Active Learning from Blackbox to Timed Connectors

Yi Li, Meng Sun and Yiwu Wang

LMAM & Department of Informatics, School of Mathematical Sciences, Peking University, Beijing, China

liyi_math@pku.edu.cn, summeng@math.pku.edu.cn, yiwuwang@126.com

Abstract—**Coordination models and languages play a key role in formally specifying the communication and interaction among different components in large-scale concurrent systems. In this paper, we use active learning to extract timed connector models from black-box system implementation. Firstly, parameterized Mealy machines(PMM) is introduced as an operational semantic model for channel-based coordination language Reo. With product and link operators defined, we can construct complex connectors by joining basic channels in form of PMM. Moreover, with a concretize mapping function, PMMs can be easily transformed into Mealy machines, and the latter can be extracted by an optimized L* algorithm.**

Index Terms—**Active Learning, Coordination, Connectors**

I. INTRODUCTION

Distributed real-time embedded systems (DRES) are reforming our lives with the Internet of Things(IoT), wherein individual components are composed via *connectors* to build complex systems. In this case, we need to specify the coordination processes with *coordination languages*, such that formal techniques can be applied to guarantee their reliability.

As a real-time extension of *Reo*, *Timed Reo* [3], [10] is used in this paper to formalize connectors in DRES. Reo is a channel based exogenous coordination language proposed by F. Arbab in [2], which is a powerful "glue language" in component-based development [5].

Formal verification and validation techniques have also been proved applicable for timed connectors, e.g. a comformance testing framework on timed connectors has been proposed in [9]. A SAT-based bounded model checking approach was adapted in [8] for verifying connectors. Although these solutions are practical and impressive, there is one important question faced by most of them: *how to obtain the formal models?*

Correctness of connectors is closely related to some low-level implementation details. For example, well-written code may behave dramatically weird with an improper set of concurrency primitives. Such scenarios happen frequently in embedded systems with different hardware platforms or operating systems. Consequently, manually extracting a proper model from an existing connector implementation seems rather unreliable, even with reference to its source code.

As a branch of *machine learning* technique, active learning offers a means to obtain high-level models from low-level models. Works in [1], [4], [11] show interesting examples where active learning is used to extract *Mealy machines* or *regular languages* without time domain.

In this paper we address this question by proposing an active learning framework as shown in Fig. **??** to automatically extract timed connector models from blackbox implementations. To achieve this goal, we first introduce *parameterized Mealy machine* as a parameterized semantics for timed Reo connectors, which can be transformed to concrete Mealy machines with a given alphabet. Then the $L*$ algorithm [1] is adapted and optimized to extract Mealy machines with timed action from the connector implementations as blackboxes.

The rest of the paper is organized as follows: In Section Section II, we defines an operational semantics of Reo, which is used in Section III to show how to extract Reo models from blackboxes by means of active learning. In Section IV, we discuss the implementation and optimization of the approach. Finally, Section VI concludes the paper.

II. TIMED CONNECTORS AS MEALY MACHINES

In this section, we illustrate how to transform a timed connector into a Mealy machine with timed action. Since time is not involved in original Mealy machine, we first discuss how to formalize the time dimension in Mealy machine. After that, we present the notion of *parameterized Mealy machine* as a bridge between connectors and Mealy machines.

A. Parameterized Mealy Machine

We present a model named *parameterized Mealy machine* to model timed connectors. PMM is supposed to behave as a bridge between Mealy machines and timed connectors. Connectors are firstly defined as PMMs, and composed by *product* and *link* operators. Then, concrete Mealy-machine model can be generated from the PMM model. Following the formal definition of Mealy machine in Section **??**, we define the *parameterized Mealy machine* as follows.

