2, for the entire system without any alphabet partitioning. We call this the original automaton and its size is 1.5 million states. Then we produced one automaton for each part by allowing only that part’s local actions in addition to the global actions. The global actions are always allowed. This produced six models in total: (1) The pedals component consisting of 2340 states and 15 thousand transitions of which a reduced version is shown in Figure 10; (2) The zoom component (Figure 9) of eight states; and four components for joystick controls: the frontal angulate beam, the lateral angulate
We presented a theorem on the conditions for learning systems in parts. We formalised them as three constraints, gave the proof as well as counterexamples against possible weakenings of those constraints.

7.5 Results
We used mCRL2 to compute the synchronous product of these six automata which resulted in a model of over 1.5 million states and 18 million transitions, matching the size of the original automaton. We compared both automata, original with synchronous product modulo strong bisimulation, and indeed the tool verified that they are strongly bisimilar.

Therefore, the original LTS has 600 times more states than all the parallel components combined. As explained in Section 5.5, the number of queries a typical learning algorithm performs to merely build a hypothesis for this system, let alone what is needed to check it, is in the order of 2 trillion queries whereas the matching number of queries needed for learning the system in parts is in the order of only 6 million queries. This result proves a dramatic decrease in the learning time and is promising enough to work towards employing these techniques in a real industrial setting.

8 FUTURE WORK
The questions we have for future work are primarily about the practicality of this result, especially that this work is motivated by a case from the industry. The questions are:

1. In a practical setup, how do we verify, with minimal effort, that a system satisfies the constraints for learning in parts?
2. Supposing the conditions of learning in parts are indeed satisfiable on a certain system given a certain partitioning of actions, then how do we find that partitioning?

9 CONCLUSION
We wish to thank Rick Erkens, Joshua Moerman, Hans van Wezep, Olav Bunte, Arjan Mooij, Ferry Timmers, Mark Bouwman and Thomas Neele for sharing their most valuable insight.

ACKNOWLEDGMENTS
Through our industrial example, we demonstrated that, with a simple splitting of actions, we can learn in parts a number of states that is an order of hundreds of times less than what is needed to be learned otherwise and costs hundreds of thousands less queries.

REFERENCES
A SPECIFICATION OF X-RAY MACHINE EXAMPLE IN MCRL2

This appendix includes excerpts from the mCRL2 specification of Active Learning of Decomposable Systems. The process is defined as the choice of multiple possible actions where \( + \) is the choice operator. The \( \Rightarrow \) is the sequential operator. The \( \Rightarrow \) operator means ‘if then’ and the \( \Rightarrow \) operator means ‘append to list’. The \( \Rightarrow \) operator means ‘if then’ and the \( \Rightarrow \) operator means ‘append to list’.

In most of the choices of process, we see a condition enabling an action followed sequentially by a call to the process with certain parameters. If \( S \) is called with empty parentheses like \( S() \), then it is called without changing the value of any parameter. Sometimes only the key parameter is specified for brevity. Furthermore, we shorten the names of the previously defined sorts and parameters according to Table 1.

### Table 1: Abbreviations of variable names in mCRL2 specification.

```
<table>
<thead>
<tr>
<th>Abbreviated</th>
<th>Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>k</td>
<td>knobSetting</td>
</tr>
<tr>
<td>ff</td>
<td>fluoFlavor</td>
</tr>
<tr>
<td>radS</td>
<td>radStatus</td>
</tr>
<tr>
<td>ed</td>
<td>isInEditMode</td>
</tr>
<tr>
<td>radON</td>
<td>radiationStatusON</td>
</tr>
<tr>
<td>radOFF</td>
<td>radiationStatusOFF</td>
</tr>
<tr>
<td>isR0</td>
<td>isRadDisallowed</td>
</tr>
<tr>
<td>inb</td>
<td>inputBuffer</td>
</tr>
<tr>
<td>flk</td>
<td>isFluqLockOn</td>
</tr>
<tr>
<td>fabp</td>
<td>fAngBeamPos</td>
</tr>
<tr>
<td>frbp</td>
<td>iRotBeamPos</td>
</tr>
<tr>
<td>labp</td>
<td>lAngBeamPos</td>
</tr>
<tr>
<td>lrbp</td>
<td>lRotBeamPos</td>
</tr>
</tbody>
</table>
```

#### Example Program

```mcrl2
sort RadiationMode = struct (Fluo | ExposureMode | SingleShotMode | ProgMode);
sort RadiationChannel = struct (Frontal | Lateral | Biplane);
sort Pedal = struct (FluoFrontal | FluoLateral | FluoBiplane | Exposure | SingleShot);
sort Beam = struct (frontalAngle | frontalRotate | lateralAngle | lateralRotate);

Then a list of actions is defined. Some actions can take a certain sort as an argument such as the press and release actions which operate on a certain pedal.

```mcrl2
set press, release Pedal;
radON, RadiationMode # RadiationChannel;
radOFF, updateParam, editParam, cancelEditParam;
select Fluoflavor, setFFlavor, FFlavor;
RFSInst, RFSDec;
Inc Beam, Dec Beam;
```

Then internal functions that perform operations such as type conversions or data manipulation or pre-processing are defined. These serve to make the specification more modular, but are not central to our purpose, so we skip them.

Next, the main process is defined. The process is represented as a state machine. We see the signature of the process \( S \) represented like a state where the parameters of \( S \) are held by the state and their values are what determines a state in process \( S \). The colon operator means ‘of type’. Process \( S \) is defined as the choice of multiple possible actions where \( + \) is the choice operator. The \( \Rightarrow \) is the sequential operator. The \( \Rightarrow \) operator means ‘if then’ and the \( \Rightarrow \) operator means ‘append to list’.

In most of the choices of process, we see a condition enabling an action followed sequentially by a call to the process with certain parameters. If \( S \) is called with empty parentheses like \( S() \), then it is called without changing the value of any parameter. Sometimes only the key parameter is specified for brevity. Furthermore, we shorten the names of the previously defined sorts and parameters according to Table 1.
Proc Sinit describes the original behaviour. After that, each component is defined through a subset of the action set through the allow operator. For example the zoom component is defined by:

\[
\text{proc zoomIn} = \text{allowZoom} \\
\text{proc zoomOut} = \text{allowZoom} \quad |\quad \text{proc Total} = \text{comm}
\]

Next, the components are composed together using the synchronous product using the parallel operator \(||\). The synchronisation is specified, using the comm operator, on global actions On, Off, RadOn and RadOff. The product is then compared with the total behaviour modulo strong bisimulation using the mCRL2 toolset.