Andrea Pferscher[1], BSc

Active Model Learning of Timed Automata via Genetic Programming

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor

Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Bernhard K. Aichernig

Institute of Software Technology (IST)

Graz, 16. Mai 2019

[1] E-mail: pferscher.andrea@gmail.com

AFFIDAVIT

Abstract

In many software systems the access to internal components is limited. This limited access impedes the quality assurance of such black-box software systems. Therefore, the model learning of black-box systems becomes an increasing topic in software engineering. With learning we can create behavioral models of such systems which can be used for system verification.

This thesis presents an active learning approach for timed automata via genetic programming. Genetic programming is a machine learning technique that provides the possibility to automatically develop programs. We use genetic programming to learn models of real-time systems. For this, we use an active learning approach which requires a conformance check between the timed system and the learned automaton. The bottleneck of this conformance check is the required number of tests. Especially for real-valued timed systems the number of tests to cover the entire state space is infinite. Our approach overcomes this problem by using a conformance testing approach based on heuristics. This conformance testing approach uses tests which are generated by random walks through the learned automaton.

Our active learning approach is an extension of the passive learning approach proposed by Tappler et al. that also learns timed automata via genetic programming. Our goal is to optimize the costs of learning by minimizing the amount of training data since genetic programming approaches are often time-consuming and therefore expensive.

Our evaluation shows that we can achieve significant improvements compared to the passive learning approach of Tappler et al. This evaluation includes 43 different timed systems and 18 900 learned timed automata. Three examples of the timed systems are from the industry and the remaining 40 examples are randomly generated systems. The examples contain up to 20 locations. The results of our evaluation show that we can correctly learn all timed systems with only 100 tests. Additionally, we can decrease the runtime of the genetic programming approach. In one example we are able to decrease the median training set size by a factor of nearly ten and the learning runtime in another example by a factor of more than 800.

Keywords: active automata learning, passive automata learning, genetic programming, timed automata, conformance testing, test-case generation.

Kurzfassung

In vielen Software-Systemen ist nur ein eingeschränkter Zugriff auf einzelne Komponenten vorhanden. Durch den eingeschränkten Zugriff wird die Qualitätssicherung und Verifikation von solchen Software-Systemen erschwert. Dies ist ein Grund für das steigende Interesse am Lernen von Modellen eines Black-Box-Systems in vielen Bereichen der Softwareentwicklung. Diese gelernten Modelle können in weiterer Folge für die Verifikation dieser Systeme verwendet werden.

Diese Arbeit stellt einen aktiven Lernansatz für Timed Automata dar, wobei das Lernen über genetische Programmierung erfolgt. Genetische Programmierung kommt aus dem Bereich des maschinellen Lernens und bietet die Möglichkeit einer automatischen Programmentwicklung. In unserem Ansatz nutzen wir genetische Programmierung um Systeme zu lernen, welche durch zeitliche und reellwertige Variablen eingeschränkt sind. Wir verwenden dazu einen aktiven Lernansatz. Für die Umsetzung des aktiven Lernansatzes benötigen wir die Möglichkeit das gelernte Modell und das System miteinander zu vergleichen. Dieser Vergleich basiert auf einer endlichen Menge von Tests. Dabei stellt die Anzahl der benötigten Tests für das aktive Lernen eine Schwachstelle dar. Vor allem für zeitlich angepasste Systeme, die die Zeitkomponenten in reellen Werten darstellen, ist die Anzahl der mögliche Zustände unendlich. Unser Ansatz wirkt diesem Problem entgegen, indem unser Testverfahren auf Heuristiken basiert. Die verwendeten Tests in diesem Verfahren wurden durch Zufallswege in den gelernten Automaten erstellt.

Unser aktiver Lernansatz ist eine Erweiterung des passiven Lernansatzes, welcher von Tappler et al. entwickelt wurde. Dieser passive Ansatz lernt mithilfe von genetischer Programmierung Timed Automata. Unser Ziel ist die Optimierung der Lernkosten. Diese Optimierung erfolgt durch die Reduktion der Trainingsdaten, da die Menge der Trainingsdaten im direkten Zusammenhang mit dem Zeitaufwand und damit auch den Kosten steht.

Unsere Evaluierung zeigt, dass wir im Vergleich zu dem passiven Lernansatz von Tappler et al. merkliche Verbesserungen erzielen können. Wir evaluierten dabei 43 unterschiedliche, zeitliche gesteuerte Systeme. Die Evaluierung des aktiven Lernansatzes für diese Systeme basiert auf 18 900 gelernten Timed Automata. Drei dieser Systeme stellen Beispiele aus praktischen Anwendungen dar und die restlichen 40 Systeme enthalten willkürlich erstellte Eigenschaften. Dabei können diese Beispiele bis zu 20 verschiedene Knoten enthalten. Die Ergebnisse unserer Evaluierung zeigen, dass wir in der Lage sind zeitlich gesteuerte Systeme mit nur 100 Tests zu erlernen. Zusätzlich können wir die benötigte Laufzeit des Algorithmus verringern. In einem Beispiel können wir den Median der benötigten Trainingsmenge um den Faktor zehn reduzieren und für ein anderes Beispiel wird die Laufzeit um einen Faktor größer als 800 reduziert.

Schlagworte: Aktives Automaten-Lernen, Passives Automaten-Lernen, Genetische Programmierung, Timed Automata, Testgenerierung, Konformitätstestung

Acknowledgements

I would like to say a big thank you to everyone who accompanied me during my studies. Especially, I would like to thank those persons with which I had interesting conversations and mastered many tricky assignments.

Special thanks to my supervisor Bernhard K. Aichernig, who provoked my curiosity about software-testing and quality assurance with his impressive commitment to teaching. I would like to thank him for the discussions about new interesting ideas and possible improvements for the thesis.

Furthermore, I would also like to thank Bernhard K. Aichernig for recommending the workshop about timed automata. Regarding this workshop, special thanks go to Kim G. Larsen for the interesting lessons and the inspiring talks about timed automata. Without this workshop this thesis would possibly not exist.

One important reason for the achieved results of this work is the comprehensive support of Martin Tappler. I would like to thank him for the continuous possibility to ask questions, which he answered always immediately and precisely. At the same time I would like to thank him for the functional support, but also for giving me a comprehensive insight in scientific working. Special thanks to Martin Tappler for the pleasant collaboration and for the provision of time that was required to co-supervise this thesis.

Special thanks to my parents, Helga und Josef Pferscher, who always supported me in my studies. Without their support I would not have been able to study in this way. Furthermore, I would like to thank my sister, Martina Pferscher, for the continuous encouragement and support throughout my entire life. Moreover, special thanks to my good friend and colleague Angela Promitzer for the pleasant collaborations during my studies and for the correction of this thesis. Finally, I would like to thank my partner, Florian Schwarz, for the mental support and the motivation during the entire time and for the precise correction of this thesis.

Andrea Pferscher
Graz, Austria, 16. Mai 2019

Danksagung

Ich bedanke mich herzlich bei allen, die mich während des Studiums begleitet haben. Vor allem danke ich jenen Personen, mit denen ich im Laufe des Studium spannende Diskussionen führen und gemeinsam so manch knifflige Aufgabe lösen durfte.

Ein besonderer Dank gilt meinem Betreuer Bernhard K. Aichernig. Aufgrund seines Engagement im Bereich der Lehre wurde ich auf die Bereiche Software-Testing und Qualitätsicherung aufmerksam. Ich danke ihm vor allem für die interessanten Ideen und Anmerkungen zur Verbesserung dieser Masterarbeit, welche immer wieder in aufschlussreichen Gesprächen entstanden sind.

Des Weiteren danke ich Bernhard K. Aichernig auch dafür, mich auf die Möglichkeit zur Teilnahme eines Workshops zu Timed Automata hinzuweisen. Ein großer Dank gilt dabei auch Kim G. Larsen für die interessante Aufbereitung des Themas und für das Regen meines Interesses für Timed Automata. Ohne dieses Seminar würde diese Arbeit in dieser Form vermutlich nicht existieren.

Ein ausschlaggebender Grund für die Erreichung der Resultate dieser Arbeit gehen auf die umfangreiche Unterstützung von Martin Tappler zurück. Ich danke ihm dafür, dass ich ihm jederzeit Fragen stellen durfte und er mir diese auch umgehend sehr gewissenhaft beantwortet hat. Dabei danke ich nicht nur für eine fachliche Unterstützung, sondern auch für den umfangreichen Einblick in das wissenschaftliche Arbeiten. Ich danke Martin Tappler auch für die angenehme Zusammenarbeit und für die Aufwendung der Zeit, die er sich für die Betreuung dieser Arbeit nahm.

Ein ganz besonderer Dank gilt meinen Eltern, Helga und Josef Pferscher, die mich meine ganze Studienzeit unterstützt haben und ohne die das Studium in dieser Form nicht möglich gewesen wäre. Weiters danke ich meiner Schwester, Martina Pferscher, für die ständige Ermutigung und Unterstützung auf meinem gesamten Lebensweg. Außerdem danke ich meiner guten Freundin und Studienkollegin Angela Promitzer für die angenehme Zusammenarbeit während des gesamten Studiums und für die Korrekturvorschläge für diese Arbeit. Meinem Partner, Florian Schwarz, danke ich für die mentale Unterstützung und Motivation während der gesamten Zeit und für die gewissenhafte Korrektur meiner Arbeit.

Andrea Pferscher
Graz, Österreich, 16. Mai 2019

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

min	minutes
CAS	Car Alarm System
DFA	deterministic finite automaton
GA	Genetic Algorithm
GP	Genetic Programming
GUI	Graphical User Interface
LTS	Labeled Transition System
MAT	Minimally Adequate Teacher
SUT	System Under Test
SVG	Scalable Vector Graphic
TA	Timed Automaton
TTS	Timed Transition System

1 Introduction

1.1 Motivation

One of the central challenges in software engineering is to keep up with the tremendously fast development of new software systems. Additionally, the number of requirements on the software systems rise, which in turn leads to increasingly complex software systems. Despite the increase of complexity, software systems should be reliable. Reliability is especially important for critical systems, where an incorrect behavior has a huge impact on other systems. This increases the need for rigorous quality assurance in software engineering. However, the quality assurance and maintenance of highly complex systems is a challenging task. Therefore, we are in need of new sophisticated approaches to ensure quality of software systems.

In addition to the increasing complexity, Peled et al. [43] stress that the insight to software systems is often limited. Such limitations are often found when software systems use third party components. These limited insights in composed software systems are a new challenge in the field of quality assurance. To overcome this challenge, Peled et al. propose *Black-Box Checking* which provides a formal method to check properties of systems where the internal structures are unknown. In their approach, they use learning techniques to infer a behavioral model of the underlying black-box system.

Especially due to the work of Peled et al. [43] learning of systems has become an active research area in quality assurance. One promising solution in this research area is the open-source framework *LearnLib* [27] which supports several learning algorithms. One of these implemented algorithms is the $L*$ algorithm by Angluin [9]. Angluin's $L*$ algorithm was already introduced in the 1980s and is still the base for many learning approaches.

The main focus of the LearnLib framework is the learning of finite-state automata, e.g. deterministic finite automata (DFAs) or Mealy Machines. However, there is less literature on the learning of more feature rich automata. Particularly time is one of the key components of many systems, but interestingly there is little work on learning of timed models. This includes, e.g., procedures that take a certain time or outputs that are observable after a certain amount of time. Such aspects of timing can have a significant impact on the behavior of a system and are therefore crucial factors in the reliability of real-time systems. Alur and Dill [7] introduced the idea of Timed Automata (TAs) which extend finite-state automata with clocks. Based on these clocks, timed restrictions on the behaviour can be defined. Therefore, the modeling of TAs is a powerful method to model and verify real-time systems.

In this thesis, we introduce a learning approach for TAs which uses Genetic Programming (GP). GP [31] is a machine learning technique which uses the basic idea of evolution and natural selection. In nature we see that species develop over thousands of years and adapt their behavior and characteristics over the years to changing environmental conditions. The individuals which adapt best to the changing conditions survive and further improve the population. The same concept is used in GP where a population of programs evolves by mutation and crossover operations. The target is to automatically develop a program with a desired behavior. In our case we use GP to learn a Timed Automaton (TA) which should represent a behavioral model of a black-box timed system.

1.2 Research Problem and Goals

The goal is to improve the already existing GP approach for TAs which was proposed by Tappler et al. [47]. Their work implements a passive learning algorithm. A passive algorithm learns a behavioral model from a given set of test-cases. They propose that their work can be extended to an active learning approach.

An active learning approach generates a test set during learning. This test set is iteratively improved by the active learning approach. The improvement of the test set is done according to the currently learned model. In the literature, we find different active learning approaches. The base for many active approaches is Angluin's L^* algorithm which requires an equivalence oracle. However, performing an equivalence check on black-box timed systems is impossible since the time component introduces an infinite number of possible states. In literature [29, 20, 21] various approaches that actively learn timed systems with L^* are described. However, these approaches have restrictive assumptions on the learned timed systems and additionally a high complexity for timed systems with more locations or clocks.

We implement an approximate equivalence oracle via testing, thus following a similar approach as Walkinshaw et al. [53]. Their approach learns the behavioral model with a set of tests, which is extended iteratively. The tests are selected based on a predefined conformance relation between the timed system and the corresponding TA. In this selection process our goal is to find counterexamples that show that the learned TA is incorrect.

Tappler et al. [47] showed that they can correctly learn timed systems via GP. Moreover, they learn timed systems which have less restrictive assumptions than other approaches. In this thesis, we compare the results and performance of our active approach with the passive approach presented by Tappler et al. This evaluation uses examples from the industry as well as randomly generated timed systems with several locations and clocks.

1.3 Structure

This thesis is composed of six chapters.

Chapter 2 comprises the preliminaries for this work. First, we discuss the concepts for our active learning approach. Afterwards, we define the syntax and the semantics of the used TAs. Furthermore, we introduce conformance testing, which is the basis for our conformance checks. Afterwards, we discuss the main concept of GP and the used extensions. Finally, this chapter explains the functionality of the passive learning approach.

Chapter 3 introduces our active learning approach. First, we discuss the required conformance relation for timed systems which defines the criterion for the test selection. Second, we give an overview of the basic procedure of our algorithm. Then, we discuss different alternatives for the required initial model of the timed systems. Finally, we introduce the test selection and generation procedure in more detail.

Chapter 4 introduces the created Java implementation. In this chapter we dicuss the functionality of the used modules. Afterwards, this chapter comprises a brief manual of the created Graphical User Interface (GUI).

Chapter 5 discusses the results of the performed evaluation. This evaluation contains results of 18 900 learned TAs. First, we discuss timed systems that are inspired by industrial applications. Second, we introduce the results of the evaluation with randomly generated timed systems.

Chapter 6 concludes this thesis. In this chapter we first discuss related work. Afterwards, we state possible extensions for the future work. Finally, we summarize the thesis and provide a conclusion.

2 Preliminaries

This chapter discusses the concepts, which are the base for the remaining chapters of this thesis.

2.1 Set Notation

We define a *set* as a collection of elements, where every element occurs at most once. Unlike elements in a set, elements in a *multiset* can occur more than once [11]. The *multiplicity* denotes how many times the element occurs in the multiset. Let S be a multiset, $S(x)$ returns the multiplicity of the element x in the multiset S. Furthermore, according to the notation proposed by Blizard [11], we denote the union of two multisets S, T by $S \uplus T$.

2.2 Active Automata Learning

In software engineering, the aim of automata learning is usually to reverse engineer an automaton model of a system. In the literature, this reverse engineering process is denoted as learning. Furthermore, we refer to the reverse engineered system as System Under Test (SUT). The learning process uses a set of execution traces to reverse engineer the system. An execution trace is a record of a single execution in the system. To learn a system, we require a set of execution traces which covers the behavior of the system. As a result, the reverse engineering of an automaton can only be as good as the underlying set of execution traces. Therefore, the generation of execution traces is a crucial part of the learning process. With respect to the generation of execution traces, we distinguish between a *passive* and an *active* learning approach.
In the *passive* approach, all execution traces are created before the learning process starts. Furthermore, this creation process is independent from the learning process. Hence, in a passive approach, we learn the automaton based on a given set of execution traces. The advantage of this approach is that it only requires an interaction with the SUT prior to the start of the learning process. Therefore, no further access to the SUT after the trace generation is needed. However, this approach is heavily dependent on the quality of the trace generator. If the generator is weak, the traces may not contain all relevant information about the behavior of the system. In addition, according to Walkinshaw et al. [53], to cover every behavioral aspect of the system, the passive approach requires a large set of execution traces.
An *active* approach retrieves knowledge on demand by asking questions about the SUT. Instead of knowing everything in advance we can ask an oracle. We assume that such an oracle can correctly answer questions about the SUT. Therefore, active approaches require less information about the SUT at the beginning of the learning process. Furthermore, the active approach creates the possibility to select only the traces which are required to accurately learn a model. Hence, this technique usually requires fewer tests, thus overcoming the limitation of the passive approach.

Already in 1987 Angluin [9] introduced the active learning algorithm $L*$. This algorithm learns the DFA of a regular language L. The basic principle of Angluin's approach is that the learning algorithm asks a so-called *teacher* questions about the regular language. The *teacher* is an oracle, which dispenses information about the regular language. Angluin introduces a Minimally Adequate Teacher (MAT) framework which provides answers to two different types of questions. The first type of questions are *membership queries*, where the teacher simply answers whether a word composed of a provided alphabet is part of the language. The teacher answers this question simply with *yes* or *no*. The second type of questions are *equivalence queries*, which check the equivalence of a hypothesis model and the language. If the language of the hypothesis is equivalent one possible solution is found. Otherwise the teacher provides a counterexample which reveals the difference between the learned hypothesis automaton and the language to be learned.

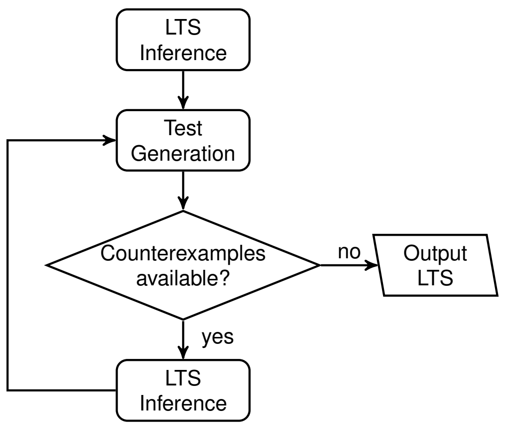


Figure 2.1: General procedure that is used in the iterative refinement approach for LTS proposed by Walkinshaw et al. [53]

.

Using these queries, $L*$ correctly infers a DFA of the regular language. However, in order to produce a correct DFA, $L*$ always has to inquire all possible states. This may lead to a vast amount of required questions to identify a hypothesis model. According to the complexity analysis of Angluin [8] the number of membership queries to learn a DFA with n states, where n is known, is exponential in n. However, Angluin [8] highlights that a DFA can be inferred in polynomial time if the set of traces provides a representative image of the language.

Dupont et al. [17] picked up on the idea of decreasing the number of queries to a size which can be described by a polynomial function. This approach is based on a *state-merging* technique to infer a language. *State-merging* is a passive automaton learning alternative to the $L*$ algorithm. Lang et al. [33] describe the basic principle of state-merging techniques, which is to fold up equal nodes of a tree until no more folding is possible. The tree is an acceptor for a set of strings of the language that is learned. The completely folded tree represents the final hypothesis model. Dupont et al. [17] extend this basic state-merging approach by considering membership queries which are answered by an end-user. In contrast, Walkinshaw et al. [53] hold the view that questioning a human might not be sufficient, since we cannot assure that humans have a comprehensive knowledge about the SUT. As a consequence, Walkinshaw et al. [53] proposed a fully automatic iterative refinement approach to learn a Labeled Transition System (LTS) of the SUT which uses model-based testing to generate execution traces. The learning of the LTS is done via state-merging.

Figure 2.1 shows the basic process of the Walkinshaw et al. [53] iterative refinement approach. First, the algorithm starts with the initial inference of a hypothesis based on a provided set of execution traces. We denote an execution trace in an LTS as test. After the creation of an initial hypothesis, the algorithm generates a set of tests. These tests are only generated if counterexamples are found that show that the learned LTS is different from the SUT. If counterexamples exist, the algorithm checks if the current test fails. A test case fails if the execution trace cannot be reconstructed in the learned LTS, otherwise the test passes. A failing test leads to a refinement of the LTS. Independent of whether a new LTS is inferred or not, a new set of tests is generated. The algorithm terminates in the case that no counterexamples are found and returns the iteratively refined LTS.

