Active Learning of Industrial Software with Data

Lisette Sanchez[1,2], Jan Friso Groote[1,2](✉), and Ramon R. H. Schiffelers[1,2]

[1] Eindhoven University of Technology, 5612 AZ Eindhoven, The Netherlands
j.f.groote@tue.nl

[2] ASML, 5504 DR Veldhoven, The Netherlands
{lisette.sanchez,ramon.schiffelers}@asml.com

Abstract. Active automata learning allows to learn software in the form of an automaton representing its behavior. The algorithm SL*, as implemented in RALib, is one of few algorithms today that allows learning automata with data parameters. In this paper we investigate the suitability of SL* to learn software in an industrial environment.

For this purpose we learned a number of industrial systems, with and without data. Our conclusion is that SL* appears to be very suitable for learning systems of limited size with data parameters in an industrial environment. However, as it stands, SL* is not scalable enough to deal with more complex systems. Moreover, having more data theories available will increase practical usability.

Keywords: Active automata learning · SL* · Industrial environment

1 Introduction

For large and complex software systems, tasks like optimization and re-design tend to be time-consuming as they require an in-depth knowledge of the behavior of the system. Though such behavior ought to be properly documented, reality shows such documentation to often be incomplete, outdated or inconsistent. To be able to more efficiently execute said tasks, one would ideally, be able to obtain a good understanding of the behavior of a software system with minimum effort and within a limited time period.

Automata learning offers a solution to this problem, allowing one to learn the behavior of a system by sending commands to the system and observing its response. It allows for the automatic generation of formal models by applying this technique to either known systems (white-box) or unknown systems (black-box). This can be done in a passive sense by collecting and studying traces or in an active sense by firing input at the system and waiting for a response. All approaches have their pros and cons. While this research field is active in all these directions, this paper focuses only on black-box active automata learning.

H. Hojjat and M. Massink (Eds.): FSEN 2019, LNCS 11761, pp. 95–110, 2019.
https://doi.org/10.1007/978-3-030-31517-7_7

A lot of the black-box active automata learning techniques ignore data parameters as they concentrate on the control flow avoiding the intricacies of data. Such techniques offer some insight into the behavior of the system but they do not show the effect that data parameters may have on this behavior, while in reality such knowledge can be crucial to effectively optimize or re-design a system.

One of the few algorithms that allow learning with data is SL*, an extension of the famous L* algorithm of [4], presented by [8] and implemented in RALib (by H.M. Falk and P.F. Brostean, available at bitbucket.org/learnlib/ralib/). In contrast with the finite state machines that L* infers, SL* infers register automata, a type of extended finite state machine (EFSM) which holds registers and transition guards that compare registers with data parameters.

The research question this paper is concerned with is how suitable SL* is for learning software behavior including data parameters through active automata learning in an industrial environment. In particular we want to know what its shortcomings are and how complex the systems are that it can cope with.

For this purpose we learn a number of systems at ASML, which is a company in Veldhoven, The Netherlands, making wafer scanners. Wafer scanners repeatedly project images on silicium wafers to produce integrated circuits at a nano meter scale. The challenge is to project each image exactly on top of each other. These scanners consist of highly advanced hardware controlled by 50Mline of code. ASML wants to replace parts of the existing code base by model based software. Therefore, ASML explores whether such models can be learned automatically from the code. We symbolically learned both standalone components, and combinations of them. We compare this with learning the system without data parameters by instantiating data parameters with a few concrete values.

Modulo some effort to adapt the tools to the industrial environment and struggling with implementations errors, we can conclude that SL* is suitable for learning software behavior with data parameters in an industrial environment for systems with limited complexity. For more complex systems learned partial results may also provide useful insights into the behavior of the system and potentially indicate errors in the implementation. However, as it stands, learning full industrial systems, constellations of components, and even complex individual components is not within reach.

In order to make learning more applicable in an industrial context, it is very useful that SL* is extended with additional theories, especially those that allow the use of constants, lists and queues. Furthermore, scalability needs to be addressed, for instance by dividing the learning process into steps, containing subsets of the input alphabet or subsets of the data parameters and combining the results somehow.

Related Work. Looking at the field of black-box active automata learning we see that many efficient algorithms produced over time are based on Dana Angluin's approach as presented in [4].

Angluin presented an algorithm L* that is capable of inferring deterministic finite state machines from an unknown system, also referred to as a system under learning (SUL). Her technique uses the concepts of a learner and a teacher, where the teacher knows the SUL and the learner initially only knows the input alphabet and the output alphabet of the SUL. By firing two types of queries, namely (1) membership queries asking if a provided sequence of inputs and outputs is accepted by the SUL or not, and (2) equivalence queries asking whether or not the model learned so far is equivalent to the SUL, the learner is able to eventually learn the SUL.

Over time, improvements and adaptations of this algorithm have been designed, of which a short summary is given in [18]. Such improvements include research on how to perform membership queries [10,16] as well as equivalence queries [14], but also data structure improvements [11–13,17], and research to improve scalability [3,5,9].

However, it is not until recently that effort has been put into the design of learning techniques that also consider the data flow of a system when learning its behavior [1,8]. Previous algorithms can only learn behavior depending on data when data is encoded into control by instantiating data to a few concrete data values. We make use of the SL* algorithm of [8] where data is assumed to stem from data domains with very specific properties, such as $\langle \mathbb{N}, \{=\}\rangle$, i.e., the natural numbers with only equality, or $\langle \mathbb{R}, \{=, <, >\}\rangle$, i.e., the real numbers with an ordering.

