






Componentwise Automata Learning for System Integration (Extended Version)

Hiroya Fujinami^{1,5} , Masaki Waga^{2,1} , Jie An^{3*} , Kohei Suenaga^{2,1} ,
Nayuta Yanagisawa^{4**}, Hiroki Iseri^{4**}, and Ichiro Hasuo^{1,5,6} 

¹ National Institute of Informatics, Tokyo, Japan
`{makenowjust,hasuo}@nii.ac.jp`

² Kyoto University, Kyoto, Japan

`{mwaga,ksuenaga}@fos.kuis.kyoto-u.ac.jp`

³ Institute of Software, Chinese Academy of Sciences, Beijing, China
`anjie@iscas.ac.cn`

⁴ Toyota Motor Corporation, Tokyo, Japan

`{nayuta_yanagisawa,hiroki_iseri}@mail.toyota.co.jp`

⁵ SOKENDAI (The Graduate University for Advanced Studies), Kanagawa, Japan

⁶ Imiron Co., Ltd., Tokyo, Japan

Abstract. *Compositional automata learning* is attracting attention as an analysis technique for complex black-box systems. It exploits a target system’s internal compositional structure to reduce complexity. In this paper, we identify *system integration*—the process of building a new system as a composite of potentially third-party and black-box components—as a new application domain of compositional automata learning. Accordingly, we propose a new problem setting, where the learner has direct access to black-box components. This is in contrast with the usual problem settings of compositional learning, where the target is a legacy black-box system and queries can only be made to the whole system (but not to components). We call our problem *componentwise automata learning* for distinction. We identify a challenge there called *component redundancies*: some parts of components may not contribute to system-level behaviors, and learning them incurs unnecessary effort. We introduce a *contextual componentwise learning* algorithm that systematically removes such redundancies. We experimentally evaluate our proposal and show its practical relevance.

Keywords: automata learning · compositional automata learning · systems engineering · Moore machine

* J.A.’s technical contribution was made when he was at National Institute of Informatics, Tokyo, Japan.

** This paper presents a theoretical investigation that is independent of any testing procedures conducted at the institutions or companies with which the industry-affiliated co-authors are associated.

1 Introduction

Automata Learning (*Active automata learning*) is a problem to infer an automaton recognizing the target language $\mathcal{L}_{\text{tgt}} \subseteq \Sigma^*$ via a finite number of queries to an oracle. The L^* algorithm [2], the best known active automata learning algorithm by Angluin, infers the minimum DFA recognizing the target regular language \mathcal{L}_{tgt} via two kinds of queries: *membership* and *equivalence* queries.

- In a membership query, the learner asks if a word $w \in \Sigma^*$ is in \mathcal{L}_{tgt} . (For machines with output (e.g. Mealy and Moore), membership queries are called *output queries*, a term we will be using in this paper.) The answers to those membership/output queries are recorded in an (*observation*) *table*; once the table is *closed* it induces a *hypothesis DFA*.
- In an equivalence query, the learner asks if a hypothesis DFA \mathcal{A}_{hyp} recognizes \mathcal{L}_{tgt} . If not, the oracle returns a *counterexample* $\text{cex} \in \mathcal{L}_{\text{tgt}} \triangle \mathcal{L}(\mathcal{A}_{\text{hyp}})$ that witnesses the deviation of \mathcal{A}_{hyp} ’s language from \mathcal{L}_{tgt} .

A target of automata learning is commonly called a *system under learning (SUL)*.

After the seminal work [2], various algorithms have been proposed, for example, to improve the efficiency [11, 25, 28] and to learn other classes of automata (e.g. Mealy machines [21], weighted automata [10, 18], symbolic automata [3, 5, 7], and visibly pushdown automata [1]). The LearnLib library offers an open source framework for automata learning [12]. Many real-world applications of automata learning have been reported, too. See e.g. [4, 6].

In the context of verification and testing, active automata learning is used to approximate *black-box* systems and obtain a surrogate model amenable to white-box analysis. For example, automata learning of Moore or Mealy machines has been applied for model checking [19, 23, 26] and controller synthesis [29].

Compositional Automata Learning Recently, algorithms for *compositional automata learning* are attracting attention [6, 8, 14, 15, 20]. Assuming that the SUL M is a composition of some subsystems M_1, \dots, M_n (called *components*), those algorithms try to learn individual components M_i and construct a model of M as their composition, rather than *monolithically* learning the SUL M itself.

A major benefit of such compositional approaches is *complexity*: if each M_i has k_i states, the SUL has $k_1 \times \dots \times k_n$ states and the monolithic learning has to learn these, while the compositional learning has to learn only $k_1 + \dots + k_n$ states in total. Since many real-world systems are constructed using components, compositional automata learning is a promising approach to scalable learning.

It is important to note that different compositional automata learning algorithms assume very different problem settings. The differences lie in the type of automata to learn, how they are composed, the learning interface, etc. We will make a detailed comparison later; its summary is in Table 1.

In most existing works including [6, 8, 14, 15, 20], the learner has no access to individual components to make queries. It thus tries to learn components indirectly via system-level queries. A typical application scenario is where the SUL M is a *legacy* black-box system: M ’s compositional structure may be known, e.g. via old documentation; yet M ’s components are buried in the black-box system

M and their interface is not exposed. In this case, the technical challenge is *how to throw component-level queries indirectly*, that is, to translate component-level queries (that the learner wants to ask) to system-level queries (that the learner can ask in reality). The works [6, 8, 14, 15, 20] propose different solutions to this challenge, specializing in each problem setting.

Our problem setting—we call it *componentwise learning* for distinction—is very different from the above; so is the main technical challenge there. We first motivate our problem setting with *system integration* as application.

Motivation: System Integration with Black-Box Components *System integration (SI)* in ICT industry refers to “the process of creating a complex information system that may include designing or building a customized architecture or application, integrating it with new or existing hardware, packaged and custom software, and communications.”⁷ SI is nowadays a norm in various layers of ICT system development:

- Large-scale ICT systems for banks, e-commerce, and other business processes are products of SI where different software components, typically developed by different parties, get integrated.
- Smaller software pieces also rely on existing software components offered as libraries (e.g. `pip` for Python). They can be thus seen as products of SI.

SI is not unique to ICT. In fact, our original motivation comes from the automotive industry, where various systems (a car, an engine, control software, etc.) get built by assembling parts that are often manufactured by other parties.

In this paper, a body that conducts SI is called a *system integrator (SIer)*. SIers have to make sure that the composite system behaves as expected. This is not easy, however, since components that constitute the composite system are usually black-box systems. This situation thus makes SI a natural target of automata learning. Moreover, *compositional* automata learning can be used, since an SIer knows the compositional structure that combines black-box components.

Contribution: Contextual Componentwise Automata Learning In SI, the learner (an SIer) is building a *new* composite system. The learner has (raw) component in its hands, and thus has direct interface for component-level queries. This is in stark contrast with other works [6, 8, 14, 15, 20] where the target is an *old* (legacy) system and the main challenge is indirect component-level queries. To highlight this difference, we use the term *componentwise automata learning* for compositional learning where direct component-level queries are available.

A new challenge that we face in componentwise automata learning is *component redundancies*. In an SI scenario (no matter if it is ICT or automotive), components are rarely *lean*, meaning that most of the time they come with more functionalities than an SIer needs for the composite system. Learning those *rich* components holistically, including redundancies, is costly and wasteful.

This problem of redundancy is practically relevant. It is widely recognized in software engineering, resulting in active research on *dead code identification* and *elimination* (see e.g. [17]). As a specific example, in our `MQTTLighting`

⁷ Gartner Information Technology Glossary, <https://www.gartner.com/en/information-technology/glossary/system-integration>

benchmark (§5), there is a component that can handle many different modes of a communication protocol, but the composite system uses only one mode.

Towards the goal of eliminating component redundancies, we devise a *contextual* componentwise automata learning algorithm, where observation tables for automata learning are pruned to system-level relevant behaviors.

To describe how our algorithm works, we first introduce our system model.

Formalization by Moore Machine Networks In this paper, we model each component as a Moore machine (MM), and their composition as what we call a *Moore machine network (MMN)*. The latter arranges Moore machine components as nodes of a graph, and edges of the graph designate either system-level or inter-component input/output. An example is in Ex. 1.1, where two component MMs operate, driven by system-level input words. The component M_{c1} passes its output to M_{c2} , and M_{c2} produces system-level output.

Components of an MMN operate in a fully synchronized manner. They share the same clock, and at each tick of it, each component M_c produces an output character a_e at each outgoing edge e from M_c . In case the edge e points to another component $M_{c'}$, the character a_e becomes (the e -component of) the input character to $M_{c'}$ at that tick. System-level input/output characters are consumed/produced synchronously, too. (The choice of Moore machines over Mealy machines is crucial for this operational semantics; see Appendix C.1.)

Therefore our formalism models *structured*, *synchronized* and *dense* composition of components. This is suited for system integration, where 1) the learner (the SIer) arranges components with explicit interconnections, and 2) many components continuously receive signals from, and send signals to, other components (as is the case with many automotive, cyber-physical, web, and other systems).

This formalization of ours is in contrast with *flat*, (mostly) *interleaving* and *sparse* composition of components in other compositional works [6, 14, 15, 20]. This difference mirrors different target applications. See Table 1.

Context Analysis by Reachability Analysis in a Product Based on the MMN formalization, we shall sketch our technique for eliminating component redundancies. Component redundancy gets formalized as the fact that *system-level input words do not necessarily induce all component-level input words*. To see which input character a component M_c can receive at its state q_c , it suffices to know at which state $q_{c'}$ every other component $M_{c'}$ can be at the same time.

We conduct this *context analysis* (CA) using the hypothesis automata learned so far for the components. Identifying all state tuples $(q_c, (q_{c'})_{c'})$ which can be simultaneously active is done by the reachability analysis in the product automaton of the hypotheses. This can

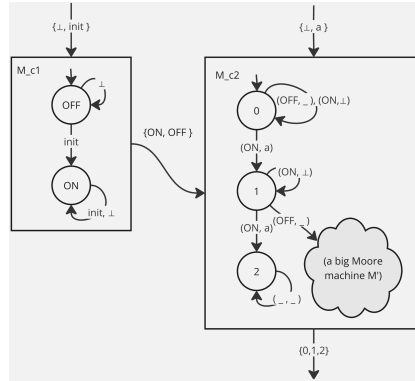


Fig. 1: an MMN example, a counter with initialization

be costly; we therefore introduce two *context analysis parameters (CA-parameters)*, namely \mathcal{E} (for abstracting contexts by quotienting hypothesis automata) and \mathcal{R} (for limiting reachability analysis).

These parameters give us flexibility in the cost-benefit trade-off of context analysis. This kind of flexibility is important in automata learning since different application scenarios have different cost models. Specifically, running the SUL can be costly—processing one input character can take some milliseconds (in an embedded system) or even seconds (in a hardware-in-the-loop simulation (HILS) setting). In this case, extensive CA on the learner side (that can use a fast laptop) will pay off. In other cases where the SUL is fast, the cost of CA can be a bottleneck, and we might choose a cheaper and coarser CA-parameters.

We evaluated our contextual componentwise learning algorithm (called CCwL*) with experiments. Comparison is against two baselines: 1) *monolithic L** (MnL^*) that learns the whole SUL, 2) (naive) *componentwise L** (CwL^*) that learns each component individually (without CA and thus component redundancies). We used a few realistic benchmarks, including one inspired by robotics application, together with some toy benchmarks. We also evaluated the effect of different CA-parameters (\mathcal{E}, \mathcal{R}). Overall, the experiment results indicate the value of our algorithm in application scenarios of system integration.

Example 1.1 (counter with initialization). The MMN in Fig. 1 consists of two component MMs (M_{c_1} and M_{c_2}). Each has a system-level input edge with the designated alphabet, and M_{c_2} has a system-level output edge. The output of M_{c_1} is plugged in as input of M_{c_2} , too.

This example exhibits component redundancies: the M' part of M_{c_2} is irrelevant to system-level behaviors. Indeed, M' is never activated—once M_{c_1} moves to the ON state, it never goes back to OFF. Our contextual componentwise learning algorithm detects and exploits this fact; it learns M_{c_1} and M_{c_2} separately, but in the latter it prunes the unreachable part M' .

Contributions Our contributions are summarized as follows.

- We identify *system integration* as a new application domain of compositional automata learning. There, component-level queries are fully available; we use the term *componentwise automata learning* for distinction.
- We formalize the problem using *Moore machine networks* and identify the main challenge to be eliminating component redundancies.
- We introduce a *contextual* componentwise learning algorithm. It eliminates component redundancies by pruning observation tables using reachability analysis in the product of (hypothesis automata for) the components.
- We show its practical values through experiments.

Related Work Many works on automata learning in general have been already discussed; here we focus on compositional approaches. A comparison of works on compositional automata learning is summarized in Table 1. All works but ours allow only system-level queries, and many are aimed at learning a *legacy* black-box system. In contrast, in our system integration applications, we are usually building a *new* system.

Table 1: comparison of compositional automata learning frameworks, settings and challenges. Shading is made to signify the span of combined cells

	current work	[15]	[20]	[6]	[14]	[8]
typical application	system integration	analysis of legacy systems				learning beyond regular
querying interface	system- & component-level	system-level only				
target systems	Moore machine networks	Mealy machines	LTSs	LTSs	Mealy machines	SPAs
component interaction	structured, synchronized, dense	flat, (mostly) interleaving, sparse				procedure calls
challenge	eliminating component redundancies	querying components via system-level queries				

The compositional algorithm in [15] assumes that the SUL is a parallel composition of Mealy machines M_1, \dots, M_n whose input alphabets $\Sigma_1, \dots, \Sigma_n$ are disjoint. The components operate in the interleaved manner, where each component M_i takes care of those input characters in Σ_i . The partition $\Sigma = \Sigma_1 \sqcup \dots \sqcup \Sigma_n$ as well as the number n of components is not known to the learner, and the challenge is to find them. Their algorithm first assumes the finest partition ($n = |\Sigma|$ and each Σ_i is a singleton), and merges them in a counterexample-guided manner, when it is found that some input characters must be correlated.

The algorithm in [20] relies on more specific assumptions, namely that 1) the SUL is a parallel composition of LTSs which can synchronize by shared input characters, and 2) the output/observation to input words is whether the system gets stuck (i.e. there is no outgoing transition). The challenge here is that, when the SUL gets stuck for an input word w and w contains characters shared by different components, the learner may not know which component to blame. Their solution consists of 1) constructing an “access word” w' that extends w and localizes the blame, and 2) allowing “unknown” in observation tables in case there is no such w' . Unlike in [15], the number of components and their input alphabets must be known. The work [20] shows that some Petri nets yield such combinations of LTSs; it is not clear how other types of systems (such as Mealy machines) can be learned by this algorithm.

The works [6, 14] can be thought of as variations of [15] with similar problem settings. In [6], the disjointness assumption in [15] is relaxed, and they give some graph-theoretic conditions that enable compositional learning. These conditions, however, are detailed and the learner has to somehow know that they hold in the SUL. In [6], dually to [15], they separate components according to output. It is yet to be identified what SULs are suited for this algorithm.

The work [8] has a different flavor from others. It is in the line of work on learning more expressive formalisms than (usual) automata and regular languages, such as (visibly) pushdown automata. Indeed, their target systems (system of procedural automata, SPA) are described much like context-free gram-

mars. They assume that the invocation of nonterminals (*procedure calls* in the SPA terminology) is observable; this enables application of automata learning. Otherwise the setting is similar to those in [6, 14, 15, 20]; in particular, the challenge is the same, namely to query components via system-level queries.

Besides the works compared in Table 1, *distributed reactive synthesis* [24] and *synthesis from component libraries* [16] are related to our work in their emphasis on compositionality. These works target at *synthesis* of automata from given logical specifications, a goal different from ours or the works in Table 1 (namely active automata learning).

