Bachelor thesis
Computing Science

Radboud University

Learning Observation Trees of Machines with Multiple Timers

Author:
Martan van der Straaten
s1042145

First supervisor/assessor:
Prof.dr. Frits W. Vaandrager
I.Berenbroek@cs.ru.nl

[Second supervisor:]
dr. Jurriaan C. Rot
j.rot@cs.ru.nl

Second assessor:
title, name
e-mail adress

April 4, 2022

Abstract

The abstract of your thesis is a brief description of the research hypothesis, scientific context, motivation, and results. The preferred size of an abstract is one paragraph or one page of text.

Contents

Chapter 1

Introduction

One of the big problems in software engineering is proving correctness of software implementations. Some of the systems that are implemented are absolutely crucial to society, and mistakes in such systems should not exist in an ideal world. As of now the reality is however, that proving a that a systems implementation adheres to some specification is far from trivial.

There are several methods that have surfaced over the years that attempt to address this issue. One of these methods is active automata learning, first presented by Dana Angluin in her famous paper on L^*[1]. In this paper Angluin shows that the set of regular languages can be learned efficiently, using what she calls the *MAT* framework; In which the system is represented by some *Minimally Adequate Teacher*. Learning can be represented as a game. In this game the learner has to infer a DFA for language L by asking queries to a teacher. The learner has two options for queries: "Is word w in L?" (membership queries), and "Is the language recognized by DFA D equal to L?" (equivalence queries).

After this initial paper, many extensions to this framework have been created. Examples of these are the $L^\#$[4] algorithm which learns Mealy Machines efficiently using observation trees, or the extension to Mealy Machines with 1 timer (MM1T's)[3]. These papers solve some of the issues regarding Angluin's original paper. A first issue was that DFAs do not translate well to real systems. Shifting the learning-process to Mealy Machines solved part of this issue, as Mealy Machines give a far better representation of most systems. Another problem that was encountered were systems with timers.

When an implementation of TCP was learned [2] the problem of timers presented itself. There was no natural way to fit these timers into the existing learning algorithms. This problem was partly solved by [3]. However the solution presented here relies on the existence of exactly one timer, and does not extend to the general case. For verification of the implementation of systems with multiple timers, a learning algorithm which learns systems with multiple timers is a crucial next step. This raises the question this paper will attempt to answer "How can we extend Mealy Machine learning such that the learning process incorporates multiple timers?".

How does it work

How does it compare to the previous work

This paper outlines a proposal for such a learning algorithm that learns systems with mul-

tiple timers (MMT's) in a *MAT* environment. It describes how from this algorithm we can derive learning algorithms for systems with N timers, aka MMNT's. The complexity will be described and correctness of the MMT algorithm will be shown. After this an implementation of the MM2T and MM3T algorithm is presented, and its performance on several benchmarks will be provided with comparison to existing techniques.

Chapter 2

Preliminaries

Add many many citations

2.1 Languages and Automata

In software verification it is often extremely useful to be able to describe systems in a formal manner. Using models that approximate our program we can verify that no mistakes are present. This allows us to know with relative certainty that our program adheres to our specification. One very useful type of model for a program is called an automaton. An automaton is a model that behaves in a certain way depending on the input it receives, and possible other factors. An automaton generally works on one or more alphabets of in- and output.

Definition 2.1.1. An alphabet Σ is a finite set of symbols.

Definition 2.1.2. A word $w \in \Sigma^*$ is a concatenation of symbols from the alphabet Σ of any length. The empty word has length 0 and is denoted as λ.

Definition 2.1.3. A language L then is a set of words such that $L \subseteq \Sigma^*$.

2.2 DFAs

One of the most basic types of automaton is the Deterministic Finite Automaton (DFA). A DFA D describes some language $L_D \subseteq \Sigma^*$. Each word w in $\{a, b\}^*$ is in L_D iff D ends in a final state after executing with w on the tape from initial state q_0.

Definition 2.2.1. A DFA can be represented by a 5-tuple $\langle Q, \Sigma, \delta, q0, F \rangle$ where:

1. Q is a finite set of states
2. Σ is a finite set of symbols called the alphabet
3. δ is the transition function where $\delta : Q \times \Sigma \to Q$
4. q_0 is the initial state from where any input is processed ($q_0 \in Q$)
5. F is a set of final states in Q such that $F \subseteq Q$

2.3 Mealy Machines

Another type of automaton is the Mealy Machine (MM). Where a DFA classifies acceptance on words, a MM is used to generate some sequence of outputs, based on a sequence of inputs. A Mealy Machine is a finite-state machine whose output value at any place is completely determined both by its current state and the current input.