So far, besides SL* there is only one other major method, namely Tomte [1,2], that can deal with data. Tomte uses a similar technique but a different framework architecture where a separate mapper component maps abstract data to concrete values. This makes the learning algorithm independent from handling the data. In SL* data is completely integrated into the learning algorithm. This is why we chose to use SL* in our investigation.

Outline. We first provide some preliminaries in Sect. 2 after which we summarize how SL* works in Sect. 3. Section 4 reports on the suitability of the adapted version of SL* in an industrial environment after which Sect. 5 follows with a discussion and conclusion.

2 Preliminaries

The SL* algorithm uses several important concepts [7,8] that are summarized in this section before the algorithm itself is explained in the next section.

2.1 Theories and Data Languages

The automata learning algorithm SL* learns automata with data registers and data input and output. The data ranges over specific theories that have the following shape.

Definition 2.1. A theory is a pair $\langle \mathcal{D}, \mathcal{R}\rangle$ where

1. $\mathcal{D}$ is a possibly unbounded domain of data values.
2. $\mathcal{R}$ is a set of relations on D.

We say that two sequences of data values $\langle d_1, \ldots, d_n\rangle$ and $\langle d'_1, \ldots, d'_n\rangle$, with $d_1, \ldots, d_n$, $d'_1, \ldots, d'_n \in \mathcal{D}$, cannot be distinguished by the relations in $\mathcal{R}$ iff for all $R \in \mathcal{R}$, we have that $R(d_{i_1}, \ldots, d_{i_j}) \iff R(d'_{i_1}, \ldots, d'_{i_j})$ with $i_1, \ldots, i_j$ being indices between 1 and n.

We assume that all elements of any $\mathcal{D}$ in this paper are denotable. The current implementation of SL* comes with two theories, namely the *IntegerEquality* theory $\langle \mathbb{N}, \{=\}\rangle$ and the *DoubleInequality* theory $\langle \mathbb{R}, \{=, <, >\}\rangle$.

An alphabet $\mathcal{E}$ is a set of actions which can be split into an input alphabet $\mathcal{E}_{in}$ and an output alphabet $\mathcal{E}_{out}$, with $\mathcal{E} = \mathcal{E}_{in} \cup \mathcal{E}_{out}$ and $\mathcal{E}_{in} \cap \mathcal{E}_{out} = \emptyset$. A parameterized symbol $\alpha(p)$ is an action $\alpha \in \mathcal{E}$ with a formal parameter p. For some fixed theory $\langle \mathcal{D}, \mathcal{R}\rangle$, a data word is a concatenation of data symbols $\alpha(d)$ with $\alpha \in \mathcal{E}$ and $d \in \mathcal{D}$, i.e., $\alpha_1(d_1) \cdot \alpha_2(d_2) \cdots \alpha_n(d_n)$ with $\alpha_1, \alpha_2, \ldots, \alpha_n \in \mathcal{E}$ and $d_1, d_2, \ldots, d_n \in \mathcal{D}$. Similarly, a parameterized word is a concatenation of parameterized symbols $\alpha(p)$ with $\alpha \in \mathcal{E}$ and a formal parameter p.

Two data words w and w' are said to be $\mathcal{R}$-indistinguishable, denoted by $w \approx_{\mathcal{R}} w'$, iff their action sequences are the same and their data parameters cannot be distinguished by the relations in $\mathcal{R}$. For example, for some action $a \in \mathcal{E}$, $\mathcal{D} = \mathbb{Z}$ and $\mathcal{R} = \{<\}$, we have that data words $\alpha(2) \cdot \alpha(1)$ and $\alpha(3) \cdot \alpha(2)$ are $\mathcal{R}$-indistinguishable, since their action sequences $\alpha \cdot \alpha$ are the same and since we have $2<2 \Leftrightarrow 3<3$, $1<1 \Leftrightarrow 2<2$, $2<1 \Leftrightarrow 3<2$ and $1<2 \Leftrightarrow 2<3$.

Definition 2.2. Given a theory $\langle \mathcal{D}, \mathcal{R}\rangle$ and $k \in \mathbb{N}$, we say that a data word u is *k-extendable* iff either

- k = 0, or
- for any data word u' with $u \approx_{\mathcal{R}} u'$ and any data symbol $\alpha(d')$ with $\alpha \in \mathcal{E}$ and $d' \in \mathcal{D}$, we have that there is a data symbol $\alpha(d)$ with $d \in \mathcal{D}$ such that $u \cdot \alpha(d) \approx_{\mathcal{R}} u' \cdot \alpha(d')$, and such that $u \cdot \alpha(d)$ is $(k{-}1)$-extendable.

For example consider some theory $\langle \mathbb{N}, \{<\}\rangle$ and a data word $u = \alpha(1) \cdot \alpha(2)$. We have that u is not 1-extendable, because for $u' = \alpha(2) \cdot \alpha(4)$ we have $u \approx_{\mathcal{R}} u'$ but for $\alpha(d') = \alpha(3)$ there is no $\alpha(d)$ such that $\alpha(1) \cdot \alpha(2) \cdot \alpha(d) \approx_{\mathcal{R}} \alpha(2) \cdot \alpha(4) \cdot \alpha(3)$.