Notations $X \sqcup Y$ denotes the disjoint union of sets X and Y . The powerset of X is denoted by 2^X . The set of all partial functions from X to Y is denoted by $X \rightharpoonup Y$. For $f: X \rightharpoonup Y$ and $x \in X$, we write $f(x) \downarrow$ if $f(x)$ is defined, and $f(x) \uparrow$ otherwise.

Given an equivalence relation $\sim \subseteq X \times X$, equivalence classes are denoted by $[x]_\sim$ using $x \in X$, and the quotient set is denoted by X/\sim , as usual.

2 Problem Formalization by Moore Machine Networks

We start by some basic definitions. Let X be a nonempty finite set. X^* denotes the set of finite strings (also called words) over X ; ε denotes the empty string; $|s|$ denotes the length of $s \in X^*$; and $s_1 \cdot s_2$ (or simply $s_1 s_2$) denotes the concatenation of strings $s_1, s_2 \in X^*$.

In this paper, we use the 0-based indexing for strings. For a string $s \in X^*$ and an integer $i \in [0, |s|]$, $s[i]$ denotes the $(i+1)$ -th character of s (thus $s = s[0]s[1] \dots s[|s|-1]$); for $i, j \in [0, |s|]$ such that $i \leq j$, $s[i, j)$ denotes the substring $s[i]s[i+1] \dots s[j-1]$. Note that $s[i, i) = \varepsilon$ for each i .

Let $(X_k)_{k \in K}$ be a (K -indexed) family of sets. Its product is denoted by $\prod_{k \in K} X_k$, with its element denoted by a tuple $(x_k)_{k \in K}$ (here $x_k \in X_k$).

The restriction of a tuple $t = (x_k)_{k \in K} \in \prod_{k \in K} X_k$ to a subset $K' \subseteq K$, denoted by $t|_{K'}$, is $(x_k)_{k \in K'} \in \prod_{k \in K'} X_k$. This restriction of tuples t along $K' \subseteq K$ is extended, in a natural pointwise manner, to subsets $S \subseteq \prod_{k \in K} X_k$ and sequences $s \in (\prod_{k \in K} X_k)^*$, resulting in the notations $S|_{K'}$ and $s|_{K'}$.

2.1 Moore Machines

Our algorithm learns the following (deterministic) Moore machines (MMs).

Definition 2.1 (Moore machine). A *Moore machine* (MM) is a tuple $M = (Q, q_0, I, O, \Delta, \lambda)$, where

- Q is a finite set of states, $q_0 \in Q$ is an initial state,
- I is an input alphabet, O is an output alphabet,
- $\Delta: Q \times I \rightharpoonup Q$ is a transition (partial) function, and
- $\lambda: Q \rightarrow O$ is an output function that assigns an output symbol to each state.

A Moore machine is *complete* if $\delta(q, i) \downarrow$ for all $q \in Q$ and $i \in I$; otherwise, it is called *partial*.

We will also use *nondeterministic* MMs, later in §4, but only for the purpose of approximate context analysis (CA). It is emphasized that we do *not* learn nondeterministic MMs. The theory of nondeterministic MMs (their definition, semantics, etc.) is obtained in a straightforward manner; it is in Appendix A.1.

As usual, the transition function Δ can be extended to an input string $w \in I^*$. Precisely, $\Delta(q, \varepsilon) = q$ and $\Delta(q, wi) = \Delta(\Delta(q, w), i)$. Similarly, the output function is extended by $\lambda(q, w) = \lambda(\Delta(q, w))$. When starting from the initial state q_0 , often we simply write $\Delta(w) = \Delta(q_0, w)$ and $\lambda(w) = \lambda(q_0, w)$.

Given a Moore machine $M = (Q, q_0, I, O, \Delta, \lambda)$ and a state $q \in Q$, the *semantics* of M , denoted by $\llbracket M \rrbracket_q: I^* \rightarrow O^*$ and defined below, represents the behavior of the machine when starting from q . For each $w \in I^*$,

$$\llbracket M \rrbracket_q(w) = \lambda(q, w_{[0,0)}) \lambda(q, w_{[0,1)}) \cdots \lambda(q, w_{[0,k)}), \quad (1)$$

where k is the smallest number such that $\lambda(q, w_{[0,k+1)})$ is undefined, or $|w|$ if no such k exists. Note that $\llbracket M \rrbracket_q: I^* \rightarrow O^*$ is a *total* function: even if Δ gets stuck (making $\lambda(q, w_{[0,k+1)})$ undefined), it does not make $\llbracket M \rrbracket_q(w)$ undefined, while it does make $\llbracket M \rrbracket_q(w)$ shorter. When starting from the initial state q_0 , we write $\llbracket M \rrbracket(w)$ for $\llbracket M \rrbracket_{q_0}(w)$ (much like for Δ and λ).

Using this semantics, we define the equivalence of Moore machines.

Definition 2.2 (equivalence of Moore machines). Two Moore machines M_1 and M_2 are said to be *equivalent* if and only if $\llbracket M_1 \rrbracket = \llbracket M_2 \rrbracket$.

2.2 Moore Machine Networks

A *directed graph* is a tuple $G = (V, E)$ of a finite set V of nodes (or vertices) and a set $E \subseteq V \times V$ of (directed) edges.

Let $G = (V, E)$ be a directed graph. Let E_v^{in} denote the set of *incoming edges* for a node $v \in V$, i.e. $E_v^{\text{in}} = \{(u, v) \in E \mid u \in V\}$. Similarly, E_v^{out} denotes the set of *outgoing edges* ($E_v^{\text{out}} = \{(v, u) \in E \mid u \in V\}$). For a set $U \subseteq V$ of nodes, we define $E_U^{\text{in}} = \bigcup_{u \in U} E_u^{\text{in}}$ and $E_U^{\text{out}} = \bigcup_{u \in U} E_u^{\text{out}}$.

Towards our definition of MMNs (Def. 2.3; see also Fig. 1), we introduce the following classification of nodes: a node $v \in V$ is 1) a *system-level input node* if $E_v^{\text{in}} = \emptyset$, 2) a *system-level output node* if $E_v^{\text{out}} = \emptyset$, and 3) a *component node* otherwise. We denote the sets of input, output, and component nodes by V^{in} , V^{out} , and V^c , respectively. We impose the condition on $G = (V, E)$ that $V = V^{\text{in}} \sqcup V^{\text{out}} \sqcup V^c$ is a disjoint union of three nonempty sets. (See Fig. 1, where system-level input and output nodes are implicit.)

An edge $e = (v, v') \in E$ is called a *system-level input edge* if $v \in V^{\text{in}}$, and a *system-level output edge* if $v' \in V^{\text{out}}$. The set of system-level input and output edges of $G = (V, E)$ are denoted by E^{in} and E^{out} , respectively, and are given by $E^{\text{in}} = E_{V^{\text{in}}}^{\text{out}}$ and $E^{\text{out}} = E_{V^{\text{out}}}^{\text{in}}$.

A *Moore machine network (MMN)* is a directed graph, with a Moore machine associated with each component node $c \in V^c$. Here we present a deterministic definition. Later in §4 we also use a nondeterministic version (not to learn, but for CA); this is an easy adaptation. See Appendix A.2 for explicit definitions.

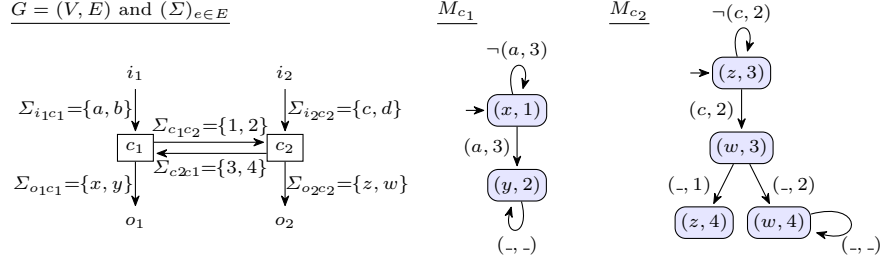


Fig. 2: an example MMN \mathcal{M}_{ex} . On the left we show its network $G = (V, E)$ and the alphabets $(\Sigma_e)_{e \in E}$ for edges. The component MMs are shown on the right, where state labels designate output. In the transition labels, $\neg i$ stands for all characters other than i , and the symbol $_$ matches any character

Definition 2.3 (Moore machine network). A (deterministic) *Moore machine network* (MMN) is a tuple $\mathcal{M} = (G, (\Sigma_e)_{e \in E}, (M_c)_{c \in V^c})$, where

- $G = (V, E)$ is a directed graph representing the network structure,
- Σ_e is an alphabet associated with each edge $e \in E$, and
- M_c is a (deterministic) Moore machine associated with a component $c \in V^c$.

On each component Moore machine $M_c = (Q_c, q_{0,c}, \Sigma_c^{\text{in}}, \Sigma_c^{\text{out}}, \Delta_c, \lambda_c)$, we require that its input and output alphabets are in accordance with the edge alphabets Σ_e . Specifically, we require $\Sigma_c^{\text{in}} = \prod_{e \in E_c^{\text{in}}} \Sigma_e$ (the product of the alphabets of all incoming edges) and, similarly, $\Sigma_c^{\text{out}} = \prod_{e \in E_c^{\text{out}}} \Sigma_e$.

For an MMN \mathcal{M} , we define three *system-wide alphabets*: 1) $\Sigma^{\text{in}} = \prod_{e \in E^{\text{in}}} \Sigma_e$ is the *system-level input alphabet*, 2) $\Sigma^{\text{out}} = \prod_{e \in E^{\text{out}}} \Sigma_e$ is the *system-level output alphabet*, and 3) $\bar{\Sigma} = \prod_{e \in E \setminus E^{\text{in}}} \Sigma_e = \prod_{c \in V^c} \Sigma_c^{\text{out}}$ is the *total output alphabet*. Note that $\bar{\Sigma}$ collects also those characters sent from a component to another.

Example 2.4. An example of MMN is in Fig. 2. Its detailed formalization is in Appendix C.2.

We move on to define the semantics of MMNs. It is via a translation of an MMN \mathcal{M} to an MM $[\mathcal{M}]$; in the translation, the component MMs in \mathcal{M} operate in a fully synchronized manner. The definition is intuitively straightforward, although its precise description below involves somewhat heavy notations.

The set of (*system-level*) *configurations* of \mathcal{M} , denoted by \mathbf{Q} , is defined by $\mathbf{Q} = \prod_{c \in V^c} Q_c$. The *initial configuration* is $\mathbf{q}_0 = (q_{0,c})_{c \in V^c}$.

Given a configuration $\mathbf{q} = (q_c)_{c \in V^c} \in \mathbf{Q}$, the *total output* of \mathcal{M} at \mathbf{q} , denoted by $\bar{\lambda}(\mathbf{q}) \in \bar{\Sigma}$, is defined by $\bar{\lambda}(\mathbf{q}) = (\lambda_c(q_c))_{c \in V^c}$. Similarly, the *system-level output* of \mathcal{M} at \mathbf{q} is defined by $\lambda(\mathbf{q}) = \bar{\lambda}(\mathbf{q})|_{E^{\text{out}}} \in \Sigma^{\text{out}}$. (Recall the restriction operation | from §2.)

Given a configuration $\mathbf{q} = (q_c)_{c \in V^c} \in \mathbf{Q}$ and a system-level input character $\mathbf{i} \in \Sigma^{\text{in}}$, we define Δ , the *system-level transition function* of \mathcal{M} , by $\Delta(\mathbf{q}, \mathbf{i}) =$

$\left(\Delta_c(q_c, (\mathbf{i}, \bar{\lambda}(\mathbf{q}))|_{E_c^{\text{in}}}) \right)_{c \in V^c}$. Intuitively: the tuple $\bar{\lambda}(\mathbf{q})$ of characters is output from the current states $\mathbf{q} = (q_c)_{c \in V^c}$; it is combined with the system-level input \mathbf{i} and fed to each component's transition function Δ_c .

We formalize the following definition, using the above constructions.

Definition 2.5 (Moore machine $[\mathcal{M}]$). Let \mathcal{M} be an MMN. The *Moore machine* $[\mathcal{M}]$ induced by \mathcal{M} is $[\mathcal{M}] = (\mathbf{Q}, \mathbf{q}_0, \Sigma^{\text{in}}, \Sigma^{\text{out}}, \Delta, \lambda)$.

The semantics of \mathcal{M} is defined by $\llbracket \mathcal{M} \rrbracket_{\mathbf{q}} = \llbracket [\mathcal{M}] \rrbracket_{\mathbf{q}}$ for any $\mathbf{q} \in \mathbf{Q}$.

When we turn to completeness of MMs (Def. 2.1), the completeness of $[\mathcal{M}]$ does not necessarily imply that of each component MM M_c . This is obvious from Ex. 1.1—it does not matter even if some transitions are not defined in M' , since M' is never invoked. This is an implication of component redundancies (§1).

3 An L*-Style Algorithm

We review the classic L* algorithm from [2]. We formulate it in such a way that it easily adapts to our componentwise and contextual algorithm in §4. Consequently, the algorithm we present (Alg. 1) differs slightly from the original L*; nevertheless, for simplicity, we call it L* throughout the paper.

We start with formulating the problem.

Problem 3.1 (Moore machine learning).

Input: the problem takes two alphabets I, O and two oracles OQ, EQ as inputs:

- 1) I and O are input and output alphabets, respectively; 2) the *output query* oracle OQ, given an input string $w \in I^*$, returns a string $\text{OQ}(w) \in O^*$; and
- 3) the *equivalence query* oracle EQ, given a (hypothesis) Moore machine H , returns “yes” or a *counterexample* sequence $w \in I^*$.

Output: a Moore machine M .

Here, OQ and EQ form an abstraction of a black-box SUL, and the goal is to learn M that behaves the same as the SUL.

Our L*-style algorithm (Alg. 1), henceforth simply called

L*, uses an *observation table* (S, R, E, T) . Here

- $S \subseteq I^*$ is a set of *prefixes*,
- $R \subseteq (S \cdot I) \setminus S$ is a set of *1-step extensions* of S ,
- $E \subseteq I^*$ is a set of *suffixes*, and
- $T: (S \cup R) \cdot E \rightarrow O$ is the *entry map*, where $(S \cup R) \cdot E$ collects all concatenations of strings from $S \cup R$ and those from E .

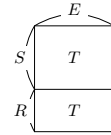


Fig. 3: an observation table

See Fig. 3. We initialize an observation table as $(\{\varepsilon\}, \emptyset, \{\varepsilon\}, (\varepsilon \mapsto \text{OQ}(\varepsilon)))$, and both S and E will always contain the empty string ε . We write $\text{row}(s)$ for the row of s in the observation table, i.e. $\text{row}(s): e \in E \mapsto T(s \cdot e)$. An observation table is *closed* if every $r \in R$ has some $s \in S$ such that $\text{row}(r) = \text{row}(s)$.