Definition 2.3.1. A Mealy Machine can be represented by the 6-tuple $\langle Q, q_0, I, O, \delta, \lambda \rangle$ where:

1. Q is a finite set of states
2. q_0 is the initial state from where any input is processed with $q_0 \in Q$
3. I is a finite set of input symbols
4. O is a finite set of output symbols
5. δ is the transition function of the form $\delta : Q \times I \to Q$
6. λ is the output function of the form $\lambda : Q \times I \to O$

When a Mealy machine is ran on a word $w = i_0 i_1 \dots i_{n-1} i_n$, it will produce output $v = o_0 o_1 \dots o_{n-1} o_n$. With $o_0 = \lambda(q_0, i_0)$, $q_n = \delta(q_{n-1}, i_{n-1})$ and $o_n = \lambda(q_n, i_n)$, etc.

The notation $q \xrightarrow{i/o} q'$ is used to denote a transition in the Mealy Machine for $q, q' \in Q$, $i \in I$, $o \in O$. It denotes $\lambda(q, i) = o$ and $\delta(q, i) = q'$.

2.4 Partial functions

Definition 2.4.1. We write $f : X \rightharpoonup Y$to denote that f is a partial function from X to Y. We write $f(x) \downarrow$ to mean that the result is defined for x, that is, $\exists y : f(x) = y$, and $f(x) \uparrow$ if the result is undefined.

If f and g are partial functions, then $f(x) = g(y)$ holds if either both $f(x)$ and $g(y)$ are undefined, or both $f(x)$ and $g(y)$ are defined and have the same result.

2.5 Observation Trees

Definition 2.5.1. For Mealy machines $\mathcal{M}$ and $\mathcal{N}$, a functional simulation $f : \mathcal{M} \to \mathcal{N}$ is a map $f : Q^{\mathcal{M}} \to Q^{\mathcal{N}}$ with $f(q_0^{\mathcal{M}}) = q_0^{\mathcal{N}}$ and $q \xrightarrow{i/o} q'$ implies $f(q) \xrightarrow{i/o} f(q')$

Definition 2.5.2. A partial Mealy machine is a tuple $\langle Q, q_0, I, O, \delta, \lambda \rangle$, where:

1. Q is a finite set of states.
2. $q_0 \in Q$ is the initial state.
3. I is a finite set of input symbols.
4. O is a finite set of output symbols.

5. $\langle \lambda, \delta \rangle : Q \times I \rightharpoonup O \times Q$ is a partial map whose components are an output function $\lambda : Q \times I \rightharpoonup O$ and a transition function $\delta : Q \times I \rightharpoonup Q$ (hence, $\delta(q,i)\downarrow \Leftrightarrow \lambda(q,i)\downarrow$, for $q \in Q$ and $i \in I$).

Definition 2.5.3 (**Observation Trees**)**.** A Partial Mealy Machine T is a tree if for each $q \in Q^T$ there is a unique sequence $\sigma \in I^*$ s.t. $\delta^T(q_0^T, \sigma) = q$. e write $access(q)$ for the sequence of inputs leading to q. A tree T is an observation tree for a Mealy machine $\mathcal{M}$ if there is a functional simulation $f : T \to \mathcal{M}$

2.6 MMT

In this paper we look to elaborate on the work of [TODO] in defining automata learning for Mealy Machines with multiple Timers (MMTs). We first declare what is mean by an MMT.

Definition 2.6.1. Let $X = \{x_1, x_2, x_3, \dots\}$ be the infinite set of timers and let TO be the set of timeout-events of shape $to[x]$ for $x \in X$. For our set of input symbols inp, we have that $\hat{I} = I \cup TO$

A Mealy Machine with timers (MMT) is a 6-tuple $M = \langle I, O, Q, q_0, X, \delta, \lambda, \pi \rangle$