Definition 1 (Parameterized Mealy Machine): A *Parameterized Mealy Machine* with a parameter Σ is defined as a 6-tuple $\mathcal{PM} = \langle S, s_0, I, O, \delta, \lambda \rangle$ where

- The value of Σ is a *finite* data set (hereinafter referred as alphabet), e.g. $\{a, b\}$,
- S maps an alphabet to a *finite* set of states,
- s_0 is the initial state. It satisfies $\forall \Sigma.s_0 \in S(\Sigma)$,
- I is a finite set of source-ends,
- O is a finite set of sink-ends,

- δ maps an alphabet to an *output function*. We use $\delta(\Sigma) : S(\Sigma) \times Input(\Sigma, I, O) \to Output(\Sigma, O)$ to denote the output function,
- λ maps an alphabet to a *transition function*. We use $\lambda(\Sigma) : S(\Sigma) \times Input(\Sigma, I, O) \to S(\Sigma)$ to denote the transition function.

In the definition above, *Input* and *Output* are used to generate the set of input actions and output actions from the corresponding alphabets and source/sink ends. $Input(\Sigma, I, O)$ is defined as a set of functions on $I \cup O$ and an additional *time action* T, where for all $f \in Input(\Sigma, I, O) \setminus \{T\}$ we have

$$\forall i \in I.f(i) \in \Sigma \cup \{\bot\} \land \forall o \in O.f(o) \in \{\triangleright, \otimes\}$$

where we use $\bot$ to indicate that there is *no* data item on a channel end. Besides, $\triangleright$ means that a sink end is ready for writing, $\otimes$ otherwise. In the same way, $Output(\Sigma, O)$ is defined as a set of functions on O, with an additional symbol $\ominus$ indicating an input failure. Similarly, for all $f \in Output(\Sigma, O) \setminus \{\ominus\}$,

$$\forall o \in O.f(o) \in \Sigma \cup \{\bot\}$$

A non-timed input action $a \in Input(\Sigma, I, O)$ can be restricted to $I' \subseteq I$, denoted by $a \downarrow_{I'}$ which is defined on $Input(\Sigma, I', O)$ and satisfies

$$\forall i \in I'.a \downarrow_{I'} (i) = a(i)$$

Restriction on output actions can be defined similarly.

Example 1 (Input and Output): If we have a simple alphabet with only one item d_0, a source-end named A, and a sink-end named B, the input actions would be

$$
\begin{aligned}
&Input(\{d_0\}, \{A\}, \{B\}) \\
=\ &\{\{A \mapsto d_0, B \mapsto \triangleright\}, \{A \mapsto d_0, B \mapsto \otimes\}, \\
&\{A \mapsto \bot, B \mapsto \triangleright\}, \{A \mapsto \bot, B \mapsto \otimes\}, T\}
\end{aligned}
$$

and its output actions

$$Output(\{d_0\}, \{A\}, \{B\}) = \{\{B \mapsto d_0\}, \{B \mapsto \bot\}, \ominus\}$$

We use $\varnothing$ to denote the empty output when no data item is written to any sink end. For example, in this case $\{B \mapsto \bot\}$ can be briefly denoted as $\varnothing$.

Parameterized Mealy machines can be seen as an abstraction of Mealy machines, and we still need a convert function between the two models.

Definition 2 (Concretize Mapping): We use $[\mathcal{PM}]_\Sigma$ to denote the concrete Mealy machine which is determined by an abstract parameterized Mealy machine $\mathcal{PM}$ and the alphabet Σ. Apparently $[\mathcal{PM}]_\Sigma$ can be described as

$$
\begin{aligned}
[\mathcal{PM}]_\Sigma =\ &\langle \mathcal{PM}.S(\Sigma), \mathcal{PM}.s_0, \\
&Input(\Sigma, \mathcal{PM}.I, \mathcal{PM}.O), Output(\Sigma, \mathcal{PM}.O), \\
&\mathcal{PM}.\delta(\Sigma), \mathcal{PM}.\lambda(\Sigma), \rangle
\end{aligned}
$$

Now we can use PMMs to specify timed Reo. Here we take the asynchronous channel (FIFO1), the synchronous channel (Sync) and the timed channel (Timer) as examples. For simplicity, we use $A = \_$ to indicate that there is no constraint on channel end A.