From a closed observation table (S, R, E, T) , we can construct a deterministic Moore machine $H = (Q, q_0, I, O, \delta, \lambda)$ with $Q = \{\text{row}(s) \mid s \in S\}$, $q_0 = \text{row}(\varepsilon)$,

Algorithm 1 an L^* -style Moore machine learning algorithm (simply called L^*)

```

1: procedure  $L^*(I, O, OQ, EQ)$ 
2:    $S \leftarrow \{\varepsilon\}$ ,  $R \leftarrow \emptyset$ ,  $E \leftarrow \{\varepsilon\}$ , and  $T(\varepsilon) \leftarrow OQ(\varepsilon)$ 
3:   repeat
4:     while  $(S, R, E, T)$  is not closed do
5:       Find  $s \cdot i \in R$  s.t.  $\text{row}(s \cdot i) \neq \text{row}(t)$  for all  $t \in S$ 
6:        $S \leftarrow S \cup \{s \cdot i\}$  and  $R \leftarrow R \setminus \{s \cdot i\}$ 
7:       Let  $H$  be the hypothesis Moore machine constructed from  $(S, R, E, T)$ 
8:        $D \leftarrow \{(s, i) \in 1\text{EXT}^{L^*}(H) \mid s \cdot i \notin S \cup R\}$  ▷  $1\text{EXT}^{L^*}$  is defined in the main text
9:       if  $D \neq \emptyset$  then
10:        for  $(s, i) \in D$  do
11:           $R \leftarrow R \cup \{s \cdot i\}$  and  $T(s \cdot i \cdot e) \leftarrow OQ(s \cdot i \cdot e)$  for each  $e \in E$ 
12:        continue
13:       if  $EQ(H) \neq \text{true}$  then
14:         Let  $w$  be a counterexample reported by  $EQ(H)$ 
15:          $\text{ANALYZE} \text{CEX}^{L^*}(H, w)$ 
16:   until  $EQ(H) = \text{true}$ 

17: procedure  $\text{ANALYZE} \text{CEX}^{L^*}(H, w)$ 
18:   Find the decomposition  $(s, i, d)$  of the counterexample  $w$  s.t.  $s \cdot i \cdot d = w$  and
    $OQ(t \cdot i \cdot d) \neq OQ(t' \cdot d)$  where  $\text{row}(t) = \delta_H(s)$  and  $\text{row}(t') = \delta_H(s \cdot i)$ 
19:    $E \leftarrow E \cup \{d\}$  and  $T(s \cdot d) \leftarrow OQ(s \cdot d)$  for each  $s \in S \cup R$ 

```

$\delta(\text{row}(s), i) = \text{row}(s \cdot i)$ for $s \cdot i \in R$, and $\lambda(\text{row}(s)) = T(s)$. This MM is called the *hypothesis Moore machine* from the observation table (S, R, E, T) .

The L^* algorithm Alg. 1 works by initializing an observation table, growing it using OQ till it is closed, making a hypothesis MM H , checking if H is good using EQ , and if H is not good, using the counterexample to further grow the table. When OQ and EQ are based on some MM M , Alg. 1 terminates and returns a MM equivalent to M .

The differences between Alg. 1 and the original L^* [2] are as follows.

1. L^* grows an observation table by 1) “extending input words” by picking $s \in S$ and $i \in I$ and adding $s \cdot i$ to R (Lines 8–12), and 2) “closing the table” by moving rows from R to S (Lines 4–6). We parameterize the first part with a function 1EXT^{L^*} . This parameter is set in the usual L^* manner in Alg. 1 (namely $1\text{EXT}^{L^*}(H) = \{(s, i) \mid \text{row}(s) \in Q_H \wedge i \in I\}$ where Q_H is the state space of H). Changing this parameter will be central in the next section.
2. In the original L^* [2], all the prefixes of the counterexample are added to S , but this can break a property called consistency. We, instead, add to E a suffix d that satisfies the condition in Line 18. This maintains consistency. This is a well-known technique; see e.g. [13].

An appropriate suffix d in Line 18 is effectively searched by the binary search [13, 25]; this leads to the following complexity bounds.

Theorem 3.2 (OQ and EQ complexities of L^* (Alg. 1)). *Assume that OQ and EQ are implemented using an MM M . Then Alg. 1 can correctly infer M with at most $O(\ell n^2 + n \log m)$ output queries and $O(n)$ equivalence queries, where n is the number of states of M , m is the maximal length of counterexamples, and ℓ is the input alphabet size.*

4 Our Contextual Componentwise Learning Algorithm

We introduce our main contribution, a *contextual componentwise L^* algorithm* CCwL*. Two baselines are *monolithic L^** (MnL*) and *componentwise L^** (CwL*).

Problem We formulate the problem. As discussed in §1, it is tailored to system integration applications where the learner has more access to the SUL.

Problem 4.1 (componentwise automata learning).

Input: the problem takes the following inputs: 1) a directed graph $G = (V, E)$; 2) alphabets $(\Sigma_e)_{e \in E}$; 3) system-level oracles OQ and EQ; 4) component-level oracles OQ_c and EQ_c for each $c \in V^c$

Output: a Moore machine M

Typically, all the oracles are implemented by a black-box MMN \mathcal{M} , and the goal is to learn an MM M that is equivalent to $[\mathcal{M}]$.

Two Baselines The monolithic L^* (MnL*) simply applies L^* (Alg. 1) to OQ and EQ, ignoring the network structure G and the component-level oracles. This way one has to learn a large MM, as discussed in §1.

The (naive) componentwise L^* (CwL*), in contrast, runs L^* (Alg. 1) with the component-level oracles OQ_c and EQ_c , and learns each component MM \mathcal{M}_c separately. Once it is done, it combines the learned MMs along the graph G , gets an MMN \mathcal{H} . This can exploit compositionality and decrease the states to learn; yet it may still suffer from *component redundancies* (the cost of learning parts of components that are not relevant system-level). See §1.

Our Algorithm CCwL* Our contextual algorithm CCwL* is shown in Alg. 2; it aims to alleviate component redundancies. We list its core features.

1. CCwL* learns components separately. This is much like CwL*. Therefore it keeps an observation table (S_c, R_c, E_c, T_c) for each component $c \in V^c$.
2. CCwL* only uses (component-level) OQ_c and (system-level) EQ. This is unlike CwL* that uses only component-level OQ_c and EQ_c .
3. Learning each component c is much like L^* (Alg. 1), but CCwL* uses different procedure/function there (namely, $\text{1EXT}_{\mathcal{E}, \mathcal{R}}$ and ANALYZECEx^C in Lines 9 and 15). Notably, these $\text{1EXT}_{\mathcal{E}, \mathcal{R}}$ and ANALYZECEx^C are *contextual*—they depend not only on the component c but also on the other components.

The oracle $\overline{\text{OQ}}: (\Sigma^{\text{in}})^* \rightarrow \overline{\Sigma}^*$ in Line 27 is for *total output queries*: it answers what strings are observed *at all edges* (including system-level output and inter-component edges), given an input string. The learner can compute it using component-level output query oracles $(\text{OQ}_c)_{c \in V^c}$ in a natural way.

On AnalyzeCex^C Overall, Alg. 2 mirrors the structure of Alg. 1, with differences only in $\text{1EXT}_{\mathcal{E}, \mathcal{R}}$ and ANALYZECEx^C (Lines 9 and 15)). To motivate the latter, recall that CCwL* uses only system-level EQs—using component-level EQs means we try to learn everything about a component and thus goes against our goal of eliminating component redundancies. Therefore the counterexamples obtained from EQs are system-level input strings \mathbf{w} . The procedure ANALYZECEx^C in Alg. 2 lets such \mathbf{w} generate a component-level input string

Algorithm 2 our contextual componentwise L* algorithm CCwL*

```

1: procedure CCwL* $((V, E), (\Sigma_e)_{e \in E}, \text{OQ}, \text{EQ}, (\text{OQ}_c)_{c \in V^c}, (\text{EQ}_c)_{c \in V^c})$ 
2:   for  $c \in V^c$  do
3:      $S_c \leftarrow \{\varepsilon\}$ ,  $R_c \leftarrow \emptyset$ ,  $E_c \leftarrow \{\varepsilon\}$ , and  $T_c(\varepsilon) \leftarrow \text{OQ}_c(\varepsilon)$ 
4:   repeat
5:     while  $(S_c, R_c, E_c, T_c)$  is not closed for some  $c \in V^c$  do
6:       Find  $\mathbf{s} \cdot \mathbf{i} \in R_c$  s.t.  $\text{row}_c(\mathbf{s} \cdot \mathbf{i}) \neq \text{row}_c(\mathbf{t})$  for all  $\mathbf{t} \in S_c$ 
7:        $S_c \leftarrow S_c \cup \{\mathbf{s} \cdot \mathbf{i}\}$  and  $R_c \leftarrow R_c \setminus \{\mathbf{s} \cdot \mathbf{i}\}$ 
8:       Let  $\mathcal{H}$  be the hypothesis MMN constructed from  $(S_c, R_c, E_c, T_c)_{c \in V^c}$ 
9:        $D \leftarrow \{(c, \mathbf{s}, \hat{\mathbf{i}}) \in \text{1EXT}_{\mathcal{E}, \mathcal{R}}(\mathcal{H}) \mid \mathbf{s} \cdot \hat{\mathbf{i}} \notin S_c \cup R_c\}$ 
10:      if  $D \neq \emptyset$  then
11:        for  $(c, \mathbf{s}, \hat{\mathbf{i}}) \in D$  do
12:           $R_c \leftarrow R_c \cup \{\mathbf{s} \cdot \hat{\mathbf{i}}\}$  and  $T_c(\mathbf{s} \cdot \hat{\mathbf{i}} \cdot \mathbf{e}) \leftarrow \text{OQ}_c(\mathbf{s} \cdot \hat{\mathbf{i}} \cdot \mathbf{e})$  for each  $\mathbf{e} \in E_c$ 
13:          continue
14:        if  $\text{EQ}(\mathcal{H}) \neq \text{true}$  then
15:          Let  $\mathbf{w}$  be a counterexample reported by  $\text{EQ}(\mathcal{H})$ 
16:           $\text{ANALYZECEx}^C(\mathcal{H}, \mathbf{w})$ 
17:      until  $\text{EQ}(\mathcal{H}) = \text{true}$ 

17: function  $\text{1EXT}_{\mathcal{E}, \mathcal{R}}(\mathcal{H})$ 
18:    $\tilde{\mathcal{H}} \leftarrow$  the quotient MMN  $\mathcal{H}/\mathcal{E}$  (Def. A.3) with respect to  $(\mathcal{E}(c))_{c \in V^c}$ 
19:    $D \leftarrow \emptyset$ 
20:   for  $\tilde{\mathbf{q}} = ([\text{row}(\mathbf{s}_c)]_{\mathcal{E}(c)})_{c \in V^c} \in \mathcal{R}(\tilde{\mathcal{H}})$  do
21:     for  $\mathbf{i} \in \Sigma^{\text{in}}$ ,  $c \in V^c$ ,  $\bar{\mathbf{o}} \in \bar{\lambda}(\tilde{\mathbf{q}})$ , and  $\text{row}(\mathbf{s}') \in [\text{row}(\mathbf{s}_c)]_{\mathcal{E}(c)}$  do
22:       Let  $\hat{\mathbf{i}}$  be a possible input character to  $c$  in  $\mathcal{H}$  on  $\mathbf{q}$  and  $\mathbf{i}$ , i.e.,  $\hat{\mathbf{i}} = (\mathbf{i}, \bar{\mathbf{o}})|_{E_c^{\text{in}}}$ 
23:        $D \leftarrow D \cup \{(c, \mathbf{s}', \hat{\mathbf{i}})\}$ 
24:   return  $D$ 

25: procedure  $\text{ANALYZECEx}^C(\mathcal{H}, \mathbf{w})$ 
26:    $\triangleright$  this ANALYZECExC is for sound  $(\mathcal{E}, \mathcal{R})$ ; otherwise ext. is needed (Appendix C.4)  $\triangleleft$ 
27:   Find a component  $c \in V^c$  that produces an incorrect output,
28:   that is,  $\text{OQ}(\mathbf{w})|_{E_c^{\text{out}}} \neq \llbracket \mathcal{H} \rrbracket(\mathbf{w})|_{E_c^{\text{out}}}$   $\triangleright$  the oracle  $\text{OQ}$  is described in the main text
29:   Construct an input  $\hat{\mathbf{w}}$  to the component  $c$  from the system-level input  $\mathbf{w}$ 
   where  $\hat{\mathbf{w}}_{[k]} = (\mathbf{w}_{[k]}, \text{OQ}(\mathbf{w})_{[k]})|_{E_c^{\text{in}}}$  for each  $k \in [0, |\mathbf{w}|)$ 
   Apply  $\text{ANALYZECEx}^{\text{L}^*}(H_c, \hat{\mathbf{w}})$   $\triangleright$  ANALYZECExL* is from Alg. 1

```

\mathbf{w}' for c in Line 27, and passes it to the analysis routine in Alg. 1 (namely $\text{ANALYZECEx}^{\text{L}^*}$).

On $\text{1EXT}_{\mathcal{E}, \mathcal{R}}$ On the other difference from L* ($\text{1EXT}_{\mathcal{E}, \mathcal{R}}$ in Line 9), we note that the function $\text{1EXT}_{\mathcal{E}, \mathcal{R}}$ has two parameters \mathcal{E} (called *component abstraction*) and \mathcal{R} (called *reachability analysis bound (RA bound)*). Combined, they are called *context analysis parameters (CA-parameters)*. We start with some intuitions.

Firstly, the goal of $\text{1EXT}_{\mathcal{E}, \mathcal{R}}$ is to find out *what input character \mathbf{i} a component c can receive, when c runs in the MMN \mathcal{M} and c 's current state is (represented by the prefix) \mathbf{s}'* . It adds all such tuples $(c, \mathbf{s}', \hat{\mathbf{i}})$ to D (Line 23).

In principle, it does so via the reachability analysis \mathcal{R} of the current hypothesis MMN \mathcal{H} . Specifically, $\text{1EXT}_{\mathcal{E}, \mathcal{R}}$ identifies all the combinations $\tilde{\mathbf{q}}$ of component states that \mathcal{H} can encounter (Line 20), collects all output characters $\bar{\mathbf{o}}$ given by such component states $\tilde{\mathbf{q}}$ (Line 21), and combines this $\bar{\mathbf{o}}$ with system-level input \mathbf{i} to find a possible input character $\hat{\mathbf{i}}$ to c (Line 22).

This baseline behavior of $1\text{EXT}_{\mathcal{E}, \mathcal{R}}$ is what happens with the most fine-grained CA-parameters ($\mathcal{E} = \text{Eq}, \mathcal{R} = \text{D}_\infty$). We present this special case in Appendix C.3 for illustration.

CA-Parameters \mathcal{E}, \mathcal{R} However, this full reachability analysis can be very expensive; the CA-parameters \mathcal{E}, \mathcal{R} are there to relieve it. The basic idea here is that we quotient hypothesis MMNs in order to ease reachability analysis. Those quotients naturally come with nondeterminism; thus we need the notions of non-deterministic MM and MMN (see Appendix A).

The *component abstraction* \mathcal{E} specifies how we quotient the components in the hypothesis MMN \mathcal{H} (Line 18). Note that it is used only within $1\text{EXT}_{\mathcal{E}, \mathcal{R}}$ (i.e. for context analysis); in particular, it is not directly used in observation tables.

- $\mathcal{E} = \text{Eq}$ (*equality*) means no quotienting.
- $\mathcal{E} = \text{Eq}_k$ (*k-equivalence*), with $k \in \mathbb{N}$, is $\text{Eq}_k = \{(s, t) \mid \lambda(s \cdot w) = \lambda(t \cdot w) \text{ for all strings } w \text{ with } |w| \leq k\}$. In particular, $\text{Eq}_0 = \{(s, t) \mid \lambda(s) = \lambda(t)\}$.
- $\mathcal{E} = \text{Uni}$ (*universal*) is given by $Q_c \times Q_c$ and collapses each component MM to a single state.

We define quotients of MMs (see Def. A.3) so that quotienting always leads to an *overapproximation* of output behaviors. Therefore, a possible input character $\hat{\mathbf{i}}$ (Line 23) is never missed. Such a choice of CA-parameters is said to be *sound*.

The *RA bound* \mathcal{R} specifies how complete our reachability analysis should be (for finding $\tilde{\mathbf{q}}$, Line 20). We do so by limiting the depth of breadth-first search.

- $\mathcal{R} = \text{D}_\infty$ means we set no bound and run full breadth-first search.
- $\mathcal{R} = \text{D}_d$ means we set the limit of depth $d \in \mathbb{N}$.

Here, unlike with \mathcal{E} , the use of $\mathcal{R} \neq \text{D}_\infty$ may lead to missing some $\tilde{\mathbf{q}}$ and thus some possible input $\hat{\mathbf{i}}$. Such a choice of CA-parameters is said to be *unsound*.