The novelty of this approach from Walkinshaw et al. [53] is that they use model-based testing to refine the hypothesis model. In model-based testing [44], tests are retrieved from a behavioral model and used to check the behavior of the SUT. However, models are created on a more abstract domain than the SUT. Therefore, the abstract tests, which are retrieved from the model, have to be translated to the domain of the SUT. These translated test can then be executed on the SUT. Walkinshaw et al. use this approach to retrieve tests form the LTS and run these tests against the SUT. In this approach of Walkinshaw et al. the LTS represents the behavioral model, but the retrieved tests are not used to reveal a wrong behavior of the SUT. Instead, a failing test shows that the assumed model is wrong and should be refined according to the failing test. Consequently, model-based testing, which is based on heuristics, provides a possibility to generate membership queries similar to Angluin's $L*$ algorithm.

2.3 Timed Automata

In many systems time is a critical component. Considering time in the model of a system may provide new opportunities to perform checks based on issues due to wrong timings. To represent time, we require an approach to model a timed system. One solution for the representation of timed systems is proposed by Alur and Dill [7]. They introduce Timed Automata (TAs) which are finite state machines equipped with real-valued variables. The finite state machines consist of locations which are connected with edges. The edges are labeled with input and output actions, and guards. Guards restrict which edges can be used. The real-valued variables store time values which represent real-time clocks. When in a location time elapses all variables increase simultaneously. Clock variables are used to build guards for edges. These guards provide a possibility to restrict the behavior of a finite state machine by considering the clock values.

Since this thesis builds upon on the work of Tappler et al. [47], this section uses equal definitions and assumptions on TAs. Let $\mathcal{C}$ be a finite set of clocks where the time domain is a set of non-negative real numbers $\mathbb{R}_{\geq 0}$. The set of clock constraints, which are called guards, is denoted as $\mathcal{G}(\mathcal{C})$ over $\mathcal{C}$. A guard g of the set $\mathcal{G}(\mathcal{C})$ labels an edge of the TA and shall be written as conjunction $\bigwedge c \oplus k$, with $c \in \mathcal{C}$, $\oplus \in \{<, \leq, \geq, >\}$, $k \in \mathbb{N}$. Note that guard conjunctions may be empty. Therefore, we denote non existing constraints by $\top$, which means that these guards are always satisfied.

In addition, edges are labeled with actions. We denote Σ as the set of actions. Furthermore, we distinguish between the set of input actions Σ_{I} and output actions Σ_{O}. Former are equipped with a questions mark "?" at the end, whereas outputs are suffixed by an exclamation mark "!". A TA over $(\mathcal{C}, \Sigma)$ is a triple $\langle L, l_0, E\rangle$ consisting of a finite set of locations L with l_0 as the initial location and a set of edges denoted by E with $E \subseteq L \times \Sigma \times \mathcal{G}(\mathcal{C}) \times 2^{\mathcal{C}} \times L$. Let $2^{\mathcal{C}}$ be the power set of $\mathcal{C}$, i.e., $2^{\mathcal{C}}$ is the set of all subsets of $\mathcal{C}$. We denote $l \xrightarrow{g,a,r} l'$ as an edge $\langle l, g, a, r, l'\rangle \in E$ with $g \in \mathcal{G}(\mathcal{C})$, $a \in \Sigma$ and $r \in 2^{\mathcal{C}}$. Here, r denotes a set of clock resets to zero.

Example 1 (Syntax of Smart Light Switch TA)**.** Figure 2.2 shows an example of a Smart Light Switch, which an adaption of the Touch Sensitive Switch presented by Hessel et al. [24]. In Figure 2.2 is represented as a TA. The light switch can either be pressed or released which offers two possibilities to interact with the system, i.e., $\Sigma_{\mathrm{I}} = \{press?, release?\}$. Depending on how long the switch is pressed three different outputs are observable, i.e., $\Sigma_{\mathrm{O}} = \{touch!, starthold!, endhold!\}$. We can model the behavior of the system with one clock $c \in \mathcal{C}$. Since we have only one clock $c \in \mathcal{C}$ the power set $2^{\mathcal{C}}$ is $\{\{\}, \{c\}\}$, i.e. we can either reset no clock or the clock c. The TA has five locations $L = \{l_0, \ldots, l_4\}$, where l_0 is the initial location. These locations are connected via seven edges $E = \{l_0 \xrightarrow{\top, press?, \{c\}} l_1, \ldots\}$.

2.3.1 Operational Semantics

We define the semantics of a TA $\mathcal{T} = \langle L, l_0, E\rangle$ over $(\mathcal{C}, \Sigma)$ by a Timed Transition System (TTS). Let $\mathbb{R}^{\mathcal{C}}_{\geq 0}$ be the set of all clock valuations over $\mathcal{C}$. A clock valuation $\nu \in \mathbb{R}^{\mathcal{C}}_{\geq 0}$ is a mapping $\nu : \mathcal{C} \to \mathbb{R}_{\geq 0}$ which assigns a real value to each clock. A TTS $[\![\mathcal{T}]\!] = \langle Q, q_0, \Sigma, T\rangle$ consists of a set of states $Q = L \times \mathbb{R}^{\mathcal{C}}_{\geq 0}$, the initial state q_0, and transitions T with $T \subseteq Q \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Q$. A state $q \in Q$ is denoted as the pair (l, ν) with the location l and the clock valuation ν. We define $\mathbf{0}_{\mathcal{C}}$ as the assignment of 0 to every clock. The initial state q_0 is $(l_0, \mathbf{0}_{\mathcal{C}})$. The evolution of time is denoted by increasing the valuation ν by a delay $d \in \mathbb{R}_{\geq 0}$. Therefore, we increase the valuation ν for every clock $c \in \mathcal{C}$ by the delay d which we denote as $(\nu + d)(c) = \nu(c) + d$. Furthermore, let r be a subset of C, we denote resets by $\nu[r]$, where every $c \in r, \nu[r](c) = 0$ and for every $c \in \mathcal{C} \setminus r, \nu[r](c) = \nu(c)$.

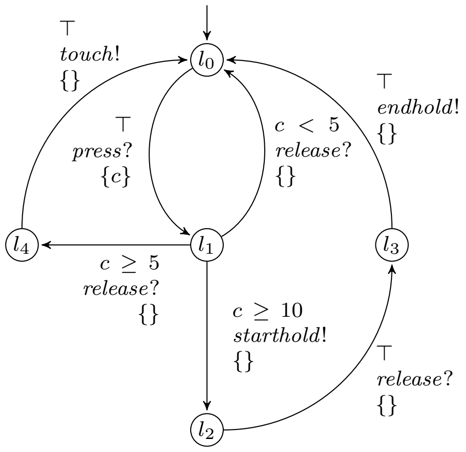


Figure 2.2: This TA is a representation of the Smart Light Switch, which is an adaption of the Touch Sensitive Switch presented by Hessel et al. [24].

We shall write $\nu \models g$ if the clock valuation ν satisfies the guard $g \in \mathcal{G}(\mathcal{C})$. A TTS distinguishes between two types of transitions: timed and discrete transitions. The former, on the one hand, are transitions where a delay is added but the location l stays unchanged. We denote such a transition by $(l, \nu) \xrightarrow{d} (l, \nu + d)$ with the delay $d \in \mathbb{R}_{\geq 0}$. Discrete transitions, on the other hand, may lead to a location change and always follow an edge $l \xrightarrow{g,a,r} l'$. Formally, we denote such a transition as $(l, \nu) \xrightarrow{a} (l', \nu[r])$ for an edge $l \xrightarrow{g,a,r} l'$ such that $\nu \models g$.

2.3.2 Timed Traces

We interact with a TA by executing inputs. In timed systems, however, we also have to consider the point in time when the input was executed. We denote the sequence of inputs that are executed at specific points in time as *test sequence*. Formally, let a test sequence ts be an ascendingly ordered sequence of time stamps t_j and inputs i_j, i.e., $ts = t_1 \cdot i_1 \cdots t_n \cdot i_n \in (\mathbb{R}_{\geq 0} \times \Sigma_I)^*$ with $\forall j \in \{1, \ldots, n-1\} : t_j \leq t_{j+1}$. If ts is applied to a TA, the system may produce outputs. This sequence of inputs and outputs at a specific time is denoted as timed trace $tt \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$. Like ts, the timed trace $tt = t_1 \cdot a_1 \cdots t_n \cdot a_n$, with $t_j \in \mathbb{R}_{\geq 0}$ and $a_j \in \Sigma$, is sorted in ascending order with respect to the timestamp. We can extend a timed trace tt by adding a time stamp $t \in \mathbb{R}_{\geq 0}$ and the corresponding action $a \in \Sigma$ to the timed trace. We denote this concatenation of a timed trace tt, a time stamp $t \in \mathbb{R}_{\geq 0}$ and an action $a \in \Sigma$ by $tt \cdot t \cdot a$. The size of a timed trace $tt = t_0 \cdot a_0 \cdots t_{\mathrm{n}-1} \cdot a_{\mathrm{n}-1}$ is n elements, where one element is a pair of a time stamp $t_i \in \mathbb{R}_{\geq 0}$ and an action $a_i \in \Sigma$, with $i \in [0, n)$. We denote the size of a timed trace by $|tt| = n$.

Aichernig et al. [3] denote a run of a TTS $[\![\mathcal{T}]\!] = \langle Q, q_0, \Sigma, T \rangle$ as a sequence of delay transitions and discrete transitions in form of

$$(l_0, \nu_0) \xrightarrow{d_0} (l_0, \nu_0 + d_0) \xrightarrow{a_0} (l_1, \nu_1) \xrightarrow{d_1} (l_1, \nu_1 + d_1) \xrightarrow{a_1} (l_2, \nu_2) \xrightarrow{d_2} \cdots$$

.

Here, the sequence of delays $d_i \in \mathbb{R}_{\geq 0}$ and actions $a_i \in \Sigma$ can be transformed to a timed trace $tt = t_1 \cdot a_1 \cdots t_i \cdot a_i \cdots t_n \cdot a_n$ with all previous delays $d_1 \cdots d_j$ summed up to the time stamp $t_i \in \mathbb{R}_{\geq 0}$.

Example 2 (Timed Trace of Smart Light Switch TA)**.** We can informally describe the semantics of Figure 2.2 in the following way: If we are in the initial location l_0 and *press* the switch, we go to the location l_1. By executing this action we reset the clock c to zero. The switch now offers three different scenarios. The first scenario is that the switch is *released* before the time evolves more than five time units. In this case we return to the initial location l_0. The second scenario occurs if we hold the switch for at least ten time units. After ten time units the system outputs $starthold!$ and immediately after a release of the switch an $endhold!$ output is observable and we return to l_0. The last scenario only occurs if the switch is held between five and ten time units. A release in this time frame leads to a $touch!$ output and, as in every other scenario, we return to the initial location l_0. The following sequence shows a possible run of the TTS:

$$(l_0, \{c \mapsto 0.0\}) \xrightarrow{10.8} (l_0, \{c \mapsto 10.8\}) \xrightarrow{press?} (l_1, \{c \mapsto 0.0\}) \xrightarrow{7.6} (l_1, \{c \mapsto 7.6\}) \xrightarrow{release?}$$

$$(l_4, \{c \mapsto 7.6\}) \xrightarrow{0.0} (l_4, \{c \mapsto 7.6\}) \xrightarrow{touch!} \cdots$$

2.3.3 Assumptions on Timed Systems

In the previous Section 2.3.2 we describes the application of test sequences to timed systems in order to generate timed traces. However, to ensure a proper testability of timed systems certain properties must hold true. Therefore, we make some assumptions on timed systems. The assumptions we make are based on those originally proposed by Springintveld et al. [46] and similar to those adapted by Hessel et al. [24] and Tappler et al. [47]. The following properties are based on the semantics of a TA and must hold $\forall q, q' \in Q$, $\forall a \in \Sigma$, $\forall i \in \Sigma_I$, and $\forall o, o' \in \Sigma_O$:

Determinism. A TA is deterministic iff $q \xrightarrow{a} q'$, $q \xrightarrow{a} q''$ then $q' = q''$.

Input Enabling. A TA is input enabled if it is possible to execute every input action in every state, formally, iff there $\exists q' : q \xrightarrow{i} q'$.

Output Urgency. Output urgency exists if an output action emerges without delay, i.e. iff $\exists q' : q \xrightarrow{o} q'$ and $\nexists q'' : q \xrightarrow{d} q''$ with $d \in \mathbb{R}_{\geq 0}$.

Isolated Outputs. An isolated output exists if an output action is enabled. Then, no other output actions in the same state are possible. Formally, if $\exists\, q', q'' : q \xrightarrow{o} q'$ and $q \xrightarrow{o'} q''$ then $o = o'$.

The aim is to test the conformance of the SUT with the hypothesis. Hessel et al. [24] suggest that these assumptions improve the testability of the SUT as well as the conformance checking for the generated hypothesis. For example, according to Tappler et al. [47], input enabledness extends the range of acceptable test sequences, since in every state $q = (l, \nu)$ every input action is possible. To provide a clear representation of TAs, we do not depict any edges of input-self-loops in figures of TAs. For example, in Figure 2.2 it is possible to perform an input-self-loop by performing the input $release?$ in location l_0. However, we do not depict such edges, since we want to avoid overloaded figures.

Furthermore, in timed systems, we may observe locations where the length of the stay is limited by a certain amount of time. Other definitions of TAs, see Section 2.3.4, consider invariants for locations. These invariants act like guards for locations and can, therefore, limit the sojourn time in a location, e.g. they may specify that a clock variable must not exceed five time units at a location. Output urgency can either be modeled by invariants or deadlines. Deadlines define a point in time where an action must happen, i.e. no further time progress is possible. According to Bornot et al. [12], deadlines provide a possibility to enable edges of a TA by stopping the progress of time. Like in our work, they use edges that are labeled with actions, guards and clock resets, but their labels of the edges also contain deadlines. In this thesis, we assume that output actions are eager. Thus the guard represents the corresponding deadline. This means that an output is triggered as soon as the guard of an edge is satisfied.

Example 3 (Urgent Outputs)**.** The TA in Figure 2.2 outputs $starthold!$ in location l_1 as soon as the clock variable has evolved by at least ten time units. The output $endhold!$ occurs immediately after entering the location l_3. The same behavior is observable with the output $touch!$ in location l_4. In both cases we immediately enter the location l_0 afterwards.

2.3.4 UPPAAL

The concept of equipping systems with clocks offers new possibilities to test and verify real-time systems. Larsen et al. developed UPPAAL [1] [35] which is a collection of tools for processing TAs. The name UPPAAL is a composition of the first three letters of the two involved Universities in Uppsala, Sweden and Aalborg, Denmark. The toolbox of UPPAAL provides components to model and analyze real-time systems. One of the major benefits that UPPAAL provides is easy usage, e.g., modeling can either be done via a textual or graphical interface. The analysis part includes a simulation of traces and a model-checker. While simulation helps to visualize traces and the behavior of the systems, the provided model-checking tool examines properties like the reachability in a system. [35]

2.4 Conformance Testing

In conformance testing we assess whether an implementation conforms to a specification. Hence, we compare two components: the *specification* and the *implementation*.

The aim of the *specification* is to define the expected behavior of a system. Different methods have been proposed to specify the behavior of a system. According to Alur [6], one promising technique is the creation of an automaton model. In many systems time is a crucial component and some behavioral characteristics of the system can be missed if the model does reflect time constraints. Hence, TAs are one possibility to specify a system. In the remainder of this thesis we assume that we can specify systems by TAs.

The *implementation* is the realization of a system. For example, an implementation can be pieces of hardware or the code of a software. To interact with an implementation we require an interface, where we can execute inputs and observe outputs. Through this interface we can test the implementation and check whether the behavior of the implementation conforms to the specification.

One challenge in conformance testing is that the specification is well defined by a modeling language whereas the form of the implementation rarely has constraints. To compare the implementation and the specification, we make the assumption that the implementation can be represented by a TA.Tretmans [48] provides an overview of model-based testing with respect to implementation relations $\mathbf{imp} \subseteq MOD \times SPEC$, where MOD is the set of all implementation models and $SPEC$ the set of valid expressions in the modeling language. This relation is satisfied if the implementation $\mathcal{I} \in MOD$ conforms to the specification $\mathcal{S} \in SPEC$, which is then denoted as $\mathcal{I}$ **imp** $\mathcal{S}$.

The crucial question in conformance testing is how to check whether the implementation conforms to the specification. Intuitively, the conformance relation is satisfied if two systems behave equally. To check if a behavior is equivalent, we have to execute test sequences on $\mathcal{I}$ and $\mathcal{S}$ and check whether the observed timed traces are equal. Let $\mathit{Traces}(q)$ be a function that returns the timed traces of all runs starting at the state $q = (l, \nu)$. For a TTS $[\![\mathcal{T}]\!]$ we denote $\mathit{Traces}([\![\mathcal{T}]\!])$ as the set of all timed traces starting at the initial state q_0 of $[\![\mathcal{T}]\!]$. Using this function we can define conformance as follows:

$$\mathcal{I}\,\mathbf{imp}\,\mathcal{S} \iff \mathit{Traces}([\![\mathcal{I}]\!]) = Traces([\![S]\!])$$

Using the defined conformance relation we can state when two systems are conforming. In the literature, we find other conformance relations. Two prominent examples of conformance releations for timed systems are the **tioco** [45] and the **rtioco** [23, 34] conformance relations. Both relations are based on **ioco** conformance relation proposed by Tretmans [48]. The **ioco** relation is fulfilled if the outputs produced by the implementation are a subset of the produced outputs of the specification. This means that a conform implementation cannot produce different outputs than the specification. However, in our approach we use a tighter conformance relation that is based on trace equivalence.

[1] `http://www.uppaal.org/`

Conformance testing is a powerful tool to asses the quality of a software system. However, we have to consider that this conformance check depends on a set of tests. Testing all possible cases is infeasible for many timed systems. Consequently, the set of tests is limited and, therefore, the conformance check is only as good as the used set of tests. The aim of this work is to improve the test generator, which should in turn also optimize the automaton learning process. In the next section, we will introduce our methodology of the used learning process.

2.5 Genetic Programming

Genetic Programming (GP) is a method to automatically generate a program. Therefore, GP provides the opportunity to solve a problem without explicitly programming it. GP is a further development of Genetic Algorithms (GAs) which were already introduced in the 1960s and 1970s [40]. The book *Adaptation in Natural and Artificial Systems* by Holland [25] introduces a general framework to solve problems with *computational evolution* [40]. GAs are inspired by nature and the idea of the evolutionary process. Basically, everything relies on the groundbreaking work *On the Origin of Species by Means of Natural Selection* by Charles Darwin [15] describing the evolutionary process in 1859. Darwin illustrates the evolution of species and observes that individuals which are better suited to environmental conditions are more likely to survive and reproduce. He introduces the well-known phrase "*Survival of the fittest*" which is the basic principle of natural selection.

Based on Darwin's views about common descent, GAs attempt to simulate natural selection. To apply natural selection, we need a pool of individuals from which we can pick the fittest candidates. We denote this pool of individuals as our population which is composed of candidate solutions. Given some computational problem, we generate populations of candidate solutions and evolve them over several generations. The target is to evolve a population until one individual is found which serves as an appropriate solution for the given problem. The evolution of a new generation is based on the fitness of each individual. Like the Darwinian principle of survival and reproduction, GAs favor the fittest individuals of each generation. The key to success is to enhance fitness by improving the properties of the individuals. To improve the fitness of the whole population, the fittest individuals are selected and different operations, e.g., reproduction, crossover or mutation are applied to them for the creation of a new set of individuals.

Koza [31] defines four aspects that must be considered before a Genetic Algorithm (GA) can be applied. These four aspects are: *Representation*, *Fitness Calculation*, *Parameters*, *Termination Criterion*. First, we have to determine the representation scheme for an individual of the population. The representation of an individual is a specified data structure that defines the individual. This data structure makes it possible to distinguish the individual from other individuals and saves all relevant characteristics of the individuals. Second, after the representation of the individuals is specified, we require criteria to rank the individuals. Given these criteria, we can measure the fitness of an individual and are thus able to rank them. According to Koza, the third aspect that we have to consider is that GAs comprise many different parameters and variables, e.g. population size, number of generations, etc. These parameters and variables affect the behavior of the algorithm and have to be tuned to the currently treated problem. Finally, we need to define a termination criterion. This criterion defines the point when we stop generating a new population and output the fittest individual so far.

