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 distributed and concurrent systems. In this paper, we propose an active learning framework to extract timed connector models from black-box system implementation. We first introduce parameterized Mealy machine(PMM) as an operational semantic model for channel-based coordination language Reo. PMM serves as a bridge between Reo connectors and Mealy machines. With the product operator, complex connectors can be constructed by joining basic channels and the PMMs of connectors can be transformed into Mealy machines. Moreover, we adapt L*, a well-known learning algorithm, to timed connectors (in the form of Mealy machines). The new algorithm has shown its efficiency in multiple case studies. This framework has been implemented in `Golang`.**

***Index Terms*—Active Learning, Coordination Languages, Timed 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. Such systems could be distributed logically or physically, which makes coordination processes even more complicated. In this case, we need to specify the coordination processes with *coordination languages*, such that formal techniques can be applied to guarantee their reliability.

Timed Reo is a real-time extension of the coordination language *Reo*, which can be used to describe the coordination process in DRES clearly and intuitively. Different formal semantics have been proposed to specify the behavior of timed Reo. For example, an operational semantics based on *timed constraint automata* was raised by Baier et al. [3], where a variant of LTL was also proposed to describe the properties of timed connectors. In [15], a UTP design model was proposed to specify the behavior of timed connectors..

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 [14]. A SAT-based bounded model checking approach was adapted in [13] 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], [8], [18] show interesting examples where active learning is used to extract *Mealy machines* or *regular languages* without time domain.

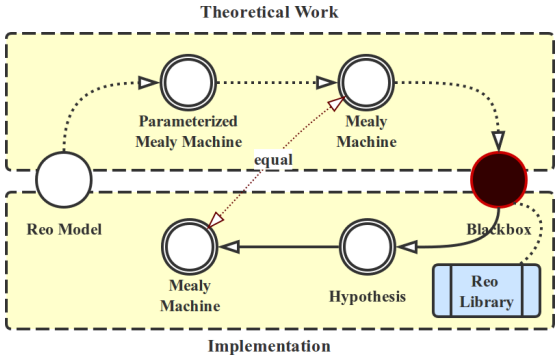

Fig. 1. The Active Learning Framework

In this paper we address this question by proposing an active learning framework as shown in Fig. 1 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 II we briefly illustrate some basic concepts, including the coordination language Reo, Mealy machine, and active automata learning. Section III defines an operational semantics of Reo, which is used in Section IV to show how to extract Reo models from blackboxes by means of active learning. In Section V, we discuss the implementation and optimization of the approach in `Golang`. Finally, Section VI concludes the paper.

II. PRELIMINARIES

A. Reo Coordination Language

We provide here a brief overview of the main concepts in Reo. Further details can be found in [2], [4].

Reo is a channel based exogenous coordination language proposed by F. Arbab in [2]. A coordinator in Reo, also called *connector*, provides the protocol that controls and organizes the communication, synchronization and cooperation among the components which communicate through the connector. Connectors can be defined without dependence on components, which makes Reo a powerful "glue language" in component-based development [10].

In Reo, complex connectors are made up of simpler ones. The atomic connectors are called *channels*, and each of them has two *channel ends*. There are two types of channel ends: *source* and *sink*. Source channel ends accept data into the channel, while sink channel ends dispense data out of the channel.

Fig. 2. Basic Reo Channels

Fig. 2 shows the graphical representation of some basic channel types in Reo, whose behavior are described as follows:

- A *Sync* channel accepts a data item from its source end iff the data item can be dispensed through its sink end simultaneously.
- A *LossySync* channel is always ready to accept data items. These items will be dispensed through its sink end simultaneously if possible, otherwise they will be lost.
- A *SyncDrain* channel has two source ends and no sink end. It can accept a data item through one of its source end iff a data item is also available to be accepted simultaneously through the other end. Then both items will be dropped.
- A *FIFO1* channel is an asychronous channel with a buffer cell. It accepts a data item whenever the buffer is empty.
- A t-timer channel accepts any data item from its source end, and later dispense it to its sink end after a delay of t time units. [1]