A theory is said to be *strongly extendable* iff all data words are ∞-extendable and a theory is said to be *weakly extendable* iff for all data words u and for all $k \in \mathbb{N}$, there is a data word u' with $u' \approx_{\mathcal{R}} u$ that is k-extendable.

Note. SL* requires a theory to be either weakly extendable or strongly extendable. The two theories currently implemented in RALib, namely $\langle \mathbb{N}, \{=\}\rangle$ and $\langle \mathbb{R}, \{=, <, >\}\rangle$ are both strongly extendable.

A data language $\mathcal{L}$ is a set of data words, such that for all two words w and w' that are $\mathcal{R}$-indistinguishable, we have that $w \in \mathcal{L} \iff w' \in \mathcal{L}$. A word w is said to be *accepted* by $\mathcal{L}$ iff $w \in \mathcal{L}$, and *rejected* otherwise.

Furthermore, we make the following assumptions about any data language $\mathcal{L}$ (w.r.t. a theory $\langle \mathcal{D}, \mathcal{R}\rangle$)

- $\mathcal{L}$ is prefix-closed, i.e., for any two words w, w' we have that if $w \cdot w' \in \mathcal{L}$ then also $w \in \mathcal{L}$.

- $\mathcal{L}$ is input/output alternating, i.e., all data words in $\mathcal{L}$ contain alternating input and output data symbols and start with an input data symbol.
- $\mathcal{L}$ is output-deterministic, i.e., for any word w ending in an input symbol, we have, for all $\alpha(d), \alpha(d')$ with $\alpha \in \mathcal{E}_{out}$ and $d, d' \in \mathcal{D}$, that if both $w \cdot \alpha(d)$ and $w \cdot \alpha(d')$ are in $\mathcal{L}$ then words $w \cdot \alpha(d)$ and $w \cdot \alpha(d')$ are $\mathcal{R}$-indistinguishable.

A system under learning (SUL) as used in this paper is an implementation of a data language.

2.2 Register Automata

Register automata (RA) are a type of extended finite state machines that can be used to model data languages.

We assume a theory $\langle \mathcal{D}, \mathcal{R} \rangle$, an alphabet $\mathcal{E}$ and a set of registers $\mathcal{G} = \{x_1, \ldots, x_n\}$. A guard is a conjunction of negated or unnegated relations from $\mathcal{R}$ over registers and formal parameters p used in parameterized symbols $\alpha(p)$. A register automaton is then defined as follows.

Definition 2.3. A register automaton is a tuple $\mathcal{A} = (\mathcal{S}, s_0, \mathcal{X}, \Gamma, \lambda)$ where

- $\mathcal{S}$ is a finite set partitioned in input states $\mathcal{S}_{in}$ and output states $\mathcal{S}_{out}$.
- $s_0 \in \mathcal{S}$ is the initial state.
- $\mathcal{X} : \mathcal{S} \to \mathcal{G}$ is a mapping that maps each state to a finite set of registers.
- Γ is a finite set of transitions, each of the form $(s, \alpha(p), g, \pi, s')$ where
  - s is the source state.
  - $\alpha(p)$ is a parameterized symbol. If s is an input state, $\alpha \in \mathcal{E}_{in}$ and s' is an output state. Otherwise, i.e., if s is an output state, then $\alpha \in \mathcal{E}_{out}$ and s' is an input state.
  - g is a guard over p and $\mathcal{X}(s)$.
  - π is an assignment that updates registers in $\mathcal{X}(s')$ with values of p and registers in $\mathcal{X}(s)$.
  - s' is the target state.
- $\lambda : \mathcal{S} \to \{+, -\}$ is a mapping that maps each state to either $+$ or $-$, indicating whether a state is accepting.

We write $s \xrightarrow{\alpha(p),g,\pi} s'$ iff $(s, \alpha(p), g, \pi, s') \in \Gamma$. We write $s \xrightarrow{\alpha(p),g,\pi}$ iff there is an $s' \in S$ such that $s \xrightarrow{\alpha(p),g,\pi} s'$.

We assume that the RAs in this paper are deterministic, i.e., there are no data words that lead to both accepting and rejecting states and we say that a register automaton has runs over all data words iff every input state has outgoing transitions for all actions in $\mathcal{E}_{in}$ and every output state has outgoing transitions for all actions in $\mathcal{E}_{out}$. In this case unwanted actions lead to rejected states.

In general the initial state s_0 is an input state, i.e., $s_0 \in \mathcal{S}_{in}$. However, we employ symbolic decision trees that are instances of register automata where the initial state can also be an output state.

We use SRAs to represent a SUL:

Definition 2.4. A simple register automaton (SRA) is a register automaton $\mathcal{A} = (\mathcal{S}, s_0, \mathcal{X}, \Gamma, \lambda)$ with $\mathcal{X}(s_0) = \emptyset$ that has runs over all data words.

Note. When visualizing RAs in this paper, input states are indicated by solid lines and output states by dotted lines. Accepted states are indicated by double lines and rejected states by singular lines. Furthermore, input actions are typically prepended with a question mark and output actions are typically prepended with an exclamation mark.

2.3 Symbolic Decision Trees

One of the most distinguishing differences between SL* and its predecessor L* is that SL* uses symbolic decision trees to represent sets of data words.