1. I and O are finite sets of input events and output events, respectively
2. Q is a finite set of states, with $q_0 \in Q$ the initial state
3. $\mathcal{X} : Q \to \mathcal{P}_{\text{fin}}(X)$, with $X(q_0) = \emptyset$. Where $\mathcal{P}_{\text{fin}}(X)$ is the set of finite subsets of X
4. $\delta : Q \times \hat{I} \rightharpoonup Q$ is a transition function
5. $\lambda : Q \times \hat{I} \rightharpoonup O$ is an output function
6. $\pi : Q \times \hat{I} \rightharpoonup (X \rightharpoonup \mathbb{N}^{>0})$ is a timer update function

We write $q \xrightarrow{i/q,\rho} q'$ if $\delta(q,i) = q', \lambda(q,i) = o, \pi(q,i) = \rho$

Take $q \in Q, i \in \hat{I}, q' = \delta(q,i)$ and $\rho = \pi(q,i)$. We require that input events are enabled for every state q, and timeout events are enabled for all timers that are active in the current state, meaning that:

- $\delta(q,i), \lambda(q,i), \pi(q,i)$ are defined iff $i \in I$ or $i = to[x]$ for some $x \in \mathcal{X}(q)$

We also require that $X(q')\backslash X(q) \subseteq dom(\rho) \subseteq X(q')$. Which states the following:

- $X(q')\backslash X(q) \subseteq domp(\rho)$ which tells us that: If a timer x is in $X(q')\backslash X(q)$, which means it is on in state q' but was off in q then it is in the domain of $\pi(q,i)$ and thus timer x is updated by $\pi(q,i)$.
- $dom(\rho) \subseteq X(q')$ tells us that all timers that were updated by $\pi(q,i)$ will ensure that those timers are on in state q'.

Lastly we require that when timer x expires, it is either stopped or restarted:

- if $i = \text{to}[x]$, for some x, then $x \notin X(q')\backslash dom(\rho)$.

As further clarification on our update function π. It determines how timers are affected when an event occurs. Suppose we have transition $q \xrightarrow{i/o,\rho} q'$. One of four things can happen:

1. If $x \in X(q) \backslash TO(q')$ then input i stops timer x.
2. If $x \in X(q') \backslash TO(q)$ then input i stops timer x.
3. if $x \in X(q) \cap dom(\rho)$ then i restarts timer x with value $\rho(x)$.
4. Finally, if $x \in X(q') \backslash dom(\rho)$ then timer x is unaffected by i.

2.7 Semantics of MMTs

Rewrite in own words, and possibly remove unneccesary, and add explanations

The semantics of MMT $\mathcal{M}$ is defined via an infinite state transition system that describes all possible configurations and the transitions between them. A valuation $\kappa : X \rightharpoonup \mathbb{R}^{\geq 0}$ we write $Val(Y)$ for the set of valuations $Y \subseteq X$. A configuration of an MMT is a pair (q, κ) for $q \in Q, \kappa \in Val(X(q))$. The initial configuration is (q_0, κ_0) with κ_0 the empty function.

If κ is a valuation in which all timers have a value of at least d then d time units can pass. As a result of such a delay, the value of all these timers is decreased by d. Formally

$$\kappa \xrightarrow{d} \kappa' \Leftrightarrow [dom(\kappa) = dom(\kappa') \wedge \forall x \in dom(\kappa) : \kappa'(x) = \kappa(x) - d]$$

For current valuation κ we can have timout event $to[x]$ only if $\kappa(x) = 0$. κ is updated by update function ρ after an input or timeout event occurs. The value of timers that are not affected by ρ remain unchanged. Formally,

$$\begin{aligned} \kappa \xrightarrow{i/o,\rho} \kappa' \Leftrightarrow & [dom(\kappa') \backslash dom(\kappa) \subseteq dom(p) \subseteq dom(k') \wedge \\ & [\forall x \in dom(k') : k'(x) = \text{ if } x \in dom(\rho) \text{ then } \rho(x) \text{ else } \kappa(x)] \wedge \\ & \forall x \in X : i = to[x] \implies (\kappa(x) = 0 \wedge x \notin dom(\kappa') \backslash dom(\rho))] \end{aligned}$$

We create the following two rules to lift this behaviour to configurations.

$$\frac{q = q' \quad \kappa \xrightarrow{d} \kappa'}{(q, \kappa) \xrightarrow{d} (q', \kappa')} (1) \qquad \frac{q \xrightarrow{i/o,\rho} q' \quad \kappa \xrightarrow{i/o,\rho} \kappa'}{(q, \kappa) \xrightarrow{i/o} (q', \kappa')} (2)$$