Channels are joint together in *nodes*. There are three types of nodes: *source*, *sink* and *mixed* nodes, depending on whether all channel ends that coincide on a node are source ends, sink ends or a combination of both. (see Fig. 3)

Components can be linked to source nodes or sink nodes. A component can write data items to its corresponding source node only if the data item can be dispensed simultaneously to all source ends on this node. Similarly, a component can read a data item only if there is at least one readable sink end on its corresponding node. A mixed node non-determinstically selects and takes a data item from one of its coincident sink ends and replicates it into all of its coincident source ends.

[1]Here the interpretation of t-Timer is a bit different as in other works like [3], [15], but we can easily transform between the two kinds of behavior by composing with some other basic channels.

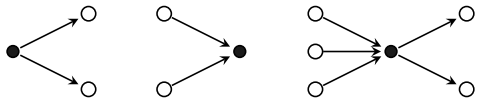

Fig. 3. Source, Sink and Mixed Nodes in Reo

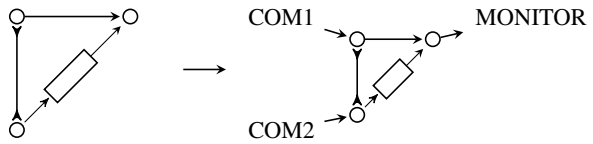

Fig. 4. An Example of Coordination with Reo Connectors

Fig. 4 provides a simple example to illustrate how complex connectors are constructed and used in coordination. In this example, we have three components COM1,COM2 and MONITOR. With an *alternator* connector as shown in Fig. 4, MONITOR can obtain data items from COM1 and COM2 alternately. This example has been implemented in `Golang` as shown in Section V-C. Besides, all the connectors are supposed to have *deterministic* behavior hereinafter to meet the requirement of active learning algorithm L*.

B. Mealy Machines

In this work, we use *Mealy machines* to model Reo connectors. Mealy machine was first proposed by George. H. Mealy in [9] as an extension of finite state machine. Compared with other variants, Mealy machines are designed to model reactive systems, where outputs of a machine are determined by not only its current state but also the current inputs. Besides, Mealy machines are supposed to be *input enabled*, which means that all possible inputs should be acceptable in all states. In other words, if an input is invalid for some state, we need to manually define an additional state to describe such exceptions.

Various forms of Mealy machines have been defined in the literature [5], [9], [20]. Since active automata learning requires the system-under-learn to be deterministic, in this paper we take the deterministic Mealy machine model as defined in [20].

Definition 1 (Mealy machine): A Mealy machine is a 6-tuple $(S, s_0, I, O, \delta, \lambda)$ consisting of:

- a finite set of states S,
- an initial state $s_0 \in S$,
- a finite set of inputs I,
- a finite set of outputs O,
- an output function $\delta : S \times I \to O$ mapping a pair of a state and an input to the corresponding output,
- a transition function $\lambda : S \times I \to S$ mapping a pair of a state and an input to the corresponding successor state.

We say that a Mealy machine is *finite* if the set S of states and the set I of inputs are both finite. Hereinafter, Mealy machines are always supposed to be finite in this paper.

C. Active Learning

In this section, we briefly introduce the main ideas of active automata learning.

Active learning [19] is a special case of semi-supervised machine learning where a learning algorithm is able to interactively query the target systems to obtain the desired outputs under certain inputs. By well-designed query strategy, active learning is able to obtain more accurate models with smaller dataset.

