Active Learning from Blackbox to Timed Connectors

Yi Li*, Yiwu Wang* and Meng Sun*
*Department of Informatics, School of Mathematical Sciences, Peking University, Beijing, China
liyi_math@pku.edu.cn, yiwuwang@126.com, summeng@math.pku.edu.cn

***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 as an operational semantic model for channel-based coordination language Reo. Parameterized mealy machine serves as a bridge between Reo connectors and mealy machines. With the product operator, complex connectors can be constructed by joining basic channels and 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. Implementations of this framework is provided as a package in `Golang`.**

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

I. Introduction

Distributed real-time embedded system (DRES) is reforming our lives with the name *IoT*, the internet of things, wherein systems are usually composed of individual components and a middleware serving as a *connector*. Such systems could be distributed logically or physically, which makes the coordination even more complicated. In this case, we need to specify these coordination processes with so-called *coordination languages*, so that formal techniques can be applied to guarantee their reliability.

Timed Reo is a real-time extension of the coordination language *Reo*. With timeline involved, coordination process in DRES can be depicted clearly and intuitively. Different formal semantics are proposed to specify the behavior of timed Reo. For example, in [14], a UTP-based (*Unifying Theories of Programming*) semantics is provided to verify connectors as *designs*. An operational semantics based on *constraint automata* was raised by Baier et al., where a variant of LTL was also proposed to describe the properties of timed connectors.

Formal verification techniques have also been proved applicable in timed connectors. [13] has shown us a comformance testing method on timed connectors. Bounded model checking methods were also adapted in [12], based on SAT solvers. All these solutions seem practical and impressive, however, a common question is faced by most of them, which is: *How to obtain these formal models?*

Correctness of connectors is very much related to some low-level implementation details. For example, well-writen code may behave dramatically weird with an improper set of concurrency primitives. Such things happen frequently in embedded systems with different hardware platforms or operating systems. Consequently, manually modeling an existing connector seems rather unreliable, even with reference to its source code.

As a branch of *machine learning* technique, active learning offers a way to obtain models from low-level models. Works in [1], [7], [16] shows impressive examples where active learning is used to extract *Mealy machines* or *regular languages* without time domain.

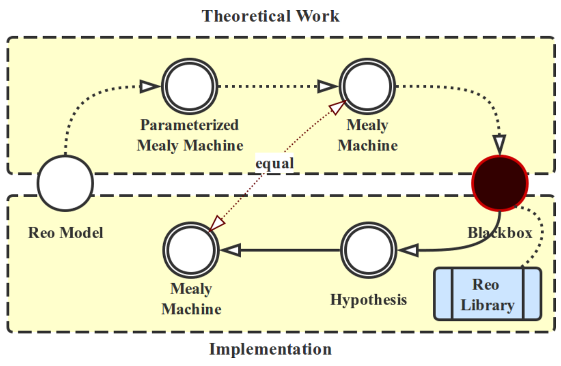

Fig. 1. The Active Learning Framework

In this paper, we proposed a active learning framework (shown in Figure 1) to automatically extract timed connectors from blackbox models. We present *Parameterized Mealy machine* as a parametierized semantics for timed Reo channels, which enables us to generate Mealy machines with a given alphabet. Then the $L*$ algorithm is adapted and optimized to extract Mealy machines with time action from the *connectors in blackboxes*.

The rest of the paper is organized as follows. After this general introduction, in Section II we briefly illustrate some basic concepts, including Reo the coordination language, Mealy machine, and active automata learning. Section III defines the Mealy-machine-based operational semantics of Reo, which is used in Section IV to show how to extract Reo models from blackboxes by means of active learning. Finally, in Section V, we discuss the optimization and implementation in `Golang`.

II. Preliminaries

A. Reo Coordination Language

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

Reo is a channel based exogenous coordination language proposed by F. Arbab in [2]. A Reo model, also called *connector*, provides the protocol that formalizes the communication, synchronization and cooperation among the components which communicate through the connector. Connectors can be defined with no knowledge of the components, which makes Reo a powerful "glue language" in component-based development [9].