We can now define a timed run as to be a sequence of configurations C_j, C'_j with C_0 the original configuration:

$$\alpha = C_0 \xrightarrow{d_0} C'_0 \xrightarrow{i_0/o_0} C_1 \xrightarrow{d_1} C'_1 \xrightarrow{i_1/o_1} C_2 \ldots \xrightarrow{d_k} C'_{k-1} \xrightarrow{i_k/o_k} C_k$$

A timed word is of shape $w = d_0 i_0 o_0 \; d_1 i_1 o_1 \; \ldots \; d_k i_k o_k$

2.8 Limitations and requirements

2.8.1 Non-zero delays

We assume that timers have non-zero delays. Downsides and why. (We need this for wiggling backwards)

2.8.2 Non-determinism

Inside any timed automaton non-determinism can exist. This can lead to problems when learning the model. The non-determinism stams from two possible things:

1. Either two timers $x, y \in X$ timeout at the exact same moment. At this time it would not be decidable which of the two's timers behaviour we will observe first, if the second timer is observed at all.
2. Or we have timer $x \in X$ set to value a, and we provide some input $i \in I$ after exactly a time-units. Posing the same issue as shown in (1).

It can be observed that there are two types of determinism, controllable and non-controllable.

Controllable non-determinism and single-timer transitions

Controllable non-determinism is non-determinism that is not controlled by the timings provided to the automaton. No matter what timings we provide, the automaton is still not deterministic. [TODO IMAGE]

A way to circumvent this type of non-determinism is to only consider models in which one timer is set at each transition.

Non-controllable non-determinism

Non-controllable non-determinism is determinism that does rely on the precise timing we give with our inputs. Non-determinism is thus fixable by tweaking the input timing, however there is no easy way to guarantee that no such non-determinism is present. [TODO IMAGE + rewrite]

2.8.3 Ghost timers

When a timer is started, but there is no possible way to observe its timeout, we describe such a timer as a "ghost timer". The underlying untimed-semantics for our SUL will clearly have this timer present. It is however impossible for our model to learn an unobservable timer, therefore we cannot have untimed equivalence for any model with ghost-timers. However, it could be argued that for most realistic systems it is irrelevant whether such a timer is set or not when it cannot be observed.

2.9 Blocks

...

Chapter 3

Blocks and Wiggling

The algorithm that will build the tree is based on $L^{\#}$[TODO]. It will use the implementation and theory that is specified there to learn the transitions of the system. The algorithm that is described below will provide means to extend the original $L^{\#}$ algorithm in such a way that it can now learn models for a SUL with multiple timers.

The algorithm will attempt to provide the exact timing for transitions in a given run on the SUL. We assume that all timers are off in our initial state, no ghost-timers are present [TODO] and no explicit-recursion is present [TODO].

3.1 Blocks

Let define an algorithm for finding blocks.

Algorithm 1 FindBlocks

Require: $\{\rho_1, \ldots, \rho_k\}$
Ensure: $\texttt{Blocks}_k$
1: $\texttt{Blocks} = \{\{\rho_1\}, \ldots, \{\rho_k\}\}$
2: $n \leftarrow k$
3: **while** $n > 0$ **do**
4: **if** ρ_{k+1}`.i == to[`x_i`]` **then**
5: `Blocks[i].append(`ρ_n`)`
6: `Blocks.pop(n)`
7: **end if**
8: **end while**
9: **return** Blocks

Lemma 1. Given some run $q_0 \rightarrow q_1 \rightarrow \ldots q_{k-1} \rightarrow q_k$ we can construct the proper sets of blocks using Algorithm 1.

Proof. It should hold that every block is a maximal subset indexes $\{\rho_1, \ldots, \rho_k\}$ of the transitions such that i_{ρ_j} triggers i_{ρ_j+1} within the block.

1. We first want to show that indeed i_{ρ_j} triggers i_{ρ_j+1} within the block. This follows from the definition of our algorithm. Since we start with singleton blocks, and only increase the size after we can use induction.
    i. Property. If our block is $\{\rho_1, \ldots, \rho_n\}$ then i_{ρ_j} triggers i_{ρ_j+1} within the block.
    ii. Base case: We take n to be one. Clearly the property vacuously holds.
    iii. For block B with size $k+1 > 1$ of form $\{\rho_1, \ldots, \rho_{k+1}\}$ there must be a last index u for which ρ_u was appended to B. Now take B' to be $B \backslash \rho_u$.