Definition 2.5. A symbolic decision tree (SDT) is a register automaton $\mathcal{T} = (\mathcal{S}, s_0, \mathcal{X}, \Gamma, \lambda)$ where $\mathcal{S}$ and Γ form a tree with root s_0. We write $\mathcal{X}(\mathcal{T})$ to denote $\mathcal{X}(s_0)$.

An SDT models (part of) the data language based on the valuation of $\mathcal{X}(\mathcal{T})$. For example, consider the theory $\langle \mathbb{R}, \{=, <, >\}\rangle$. There are two registers x_1 and x_2 of which only the latter is used. A symbolic decision tree can express that traces ϵ and $\alpha(p)$ can be accepted provided $p \geq x_2$. This SDT with a depth 1 is depicted in Fig. 1. For any sequence of actions σ, an SDT of depth $|\sigma|$ can be constructed.

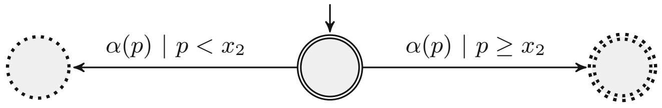


Fig. 1. SDT created for the prefix and suffix as shown in Table 1

Equivalence. To test the equivalence of two SDTs, the following notion of isomorphism is used.

Definition 2.6. Let $\mathcal{T} = (\mathcal{S}, s_0, \mathcal{X}, \Gamma, \lambda)$ and $\mathcal{T}' = (\mathcal{S}', s_0', \mathcal{X}', \Gamma', \lambda')$ be two SDTs with sets of registers $\mathcal{G}$ and $\mathcal{G}'$ respectively. Let $\gamma : \mathcal{G} \to \mathcal{G}'$ be a bijection. We say that $\mathcal{T}$ and $\mathcal{T}'$ are isomorphic under γ, denoted $\mathcal{T} \simeq_\gamma \mathcal{T}'$, iff there is a bijection $\phi : \mathcal{S} \to \mathcal{S}'$ such that:

- $\phi(s_0) = s_0'$,
- $\gamma(\mathcal{X}(s)) = \mathcal{X}'(\phi(s))$ for all $s \in \mathcal{S}$,
- $\lambda(s) = \lambda'(\phi(s))$ for all $s \in \mathcal{S}$, and
- $(s_1, \alpha(p), g, \pi, s_2) \in \Gamma \iff (\phi(s_1), \alpha(p), g^\gamma, \pi^\gamma, \phi(s_2)) \in \Gamma'$

where g^γ and π^γ are respectively a guard and an update with the registers replaced according to γ.

Definition 2.7. Let $\mathcal{T}$ and $\mathcal{T}'$ be two SDTs with $\mathcal{T} \simeq_\gamma \mathcal{T}'$. We say that $\mathcal{T}$ and $\mathcal{T}'$ are isomorphic, denoted $\mathcal{T} \simeq \mathcal{T}'$, iff γ is a bijection.

2.4 Observation Table

An observation table is a data structure used to store results on which data words are accepted by the SUL and which are not.

Definition 2.8. Let $\mathcal{F}$ represent the set of all SDTs. Given an alphabet $\mathcal{E}$ and a theory $\langle \mathcal{D}, \mathcal{R} \rangle$, an observation table is a tuple $\mathcal{O} = (\mathcal{U}, \mathcal{U}^+, \mathcal{V}, \mathcal{Z})$ where

- $\mathcal{U}$ is a prefix-closed set of data words, referred to as short prefixes.
- $\mathcal{U}^+ = \{u \cdot \alpha(d) \mid u \in \mathcal{U} \text{ and } \alpha \in \mathcal{E}\}$ (adhering to the input/output alternating assumption (Sect. 2.1)) and for some $d \in \mathcal{D}$, is a set of extended prefixes.
- $\mathcal{V}$ is a set of parameterized words, referred to as symbolic suffixes.
- $\mathcal{Z} : (\mathcal{U} \cup \mathcal{U}^+) \rightarrow \mathcal{F}$, is a mapping that maps each prefix to an SDT.

An observation table is considered *closed* iff for every $u' \in \mathcal{U}^+$ there is a $u \in \mathcal{U}$ and a γ such that $\mathcal{Z}(u') \simeq_\gamma \mathcal{Z}(u)$. Intuitively this means that for every extended prefix there should be a short prefix such that their SDTs are isomorphic under some γ. In this way, the number of states required to represent $\mathcal{L}$ is limited to $|\mathcal{U}|$.

An observation table is considered *register-consistent* iff for every $u \cdot \alpha(d) \in \mathcal{U}^+$ that requires an initial register, i.e., $x_i \in \mathcal{X}(\mathcal{Z}(u \cdot \alpha(d)))$, we also have $x_i \in \mathcal{X}(\mathcal{Z}(u))$. Intuitively this means that if some SDT requires an initial register, this register should have been stored previously.

Intuitively, an SDT $\mathcal{Z}(u)$ indicates in a generic way how a SUL responds after it is requested to perform a data word u. This response matches the actions from a suffix, and it is formulated abstractly in terms of registers and conditions, where the respective data values in u correspond to the registers in $\mathcal{Z}(u)$.