In Reo, complex connectors are made up of simpler ones, where the atomic connectors are called *channels*. Each channel has two *channel ends*. There are two type of channel ends: *source* and *sink*. Source channel ends accepts data into the channel, while sink channels ends release them out of the channel.

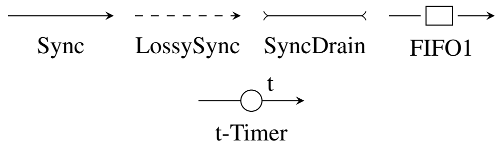

Fig. 2. Basic Reo Channels

The behavior of some channels are informally described as follows. (Graphical representations can be found in Figure 2).

- A *Sync* channel accepts a data item from its source end iff. the data item can be dispensed to its sink end simultaneously.
- A *LossySync* channel is always prepared to accept data items. These items will be send to its sink end simutaneously if possible, otherwise they will be dropped.
- 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 for it to simultaneously accept through the other end. Then both two data items will be lost.
- A *FIFO1* channel is an asychronous channel with one buffer cell. It accepts a data item whenever the buffer is empty.

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

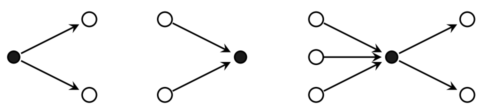

Fig. 3. Source, Sink and Mixed Nodes in Reo

With definition of more basic channels, it's easy to extend Reo to formalize coordination in different areas. In this paper, we take timed Reo [3] as our formal model. Timed Reo includes several timed channels, where the most commonly-used one, called *timer*, is also shown in Figure 2. 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.

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. Meanwhile, 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 channel ends and replicates it into all of its coincident source channel ends.

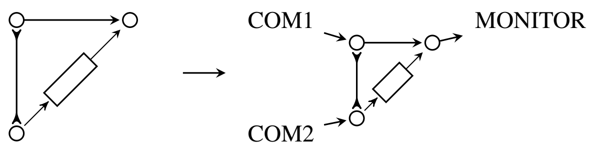

Fig. 4. Coordination with Reo Connectors

As shown in Figure 4, we use a simple example to illustrate how complex connectors are constructed and used in coordination. In this example, COM1,COM2 and MONITOR are components. With an *alternator* connector, MONITOR can receive data items from COM1 and COM2 alternately. The *alternator* example has been implemented in `Golang` as shown in Section V-C.

B. Mealy Machines

In this paper, we use *Mealy machine* to model SUTs. As an extension of *finite state machine*, Mealy machine was first proposed by George. H. Mealy in [8]. Compared with other variants, Mealy machines are designed to model reactive systems, where outputs are determined not only by 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 use an additional state to describe such exceptions.

As far as we can see, various forms of Mealy machines are defined in different works, wherein some are deterministic and some are not. Since active automata learning very much depends on the system-under-learn to be deterministic, in this paper we formally define a deterministic version of Mealy machine following [18].

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

- a finite set of states S
- a start state (also called initial state) s_0 which is an element of S
- a finite set called the input alphabet I
- a finite set called the output alphabet O

- a transition function $\delta : S \times I \to S$ mapping pairs of a state and an input symbol to the corresponding next state.
- an output function $\lambda : S \times I \to O$ mapping pairs of a state and an input symbol to the corresponding output symbol.

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

A Mealy machine is in some state $q \in S$ is always ready for inputs at any time. If an input symbol $i \in I$ is provided, the Mealy machine generates output $o = \lambda(q, i)$ and jumps to state $q' = \delta(q, i)$. Such an edge is also denoted as $q \xrightarrow{i/o} q'$.

C. Active Learning

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

Active learning [17] 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 on certain inputs. With such flexibility, active learning makes it able to use targeted and efficient queries to obtain more accurate models with smaller dataset.

