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* [4], [10] is used in this paper to formalize connectors in DRES. Reo is a channel based exogenous coordination language proposed by F. Arbab in [3], which is a powerful "glue language" in component-based development [6].

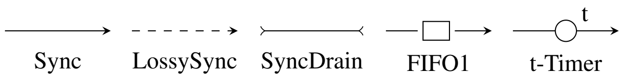

Fig. 1. Basic Reo Channels

Formal verification and validation techniques have also been proved applicable for timed connectors [8], [9] However, most of these tools heavily rely on manually modelling. *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.

In this paper we address this question by means of active learning [2], [12]. First of all, we introduce *parameterized Mealy machine* as a parameterized semantics for timed Reo, which can be transformed to concrete Mealy machines with a specified alphabet. Then the $L*$ algorithm [2] 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 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. Due to the limit of space, many technical details are omitted in this paper. Please refer to [1] for further contents.

II. Timed Connectors as Mealy Machines

In this section, we introduce PMM as an intermediate form to transform timed connectors into Mealy machines with timed action. In this paper, we assume that time dimension is defined on the national number $\mathbb{Q}$, which is generally as expressive as $\mathbb{R}$ and much easier to formalize [11]. Since connectors contain only finite number of channels, we can always find a minimal time unit (denoted as T) where real-time connectors can be discretized.

A. Parameterized Mealy Machine

In this work, we use *Mealy machines* to model Reo connectors. Mealy machine is an extension of finite state machine to model reactive systems. Following its definition in [14], we present a model here named *parameterized Mealy machine* to formalize timed connectors.

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 $\delta(\Sigma) : S(\Sigma) \times Input(\Sigma, I, O) \rightarrow Output(\Sigma, O)$ to denote the output function,
- λ maps an alphabet to a *transition function*. We use $\lambda(\Sigma) : S(\Sigma) \times Input(\Sigma, I, O) \rightarrow S(\Sigma)$ 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 the intersection of the set of input events Evt_{in} and an additional *time action* T. Here every $in \in Evt_{in}$ is an evaluation on both sink and source ends and

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

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 output events in form of evaluations on O, with an additional symbol $\ominus$ indicating an input failure.

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

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

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

Parameterized Mealy machines can be seen as an abstraction of Mealy machines, hence 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 timed channel (Timer) as an example. In timed Reo, a n-timer accepts a data item from its source end, and then put it to its sink end after n time units. Throughout the duration, the channel cannot accept any more data items. 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:

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

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

- transition function

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

Similarly, we can use PMMs to describe the behavior of other basic timed Reo channels. In Reo, as mentioned before, we compose complex connectors with simpler ones and basic channels recursively. Operator *prod* and *link* are used to apply such composing operation.

Definition 3 (prod): For two PMMs $\mathcal{PM}_1$ and $\mathcal{PM}_2$, we use the *prod* operator $\mathcal{PM}' = prod(\mathcal{PM}_1, \mathcal{PM}_2)$ to denote their production, where every sink-end in $\mathcal{PM}_1$ is connected with it's homonymic source-end in $\mathcal{PM}_2$.

Definition 4 (link): For a PMM $\mathcal{PM}$, we use the *link* operator to connect its sink-end OUT with its source-end IN, formally denoted as $\mathcal{PM}' = link(\mathcal{PM}, \mathrm{OUT}, \mathrm{IN})$.

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 provide a brief introduction on the well-known L* algorithm, which is used 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. An Overview of Active Learning

Active learning [13] 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 data-set.

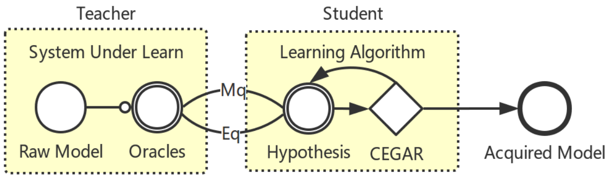

Fig. 2. Active Automata Learning

Fig. 2 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 [5]. 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*. 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* [2] interface, which is responsible for two types of queries.