Example 2 (n-Timer): The PMM of an n-Timer with a source-end A and a sink-end B can be defined as:

- $S(\Sigma) = \{q_{i,d} | 0 \le i \le n, d \in \Sigma\} \cup \{q_0\}$
- $I = \{A\}, O = \{B\}, s_0 = q_0$
- output function

$$
\delta(\Sigma)(s,i) = \begin{cases}
\{B \mapsto d\} & s = q_{n,d} \land i = \{A \mapsto \_, B \mapsto \triangleright\} \\
\ominus & s = q_{n,d} \land \\
& (i = T \lor i = \{A \mapsto \_, B \mapsto \otimes\}) \\
\ominus & s = q_{j,d} \land 0 \le j < n \land \\
& (i = \{A \mapsto \_, B \mapsto \triangleright\} \lor \\
& i = \{A \mapsto d, B \mapsto \_\}) \\
\varnothing & otherwise
\end{cases}
$$

- transition function

$$
\lambda(\Sigma)(s,i) = \begin{cases}
q_{0,d} & s = q_0 \land i = \{A \mapsto d, B \mapsto \_\} \\
q_{j+1,d} & s = q_{j,d} \land i = T \land 0 \le j < n \\
q_0 & s = q_{n,d} \land i = \{A \mapsto \bot, B \mapsto \triangleright\} \\
q_{0,d'} & s = q_{n,d} \land i = \{A \mapsto d', B \mapsto \triangleright\} \\
q_0 & s = q_{n,d} \land i = T \\
s & otherwise
\end{cases}
$$

Similarly, we can use PMMs to describe the behavior of other basic timed Reo channels. Now we show how to compose these channels into more complicated connectors.

Definition 3 (Product): For two PMMs $\mathcal{PM}_1$ and $\mathcal{PM}_2$, if $\mathcal{PM}_2.O \cap \mathcal{PM}_1.I = \varnothing$, their product

$$\mathcal{PM}' = prod(\mathcal{PM}_1, \mathcal{PM}_2)$$

can be defined as follows.

- $\forall \Sigma.\mathcal{PM}'.S(\Sigma) = \mathcal{PM}_1.S(\Sigma) \times \mathcal{PM}_2.S(\Sigma)$
- $\mathcal{PM}'.I = (\mathcal{PM}_1.I \cup \mathcal{PM}_2.I) \setminus \mathcal{PM}_1.O$
- $\mathcal{PM}'.O = (\mathcal{PM}_1.O \cup \mathcal{PM}_2.O) \setminus \mathcal{PM}_2.I$

(Here we assume that sink ends of $\mathcal{PM}_1$ can be connected to source ends of $\mathcal{PM}_2$, but not vise versa)

- $\mathcal{PM}'.s_0 = (\mathcal{PM}_1.s_0, \mathcal{PM}_2.s_0)$
- $\forall \Sigma.\mathcal{PM}'.\delta(\Sigma)((s_1, s_2), i) =$

$$
\begin{cases}
(\bigcup_{j=1,2} Out_j) \downarrow_{\mathcal{PM}'.O} & i \ne T \land (\bigwedge_{j=1,2} Out_j \ne \ominus) \\
\varnothing & i = T \\
\ominus & otherwise
\end{cases}
$$

where we have

* $i \in Input(\Sigma, \mathcal{PM}'.I, \mathcal{PM}'.O)$
* $In_1 = i \downarrow_{\mathcal{PM}_1.I}$
* $In_2 = (Out_1 \cup i) \downarrow_{\mathcal{PM}_2.I}$
* $Out_j = \mathcal{PM}_j.\delta(\Sigma)(s_j, In_j)$ for $j = 1, 2$

- $\forall \Sigma.\mathcal{PM}'.\lambda(\Sigma)((s_1, s_2), i) = (s_1', s_2')$ where for $j = 1, 2$, $s_j' = \mathcal{PM}_j.\lambda(\Sigma)(s_j, In_j)$.