Creating SDTs. Given a theory $\langle \mathcal{D}, \mathcal{R} \rangle$, a prefix $u \in (\mathcal{U} \cup \mathcal{U}^+)$ and a set of symbolic suffixes $\mathcal{V}$, let $\mathcal{D}'$ represent the set of (instantiated) data values in the prefix and let $\mathcal{P}'$ represent the set of (uninstantiated) formal parameters in $\mathcal{V}$. A set of test cases $R(p', d') \cup R(d', p')$ is then created for all $R \in \mathcal{R}$, $d' \in \mathcal{D}'$ and $p' \in \mathcal{P}'$. For each such test case, all $p' \in \mathcal{P}'$ are instantiated with appropriate data values $d \in \mathcal{D}$.

For example, consider a SUL that disallows decreasing numbers and consider the theory $\langle \mathbb{R}, \{=, <, >\} \rangle$, a data word $u = \alpha(1)\cdot ok\cdot\alpha(3)\cdot ok$ for which the data values 1 and 3 are stored in registers x_1 and x_2 respectively, and $\mathcal{V} = \{\alpha(p)\}$. We then have $\mathcal{D}' = \{1, 3\}$ and $\mathcal{P}' = p$.

Table 1. Test cases for a specific prefix and symbolic suffix

Prefix	Symbolic suffix	Test cases	Instantiated suffix	Accepted
$\alpha(1)\cdot ok\cdot\alpha(3)\cdot ok$	$\alpha(p)$	$p < 1$	$\alpha(0)$	no
		$1 < p < 3$	$\alpha(2)$	no
		$p = 1$	$\alpha(1)$	no
		$p = 3$	$\alpha(3)$	yes
		$3 < p$	$\alpha(4)$	yes

Table 1 shows the test cases generated for this example. For each test case an instantiation is created formed by the concatenation of the prefix and the instantiated suffix that satisfies the test case. The instantiated suffix is also depicted in Table 1. The instantiation is sent to the SUL which either accepts or rejects it (also indicated in Table 1). From these results an SDT can then be created as is shown in Fig. 1. This SDT indicates that all data words $\alpha(1) \cdot ok \cdot \alpha(3) \cdot ok \cdot \alpha(p)$ with $p \geq x_2$ are accepted by the SUL and all data words $\alpha(1) \cdot ok \cdot \alpha(3) \cdot ok \cdot \alpha(p)$ with $p < x_2$ are rejected by the SUL.

3 The Algorithm SL*

3.1 Algorithm

The algorithm SL^* presented by Cassel et al. is an extension of Dana Angluin's algorithm L^*. For a more detailed description of either algorithm we refer to the original papers [4,8]. In this section we provide a brief summary of SL^* which should contain sufficient information for the purpose of this paper.

The main idea of SL^* is similar to that of L^*, where concepts of a learner and teacher are used. The learner attempts to learn a black-box system under learning (SUL) that models a data language $\mathcal{L}$ with alphabet $\mathcal{E}$. The SUL is represented as a register automaton (RA, Sect. 2.2) and $\mathcal{L}$ is inferred by asking the teacher so-called membership queries and equivalence queries.

The learner makes use of an observation table (Sect. 2.4) to create, and store the results of membership queries and to build a hypothesis automaton based on this table. The rows of an observation table consist of a set of prefixes, containing specific data values, and the columns consist of a set of symbolic suffixes, which are abstracted from specific data values. Every cell represents a membership query, which is a data word w (Sect. 2.1), where w is the concatenation of the prefix and suffix of the cell. The answers from the teacher to each membership query are transformed into a symbolic decision tree (SDT, Sect. 2.3), which represents, for a given prefix, for which instantiations of parameters in the suffix the SUL accepts the query (Sect. 2.3).

The learner continues to update the observation table by asking the teacher membership queries until the observation table is both closed and register-consistent (Sect. 2.4), at which point it creates a hypothesis automaton from the table, represented as an RA, and sends it to the teacher in the form of an equivalence query. Should the hypothesis automaton be equivalent to the SUL the reply will be positive. Otherwise the teacher will provide a counterexample in the form of a query that is accepted by the hypothesis but not by the SUL or vice versa, after which the learner will continue with an updated observation table and another set of membership queries until it creates the correct hypothesis automaton.

Given an RA with t transitions and at most r registers per state, that models a data language $\mathcal{L}$, SL^* infers $\mathcal{L}$ with $O(tr)$ equivalence queries and $O(t^2r + trm)$ membership queries, where m is the length of the longest counterexample [8].

3.2 Example

In this section, we demonstrate the algorithm SL^* by means of an example over the data theory $\langle \mathbb{R}, \{=, <, >\} \rangle$.

Consider the SUL as presented in Fig. 2, with input alphabet $\mathcal{E}_{in} = \{enter(p)\}$ with $p \in \mathbb{N}$, and output alphabet $\mathcal{E}_{out} = \{ok, nok\}$. Any transitions not shown in the figure lead to sink states which are omitted from the figure.

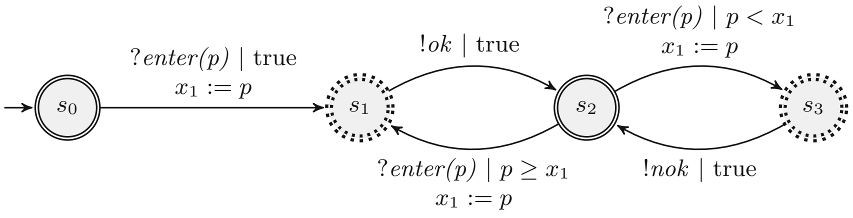
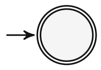
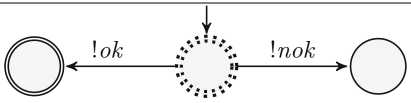


Fig. 2. Example SUL as described above

The observation table is initialized with $\mathcal{U} = \{\epsilon\}$ and $\mathcal{V} = \{\epsilon\} \cup \mathcal{E}_{out}$, as is shown in Table 2. True guards are omitted. Transitions that do not follow the assumption of alternating input and output symbols are not processed.