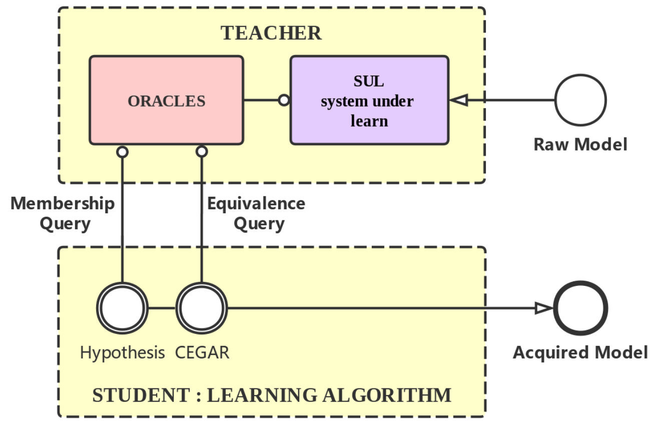

Fig. 5. Active Automata Learning

Figure 5 shows the sketch of active automata 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. The name comes from a well-known concept SUT (System Under Test) in software testing. In this case, we use blackbox models as our SULs.
- *CEGAR* indicates Counter-Example Guided Abstraction Refinement [6]. In active learning, we need counter-examples to guide us on further quries and cover the undistinguished states.

When applying active automata learning on some model, firstly we assume that it should be equivalent to some *Mealy machine*. In other words, a deterministic model accepting a finite set of input symbols and mapping them to a finite set of output symbols.

Such a model is encapsulated as a *teacher* by the *Oracle*, which handles all communication with the model, both providing input and obtaining output. Also, it serves as a so-called *Minimal Adequate Teacher* interface, which is responsible for two types of queries.

- **Membership Query** (hereinafter referred to as *mq*) The name comes from some grammar-learning papers (e.g. [1]), where *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** (hereinaftr 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, *equivalence query* is irrealizable when SUT is a blackbox. So we tactically use membership queries to achieve the approximate results.

These queries are given by *learning algorithms*, or so-called *students* in Figure 5. From the *mq* results, a learning algorithm should construct a *hypothesis* and 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 show how a timed connector be transformed into a Mealy machine with time action.

Since time is not involed in original Mealy machine, we first discuss how to formalize the time dimension in Mealy machine and timed Connectors. After that, we present a *parameterized Mealy machine* as a bridge between connectors and Mealy machines.

A. Time Domain

Time is involved in several extension version of Reo. For example, Timed Reo [3], Hybrid Reo [5], etc. Generally, these models are designed to handle real-time behavior where time is defined in $\mathbb{R}$. Besides, we also found some works like [15] 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 behaviors greatly.

As presented in section II-A, all real-time behavior in timed Reo comes with the *t-timer* channels, and the number of these channels are apparently finite. We use $t_i \in \mathbb{Q}$ to denote the delays of these timer channels, and now we can define a precision function *prec*.

$$prec(t_1, \cdots, t_n) = \max_T \{\forall t_i.\exists n_i \in \mathbb{N}.t_i = n_i \cdot T\}$$

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

In real systems, the concept *time precision* is widely used with the name "clock-period". Most of the time, we know the clock-cycle of some hardware components, even without any idea of its structure. With such precision T given, it's reasonable to assume that all t-timers are actually nT-timers. In following sections, we'll use n-timers instead.

Besides, we're going to add a "T" action in mealy machines. It indicates that a transition will take a time unit to finish, and all outputs would come out after that.

B. Parameterized Mealy Machine

We present a model named *parameterized Mealy machine* (hereinafter referred to as PMM) to represent timed connectors. PMM is supposed to behave as a middle representation. Connectors are defined as parameterized mealy machines, and composed via its production operator. Then original mealy-machine model will be taken as semantics of PMM model.

Following the formal definition of Mealy machine in Section II, we present the definition of *parameterized Mealy machine* as follows.