    Then, since we know that the if statement was called in order to append p_u, we know from our algorithm that B' is the block which contains p_v such that p_v sets the timer that triggers p_u.

    That, combined with the fact that using our IH we can assume that our property holds for B' tells us that it must be the case that indeed the property holds for blocks of size $k+1$.
    iv. Now by induction we know that our algorithm produces correct blocks of size k for $k \in \mathbb{N}_{\geq 1}$.
2. We want to show that each block is the maximal subset after termination. We can easily show this by contradiction.
    i. Assume that there is some blocks B_1, B_2 in our final set of block $Blocks$ such that the triggers property holds for $B_1 \cup B_2$.
    ii. Then we must have one of the following situations, but both give a contradiction.
        - $x \in B_1$ and $y \in B_2$ such that x triggers y. But then x sets some timer t which expires to cause timeout $to[t]$ as input for y. However, if that were the case y would by definition be placed in the block of x due to the if-statements body.
        - Similar argument for $x \in B_1$ and $y \in B_2$ such that y triggers x.

□

Now that we have described how to find such blocks, we can describe some of their properties.

Lemma 2. Given some run $q_0 \to q_1 \to \ldots q_{k-1} \to q_k$, and a set of blocks $Blocks$ for this run. If we know that timer x was started in some block $B \in Blocks$, then we can determine exactly which transition in B set this timer.

Proof. This proof follows from the definition of a block

1. A block $\{p_1, \ldots, p_k\}$ consists of transitions such that p_i triggers p_{i+1} for $1 \leq i \leq k$.
2. If p_i triggers p_{i+1} then this means that p_i sets some timer x such that in p_{i+1} we observe input $to[x]$.
3. As we can only set one timer at each transition, it must hold that indeed only transition p_k currently has possible uncertainty about the timers it sets. As any other timer p_j with $j < k$ by definition sets the timer that causes p_{j+1}.

4. If we know that it is set in this block, but for all $i \neq k$ we have that p_i did not set the timer, then it must follow that p_k set the timer.

□

3.2 Active timers

We cannot learn perfect information about the active timers at every level of the tree. As such timers might or might not be set, and observations will only happen much lower in the tree.

(might be neccesary to state, probably not)

3.3 Wiggling

We find the blocks for our run using Algorithm 1 as *Blocks*. Now we have 2 cases:

If there is one block, we cannot wiggle at all. There also however, is no need to wiggle, as we trivially know exactly what transition set the timer by Lemma 2. So we take the last transition in the block to be our culprit.

If there is more than one block we can wiggle our blocks backwards, one by one, to determine which block sets the timer. If the block that starts the timer was moved backwards, the time-out corresponding to this block will backwards as well.

Let us define the concrete algorithm in Algorithm 3.

We use several properties of transitions, such as the actTime, timer and index. actTime stands for actual time, timers stands for the value of the timer it sets (unknown is set to 0) and index is the position of this transition with respect to the whole run. This can all be easily pre-computed, and the assumption is made that this has been done beforehand.

Algorithm 2 getSAT

Require: k ▷ Number of timers
Require: `Blocks`
Require: `transitions` $= \{\rho_1, \ldots, \rho_k\}$
Require: `BIndex` ▷ Block we want to wiggle
Ensure: `new transitions`

`requirements` ← `String[]`
`requirements.push`($t_0 = 0$)

for i in range(k) **do**
    `requirements.push`($t_i < t_{i+1}$)
end for

for `Block : Blocks` **do**
    j ← `Block[0].index` ▷ index of first transition with regards to the full run

    if j `== BIndex` **then**
        `requirements.push(` $t_j =$ `Block[0].actTime -` d`)` ▷ Wiggle this block
        `requirements.push(` $0 < d <$ `smallestTimer)`
    else
        `requirements.push(` $t_j =$ `Block[0].actTime)`
    end if

    for i `in len(Block-1)` **do**
        l ← `Block[i].index`
        m ← `Block[i+1].index`
        `requirements.push`($t_m = t_l$+`Block[i].timer`)
    end for
end for

`Timers` ← SAT(requirements)

for i `in range(1, transitions.length)` **do**
    ρ_i`.timer =` $t_i - t_{i-1}$
end for

return $\{\rho_1, \ldots, \rho_k\}$

Algorithm 3 Wiggle

Require: $\{\rho_1, \ldots, \rho_k\}$
Require: `t` ▷ Moment that the timer expired
Require: $\mathcal{M}$
Ensure: `B` ▷ Block that started the timer

