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 coordinator. 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.

researchers have been focusing on this area, and come up with a series of impressive works. However, most of these works are based on models, instead of binaries. Then it comes a well-known problem: *how can we obtain these models?*

To solve this problem, many techniques in model constructing were proposed, for instance in [1], [8], [13].

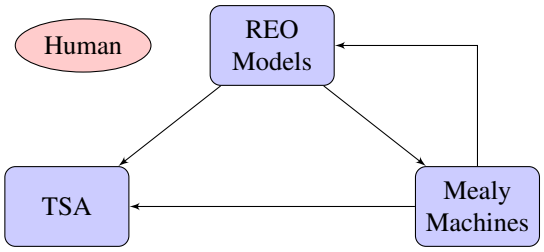

Fig. 1. Our Idea

In this paper, we presented an adapted active learning algorithm to extract timed Reo connectors from binaries with no source code needed.

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. 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 [10].

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.

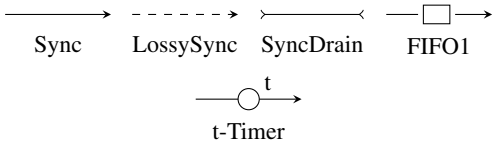

Fig. 2. Basic Reo Channels

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 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)

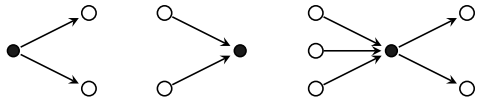

Fig. 3. Source, Sink and Mixed Nodes in Reo

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 returns on its sink end a timeout signal 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.

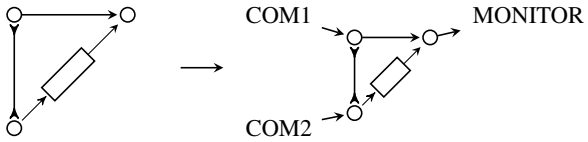

Fig. 4. Coordination with Reo Connectors

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

As an extension of *finite state machine*, Mealy machine was first proposed by George. H. Mealy in [9]. 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 [15].

Definition 1 (Mealy machine): A Mealy machine is a 6-tuple (S, s_0, I, O, T, G) 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.

C. Active Learning

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

Active learning [14] 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.

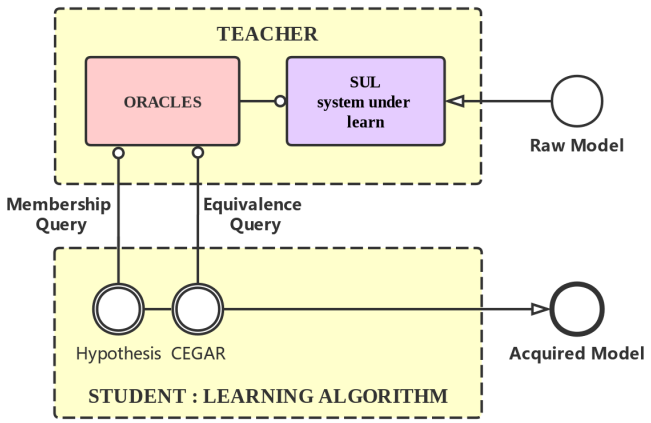

Fig. 5. Active Automata Learning

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 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.

example, also a mealy-machine example

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

A. External Behavior of Reo Connectors

reason?

So far as we can tell, most works [6], [13] on automata learning are not capable of infinite models. Considering the semantics defined in section II-A, it's apparent that every finite connector has a corresponding constraint automata, and the automata is also finite. Unfortunately, when checking the further definition of Reo connectors' input, we find that it's not the case.

an example as graph

Fig. 6. Infinite States in a Reo Model

While focusing on behavior of connectors, Reo doesn't give detail depiction on behavior of components. As shown in Fig.6, connectors are able to reject any datum if they are not ready. But what will happen outside the connector, if the datum is rejected? This question deserves careful consideration if we are taking an external view.

B. 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 [**?**] rational time is chosen to make 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.

C. Parameterized Mealy Machine

We present a model named *parameterized mealy machine* to represent timed connectors (hereinafter referred to as PMM).

This model 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.

Definition 2 (Parameterized Mealy Machine): A *Parameterized Mealy Machine* is defined as a function $\mathcal{PM}(\Sigma) = \langle S(\Sigma), s_0, I, O, \delta(\Sigma), \lambda(\Sigma) \rangle$ that maps an alphabet to its corresponding Mealy Machine.

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

In the definition above, *Input* and *Output* are used to generate input actions and output actions by the corresponding alphabets and ends.

How to say?

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

where we use T to denote the time action.

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

Now we can use Parameterized Mealy Machines to define a new semantics to Reo.

Example 1 (PMM Semantics of FIFO channel PM_{FIFO}): 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

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

- transition function

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

Besides, the concrete mealy machine (where $\Sigma = \{a\}$) is shown in Figure 7.

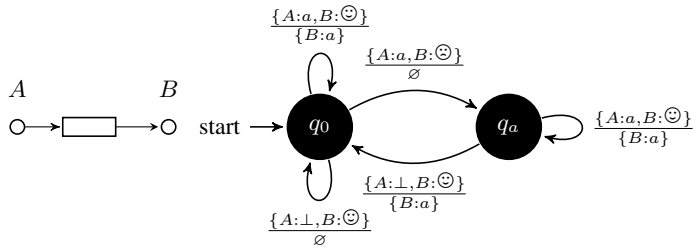

Fig. 7. PMM-based Semantics of FIFO, where $\Sigma = \{a\}$

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 3 (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 $PM_2.O \cap PM_1.I = \varnothing$

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

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

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

example of production, maybe fifo+timer

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 eaiser, we have only untimed Reo here.

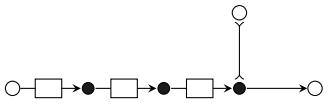

Fig. 8. A Switching Connector S with Three Buffers

A *Switching* connector in Fig.8 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.9.

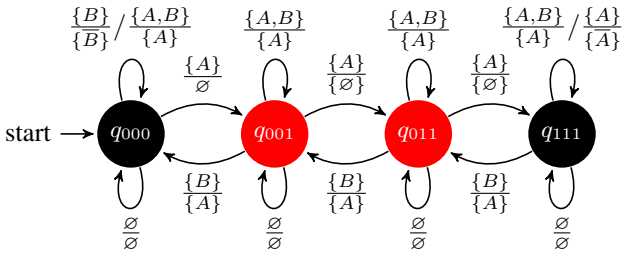

Fig. 9. *Gate* as Mealy Machine $\mathcal{M}(S)$

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 *similiar* 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.

find a better phrase?

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

1) Hypothesises 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 signficiantly. In this work, we use a multiway tree to buffer these results. A brief example of such trees can be found in Figure 10.

Example 2 (Multiway-Tree Cache): Considering the FIFO channel presented in Example 1. Note that in this figure, only

the graph need further details

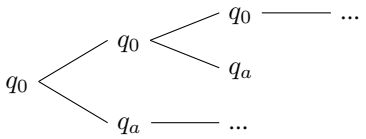

Fig. 10. Multiway-Tree Cache

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] 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.

[14] 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.

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

[16] 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