Definition 2 (Parameterized Mealy Machine): A *Parameterized Mealy Machine* is defined as a 6-tuple $\mathcal{PM} = \langle S(\Sigma), s_0, I, O, \delta(\Sigma), \lambda(\Sigma)\rangle$ where

- Σ is a *finite* datum alphabet (hereinafter referred to as an alphabet)
- S is a function that maps an alphabet to a *finite* set of states. We use $S(\Sigma)$ to denote the state set.
- I is a finite set of source-ends.
- O is a finite set of sink-ends.
- s_0 is the initial state. It satisfies $\forall\Sigma, s_0 \in S(\Sigma)$
- δ maps a *finite* datum alphabet to an *output function*. We use $\delta(\Sigma) : S(\Sigma) \times Input(\Sigma, I, O) \rightarrow Output(\Sigma, I, O)$ to denote the output function.
- λ maps an alphabet to a *transition function*. We use $\lambda(\Sigma) : S(\Sigma) \times Input(\Sigma, I, O) \rightarrow S(\Sigma)$ to denote the transiton function.

In the definition above, *Input* and *Output* are used to generate a set of input actions and output actions from the corresponding alphabets and ends. The value of $Input(\Sigma, I, O)$ is defined as a set of functions with on $I \cup O$ where $forall f \in Input(\Sigma, I, O)$, we have

$$\forall i \in I, f(i) \in \Sigma \cup \{\bot\} \wedge \forall o \in O, f(o) \in \{\rhd, \oslash\}$$

where we use $\rhd$ to indicate that a end-sink is ready for write, and $\oslash$ otherwise. Note that a $\bot$ means that there is *no* data items on a source (or sink) end. In the same way, $Output(\Sigma, I, O)$ is defined as a set of functions on domain O, with an additional symbol $\ominus$ indicating an input failure. Similarly, $\forall f \in Output(\Sigma, I, O), f \neq \ominus$ implies

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

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

$$\begin{aligned} Input(\{d_0\}, \{A\}, \{B\}) &= \{\{A : d_0, B : \rhd\}, \{A : d_0, B : \oslash\}, \\ &\quad \{A : \bot, B : \rhd\}, \{A : \bot, B : \oslash\}, T\} \end{aligned}$$

and its output actions

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

If there no data item written to all sink ends, we use $\varnothing$ to denote the empty output. For example, in the case given above, $\{B : \bot\}$ can be briefly rewritten as $\varnothing$.

Parameterized Mealy machines can be seen as an abstract form of Mealy machines, which means that 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

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

Now we can use Parameterized Mealy Machines to define a new semantics for timed Reo. Here we take one asynchronous channel (FIFO1), one asynchronous channel (Sync) and one timed channel (Timer) as examples.

Example 2 (PMM Semantics of FIFO1 channel): The semantics of FIFO channel 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

$$\delta(\Sigma)(s, i) = \begin{cases} (B : \bot) & s = q_0 \wedge i = (A : \_, B : \oslash) \\ (B : \bot) & s = q_d \wedge i = (A : \_, B : \oslash) \\ (B : d) & s = q_0 \wedge i = (A : d, B : \rhd) \\ (B : d) & s = q_d \wedge i = (A : \_, B : \rhd) \\ \ominus & s = q_d \wedge i = (A : d, B : \oslash) \\ \ominus & s = q_0 \wedge i = (A : \bot, B : \rhd) \end{cases}$$

- transition function

$$\lambda(\Sigma)(s, i) = \begin{cases} q_d & s = q_0 \wedge i = (A : d, B : \oslash) \\ q_d & s = q_{d'} \wedge i = (A : d, B : \rhd) \\ q_0 & otherwise \end{cases}$$

Besides, the concrete mealy machine (where $\Sigma = \{a\}$) is shown in Figure 6, where labels of edges are in form of $\frac{input}{output}$.