On AnalyzeCex^C, Again In case unsound CA-parameters are chosen (i.e. $\mathcal{R} \neq \text{D}_\infty$), a counterexample can arise not only in the usual L^* way (wrong output), but also by finding out that the hypothesis MM for a component c is not prepared for some input character $\hat{\mathbf{i}}$, missing a transition for $\hat{\mathbf{i}}$. Therefore ANALYZECex^C must be extended to handle such counterexamples. Doing so is not hard, and the extension is shown in Appendix C.4. The extension subsumes the one in Alg. 2; one can use the extension regardless of soundness of CA-parameters.

Query Complexities We state the following result.

Theorem 4.2 (OQ and EQ complexities of CCwL*). *Assume that \mathcal{E}, \mathcal{R} is sound (i.e. $\mathcal{R} = \text{D}_\infty$). The CCwL* algorithm (Alg. 2), assuming that all oracles are implemented using an MMN \mathcal{M} , can correctly infer \mathcal{M} with at most $O(\ell n^2 + n|V^c| \log m)$ component-level output queries and $O(n)$ system-level equivalence queries. Here n is the sum of the numbers of states of component Moore machines in \mathcal{M} , m is the maximal length of counterexamples, and ℓ is the system-level input alphabet size.*

If \mathcal{E}, \mathcal{R} is unsound, then the number of component-level OQs is bounded by $O(\ell n^2 + n|V^c| \log m + \ell n|V^c|)$, and that of system-level EQs is $O(n + \ell n)$.

Here is a proof sketch. The sound case adapts Thm. 3.2; the extra $|V^c|$ factor comes from the use of total output queries $\overline{OQ}(w)$ in ANALYZECEx^C . For the unsound case, EQs may also increase transitions (besides states, as in L^*); this increase the bound for EQs. The bound for OQs grows because calls of ANALYZECEx^C increase and OQs are used there.

5 Implementation and Experiments

The code of the implementations, as well as all experiment scripts, is available [9].

Implementation We implemented our proposal CCwL^* , together with two baselines MnL^* and CwL^* , in Scala. It takes an MMN as input, which is treated as a black-box teacher and used only for answering queries. Equivalence queries (EQs) are implemented through testing by randomly generated input words.

Benchmarks We used two families: *random* benchmarks where random components are arranged in a fixed network, and *realistic* benchmarks.

The random benchmarks $\text{Rand}(\text{nwk}, \text{comp})$ use the following parameters.

- The parameter nwk specifies the network topology. We use three families of network topologies: $\text{Comp1}(k)$ (a complete graph of k components), $\text{Star}(k)$ (a “frontend” component interconnected with k “backend” components), and $\text{Path}(k)$ (k components serially connected). See Fig. 4a.
- The parameter $\text{comp} \in \{\text{LeanComp}, \text{RichComp}\}$ specifies how each component is randomly generated. When $\text{comp} = \text{LeanComp}$, each component is a Moore machine whose number of states is chosen from the normal distribution $N(10, 1)$. For each (inter-component) edge, its alphabet size is picked from the uniform distribution over $\{2, 3, 4, 5\}$.

When $\text{comp} = \text{RichComp}$, we augment each component in such a way that roughly a half of it is redundant. Specifically, 1) each component is the interleaving product $M_c^\circ \times M_c^\bullet$ of two Moore machines generated in the above way (for LeanComp); 2) the two machines M_c°, M_c^\bullet have disjoint input and output alphabets; 3) therefore the alphabet for each edge in the MMN is bigger than for LeanComp ; 4) nevertheless, the system-level input alphabets as well as component output alphabets are chosen so that only the first machine M_c° is invoked. This way we force the redundancy of M_c^\bullet .

Our realistic benchmarks are MQTT_Lighting and $\text{BinaryCounter}(k)$. The latter models a k -bit counter; its details are in Appendix B.2. In what follows, we describe MQTT_Lighting in some detail (further details are in Appendix B.1).

The MMN MQTT_Lighting models a lighting system in which two sensors and one light communicate (Fig. 4b). Notably, it uses the *MQTT protocol* [22]—a protocol commonly used for IoT applications—and thus has a component called an (*MQTT*) *broker*. In this system, (1) the brightness sensor uses *QoS 1* of MQTT—meaning that, for each sensing data, four messages **Connect**, **ConnAck**, **Publish**, **PubAck** are exchanged; and (2) the motion sensor uses *QoS 2* of MQTT. It uses six messages: **Connect**, **ConnAck**, **Publish**, **PubRec**, **PubRel**, and **PubComp**. Different QoS levels provide different guarantees; see e.g. [22]. Our Moore ma-

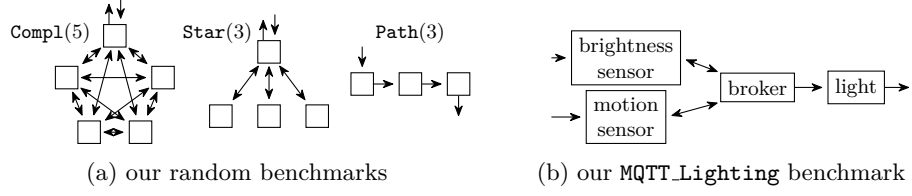


Fig. 4: network topologies for our random and MQTT_Lighting benchmarks

chine model for the broker,⁸ without knowing who uses which QoS, prepares for both QoS levels for each client. This redundancy (e.g. QoS 2 for brightness) is what we would like to eliminate via context analysis.

Experiment Settings We conducted experiments on AWS EC2 r7i.2xlarge instances, with 3.2 GHz Intel Xeon Scalable (Sapphire Rapids), 8 virtual cores, and 64GB RAM, with OpenJDK 23 (OpenJDK 64-Bit Server VM Temurin-23.0.2+7). Both the learner and an SUL were executed in the same machine. We set a timeout of 3600 seconds for the whole learning process; it returns once a system-level equivalence query succeeds.

After a successful return, we ran extra *validation* where, unlike equivalence queries during learning (these are by random-word testing), rigorous system-level equivalence verification is conducted between the learned system and the SUL. Note that this validation time is not included in the aforementioned timeout.

For random benchmarks $\text{Rand}(\text{nw}, \text{comp})$, we generated 10 instances for each parameter value (nw, comp) , and we report the average.

In evaluation, noting that the speed of SUL execution can vary greatly in different applications (cf. §1), we are not so interested in the total execution time as in the number of queries. Following [15], we report

- the number of *steps* (i.e. the number of input characters, but we use the word “step” since our input character can be a tuple $(a_e)_e$ in our setting), and
- the number of *resets* (resets can be much more costly than steps, see [15]).

We report these numbers separately for OQs and EQs. This is because the numbers for EQs depend heavily on how we choose to implement EQs, namely which method to use (random testing, conformance testing, black-box checking etc.), how many and how long words, etc.⁹

Results and Discussions We report the results in Tables 2 and 3. We discuss them along some research questions (RQs).

RQ1: Is the flexibility of CA-parameters useful? Which parameter to use?

⁸ Our broker model is adapted from Automata Wiki <https://automata.cs.ru.nl/BenchmarkMQTT-TapplerEtAl2017/Description>. It is originally from [27].

⁹ We can imagine application scenarios where we need a more refined view, separating the numbers of resets and steps for system-level queries and component-level ones. This is the case, for example, when a component-level interface is well-developed and fast (since a component is a commodity) but a system-level interface is slow (since it is a system under development). This data is not shown due to limited space.

Table 2: experiment results I. The rows are for different algorithms: two baselines (MnL*, CwL*) and our proposal (CCwL*) with different CA-parameters (§4). In the columns, *st.* is the number of learned states, *tr.* is that of learned transitions, *OQ reset* is the number of resets caused by output queries (it coincides with the number of OQs), *OQ step* is the number of steps caused by EQs, *EQ reset* is the number of resets caused by EQs, *EQ step* is the number of steps caused by EQs, *L. time* (“Learner time”) is the time (seconds) spent for the tasks on the learner’s side (context analysis, counterexample analysis, building observation tables, etc.), and *valid?* reports the numbers of instances of “result validated,” “result found incorrect,” and “timeout.” Note that we used 10 instances for each random benchmark.

algo.	Rand(Comp1(5),LeanComp)										Rand(Star(5),LeanComp)										Rand(Path(5),LeanComp)											
	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?
MnL*	27K	53K	591K	18M	15.0	129	32K	1.0K	0/1/9	26K	53K	917K	33M	23.5	417	105K	1.0K	0/2/8	5.5K	21K	496K	15M	19.7	223	54K	221	1/8/1					
CwL*	46.9	12K	15K	32K	6.20	501	137K	0.86	10/0/0	55.6	26K	26K	51K	17.6	613	165K	1.14	10/0/0	47.7	166	570	2.8K	17.9	513	137K	0.53	10/0/0					
CCwL*(Eq, D _∞)	46.9	9.0K	11K	35K	2.00	101	27K	30.8	10/0/0	54.8	18K	18K	46K	7.30	107	28K	188	10/0/0	46.8	140	457	2.5K	6.70	110	29K	6.43	10/0/0					
CCwL*(Eq, D ₀)	46.9	8.3K	16K	911K	5.4K	133K	33M	1.4K	0/10/0	54.3	7.0K	14K	1.2M	6.8K	231K	59M	1.6K	0/9/1	46.3	133	490	6.6K	64.3	203	41K	1.35	7/3/0					
CCwL*(Eq ₀ , D _∞)	46.9	9.0K	11K	35K	2.00	101	27K	30.4	10/0/0	54.8	18K	18K	47K	7.10	107	28K	24.0	10/0/0	47.3	151	501	2.8K	6.60	110	29K	2.01	10/0/0					
CCwL*(Eq ₀ , D ₀)	46.9	8.3K	16K	911K	5.4K	133K	33M	1.3K	0/10/0	54.3	7.0K	14K	1.2M	6.8K	231K	59M	1.6K	0/9/1	46.4	134	490	6.2K	58.0	191	39K	1.37	7/3/0					
CCwL*(Uni, D ₀)	46.9	9.0K	11K	29K	2.20	101	27K	93.0	10/0/0	54.8	19K	19K	47K	7.10	107	28K	32.5	10/0/0	47.3	152	506	2.7K	6.60	110	28K	2.06	10/0/0					
algo.	Rand(Comp1(5),RichComp)										Rand(Star(5),RichComp)										Rand(Path(5),RichComp)											
	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?
MnL*	—	—	—	—	—	—	—	—	0/0/0	41K	82K	1.0M	31M	22.0	272	67K	2.0K	0/1/9	6.8K	26K	732K	23M	22.1	301	75K	361	1/9/0					
CwL*	747	239K	1.3M	8.4M	23.6	519	138K	20.7	10/0/0	910	22K	70K	583K	40.1	637	167K	1.96	10/0/0	681	4.8K	39K	404K	36.4	536	139K	1.33	10/0/0					
CCwL*(Eq, D _∞)	47.3	9.4K	10K	31K	1.90	101	27K	23.4	10/0/0	56.7	11K	11K	29K	8.00	107	28K	190	10/0/0	44.4	138	462	2.6K	7.10	106	28K	6.76	8/2/0					
CCwL*(Eq, D ₀)	47.2	7.8K	14K	838K	5.0K	120K	30M	1.2K	0/9/1	56.7	6.0K	12K	991K	5.8K	187K	48M	1.3K	0/10/0	44.4	136	512	7.7K	68.4	216	44K	1.54	6/4/0					
CCwL*(Eq ₀ , D _∞)	47.3	9.4K	10K	31K	1.90	101	27K	22.4	10/0/0	56.7	11K	11K	29K	8.00	107	28K	16.1	10/0/0	44.6	145	482	2.7K	7.00	106	28K	1.89	8/2/0					
CCwL*(Eq ₀ , D ₀)	47.2	7.8K	14K	838K	5.0K	120K	30M	1.2K	0/9/1	56.7	6.0K	12K	991K	5.8K	187K	48M	1.4K	0/10/0	44.4	137	505	7.0K	59.2	202	43K	1.47	6/4/0					
CCwL*(Uni, D ₀)	47.3	9.4K	11K	26K	1.90	101	27K	74.3	10/0/0	56.7	11K	11K	28K	8.20	108	28K	20.5	10/0/0	45.0	147	488	2.7K	7.10	106	28K	1.99	9/1/0					
algo.	MQTT Lighting										BinaryCounter(5)										BinaryCounter(10)											
	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQ	EQ	L.	valid?
MnL*	169	1.5K	47K	1.4M	10.9	238	59K	17.6	0/10/0	70.0	140	212	5.9K	2.00	101	26K	0.76	10/0/0	—	—	—	—	—	—	—	—	—	—	—	—	—	0/0/10
CwL*	39.0	2.5K	12K	61K	11.8	412	105K	0.76	10/0/0	15.0	30.0	39.1	80.6	6.00	501	129K	0.41	10/0/0	30.0	60.0	74.1	141	11.0	1.0K	258K	0.72	10/0/0					
CCwL*(Eq, D _∞)	27.0	130	348	2.4K	4.40	112	28K	1.36	10/0/0	14.0	25.0	30.0	45.0	1.00	100	26K	0.88	10/0/0	29.0	50.0	60.0	90.0	1.00	100	26K	2.27	10/0/0					
CCwL*(Eq, D ₀)	27.0	129	412	7.4K	72.5	292	59K	1.94	10/0/0	14.0	25.0	44.4	357	15.4	115	26K	0.88	10/0/0	25.0	41.0	73.4	2.0K	23.4	128	28K	1.73	0/10/0					
CCwL*(Eq ₀ , D _∞)	27.0	134	362	2.8K	4.60	116	29K	1.21	10/0/0	14.0	25.0	30.0	45.0	1.00	100	26K	0.87	10/0/0	29.0	50.0	60.0	90.0	1.00	100	26K	2.27	10/0/0					
CCwL*(Eq ₀ , D ₀)	27.0	129	413	7.6K	71.5	311	64K	1.96	9/1/0	14.0	25.0	44.4	357	15.4	115	26K	0.87	10/0/0	25.0	41.0	73.4	2.0K	23.4	128	28K	1.78	0/10/0					
CCwL*(Uni, D ₀)	28.0	488	1.3K	7.2K	4.60	116	29K	2.54	10/0/0	14.0	28.0	33.0	54.0	1.00	100	26K	0.87	10/0/0	29.0	58.0	68.0	110	1.00	100	26K	9.73	10/0/0					

Table 3: experiment results II. The legend is the same as Table 2

algo.	Rand(Star(3),LeanComp)									Rand(Star(7),LeanComp)								
	st.	tr.	OQ	OQ	EQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQs	EQ	EQ	L.	valid?
			reset	step		reset	step	time				reset	step		reset	step	time	
MnL*	3.3K	13K	347K	6.8M	26.8	474	119K	89.1	1/9/0	—	—	—	—	—	—	—	—	0/0/10
CwL*	38.4	1.7K	2.6K	6.9K	12.9	409	110K	0.50	10/0/0	74.7	280K	280K	534K	26.6	819	220K	5.01	10/0/0
CCwL*(Eq, D _∞)	38.4	1.3K	2.0K	6.5K	6.80	106	28K	3.57	10/0/0	66.0	9.4K	10K	27K	9.00	108	28K	1.1K	1/0/9
CCwL*(Eq, D ₀)	38.4	1.1K	2.7K	160K	1.0K	16K	3.9M	79.8	2/8/0	66.0	6.4K	13K	1.0M	6.2K	200K	51M	1.9K	0/1/9
CCwL*(Eq ₀ , D _∞)	38.4	1.3K	2.1K	6.6K	6.90	106	28K	1.65	10/0/0	74.5	179K	179K	447K	10.0	109	28K	646	10/0/0
CCwL*(Eq ₀ , D ₀)	38.4	1.1K	2.7K	160K	1.0K	16K	3.9M	77.9	2/8/0	66.0	6.4K	13K	1.0M	6.2K	200K	51M	2.0K	0/1/9
CCwL*(Uni, D ₀)	38.4	1.4K	2.2K	6.8K	6.40	106	28K	1.98	10/0/0	74.5	183K	183K	415K	9.50	109	28K	693	10/0/0

algo.	Rand(Star(3),RichComp)									Rand(Star(7),RichComp)								
	st.	tr.	OQ	OQ	EQs	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQs	EQ	EQ	L.	valid?
			reset	step		reset	step	time				reset	step		reset	step	time	
MnL*	3.2K	12K	238K	4.1M	22.1	298	74K	56.0	1/9/0	—	—	—	—	—	—	—	—	0/0/10
CwL*	540	5.6K	37K	360K	27.4	430	112K	1.21	10/0/0	1.2K	471K	535K	1.6M	57.2	851	222K	9.01	10/0/0
CCwL*(Eq, D _∞)	37.4	1.3K	2.2K	7.0K	4.90	104	28K	2.76	10/0/0	73.5	100K	101K	279K	10.5	110	28K	2.3K	4/0/6
CCwL*(Eq, D ₀)	37.4	1.2K	3.0K	163K	1.1K	16K	4.0M	85.4	1/9/0	75.0	8.3K	17K	1.4M	8.1K	297K	76M	3.5K	0/1/9
CCwL*(Eq ₀ , D _∞)	37.4	1.3K	2.3K	7.5K	5.30	105	28K	1.54	10/0/0	74.1	204K	204K	520K	9.50	108	28K	616	10/0/0
CCwL*(Eq ₀ , D ₀)	37.4	1.2K	3.0K	163K	1.1K	16K	4.0M	83.4	1/9/0	75.0	8.3K	17K	1.4M	8.1K	297K	76M	3.0K	0/1/9
CCwL*(Uni, D ₀)	37.4	1.4K	2.4K	7.6K	5.00	104	28K	1.73	10/0/0	74.2	214K	215K	487K	9.80	109	28K	676	10/0/0

A natural theoretical expectation of benefit, and also the learner’s computational cost ($L. time$), is $Eq > Eq_0 > Uni$ (on \mathcal{E}) and $D_\infty > D_0$ (on \mathcal{R}). The experimental results confirm that this expectation is largely correct.