As mentioned previously, GP is an extension of GAs, where the population consists of programs. The target of the evolutionary process is to automatically create a program for a specific problem. In every generation the fitness of each program is determined and according to the principle of natural selection the fittest programs are selected for further development. However, the determination of an appropriate fitness function is a challenging task in GP and must be chosen for each investigated problem individually. The process of creating a new generation involves the following three operations which are applied to the individuals based on their fitness [40]:

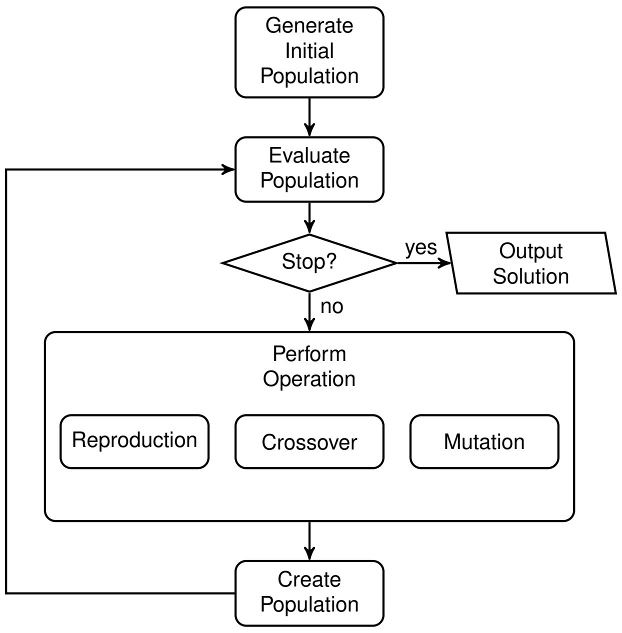


Figure 2.3: General procedure of GP

- *Reproduction*. The fittest individuals are selected and copied without modification.
- *Crossover*. Two individuals exchange parts of their program and create therefore new individuals.
- *Mutation*. A single individual is picked and some parts of this individual are modified.

Finally, the algorithm terminates if a suitable solution is found or the maximum number of generations has been reached.

2.5.1 Concepts of Design

The fundamental design of GP is usually based on the general procedure shown in Figure 2.3. Nevertheless, there exist many additional concepts to improve the problem-specific implementation. In this section, we will discuss two remarkable concepts, namely *subpopulation and migration*, and *elitism*.

Subpopulation and Migration

Nowostawski and Poli [41] found that many methods exist to implement parallelism in GP. According to them, one of the most popular methods is the evolution of multiple populations at the same time. This design principle allows to develop multiple populations simultaneously. Individual populations may develop differently which increases the explored search space for the solution. Another possibility of this idea is to exchange promising individuals within the different populations, i.e. migrate individuals. Hence, individuals only compete with individuals from their subpopulation and, based on a problem-specific selection strategy, individuals are copied to another subpopulation.

Elitism

Due to the creation and modification process of GP there exist a certain risk to destroy an already good solution. According to Mitchell [40] one way to overcome this problem is to adopt the reproduction of the elite to the next generation. The set of the fittest individuals of the former generation, i.e. the elite, is then part of the newly created population.

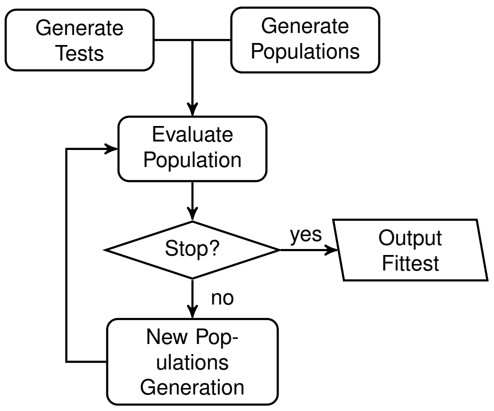


Figure 2.4: Tappler et al. [47] proposes a GP procedure that learns based on a given set of tests. The proposed GP procedure uses two different populations.

2.6 Genetic Programming for Timed Automata

The previous chapter showed that GP can be a promising method to automatically find solutions to computational problems. Using GP, Tappler et al. [47] show that TAs can also be learned automatically and propose a framework to learn TAs via GP. Figure 2.4 depicts the functioning principle of their proposed framework, which is in principle equal to the general procedure of GP shown in Figure 2.3. However, adaptions are made to adjust the general procedure of GP for the inference of TAs. In this section, we discuss these adaptions.

In Section 2.5 we discussed Koza's [31] four actions (representation scheme, fitness measurement, parameters and variables, and termination) which should be considered in the design of an application for GP. Using these four aspects, we can characterize the GP approach for TAs proposed by Tappler et al. [47]. The representation scheme of an individual of the population is a TA. The fitness of a TA is measured according to two different criteria. The first one is based on test results and the second on characteristics of the TA itself. Intuitively, the fitness increases the more tests are passed. The used tests are timed traces. A test passes if a test sequence of the timed trace executed on a generated TA produces the same timed trace on the SUT, i.e., the same outputs are generated at the same time on both systems. The function $\mathrm{sim}(\mathcal{G}, tt)$ simulates the timed trace tt on the generated TA $\mathcal{G}$ and returns a set of timed traces tts generated by $\mathcal{G}$. The set tts contains timed traces which are prefixes of tt. The evaluation of tts is done via a verdict function. If tts contains only one timed trace, a deterministic solution is found. If this single element of tts is equal to tt the test passes. If tts has more than one element and one of these elements contains the same outputs at the same time as the SUT, the function judges the test as *non-deterministic*. In all other cases the test fails. We stop executing inputs from the timed trace if a different output is generated by $\mathcal{G}$. Therefore, the timed traces are equal if they have the same length. Thus, the verdict function is denoted as follows:

$$\textsc{verdict}(tts) = \begin{cases} \texttt{PASS} & \text{if } |tts| = 1 \land tt \in tts \\ \texttt{NONDET} & \text{if } |tts| > 1 \land \exists tt' \in tts \,:\, |tt'| = |tt| \\ \texttt{FAIL} & \text{otherwise} \end{cases}$$

However, the calculation of the fitness value involves further characteristics of the generated TA $\mathcal{G}$. One of the overall goals is to find a simple solution. We denote a solution as simple if the size of $\mathcal{G}$ is small. The size is defined by the number of edges of $\mathcal{G}$. Let $\textsc{size}(\mathcal{G})$ be a function that returns the number of edges of the TA $\mathcal{G}$. In addition, beside characteristics of the automaton itself also properties of the generated tts for every tt are considered. Firstly, traces in tts with more outputs are rewarded. Secondly, traces should be deterministic and therefore every deterministic step is rewarded. However, due to the random creation of edges non-deterministic automata may be generated. To measure these properties, two functions are defined. First, $\text{STEPS}(tts)$ returns the number of inputs that can be performed deterministically while simulating tts. Second, $\text{OUT}(tts)$ returns the number of triggered outputs.

As these properties have a distinct importance they are adjusted with weights. We consider the size of $\mathcal{G}$ with the weight w_{SIZE}, the number of outputs with the weight w_{OUT} and the number of deterministic steps with w_{STEPS}. In addition we weight the verdict of the trace. Therefore, we consider three different weights: w_{PASS}, w_{FAIL} and w_{NONDET}.

The fitness value can be described by the fitness function $\text{FIT}(\mathcal{G})$. This function calculates the fitness of an automaton $\mathcal{G}$ for a set of timed traces $tt \in \mathcal{TT}$, where $tts = \text{sim}(\mathcal{G}, tt)$. Tappler et al. then defines the fitness function for $\mathcal{G}$ evaluated on set of timed traces $\mathcal{TT}$ as follows:

$$\text{FIT}(\mathcal{G}) = \sum_{tt \in \mathcal{TT}} \text{FIT}(\mathcal{G}, tt) - w_{\text{SIZE}}\ \text{SIZE}(\mathcal{G}) \text{ where} \tag{2.1}$$

$$\text{FIT}(\mathcal{G}, tt) = w_{\text{VERDICT}(tts)} + w_{\text{STEPS}}\ \text{STEPS}(tts) + w_{\text{OUT}}\ \text{OUT}(tts) \tag{2.2}$$

According to Koza [31] as a third step parameters and variables have to be defined. Parameters influence the behavior of the algorithm and therefore the output of the GP procedure. For example, normally a bigger population and more generations lead to better results, but may lead to a bad performance. Therefore, the used parameters have to be chosen carefully. There exist a number of parameters and variables which are required to learn a TA via GP. Therefore, we provide an overview of all used parameters in the Appendix B.

As a last step, the GP approach for TAs requires a termination criterion. There exist two criteria for stopping a further creation of a new population. Firstly, the algorithm terminates if a suitable TA is found. This means that a TA passes all simulations of timed traces, i.e., produces the same outputs as the SUT for all given test sequences. Furthermore, the fitness of a suitable TA must not change over g_{change} generations. This condition corresponds to the principle of creating less complicated solutions. As previously explained, the algorithm penalizes size. As a result a TA is fitter if it passes all test with a smaller size. Secondly, if the algorithm does not find a suitable solution after $g_{\max}$ generations the algorithm terminates and outputs the fittest TA evolved so far.

Based on these four steps proposed by Koza [31] we can describe the GP procedure for TAs proposed by Tappler et al. [47]. Figure 2.4 shows the sequence of events. The algorithm starts with generating the initial n_{test} test sequences. The sequence is generated by randomly selecting inputs from Σ_{I}. These inputs are executed after a certain amount of time has elapsed. The amount of this delay is selected based on a set of relevant constants of the SUT or a user-selected maximum constant. In the remainder of this thesis we denote this set of relevant constants as *hints*. We generate delays either with hints or a user-selected constant. After the generation of the n_{test} test sequences, these sequences are executed on the SUT to generate n_{test} timed traces. These timed traces will later be compared to the traces produced by the individuals. We refer to the traces from which we learn as *training set.*

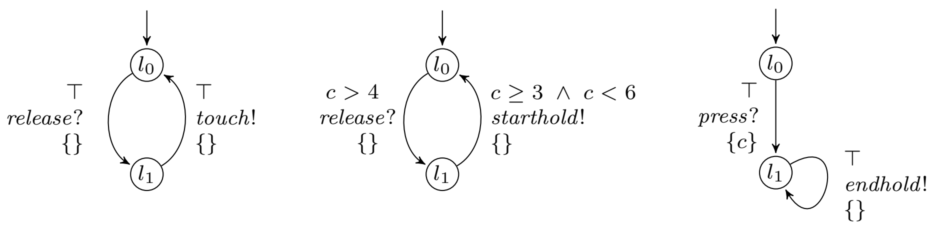
Example 4. We want to learn the TA of Figure 2.2. Therefore, we randomly generate timed traces by executing input sequences of the given alphabet after random delays. We use these timed traces as test cases for learning the TA.

$$tt_1 = (9.08, press?), (15.14, press?), (16.44, release?), (16.44, touch!) \tag{2.3}$$

$$tt_2 = (0.86, press?), (1.41, press?), (10.86, starthold!) \tag{2.4}$$

$$tt_3 = (10.94, press?), (11.36, release?), (13.68, press?), (23.68, starthold!) \tag{2.5}$$

Independent of the test generation process, the initial population is generated. The algorithm uses the principle of *subpopulation* and *migration.* Two populations are evolved simultaneously. On the one hand, there exists the so-called global population which is evaluated upon all n_{test} traces. On the



(a) First Individual **(b)** Second Individual **(c)** Third Individual

Figure 2.5: Examples of initial population that are created via GP for the learning of the Smart Light Switch of Example 1.

other hand, there is the local population which evolves on the base of the failing traces $\mathcal{T}_{\text{fail}}$ of the fittest automaton from the global population. The fittest individuals of the local population get migrated to the global population, which helps to create more diversity within the population.

For simplicity, the initial population only contains automata with two locations. Automata grow within the process of creating a new generation via mutation and crossover operations. Edges between the locations are generated randomly. However, the labels of the edges depend on predefined parameters. The input Σ_{I} and output Σ_{O} actions, the number of clocks n_{clock} and an approximation of the largest clock constant are provided.

Example 5. Figure 2.5 shows a selection of the initial population. For this example, we select these three randomly generated TAs of the GP procedure for Smart Light Switch. All initial individuals are composed of two locations. The edges are generated randomly. The labels of the edges consist of random guards and clock resets, with the maximum number of clocks given. The actions are randomly chosen from the known alphabet Σ.

After the generation process, the population gets evaluated. As previously mentioned, the fitness of a TA is based on the conformance to the initially generated timed traces and different properties of the learned TA itself. During this evaluation process, every individual of the population is assigned a fitness value. The higher the fitness value, the better is the learned solution. The individual with the highest fitness value is assumed to be the best learned solution at this point of the execution and is denoted as fittest TA. We denote the fittest TA as *hypothesis* $\mathcal{H}$.

Example 6. We want to evaluate the initial population of Figure 2.5. The basis for the evaluation are the randomly generated timed traces from the SUT of Example 4. If we execute the timed traces on the individuals of the initial population we observe their corresponding set of timed traces tts which we pass to the function VERDICT(tts). The first automaton 2.5a passes the first test in Equation (2.3), but fails the second test in Equation (2.4) and the output of the third test in Equation (2.5) is non-deterministic. The second 2.5b and third 2.5c automaton do not pass any of the tests from Example 4. Hence, the first automaton 2.5a is the fittest individual of this generation and will be copied to the next generation.

After the fitness evaluation, the approach of Tappler et al. [47] checks if one of the termination criteria is satisfied or if a new generation should be created. The development of a new generation includes the creation of two new populations: a global and local population. Both are created with akin techniques. However, there are differences since individuals from the local population are migrated to the global population. The migration is done via copying and crossover between a global and a local individual. Additionally, instead of the crossover of two individuals, mutation operations on a single individual can be applied to change the behavior of a TA. Such behavioral changes can be, e.g., adding or splitting locations, or splitting edges, or changing guards, or changing clock resets. Furthermore, the concept of elitism is applied by always keeping track of the fittest TA of the previous generation, i.e., the fittest TA of the former generation is always part of the new generation.

The GP procedure terminates either if a solution is found which passes all timed traces and cannot be further improved, or if the maximum number of generations is reached. In any case, the procedure outputs the fittest TA.

3 Active Learning via Genetic Programming

In this chapter, we present an active learning algorithm for TAs. We propose a learning algorithm that uses GP to reverse-engineer a TA from a set of timed traces. The foundation of our active learning algorithm is the passive learning technique introduced by Tappler et al. [47]. In their work, they propose that their passive learning technique is open for extensions, like an active refinement of TAs. Based on this proposal, we developed an algorithm that actively learns a TA via GP. To make the passive approach of Tappler et al. [47] active, we present a technique that uses the iterative refinement technique presented by Walkinshaw et al. [53]. For this, our active approach learns a TA from a set of dynamically selected timed traces.

This chapter is divided into five parts. First, we present the conformance relation. This conformance relation is the base for our strategy to select tests. Section 3.2 discusses the basic procedure of our active learning approach. Section 3.3 gives a brief overview of the characteristics of the initial hypothesis model that we later iteratively update. Since the selection and generation of timed traces are significant parts of our active approach, we take a closer look on both. In Section 3.4, we introduce the timed traces selection for learning. Section 3.5 presents then the generation of timed traces, which is used in the timed trace selection for learning.

3.1 Conformance Relation

Our conformance relation is based on the relation we introduced in Section 2.4. We use this conformance relation to check whether an implementation conforms to a specification. However, this method of conformance checking is dependent on the presence of a specification. According to many in the field [53, 52], a specification does not always exist. For example, Volpato and Tretmans [52] claim that often there exist no specification due to third party components. Furthermore, according to Walkinshaw [53] in many cases there is no time to create an appropriate specification due to tight time schedules in the development process. Nevertheless, we can use the conformance relation to asses whether a learned TA conforms to the SUT. We denote the learned TA as our hypothesis $\mathcal{H}$ and the SUT as implementation $\mathcal{I}$. Furthermore, we assume that the implementation can be represented as TA. To assess whether $\mathcal{H}$ conforms to $\mathcal{I}$, we define a conformance relation between $\mathcal{H}$ and $\mathcal{I}$ in form of

$$\mathcal{H}\ \mathbf{imp}\ \mathcal{I} \iff \mathit{Traces}(\llbracket\mathcal{H}\rrbracket) = \mathit{Traces}(\llbracket\mathcal{I}\rrbracket) \tag{3.1}$$

Moreover, according to Fiterau-Brostean et al. [18] this conformance relation also provides the possibility to assess the quality of the implementation, if we have access to a specification $\mathcal{S}$. Due to the transitivity of our conformance relation we can say that if $\mathcal{H}\ \mathbf{imp}\ \mathcal{I}$ and $\mathcal{H}\ \mathbf{imp}\ \mathcal{S}$ then $\mathcal{I}\ \mathbf{imp}\ \mathcal{S}$.

In more detail, $\mathcal{H}$ conforms to $\mathcal{I}$ if the sets of timed traces of $\llbracket\mathcal{H}\rrbracket$ and $\llbracket\mathcal{I}\rrbracket$ are equal. In Section 2.6 we introduced the function $\mathrm{sim}(\mathcal{G}, tt)$, which returns the timed traces generated by $\mathcal{G}$ when we execute the timed trace tt on $\mathcal{G}$. We can use the function $\mathrm{sim}(\mathcal{G}, tt)$ to test the equivalence between two systems. Considering this function we can now rewrite the conformance relation for the set of timed traces $\mathcal{TT}$ of $\llbracket\mathcal{I}\rrbracket$ to

$$\mathcal{H}\ \mathbf{imp}\ \mathcal{I} \iff \forall tt \in \mathcal{TT} : \mathrm{sim}(\mathcal{H}, tt) = \mathrm{sim}(\mathcal{I}, tt) \tag{3.2}$$

However, evaluating all possible traces of a TA is impossible, since the number of traces is infinite due to the real-valued time. Using this conformance relation for learning, we learn $\mathcal{H}$ based on a finite number of timed traces $\mathcal{TT}'$. If a timed trace $tt \in \mathcal{TT}'$ simulated on $\mathcal{H}$ as well as on $\mathcal{I}$ produces the same timed trace on both systems, $\mathcal{H}$ **passes** tt, if the produced timed traces are different then $\mathcal{H}$ **fails** tt. Considering this notation for testing we denote a timed trace as a *test* and the set of timed traces $\mathcal{TT}'$ as our *test suite*. Ideally, the following equivalence holds:

$$\mathcal{H}\,\mathbf{imp}\,\mathcal{I} \iff \forall tt \in \mathcal{TT} : \mathcal{H}\,\mathbf{passes}\,tt \tag{3.3}$$

Nevertheless, the learning of a hypothesis based on the Conformance Relation 3.3 depends on the quality of the test suite and is, therefore, only as good as the used tests of the test suite. According to Tretmans [48] the quality of a test suite can be classified based on three different criteria: *completeness*, *soundness* and *exhaustiveness*. Tretmans [48] denotes a test suite as *complete*, if it can distinguish in all cases whether the hypothesis is conforming or non-conforming. However, this is a very strong requirement and in many cases impossible to achieve. A criterion that may be easier to achieve is *soundness*. The test suite is *sound* if a correct hypothesis passes all timed traces. A hypothesis that fails at least one timed trace is definitely incorrect. However, the soundness of the test suite does not exclude that an incorrect hypothesis may pass all timed traces. Tretmans [48] defines soundness as the left-to-right implication in the conformance relation 3.3, whereas the right-to-left implication is *exhaustiveness*. *Exhaustiveness* is given if all incorrect hypotheses are detected. This criteria is basically as difficult to achieve as completeness. Furthermore, Tretmans [48] stresses that in testing these criteria are used for error detection. Therefore, soundness is based on the idea to not detect any false error, whereas exhaustiveness and completeness guarantee that all errors are detected.