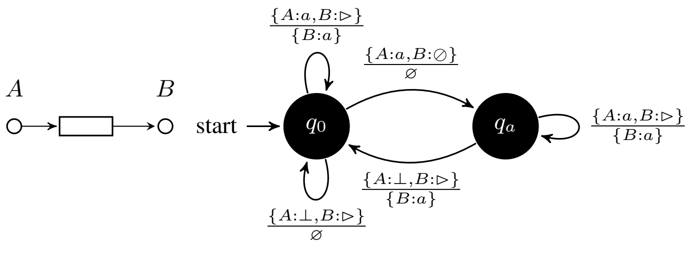
A
B
start
q_0
q_a
$\frac{\{A:a,B:\rhd\}}{\{B:a\}}$
$\frac{\{A:a,B:\oslash\}}{\varnothing}$
$\frac{\{A:a,B:\rhd\}}{\{B:a\}}$
$\frac{\{A:\bot,B:\rhd\}}{\{B:a\}}$
$\frac{\{A:\bot,B:\rhd\}}{\varnothing}$

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

Example 3 (PMM Semantics of Sync channel): The PMM-based semantics of a Sync channel 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

$$\delta(\Sigma)(s,i) = \begin{cases} (B:\bot) & s = q_0 \land i = (A:\bot, B:\oslash) \\ (B:d) & s = q_0 \land i = (A:d, B:\triangleright) \\ \ominus & s = q_0 \land i = (A:\bot, B:\triangleright) \\ \ominus & s = q_0 \land i = (A:d, B:\oslash) \end{cases}$$

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

Example 4 (PMM Semantics of n-Timer channel): Considering a n-Timer channel with a source-end A and a sink-end B, we define its PMM-based semantics as:

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

$$\delta(\Sigma)(s,i) = \begin{cases} (B:d) & s = q_{n,d} \land i = (A:d, B:\triangleright) \\ \ominus & s = q_{n,d} \land \\ & (i = T \lor i = (A:\_, B:\oslash) \\ \ominus & s = q_{j,d} \land 1 \leq j < n \land \\ & (i = (A:\_, B:\triangleright) \lor \\ & i = (A:d, B:\_)) \\ (B:\bot) & otherwise \end{cases}$$

- transition function

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

Similarly, we can use parameterized mealy machines to define the semantics of other basic timed Reo channels. Now we're going to show how to compose these channels into complicated connectors.

check the timed behavior

Definition 4 (Production Operator): Now we're going to define the production operator *prod* of two parameterized Mealy machines as,

$$prod(PM_1, PM_2) = PM_3$$

as follows. Here we assume that $PM_2.O \cap PM_1.I = \varnothing$

- $\forall \Sigma, PM_3.S(\Sigma) = PM_1.S(\Sigma) \times PM_2.S(\Sigma)$
- $PM_3.I = PM_1.I \cup PM_2.I - PM_1.O$
- $PM_3.O = PM_1.O \cup PM_2.O - PM_2.I$

(Here we assume that sink ends of PM_1 can be connected to source ends PM_2, but not vise versa)

- $PM_3.s_0 = (PM_1.s_0, PM_2.s_0)$
- $\forall \Sigma, PM_3.\delta(\Sigma)((s_1,s_2),i) =$

  $$\begin{cases} (Out_1 + Out_2)|_{PM_3.O} & Out_1 \neq \ominus \land Out_2 \neq \ominus \\ \ominus & otherwise \end{cases}$$

  where we have
  - $In_1 = i|_{PM_1.I}$
  - $Out_1 = PM_1.\delta(\Sigma)(s_1, In_1)$
  - $In_2 = (Out_1 + i)|_{PM_2.I}$
  - $Out_2 = PM_2.\delta(\Sigma)(s_2, In_2)$
- $\forall \Sigma, PM_3.\lambda(\Sigma)((s_1,s_2),i) = (s_1', s_2')$ where we have
  - $s_1' = PM_1.\lambda(\Sigma)(s_1, In_1)$
  - $s_2' = PM_2.\lambda(\Sigma)(s_2, In_2)$

The idea in *production* is quite simple. The output of PM_1 would be provided as part of the input of PM_2. Time actions T is only executed simutaneously betwenn PM_1 and PM_2.

As mentioned in the notation above, we cannot connect a sink end of PM_2 and a source end of PM_1, which makes it difficult to construct connectors like alternator in Figure 4. Now we define a *link* operation to connect sink ends and source ends in the same connector.