On benefit, indeed, finer-grained CA (e.g. (Eq, D_∞)) yielded smaller automata with fewer resets and steps. This is more notable in \mathcal{R} than in \mathcal{E} .

As an anomaly, (Uni, D₀) performed pretty well on $\text{Rand}(_, \text{RichComp})$. But it did not on MQTT_Lighting . This is natural: the redundancy in $\text{Rand}(_, \text{RichComp})$ is non-temporal (some input characters are never used) and even coarse-grained (Uni, D₀) could detect it; but the redundancy in MQTT_Lighting is temporal (what input characters are not used changes over time) and finding it was harder.

On the learner’s cost ($L. time$), the above expectation is not always correct: coarser CA often led to explosion of queries, which incurred the learner’s book-keeping cost. That said, the coarsest $\mathcal{E} = Uni$ did not suffer from this problem.

Overall, these observations suggest the following. There are different classes of SULs: in one class (e.g. MQTT_Lighting), component redundancies are temporal, and only fine-grained CA e.g. with (Eq, D_∞) can detect them; in another class (e.g. $\text{Rand}(_, \text{RichComp})$), redundancies are totally not temporal, and coarse-grained CA with e.g. (Uni, D₀) can detect them without much overhead. This will guide a choice of CA-parameters when the nature of an SUL is known (which class it belongs to?). When an SUL’s nature is unknown, one can try some intermediate CA-parameters; in Appendix C.5, we introduce three such ($\mathcal{R} = D_{\text{sum}}, D_{\text{max}}, D_{\text{min}}$) and evaluate them.

RQ2: How does CCwL*’s performance compare with that of CwL* or MnL*?

Henceforth, we follow the suggestion in RQ1 and focus on the CA-parameters CCwL*(Eq, D_∞) and CCwL*(Uni, D₀).

The advantages of CwL* and CCwL*—both are componentwise—over monolithic MnL* are observed in general. This is as expected (cf. §1).

In the comparison of CCwL* and (naive) CwL*, we observe that our goal (CA for eliminating component redundancies) is fulfilled: in the benchmarks with such redundancies (`Rand`(`_`, `RichComp`), `MQTT Lighting`, `BinaryCounter(k)`), CCwL* clearly outperformed CwL* in terms of automata size, resets and steps.

On the other benchmarks (namely `Rand`(`_`, `LeanComp`)), we still observe that 1) CCwL* and CwL* perform similarly, and 2) CCwL* can reduce the cost of EQs. The latter is because EQs in CCwL* are system-level, unlike component-level EQs in CwL*; a counterexample from the former can be reused for multiple components and suggest many new states.

RQ3: What is the cost of context analysis? Is it tolerable?

The additional cost for context analysis is part of *L. time*. This cost is on the learner’s side and can often be discounted (an SUL is usually slower and is more likely to be a bottleneck); still we want to confirm that the cost is tolerable.

Indeed, *L. time* is often much larger for CCwL* than for CwL*: in a large benchmark `Rand`(`Star(7)`, `LeanComp`), a few seconds for CwL* but hundreds of seconds for CCwL*. Whether this cost is tolerable depends on the cost model. For example, in embedded or HILS applications, taking 1 sec. for a reset and 10 ms. for a step is a norm. The gap of *L. time* then becomes ignorable.

RQ4: How does CCwL* scale to complex SULs? What SULs are suited?

CCwL* is designed to exploit redundancy of components. We have seen that, indeed, it performs well with benchmarks with redundancy.

The scalability question can be interpreted in two ways. One is *whether CCwL* can extract a small essence from a seemingly complex system*; then the answer is yes. For example, on `MQTT Lighting` and `Rand`(`Star(7)`, `LeanComp`), it learned much smaller automata than MnL* and CwL* did.

The other possible question is *whether CCwL* can extract an essence even if it is large* and our experiments do not allow us to answer yes. The largest MMN learned by CCwL* so far is of dozens of states, not more. The challenge here is the alphabet size—it grows exponentially with respect to the number of incoming edges—and the cost of CA that is impacted by it. As future work, we plan to work on deal with such large alphabets, e.g. by abstracting alphabets.

Summarizing the above discussions along RQ1–4, we conclude that 1) we are yet to investigate in-depth the practical scalability of our redundancy elimination methods, but 2) with the experimental results that show the efficiency of CCwL* for several benchmarks, the current work definitely opens promising avenues for future research. Regarding the first point, the main difficulty is that there are no existing benchmarks suited for our purpose, namely large real-world compositional systems whose network structures are known and formalized. We are currently mining IoT and robotics applications for such benchmarks.

6 Conclusions and Future Work

For compositional automata learning, we identified a new application domain of *system integration*, formalized its problem setting using *Moore machine networks*, and presented a novel *contextual componentwise learning* algorithm CCwL*.

It assumes that both system-level and component-level queries are available; to cope with the challenge of complexities due to redundancies in components (some parts do not contribute to the whole system), CCwL* performs *context analysis*. Our experimental evaluation shows its practical relevance.

One important future direction is to deal with large alphabets, as mentioned in §5. For example, in those applications where inter-component interactions are *sparse*—the characters transmitted are \perp (“do nothing”) most of the time—a theoretical framework that exploits this sparseness will be useful. We are also considering abstraction using *symbolic automata* [5].

References

1. Alur, R., Madhusudan, P.: Visibly pushdown languages. In: Babai, L. (ed.) Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004. pp. 202–211. ACM (2004). <https://doi.org/10.1145/1007352.1007390>
2. Angluin, D.: Learning regular sets from queries and counterexamples. *Inf. Comput.* **75**(2), 87–106 (1987). [https://doi.org/10.1016/0890-5401\(87\)90052-6](https://doi.org/10.1016/0890-5401(87)90052-6)
3. Argyros, G., D’Antoni, L.: The learnability of symbolic automata. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10981, pp. 427–445. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_23
4. Bainczyk, A., Schieweck, A., Steffen, B., Howar, F.: Model-based testing without models: The todomvc case study. In: Katoen, J., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10500, pp. 125–144. Springer (2017). https://doi.org/10.1007/978-3-319-68270-9_7
5. Drews, S., D’Antoni, L.: Learning symbolic automata. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I. Lecture Notes in Computer Science, vol. 10205, pp. 173–189 (2017). https://doi.org/10.1007/978-3-662-54577-5_10
6. al Duhaiby, O., Groote, J.F.: Active learning of decomposable systems. In: Bae, K., Bianculli, D., Gnesi, S., Plat, N. (eds.) FormaliSE@ICSE 2020: 8th International Conference on Formal Methods in Software Engineering, Seoul, Republic of Korea, July 13, 2020. pp. 1–10. ACM (2020). <https://doi.org/10.1145/3372020.3391560>
7. Fisman, D., Frenkel, H., Zilles, S.: Inferring symbolic automata. *Log. Methods Comput. Sci.* **19**(2) (2023). [https://doi.org/10.46298/LMCS-19\(2:5\)2023](https://doi.org/10.46298/LMCS-19(2:5)2023)
8. Frohme, M., Steffen, B.: Compositional learning of mutually recursive procedural systems. *Int. J. Softw. Tools Technol. Transf.* **23**(4), 521–543 (2021). <https://doi.org/10.1007/s10009-021-00634-y>
9. Fujinami, H., Waga, M., Hasuo, I.: Artifact archive for “componentwise automata learning for system integration” (Jul 2025). <https://doi.org/10.5281/zenodo.15846781>

10. van Heerdt, G., Kupke, C., Rot, J., Silva, A.: Learning weighted automata over principal ideal domains. In: Goubault-Larrecq, J., König, B. (eds.) Foundations of Software Science and Computation Structures - 23rd International Conference, FOSSACS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12077, pp. 602–621. Springer (2020). https://doi.org/10.1007/978-3-030-45231-5_31
11. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: A redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8734, pp. 307–322. Springer (2014). https://doi.org/10.1007/978-3-319-11164-3_26
12. Isberner, M., Howar, F., Steffen, B.: The open-source learnlib - A framework for active automata learning. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Lecture Notes in Computer Science, vol. 9206, pp. 487–495. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_32
13. Isberner, M., Steffen, B.: An abstract framework for counterexample analysis in active automata learning. In: Clark, A., Kanazawa, M., Yoshinaka, R. (eds.) Proceedings of the 12th International Conference on Grammatical Inference, ICGI 2014, Kyoto, Japan, September 17-19, 2014. JMLR Workshop and Conference Proceedings, vol. 34, pp. 79–93. JMLR.org (2014), <http://proceedings.mlr.press/v34/isberner14a.html>
14. Koenders, R., Moerman, J.: Output-decomposed learning of mealy machines. CoRR **abs/2405.08647** (2024). <https://doi.org/10.48550/ARXIV.2405.08647>, presented at LearnAut 2024
15. Labbaf, F., Groote, J.F., Hojjat, H., Mousavi, M.R.: Compositional learning for interleaving parallel automata. In: Kupferman, O., Sobocinski, P. (eds.) Foundations of Software Science and Computation Structures - 26th International Conference, FoSSaCS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13992, pp. 413–435. Springer (2023). https://doi.org/10.1007/978-3-031-30829-1_20
16. Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. Int. J. Softw. Tools Technol. Transf. **15**(5-6), 603–618 (2013). <https://doi.org/10.1007/S10009-012-0236-Z>
17. Malavolta, I., Nirghin, K., Scoccia, G.L., Romano, S., Lombardi, S., Scanniello, G., Lago, P.: Javascript dead code identification, elimination, and empirical assessment. IEEE Transactions on Software Engineering **49**(7), 3692–3714 (2023). <https://doi.org/10.1109/TSE.2023.3267848>
18. Maletti, A. (ed.): Algebraic Informatics - 6th International Conference, CAI 2015, Stuttgart, Germany, September 1-4, 2015. Proceedings, Lecture Notes in Computer Science, vol. 9270. Springer (2015). <https://doi.org/10.1007/978-3-319-23021-4>
19. Meijer, J., van de Pol, J.: Sound black-box checking in the learnlib. Innov. Syst. Softw. Eng. **15**(3-4), 267–287 (2019). <https://doi.org/10.1007/s11334-019-00342-6>
20. Neele, T., Sammartino, M.: Compositional automata learning of synchronous systems. In: Lambers, L., Uchitel, S. (eds.) Fundamental Approaches to Software Engi-

- neering - 26th International Conference, FASE 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings. Lecture Notes in Computer Science, vol. 13991, pp. 47–66. Springer (2023). https://doi.org/10.1007/978-3-031-30826-0_3
21. Niese, O.: An integrated approach to testing complex systems. Ph.D. thesis, Technical University of Dortmund, Germany (2003), http://eldorado.uni-dortmund.de:8080/0x81d98002_0x0007b62b
 22. OASIS: MQTT Version 5 (March 07 2019), <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf>
 23. Peled, D.A., Vardi, M.Y., Yannakakis, M.: Black box checking. In: Wu, J., Chanson, S.T., Gao, Q. (eds.) Formal Methods for Protocol Engineering and Distributed Systems, FORTE XII / PSTV XIX’99, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), October 5-8, 1999, Beijing, China. IFIP Conference Proceedings, vol. 156, pp. 225–240. Kluwer (1999)
 24. Pnueli, A., Rosner, R.: Distributed reactive systems are hard to synthesize. In: 31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II. pp. 746–757. IEEE Computer Society (1990). <https://doi.org/10.1109/FSCS.1990.89597>
 25. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. **103**(2), 299–347 (1993). <https://doi.org/10.1006/INCO.1993.1021>
 26. Shijubo, J., Waga, M., Suenaga, K.: Probabilistic black-box checking via active MDP learning. ACM Trans. Embed. Comput. Syst. **22**(5s), 148:1–148:26 (2023). <https://doi.org/10.1145/3609127>
 27. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing iot communication via active automata learning. In: 2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017. pp. 276–287. IEEE Computer Society (2017). <https://doi.org/10.1109/ICST.2017.32>
 28. Vaandrager, F.W., Garhewal, B., Rot, J., Wißmann, T.: A new approach for active automata learning based on apartness. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13243, pp. 223–243. Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_12
 29. Zhang, H., Feng, L., Li, Z.: Control of black-box embedded systems by integrating automaton learning and supervisory control theory of discrete-event systems. IEEE Trans Autom. Sci. Eng. **17**(1), 361–374 (2020). <https://doi.org/10.1109/TASE.2019.2929563>

A Nondeterministic Moore Machines and MMNs

A.1 Nondeterministic MMs

Definition A.1 (Moore machine). A (nondeterministic) *Moore machine* (*MM*) is a tuple $M = (Q, Q_0, I, O, \Delta, \lambda)$, where

- Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states,

- I is an input alphabet, O is an output alphabet,
- $\Delta: Q \times I \rightarrow 2^Q$ is a transition function, and
- $\lambda: Q \rightarrow (2^O \setminus \{\emptyset\})$ is an output function that assigns a nonempty set of output symbols to each state.

A Moore machine is *deterministic* if $|\Delta(q, i)| \leq 1$ for all $q \in Q$ and $i \in I$, $|Q_0| = 1$, and $|\lambda(q)| = 1$ for all $q \in Q$. For a deterministic Moore machine M , the transition function in M is denoted as a partial function by $\delta: Q \times I \rightarrow Q$ and the initial state is $q_0 \in Q$. A deterministic Moore machine is *complete* if $\delta(q, i) \downarrow$ for all $q \in Q$ and $i \in I$; otherwise, it is called *partial*.

The following definitions are standard, but embracing nondeterminism has incurred some notational complications. The reader can first skim through them and come back later for checking details.

As usual, the transition function Δ can be extended to a set $P \subseteq Q$ of states and an input string $w \in I^*$. Precisely, $\Delta(P, \varepsilon) = P$ and $\Delta(P, wi) = \bigcup_{q \in \Delta(P, w)} \Delta(q, i)$. Similarly, the output function is extended by $\lambda(P, w) = \bigcup_{q \in \Delta(P, w)} \lambda(q)$. When starting from the set Q_0 of initial states, often we simply write $\Delta(w) = \Delta(Q_0, w)$ and $\lambda(w) = \lambda(Q_0, w)$.