Our target is to find timed traces that reveal that the learned TA behaves differently than the implementation. In the case of an incorrect hypothesis the Conformance Relation 3.2 is not satisfied. However, the set of traces is infinite. As mentioned previously, we have to limit the set of traces to make the relation applicable. The challenge is to find an appropriate test suite. Considering the properties introduced by Tretmans [48] the minimal requirement for our test suite is soundness. However, we can also improve the exhaustiveness of the test suite. The improvement of the test suite is based on finding failing traces that serve as counterexamples to the correctness of the learned TA. By the generation of counterexamples to conformance between the hypothesis and the implementation we refine the conformance relation for a finite set of traces. Based on an improved finite set of timed traces $\mathcal{TT}' \subsetneq \mathcal{TT}$ we define a restricted conformance relation in form of

$$\mathcal{H}\,\mathbf{imp}_{\mathcal{TT}'}\,\mathcal{I} \iff \forall tt \in \mathcal{TT}' : \mathrm{sim}(\mathcal{H}, tt) = \mathrm{sim}(\mathcal{I}, tt)$$

The aim of this work is to improve the generation of the test suite $\mathcal{TT}'$. We want to generate a set of timed traces that is sound and in addition also supports an efficient active learning via genetic programming. Therefore, we try to keep the test suite small, but still sound. With a conformance test in every iteration we want to detect further errors in the learned TA and improve the completeness of the test suite in every iteration.

3.2 Basic Procedure

The basic procedure of our active automata learning algorithm is to iteratively learn a TA of the SUT by adding timed traces that reveal a faulty behavior of the current model. This approach is based on the iterative refinement technique proposed by Walkinshaw et al. [53]. To ensure that we can efficiently learn a TA via GP, we have to make adaptions to Walkinshaw's iterative refinement approach. In this section we first introduce our basic procedure and stress the adaptions to Walkinshaw's approach. Afterwards, we introduce our algorithm that iteratively learns TAs.

Figure 3.1 models our active learning approach. The approach starts with a test suite which contains randomly generated tests. The tests are timed traces which are randomly generated akin to the passive learning approach proposed by Tappler et al. [47]. Based on this initial test set we learn a first model

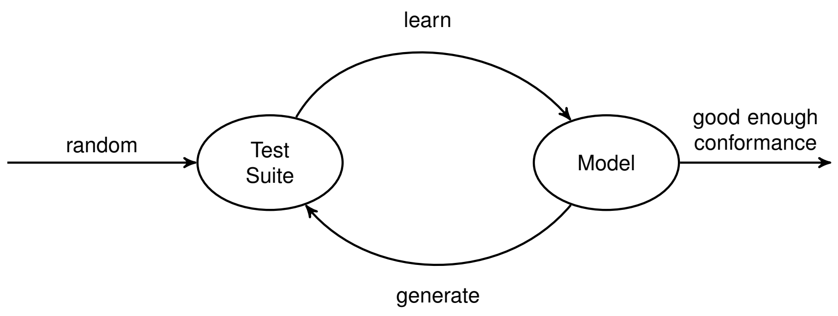


Figure 3.1: In recurring order, we learn a model and extend our test suite by newly generated counterexamples. We start with a random test suite and stop learning if we find a model with a good enough conformance.

of the SUT. We define this learned TA as our hypothesis of the SUT. After the learning of our hypothesis, we test whether the hypothesis conforms to the SUT. This conformance evaluation is done according to the conformance relation defined in Section 3.1. During this conformance test we improve our test suite by adding generated counterexamples. These counterexamples reveal that our hypothesis is not yet conforming to the SUT. Based on the improved test suite we learn a new hypothesis. We continue this process until we find a hypothesis that fulfills our conformance relation, i.e. we cannot find counterexamples.

We make two adaptions to the refinement approach of Walkinshaw et al. [53], namely restricting the type of added tests and increasing the number of simultaneously added tests. The first adaption we make is that we only add failing timed traces to the test suite. Since the evolution of the population is based on the fitness of each individual, adding passing timed traces might interfere with the development of new functionality, because the fitness increases without the implementation of new functionalities of the SUT. We want to achieve that it pays off to implement the failing timed traces and therefore we keep the number of traces which contain already implemented knowledge low. Additionally, we support this objective by adding more than one failing timed trace. In genetic programming, where we rank individuals based on their fitness value, adding a single failing timed trace might have too little influence, especially, if the set of passing traces is large. We may force the population to improve by significantly rewarding the implementation of the new timed traces. In this context, we define n_{fail} as the maximum number of failing timed traces simultaneously added in one iteration.

Algorithm 1 presents the basic procedure of our active learning approach. The procedure requires an initial hypothesis $\mathcal{H}$, the interface to the SUT $\mathcal{I}$, a set of timed traces $\mathcal{TT}$, the probability p_{input} and the parameter $\mathit{reduceInput}$. $\mathcal{TT}$ may contain already generated timed traces, which are derived from the SUT, or may be empty. We require p_{input} for the generation of timed traces, where p_{input} defines the probability of performing a special type of input that may help to reveal yet undiscovered behavior of the SUT. Since we assume that the hypothesis improves in every iteration, the demand to reveal new behavior may decrease. Therefore, in Line 8 we decrease p_{input} in every iteration by the parameter $\mathit{reduceInput}$. We explain p_{input} in more detail in Section 3.5.

We split the discussion of the algorithm in two different parts: the learning of the hypothesis and the termination of the algorithm.

Hypothesis learning. The learning of the hypothesis is done over several iterations. In each iteration we add n_{fail} timed traces, until the overall maximum number of timed traces n_{test} is reached. We check if the maximum number of traces has been reached in Line 4. Each trace in the set of added timed traces reveals that the current hypothesis behaves incorrectly, i.e. the timed trace fails on the current hypothesis. Our target is to improve the hypothesis with the detected failing timed traces, so that these traces may not fail in the new hypothesis. In Line 5 we get these traces from the function GETCOUNTEREXAMPLE($\mathcal{H}, \mathcal{I}, p_{\text{input}}$), which takes the currently best hypothesis $\mathcal{H}$ of the SUT, the interface to the SUT $\mathcal{I}$ and the probability value p_{input}. The function returns a set of timed traces $\mathcal{TT}'$, containing only traces that reveal that $\mathcal{H}$ behaves differently than the SUT. Section 3.4 discusses the selection criteria of $\mathcal{TT}'$ in more detail. In Line 7 the newly learned timed traces of $\mathcal{TT}'$ are then added to the already existing set of timed traces $\mathcal{TT}$. Based on the new extended set of timed traces $\mathcal{TT}$ we genetically program a new hypothesis $\mathcal{H}$ in Line 11. The function $run(\mathcal{TT}, g_{\text{max}_{\text{active}}})$ uses GP to learn a new TA with at most $g_{\text{max}_{\text{active}}}$ generations based on the approach proposed by Tappler et al. [47] with the difference that $\mathcal{TT}$ also contains actively selected timed traces. One advantage of this active learning approach based on GP is that we can reuse the population that we learned in the previous GP execution, instead of creating a whole new population in every genetic programming run.

Termination. The procedure terminates when an appropriate representation of the SUT is found. Based on our conformance relation introduced in Section 3.1 we assume that $\mathcal{H}$ models the behavior of the SUT if we cannot find any counterexample and none of the previously created traces in $\mathcal{TT}$ fails. We determine the failing traces with the function $failingTT(\mathcal{H}, \mathcal{TT})$. This auxiliary function tests all traces in $\mathcal{TT}$ and returns a subset of $\mathcal{TT}$, where each trace in this subset fails on $\mathcal{H}$. We test if this termination criterion is satisfied in Lines 6 and 10. Nonetheless, the termination criterion in Line 3 is also satisfied if the maximum number of generations g_{max} is reached. Let $generations()$ be a function that returns the number of generations that have been required to learn a TA in the last GP run. The variable gen stores the sum over all required generations in all genetic programming executions. In each iteration we increase in Line 12 gen by the number of required generations until gen is larger than g_{max}. Regardless of whether we find a hypothesis that conforms to the SUT or we run out of generations, the procedure returns the best TA learned so far.

Algorithm 1 Basic Procedure

```
Input: Initial TA $\mathcal{H}$, SUT $\mathcal{I}$, timed traces $\mathcal{TT}$, float $p_{\text{input}}$, float $reduceInput$
Output: Inferred TA $\mathcal{H}$
1: function ACTIVEGENETICPROGRAMMING
2:     $gen \leftarrow 0$
3:     while $gen \leq g_{\text{max}}$ do
4:         if $|\mathcal{TT}| < n_{\text{test}}$ then
5:             $\mathcal{TT}' \leftarrow$ GETCOUNTEREXAMPLES($\mathcal{H}, \mathcal{I}, p_{\text{input}}$)
6:             if $|\mathcal{TT}'| = 0 \wedge |failingTT(\mathcal{H}, \mathcal{TT})| = 0$ then return $\mathcal{H}$
7:             $\mathcal{TT} \leftarrow \mathcal{TT} \cup \mathcal{TT}'$
8:             $p_{\text{input}} \leftarrow p_{\text{input}} - reduceInput$
9:         else
10:            if $|failingTT(\mathcal{H}, \mathcal{TT})| = 0$ then return $\mathcal{H}$
11:        $\mathcal{H} \leftarrow run(\mathcal{TT}, g_{\text{max}_{\text{active}}})$
12:        $gen \leftarrow gen + generations()$
13:    return $\mathcal{H}$
```

3.3 Initial Hypothesis

Our active learning approach requires an initial TA. This initial TA serves as our first hypothesis of the SUT. Since we improve our hypothesis in later iterations, we can start the active learning with a less advanced model. However, there exist different strategies to select the initial hypothesis. We have a closer look at three different approaches: (1) the most minimal setup; (2) an already existing specification; (3) a genetically programmed TA using the passive learning approach.

The most minimal setup for the initial hypothesis is a TA with only one location and no edges. Based on our assumption that a TA is input-enabled, we can generate timed traces by executing inputs. This approach may perform badly in the first iterations, as the initial hypothesis does not generate outputs. The reason for the bad performance is that due to the assumed input-enabledness we can only reveal differences based on observed outputs. Thus, to reveal that the initial hypothesis is false we have to produce sequences of inputs that trigger an output of the SUT. However, the advantage of this approach is that this initial TA is applicable as hypothesis for every SUT. Due to our approach for the selection of timed traces, which we introduce in Section 3.4, new knowledge about the systems is incrementally added. Therefore, we assume that with the minimal setup, the accuracy of the hypothesis also steadily increases.

Another possibility is to use a more sophisticated initial hypothesis. According to many in the field [53], developers might not have time to maintain a specification after every system update due to tight project schedules. However, we can use an outdated specification as an initial hypothesis and iteratively improve the specification through our active learning approach. The advantage is that we do not have to learn the entire model from scratch. However, this approach is only applicable if during the development process a specification in form of a TA has been created.

The last approach is more specific due to the used framework of our active learning algorithm. For this approach, we genetically program the first hypothesis akin to the passive approach proposed by Tappler et al. [47]. Depending on the quality of the randomly generated set of timed trace, a more or less appropriate hypothesis is learned. Like with an outdated specification, we may also iteratively improve the passively learned TA. Another advantage is that instead of using a single TA as initial hypothesis we can reuse the whole population of the genetic programming process. The reuse of the entire population of the passive approach significantly increases the pool of possible solutions. In addition, we may also add the passively generated timed traces to the set of timed traces of the active learning approach.

In general, these concepts for the initial hypothesis are not mutually exclusive. We can pick single properties of the approaches and create a new combined approach. For our case studies, which we discuss in Chapter 5, we rely on an initial hypothesis that is genetically programmed as explained in the last approach. However, we keep the initial set of timed traces small so that the hypothesis steadily grows. This strategy should avoid a complicated and computational expensive inference of the first hypothesis. Since the inferred TA may be not highly developed, we can improve the TA incrementally in each iteration as explained in the first approach for that we initialize $\mathcal{TT}$ with a set of randomly generated timed traces. We introduce the setup for our initial hypothesis in more details in Chapter 5.

3.4 Timed Trace Selection for Learning

The objective of the timed trace selection is to choose traces that improve the learning process of the current hypothesis of the SUT. In Section 3.1 we explained that our target is to enhance the completeness of the test suite. Therefore, we retrieve counterexamples to the conformance between the SUT and the hypothesis. Furthermore, we make additional adaptions to the found counterexamples which should optimize the learning process. This section introduces the criteria for choosing timed traces and then discusses further optimizations for the chosen traces.

Algorithm 2 presents our approach for the selection of timed traces. This algorithm uses the hypothesis $\mathcal{H}$ of the SUT to select timed traces. The algorithm returns a set of timed traces $\mathcal{TT}'$, where each trace in $\mathcal{TT}'$ is a counterexample to the correctness of the hypothesis $\mathcal{H}$. The algorithm collects up to n_{fail} timed traces by performing n_{attempt} attempts to find a counterexample via testing. In every attempt we generate a timed trace by randomly walking through $\mathcal{H}$ and, afterwards, we check if the same timed trace is observable in the SUT $\mathcal{I}$. To generate traces, the algorithm requires an hypothesis $\mathcal{H}$, whose behavior is compared to the SUT $\mathcal{I}$, and the probability p_{input}, which is used during the trace generation. In addition, the function requires three global parameters: n_{fail}, n_{attempts} and p_{stop}.

Algorithm 2 Selection of Timed Traces

```
Input: TA $\mathcal{H}$, SUT $\mathcal{I}$ ,$p_{\text{input}}$
Output: Set of Timed Traces $\mathcal{TT}'$
1:  function GETCOUNTEREXAMPLES
2:      $\mathcal{TT}' \leftarrow \{\}$
3:      for $i \leftarrow 0$ to $n_{\text{fail}}$ do
4:          for $j \leftarrow 0$ to $n_{\text{attempts}}$ do
5:              $tt_h \leftarrow$ RANDOMWALK$(\mathcal{H}, p_{\text{input}})$
6:              $tt_s \leftarrow execute(\mathcal{I}, tt_{\text{h}})$
7:              $length \leftarrow compareTraces(tt_h, tt_s)$
8:              if $length \neq |tt_s|$ then
9:                  if $containsOutput(tt_s)$ then
10:                     $tt_{\text{sub}} \leftarrow sub(tt_s, length)$
11:                     $tt_{\text{sub}} \leftarrow execute(\mathcal{I}, tt_{\text{sub}})$
12:                     if $ContainsOutput(tt_{\text{sub}})$ then
13:                         $tt_{\text{s}} \leftarrow tt_{\text{sub}}$
14:                     $\mathcal{TT}' \leftarrow \mathcal{TT}' \cup \{tt_{\text{s}}\}$
15:                     break
16:                 else
17:                     $tt_{\text{e}} \leftarrow extendTimedTrace(\mathcal{I}, tt_{\text{s}}, p_{\text{stop}})$
18:                     if $containsOutput(tt_e)$ then
19:                         $\mathcal{TT}' \leftarrow \mathcal{TT}' \cup \{tt_{\text{e}}\}$
20:                         break
        return $\mathcal{TT}'$
```

n_{fail} We limit the number of selected timed traces by n_{fail}, i.e. the size of $\mathcal{TT}'$ is at most n_{fail}. We want to find a sound set of traces and, with respect to our conformance relation, the set of traces should be as complete as possible. In addition, the genetic programming of a TA should be efficient. Keeping the set of traces small achieves efficiency since all traces must be executed in every generation on each individual of the population. To avoid that the size of $\mathcal{TT}'$ explodes in one iteration, we set the limit of added traces to n_{fail}.

n_{attempts} This constant defines how many attempts we have to find a counterexample. Searching the whole state space of $[\![\mathcal{H}]\!]$ is impossible since the state space is infinite due to the considered time values. Therefore, our approach is based on heuristics, where we assume that we get an accurate representation of the behavior of a TA with n_{attempts} timed traces.

p_{stop} We use the probability p_{stop} to stop the extension of timed traces and thus p_{stop} limits the length of the timed trace.

In Line 2 the algorithm starts with the initialization of $\mathcal{TT}'$ to an empty set. After the initialization we start to search for timed traces in Line 3. Since n_{fail} defines the number of the maximal added traces, the number of iterations in which we search for an individual counterexample is limited by n_{fail}. The iteration starts in Line 4. In every iteration, we have n_{attempts} to find a trace that reveals a counterexample. Each attempt first generates a timed trace. The generation of timed traces is done in Line 5 using the function RANDOMWALK$(\mathcal{H}, p_{\text{input}})$. This function returns a timed trace tt_h which is generated by randomly walking through $\mathcal{H}$. Beside $\mathcal{H}$, this function also requires p_{input} which defines the probability of performing an unknown input during the random walk. We explain the procedure of this random walk through a TA in Section 3.5 in more detail. After the generation of tt_h, we check if tt_h is also observable in the SUT. Therefore, we simulate tt_h on the SUT in Line 6. The simulation of tt_h on the SUT $\mathcal{I}$ is done via the function $execute(\mathcal{I}, tt_h)$. Since we assume that the SUT is deterministic, the function returns one corresponding timed trace tt_s. After the generation of both timed traces, we compare them in

Line 7 for which we use the function COMPARETRACES$(tt, tt^{'})$. The aim of this function is to compare two timed traces. Let $tt, tt^{'} \in (\mathbb{R}_{\geq 0} \times \Sigma)$ be the timed traces, where $tt = t_0 \cdot a_0, \cdots, t_{\mathrm{m}-1} \cdot a_{\mathrm{m}-1}$ and $tt^{'} = t^{'}_0 \cdot a^{'}_0, \cdots, t^{'}_{\mathrm{n}-1} \cdot a^{'}_{\mathrm{n}-1}$. We compare each element of the timed traces until the end of the shorter trace is reached. One element of the timed trace is a pair $\langle t_i, a_i \rangle$ of a time stamp $t_i \in \mathbb{R}_{\geq 0}$ and an action $a_i \in \Sigma$ with $0 < i < \min(m, n)$. The function returns the length of $tt^{'}$ if the traces are equal, otherwise the index i of the first found element where t_i or a_i is different is returned. We can formalize the function as follows:

$$\textsc{compareTraces}(tt, tt') = \begin{cases} \min\{i \mid \exists\, t_{\mathrm{i}}a_{\mathrm{i}} \in tt, t'_{\mathrm{i}}a'_{\mathrm{i}} \in tt' \;:\; t_{\mathrm{i}} \neq t'_{\mathrm{i}} \vee a_{\mathrm{i}} \neq a'_{\mathrm{i}}\} & \text{if there is such an } i \\ |tt'| & \text{if there is no such } i \end{cases}$$

In Line 7 we compare the traces tt_h and tt_s with the function COMPARETRACES(tt_h, tt_s). If the traces tt_h and tt_s are equal, COMPARETRACES(tt_h, tt_s) returns the size of tt_s. We then compare the return value of the function $length$ with the actual size of tt_s in Line 8. By the comparison in Line 8 we may skip failing timed traces, where $|tt_h|$ is larger than $|tt_s|$ due to the definition of COMPARETRACES(tt_h, tt_s). However, long timed traces may cause a bad performance, because we assume that the learning of longer traces is more difficult. Therefore, we skip this kind of counterexamples and try to find a shorter counterexample in another attempt.

For the case that we find a counterexample, we perform some further operations on the counterexample in the Lines 9 and 20 which aim to optimize the learning of a new automaton. To perform this optimization, we define three new external functions: $containsOutput(tt)$, $sub(tt, Integer)$ and $extendTimedTrace(\mathcal{I}, tt, p_{\mathrm{stop}})$.

$containsOutput(tt)$ This function returns *true* if the provided timed trace tt contains an output action, otherwise *false* is returned.

$sub(tt, len)$ This function returns a prefix of the timed trace tt with the size of len.

$extendTimedTrace(\mathcal{I}, tt, p_{\mathrm{stop}})$ This function extends tt. The extension is done by executing the timed trace tt on the SUT $\mathcal{I}$ and afterwards performing further randomly chosen inputs until an output is observed or we stop with the probability p_{stop}. The generation of the extension is akin to the passive trace generation. The function returns the extended timed trace.

First, in Line 9 we check with the function $containsOutput(tt_s)$ whether tt_s contains at least one output action. If an output action in tt_s is present, we reduce tt_s to the relevant part of the sequence in Line 10. We denote the relevant part of a timed trace $tt_s = \langle t_0 \cdot a_0 \cdots t_n \cdot a_n \rangle$ as the shortest prefix $tt_{\mathrm{sub}} = \langle t_0 \cdot a_0 \cdots t_{\mathrm{length}} \cdot a_{\mathrm{length}} \rangle$ of the trace, where the size of the prefix is the smallest length that reveals a difference between the traces. The function $sub(tt_s, length)$ returns the relevant part tt_{sub} of tt_s.