Definition 5 (Link Operator): The *link* operation constructs connector by linking a sink end OUT to a source end IN within the same connector.

$$PM' = link(PM, OUT, IN)$$

Here we asuume that $IN \in PM.I$ and $OUT \in PM.O$.

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

  $$PM.\delta(\Sigma)(s,T)|_{PM'.O} \tag{1}$$

  if $i = T$. When i is a non-time action, we need to check whether there's a data item $d \in \Sigma$ satiefies

  $$PM.\delta(\Sigma)(s, i \cup \{IN:d\})|_{OUT} = d$$

  if *true*, we have $PM'.\delta(\Sigma)(s,i) =$

  $$PM.\delta(\Sigma)(s, i \cup \{IN:d\})|_{PM'.O}$$

  otherwise $PM'.\delta(\Sigma)(s,i) = \ominus$
- $\forall \Sigma, i \in Input(\Sigma, PM'.I, PM'.O).'$ the transition function can be defined as $PM.\lambda(\Sigma)(s,i) =$

  $$\begin{cases} PM.\lambda(\Sigma, T) & i = T \\ PM.\lambda(\Sigma, i \cup \{IN:d\}) & \exists d \in \Sigma \text{ satisfes Equation 1} \\ s & otherwise \end{cases}$$

With *link* and *prop* defined, complex connectors can be formed by simple channels in form of *parameterized mealy machines*. It will be transformed into a concrete Mealy machine once the alphabet Σ is provided.

IV. From Blackbox to Timed Connectors

In this section, we will show how to use L* algorithm to extract models from blackboxes, which mainly includes 3 steps: *constructing hypothesis*, *enclosing hypothesis* and *checking hypothesis*. Besides, we assume that input symbols and output symbols of the blackbox have been provided as $\mathcal{I}$ and $\mathcal{O}$.

A. Observation Table

As mentioned above, membership queries provide us information with the form $mq(s) = o$, where s is an input sequence in Σ^+. Now the question is, *how to find the connection between such query results and Mealy machines?*

Generally, Mealy machines are composed of *states* and *transitions*. Provided with a Mealy machine m, any input sequence in $m.I^+$ leads to a unique state. Such a sequence is called an *access sequence* of this state. Specially, we use ε, the *empty sequnce*, to denote the initial state. Similarly, if a blackbox model has the same bahavior with a Mealy machine, we can use access sequences to label its states and, in the same way, input actions to label its transitions. For example, we consider a Mealy machine stays is staying in some state labelled by s. With an action a provided, the Mealy machine will jump to a new state labelled by $s' = sa$ and generate a output $mq(sa)$.

Different access sequences may lead to similar states. For example, a 2-Timer channel will dispense any accepted data item in 2 time units. After that, the state of this channel is exactly same as its initial state. Here *suffixes* are used to distinguish different states as shown in the following definition, where D is initialized as $\mathcal{I}$. Under such a suffix set, two states are considered different only when they have different one-step behaviors.

Definition 6 (Equivalence under Suffixes): provided with two access sequences s_1, s_2 and a set of suffixes $D \subset \mathcal{I}^+$, we say the corresponding states of s_1 and s_2 are equivalent under D (denoted as $s_1 \sim_D s_2$) iff $\forall d \in D, mq(s_1 d) = mq(s_2 d)$.

Since all this *inputs* and *outputs* can be observed outside the model, which means, even facing a blackbox we can use the same notations to represent it as a hypothetical Mealy machine.

Following this idea, *Observation Tables*, proposed in [1], shows a tabular form of such hypothetical models. In observation tables, rows represent states with label of their access sequences and columns are labelled suffixes. Successors of a state s is, intuitively, denoted as extended access sequences sa where a is a single input action. A cell with column-label d and row-label s should contain the value $mq(sd)$. Algorithm 1 shows how to build such an observation table.