The idea in *product* is quite simple. The output of $\mathcal{PM}_1$ would be provided as part of the input of $\mathcal{PM}_2$. Timed action T is only executed simultaneously between $\mathcal{PM}_1$ and $\mathcal{PM}_2$.

As mentioned above, we cannot connect a sink end of $\mathcal{PM}_2$ to a source end of $\mathcal{PM}_1$, which makes it difficult to construct connectors like alternator in Fig. **??**. To solve the problem, we define a *link* operator to connect sink ends and source ends in the same connector.

Definition 4 (Link): The *link* operator constructs connector by connecting a sink end OUT to a source end IN within the same connector.

$$\mathcal{PM}' = link(\mathcal{PM}, \text{OUT}, \text{IN})$$

Here we assume that $\text{IN} \in \mathcal{PM}.I$ and $\text{OUT} \in \mathcal{PM}.O$.

- $\forall \Sigma. \mathcal{PM}'.S(\Sigma) = \mathcal{PM}.S(\Sigma)$
- $\mathcal{PM}'.I = \mathcal{PM}.I \backslash \{\text{IN}\}$
- $\mathcal{PM}'.O = \mathcal{PM}.O \backslash \{\text{OUT}\}$
- $\mathcal{PM}'.s_0 = \mathcal{PM}.s_0$
- $\forall \Sigma, i \in Input(\Sigma, \mathcal{PM}'.I, \mathcal{PM}'.O). \mathcal{PM}'.\delta(\Sigma)(s, i) =$

$$\begin{cases} \varnothing & i = T \\ \mathcal{PM}.\delta(\Sigma)(s, i \cup \{\text{IN} \mapsto d\}) \downarrow_{\mathcal{PM}'.O} & (*) \\ \ominus & otherwise \end{cases}$$

where the condition (*) is defined as

$$\exists d. \mathcal{PM}.\delta(\Sigma)(s, i \cup \{\text{IN} \mapsto d\})(\text{OUT}) = d$$

- $\forall \Sigma, i \in Input(\Sigma, \mathcal{PM}'.I, \mathcal{PM}'.O).$

$$\mathcal{PM}.\lambda(\Sigma)(s, i) = \begin{cases} \mathcal{PM}.\lambda(\Sigma, T) & i = T \\ \mathcal{PM}.\lambda(\Sigma, i \cup \{\text{IN} \mapsto d\}) & (*) \\ s & otherwise \end{cases}$$

With *prod* and *link* defined, connectors can be constructed by composing simpler ones in the form of PMMs. It can be transformed into a concrete Mealy machine later, once the alphabet Σ is provided.

III. FROM BLACKBOX TO TIMED CONNECTORS

In this section, we use L* algorithm to extract Reo coordinators from blackbox implementations in 3 steps: *constructing hypothesis*, *enclosing hypothesis* and *validating hypothesis*. The input and output symbols of the blackbox implementation are provided as $\mathcal{I}$ and $\mathcal{O}$, and the blackbox implementation is supposed to be equivalent to a Mealy machine where all states are reachable.

A. Observation Table

Observation Tables, proposed in [1], are used to construct hypothetical Mealy machines from the results of membership queries.

Mealy machines consist of *states* and *transitions*. As for blackboxes, input sequence $s \in \mathcal{I}^*$ can be used to denote its state after executing s. Similarly, for all $a \in \mathcal{I}$, we use sa to denote the successor state of s after executing a. However, under such notations, there are *infinite* number of states, among which most are duplicate.

States are distinguished, usually, by their *subsequent behavior*. Suppose $s_1 \neq s_2 \in \mathcal{I}^*$, we say the corresponding of states s_1 and s_2 are equivalent iff $\forall d \in \mathcal{I}^+. mq(s_1 d) = mq(s_2 d)$. Since we can never check it on every $d \in \mathcal{I}^+$, an alternative way is to use a set of *suffixes* to distinguish them approximately.