If we detect a difference in the compared traces due to a different delay or a different observed action we discard all following actions after the detected difference. As a result, we may skip actions that happen immediately, i.e. without a delay, after the found difference. If there are urgent outputs and we skip them, they might not be represented in our newly learned TA and we have to perform another conformance check on the new TA to detect the missing implementation of the urgent outputs. As a result, we have to add another timed trace, that contains these urgent outputs. Afterwards, we have to learn a new TA based on this timed trace including the urgent outputs. To mitigate these unnecessary iterations, we add these possible urgent outputs from the beginning. We consider these urgent outputs by executing the truncated trace tt_{sub} again on the SUT in Line 11.

After the repeated execution, we check once again in Line 12 whether the truncated timed trace tt_{sub} contains an output. If tt_{sub} it does, we add tt_{sub} in Line 14 to the set of timed traces $\mathcal{TT}'$, otherwise

tt_s without truncation is added. For the case that tt_s has no output action at the initial execution on the SUT, we extend tt_s with the function $\mathit{extendTimedTrace}(\mathcal{I}, tt_s, p_{\text{stop}})$ in Line 17. After the extension we check whether the extended timed trace tt_e now contains an output action and if this is the case we add tt_e to $\mathcal{TT}'$ in Line 19.

Considering this selection algorithm it is possible to find a counterexample that is not added to $\mathcal{TT}'$ because it has no output action in the timed trace. According to the assumption that a TA is input-enabled, traces without an output action may add less knowledge about the SUT than traces with at least one output, as traces that trigger no output action are valid in various TA. Therefore, we skip traces without any output action and start a new attempt. For the case that we find an appropriate counterexample with an output we stop all further attempts of this iteration.

If we find a counterexample, we reset the number of attempts and start searching for a new timed trace. In every iteration where we cannot find a counterexample, i.e. all generated timed traces from the hypothesis show behavior equal to SUT, we do not add any timed trace and reset n_{attempts} for the next iteration. The algorithm terminates after n_{fail} iterations, with n_{attempts} attempts to find an appropriate counterexample in each iteration.

After the termination of searching timed traces the algorithm returns the found and processed timed traces $\mathcal{TT}'$. If we do not find any counterexamples, we return an empty set, otherwise we return up to n_{fail} counterexamples.

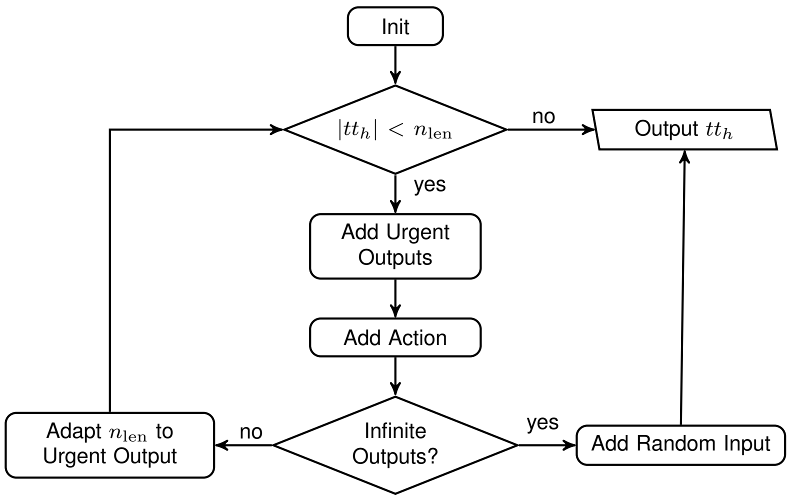
3.5 Timed Trace Generation

In the timed trace generation we want to create timed traces that represent the behavior of the SUT. However, due to the consideration of real-valued time variables the number of possibilities to generate a trace in a Timed Transition System (TTS) is infinite. Hence, covering the whole state space is impossible. Since a TA has an infinite number of timed traces, our target is to generate timed traces that provide an appropriate representation of the behavior of the SUT. According to Aichernig and Tappler [5] an appropriate coverage, though, may also be achieved by a random walk through a hypothesis model. In a walk through a TA we follow edges and may change locations. This generates a run through the TA, which corresponds to a timed trace. Figure 3.2 represents our approach to perform a random walk through the TA, where Figure 3.2a shows the general procedure and Figure 3.2b explains the selection of actions in more detail.

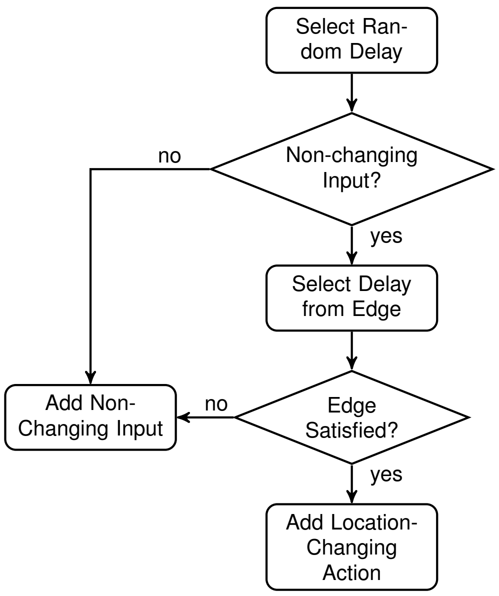
In this section we will have a closer look at the different steps of the generation of timed traces. We divide the generation process in for major steps: *Initialization*, *Checking Enabled Outputs*, *Adding Actions* and *Termination*. We first discuss the initialization of the required components of the random walk. Secondly, we introduce our approach to check enabled outputs. Afterwards, we explain how we find a possible action which can be added to the timed trace. Finally, we discuss the termination criteria for our timed trace generation approach.

3.5.1 Initialization

As depicted in Figure 3.2a before we start our random walk we initialize the used variables. Let l be a variable that indicates the current location during the random walk. A random walk through a TA $\mathcal{H} = \langle L, l_0, E \rangle$ over $(\mathcal{C}, \Sigma)$ always starts at the initial location l_0. At the start of a random walk l is always l_0. Besides the current location we always have to keep track about the components that handle time. To store the values of each clock in $\mathcal{C}$ for $\mathcal{H}$, we create a clock valuation ν which is initialized with $\mathbf{0}_{\mathcal{C}}$. We denote the variable $t \in \mathbb{R}_{\geq 0}$ as our time variable, which stores the overall elapsed time during the random walk. This time variable t is initialized to zero. After the initialization we start our random walk and keep walking through the TA until our timed trace is long enough. We discuss the criteria that define whether a timed trace is long enough in Section 3.5.4.



(a) Random Walk through a TA



(b) Add Action

Figure 3.2: Generation of timed traces tt_h of the hypothesis

3.5.2 Checking Enabled Outputs

During a random walk we have to consider that there may be edges whose selection is obligatory. Since we assume that our TA has urgent outputs, we have to follow all edges that are labeled with an enabled output immediately. For the case the current location has an outgoing edge with an enabled output $o \in \Sigma_O$, we add the output $o \in \Sigma_O$ with the current time t to our timed trace. Since we follow an edge, we have to perform all necessary updates, that is, changing the location and performing possible clock resets. After we performed these updates, we have to check again if we have another enabled output in the new location. We repeat these checks and updates on enabled outputs until we find no further enabled outputs.

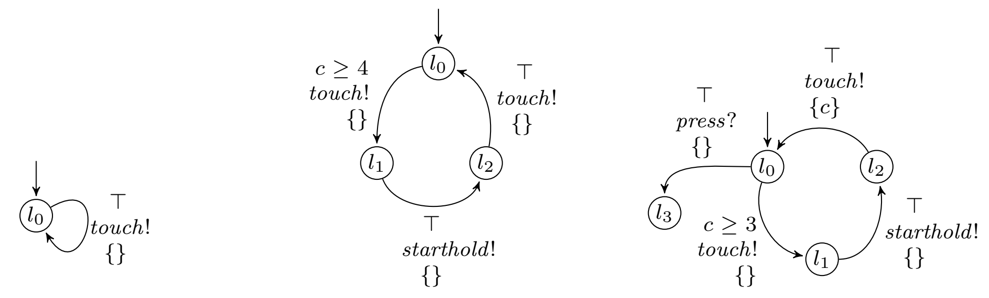
During this check for enabled outputs we also have to consider that the generation process of TAs via genetic programming may create non-deterministic TAs. One case of non-determinism that affects our output check occurs if the TA has a location where we find several outputs that are enabled. In this case we follow the first found edge that triggers an output and ignore the other edges with enabled outputs. According to our timed trace selection process described in Section 3.4 this timed trace is detected as a failing timed trace since the simulation of the timed trace reveals a non-deterministic behavior. Therefore, we consider the random selection of one enabled output as sufficient to reveal the incorrect behavior of the TA since we assume that the prefix of the timed trace which leads to non-deterministic behavior is more important than the chosen output action.

Besides the problem that outputs may be non-deterministic, infinite sequences of outputs can also cause a problem. We denote a sequence of outputs as infinite, if the occurrence of enabled outputs does not terminate. We assume that we observe such a behavior, if we always return to a previously entered location where an already observed output is still enabled. In this case the outputs repeat in an infinite loop. Such behavior is usually called Zeno behavior [10]. We assume that the SUT does not implement such behavior, as it would basically require the SUT do stop time while producing infinitely many outputs. Since we assume that the SUT does not implement such infinite output sequences, we add traces that reveal that the hypothesis produces prohibited infinite output sequences.

Example 7 (Infinite Output Sequence)**.** Figure 3.3 shows three automata which may be learned based on the TA introduced in Example 1. Each of these three automata produces a sequence of outputs, which may not repeat more than once. However, two (Figure 3.3a and Figure 3.3b) out of these three automata produce infinite sequences of outputs. In Figure 3.3a the output sequence $tt = 0.0 \cdot touch! \cdot 0.0 \cdot touch! \cdots$ is infinite as the one edge labeled with an output is always enabled. To observe the infinite output sequence in Figure 3.3b, time has to elapse by at least four time units, but after this amount of time we observe infinite enabled outputs. For example, one possible trace that leads to an infinite sequence is: $tt = 3.0 \cdot press? \cdot 3.4 \cdot release? \cdot 4.0 \cdot touch! \cdot 4.0 \cdot starthold! \cdot 4.0 \cdot touch! \cdot 4.0 \cdot touch! \cdots$. The prohibited behavior in Figure 3.3a and Figure 3.3b should be detected. However, Figure 3.3c also triggers a sequence of outputs if the time elapses at least three time units in location l_0. According to Asarin et al. [10] this sequence, though, is not infinite since we reset the clock along the execution of outputs. To repeat this sequence, time has to elapse. Therefore, we may observe a sequence of same outputs, e.g. $tt = 2.0 \cdot release? \cdot 3.0 \cdot touch! \cdot 3.0 \cdot starthold! \cdot 3.0 \cdot touch! \cdot 6.0 \cdot touch! \cdot 6.0 \cdot starthold! \cdots$, but we do not define this sequence as infinite since we can perform inputs in the location l_0. Therefore, Figure 3.3c should not be detected as a TA that produces a prohibited infinite sequence of outputs.

Algorithm 3 presents our approach to detect an infinite sequence of outputs in a TA. Since we perform this check during a random walk through the system, we have to consider the current status of the TA $\mathcal{H} = \langle L, l_0, E \rangle$ over $(\mathcal{C}, \Sigma)$. Let $l \in L$ be a location in the TA, which we reach during our random walk. The location l is the start of our infinite output check. Furthermore, we have to consider the time components like the overall elapsed timed $t \in \mathbb{R}_{\geq 0}$ and the valuation of each clock ν.

The algorithm starts in Line 2 and 3 with the initialization of *infiniteOutputs* and L_{urgent}. The Boolean variable *infiniteOutputs* indicates whether an infinite output sequence is found or not. L_{urgent} is an auxiliary multiset that saves all reached locations along a sequence of urgent outputs. In Line 4



(a) Infinite outputs over one location **(b)** Infinite outputs over multiple locations **(c)** No output infinity

Figure 3.3: Examples of two infinite outputs and one with a clock reset that avoids infinite outputs

we retrieve all edges labeled with output actions E_O, from the set of all outgoing edges of the current location l. We iterate E_O and check in each iteration whether any clock variable of the clock valuation ν satisfies the guard g of the currently considered edge e_O. This check is performed in Line 6. If any of the guards of edges in E_O is satisfied, we found an enabled output o.

In the case of a found output we check in Line 7 whether the sequence of outputs is infinite. If l already exists twice in the multiset L_{urgent}, we assume that we found an infinite output sequence. Since L_{urgent} is a multiset we can check this condition with the multiplicity of the current location l. This condition for infinite outputs is based on the assumption that we consider a return to a previous location with an urgent output as a valid output sequence. If the currently explored loop contains resets and guards on the same clock, then we will not explore the location more than twice, therefore the check is sound. However, if we have already visited the current location twice, there are no relevant clock resets or guards along the way to the same location. Therefore, we follow an infinite path if we visit the location more than twice. In this case we stop adding urgent outputs and set in Line 8 the Boolean variable $infiniteOutputs$ to $\top$. If we have not found an infinite sequence, we add in Line 10 the urgent output with the corresponding time t to tt_h.

While walking through this sequence of outputs, we always have to keep the variables updated, which we do from Line 11 to Line 14. These updates include (1) resetting the clocks in ν if the edges are labeled with clock resets, (2) adding the location to the set of already observed locations L_{urgent}, (3) changing the current location l and (4) retrieving the new set of output edges E_O. Updating E_O also implements the handling of non-determinism, because we may skip further urgent outputs of this location. Since several urgent outputs in one location indicate non-determinism, we assume that other urgent outputs in E_O can be skipped. If we have not found an infinite sequence we continue adding urgent outputs until no further urgent output is found.

Example 8 (Infinite vs. Finite Sequences)**.** Figure 3.3c shows a TA that may return during a sequence of outputs, e.g. $tt = 2.0 \cdot release? \cdot 3.0 \cdot touch! \cdot 3.0 \cdot starthold! \cdot 3.0 \cdot touch! \cdots$, to a previous visited location with an output. In this case, the visited edges lead to a loop. However, due to the clock resets and the guards on these edges, the sequence of enabled outputs terminates since the guard $c \geq 3$ is not satisfied anymore. Let l_0 be the location where we start our check for urgent outputs. We assume that all our clocks have the value zero at the beginning of the random walk. If the time elapses at least three time units and we do not perform the input $press?$ in the meantime, we have to follow the edge which triggers the output $touch!$. After the performed updates we now repeat the check for urgent outputs in location l_1. The TA now triggers further enabled outputs. First $starthold!$ in location l_1 and then $touch!$ in l_2. However, during the updates which are performed when we follow the edge from l_2 to l_0, we reset the clock c. If we repeat this loop we now stop at location l_0 as the guard $c \geq 3$ is not satisfied anymore. Therefore, no location is added twice to the set of locations L_{urgent} and no infinite output sequence is detected. We may however observe the same sequence of outputs again and pass the same locations

Algorithm 3 Detection of Infinite Outputs

```
Input: TA $\mathcal{H} = \langle L, l_0, E \rangle$ over $(\mathcal{C}, \Sigma)$, Current location $l$, Overall time $t$, Clock valuation $\nu$
 1: function InfiniteOutputDetection
 2:     $infiniteOutputs \leftarrow \bot$
 3:     $L_{\text{urgent}} \leftarrow \{\}$
 4:     $E_O \leftarrow \{e | \exists l', g, r : e = l \xrightarrow{g,o,r} l', o \in \Sigma_O\}$
 5:     for all $(l, g, o, r, l') \in E_O$ do
 6:         if $\nu \models g$ then
 7:             if $L_{\text{urgent}}(l) = 2$ then
 8:                 $infiniteOutputs \leftarrow \top$
 9:                 break
10:             $tt_h \leftarrow tt_h \cdot t \cdot o$
11:             $\nu \leftarrow \nu[r]$
12:             $L_{\text{urgent}} \leftarrow L_{\text{urgent}} \uplus \{l\}$
13:             $l \leftarrow l'$
14:             $E_O \leftarrow \{e | \exists l', g, r : e = l \xrightarrow{g,o,r} l', o \in \Sigma_O\}$
15:     return $infiniteOutputs$
```

several times. For example, the timed trace $tt = 2.0 \cdot release? \cdot 3.0 \cdot touch! \cdot 3.0 \cdot starthold! \cdot 3.0 \cdot touch! \cdot 6.0 \cdot touch! \cdot 6.0 \cdot starthold! \cdots$ shows a walk where we perform the same loop of outputs twice. However, the sequence is not infinite since it is possible to perform inputs in location l_0.

3.5.3 Adding Actions

Unless we found an enabled output, we need a strategy to decide which kind of edges we want to follow. Figure 3.2b depicts the control flow of selecting edges and adding the corresponding actions. Basically, we distinguish between two different types of edges. First, since our TA is input-enabled we may perform an implicitly enabled input action and stay in the same location. Second, we can follow an edge with particular labels, guards, and clock resets by performing the corresponding action of the edge. However, as we perform a random walk in a TA we also have to consider the elapse of time after we performed an action. In this section, we introduce our approach to generate delays and then we explain the methodology of performing specific actions.

The first step in our edge selection process is to delay the system. Therefore, we have to generate a delay. For this, we introduce two different *random delay generators*. The first generator is similar to the idea proposed by Tappler et al. [47]. They use the delay generator to generate timed traces for the passive inference of TAs and select delays based on a set of constants of the SUT. We denote this set of constants as *hints*. Their random delay generator creates a delay according to two different strategies, where the applied strategy is selected uniformly at random whenever the generator is used. The first strategy is to select uniformly at random a delay $d \in \mathbb{R}_{\geq 0}$ from the provided set of hints of the SUT. In addition, they may modify the selected constant with an offset. This offset is calculated based on the smallest constant in the collection of constants of the SUT. Their second strategy chooses the delay $d \in \mathbb{R}_{\geq 0}$ from the range $[0, c_{\text{hint}_{\max}} \times 2]$ uniformly at random, where $c_{\text{hint}_{\max}}$ is the largest value of the hints.

However, in our active approach, knowledge about the relevant constants of the SUT is not required since we assume that our approach learns these constants. This is done by using the constants from the guards of the learned TA. This constant learning approach is explained later in this section in more detail. In general, we propose a random delay generator which is akin to the second strategy of the generator from Tappler et al. [47], where our generator chooses a delay $d \in \mathbb{R}$ between $(0, c_{\max}]$ uniformly at random, where $c_{\max}$ is defined by the user. In Chapter 5 we explain which random delay generator we use and how the results of both are different.

Algorithm 4 Selection of Action

```
Input: TA $\mathcal{H} = \langle L, l_0, E\rangle$ over $(\mathcal{C}, \Sigma)$, Current location $l$, Overall time $t$, Clock valuation $\nu$, Delay $d$
1: procedure PERFORMACTION
2:     $edge \leftarrow Nil$
3:     if $\neg probChoice(p_{\text{input}})$ then
4:         $edge \leftarrow$ PERFORMCHANGINGACTION$(\mathcal{H}, l, t, \nu, d)$
5:     if $edge = Nil$ then
6:         PERFORMNONCHANGINGACTION$(\mathcal{H}, l, t, \nu, d)$
```

After selecting a delay, we have to decide whether we want to find an unexplored functionality of the TA or check an already existing functionality. We denote a *non-location-changing action* as the execution of an input that does not change the location l, does not reset any clock of the valuation ν and is not restricted by a clock guard. We perform such actions to find an unexplored behavior of the TA. If any of these conditions is violated, we describe the action as *location-changing*. Algorithm 4 describes the procedure which decides if we perform a location-changing or a non-location-changing action. p_{input} indicates the probability for performing a non-location-changing action. In Line 3 the procedure starts with the check of the probability p_{input}. With the probability $1 - p_{\text{input}}$ we perform a location-changing action in Line 4. Algorithm 5 presents the approach to select such a location-changing action. With the probability p_{input} or if we did not find a location-changing action we perform a non-location-changing action in Line 6, which is described in Algorithm 6.