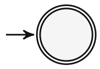
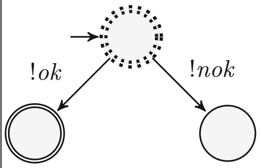
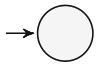
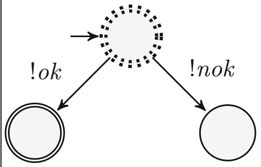
Table 2. Observation table after the first round

		$\mathcal{V} = \{\epsilon, ok, nok\}$
$\mathcal{U}$	ϵ	
$\mathcal{U}^+$	$enter(1)$	

The top SDT shown in the table indicates that the empty data word is accepted by the SUL (indicated by the doubly lined state) but it does not show the results for data words *ok* and *nok* as they do not follow the assumption of alternating input and output symbols.

The bottom SDT shown in the table indicates with its initial, doubly dotted state that the data word *enter*(1) leads to an accepting output state, i.e., a state that requires the next action to be from the output alphabet (Sect. 2.2). Furthermore, the SDT indicates that the data word *enter*(1) $\cdot$ *ok* is accepted by the SUL and the data word *enter*(1) $\cdot$ *nok* is rejected by the SUL (indicated by the singly lined state). Since $\mathcal{V}$ only contains the data symbols ϵ, *ok* and *nok*, no other transitions are processed for this SDT.

Table 3. Observation table after the fourth round

		$\mathcal{V} = \{\epsilon,\ ok,\ nok\}$
$\mathcal{U}$	ϵ	
	$enter(1)$	
	$enter(1) \cdot nok$	
	$enter(1) \cdot nok \cdot enter(2)$	
$\mathcal{U}^+$	$enter(1) \cdot ok$	same as ϵ
	$enter(1) \cdot nok \cdot enter(2) \cdot ok$	same as $enter(1) \cdot nok$
	$enter(1) \cdot nok \cdot enter(2) \cdot nok$	same as $enter(1) \cdot nok$

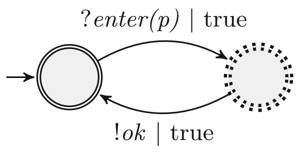


Fig. 3. The hypothesis automaton based on the observation table shown in Table 3

Table 2 is not closed however, since for row $enter(1)$ in $\mathcal{U}^+$ there is no equivalent row in $\mathcal{U}$, hence row $enter(1)$ is added to $\mathcal{U}$ and $\mathcal{U}^+$ is adapted appropriately. Table 3 shows the observation table that is eventually obtained, which is both closed and register-consistent.

A hypothesis automaton is created based on this table (Fig. 3). This results in the following counterexample from the SUL: $enter(1) \cdot ok \cdot enter(2) \cdot ok \cdot enter(0) \cdot nok$, which is accepted by the SUL but not by the hypothesis automaton.

Eventually, the algorithm obtains another closed and register-consistent table, for which a new hypothesis automaton $\mathcal{H}$ is created (Fig. 4) and sent to the teacher, resulting in a positive reply, meaning the learning process is complete and the learner has learned the SUL. Looking past some syntactic differences we can see that the automata in Figs. 2 and 4 are isomorphic under $x_1 \rightarrow r_1$.

4 Industrial Setting

4.1 Experimental Setup

As the main purpose of this paper is to investigate the suitability of SL* in an industrial environment, the algorithm has been applied to several case studies extracted from the coding environment of ASML. In this section we elaborate on the case studies themselves and on applying the algorithm to these case studies.

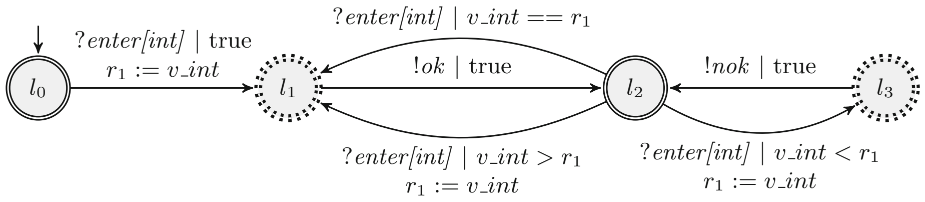


Fig. 4. Result of *SL** after learning the SUL of Fig. 2

Case Studies. In some departments within ASML, a modelling environment called ASD:Suite [6] (see also [15] in this proceedings for a compact explanation of ASD) is used to model the behavior of software from which the source code is generated. Using ASD:Suite, a major component can be decomposed into many smaller components. In particular there is a very large component within the code base of ASML that is decomposed into 200–300 smaller components. For the purpose of this paper, several of these smaller components are considered as case studies.

It is important to note that components modeled in the ASD:Suite use guards to make control flow decisions based on state variables, i.e., variables used to describe the state of a system, as the use of such state variables provides a more compact model of the behavior of the component. Any component using state variables however, can also be modeled as a component without state variables, resulting in a more extensive behavioral model with more explicit states. Visually, this means that the second model has a layered structure, where different layers of states represent the different values of an otherwise present state variable. Models learned by SL* do not contain state variables and thus contain such a layered structure when inferring ASD:Suite components that do contain state variables.