1: `Blocks` ← `FindBlocks()`
2: `SearchBlocks` ← `Blocks`
3:
4: `t'` ← t
5: `Result` ← 0
6:
7: **do**
8: `B` ← `SearchBlocks.pop()`
9:
10: $\{\rho_1^1, \ldots, \rho_{k-1}^1\}$ ← `getSAT(k, Blocks,` $\{\rho_1, \ldots, \rho_{k-1}\}$`, B.index)`
11: $\{\rho_1^1, \ldots, \rho_k^1\}$ `=` $\mathcal{M}$`.test(`$\{\rho_1^1, \ldots, \rho_{k-1}^1\}$
12:
13: `Result` ← `B`
14: $t' = \rho_k^1$`.actTime`
15: **while** t'=t
16: **return** `Result`

Chapter 4

Algorithm

Using the theory outlined in the previous chapter we can define Algorithm 4.

4.1 Example run of Wiggle

We have the following machine, for which we need to learn a tree:

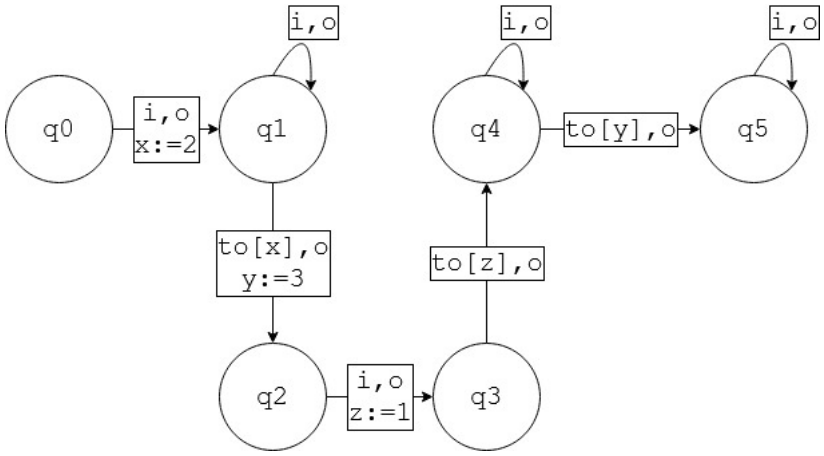

Figure 4.1: Example machine, hidden from learner

We assume we have the following run α, and we have a tree that contains information until q4. We now observe q5 after some timeout and want to expand and update the tree.

$$\alpha = q_0 \xrightarrow{5} q'_0 \underbrace{\xrightarrow{i/o,x:=2}}_{t_1} q_1 \xrightarrow{2} q'_1 \underbrace{\xrightarrow{to[x]/o}}_{t_2} q_2 \xrightarrow{1} q'_2 \underbrace{\xrightarrow{i/o,z:=1}}_{t_3} q_3 \xrightarrow{1} q'_3 \underbrace{\xrightarrow{to[z]/o}}_{t_4} q_4 \xrightarrow{1} q'_4 \underbrace{\xrightarrow{to[y]/o}}_{t_5} q_5$$

In the original model we can find the appropriate assignment of y to be as follows:

$$\alpha' = q_0 \xrightarrow{5} q'_0 \xrightarrow{i/o,x:=2} q_1 \xrightarrow{2} q'_1 \xrightarrow{to[x]/o,y:=3} q_2 \xrightarrow{1} q'_2 \xrightarrow{i/o,z:=1} q_3 \xrightarrow{1} q'_3 \xrightarrow{to[z]/o} q_4 \xrightarrow{1} q'_4 \xrightarrow{to[y]/o} q_5$$

Algorithm 4 buildTree

Require: `Run =` $q_0 \xrightarrow{d_1} q'_0 \xrightarrow{i_1/o_1} q_1 \ldots \xrightarrow{d_k} q'_{k-1} \xrightarrow{i_k/o_k} q_k$
Require: Δ ▷ Maximum time waiting for timeout
Require: $\mathcal{M}$ ▷ SUL
Ensure: `T` ▷ Tree with timers