Definition 5 (D-Equivalence): Provided with a set of suffixes $D \subset \mathcal{I}^+$, the corresponding states of $s_1, s_2 \in \mathcal{I}^*$ are D-equivalent (denoted as $s_1 \sim_D s_2$) iff $mq(s_1 d) = mq(s_2 d)$ for all $d \in D$.

Now we use *access sequence* to describe the states. Given an input sequence s, the access sequence $acc(s, D)$ of s is defined as the shortest sequence in $\{s' \in \mathcal{I}^* | s' \sim_D s\}$[1].

Formally, an observation table (see Fig. **??** as an example) is determined by a tuple (S, D) where $S \subset \mathcal{I}^*$ is a set of the access sequences of states and $D \subset \mathcal{I}^*$ is a set of suffixes. In such tables, columns are labelled with suffixes in D, and rows are labelled with input sequences. As shown in Fig. **??**, rows are divided into two parts. The upper part, labelled with access sequence $s \in S$, denotes the states covered in the current hypothesis. The lower part, labelled with sequences in $\{sa | s \in S, a \in \mathcal{I}\} \backslash S$ (representing any potentially unclosed transition) denotes all the transitions outside $s \in S$. A cell with row label s and column label d is filled with $mq(sd)$.

An observation table (S, D) is *closed* if

$$\forall s \in S, a \in \mathcal{I}. \exists s' \in S. sa \sim_D s'$$

From a closed observation table *obs*, we can construct a hypothetical Mealy machine $\mathcal{H} = \langle S, s_0, \mathcal{I}, \mathcal{O}, \delta, \lambda \rangle$ in a straight-forward way:

- Every state in S corresponds to an input sequence in $obs.S$
- s_0 corresponds to the empty sequence ε
  (we use input sequences to indicate their corresponding states in following parts)
- $\delta(s, a) = mq(sa)$
- $\lambda(s, a) = acc(sa, obs.D)$. Since *obs* is closed, we always have $\lambda(s, a) \in S$.

Algorithm 1 shows how to build a closed observation table.

Algorithm 1: EncloseTable

Input: An oracle interface mq, An input actions $\mathcal{I}$, An observation table obs

Output: A closed observation table obs'

1 **repeat**
2    $next = \{st | \forall s \in obs.S, \forall t \in \mathcal{I}\} \backslash obs.S$;
3    $unclosed = \{seq \in next | \forall a \in obs.S, seq \not\sim_{obs.D} a\}$;
4    $obs'.S = obs.S \cup unclosed$;
5    $obs'.D = obs.D$;
6 **until** $unclosed = \varnothing$;
7 **return** obs';

Taking a 2-Timer channel as an example, we now illustrate how this algorithm works. We assume that the source end of this 2-Timer channel is A, the sink end is B, and the alphabet is $\{a\}$. We briefly denote $\{A \mapsto a, B \mapsto \otimes\}$ and other input/output actions in form of $(a, \otimes)$.

Firstly, $obs.S$ is initialized with the initial state (denoted by ε), and $obs.D$ is initialized with all one-step suffixes. Then we explore all the access sequences in $obs.S$ and calculate its successors. Here *unclosed* consists of access sequences that has no equivalent sequence in $obs.S$. For example, Fig. **??** shows the *obs* of a 2-Timer channel after the first iteration. The five successors of ε are presented at the bottom of the table,

---

[1] When there are multiple satisfying sequences with the same length, we use a certain tactic to pick up one from them determinstically.

where four are equivalent with ε but $(a, \otimes)$ is not. Therefore, we take $(a, \otimes)$ as a brand-new state and all of its successors need further exploration.

In the second iteration, we explore the successors of $(a, \otimes)$. Fortunately, now every successor has an equivalence sequence in $obs.S$, *i.e.*, itself. Consequently, the algorithm terminates with a closed hypothesis where all the unclosed edges turn into self-loop.