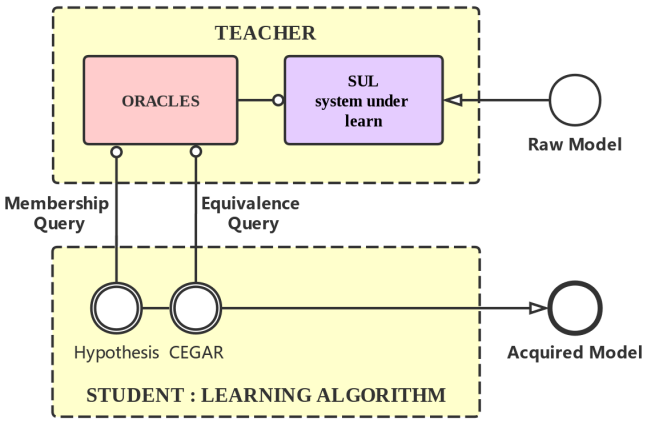

Fig. 5. Active Automata Learning

Fig. 5 shows the sketch of active learning, wherein:

- *Teacher* and *Student*: Active learning is an interactive process where students ask questions and teachers answer. Here learning algorithm plays the role of student.
- *Oracles* is an interface specifying which kind of questions can be answered by the teacher.
- *SUL* is an abbreviation of System Under Learn. In this paper, we take blackbox models as our SULs.
- *CEGAR* indicates Counter-Example Guided Abstraction Refinement [7]. In active learning, we need counter-examples to guide us on further queries and cover the undistinguished states.

When applying active automata learning on some model, we assume that the model should be equivalent to some *Mealy machine*, i.e., a deterministic model accepting a finite set of inputs and mapping them to a finite set of outputs.

Such a model is encapsulated as a *teacher* by the *Oracle*, which handles all communication with the model. The oracle also serves as a so-called *Minimal Adequate Teacher* interface, which is responsible for two types of queries.

- **Membership Query** (hereinafter referred to as *mq*) In grammar-learning [1], *mq* checks if a word is a member of certain language defined by the given grammar. When it comes to automata learning, *mq* is supposed to provide *simulation results* for given input sequences.
- **Equivalence Query** (hereinafter referred to as *eq*) Given a hypothesis (usually constructed by the learning algorithm), *eq* checks whether the hypothesis is equivalent to the system-under-learn and generates a counter-example if needed. Generally, *eq* is irrealizable when SUL is a blackbox. So we tactically use mq to achieve the approximate results.

These queries are given by *learning algorithms*, or so-called *students* in Fig. 5. From the *mq* results, a learning algorithm constructs a *hypothesis* and then check it with *eq*. If counter-examples are found, we turn back and repeat the hypothesis construction until the equivalence query returns *true*.

More details on the active automata learning algorithm will be presented in Section IV.

III. 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. Time Domain

Time has been investigated in several extensions of Reo. For example, timed Reo [3], [16], hybrid Reo [6], etc. Generally, these models are designed to handle real-time behavior where time is defined on the real number field $\mathbb{R}$. However, there are also some works like [17] where rational time indeed makes things easier. In this paper, we choose the rational number field $\mathbb{Q}$ as our time domain, which simplifies discretization of timed behavior greatly.

As presented in section II-A, all real-time behavior in timed Reo comes with a finite set of timed channels. We use $t_i \in \mathbb{Q}, i = 1, 2, \ldots, n$ to denote the delays of these timed channels, and then we can define a precision function $prec$ which calculate a gcd-style *maximal time precision*:

$$prec(t_1, \ldots, t_n) = \max\{T \in \mathbb{Q} | \forall t_i . \exists n_i \in \mathbb{N} . t_i = n_i \cdot T\}$$

It's easy to prove that such a T always exists.

In embedded systems, the maximal time precision is also called "clock-period", which is the basic time unit provided by an oscillator. For example, a widely-used Intel-80C51 microcontroller works under the frequency 16Mhz, where the clock-period is 62.5ns. A reasonable assumption comes that such a maximal time precision $dt = prec(t_1, \ldots, t_n)$ should always be provided to make sure different components are able to work together. Then all t_i-Timers can be seen as $n_i dt$-Timers for some n_i and we use n_i-Timers instead in the following sections.

Besides, we add a timed action "T" in Mealy machines. It indicates that the corresponding transition will take one time unit to finish.