Given a Moore machine $M = (Q, Q_0, I, O, \Delta, \lambda)$ and a set $P \subseteq Q$ of states, the *semantics* of M , denoted by $\llbracket M \rrbracket_P: I^* \rightarrow (2^O)^*$ and defined below, represents the behavior of the machine when starting from P :

$$\llbracket M \rrbracket_P(w) = \lambda(P, w_{[0,0)}) \lambda(P, w_{[0,1)}) \cdots \lambda(P, w_{[0,|w|)}) \quad \text{for each } w \in I^*. \quad (2)$$

This is the nondeterministic adaptation of the usual definition. Note that $|\llbracket M \rrbracket_P(w)| = |w| + 1$, as $\llbracket M \rrbracket_P(w)_{[0]}$ is the output for P without consuming any input characters. If $\llbracket M \rrbracket_P(w)_{[k+1]} = \emptyset$ for some $0 \leq k < |w|$, then $\llbracket M \rrbracket_P(w)_{[j]} = \emptyset$ for all $k < j \leq |w|$; this intuitively means, according to Def. 2.1, that 1) a Moore machine gets stuck only because of Δ (λ is nonempty), and 2) once stuck, it remains stuck. When k is the largest number with $\llbracket M \rrbracket_P(w)_{[k]} \neq \emptyset$, we say that $\llbracket M \rrbracket_P$ is *defined* on w up to k . When starting from the set Q_0 of initial states, we write $\llbracket M \rrbracket(w)$ for $\llbracket M \rrbracket_{Q_0}(w)$ (much like for Δ and λ).

Using this semantics, we define the equivalence of Moore machines.

Definition A.2 (equivalence of Moore machines). Two Moore machines M_1 and M_2 are said to be *equivalent* if and only if $\llbracket M_1 \rrbracket = \llbracket M_2 \rrbracket$.

For a deterministic Moore machine M and a state $q \in Q$, we adopt the convention that its semantics is a *total* function $\llbracket M \rrbracket_q: I^* \rightarrow O^*$. This can be done by 1) specializing Eq. (2) to deterministic M , and 2) saying that, if $\lambda(P, w_{[0,k)}) = \emptyset$, it is interpreted as ε rather than the empty set. (That is, while $\lambda(P, w_{[0,k)}) = \emptyset$ does not make the right-hand side longer, it does not make it undefined.)

The following construction, used in §4, is our central reason for accommodating nondeterminism. In the usual theory of *deterministic* automata, a quotient is taken only wrt. a suffix-closed equivalence, and this ensures that the *deterministic* quotient is well-defined. In contrast, we would like flexible quotient

automata (wrt. any equivalence) in §4; we can make them well-defined thanks to nondeterminism.

Definition A.3 (quotient Moore machine M/\sim). Let \sim be an equivalence relation on Q . The *quotient Moore machine* M/\sim of M with respect to \sim is a Moore machine $(Q/\sim, Q_0/\sim, I, O, \Delta', \lambda')$ with $\Delta'([q]_\sim, i) = (\bigcup_{q' \in [q]_\sim} \Delta(q', i))/\sim$ and $\lambda'([q]_\sim) = \bigcup_{q' \in [q]_\sim} \lambda(q')$.

For a Moore machine M and states $q, q' \in Q$, a state q is said to be *reachable* from q' in M if there exists a string $w \in I^*$ such that $q \in \Delta(q', w)$. When q' is one of the initial state Q_0 , we simply say that q is reachable in M .

A.2 Nondeterministic MMNs

Definition A.4 (Moore machine network). A (nondeterministic) *Moore machine network* (MMN) is a tuple $\mathcal{M} = (G, (\Sigma_e)_{e \in E}, (M_c)_{c \in V^c})$, where

- $G = (V, E)$ is a directed graph representing the network structure,
- Σ_e is an alphabet associated with each edge $e \in E$, and
- M_c is a Moore machine associated with each component $c \in V^c$.

On each component Moore machine $M_c = (Q_c, Q_{0,c}, \Sigma_c^{\text{in}}, \Sigma_c^{\text{out}}, \Delta_c, \lambda_c)$, we require that its input and output alphabets are in accordance with the edge alphabets Σ_e . Specifically, we require $\Sigma_c^{\text{in}} = \prod_{e \in E_c^{\text{in}}} \Sigma_e$ (the product of the alphabets of all incoming edges) and, similarly, $\Sigma_c^{\text{out}} = \prod_{e \in E_c^{\text{out}}} \Sigma_e$.

The definition of the semantics of MMNs is adapted as follows, to the current nondeterministic setting. This adaptation is easy.

The set of (*system-level*) *configurations* of \mathcal{M} , denoted by \mathbf{Q} , is defined by $\mathbf{Q} = \prod_{c \in V^c} Q_c$. The set of *initial configurations* is $\mathbf{Q}_0 = \prod_{c \in V^c} Q_{0,c}$.

Given a configuration $\mathbf{q} = (q_c)_{c \in V^c} \in \mathbf{Q}$, the *total output* of \mathcal{M} at \mathbf{q} , denoted by $\bar{\lambda}(\mathbf{q}) \subseteq \bar{\Sigma}$, is defined by $\bar{\lambda}(\mathbf{q}) = \prod_{c \in V^c} \lambda_c(q_c)$. Similarly, the *system-level output* of \mathcal{M} at \mathbf{q} is defined by $\lambda(\mathbf{q}) = \bar{\lambda}(\mathbf{q})|_{E^{\text{out}}} \subseteq \Sigma^{\text{out}}$. (Recall the restriction operation $|$ from §2.)

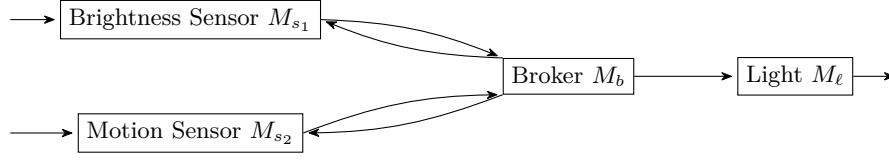
Given a configuration $\mathbf{q} = (q_c)_{c \in V^c} \in \mathbf{Q}$ and a system-level input character $\mathbf{i} \in \Sigma^{\text{in}}$, we define Δ , the *system-level transition function* of \mathcal{M} , by $\Delta(\mathbf{q}, \mathbf{i}) = \prod_{c \in V^c} \bigcup_{\bar{o} \in \bar{\lambda}(\mathbf{q})} \Delta_c(q_c, (\mathbf{i}, \bar{o})|_{E_c^{\text{in}}})$. Intuitively: \bar{o} is a tuple of characters that can be output from the current states $(q_c)_{c \in V^c}$; it is combined with the system-level input \mathbf{i} and fed to each component's transition function Δ_c ; we take the union \bigcup over all possible output \bar{o} ; and this happens at every component (\prod_c).

We formalize the following definition, using the above constructions.

Definition A.5 (Moore machine $[\mathcal{M}]$). Let \mathcal{M} be an MMN. The *Moore machine* $[\mathcal{M}]$ induced by \mathcal{M} is $[\mathcal{M}] = (\mathbf{Q}, \mathbf{Q}_0, \Sigma^{\text{in}}, \Sigma^{\text{out}}, \Delta, \lambda)$.

The semantics of \mathcal{M} is defined by $\llbracket \mathcal{M} \rrbracket_{\mathbf{P}} = \llbracket [\mathcal{M}] \rrbracket_{\mathbf{P}}$ for any $\mathbf{P} \subseteq \mathbf{Q}$.

An MMN \mathcal{M} is *deterministic* if every component Moore machine M_c is deterministic. In this case, obviously, the Moore machine $[\mathcal{M}]$ is also deterministic.

Fig. 5: outline of the MMN $\mathcal{M}_{\text{light}}$ in MQTT_Lighting.

Given an MMN $\mathcal{M} = (G, (\Sigma_e)_{e \in E}, (M_c)_{c \in V^c})$ and an indexed family of equivalence relations $(\sim_c)_{c \in V^c}$ such that $\sim_c \subseteq Q_c \times Q_c$, the *quotient MMN* \mathcal{M}/\sim of \mathcal{M} with respect to $(\sim_c)_{c \in V^c}$ is the MMN $(G, (\Sigma_e)_{e \in E}, (M_c/\sim_c)_{c \in V^c})$. Here M_c/\sim_c is defined in Def. A.3.

B Details of the benchmarks

B.1 The Benchmark MQTT_Lighting

MQTT_Lighting is our original benchmark modeling a reactive lighting system. Fig. 5 outlines the MMN $\mathcal{M}_{\text{light}}$ in MQTT_Lighting. The MMN $\mathcal{M}_{\text{light}}$ consists of the following four components in addition to the input and output nodes: the brightness sensor component s_1 , the motion sensor component s_2 , the broker component b , and the light component ℓ . The sensor components (s_1 and s_2) observe the current status of the environment and publish it to the broker component. Specifically, the brightness sensor component observes if the environment is bright or not, and the motion sensor component observes if any motion is detected. The broker component receives messages from the sensor components and forwards them to the light component. The light component receives messages from the broker component and controls the light based on the current environment's status. We use a communication protocol inspired by MQTT [22] for inter-component communication. Our encoding is inspired by the Mealy machines that model the MQTT protocol on Automata Wiki¹⁰, which is originally from [27].

Brightness sensor component Fig. 6 illustrates the Moore machine M_{s_1} for the brightness sensor component. The brightness sensor observes whether the environment is bright or dark (i.e., $\Sigma_{\text{in}, s_1} = \{\text{bright}, \text{dark}\}$) and publishes it to the broker. The messages are sent via a protocol based on MQTT QoS 1: i) The publisher (i.e., the brightness sensor component) tries to establish a connection by sending **Connect** to the broker; ii) The broker returns **ConnAck** when the connection is established; iii) Then, the publisher publishes the current status by sending either **PubQoS1(bright)** or **PubQoS1(dark)** to the broker; iv) The broker returns **PubAck** to make sure that the publication was successful; v) Finally, the publisher sends **Disconnect** to close the connection.

¹⁰ Automata Wiki: <https://automata.cs.ru.nl/BenchmarkMQTT-TapplerEtAl2017/Description>

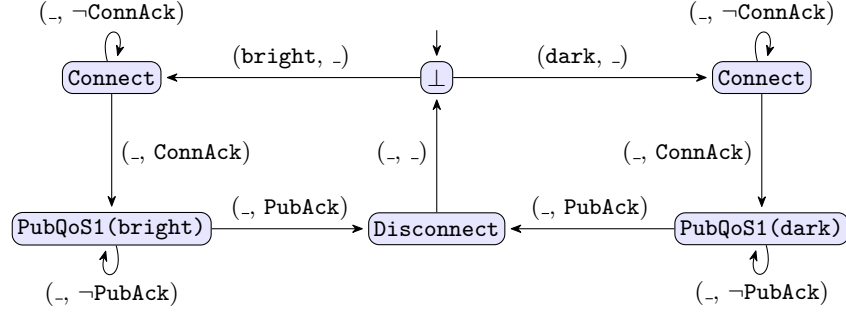


Fig.6: the Moore machine M_{s_1} for the brightness sensor component in MQTT_Lighting. Each transition is labeled with a pair (i_{in,s_1}, i_{b,s_1}) of symbols representing the subset of $\Sigma_{in,s_1} \times \Sigma_{b,s_1}$: the symbol $-$ represents any character, and for any character i , $\neg i$ represents the character other than i . For instance, $(-, \neg \text{ConnAck})$ represents $\{(i_{in,s_1}, i_{b,s_1}) \in \Sigma_{in,s_1} \times \Sigma_{b,s_1} \mid i_{b,s_1} \neq \text{ConnAck}\}$

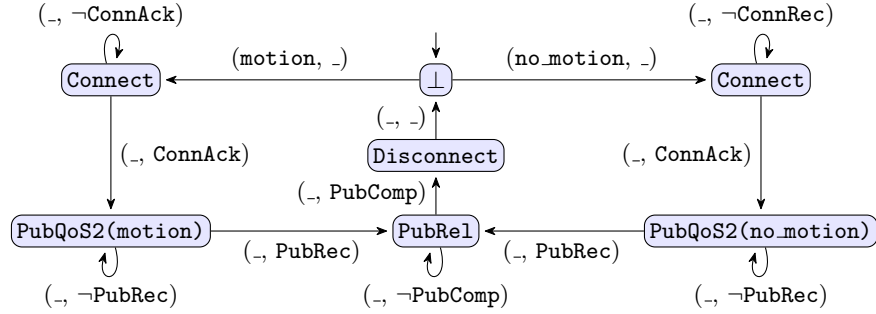


Fig.7: the Moore machine M_{s_2} for the motion sensor component in MQTT_Lighting. We use the same notation as in Fig. 7

In the above illustration, the brightness sensor component sends the following messages: **Connect**, **PubQoS1(bright)**, **PubQoS1(dark)**, and **Disconnect**. We assume that the broker does not know the QoS used by the publisher in advance and we include the messages for other QoS in the alphabet. Moreover, we allow the publisher to not send any message. In total, the alphabet $\Sigma_{s_1,b}$ from s_1 to b is as follows: $\Sigma_{s_1,b} = \{\text{Connect}, \text{PubQoS0(bright)}, \text{PubQoS0(dark)}, \text{PubQoS1(bright)}, \text{PubQoS1(dark)}, \text{PubQoS2(bright)}, \text{PubQoS2(dark)}, \text{PubRel}, \text{Disconnect}, \perp\}$.

In the above illustration, the broker component sends **ConnAck** or **PubAck** to the brightness sensor component. Including the messages not used in QoS 1 and the special character \perp that shows that no message is sent, the alphabet Σ_{b,s_1} from b to s_1 is as follows: $\Sigma_{b,s_1} = \{\text{ConnAck}, \text{PubAck}, \text{PubRec}, \text{PubComp}, \perp\}$.

Motion sensor component Fig. 7 illustrates the Moore machine M_{s_2} for the motion sensor component. The motion sensor observes whether or not a motion

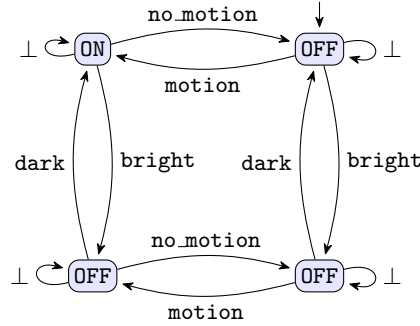


Fig. 8: the Moore machine M_ℓ for the light component in `MQTT_Lighting`. We use the same notation as in Fig. 7

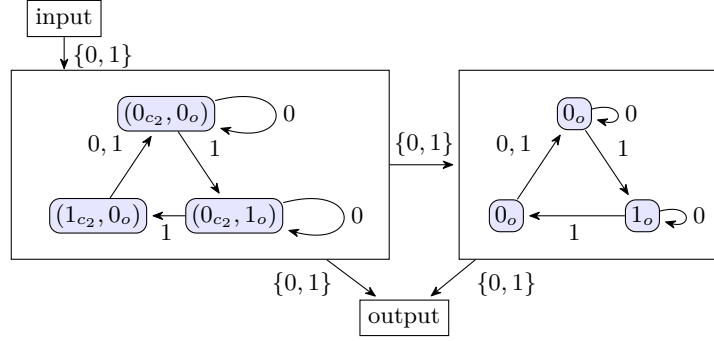
is detected (i.e., $\Sigma_{in,s_2} = \{\text{motion}, \text{no_motion}\}$) and publishes it to the broker. The messages are sent via a protocol based on MQTT QoS 2: i) The publisher (i.e., the motion sensor component) tries to establish a connection by sending `Connect` to the broker; ii) The broker returns `ConnAck` when the connection is established; iii) Then, the publisher publishes the current status by sending either `PubQoS2(bright)` or `PubQoS2(dark)` to the broker; iv) The broker returns `PubRec` to make sure that the publication was successful; v) Then, the publisher sends `PubRel` to show that `PubRec` was successfully received; vi) The broker returns `PubComp` to complete the publication; vii) Finally, the publisher sends `Disconnect` to close the connection.