In Algorithm 5 we try to add a location-changing action. Therefore, we start again with the selection of a delay $d \in \mathbb{R}_{\geq 0}$ with the probability p_{trans}. In this case we want to apply a more sophisticated generation approach for a delay, since we want to trigger an output or satisfy a guard of an edge labeled with an input. In Line 4 we collect all constants $k \in \mathbb{Z}$ from the guards $g = \bigwedge c \oplus k$ of all outgoing edges of the current location l in a multiset $\mathcal{K}$. Let *selectRandom*$(\mathcal{X})$ be an auxiliary function that returns an entry in the collection $\mathcal{X}$ chosen uniformly at random or Nil if the collection is empty. If $\mathcal{K}$ is not empty, we use the function *selectRandom*$(\mathcal{K})$ in Line 6 to randomly choose a new delay d from a multiset of time constants $\mathcal{K}$. Using this kind of delay chooser creates the opportunity to check the already learned constants of our hypothesis. By adding timed traces that reveal that the guard is wrong, we learn the constant of the SUT without providing it in advance. We will discuss this topic again later in Chapter 5.

Independent from whether we generate a delay d with the random delay generator or select a constant from the outgoing edges as delay d, we delay all clocks in Line 7 by d. After delaying all clocks, we check whether we triggered an output or not. Like in Section 3.5.2 we check if any output action of outgoing edges is enabled. Therefore, we collect in Line 8 all outgoing edges of the current location l which are labeled with an output $o \in \Sigma_O$. Afterwards we iterate over E_O and check if a guard g on an outgoing edge $l \xrightarrow{g,o,r} l'$ is satisfied. If the output is enabled, we memorize the edge and stop searching for further edges. Similar to the handling of non-deterministic outputs in the check for infinite output sequences, we ignore further enabled outputs since non-deterministic behavior is detected in any case by simulating this timed trace on the hypothesis. Furthermore, due to the output urgency we have to adjust the delay d to the minimum of time that is necessary to trigger the output. The computation of the minimum delay is done in Line 11 by the function $minDelay(g, \nu)$, where g is a clock guard and ν the clock valuation. If no edge with an output is found we continue searching edges labeled with an input action. This search for an edge with a changing input action is done in the Lines 15 and 20. Since more than one edge labeled with an input action can be satisfied, we initialize an auxiliary set $E_{I_{\text{sat}}}$, which stores all enabled edges labeled with inputs. To check which input edges are enabled, we first retrieve the set of all outgoing edges E_I of the current location that are labeled with an input $i \in \Sigma_I$. After the retrieval, we iterate them and, in Line 19, add all edges with satisfied guards to the set $E_{I_{\text{sat}}}$. After iterating all edges in E_I, we randomly select one edge of $E_{I_{\text{sat}}}$. In Line 20 we again use the function *selectRandom*$(E_{I_{\text{sat}}})$ to select uniformly at random an enabled input edge. However, if no edge is enabled the function returns Nil, i.e. we find no edge. In the case that we find an enabled action, which

Algorithm 5 Selection of Location-Changing Action

```
Input: TA $\mathcal{H} = \langle L, l_0, E \rangle$ over $(\mathcal{C}, \Sigma)$, Current location $l$, Overall time $t$, Clock valuation $\nu$, Delay $d$
 1: function PERFORMCHANGINGACTION
 2:     $edge \leftarrow Nil$
 3:     if $probChoice(p_{\text{trans}})$ then
 4:         $\mathcal{K} \leftarrow \{k \mid \exists\, l', a, r : l \xrightarrow{g,a,r} l', g = \bigwedge c \oplus k\}$
 5:         if $|\mathcal{K}| > 0$ then
 6:             $d \leftarrow selectRandom(\mathcal{K})$
 7:     $\nu \leftarrow \nu + d$
 8:     $E_O \leftarrow \{e \mid \exists\, l', g, r : e = l \xrightarrow{g,o,r} l', o \in \Sigma_O\}$
 9:     for all $(l, g, o, r, l') \in E_O$ do
10:         if $\nu \models g$ then
11:             $d \leftarrow minDelay(g, \nu)$
12:             $edge \leftarrow (l, g, o, r, l')$
13:             break
14:     if $edge = Nil$ then
15:         $E_I \leftarrow \{e \mid \exists\, l', g, r : e = l \xrightarrow{g,i,r} l', i \in \Sigma_I\}$
16:         $E_{\text{I}_{\text{sat}}} \leftarrow \{\}$
17:         for all $(l, g, i, r, l') \in E_{\text{I}}$ do
18:             if $\nu \models g$ then
19:                 $E_{I_{\text{sat}}} \leftarrow E_{I_{\text{sat}}} \cup \{(l, g, i, r, l')\}$
20:         $edge \leftarrow selectRandom(E_{I_{\text{sat}}})$
21:     if $edge \neq Nil$ then
22:         $\nu \leftarrow \nu[r]$
23:         $t \leftarrow t + d$
24:         $l \leftarrow l'$
25:         $tt_h \leftarrow tt_h \cdot t \cdot a$ with $edge = l \xrightarrow{g,a,r} l'$
    return $edge$
```

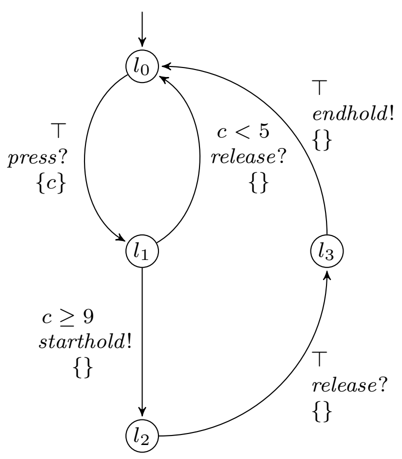


Figure 3.4: Possible hypothesis of TA of Example 1

can be either an input or an output action, we want to execute the edge labeled with this action. To execute the found edges, we have to update the variables of the system, i.e., resetting the clocks, increasing the overall time t by the chosen delay d and changing the current location l to the target location of the edge. These updates are performed from Line 22 – 24. Finally, we add the executed input or observed output with the corresponding time to the timed trace tt_h.

Example 9 (Adding Changing Action)**.** Figure 3.4 shows a learned hypothesis of the TA that we introduced in Example 1. We start our random walk at the initial location l_0 and want to perform a location-changing action after a random generated delay. First, we start with the delay selection. For this example we assume that $probChoice(p_{\text{trans}})$ is not satisfied. Therefore, we do not choose a constant of the guards, but select the random delay $d = 3.0$ with our random delay generator. Since we are at the beginning of our random walk, the overall time $t = 0$. As there is only one outgoing edge in the Location l_0 we select the input $press?$ and execute the edge $l_0 \xrightarrow{\top, press?, \{c\}} l_1$. For this, we reset the clock valuation of c, increase the overall time by 3.0 and change the current location to l_1. Finally, we update the timed trace to $tt = 3.0 \cdot press?$. In the Location l_1 there are two possible outgoing edges. Both of them have guards with constants. We now want to use these constants $\{5, 9\}$ as possible delays and choose $d = 9$ as our delay. After performing this delay the output edge $l_1 \xrightarrow{c \geq 9, starthold!, \{\}} l_2$ is enabled. Therefore, we have to add this output action with the corresponding minimal delay to the timed trace after we update the variables as previously explained. The updated timed trace is then $tt = 3.0 \cdot press? \cdot 12.0 \cdot starthold!$.

The procedure to find a location-changing action may not always find an enabled edge. To extend the timed trace without performing a changing action, we require a second approach to add other actions. In this approach we add actions that show a non-location-changing behavior. Non-location-changing actions are inputs that do not cause a location change or a clock reset, and are not restricted by any guards. Furthermore, we assume that performing non-location-changing actions may reveal that the learned behavior of the TA is incomplete since these inputs can trigger yet unobserved outputs. We consider a TA as incomplete if parts of the SUT are not modeled by the TA. To check the completeness of the inferred TA, we perform non-location-changing actions with the probability p_{input}.

The Algorithm 6 describes our approach to select this type of actions. We start the actual selection procedure of a non-location-changing action in Line 2. The procedure to find a non-location-changing action is done in a loop. We stop searching a non-changing action either in the case that we find a corresponding input action or with the probability p_{stop}. In Line 3 we delay the systems by the previously generated delay. This delay is always randomly selected by a random delay generator. Since performing a delay may cause an output, we have to check if any output is enabled. For this, we initialize in Line 4 the Boolean variable $outputSatisfied$, which is $\top$ if there exists an enabled output. To find an enabled output, we iterate all outgoing edges, which are labeled with an output action, E_O and check if at least one guard is satisfied. In the case that we find an enabled output, the chosen delay is not qualified to perform a non-changing action since we assume that outputs are urgent.

If none of the guards of the edges in E_O are satisfied, we have to perform the same check for the edges labeled with an input action. We start this check for enabled inputs in Line 11 with the assignment of all possible input actions Σ_I to an auxiliary set $\Sigma_{I_{\text{apply}}}$ which stores all inputs that do not cause a changing action at the end of the check. Since we can decide which input we want to perform, we remove all inputs that cause a location-changing action from the set $\Sigma_{I_{\text{apply}}}$. To remove the changing input actions, we iterate in Line 13 over the set E_I which contains all outgoing edges of the current location l that are labeled with an input action. In the case that the clock valuation ν satisfies a guard g of an edge that is labeled with an input action, we remove the corresponding input action from the set of applicable non-location-changing input actions $\Sigma_{I_{\text{apply}}}$. After removing all enabled edges with an input action, we check in Line 16 if there are any non-location-changing input actions remaining. We use the function $selectRandom(\Sigma_{I_{\text{apply}}})$ to select an input from the set of the remaining input actions $\Sigma_{I_{\text{apply}}}$ uniformly at random. After increasing the time t by the delay d, we add the current time and the chosen input to the timed trace tt_h. If we can add a non-location-changing input action to the timed trace, we stop searching for further actions. The search for a non-location-changing input action continues either in the case that we trigger an output or no inputs remain in the set $\Sigma_{I_{\text{apply}}}$. For these cases, we try to find another input by selecting a new delay d. For this, we have to revert the previously performed delay. In Line 21 we restore the clock valuation ν by subtracting the performed delay d. A new delay d is generated by the function $getRandomDelay()$, which simply returns a new value from the random delay generator. However, we might not find an appropriate delay to perform an non-location-changing action. Therefore, we terminate the search for an input that causes a non-changing action with the probability p_{stop}.

Algorithm 6 Selection of Non-Location-Changing Action

```
Input: TA $\mathcal{H} = \langle L, l_0, E\rangle$ over $(\mathcal{C}, \Sigma)$, Current location $l$, Overall time $t$, Clock valuation $\nu$, Delay $d$
1: procedure PERFORMNONCHANGINGACTION
2:     do
3:         $\nu \leftarrow \nu + d$
4:         $outputSatisfied \leftarrow \bot$
5:         $E_O \leftarrow \{e \mid \exists\, l', g, r : e = l \xrightarrow{g,o,r} l', o \in \Sigma_O\}$
6:         for all $(l, g, o, r, l') \in E_O$ do
7:             if $\nu \models g$ then
8:                 $outputSatisfied \leftarrow \top$
9:                 break
10:        if $\neg outputSatisfied$ then
11:            $\Sigma_{I_{\text{apply}}} \leftarrow \Sigma_I$
12:            $E_I \leftarrow \{e \mid \exists\, l', g, r : e = l \xrightarrow{g,i,r} l', i \in \Sigma_I\}$
13:            for all $(l, g, i, r, l') \in E_I$ do
14:                if $\nu \models g$ then
15:                    $\Sigma_{I_{\text{apply}}} \leftarrow \Sigma_{I_{\text{apply}}} \setminus \{i\}$
16:            if $|\Sigma_{I_{\text{apply}}}| > 0$ then
17:                $i \leftarrow selectRandom(\Sigma_{I_{\text{apply}}})$
18:                $t \leftarrow t + d$
19:                $tt_h \leftarrow tt_h \cdot t \cdot i$
20:                break
21:        $\nu \leftarrow \nu - d$
22:        $d \leftarrow getRandomDelay()$
23:    while $\neg probChoice(p_{\text{stop}})$
```

Example 10 (Adding Non-Location-Changing Action)**.** We want to perform another random walk in Figure 3.4. We start the random walk again at the initial Location l_0, but this time we want to perform a non-location-changing action. Since we start a new random walk we assume that the overall time $t = 0$ and the clock valuation $\nu = \mathbf{0}_C$. We assume that the randomly generated delay $d = 3.0$. After delaying the clock c by three time units, we check if any output is enabled. Since Location l_0 has no outgoing edges that are labeled with output action, no output is possible. In the next step we continue with the selection of possible inputs. The set of executable input actions of the TA is $\Sigma_I = press?, release?$. The Location l_0 has one outgoing edge $l_0 \xrightarrow{\top, press?, \{c\}} l_1$ which is labeled with an input action. Since this input is enabled we remove the input $press?$ from the set of possible non-changing inputs. After removing $press?$ from $\Sigma_{I_{\text{apply}}}$, the input $release?$ remains as the only possible input that we can execute to perform a non-changing action. Therefore, we increase the overall time t by 3.0 and add the updated time $t = 3.0$ and the input $release?$ to the timed trace tt_h. After the update the timed trace is $tt_h = 3.0 \cdot release?$.

3.5.4 Termination

The generation of a timed trace terminates when the timed trace tt_h has the size of n_{len}, i.e. $|tt_h| = n_{\text{len}}$. However, we may terminate earlier if we detect an infinite sequence of outputs. In this case, it is not necessary to continue the random walk since we expect that the sequence of outputs repeats continuously. To reveal that the currently investigated TA has an infinite output sequence, we add an additional input to tt_h after the termination of the random walk. This input is randomly chosen from the set of all input actions Σ_I and performed after a random delay $d \in \mathbb{R}_{\geq 0}$. According to our assumptions about a TA, which we explained in Section 2.3.3, this kind of input is not possible. In Section 3.4 we presented our timed trace selection algorithm that executes the timed trace generated by the hypothesis on the SUT. Executing this additionally added input supports the detection of wrong behavior of the hypothesis, as the hypothesis produces an infinite sequence of outputs. The SUT does not produce such a sequence, therefore the observed timed trace is different. After this input is added we return the timed trace even though it is shorter than n_{len}.

One problem that arises due to the assumption that outputs are urgent is that we may skip enabled outputs if the timed trace has reached the size of n_{len}. If a trace reflects a valid behavior and also the SUT would generate an enabled output, the trace generated by the hypothesis would be classified as a failing trace in Algorithm 2, because the output is missing. Therefore, we increase n_{len} if there is an enabled output and the size of tt_h is already equal to n_{len}. However, since long timed traces may be more difficult to learn, we extend n_{len} until it is at most $n_{\text{len}_{\max}}$. As a result, the random walk ends at the latest when the timed trace has grown to the size $n_{\text{len}_{\max}}$. In any case the algorithm returns the timed trace tt_h.

4 Design and Implementation

The aim of this chapter is to give an overview of the design of the GP application and discuss the functionalities of the Graphical User Interface (GUI). The application is an extension of the framework of the genetic programmer for TAs presented by Tappler et al. [47]. Their framework only implements the passive GP approach for TAs. We extended this framework by an implementation of our active GP approach which we introduced in Chapter 3.

The original framework is based on Java. We continued the implementation of the extension with the same programming language, which was Java version 1.8.0_151. To automatically build the tool and manage dependencies, we used Apache Maven version 3.5.2[1]. We developed the GUI with JavaFX 8[2], which runs on the Java Runtime Enviroment for the Java version 1.8.

The implementation uses two further external libraries. In the implementation TAs are exported in the `dot` format. To visualize the TAs in the GUI, we use Graphviz Java version 0.2.1[3]. We also use the library Apache Commons Lang version 3.5[4] which provides auxiliary data types like `Pair`, which we use for the representation of timed traces.

This chapter is divided into two parts. First, in Section 4.1 we introduce the four modules of our implementation. Second, in Section 4.2 we provide a brief manual for the GUI.

4.1 Modules

The implementation is composed of four modules: *Util*, *Test Driver*, *Genetic Programmer* and *GUI*. Figure 4.1 shows all four modules, where the arrows indicate the dependencies between the modules. For example, the Genetic Programmer module depends on two modules, which are the Util and the Test Driver Module. In the following sections we discuss the functionality of each module.

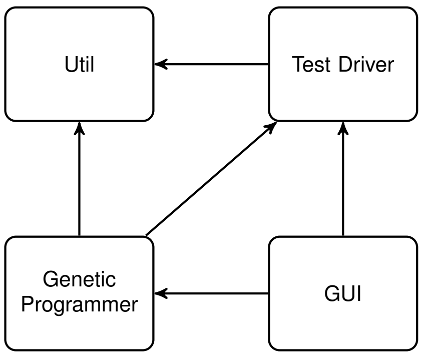


Figure 4.1: Our implementation of the active learning approach via GP is composed of four modules. These modules are akin to the implementation of the passive approach.

4.1.1 Util

The name *Util* is an abbreviation of utility. The Util module contains auxiliary functions and data structures which we require in other modules. The auxiliary functions include implementations to import and export TAs. To import a TA our implementation requires a file in `XML` format. The required structure of the `XML` document is chosen to be equal to the format that is generated by UPPAAL. UPPAAL is a tool

[1]`https://maven.apache.org/`
[2]`https://openjfx.io/`
[3]`https://mvnrepository.com/artifact/guru.nidi/graphviz-java`
[4]`https://mvnrepository.com/artifact/org.apache.commons/commons-lang3`

to model and process TAs and was already discussed in Section 2.3.4. Using this import functionality, we can use GP for timed systems that are modeled with UPPAAL. We implemented this import functionality to provide a convenient way to execute experiments that are based on existing TAs. However, we do not retrieve information directly from the imported TA, but rather use an interface to interact with the imported TA in order to learn them. Using this approach, we treat the imported TA as black-box timed system. Beside the import funtionality, the Util module also provides a possibility to export TAs as `dot` files.

Beside auxiliary functions to import and export the TAs, this module contains data structures to represent a TA, e.g., clocks, guards, actions or edges. They are all used in the class `TimedAutomaton` which is also included in this module.

4.1.2 Test Driver

The *Test Driver* comprises the entire functionality to generate tests and also defines the interface to interact with the SUT. The original Test Driver module of Tappler et al. included only an entirely random test generator for their passive approach. We extended this driver to generate tests according to our active approach.

The Test Driver provides the interface for the black-box SUT, which is equal for every experiment. The interface provides three methods that change the state of SUT and further two methods to retrieve information about the SUT. We change the state of the SUT by resetting the system, delaying the system by a time value, or executing an input action on the system. To retrieve information about the system, we can either query the possible input actions of the SUT or we can retrieve the overall elapsed time of the SUT.

Additionally, the Test Driver implements the timed trace generation. The implementation of the active approach is mainly done in the class `RandomTestGenerator`. This class implements parts of the timed trace selection proposed in Section 3.4 and the whole timed trace generation which we described in Section 3.5. This class requires access to an instance of the SUT, the used Random Delay Generator which we discussed in Section 3.5.3 and a random number generator. Using these variables, the class implements the following seven methods:

`findCounterexamples(int id, TimedAutomaton hypothesis, double pInput)`
: This method returns a found counterexample which reveals that the hypothesis is incorrect. In the case that no counterexample is found the function returns NULL. This method has three parameters: The first parameter `id` is used to assign a unique identifier to a possibly found counterexample. The second paramefter `hypothesis` is required since the method performs a random walk on the `hypothesis`. During this random walk we follow unknown inputs with a probability `pInput`, which is the third parameter. The first step in this method is to perform a random walk by calling the method `performRandomWalk`. This random walk generates a timed trace, which is then executed on the SUT. After the generation of the trace of the SUT and the trace of the hypothesis, we compare both traces. If the traces are different, a counterexample is found. In this case, the method processes the found counterexample as described in Algorithm 2 and returns the processed counterexample afterwards.

`performRandomWalk(TimedAutomaton hypothesis, double pInput)`
: This method performs a random walk in the `hypothesis` and returns the thereby generated timed trace. We discussed the implemented algorithm of this method in Section 3.5. The parameter `pInput` states the probability to perform inputs which may find an unexplored behavior. The maximum length of the random walk is accessible as global parameter in the class. In Section 3.5 we explained that the actual length of timed traces may differ from the maximum length. The trace can be shorter if we detect an infinite sequence of output actions, since we stop the random walk

after the detection. Additionally, the trace can be longer if we observe enabled outputs at the end of the timed trace. In any case, the method returns a timed trace.

`executeOnSUT(TimedTrace trace)`
: This method executes the provided timed trace `trace` on the SUT and returns the generated timed trace by the SUT.

`compareTraces(TimedTrace trace1, TimedTrace trace2)`
: This method compares each element of the timed traces `trace1` and `trace2`. We formally defined this function in Section 3.4. This method returns the length of the timed trace `trace2` if the traces are equal, otherwise the method returns the index of the first element in the timed trace that is different.