B. Counter Examples' Analysis

Apparently, the closed hypothesis presented in Fig. **??** is different with the 2-Timer channel. It's easy to find a counter-example $s_0 = (a, \otimes), T, T, (\bot, \triangleright)$ where $mq(s_0) = a$ while according to the hypothesis, the result should be $\varnothing$. In this section, we show how to find and analyze counter-examples using the method in [12].

Firstly, we give a formal definition of *counter examples*. With an observation table obs and a sequence s, we use $hq(obs, s)$ to denote the execution result of its corresponding hypothesis $\mathcal{H}$ under the input sequence s. Obviously we have $hq(obs, s) = mq(acc(s')a)$ where $s = s'a, a \in \mathcal{I}$.

Definition 6 (Counter Example): an input sequence s is a counter example of obs iff $mq(s) \neq hq(obs, s)$.

Now we consider the reason that leads to existance of counter examples. Since the suffix set D is used to distinguish states, the existance of counter-examples shows that the current suffix set D is not powerful enough to recognize the uncovered states in $next$ (see Algorithm 1).

For a counter example s, we need a new $\text{Suffix}(s, obs)$ to distinguish the uncovered state represented by s, which is defined as the longest sequence in

$$\{d \in \mathcal{I}^+ | s = s'd, mq(acc(s', obs.D)d) \neq mq(s)\}$$

After appending $\text{Suffix}(s, obs)$ to D, it's obvious that at least one uncovered state will be found.

The whole L* algorithm can be concluded as follows.

Algorithm 2: L*

Input: Oracle interfaces mq, eq, Input actions $\mathcal{I}$
Output: Observation table obs

1 $obs.S$ initialized as empty;
2 $obs.D$ initialized as $\{[i] | i \in \mathcal{I}\}$;
3 **repeat**
4     $obs = \text{EncloseTable}(mq, \mathcal{I}, obs)$;
5     $ce = eq(obs)$;
6   **if** $ce \neq true$ **then**
7         $obs.D.\text{append}(\text{Suffix}(ce, obs))$;
8 **until** $ce = true$;
9 **return** obs;

IV. EXPERIMENTS

Both *Reo Coordination Models* and *Adapted L* Algorithm* are implemented in Golang [6], which is widely known for its elegant design and impressive efficiency. Moreover, with a CSP-style [7] cocurrency model, `Golang` shares many similar ideas with Reo, and in turn makes our implementation much more natural.

All the following experiments are coded under Golang *1.2.1* and executed on a laptop with 8GB of RAM and a Core i7-3630 CPU. The source code is available at https://github.com/liyi-david/reo-learn.

A. Case Study

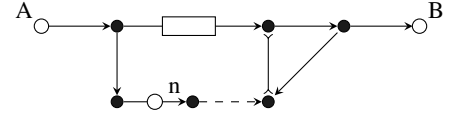A simple example of timed connector is presented to show how L* works.

Fig. 1. Expiring FIFO1 (ExpFIFO1)

Informally speaking, an expiring FIFO1 with *timeout* n is able to accept a data item and stored it in the buffer cell for n time units. If a read operation on B is performed within n time units, it will obtain the data item successfully and clear the buffer. Otherwise, the data item would be dropped if no read operation comes within n time units.
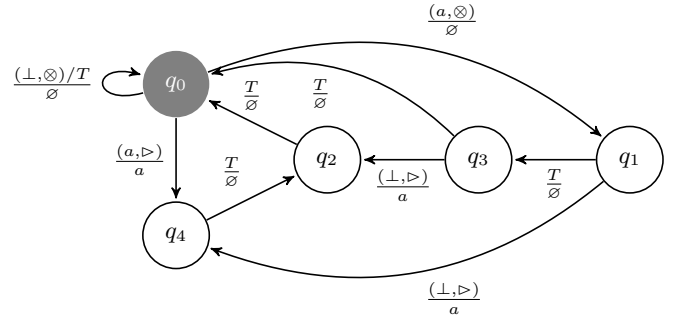

Fig. 2. Learn Result of The ExpFIFO1 where $n = 2, \Sigma = \{a\}$

Fig. 2 shows the learning result of this example where $n = 2$ and $\Sigma = \{a\}$. To simplify the graph, we ignore all the trivial transitions $\frac{(\bot, \otimes)}{\varnothing}$ and block transitions. More details of this case can be found in our *github repo*.