B. 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 II, we define the *parameterized Mealy machine* as follows.

Definition 2 (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 $f \in Input(\Sigma, I, O) \backslash \{T\}$ we have

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

where we use $\perp$ 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 $f \in Output(\Sigma, O) \backslash \{\ominus\}$,

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

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

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

and its output actions

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

We use $\varnothing$ to denote the empty output when no data item is written to any sink end. For example, in this case $\{B \mapsto \perp\}$ 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 3 (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

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

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 (FIFO1): The PMM of a FIFO1 with source end A and sink end B can be defined as follows.

- $S(\Sigma) = \{q_0\} \cup \{q_d | d \in \Sigma\}$
- $I = \{A\}$, $O = \{B\}$, $s_0 = q_0$
- output function

$$\delta(\Sigma)(s, i) = \begin{cases} \varnothing & s = q_0 \wedge i = \{A \mapsto \_, B \mapsto \otimes\} \\ \ominus & s = q_0 \wedge i = \{A \mapsto d', B \mapsto \triangleright\} \\ \ominus & s = q_0 \wedge i = \{A \mapsto \perp, B \mapsto \triangleright\} \\ \varnothing & s = q_d \wedge i = \{A \mapsto \perp, B \mapsto \otimes\} \\ \{B \mapsto d\} & s = q_d \wedge i = \{A \mapsto \_, B \mapsto \triangleright\} \\ \ominus & s = q_d \wedge i = \{A \mapsto d', B \mapsto \otimes\} \end{cases}$$

- transition function

$$\lambda(\Sigma)(s, i) = \begin{cases} q_{d'} & s = q_0 \wedge i = \{A \mapsto d', B \mapsto \_\} \\ q_{d'} & s = q_d \wedge i = \{A \mapsto d', B \mapsto \triangleright\} \\ q_d & s = q_d \wedge i = \{A \mapsto \_, B \mapsto \otimes\} \\ q_0 & otherwise \end{cases}$$

Besides, the concrete Mealy machine (where $\Sigma = \{d\}$) is shown in Fig. 6, where labels of edges are in form of $\frac{input}{output}$. All $\ominus$ edges are ignored for clearance.

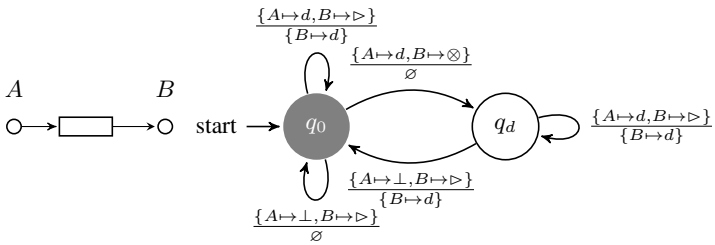

Fig. 6. PMM-based Semantics of $[FIFO1]_\Sigma$, where $\Sigma = \{d\}$

Example 3 (Sync): The PMM of a Sync with a source-end A and a sink-end B can be defined as:

- $S(\Sigma) = \{q_0\}$, $I = \{A\}$, $O = \{B\}$, $s_0 = q_0$

- output function

$$\delta(\Sigma)(s,i) = \begin{cases} \varnothing & i = \{A \mapsto \bot, B \mapsto \otimes\} \\ \{B \mapsto d\} & i = \{A \mapsto d, B \mapsto \rhd\} \\ \ominus & i = \{A \mapsto \bot, B \mapsto \rhd\} \\ \ominus & i = \{A \mapsto d, B \mapsto \otimes\} \end{cases}$$

- transition function $\lambda(\Sigma)(s,i) = q_0$.

Example 4 (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

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

- transition function

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

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 4 (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.

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

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

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

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

  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. 4. To solve the problem, we define a *link* operator to connect sink ends and source ends in the same connector.

Definition 5 (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 IN $\in \mathcal{PM}.I$ and 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.