`extendTimedTrace(TimedTrace trace)`
: This method extends the provided timed trace `trace` by performing further delays and inputs on the SUT. This either triggers an output or the extension algorithm stops with a probability p_{stop}, which is an accessible global parameter. This extension is equal to the test-case generation of the passive inference approach.

`getTransitionDelay(List<Edge> edges)`
: This method returns a list of all constants which are used in the clock guards of all edges from the provided list `edges`.

`selectRandomInput()`
: This method returns an input action of the SUT, where the action is selected uniformly at random.

Both the active and the passive approach require a delay generator to generate timed traces. Therefore, the Test Driver comprises both *Random Delay Generators* which were introduced in Section 3.5.3.

4.1.3 Genetic Programmer

In the *Genetic Programmer* module the actual GP procedure for TAs is implemented. The module comprises all mutators, the test-case simulation and the fitness calculator. In addition, we find data structures to store the required parameters for the GP procedure. The class `GeneticProgrammer` comprises the entire functionality of the passive approach. We extend this module by the class `ActiveGeneticProgammer` which comprises the entire active GP approach as described in Chapter 3.

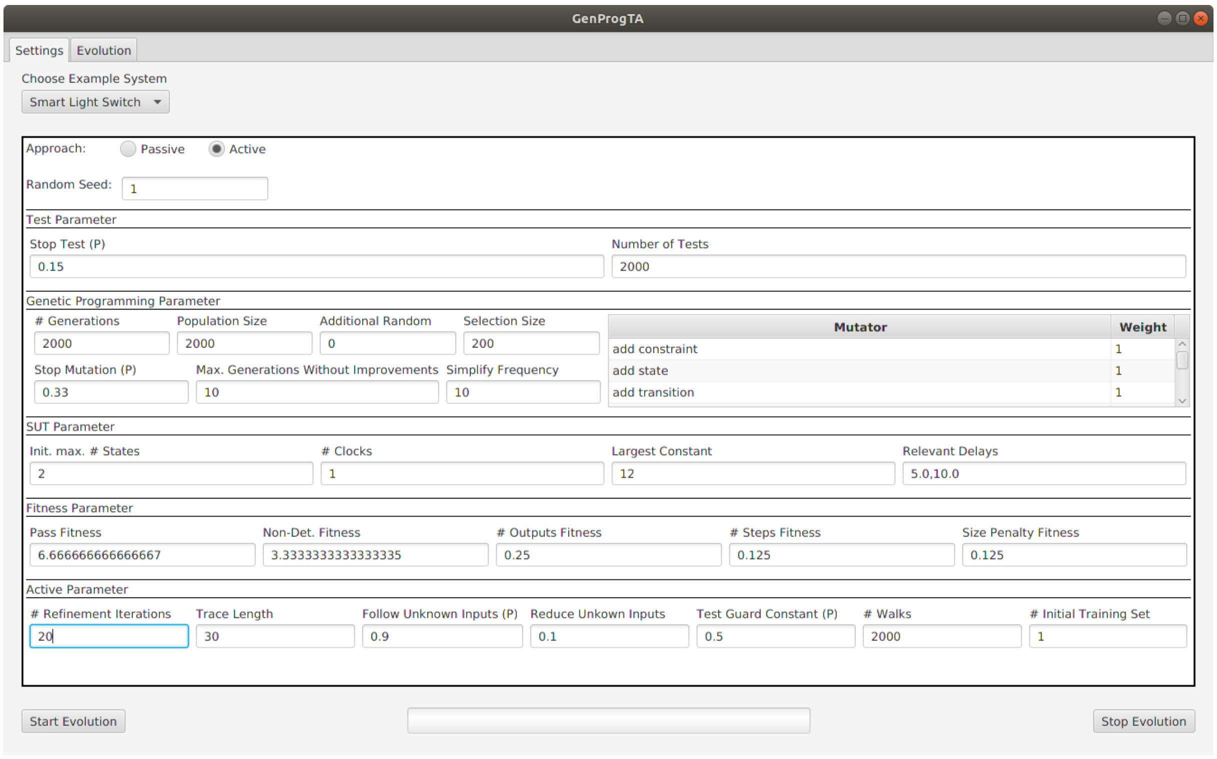
To execute and store the genetic programming experiments conveniently, we created an evaluation submodule. In this submodule we can infer models of the same timed system repeatedly with different training set sizes. To make the evaluation experiments more efficient, experiments with different sizes are executed in parallel. After finishing an experiment, the result is exported as an `XML` file.

4.1.4 GUI

The GUI provides convenient usage of the GP implementation for TAs. The advantage of the tool is, that it is able to visualize the learned timed automata and therefore gives an impression of the iterative improvement in the evolution of the generations.

We implemented the GUI with JavaFX. Using this library, the elements of the GUI are defined in an `XML` file and styled with `css`. We defined controller classes which manage the interactions of the user with the interface. The class `UiMain` defines the main class which launches the application.

The GUI contains a fixed number of examples including predefined parameters. These examples are defined in the class `Experiment`. The GUI provides the possibility to switch between these examples and configure the parameters for the GP approach.



Choose Example System: Smart Light Switch

Approach: Passive [✓ Active]

Random Seed: 1

Test Parameter

Stop Test (P)	0.15
Number of Tests	2000

Genetic Programming Parameter

# Generations	2000
Population Size	2000
Additional Random	0
Selection Size	200
Stop Mutation (P)	0.33
Max. Generations Without Improvements	10
Simplify Frequency	10

Mutator	Weight
add constraint	1
add state	1
add transition	1

SUT Parameter

Init. max. # States	2
# Clocks	1
Largest Constant	12
Relevant Delays	5.0,10.0

Fitness Parameter

Pass Fitness	6.666666666666667
Non-Det. Fitness	3.3333333333333335
# Outputs Fitness	0.25
# Steps Fitness	0.125
Size Penalty Fitness	0.125

Active Parameter

# Refinement Iterations	20
Trace Length	30
Follow Unknown Inputs (P)	0.9
Reduce Unkown Inputs	0.1
Test Guard Constant (P)	0.5
# Walks	2000
# Initial Training Set	1

Figure 4.2: GUI of the genetic programmer for TAs. This view shows all configurable settings for the learning process of the TA.

We extended the GUI for the application of the active approach. With our extension, we can select whether we want to infer the selected example based on the active or passive approach. Additionally, we can configure the parameters of the active approach. After we started the active GP, we can monitor the evolution of the hypothesis over the different iterations. The usage of the GUI is described in more detail in Section 4.2.

4.2 GUI Manual

This manual of the GUI gives a brief overview of the functionalities of the GUI. The GUI is an application to perform GP on TAs based on the passive and the active inference approach. Aligned with the topic of this thesis, this manual will primarily focus on the functionalities of the active approach. However the interface of the passive approach is very similar, except for the representation of the evolution of generations.

Figure 4.2 depicts the developed GUI. The selected view shows the interface to configure the settings for the GP procedure. As previously mentioned the GUI contains a set of predefined examples with the corresponding suggestions for the configuration of the GP approach. This version of the GUI provides four examples of timed systems that can be executed. The selected example in Figure 4.2 is the *Smart Light Switch* which is similar to the TA in Example 1.

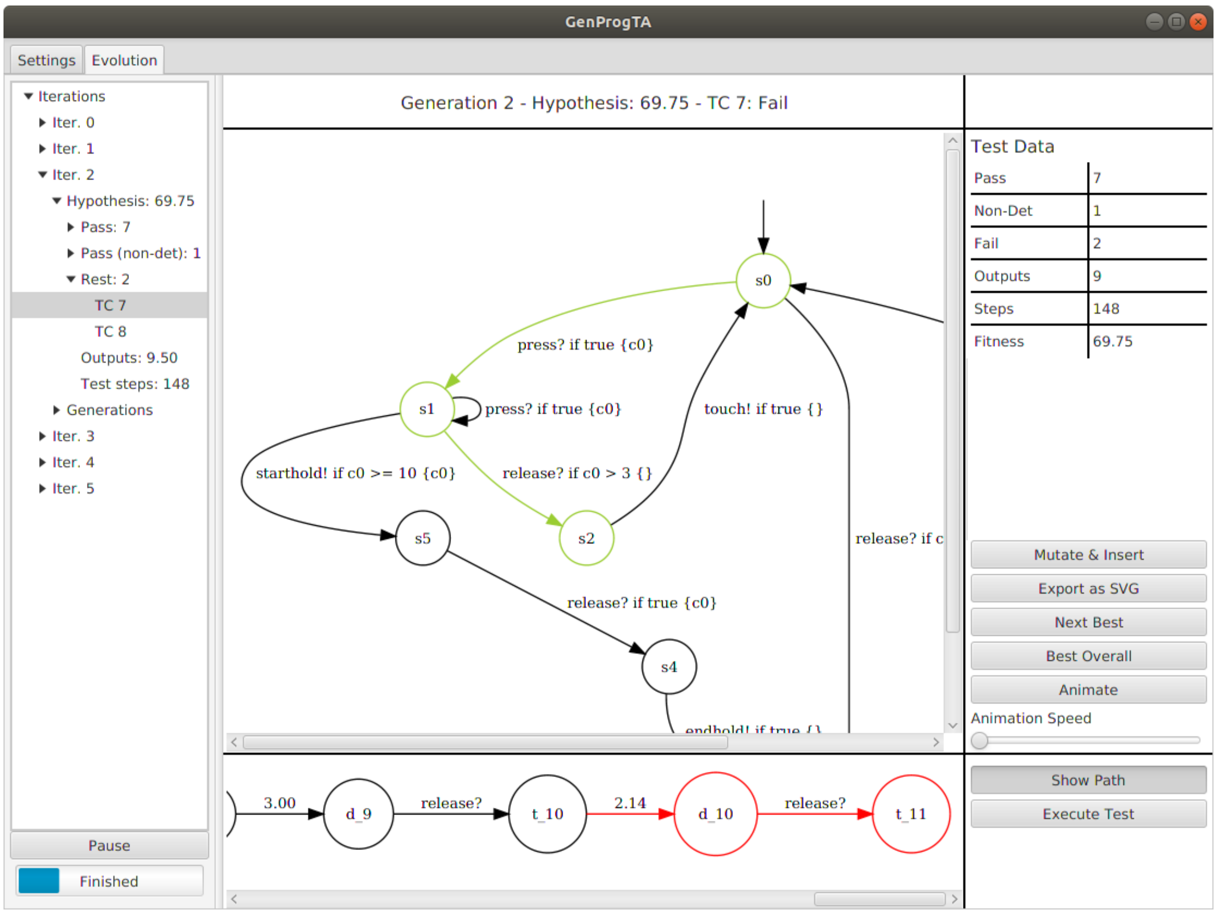
The tool provides the possibility to switch between the passive and the active approach. Since the active approach has additional parameters the interface enables the configuration of further parameters for this mode. All configurable parameters are described in the Chapters 2 and 3. A glossary of all parameters can also be found in Appendix B. To provide a more intuitive and user friendly GUI, we used more descriptive names for the parameters instead of short variable names. Table 4.1 provides a translation between the parameter naming of the GUI and the variable names used in this thesis.

Table 4.1: Translations between the configurable parameter fields in the GUI and the variables names used in this thesis.

GUI wording	Variable
Stop Test (P)	p_{stop}
Number of Tests	n_{test}
# Generations	$g_{\max}$
Population Size	n_{pop}
Selection Size	n_{sel}
Stop Mutation (P)	$p_{\text{mut}_{\text{init}}}$
Max. Generations Without Improvements	g_{change}
Simplify Frequency	g_{simp}
Init. max. # States	$\|L\|$
# Clocks	n_{clock}
Largest Constant	$c_{\max}$
Pass Fitness	$w_{\texttt{PASS}}$
Non.-Det. Fitness	$w_{\texttt{NONDET}}$
# Output Fitness	$w_{\texttt{OUT}}$
# Steps Fitness	$w_{\texttt{STEPS}}$
Size Penalty Fitness	$w_{\texttt{SIZE}}$
Refinement Iterations	$\frac{g_{\max}}{g_{\max_{\text{active}}}}$
Trace Length	n_{len}
Follow Unknown Inputs (P)	p_{input}
Reduce Unknown Input	$reduceInput$
Test Guard Constant	p_{trans}
# Walks	n_{attempts}
Set Size	n_{start}

We start the GP procedure of the selected approach by pressing the button with the label *Start Evolution*. We can stop this started evolution at any time by pressing the *Stop Evolution* button. After starting the evolution, we can switch to the evolution view, where we can observe the development of the inference process. Figure 4.3 shows this evolution view of an experiment with the Smart Light Switch. This experiment is done with the active approach. On the left we can monitor the individual iterations of the active learning approach. Each iteration outputs a hypothesis of the SUT. When we click on this hypothesis, we can see all timed traces added so far including the already found counterexamples. In addition, the GUI provides the possibility to execute timed traces on the selected hypothesis. For this, we find two buttons in the bottom right corner: *Show Path* and *Execute Test*. The *Show Path* button colors the edges which are visited during the random walk and the *Execute Test* button provides the possibility to execute the selected counterexample step by step. In case of a failing timed trace the tool also shows the reason for the non-conformance. For example, Figure 4.3 shows the execution path of a selected test case. The path of this test case in the TA is colored in green. The red colored parts in the test case indicate the element of the timed trace which contradicts the learned TA.

The table on the right presents a summary of the fitness data of the selected hypothesis. This summary includes the number of traces that pass, fail or are non-deterministic. Additionally, the applications shows information about the used set of timed traces, which includes the total number of outputs and the total number of steps. The last entry in the table contains the fitness value of the currently selected hypothesis, which is calculated with the fitness function of Section 2.6. Below this table we find further buttons. Using these buttons we can also perform manual mutations on the selected TA or export the TA as Scalable Vector Graphic (SVG). To provide an easier navigation through the iterations, the GUI provides buttons to jump to the TA with the next best fitness value (relative to the currently selected TA) and to the overall fittest TA. We can also activate an animation feature, where the GUI automatically jumps to the next best inferred TA.



Test Data

Pass	7
Non-Det	1
Fail	2
Outputs	9
Steps	148
Fitness	69.75

Figure 4.3: This evolution view shows the individual iterations of the evolution process. In this view we can monitor every inferred hypothesis and the found counterexamples. It is also possible to execute the counterexamples on the TAs.

5 Case Studies

This chapter presents the evaluation of our active learning approach. We performed an evaluation in seven different case studies. These case studies include three manually as well as four categories of randomly created timed systems. These categories of the randomly generated systems differ in number of locations and clocks. Each category contains ten different timed systems. Thus, in total we evaluated 43 different timed systems, where each experiment is repeated ten times. Therefore, these case studies comprise $18\,900$ learned TAs. In every case study we compare the evaluation of the active approach with the corresponding evaluation of the passive approach.

5.1 Experimental Setup

Our comparison between the active and the passive approach is based on four criteria:

Correctness. The correctness of a learned TA is based on our Conformance Relation 3.2 in Section 3.1. According to this conformance relation we denote a learned TA as conforming to the SUT, if the learned TA and the SUT behave equally for a set of timed traces. Let $\mathcal{TT}_{\text{test}}$ be a set of timed traces, where each trace in $\mathcal{TT}_{\text{test}}$ is generated in the same way as the traces used in the passive GP approach. We denote $\mathcal{TT}_{\text{test}}$ as our test set, which is used to assess the conformance between the learned TA and the SUT. In the following case studies the size of the test set $|\mathcal{TT}_{\text{test}}|$ is $2\,000$. The lengths of the trace in this test set are geometrically distributed. Shorter traces contain less information about the system, whereas longer traces provide more data about behavioral aspects of the SUT. The correctness states the percentage of passing traces of the test set $\mathcal{TT}_{\text{test}}$, where 100% indicates that every trace passes.

Needed Training Set Size. The needed training set size states the number of needed timed traces to learn a TA. For the passive approach this size is predefined and the maximum training set size is always equal to the size of the needed training set. In contrast, the active approach adds tests until no more counterexamples are found. Therefore, in the active approach we use only as many timed traces as required to learn a TA. However, we still define a maximum number of traces n_{test}, which the active approach cannot exceed. In our evaluation we compare the needed training set size with the maximal size. The aim of this evaluation is to assess whether the active approach is able to correctly learn a TA with fewer traces than the passive approach.

Test Execution Time. For this evaluation criterion we measure the required time to execute all needed timed traces on the system under test. The execution time of a single trace is the sum over all delays of the trace. The total test execution time of a training set is the product of the average execution time of all timed traces in the training set and the needed training set size.

Learning Runtime. This evaluation criterion states the required time to learn a TA. We define the learning runtime as the time from the start of the respective GP approach until the approach terminates and outputs a final solution for the learned SUT. This learning time is given in minutes (min).

We evaluated 43 different timed systems. The majority of the evaluated systems are similar to those used in the case studies of Tappler et al. [47]. We divide the timed systems into two main groups: The first group are manually created examples of timed systems which we find in industry, whereas the second group contains randomly generated timed systems. We evaluate three examples of industry applications while the remaining 40 evaluated systems are randomly generated.

For all evaluated systems we know the representation as TA. We also modeled the three timed system that we find in industry as TAs. However, we treat the TAs of the SUTs as black boxes for learning and access them only via the interface presented in Chapter 4. The availability of the representation as TA provides a possibility to assess the correctness of the learned TA, since we can compare the given TA of a SUT with the learned TA.

Hints. In Section 2.6 we introduced hints for delays of the SUT and introduced the concept of providing hints to learn a TA via GP. Hints are used to generate more helpful delays. We assume that providing hints improves the learning process, since the generated timed traces are more likely to have a higher coverage of the behavior of the SUT. On the contrary, not providing hints can lead to timed traces that skip behavioral aspects of the SUT, since the required delay to explore this behavioral aspect may not be generated. To show the impact of hints, we evaluate two of the three industry examples once by providing hints and once without providing hints. All other timed systems are evaluated with the provision of hints. In the following case studies we applied two different delay generators: one using hints and one not using hints. The first generator is the delay selection approach applied by Tappler et al. [47]. The second one is the random delay generator introduced in Section 3.5.3 which selects delays uniformly at random in the range $(0, c_{\max}]$. This was done to evaluate whether we are able to apply GP to learn a TA without providing hints.

Training sets. Our learning approach is always based on a set of timed traces. In Section 3.1 we explained that our target is to find a small training set that is sound and as exhaustive as possible. We want to determine how many timed traces are needed to correctly learn a TA, therefore we learned from training sets of varying sizes. We learned each timed system with 21 different training set sizes. The used sizes n_{test} are 50 and 100 to 2 000 with a step size of 100. According to our active learning approach, we add timed traces, which reveal a discrepancy between the learned TA and SUT, until the maximum number of timed traces n_{test} is reached. Therefore, in the case that no counterexamples are found, the number of timed traces needed may be smaller than the maximum number of timed traces n_{test}. The passive approach, however, learns TAs always based on a previously generated fixed set of timed traces with the size of n_{test}.

Number of experiments. In our procedure for the GP of a TA we make decisions where the outcomes are randomly distributed. Such decisions include, which kind of mutation should be applied or which delay should be chosen. Hence, the steps in the GP procedure for a TA may differ in each execution, which may influence the outcome of the learning approach or the required learning time. To mitigate a biased evaluation, we repeated each experiment ten times.

In summary, we learn 43 different timed systems, where two of these systems are also learned without using hints. This leads to 45 evaluations. Every evaluation is done for 21 different training set sizes. Each experiment for one training set size is repeated ten times. Hence, the amount of automata learned using the passive and the active approach sums up to

$$45 \times 21 \times 10 \times 2 = 18\,900$$

experiments. Since these case studies consist of such a large number of time consuming experiments we distributed the experiments over several resources including cloud services. These cloud services use virtual CPUs and, therefore, the actual computational resources can hardly be determined. Using this environment for the computation, absolute values of the learning runtime may be different when using a different setup. However, we assume that the trend of the results is independent from the underlying setup.