In the above illustration, the motion sensor component sends the following messages: `Connect`, `PubQoS2(motion)`, `PubQoS2(no_motion)`, `PubRel`, and `Disconnect`, and the broker component sends `ConnAck`, `PubRec`, and `PubComp`. Similarly to $\Sigma_{s_1,b}$ and Σ_{b,s_1} , $\Sigma_{s_2,b}$ and Σ_{b,s_2} are as follows:

$$\begin{aligned} \Sigma_{s_2,b} &= \{\text{Connect}, \text{PubQoS0}(\text{motion}), \text{PubQoS0}(\text{no_motion}), \text{PubQoS1}(\text{motion}), \\ &\quad \text{PubQoS1}(\text{no_motion}), \text{PubQoS2}(\text{motion}), \text{PubQoS2}(\text{no_motion}), \text{PubRel}, \\ &\quad \text{Disconnect}, \perp\} \\ \Sigma_{b,s_2} &= \{\text{ConnAck}, \text{PubAck}, \text{PubRec}, \text{PubComp}, \perp\} \end{aligned}$$

Light component Fig. 8 illustrates the Moore machine M_ℓ for the light component. The light component receives the current environment's status (i.e., the brightness and the detection of the motion) and controls the light. Thus, the input alphabet is $\Sigma_{b,\ell} = \{\text{bright}, \text{dark}, \text{motion}, \text{no_motion}\}$ and the output alphabet is $\Sigma_{\ell,\text{out}} = \{\text{ON}, \text{OFF}\}$. We assume that initially, the environment is dark, and no motion is detected.

Broker component The broker component b manages the aforementioned communication, mainly the data transmission from the sensor components to the

Fig. 9: the MMN of `BinaryCounter(2)`

light component. We assume that once a connection is established, the broker component only reads the messages from the connected component until the connection is explicitly closed by the publisher. For simplicity, we assume that we statically have two publishers (the sensor components s_1 and s_2) and one subscriber (the light component ℓ). For the fairness of the broker, if both publishers try to establish the connection at the same time, the broker establishes the connection with the publisher that was not the most recently communicated with.

B.2 The Benchmark `BinaryCounter(k)`

The MMN for our realistic benchmark `BinaryCounter(k)` is shown in Fig. 9. Note that the MMN shown in Fig. 9 is for `BinaryCounter(2)`, but the generalization to `BinaryCounter(k)` is straightforward. The MMN counts the number of 1s in an input string and outputs a binary representation of it. Due to the delay in the output of Moore machines, k -times 0s between each 1 are required for correctly counting. It consists of k Moore machines, with each component in charge of each of the k -bit output, and inter-component edges address carry out. The Moore machine for each component consists of 3 states: two states $(0, 0)$ and $(0, 1)$ to hold the current output value and $(1, 0)$ to pass the carry-in bit to the next component. Please note that the last component has no output to the next component.

C Omitted Proofs and Remarks

C.1 Moore Machines over Mealy Machines

Remark C.1. Our choice of Moore machines as models of components is crucial for the above operational semantics of MMNs. In particular, we find that Mealy machines are not suited for our purpose.

An example of a “Mealy machine network” is in Fig. 10. Assume that at some tick, M_{c_1} is at its top state and receives d from M_{c_2} . Assuming that it

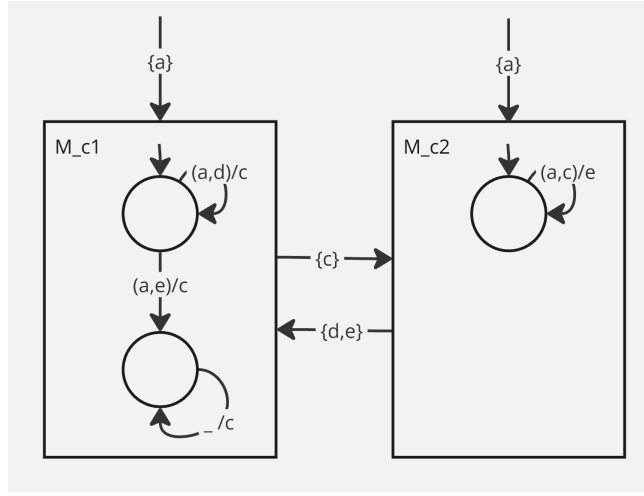


Fig. 10: “Mealy machine network” is problematic

also receives the system-level input a , the loop transition $(a, d)/c$ is fired and an output c is sent to M_{c_2} . This fires the transition $(a, c)/e$ in M_{c_2} , leading to a character e sent to M_{c_1} , which then fires the transition $(a, e)/c$ to the bottom state of M_{c_1} .

The problem is that all these should happen within a single tick—so M_{c_1} should make two transitions (the loop and the downward) at the same time. This is strange.

The key difference between Moore and Mealy machines is that, in the former, the effect of input is reflected on output with a one-tick delay. Indeed, with Moore machines, the above “chain of immediate consequences” never arises.

C.2 Details of Ex. 2.4

Example C.2. Consider the MMN \mathcal{M}_{ex} in Fig. 2, the network structure $G = (V, E)$ is

$$V = \{i_1, i_2, c_1, c_2, o_1, o_2\} \text{ and } E = \{i_1 c_1, i_2 c_2, c_1 c_2, c_2 c_1, c_1 o_1, c_2 o_2\}.$$

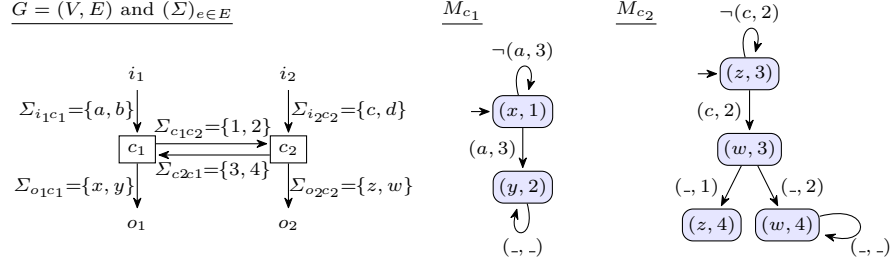


Fig. 11: an example MMN \mathcal{M}_{ex} . On the left we show its network $G = (V, E)$ and the alphabets $(\Sigma_e)_{e \in E}$ for edges. The component MMs are shown on the right, where state labels designate output. In the transition labels, $\neg i$ stands for all characters other than i , and the symbol $-$ matches any character

Algorithm 3 the function $1\text{EXT}_{\text{Eq}, D_\infty}$, a special case of $1\text{EXT}_{\mathcal{E}, \mathcal{R}}$, presented for illustration

```

1: function  $1\text{EXT}_{\text{Eq}, D_\infty}(\mathcal{H})$ 
2:    $D \leftarrow \emptyset$ 
3:   for all reachable  $\mathbf{q} = (\text{row}(\mathbf{s}_c))_{c \in V^c} \in \mathbf{Q}$  in  $\mathcal{H}$  do
4:     for  $\mathbf{i} \in \Sigma^{\text{in}}$  and  $c \in V^c$  do
5:       Let  $\hat{\mathbf{i}}$  be a possible input character to  $c$  in  $\mathcal{H}$  on  $\mathbf{q}$  and  $\mathbf{i}$ , i.e.,  $\hat{\mathbf{i}} = (\mathbf{i}, \bar{\lambda}(\mathbf{q}))|_{E_c^{\text{in}}}$ 
6:        $D \leftarrow D \cup \{(c, \mathbf{s}_c, \hat{\mathbf{i}})\}$ 
7:   return  $D$ 

```

Furthermore, the set of nodes V is partitioned by $V^{\text{in}} = \{i_1, i_2\}$, $V^{\text{out}} = \{o_1, o_2\}$, and $V^c = \{c_1, c_2\}$. The alphabets of the system and components are:

$$\begin{aligned}
\Sigma^{\text{in}} &= \Sigma_{i_1 c_1} \times \Sigma_{i_2 c_2} = \{(a, c), (a, d), (b, c), (b, d)\} \\
\Sigma^{\text{out}} &= \Sigma_{c_1 o_1} \times \Sigma_{c_2 o_2} = \{(x, z), (x, w), (y, z), (y, w)\} \\
\Sigma_{c_1}^{\text{in}} &= \Sigma_{i_1 c_1} \times \Sigma_{c_2 c_1} = \{(a, 3), (a, 4), (b, 3), (b, 4)\} \\
\Sigma_{c_1}^{\text{out}} &= \Sigma_{c_1 o_1} \times \Sigma_{c_1 c_2} = \{(x, 1), (x, 2), (y, 1), (y, 2)\} \\
\Sigma_{c_2}^{\text{in}} &= \Sigma_{i_2 c_2} \times \Sigma_{c_1 c_2} = \{(c, 1), (c, 2), (d, 1), (d, 2)\} \\
\Sigma_{c_2}^{\text{out}} &= \Sigma_{c_2 o_2} \times \Sigma_{c_2 c_1} = \{(z, 3), (z, 4), (w, 3), (w, 4)\} \\
\bar{\Sigma} &= \Sigma_{c_1}^{\text{out}} \times \Sigma_{c_2}^{\text{out}} = \{(x, 1, z, 3), (x, 1, z, 4), (x, 1, w, 3), \dots, (y, 2, w, 4)\}
\end{aligned}$$

C.3 The Function $1\text{EXT}_{\text{Eq}, D_\infty}$, for Illustration

The special case $1\text{EXT}_{\text{Eq}, D_\infty}$ of $1\text{EXT}_{\mathcal{E}, \mathcal{R}}$ is shown in Alg. 3.

C.4 The Extended Procedure AnalyzeCex^C That Accommodates Unsound $(\mathcal{E}, \mathcal{R})$

The procedure is in Alg. 4.

Algorithm 4 an extension of ANALYZECEX^C in Alg. 2 for accommodating unsound $(\mathcal{E}, \mathcal{R})$

```

1: procedure  $\text{ANALYZECEX}^C(\mathcal{H}, \mathbf{w})$ 
2:   if  $\delta(\mathbf{w}_{[0,k]}) \downarrow$  and  $\delta(\mathbf{w}_{[0,k+1]}) \uparrow$  for some  $0 \leq k < |\mathbf{w}|$  then
3:     Let  $\delta(\mathbf{w}_{[0,k]})$  be  $\mathbf{q} = (\mathbf{s}_c)_{c \in V^c}$  and  $\mathbf{i}$  be  $w_{[k]}$ .
4:     Find a component  $c \in V^c$  and an input character  $\hat{\mathbf{i}}$ 
       in which a transition is missing, that is,  $\delta_c(\mathbf{s}_c, \hat{\mathbf{i}}) \uparrow$  and  $\hat{\mathbf{i}} = (\mathbf{i}, \bar{\lambda}(\mathbf{q}))|_{E_c^{\text{in}}}$ 
5:      $R_c \leftarrow R_c \cup \{(\mathbf{s}_c, \hat{\mathbf{i}})\}$ 
6:      $T(\mathbf{s}_c \cdot \hat{\mathbf{i}} \cdot \mathbf{e}) \leftarrow \text{OQ}_c(\mathbf{s}_c \cdot \hat{\mathbf{i}} \cdot \mathbf{e})$  for each  $\mathbf{e} \in E_c$ 
7:   return
8:   Find a component  $c \in V^c$  that produces an incorrect output,
       that is,  $\text{OQ}(\mathbf{w})|_{E_c^{\text{out}}} \neq \llbracket \mathcal{H} \rrbracket(\mathbf{w})|_{E_c^{\text{out}}}$ 
9:   Construct an input  $\hat{\mathbf{w}}$  to the component  $c$  from the system-level input  $w$ 
       where  $\hat{\mathbf{w}}_{[k]} = (\mathbf{w}_{[k]}, \text{OQ}(\mathbf{w})|_{[k]}|_{E_c^{\text{in}}})$  for  $0 \leq k < |\mathbf{w}|$ 
10:  Apply  $\text{L}^*\text{-ANALYZECEX}(H_c, \hat{\mathbf{w}})$ 

```

C.5 Full Experimental Results with Intermediate CA-parameters

Full experimental results, with additional *intermediate* CA-parameters, are in Tables 4 and 5.

They extend Tables 2 and 3 by additional values $\mathcal{R} = \text{D}_{\text{sum}}, \text{D}_{\text{max}}, \text{D}_{\text{min}}$ for \mathcal{R} . Here $\mathcal{R} = \text{D}_{\text{sum}}$ means that we limit the depth of breadth-first search by the sum of the learned components' numbers of states (cf. §4). Similarly, $\mathcal{R} = \text{D}_{\text{max}}$ means that the limit is the maximum of the learned components' numbers of states; and $\mathcal{R} = \text{D}_{\text{min}}$ means we take the minimum.

An overall tendency in Tables 4 and 5 is that these intermediate values for \mathcal{R} indeed achieved intermediate performance between the two extremes (D_0 and D_∞). As discussed in §5 (RQ1), the best performer is likely to be one of these two extremes, but it depends on the nature of an SUL which is. Therefore, when the nature of an SUL is unknown, trying an intermediate parameter value is a viable option.

Table 4: experiment results I (extended). The rows are for different algorithms: two baselines (MnL*, CwL*) and our proposal (CCwL*) with different CA-parameters (§4). In the columns, *st.* is the number of learned states, *tr.* is that of learned transitions, *OQ reset* is the number of resets caused by output queries (it coincides with the number of OQs), *OQ step* is the number of steps caused by EQs, *EQ* is the number of equivalence queries, *EQ reset* and *EQ step* are the numbers of resets and steps caused by EQs, respectively (an EQ conducts testing and thus uses many input words), and *L. time* (“Learner time”) is the time (seconds) spent for the tasks on the learner’s side (context analysis, counterexample analysis, building observation tables, etc.), and *valid?* reports the numbers of instances of “result validated,” “result found incorrect,” and “timeout.” Note that we used 10 instances for each random benchmark.