IV. 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 reduplicate.

States are distinguished, usually, by their *subsequent behavior*. Suppose $s_1 \ne s_2 \in \mathcal{I}^*$, 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 6 (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 $\{s' \in \mathcal{I}^* | s' \sim_D s\}$[2].

Formally, an observation table (see Fig. 7 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. 7, 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'

```
repeat
    next = {st | ∀s ∈ obs.S, ∀t ∈ I} \ obs.S;
    unclosed = {seq ∈ next | ∀a ∈ obs.S, seq ⊀_{obs.D} a};
    obs'.S = obs.S ∪ unclosed;
    obs'.D = obs.D;
until unclosed = ∅;
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 $(a, \otimes)$.

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

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. 7 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, 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.

	$(a, \triangleright)$	$(\perp, \triangleright)$	$(a, \otimes)$	$(\perp, \otimes)$	T	acc
ε	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$	ε
$(a, \triangleright)$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$	ε
$(\perp, \triangleright)$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$	ε
$(a, \otimes)$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$unclosed$
$(\perp, \otimes)$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$	ε
T	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$	ε

Fig. 7. Observation Table and Corresponding Hypothesis

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.

	$(a, \triangleright)$	$(\perp, \triangleright)$	$(a, \otimes)$	$(\perp, \otimes)$	T	acc
ε	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$	ε
$(a, \otimes)$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$(a, \otimes)$
$(a, \triangleright)$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$	ε
$(\perp, \triangleright)$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$	ε
$(\perp, \otimes)$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$	ε
T	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$	ε
$(a, \otimes), (a, \triangleright)$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$(a, \otimes)$
$(a, \otimes), (\perp, \triangleright)$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$(a, \otimes)$
$(a, \otimes), (a, \otimes)$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$(a, \otimes)$
$(a, \otimes), (\perp, \otimes)$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$(a, \otimes)$
$(a, \otimes), T$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$(a, \otimes)$

Fig. 8. A Closed Version of Fig. 7

B. Counter Examples' Analysis

Apparently, the closed hypothesis presented in Fig. 8 is different with the 2-Timer channel. It's easy to find a counter-example $s_0 = (a, \otimes), T, T, (\perp, \triangleright)$ 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 [20].

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 7 (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 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 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

```
obs.S initialized as empty;
obs.D initialized as {[i]|i ∈ I};
repeat
    obs = EncloseTable(mq, I, obs);
    ce = eq(obs);
    if ce ≠ true then
        obs.D.append(Suffix(ce, obs));
until ce = true;
return obs;
```

V. Experiments

Both *Reo Coordination Models* and *Adapted L* Algorithm* are implemented in Golang [11], which is widely known for its elegant design and impressive efficiency. Moreover, with a CSP-style [12] 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.

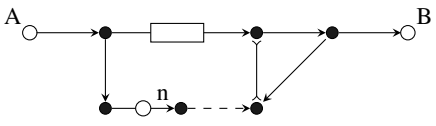

Fig. 9. Expiring FIFO1 (ExpFIFO1)

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.

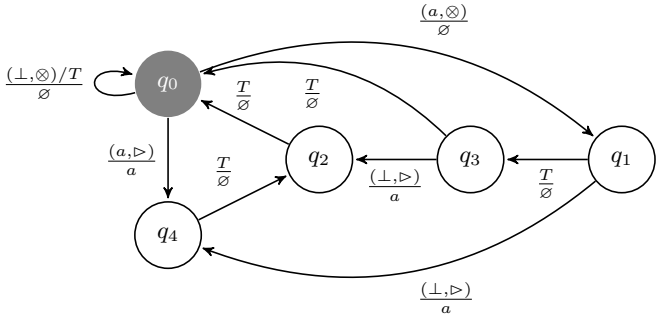

Fig. 10. Learn Result of The ExpFIFO1 where $n = 2, \Sigma = \{a\}$

Fig. 10 shows the learning result of this example where $n = 2$ and $\Sigma = \{a\}$. To simplify the graph, we ignore all the trivial transitions $\frac{(\perp,\otimes)}{\varnothing}$ 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.

TABLE I
Time-Cost Analysis

	FIFO1	Alternator	Gate
Membership Query(s)	41.571	126.468	169.161
Hypothesis Query(s)	0.001	0.003	0.004
Total Time(s)	41.715	165.114	247.098
Membership Query(%)	99.6	76.6	68.5

As shown in Table I, 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 II.

TABLE II
REDUCTION OF MEMBERSHIP QUERIES

	FIFO	Alternator	Gate
Original Algorithm	93	880	1034
Cached Algorithm	90	725	707
Reduction Rate	3.2%	21.4%	31.6%

C. An Example of the Reo Package

Our implementation in `Golang` is well-prepared not only for academic use but also for practical concurrent programming. The following code shows how to construct an alternator connector (see Fig. 4) in `Golang`. An intact version of this example can be found in our github repo.