Algorithm 1: BuildTable

```
  Input: Oracle interface $mq$, Input actions $\mathcal{I}$, suffix set $D$
  Output: Observation table $obs$
1 $obs$ initialized as empty;
2 $unclosed = \{\varepsilon\}$;
3 repeat
4     $next = \{st | \forall s \in unclosed, \forall t \in \mathcal{I}\}$;
5     append $unclosed$ to $obs$;
6     $unclosed = \{seq \in next | \forall a \in obs, seq \not\sim_D a\}$;
7 until $unclosed = \varnothing$;
8 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 $\{A : a, B : \oslash\}$ and other input/output actions in form of $a, \oslash$.

Firstly, *obs* is initialized as empty and the empty sequence ε is pushed in $unclosed$. Then we will explore all the access sequences in $unclosed$ and calculate its successors. Here the $unclosed$ set consists of access sequences that has no equivalent fellow with less length. For example, figure 7 shows the *obs* of a 2-Timer channel after the first iteration. The five successors of ε is presented as the bottom part of the table, where four are equivalent with ε but $a, \oslash$ is not. Therefore, we take $a, \oslash$ as a brand-new state and all of its successors need further exploration (see in the hypothesis Mealy machine presented in the following figure).

	$a, \rhd$	$\perp, \rhd$	$a, \oslash$	$\perp, \oslash$	T
ε	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$
$a, \rhd$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$
$\perp, \rhd$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$
$a, \oslash$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$
$\perp, \oslash$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$
T	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$

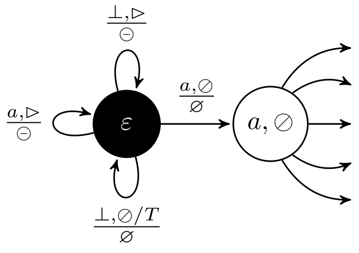
$\frac{\perp,\rhd}{\ominus}$
$\frac{a,\rhd}{\ominus}$
ε
$\frac{a,\oslash}{\varnothing}$
$a, \oslash$
$\frac{\perp,\oslash/T}{\varnothing}$

Fig. 7. Observation Table and Corresponding Hypothesis

In the second iteration, we explore the successors of $a, \oslash$. Fortunately we find that every successor has a shorted equivalence fellow (*itself*). Consequently, the algorithm terminates with a closed hypothesis where all the unclosed edge shown in Figure 7 turn into self-loop.

B. Counter-Examples' Analysis

In our framework, the oracle function eq is used to check the equivalence between hypothesis models and blackboxes. It is also responsible for generating counter-examples. Generally, as mentioned in [1] [18]. equivalence query has been proved impossible in blackbox models Nevertheless,

Apparently, the closed hypothesis presented in Figure 8 is not equal to the 2-Timer channel. It's easy to find a counter-example $s = a, \oslash - T - T - \perp, \rhd$ where $mq(s) = a$ while according to the hypothesis, the result is $\varnothing$.

As mentioned above, we use a suffix set D to distinguish different states. It's obvious that if two states has different behavior

	$a,\rhd$	$\bot,\rhd$	$a,\oslash$	$\bot,\oslash$	T
ε	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$
$a,\rhd$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$
$\bot,\rhd$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$
$a,\oslash$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$
$\bot,\oslash$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$
T	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$	$\varnothing$
$a,\oslash - a,\rhd$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$
$a,\oslash - \bot,\rhd$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$
$a,\oslash - a,\oslash$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$
$a,\oslash - \bot,\oslash$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$
$a,\oslash - T$	$\ominus$	$\ominus$	$\ominus$	$\varnothing$	$\varnothing$

Fig. 8. A Closed Observation Table

V. Experiments

Both *Reo Coordination Models* and *Adapted L* Algorithm* are implemented in Golang [10].

Golang (or Google Go) is a rising programming language started by Google Inc. The language is widely known for its elegant design and impressive efficiency. Moreover, the concurrency model of Golang comes from CSP [11]. As a channel-based model, CSP shares a similar idea with Reo and makes our implementation much more natural.

We have programmed Reo channels as a new package in Golang. The package is well-written for not only formal verification but also practical use.

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 Studies

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

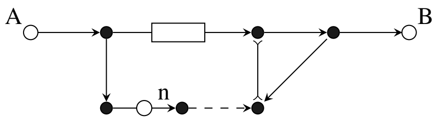

Fig. 9. Expiring FIFO1 Channel (ExpFIFO1)

Informally speaking, an expiring FIFO1 channel with *time-out* 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. However, the data item would be dropped if no read operation comes.

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

	FIFO	Alternator	Gate	ExpFIFO1
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. Since time is involved in our model, it's inevitably that simulation takes time to behave normally. Even worse, since our models are treated as blackbox. With no access to inner behaviour of the connectors, it's almost impossible to accelerate the simulation process.

Fortunately, there are still other optimization solutions. After reviewing our algorithm, we found that simulations on similar sequences were invoked frequently:

- When constructing *Obs* 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 output. Consequently, if we has simulated an 'abc' sequence, there's no reason to perform simulation on an 'ab' sequence.

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.

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%

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

C. An Example of the Reo Package

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

