

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 [15], 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. [14] has shown us a conformance testing method on timed connectors. Bounded model checking methods were also adapted in [13], 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-written

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 blackboxes. Works in [1], [8], [17] shows impressive examples where active learning is used to extract *finite state automatas* or *grammars*.

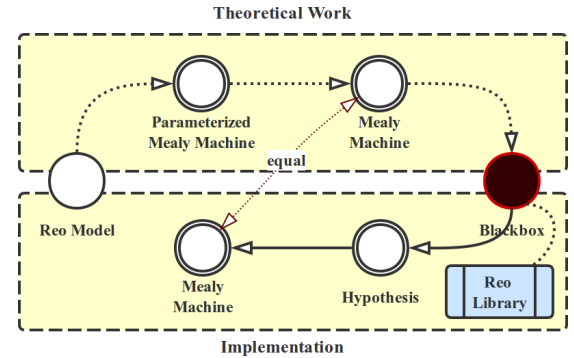


Fig. 1. The Active Learning Framework

In this paper, we proposed an active learning framework (shown in Figure 1) to automatically extract timed connectors from blackbox models. We present *Parameterized Mealy machine* as a parameterized 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.

- 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 $\delta : S \times I \rightarrow S$ mapping pairs of a state and an input symbol to the corresponding next state.
- an output function $\lambda : S \times I \rightarrow O$ 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 $\lambda(q, i)$ and jumps to state $\delta(q, i)$.

C. Active Learning

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

Active learning [18] 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.

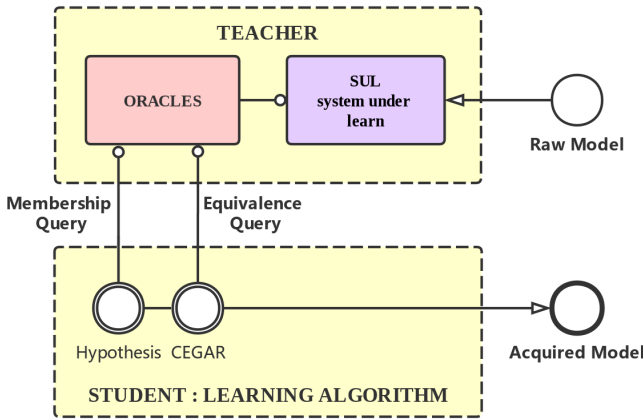


Fig. 5. Active Automata Learning

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 [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, 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** (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, *equivalence query* is unrealizable 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 involved 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 [16] 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*.

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

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 $\delta(\Sigma) : S(\Sigma) \times Input(\Sigma, I) \rightarrow Output(\Sigma, O)$ to denote the output function.
- λ maps an alphabet to a *transition function*. We use $\lambda(\Sigma) : S(\Sigma) \times Input(\Sigma, I) \rightarrow S(\Sigma)$ to denote the transition 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. For example, if we have a alphabet consisting of two items d_0, d_1 and one source-end named A , the input actions would be

$$Input(\{d_0, d_1\}, \{A\}) = \{\{A : d_0\}, \{A : d_1\}, \{A : \perp\}, T\}$$

$$Input(\Sigma, I) = \{\} \cup \{T\}$$

where we use T to denote the time action.

$$Output(\Sigma, I) = \{\}$$

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 (Substantialize Mapping):

Now we can use Parameterized Mealy Machines to define a new semantics to Reo. For lack of space, we take one

asynchronous channel (FIFO1), one asynchronous channel (Sync) and one timed channel (Timer) as examples.

Example 1 (PMM Semantics of FIFO 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

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

- transition function

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

Besides, the concrete mealy machine (where $\Sigma = \{a\}$) is shown in Figure 6.

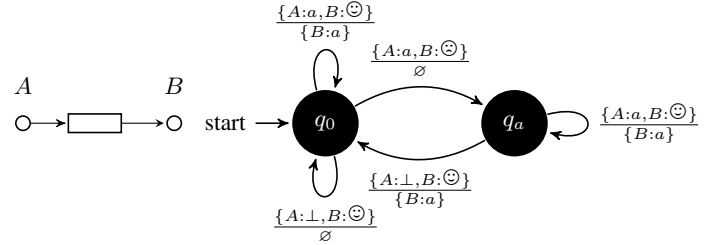


Fig. 6. PMM-based Semantics of FIFO, where $\Sigma = \{a\}$

Example 2 (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

$$\delta(\Sigma)(s, i) = \begin{cases} (B : \perp) & s = q_0 \wedge i = (A : \perp, B : \odot) \\ (B : d) & s = q_0 \wedge i = (A : d, B : \odot) \\ \odot & s = q_0 \wedge i = (A : \perp, B : \odot) \\ \odot & s = q_0 \wedge i = (A : d, B : \odot) \end{cases}$$

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

Example 3 (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:

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

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

- transition function

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

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.