The case studies we use in this paper are components referred to as c_1, c_2 and c_3, of which only the last one contains behavior that is influenced by its data parameters. These components communicate with each other as shown in Fig. 5. For confidentiality reasons the names of these components are omitted in this section, but we provide a short summary of their behavior.

Component c_3 constitutes a rather typical list implementation, with behavior that allows adding items to the list, removing items from the list, adapting

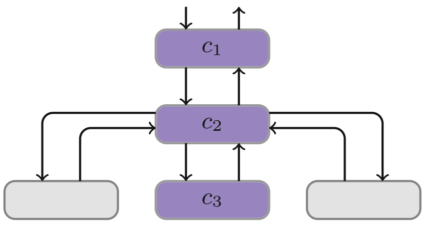


Fig. 5. Communication between the three components c_1, c_2 and c_3

items on the list and viewing items on the list. The behavior of this component is influenced by data parameters passed to the component and is therefore chosen for this case study. Component c_3 communicates only with c_2, which receives calls from c_1 concerning the list and forwards them to c_3 whilst also communicating with two other components, indicated in gray in Fig. 5. Component c_1 is the simplest component of these three, as it only functions as a communicator between other components and c_2.

Each component is learned separately, observing the running times and studying the results, as well as a combination of c_1 and c_2, and a combination of all three components, to observe the learning results and running times when dealing with increased complexity.

To illustrate the strength of SL* when it comes to learning software behavior with (abstract) data parameters, the same set of test cases is learned with concrete data parameters. For this purpose, the input alphabet of each test case is extended such that for each parameterized input i, five concrete but arbitrary inputs are created instead, with concrete values v_1, $v_2, \ldots, v_5$.

Preparation. In preparation of applying the algorithm, code is generated for all components in the SUL and for the direct environment of the SUL, i.e., the components in the wrapper. The input and output alphabet are provided, as is allowed in the black-box model, and a wrapper component is created.

Furthermore, due to the restrictions of theories $\langle \mathcal{D}, \mathcal{R} \rangle$, each parameter in both the SUL and the stub code is transformed into a parameter $d \in \mathcal{D}$. Should there be no access to the code of the SUL, then the wrapper has to map parameters $d \in \mathcal{D}$ to the appropriate parameter types as they are in the SUL. For the purpose of this paper, this extension is omitted, however. The IntegerEqualityTheory is used in all cases.

Environment. The experiments are conducted on a machine with the following properties:

- System: Windows 7, 64 bit, 8 GB RAM, 2.4 GHz CPU.
- JVM: Eclipse Neon 4.6.3, 64 bit, -Xms512m, -Xmx7144m.

4.2 Results

The results are shown below in Table 4, which contains the following information:

- **$\#\mathcal{E}_{in}$:** the number of inputs in $\mathcal{E}_{in}$.
- **$\#\mathcal{E}_{out}$:** the number of outputs in $\mathcal{E}_{out}$ prior to the final round of the algorithm, i.e., when no new outputs are added to $\mathcal{E}_{out}$ anymore.
- **sv:** the number of state variables used in the implementation.
- **time:** the total running time of the algorithm averaged over 10 runs. The word 'om' indicates that the learning process was interrupted due to an out of memory error. For these cases the last obtained results are listed.
- **states:** the number of states in the resulting model.
- **transitions:** the number of transitions in the resulting model.
- **mq:** the number of membership queries performed, averaged over 10 runs.
- **eq:** the number of equivalence queries performed, averaged over 10 runs.
- **correct:** whether or not the learned result is correct. This correctness is evaluated by visual inspection. Notations 'N.A'. and '??' denote cases that do not have a final model to inspect or cases that are too big to evaluate by visual inspection, respectively.

For each test case, two values are listed to indicate the result for the test case with abstract and with concrete data parameters. The absence of two values indicates a similar result for both cases. In the test cases that resulted in an out of memory, the DoubleInequalityTheory has also been tried, resulting in the same outcome.

For two out of five test cases the experiments with data parameters led to out of memory. In both cases one parameterized input (the same input in both cases) was removed from the input alphabet in order to be able to obtain a partial result. This input dealt with obtaining the next item from an iterator. These adapted test cases are indicated by the use of asterisks and their results are shown in Table 5.

All in all, the results mostly indicate a problem with scalability as increased complexity quickly leads to out of memory when including data parameters. For those cases that were successfully learned though, the success of SL* becomes apparent. For components that do not contain behavior based on data parameters, learning with SL* uses much less membership queries to learn the same number of states with fewer transitions, resulting in more visual models. For components that do contain behavior based on data parameters, an even more distinguishing result becomes visible. Note that the test cases without data parameters contain only 5 hard-coded values, where the cases with data parameters allow for values in an infinite domain, thereby attesting to the strength of SL* when it comes to learning software behavior with data parameters.

Table 4. Results of applying SL* to several case studies from an industrial environment

	$\#\mathcal{E}_{in}$	$\#\mathcal{E}_{out}$	sv	Time (sec)	States	Transitions	mq	eq	Correct
c_1	18	25	2	9/6	31/31	114/254	3984/9307	11/10	yes
c_2	22	39	5	12K/6.9K	660/660	4.9K/12.2K	1.4M/3.3M	135/117	yes/??
c_3	12	15	1	om/8.3K	127/2.7K	350/18K	19K/4.9M	38/245	N.A./??
c_1, c_2	20	33	6	137/66	108/108	752/1.7K	58k/132k	34/36	yes
c_1, c_2, c_3	10	17	6	om/32K	328/2.4K	1.3K/30K	177K/11.5M	77/239	N.A./??