```
1  func alternator(A, B, C Port) {
2    M := MakePorts(6)
3
4    // definition of channels
5    go ReplicatorChannel(A, M[0], M[1])
6    go ReplicatorChannel(B, M[2], M[3])
7    go MergerChannel(M[4], M[5], C)
8
9    go SyncdrainChannel(M[1], M[2])
10   go SyncChannel(M[0], M[4])
11   go FifoChannel(M[3], M[5])
12 }
```

Provided with the source and sink nodes, *alternator* function creates a series of basic channels and mixed nodes (named *Port*) to serve as the alternator connector we need. Now we can activate the components and using *alternator* function to coordinate them.

```
1 A,B,C := MakePorts(3)
2 alternator(A, B, C)
3
4 go sender(A, "MSG_A")
5 go sender(B, "MSG_B")
6 go monitor(C)
```

In this case, *senders* are goroutines (basic parallel units in `Golang`) that keep sending certain messages to some given port (A and B). A *monitor* keeps trying to read data items from the sink end C and print them on the screen. Finally, we have an interleaved sequence of "MSG_A" and "MSG_B".

VI. 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. When time is taken into consideration, the original L* algorithm run into a performance bottleneck. 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 [21], but we believe that there is still room for improvement.

ACKNOWLEDGMENTS

The work was partially supported by the National Natural Science Foundation of China under grant no. 61202069, 61272160 and 61532019.

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. Baier, M. Sirjani, F. Arbab, and J. J. M. M. Rutten. Modeling component connectors in reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.
[5] M. Broy. Verifying of interface assertions for infinite state mealy machines. *Journal of Computer and System Sciences*, 80(7):1298–1322, 2014.
[6] X. Chen, J. Sun, and M. Sun. A hybrid model of connectors in cyber-physical systems. In *Proceedings of ICFEM 2014*, volume 8829 of *LNCS*, pages 59–74. Springer, 2014.
[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of CAV 2000*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.
[8] C. De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.
[9] H. George. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.
[10] N. S. Gill. Reusability issues in component-based development. *ACM SIGSOFT Software Engineering Notes*, 28(4):4, 2003.
[11] Google. Google go. https://golang.org/.
[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
[13] S. Kemper. Sat-based verification for timed component connectors. *Science of Computer Programming*, 77(7-8):779–798, 2012.
[14] 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.
[15] S. Meng. Connectors as designs: The time dimension. In *Proceedings of TASE 2012*, pages 201–208. IEEE Computer Society, 2012.
[16] S. Meng and F. Arbab. On resource-sensitive timed component connectors. In *Proceedings of FMOODS 2007*, volume 4468 of *LNCS*, pages 301–316. Springer, 2007.
[17] P. Prabhakar, P. S. Duggirala, S. Mitra, and M. Viswanathan. Hybrid automata-based CEGAR for rectangular hybrid systems. *Formal Methods in System Design*, 46(2):105–134, 2015.
[18] 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.
[19] B. Settles. Active learning literature survey. Technical report, University of Wisconsin, Madison, 2010.
[20] 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.
[21] 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.