Definition 4 (Production of Parameterized Mealy Machines): 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 $PM_2.O \cap PM_1.I = \emptyset$

- $\forall \Sigma, PM_3.S(\Sigma) = PM_1.S(\Sigma) \times PM_2.S(\Sigma)$
- $PM_3.I = PM_1.I \cup PM_2.I - PM_1.O$
- $PM_3.O = PM_1.O \cup PM_2.O - PM_2.I$
- $PM_3.s_0 = (PM_1.s_0, PM_2.s_0)$
- $\forall \Sigma, PM_3.\delta(\Sigma)((s_1, s_2), i) =$

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

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
 - * $s'_1 = PM_1.\lambda(\Sigma)(s_1, In_1)$
 - * $s'_2 = PM_2.\lambda(\Sigma)(s_2, In_2)$

IV. FROM BLACKBOX TO TIMED CONNECTORS

A. Observation Table

B. Closeness

C. Counter-Examples

Generally, equivalence query has been proved impossible in blackbox models [1]. However, in this section, we're showing that in certain circumstances, equivalence query can be implemented with no approximation.

Equivalence queries are used to search for counter-examples. But what makes counter-examples even existing? In Reo models, we believe that *FIFO* channels and *Timer* channels are to blame. Here we present a brief example. To make things easier, we have only untimed Reo here.

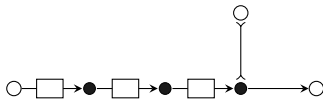


Fig. 7. A Switching Connector S with Three Buffers

A Switching connector in Fig.7 has two source-ends A, B and one sink-end C . In a nutshell, datum come from A and be

stored temporarily in buffers. These datum will never flow out until signals come to B . With the Mealy-Machine semantics given, the semantics of Switching connectors can be defined as following Fig.8.

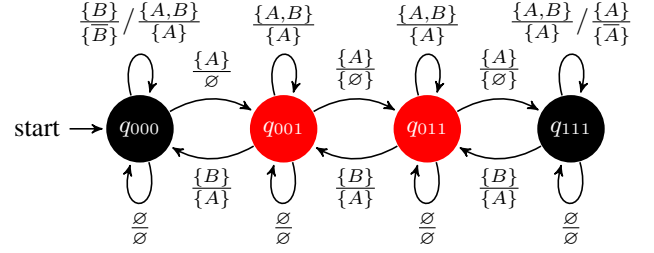


Fig. 8. Gate as Mealy Machine $\mathcal{M}(S)$

According to the production of Mealy-Machines, 8 different states should be found in $\mathcal{M}(S)$. We use q_{abc} to denote them. q_{001} indicates that the last buffer is filled and others are empty, and q_{111} means that there are no space in any buffer. It's obvious that some states like q_{100} is unreachable.

We say if two states are *similar* iff. they have the same output signature.

Theorem 1 (Bound of Counter-Example): Assume that an observation table has already been closed with maximum query length l . An counter-example of length $l + 2$ would be found iff. there're possible counter-examples.

Proof: Sketch of this proof are shown in several points:

- 1) Hypotheses are subgraphs of the semantics graph.
- 2) Inputs on reverse edges are complementary.
- 3)

V. EXPERIMENTS

Both *Reo Coordination Models* and *Adapted L* Algorithm* are implemented in Golang [11].

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 [12]. 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

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
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 significantly. In this work, we use a multiway tree to buffer these results. A brief example of such trees can be found in Figure 9.

Example 4 (Multiway-Tree Cache): Considering the FIFO channel presented in Example 1. Note that in this figure, only

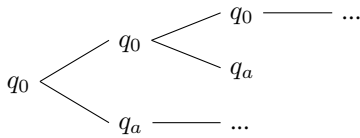


Fig. 9. Multiway-Tree Cache

the edge information is stored. States like q_0, q_a are written here only to make it clear.

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 a good 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.

```

1 package main
2
3 import . './lib/reo'
4
5 func alternator(A, B, C Port) {
6     // definition of ports
7     M0 := MakePort()
8     // M1 ... M5 defined similarly
9
10    // definition of channels
11    /* NOTE
12     go function() means that the function would
13     be executed as a new parallel task
14    */
15    go ReplicatorChannel(A, M0, M1)
16    go ReplicatorChannel(B, M2, M3)
17    go MergerChannel(M4, M5, C)
18
19    go SyncdrainChannel(M1, M2)
20    go SyncChannel(M0, M4)
21    go FifoChannel(M3, M5)
22 }

```

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 := MakePort()
2 B := MakePort()
3 C := MakePort()
4 alternator(A, B, C)
5
6 go sender(A, "MSG_A")
7 go sender(B, "MSG_B")
8 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

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] Y. Chen, C. Hsieh, O. Lengál, T. Lii, M. Tsai, B. Wang, and F. Wang. PAC learning-based verification and model synthesis. *CoRR*, abs/1511.00754, 2015.
- [7] 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.
- [8] 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.
- [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. *Sci. Comput. Program.*, 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 *2015 International Symposium on Theoretical Aspects of Software Engineering, TASE 2015, Nanjing, China, September 12-14, 2015*, pages 15–22. IEEE Computer Society, 2015.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] B. Settles. Active learning literature survey. *University of Wisconsin, Madison*, 52(55-66):11, 2010.
- [19] 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

better expressions!	1
How to say?	4
example of production, maybe fifo+timer	5
find a better phrase?	5
the graph need further details	6