algo.	Rand(Compl(5), LeanComp)										Rand(Star(5), LeanComp)										Rand(Path(5), LeanComp)												
	st.	tr.	OQ	OQ	EQ	EQ	EQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQs	EQ	EQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQs	EQ	EQ	EQ	L.	valid?	
	reset	step	reset	step	reset	step	reset	step	reset	step	time	reset	step	reset	step	reset	step	reset	step	reset	step	time	reset	step	reset	step	reset	step	reset	step	reset	step	time
MnL*	27K	53K	591K	18M	15.0	129	32K	1.0K	0/1/9	26K	53K	917K	33M	23.5	417	105K	1.0K	0/2/8	5.5K	21K	496K	15M	19.7	223	54K	221	1/8/1						
CwL*	46.9	12K	15K	32K	6.20	501	137K	0.86	10/0/0	55.6	26K	26K	51K	17.6	613	165K	1.14	10/0/0	47.7	166	570	2.8K	17.9	513	137K	0.53	10/0/0						
CCwL*(Eq, D _∞)	46.9	9.0K	11K	35K	2.00	101	27K	30.0	10/0/0	54.8	18K	18K	46K	7.30	107	28K	188	10/0/0	46.8	140	457	2.5K	6.70	110	29K	6.43	10/0/0						
CCwL*(Eq, D _{sum})	46.9	9.0K	11K	35K	2.00	101	27K	30.6	10/0/0	54.8	18K	18K	46K	7.30	107	28K	182	10/0/0	46.8	140	457	2.5K	6.70	110	29K	6.16	10/0/0						
CCwL*(Eq, D _{max})	46.9	8.9K	11K	120K	514	13K	3.2M	171	3/7/0	54.8	17K	18K	130K	506	19K	4.9M	268	1/9/0	46.8	138	456	2.9K	11.2	120	30K	3.21	9/1/0						
CCwL*(Eq, D _{min})	46.9	8.8K	12K	185K	914	22K	5.6M	335	1/9/0	54.8	14K	16K	337K	1.8K	63K	16M	730	0/10/0	46.4	134	466	3.9K	30.5	153	34K	2.12	7/3/0						
CCwL*(Eq, D ₀)	46.9	8.3K	16K	911K	5.4K	133K	33M	1.4K	0/10/0	54.3	7.0K	14K	1.2M	6.8K	231K	59M	1.6K	0/9/1	46.3	133	490	6.6K	64.3	203	41K	1.35	7/3/0						
CCwL*(Eq ₀ , D _∞)	46.9	9.0K	11K	35K	2.00	101	27K	30.4	10/0/0	54.8	18K	18K	47K	7.10	107	28K	24.0	10/0/0	47.3	151	501	2.8K	6.60	110	29K	2.01	10/0/0						
CCwL*(Eq ₀ , D _{sum})	46.9	9.0K	11K	35K	2.00	101	27K	29.2	10/0/0	54.8	18K	18K	47K	7.10	107	28K	23.7	10/0/0	47.3	151	501	2.8K	6.60	110	29K	1.99	10/0/0						
CCwL*(Eq ₀ , D _{max})	46.9	9.0K	11K	48K	85.7	2.3K	565K	64.2	3/7/0	54.8	18K	18K	47K	7.10	107	28K	22.8	10/0/0	47.3	151	501	2.8K	6.60	110	29K	1.89	10/0/0						
CCwL*(Eq ₀ , D _{min})	46.9	8.9K	11K	92K	352	8.7K	2.2M	196	1/9/0	54.8	14K	18K	533K	3.0K	109K	28M	1.1K	0/10/0	47.3	148	493	2.9K	10.5	116	29K	1.46	10/0/0						
CCwL*(Eq ₀ , D ₀)	46.9	8.3K	16K	911K	5.4K	133K	33M	1.3K	0/10/0	54.3	7.0K	14K	1.2M	6.8K	231K	59M	1.6K	0/9/1	46.4	134	490	6.2K	58.0	191	39K	1.37	7/3/0						
CCwL*(Uni, D ₀)	46.9	9.0K	11K	29K	2.20	101	27K	93.0	10/0/0	54.8	19K	19K	47K	7.10	107	28K	32.5	10/0/0	47.3	152	506	2.7K	6.60	110	28K	2.06	10/0/0						

algo.	Rand(Compl(5), RichComp)										Rand(Star(5), RichComp)										Rand(Path(5), RichComp)												
	st.	tr.	OQ	OQ	EQs	EQ	EQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQs	EQ	EQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQs	EQ	EQ	EQ	L.	valid?	
	reset	step	reset	step	reset	step	reset	step	reset	step	time	reset	step	reset	step	reset	step	reset	step	reset	step	time	reset	step	reset	step	reset	step	reset	step	reset	step	time
MnL*	—	—	—	—	—	—	—	—	—	—	0/0/10	41K	82K	1.0M	31M	22.0	272	67K	2.0K	0/1/9	6.8K	26K	732K	23M	22.1	301	75K	361	1/9/0				
CwL*	747	239K	1.3M	8.4M	23.6	519	138K	20.7	10/0/0	910	22K	70K	583K	40.1	637	167K	1.96	10/0/0	681	4.8K	39K	404K	36.4	536	139K	1.33	10/0/0						
CCwL*(Eq, D _∞)	47.3	9.4K	10K	31K	1.90	101	27K	23.4	10/0/0	56.7	11K	11K	29K	8.00	107	28K	190	10/0/0	44.4	138	462	2.6K	7.10	106	28K	6.76	8/2/0						
CCwL*(Eq, D _{sum})	47.3	9.4K	10K	31K	1.90	101	27K	23.3	10/0/0	56.7	11K	11K	29K	8.00	107	28K	182	10/0/0	44.4	138	462	2.6K	7.10	106	28K	6.60	8/2/0						
CCwL*(Eq, D _{max})	47.3	9.2K	11K	178K	872	24K	6.1M	343	5/5/0	56.7	9.8K	10K	63K	219	8.6K	2.2M	162	2/8/0	44.4	138	461	2.9K	10.3	114	29K	3.33	6/4/0						
CCwL*(Eq, D _{min})	47.3	9.1K	12K	308K	1.7K	40K	10M	523	1/9/0	56.7	8.2K	9.7K	214K	1.1K	39K	10M	529	0/10/0	44.4	137	475	3.7K	24.7	157	37K	2.06	6/4/0						
CCwL*(Eq, D ₀)	47.2	7.8K	14K	838K	5.0K	120K	30M	1.2K	0/9/1	56.7	6.0K	12K	991K	5.8K	187K	48M	1.3K	0/10/0	44.4	136	512	7.7K	68.4	216	44K	1.54	6/4/0						
CCwL*(Eq ₀ , D _∞)	47.3	9.4K	10K	31K	1.90	101	27K	22.4	10/0/0	56.7	11K	11K	29K	8.00	107	28K	16.1	10/0/0	44.6	145	482	2.7K	7.00	106	28K	1.89	8/2/0						
CCwL*(Eq ₀ , D _{sum})	47.3	9.4K	10K	31K	1.90	101	27K	22.1	10/0/0	56.7	11K	11K	29K	8.00	107	28K	16.3	10/0/0	44.6	145	482	2.7K	7.00	106	28K	1.93	8/2/0						
CCwL*(Eq ₀ , D _{max})	47.3	9.2K	11K	131K	587	17K	4.4M	297	5/5/0	56.7	11K	11K	29K	8.00	107	28K	16.4	10/0/0	44.6	145	482	2.7K	7.00	106	28K	1.85	8/2/0						
CCwL*(Eq ₀ , D _{min})	47.3	9.1K	12K	290K	1.6K	38K	9.6M	535	1/9/0	56.7	8.5K	10K	289K	1.6K	55K	14M	574	0/10/0	44.6	144	486	2.9K	11.0	112	28K	1.48	8/2/0						
CCwL*(Eq ₀ , D ₀)	47.2	7.8K	14K	838K	5.0K	120K	30M	1.2K	0/9/1	56.7	6.0K	12K	991K	5.8K	187K	48M	1.4K	0/10/0	44.4	137	505	7.0K	59.2	202	43K	1.47	6/4/0						
CCwL*(Uni, D ₀)	47.3	9.4K	11K	26K	1.90	101	27K	74.3	10/0/0	56.7	11K	11K	28K	8.20	108	28K	20.5	10/0/0	45.0	147	488	2.7K	7.10	106	28K	1.99	9/1/0						

algo.	MQTT-Lighting										BinaryCounter(5)										BinaryCounter(10)												
	st.	tr.	OQ	OQ	EQs	EQ	EQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQs	EQ	EQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQs	EQ	EQ	EQ	L.	valid?	
	reset	step	reset	step	reset	step	reset	step	reset	step	time	reset	step	reset	step	reset	step	reset	step	reset	step	time	reset	step	reset	step	reset	step	reset	step	reset	step	time
MnL*	169	1.5K	47K	1.4M	10.9	238	59K	17.6	0/10/0	70.0	140	212	5.9K	2.00	101	26K	0.76	10/0/0	—	—	—	—	—	—	—	—	—	—	—	—	—	0/0/10	
CwL*	39.0	2.5K	12K	61K	11.8	412	105K	0.76	10/0/0	15.0	30.0	39.1	80.6	6.00	501	129K	0.41	10/0/0	30.0	60.0	74.1	141	11.0	1.0K	258K	0.72	10/0/0						
CCwL*(Eq, D _∞)	27.0	130	348	2.4K	4.40	112	28K	1.36	10/0/0	14.0	25.0	30.0	45.0	1.00	100	26K	0.88	10/0/0	29.0	50.0	60.0	90.0	1.00	100	26K	2.27	10/0/0						
CCwL*(Eq, D _{sum})	27.0	130	349	2.7K	4.10	110	28K	1.34	10/0/0	14.0	25.0	37.0	298	8.00	107	26K	0.88	10/0/0	25.0	41.0	63.0	1.9K	13.0	117	28K	1.80	0/10/0						
CCwL*(Eq, D _{max})	27.0	130	354	3.1K	9.60	116	28K	1.35	10/0/0	14.0	25.0	40.0	329	11.0	110	26K	0.87	10/0/0	25.0	41.0	69.0	1.9K	19.0	123	28K	1.76	0/10/0						
CCwL*(Eq, D _{min})	27.0	129	386	5.2K	45.6	227	48K	1.77	8/2/0	14.0	25.0	42.5	347	13.5	113	26K	0.86	10/0/0	25.0	41.0	71.5	2.0K	21.5	126	28K	1.76	0/10/0						
CCwL*(Eq, D ₀)	27.0	129	412	7.4K	72.5	292	59K	1.94	10/0/0	14.0	25.0	44.4	357	15.4	115	26K	0.88	10/0/0	25.0	41.0	73.4	2.0K	23.4	128	28K	1.73	0/10/0						
CCwL*(Eq ₀ , D _∞)	27.0	134	362	2.8K	4.60	116	29K	1.21	10/0/0	14.0	25.0	30.0	45.0	1.00	100	26K	0.87	10/0/0	29.0	50.0	60.0	90.0	1.00	100	26K	2.27	10/0/0						
CCwL*(Eq ₀ , D _{sum})	27.0	134	360	2.6K	4.40	119	30K	1.16	10/0/0	14.0	25.0	37.0	298	8.00	107	26K	0.88	10/0/0	25.0	41.0	63.0	1.9K	13.0	117	28K	1.81	0/10/0						
CCwL*(Eq ₀ , D _{max})	27.0	134	364	2.9K	8.60	123	30K	1.14	10/0/0	14.0	25.0	40.0	329	11.0	110	26K	0.88	10/0/0	25.0	41.0													

Table 5: experiment results II (extended). The legend is the same as Table 2

algo.	Rand(Star(3),LeanComp)									Rand(Star(7),LeanComp)								
	st.	tr.	OQ	OQ	EQ	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQs	EQ	EQ	L.	valid?
			reset	step		reset	step	time				reset	step		reset	step	time	
MnL*	3.3K	13K	347K	6.8M	26.8	474	119K	89.1	1/9/0	—	—	—	—	—	—	—	—	0/0/10
CwL*	38.4	1.7K	2.6K	6.9K	12.9	409	110K	0.50	10/0/0	74.7	280K	280K	534K	26.6	819	220K	5.01	10/0/0
CCwL*(Eq, D _∞)	38.4	1.3K	2.0K	6.5K	6.80	106	28K	3.57	10/0/0	66.0	9.4K	10K	27K	9.00	108	28K	1.1K	1/0/9
CCwL*(Eq, D _{sum})	38.4	1.3K	2.0K	6.5K	6.80	106	28K	3.38	10/0/0	66.0	9.4K	10K	27K	9.00	108	28K	1.2K	1/0/9
CCwL*(Eq, D _{max})	38.4	1.3K	2.0K	8.0K	16.8	367	93K	4.61	7/3/0	73.2	53K	55K	305K	886	44K	12M	1.8K	0/4/6
CCwL*(Eq, D _{min})	38.4	1.3K	2.1K	15K	68.2	830	199K	6.06	2/8/0	66.0	6.7K	12K	742K	4.3K	155K	40M	1.7K	0/1/9
CCwL*(Eq, D ₀)	38.4	1.1K	2.7K	160K	1.0K	16K	3.9M	79.8	2/8/0	66.0	6.4K	13K	1.0M	6.2K	200K	51M	1.9K	0/1/9
CCwL*(Eq ₀ , D _∞)	38.4	1.3K	2.1K	6.6K	6.90	106	28K	1.65	10/0/0	74.5	179K	179K	447K	10.0	109	28K	646	10/0/0
CCwL*(Eq ₀ , D _{sum})	38.4	1.3K	2.1K	6.6K	6.90	106	28K	1.67	10/0/0	74.5	179K	179K	447K	10.0	109	28K	654	10/0/0
CCwL*(Eq ₀ , D _{max})	38.4	1.3K	2.1K	6.6K	6.90	106	28K	1.67	10/0/0	74.5	179K	179K	447K	10.0	109	28K	638	10/0/0
CCwL*(Eq ₀ , D _{min})	38.4	1.3K	2.2K	32K	178	2.2K	511K	12.4	4/6/0	71.7	212K	213K	596K	445	14K	3.7M	1.4K	0/3/7
CCwL*(Eq ₀ , D ₀)	38.4	1.1K	2.7K	160K	1.0K	16K	3.9M	77.9	2/8/0	66.0	6.4K	13K	1.0M	6.2K	200K	51M	2.0K	0/1/9
CCwL*(Uni, D ₀)	38.4	1.4K	2.2K	6.8K	6.40	106	28K	1.98	10/0/0	74.5	183K	183K	415K	9.50	109	28K	693	10/0/0

algo.	Rand(Star(3),RichComp)									Rand(Star(7),RichComp)								
	st.	tr.	OQ	OQ	EQs	EQ	EQ	L.	valid?	st.	tr.	OQ	OQ	EQs	EQ	EQ	L.	valid?
			reset	step		reset	step	time				reset	step		reset	step	time	
MnL*	3.2K	12K	238K	4.1M	22.1	298	74K	56.0	1/9/0	—	—	—	—	—	—	—	—	0/0/10
CwL*	540	5.6K	37K	360K	27.4	430	112K	1.21	10/0/0	1.2K	471K	535K	1.6M	57.2	851	222K	9.01	10/0/0
CCwL*(Eq, D _∞)	37.4	1.3K	2.2K	7.0K	4.90	104	28K	2.76	10/0/0	73.5	100K	101K	279K	10.5	110	28K	2.3K	4/0/6
CCwL*(Eq, D _{sum})	37.4	1.3K	2.2K	7.0K	4.90	104	28K	2.67	10/0/0	73.5	100K	101K	279K	10.5	110	28K	2.2K	4/0/6
CCwL*(Eq, D _{max})	37.4	1.3K	2.2K	11K	29.6	828	211K	6.85	6/4/0	74.5	37K	38K	198K	584	30K	8.0M	1.5K	0/2/8
CCwL*(Eq, D _{min})	37.4	1.2K	2.3K	30K	153	3.4K	853K	21.2	2/8/0	75.0	10K	14K	664K	3.7K	162K	42M	2.5K	0/1/9
CCwL*(Eq, D ₀)	37.4	1.2K	3.0K	163K	1.1K	16K	4.0M	85.4	1/9/0	75.0	8.3K	17K	1.4M	8.1K	297K	76M	3.5K	0/1/9
CCwL*(Eq ₀ , D _∞)	37.4	1.3K	2.3K	7.5K	5.30	105	28K	1.54	10/0/0	74.1	204K	204K	520K	9.50	108	28K	616	10/0/0
CCwL*(Eq ₀ , D _{sum})	37.4	1.3K	2.3K	7.5K	5.30	105	28K	1.53	10/0/0	74.1	204K	204K	520K	9.50	108	28K	607	10/0/0
CCwL*(Eq ₀ , D _{max})	37.4	1.3K	2.3K	7.5K	5.30	105	28K	1.47	10/0/0	74.1	204K	204K	520K	9.50	108	28K	603	10/0/0
CCwL*(Eq ₀ , D _{min})	37.4	1.3K	2.3K	22K	102	1.7K	433K	10.6	3/7/0	73.5	216K	219K	921K	2.4K	93K	24M	1.9K	0/2/8
CCwL*(Eq ₀ , D ₀)	37.4	1.2K	3.0K	163K	1.1K	16K	4.0M	83.4	1/9/0	75.0	8.3K	17K	1.4M	8.1K	297K	76M	3.0K	0/1/9
CCwL*(Uni, D ₀)	37.4	1.4K	2.4K	7.6K	5.00	104	28K	1.73	10/0/0	74.2	214K	215K	487K	9.80	109	28K	676	10/0/0