`Node`$_0 \leftarrow (\lambda, q_0, [])$ ▷ Node consisting of transition, state and children.
`Node`$_1 \leftarrow (\ \{i_1, o_1, NULL\}\ , q_0, [])$
`Node`$_0$`.children.append(Node`$_1$`)`
`T` $\leftarrow$ `Node`$_0$

`TimersActive` $\leftarrow$ `False`

for `j in range(1, k-1)` **do**
    `Test` $\leftarrow q_0 \xrightarrow{d_1} q'_0 \xrightarrow{i_1/o_1} q_1 \ldots q_j \xrightarrow{\Delta} q'_j$

    if $\mathcal{M}$`.test(Test) has some input` $j+1$ `after` d `timeunits` **then**
        if not TimersActive **then** ▷ Base case
            `T[j].timer` $\leftarrow$ "$x_j = d$"
            `Node` $\leftarrow (\ \{to[x_j]/o_{j+1}, NULL\}\ , q_{j+1}, [])$
            `T[-1].children.push(Node)`
        else ▷ Inductive case
            $l \leftarrow$ `Wiggle(`$q_0 \xrightarrow{d_1} q'_0 \xrightarrow{i_1/o_1} q_1 \ldots \xrightarrow{i_j/o_j} q_j$`, Run[j].actTime,` $\mathcal{M}$`)`
            $d = \rho_j$`.actTime -` ρ_l`.actTime`
            `T[l].timer` $\leftarrow$ "$x_l = d$"
            `Node` $\leftarrow (\ \{to[l]/o_j, NULL\}\ , q_{i+1}, [])$
            `T[-1].children.push(Node)`
        end if

        `TimersActive` $\leftarrow$ `True`

    else
        `Node` $\leftarrow (\ \{i_{j+1}/o_{j+1}, NULL\}\ , NULL), q_{j+1}, [])$
        `TimersActive` $\leftarrow$ `False`
    end if

end for

We want to find which transition set y, given that we know that it timed out. We assume by induction that the sequence of length $k-1=4$ has the correct transition setting the timer, for all observed timers. Thus $to[x]$ and $to[z]$ are the only two transitions that could have set timer y, by the fact that at most one timer can be set per transition.

4.1.1 Blocks

We can build our blocks as defined in Algorithm 1.

1. Our initial set of blocks is given by:
   $Block := \{\{t1\}, \{t2\}, \{t3\}, \{t4\}\}$
2. We see that for t_4 the input was $to[z]$, z was set by t_3, so we get:
   $Block := \{\{t1\}, \{t2\}, \{t3, t4\}\}$
3. We see that for t_3 the input was not a timeout. This means that t_3 was not caused by another transition, and therefore we keep the same set of blocks:
   $Block := \{\{t1\}, \{t2\}, \{t3, t4\}\}$
4. We see that for t_2 the input was $to[x]$, x was set by t_1, so we get:
   $Block := \{\{t1, t2\}, \{t3, t4\}\}$
5. We see that for t_1 the input was not a timeout. This means that t_1 was not caused by another transition, and therefore we keep the same set of blocks:
   $Block := \{\{t1, t2\}, \{t3, t4\}\}$

We obtain the following view of our run:

$$\alpha_{blocks} = q_0 \xrightarrow{5} \underbrace{\underbrace{q'_0 \xrightarrow[t_1]{i/o,x:=2} q_1}_{B_1} \xrightarrow{2} \underbrace{q'_1 \xrightarrow[t_2]{to[x]/o} q_2}_{B_1}}_{B_1} \xrightarrow{1} \underbrace{\underbrace{q'_2 \xrightarrow[t_3]{i/o,z:=1} q_3}_{B_2} \xrightarrow{1} \underbrace{q'_3 \xrightarrow[t_4]{to[z]/o} q_4}_{B_2}}_{B_2} \xrightarrow{1} q'_4 \xrightarrow[t_5]{to[y]/o} q_5$$

4.1.2 Wiggling

Now we can wiggle the blocks to see how they impact t_5. We first try to wiggle block B_1:

We get the following SAT requirements:

```
; Variable declarations
(declare-fun t0 () Real)
(declare-fun t1 () Real)
(declare-fun t2 () Real)
(declare-fun t3 () Real)
(declare-fun t4 () Real)

(declare-fun d () Real)

; Constraints
(assert (= t0 0))
(assert (> t1 t0))
(assert (> t2 t1))
(assert (> t3 t2))
(assert (> t4 t2))

(assert (= t2 (+ t1 2)))
(assert (= t4 (+ t3 1)))

(assert (= t1 (- 5 d) ))
(assert (> d 0))
(assert (< d 1))

(assert (= t3 8))

; Solve
(check-sat)
(get-model)
```