```
package main

import . './lib/reo'

func alternator(A, B, C Port) {
  // definition of ports
  M0 := MakePort()
```

```
    // M1 ... M5 defined similarly

    // definition of channels
    /* NOTE
    go function() means that the function would
    be executed as a new parallel task
    */
    go ReplicatorChannel(A, M0, M1)
    go ReplicatorChannel(B, M2, M3)
    go MergerChannel(M4, M5, C)

    go SyncdrainChannel(M1, M2)
    go SyncChannel(M0, M4)
    go FifoChannel(M3, M5)
}
```

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.

```
A := MakePort()
B := MakePort()
C := MakePort()
alternator(A, B, C)

go sender(A, "MSG␣A")
go sender(B, "MSG␣B")
go monitor(C)
```

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

VI. Conclusion and Future Work

Future work should focus on better representation on timed connectors and implementation of equivanlence query.

References

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 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 *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), 28-30 September 2004, Beijing, China*, 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. *Sci. Comput. Program.*, 61(2):75–113, 2006.

[5] X. Chen, J. Sun, and M. Sun. A hybrid model of connectors in cyber-physical systems. In S. Merz and J. Pang, editors, *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, volume 8829 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2014.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[7] W. Daelemans. Colin de la higuera: Grammatical inference: learning automata and grammars - cambridge university press, 2010, iv + 417 pages. *Machine Translation*, 24(3-4):291–293, 2010.

[8] H. George. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5):1045–1079, 1955.

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

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

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

[12] S. Kemper. Sat-based verification for timed component connectors. *Sci. Comput. Program.*, 77(7-8):779–798, 2012.

[13] S. Li, X. Chen, Y. Wang, and M. Sun. A framework for off-line conformance testing of timed connectors. In *2015 International Symposium on Theoretical Aspects of Software Engineering, TASE 2015, Nanjing, China, September 12-14, 2015*, pages 15–22. IEEE Computer Society, 2015.

[14] S. Meng. Connectors as designs: The time dimension. In T. Margaria, Z. Qiu, and H. Yang, editors, *Sixth International Symposium on Theoretical Aspects of Software Engineering, TASE 2012, 4-6 July 2012, Beijing, China*, pages 201–208. IEEE Computer Society, 2012.

[15] 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.

[16] H. Raffelt and B. Steffen. Learnlib: A library for automata learning and experimentation. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings*, volume 3922 of *Lecture Notes in Computer Science*, pages 377–380. Springer, 2006.

[17] B. Settles. Active learning literature survey. *University of Wisconsin, Madison*, 52(55-66):11, 2010.

[18] B. Steffen, F. Howar, and M. Merten. Introduction to active automata learning from a practical perspective. In M. Bernardo and V. Issarny, editors, *Formal Methods for Eternal Networked Software Systems - 11th International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM 2011, Bertinoro, Italy, June 13-18, 2011. Advanced Lectures*, volume 6659 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 2011.

Todo list