B. Performance Optimization

As a well-known learning algorithm, L* has proved its efficiency in models without time. However, when dealing with timed connectors, the algorithm failed to meet our expectation.

As shown in Table **??**, time consumption mainly comes from membership queries. With time involved, every single membership query takes a lot of time inevitably. After reviewing our algorithm, we found that simulations on similar sequences were invoked frequently:

- When constructing observation tables, there are lots of redundant calls to membership queries. For example, a

sequence with prefix 'aa' and suffix 'b' is exactly same as another one with prefix 'a' and suffix 'ab'.

- Simulation on Mealy machines can provide multi-step outputs. Consequently, if we have simulated an 'abc' sequence, it's useless to perform simulation on an 'ab' sequence again.

If previous simulation results are stored in a well-maintained cache, the time-cost in simulation process could be reduced signficiantly. In this work, we use a multiway tree to buffer these results.

With cache applied, we have made considerable reduction on the calls of membership queries. The results can be found in Table **??**.

V. Conclusion and Future Work

In this paper, we come up with an approach to extract timed connectors from blackbox implementations. We propose *parameterized Mealy machine* to describe the behavior of Reo connectors. Then we define the product operator and link operator, which can be used to construct complex connectors from simpler ones. Besides, we show how PMMs behave as the bridge between Reo connectors and concrete Mealy machines with timed action T provided.

We also adapt the well-known active learning algorithm L* to deal with time domain. By a tree-style cache, we make significant reduction on membership queries and, in turn, improve the performance of learning algorithm. As a by-product, we also encapsulate the Reo connector as a distributable package which could contribute to concurrent programming.

Our future work mainly focuses on better support of dense time. To describe dense time behavior, the Mealy machine model needs a lot of changes instead of a simple T action. Besides, we will also try to improve our algorithm to handle non-deterministic behavior. This is not a brand new topic [13], but we believe that there is still room for improvement.

References

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

[2] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[3] F. Arbab, C. Baier, F. S. de Boer, and J. J. M. M. Rutten. Models and temporal logics for timed component connectors. In *Proceedings of SEFM 2004*, pages 198–207. IEEE Computer Society, 2004.

[4] C. De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[5] N. S. Gill. Reusability issues in component-based development. *ACM SIGSOFT Software Engineering Notes*, 28(4):4, 2003.

[6] Google. Google go. https://golang.org/.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[8] S. Kemper. Sat-based verification for timed component connectors. *Science of Computer Programming*, 77(7-8):779–798, 2012.

[9] S. Li, X. Chen, Y. Wang, and M. Sun. A framework for off-line conformance testing of timed connectors. In *Proceedings of TASE 2015*, pages 15–22. IEEE Computer Society, 2015.

[10] S. Meng. Connectors as designs: The time dimension. In *Proceedings of TASE 2012*, pages 201–208. IEEE Computer Society, 2012.

[11] H. Raffelt and B. Steffen. Learnlib: A library for automata learning and experimentation. In *Proceedings of FASE 2006*, volume 3922 of *LNCS*, pages 377–380. Springer, 2006.

[12] B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In *Proceedings of SFM 2011*, volume 6659 of *LNCS*, pages 256–296. Springer, 2011.

[13] M. Volpato and J. Tretmans. Active learning of nondeterministic systems from an ioco perspective. In *Proceedings of ISoLA 2014, Part I*, volume 8802 of *LNCS*, pages 220–235. Springer, 2014.