- **Membership Query** (hereinafter referred to as *mq*) In grammar-learning [2], *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.

These queries are given by *learning algorithms*, or so-called *students* in Fig. 2. 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*.

B. L* Algorithm

The whole L* algorithm can be briefly concluded as follows. In this algorithm,

- *Observation table* is a widely-used data structure to describe the hypothesis. In such tables, rows correspond to a states (denoted as S), and columns correspond to *suffixes* (denoted as D).
- *Suffixes* are used to distinguish different states. For example, two input sequences a and b lead to the same state iff. $\forall c \in \mathcal{I}^*, mq(ac) = mq(bc)$.

Please refer to [2], [14] for further details.

Algorithm 1: L*

```
Input: Oracle interfaces $mq, eq$, Input actions $\mathcal{I}$
Output: Observation table $obs$
$obs.S$ initialized as empty;
$obs.D$ initialized as $\{[i] | i \in \mathcal{I}\}$;
repeat
    $obs$ = EncloseHypothesis($mq$, $\mathcal{I}$, $obs$);
    $ce = eq(obs)$;
    if $ce \neq true$ then
        $obs.D$.append(Suffix($ce, obs$));
until $ce = true$;
return $obs$;
```

The L* algorithm takes a set of input actions $\mathcal{I}$ and a oracle interface (which plays the role of SUL) as its parameters. And it returns a so-called *observation table* as the final hypothesis. (Observation table is a widely-used data structure, which can represent a Mealy machine by its external observation.) In each iteration, the algorithm will:

1) *Enclose the hypothesis.* Active learning is very similar to a breadth-first-search process on an unknown Mealy machine. At first, only the initial state is visible to us. We then explore different input actions, and check if their following state is already explored. If every edge targets to an explored state, we will say that the hypothesis is *closed*.
2) *Equivalence checking.* With a closed hypothesis generated, we still need eq to check if it is equivalent to the system-under-learn. *Eq* returns *true* if they are equivalent, or a counter-example if not.
3) *Counter-example Analysis.* As mentioned before, we use suffixes to distinguish different states. Consequently, the correctness of our hypothesis relies heavily on proper suffixes. The *suffix* function are designed to analyze counter-examples and find out which suffix is left out.

IV. Experiments

Both *Reo Coordination Models* and *Adapted L* Algorithm* are implemented in Golang. With a CSP-style [7] cocurrency model, the Google-production language `Golang` shares quite a few 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 [1].

A. Case Study

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

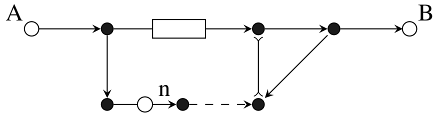

Fig. 3. Expiring FIFO1 (ExpFIFO1)

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. 4 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.

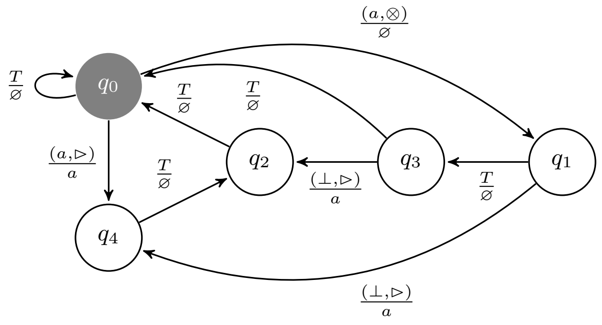

Fig. 4. Learn Result of The ExpFIFO1 where $n = 2, \Sigma = \{a\}$

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.

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

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 [15], 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] Github repository of our implementation. https://github.com/liyi-david/reo-learn.

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

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

[4] 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.

[5] 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.

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

[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] 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.

[12] 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.

[13] B. Settles. Active learning literature survey. Technical report, University of Wisconsin, Madison, 2010.

[14] 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.

[15] 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.