Table 5. Results of applying SL* to several simplified case studies from an industrial environment

	Time (sec)	States	Transitions	mq	eq	Correct
c_3 *	136/3.2K	87/842	227/5.4k	10k/895k	13/145	yes/??
c_1, c_2, c_3 *	490/3835	109/838	383/9.1K	26k/2.2M	36/147	yes/??

5 Conclusion

Learning well known software with data parameters using SL* that fit the available theories is quite impressive. Applying SL* in an industrial environment can be of use, but there are quite some limitations to consider.

First of all, two direct shortcomings were found that prevented SL* from learning the correct results. Industrial systems do not have a strict alternation of input and output. Furthermore, there are too many software flaws in the available implementation of SL*. Both had to be dealt with in order to allow SL* to correctly learn the results of the industrial case studies. Especially, the latter is not only very time consuming, but it also obfuscates conclusions about the quality of SL*.

While applying SL* to said industrial cases, another weakness became apparent, namely the limited availability of data theories, forcing the use of integers and doubles and limiting the operators usable in guards to equality, $<$ and $>$. In general other data types such as lists and sets are used in SULs but cannot be learned. More importantly, the source code of a SUL may not be accessible, and in such a case it is generally not known which data types are used; one can only hope that they match the available theories.

Another problem is the scalability of the algorithm. Where smaller sized systems can be learned quite fast, an increased complexity quickly results in out of memory errors. Unfortunately, it is not always clear in such cases whether the problems find their origin in the size of the SUL, the quality of the implementation or in the data types that must be learned.

Despite these weaknesses, the strength of SL* has become apparent when applying the algorithm to industrial case studies in comparison to learning these case studies without data parameters. Under the right circumstances, SL* can learn the behavior of a component with data much more efficiently and with a much more compact result, thereby providing valuable insights to engineers requiring to gain knowledge of this behavior. Even when using SL* to learn the behavior of a component that only employs trivial, finite data, the results can be gained more efficiently and are in such a case more compact, by representing this finite data using an infinite data domain.

It is clear that learning industrial software with data still has a long way to go. But under the right circumstances, it can certainly work. And in such cases the learned result generally offers a great amount of insight into the behavior of a system, reducing the amount of time and effort required to gain knowledge about the behavior of the system manually.

References

1. Aarts, F.: Tomte: bridging the gap between active learning and real-world systems. Ph.D. thesis, Radboud University, Nijmegen, The Netherlands (2014)
2. Aarts, F., Fiterau-Brostean, P., Kuppens, H., Vaandrager, F.: Learning register automata with fresh value generation. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) ICTAC 2015. LNCS, vol. 9399, pp. 165–183. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25150-9_11
3. Aarts, F., Jonsson, B., Uijen, J., Vaandrager, F.: Generating models of infinite-state communication protocols using regular inference with abstraction. Formal Methods Syst. Des. **46**(1), 1–41 (2015)
4. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. **75**(2), 87–106 (1987)
5. Bratus, S., Lindner, F. (eds.): 8th USENIX Workshop on Offensive Technologies, WOOT 2014, San Diego, CA, USA, 19 August 2014. USENIX Association (2014)
6. Broadfoot, G., Hopcroft, P.: Analytical software design (2003). Uploaded to researchgate.net
7. Cassel, S., Howar, F., Jonsson, B.: RALib: a LearnLib extension for inferring EFSMs. DIFTS (2015)
8. Cassel, S., Howar, F., Jonsson, B., Steffen, B.: Active learning for extended finite state machines. Formal Asp. Comput. **28**(2), 233–263 (2016)
9. Cho, C., Babic, D., Shin, E., Song, D.: Inference and analysis of formal models of botnet command and control protocols. In: Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, 4–8 October 2010, pp. 426–439 (2010)
10. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. Log. J. IGPL **14**(5), 729–744 (2006)
11. Isberner, M.: Foundations of active automata learning: an algorithmic perspective. Ph.D. thesis, Technical University Dortmund, Germany (2015)
12. Isberner, M., Howar, F., Steffen, B.: The TTT algorithm: a redundancy-free approach to active automata learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) RV 2014. LNCS, vol. 8734, pp. 307–322. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11164-3_26
13. Kearns, M., Vazirani, U.: An Introduction to Computational Learning Theory. MIT Press, Cambridge (1994)
14. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - a survey. Proc. IEEE **84**, 1090–1123 (1996)

15. Neele, T., Rol, M., Groote, J.F.: Verifying system-wide properties of industrial component-based software. In: Hojjat, H., Massink, M. (eds.) FSEN 2019. LNCS, vol. 11761, pp. 158–175. Springer, Cham (2019)
16. Peled, D., Vardi, M., Yannakakis, M.: Black box checking. J. Autom. Lang. Comb. **7**(2), 225–246 (2002)
17. Rivest, R., Schapire, R.: Inference of finite automata using homing sequences. Inf. Comput. **103**(2), 299–347 (1993)
18. Vaandrager, F.: Model learning. Commun. ACM **60**(2), 86–95 (2017)