With the following solutions presented by Z3:

```
sat
(model
  (define-fun t0 () Real
    0.0)
  (define-fun t2 () Real
    (/ 13.0 2.0))
  (define-fun d () Real
    (/ 1.0 2.0))
  (define-fun t1 () Real
    (/ 9.0 2.0))
  (define-fun t4 () Real
    9.0)
  (define-fun t3 () Real
    8.0)
)
```

Using this we find our new run:

$$\alpha_2 = q_0 \xrightarrow{4.5} q'_0 \underbrace{\xrightarrow{i/o,x:=2}}_{t_1} q_1 \xrightarrow{2} q'_1 \underbrace{\xrightarrow{to[x]/o}}_{t_2} q_2 \xrightarrow{1.5} q'_2 \underbrace{\xrightarrow{i/o,z:=1}}_{t_3} q_3 \xrightarrow{1} q'_3 \underbrace{\xrightarrow{to[z]/o}}_{t_4} q_4$$

When we run this on the automata in figure 3.2 we get the following situation:

$$\alpha_2 = q_0 \xrightarrow{4.5} q'_0 \underbrace{\xrightarrow{i/o,x:=2}}_{t_1} q_1 \xrightarrow{2} q'_1 \underbrace{\xrightarrow{to[x]/o}}_{t_2} q_2 \xrightarrow{1.5} q'_2 \underbrace{\xrightarrow{i/o,z:=1}}_{t_3} q_3 \xrightarrow{1} q'_3 \underbrace{\xrightarrow{to[z]/o}}_{t_4} q_4 \xrightarrow{0.5} q'_4 \underbrace{\xrightarrow{to[y]/o}}_{t_5} q_5$$

We see that the transition has been shifted along with our block B_1, thus B_1 must indeed have caused the timeout.

4.1.3 Constructing the new run after Wiggle

Now that we know that block B_1 causes t_5, we know that t_2 must have set it by Lemma 2. We can calculate the exact value it was set to by subtracting the time at t_2 from the time t_5 to get that $val = t_5 - t_2 = 10 - 7 = 3$.

So now we can extend the knowledge of our run to:

$$\alpha_{final} = q_0 \xrightarrow{5} q'_0 \underbrace{\xrightarrow{i/o,x:=2}}_{t_1} q_1 \xrightarrow{2} q'_1 \underbrace{\xrightarrow{to[x]/o,y:=3}}_{t_2} q_2 \xrightarrow{1} q'_2 \underbrace{\xrightarrow{i/o,z:=1}}_{t_3} q_3 \xrightarrow{1} q'_3 \underbrace{\xrightarrow{to[z]/o}}_{t_4} q_4 \xrightarrow{1} q'_4 \underbrace{\xrightarrow{to[y]/o}}_{t_5} q_5$$

4.2 Example run of our Algorithm

4.3 Correctness of the algorithm

We know we can find the timer values for some original state. Then we also know we can extend it [TODO] induction. If from induction it follows that indeed we also know the proper timer values, then it must be that we do actually know all timer values for any run of length k.

Chapter 5

Conclusions

In this chapter you present all conclusions that can be drawn from the preceding chapters. It should not introduce new experiments, theories, investigations, etc.: these should have been written down earlier in the thesis. Therefore, conclusions can be brief and to the point.

Bibliography

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

[2] Paul Fiterău-Broştean, Ramon Janssen, and Frits Vaandrager. Combining model learning and model checking to analyze tcp implementations. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 454–471, Cham, 2016. Springer International Publishing.

[3] Frits Vaandrager, Roderick Bloem, and Masoud Ebrahimi. Learning mealy machines with one timer. In Alberto Leporati, Carlos Martín-Vide, Dana Shapira, and Claudio Zandron, editors, *Language and Automata Theory and Applications*, pages 157–170, Cham, 2021. Springer International Publishing.

[4] Frits W. Vaandrager, Bharat Garhewal, Jurriaan Rot, and Thorsten Wißmann. A new approach for active automata learning based on apartness. *ArXiv*, abs/2107.05419, 2021.

Appendix A

Appendix