Parameters. To allow a fair comparison between the active and the passive approach, we chose a similar configuration of the parameters for both approaches. A detailed description of the used parameters can be found in Appendix B. For all experiments of the active and passive approach we set $n_{\text{pop}} = 2000$, $g_{\max} = 2000$, the initial $n_{\text{sel}} = \frac{n_{\text{pop}}}{10}$, $g_{\text{change}} = 10$, $p_{\text{mut}_{\text{init}}} = 0.33$, and $g_{\text{simp}} = 10$. In addition, the fitness function proposed by Tappler et al. [47] including the weighting of the individual fitness properties is equal to the one used in their case studies. For this, we set $w_{\texttt{OUT}} = 0.25$, $w_{\texttt{STEPS}} = \frac{w_{\texttt{OUT}}}{2} = w_{\texttt{SIZE}}$, $w_{\texttt{PASS}} = \frac{4w_{\texttt{OUT}}}{p_{\text{test}}}$, $w_{\texttt{NONDET}} = \frac{\texttt{PASS}}{2}$ and $w_{\texttt{FAIL}} = 0$.

We have to consider that in the proposed configuration of Tappler et al. [47] the whole knowledge about the SUT is provided from the beginning. This knowledge about the SUT is represented by the set of timed traces. On the contrary, our active approach extends this knowledge base with each iteration until the set of timed traces reaches the size n_{test}. Therefore, the active approach has additional parameters. In each iteration of the active approach we set the limit of generations $g_{\max_{\text{active}}} = 100$. For the whole active GP procedure we limit the number of generations to $g_{\max} = 2000$. Using this configuration, the active approach requires at most the same amount of generations as the passive approach.

Furthermore, in the active approach we incrementally increase the knowledge about the SUT. In each iteration we add new failing timed traces and try to refine the TA taking into consideration these newly added timed traces. Therefore, the number of failing timed traces in the active approach may be lower than the number of failing timed traces in the first few generations of the passive approach. Consequently, crossover and the development of a subpopulation may not be able to generate the desired effect in the active approach, since the number of failing test cases may be too small. For this reason, we set the crossover probability for the passive approach to $p_{\text{cr}} = 0.25$ and for the active approach we decrease it to $p_{\text{cr}} = 0.05$.

Another difference between the configurations of the active and passive approach is the distribution of the average timed trace length. In the passive approach the length is geometrically distributed and can be adjusted with the parameter p_{test}. Similar to the configuration of Tappler et al. [47] we set $p_{\text{test}} = 0.15$ for generating the initial set of traces. In our active approach we generated the timed traces by a random walk through the inferred TA. Therefore, we define $n_{\text{len}} = 40 = n_{\text{len}_{\max}}$ as the maximum length of timed traces generated by the random walk. This constant is based on the fact that in our case studies we target moderately sized systems with approximately 10 to 30 locations. However, the average size of actively generated timed traces is smaller, since we truncate the timed traces after the first found difference to the timed trace generated by the SUT.

For all evaluations of the active approach we configure $p_{\text{input}} = 0.9$, $reduceInput = 0.1$, $n_{\text{attempts}} = 2000$, and $p_{\text{trans}} = 0.5$. In each iteration we add n_{fail} failing timed traces to the training set, which is the base for the active GP approach. The value of n_{fail} defines the maximum timed traces that are selected for the improvement of the hypothesis in each iteration. This maximum number depends on the maximum size of the training set n_{test} and the number of iterations. We want to add in each iteration approximately the same number of timed traces, but all timed traces should be added in the first 90% of the iterations. This restriction is based on the assumption that if we have many failing timed traces, we still have iterations left to improve the hypothesis. We define the number of added traces n_{fail} by

$$n_{\text{fail}} = \left\lceil \frac{n_{\text{test}}}{\frac{g_{\max}}{g_{\max_{\text{active}}}} \times 0.9} \right\rceil \tag{5.1}$$

.

In the passive approach the number of needed timed traces always equals n_{test}, since the training set is given.

After the configuration of all parameters, we start the setup for the GP procedure. In the passive approach we create all n_{test} timed traces and an initial population. The setup for our active approach is akin to the passive approach. For this, we start with the generation of n_{start} random test sequences and execute them on the SUT to generate the corresponding timed traces. In the case studies we set $n_{\text{start}} = 1$. We use this single timed trace to learn an initial timed automaton, which is our first hypothesis $\mathcal{H}$ of the SUT. The advantage of this limited initial timed trace set is that the initial TA can be learned via GP with little effort.

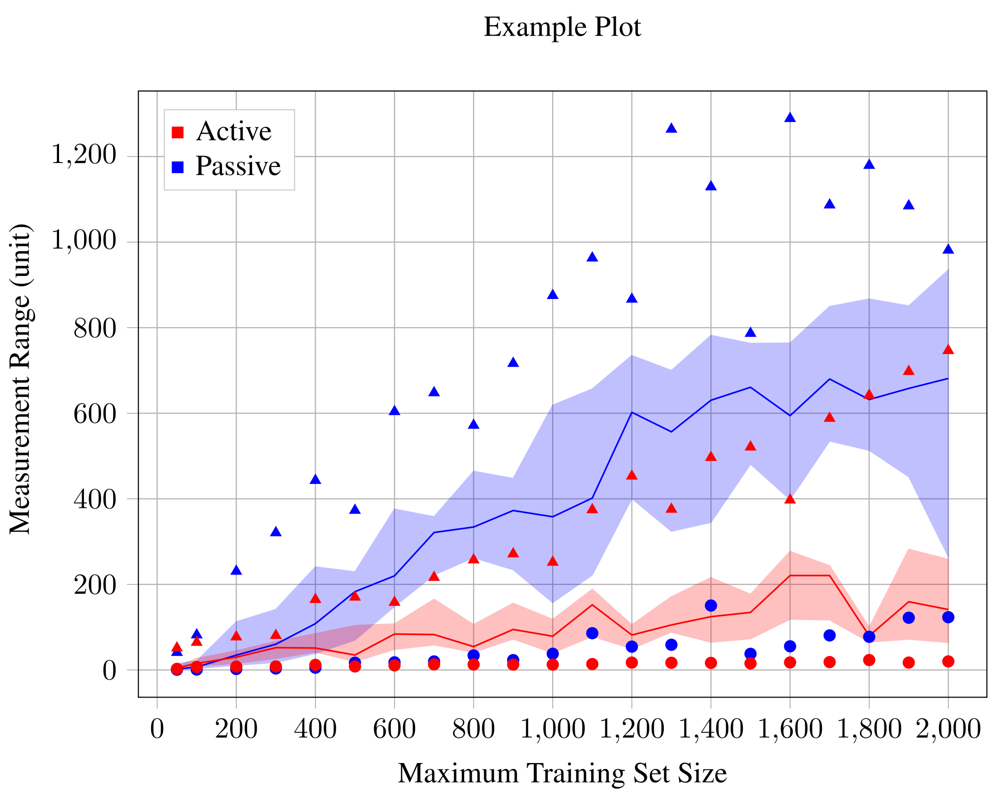


Figure 5.1: This plot shows the form of representation which is used to represent the evaluation criteria in the following case studies.

Presentation of data. Previously in this chapter we mentioned that we measure four different criteria. In our case studies we depict all four criteria by the same representation scheme. Figure 5.1 shows an example plot of the used representation scheme. Every plot depicts both the evaluation of the passive and the active approach. These statistics are computed from results of the ten repeated runs of each experiment. The active results are represented in red and the passive results are represented in blue, respectively. The solid line indicates the median of the results and the surrounding area in the corresponding color shows the range between the first and third quartile. The x-axis always represents the maximum training set size n_{test} of the experiment and the y-axis the currently considered evaluation criterion. The markers in the shape of a triangle indicate the maximum and the dots the minimum value. We only use the triangle and the dot markers in the plots where we evaluate randomly generated TAs. Plots that evaluate a single TA show only the median and the range between the first and the third quartile.

Additionally, the case studies of randomly generated timed systems include box plots with a corresponding color scheme. These box plots may also show outliers that are depicted as dots. We denote the *interquartile range* as difference between the first and third quartile. A data point is an outlier if the point is either 1.5 times the interquartile range higher than the third or lower than the first quartile.

Improvement criteria. The goal is to compare the evaluations using our presented active learning approach with the passive learning approach proposed by Tappler et al. [47]. The results of the active approach are an improvement over the passive approach if the following criteria are met: for the correctness criterion we denote the results as better if the result is higher and for all other criteria the results are better the lower the value. We quantify the improvement by the difference between the active and the passive results normalized by the active result. In this chapter we make statements about the achieved improvements of our active approach. Unless otherwise specified, the improvement is calculated from the median of the results of the ten repeated runs of each experiment.

In the following sections case studies involving three manually created TAs, which represent examples from the industry, and 40 randomly created TAs are presented. In Section 5.2 we discuss the evaluation with respect to our Smart Light Switch that we already discussed in the previous chapters. In Section 5.3, we introduce an industry example of a timed system, that shows that the alarm system of a vehicle can be learned with GP. In Section 5.4, we discuss a part of a real-world example of a particle counter. Finally, in Section 5.5 we present the evaluation with respect to 40 randomly generated TAs, which we divide into four different categories based on the number of locations and number of clocks.

5.2 Smart Light Switch

The Smart Light Switch models a sensor that shows different behavior depending on how long the sensor is touched. We introduced the Smart Light Switch in Example 1 and Figure 2.2 depicts the corresponding TA. Since this system has only two inputs $\Sigma_I = \{press?, release?\}$ and can be modeled with five locations, we assume that this example is the easiest to learn compared to the other evaluated examples.

We evaluate this example with two different configurations. In the first configuration, we provide hints and a manually picked largest constant for the generation of delays. In the second configuration, we provide an upper limit for the selection of delays instead of hints. First, we discuss the evaluation with hints and then the results without hints.

5.2.1 Evaluation with Hints

For the evaluation of the Smart Light Switch with hints, we provide the hints 5 and 10. In addition, we set $c_{\max} = 20$. We provide the medians of all four evaluation criteria and the corresponding improvements of these criteria for all training set sizes in Appendix C.

Figure 5.2 shows the correctness of the active and the passive approach. The active approach has a median test set error of zero for every training set size. The passive approach also has a median correctness of 100% for the two smallest training set sizes, 50 and 100. However, for larger training sets the median correctness of the passive approach fluctuates between 72.1 % and 100 %. We assume that the reason for this is that the fitness value of the population reaches a local maximum. A local maximum is a point in the evolution of the population where the current generation has a higher fitness value than the previous and the following generation. Performing any small mutations on the generation in a local maximum would cause a decrease in the fitness value. A possible reason for this is that the learned TA passes many timed traces of the training set. To pass the other failing traces, larger mutations would have to be made, leading to a decrease of the fitness value for the next generation and, therefore, do not pay off. As a result, no further mutations are performed and the learned TAs stay unchanged until the maximum number of generations is reached. We assume that it is more likely to observe the occurrence of getting stuck at a local maximum when we use larger training set sizes, since single failing traces have less weight. The active approach apparently has no problems with local maxima. One possible reason for this is that we add failing traces in each iteration. Therefore, the failing traces have a higher weight and it pays off to perform further mutations.

Comparing our evaluation with the results of the Smart Light Switch proposed by Tappler et al. [47], we assume that the correctness of the passive approach is lower due to the configuration of our experiment. We set $c_{\max}$ twice as large as Tappler et al. [47] and, therefore, we produce larger delays. We assume that the training set with a larger $c_{\max}$ provides a range of valid time constraints that is too large. Thus, the complexity to learn a conforming TA is higher, especially with a large predefined training set. In contrast, in the active approach we test the guards of our hypothesis and add timed traces that violate the learned guards if they are wrong. Using these techniques for the active approach, we can learn the guards of the SUT without providing an accurate largest constant.

The evaluation of the needed training set sizes of the active approach in Figure 5.3 reveals that a correct TA can be learned with a small training set. For the active approach the median training set size is between 17.5 and 238.5. Therefore, we can improve the median of the needed training set size by up

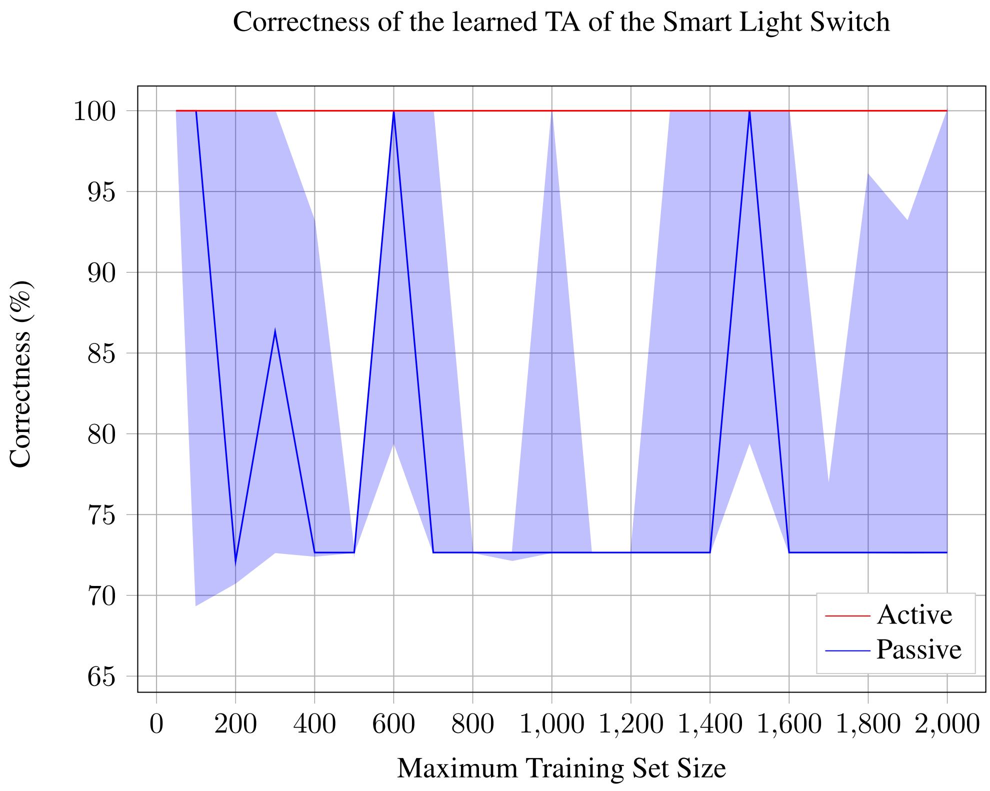


Figure 5.2: Correctness of the learned TA of the Smart Light Switch using hints

to 895 % with the active approach. In addition, Figure 5.4 shows akin results for the test execution time. The improvement of the execution time when using the active approach is at least 87.2% and increases up to 731.6%. According to these numbers the median execution time of the actively generated training set is up to 8.3 times lower than the execution time of the passive set.

We observed the largest improvement with the active approach when evaluating the learning runtime as shown in Figure 5.5. However, to get a correct impression of the real improvement, we have to take the correctness evaluation in Figure 5.2 into account. The passive approach achieves a median correctness of 100% for the training set sizes 50, 100, 600 and 1 500. We also see a drop in the execution time for the same training set sizes. The reason for this is that the passive approach terminates after learning a correct solution, otherwise this approach tries to find a better solution until the maximum number of generations is reached. For all other training set sizes the passive approach runs out of generations without finding a solution where every trace of the training set passes. Nevertheless, if we only consider the training sets where the passive approach is able to learn conforming TAs, the active approach can also achieve an improvement. The median required runtime to learn a correct TA of the Smart Light Switch with the active approach is between 0.9 and 9.8 minutes. The median runtimes of the passive approach, where the approach achieves 100% correctness, show that the passive approach requires at least 10 minutes to learn a correct TA and can take up to 90.6 minutes.

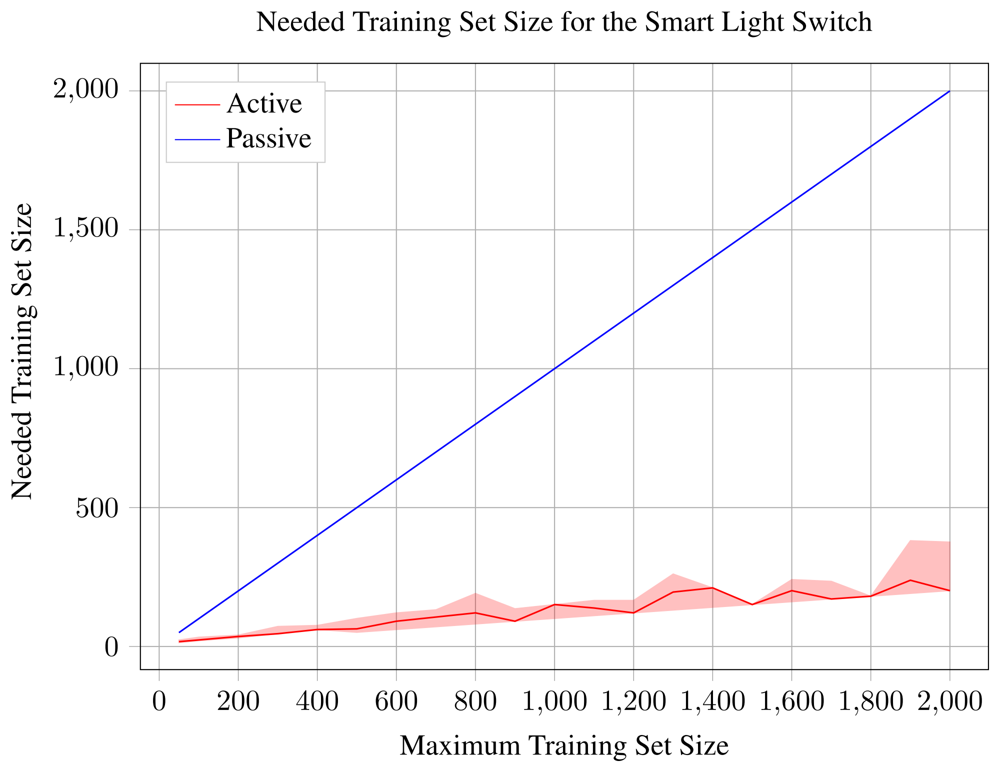


Figure 5.3: Needed training set size for the Smart Light Switch using hints

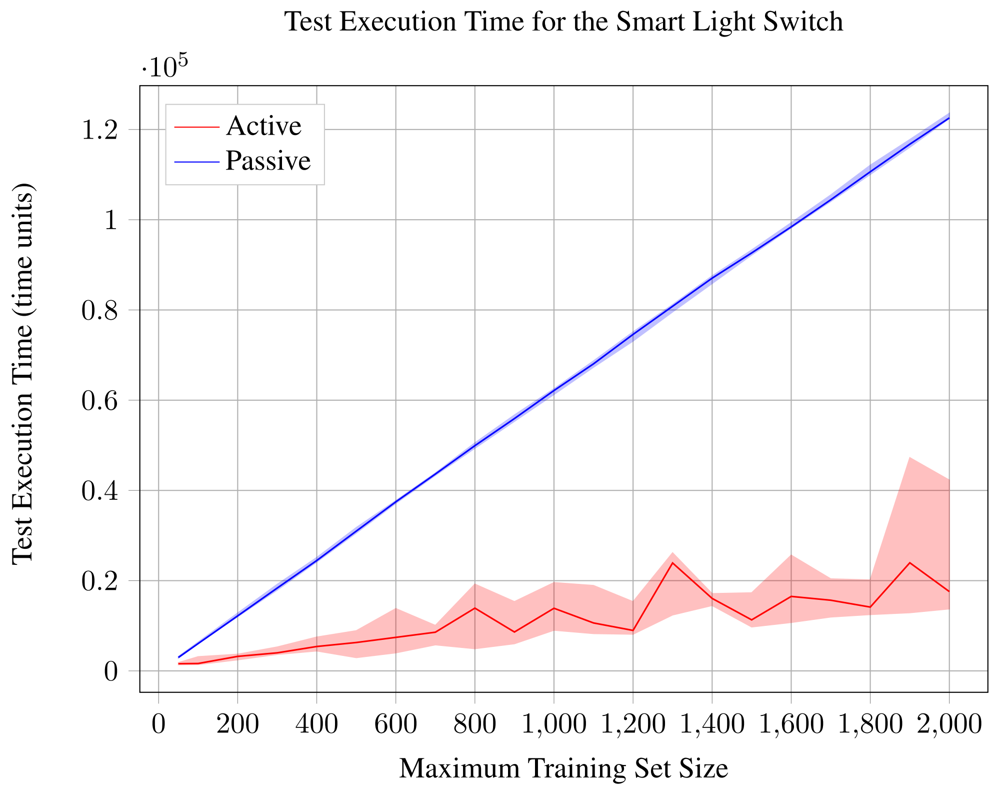


Figure 5.4: Execution time of training set, which is used to learn the TA of the Smart Light Switch using hints

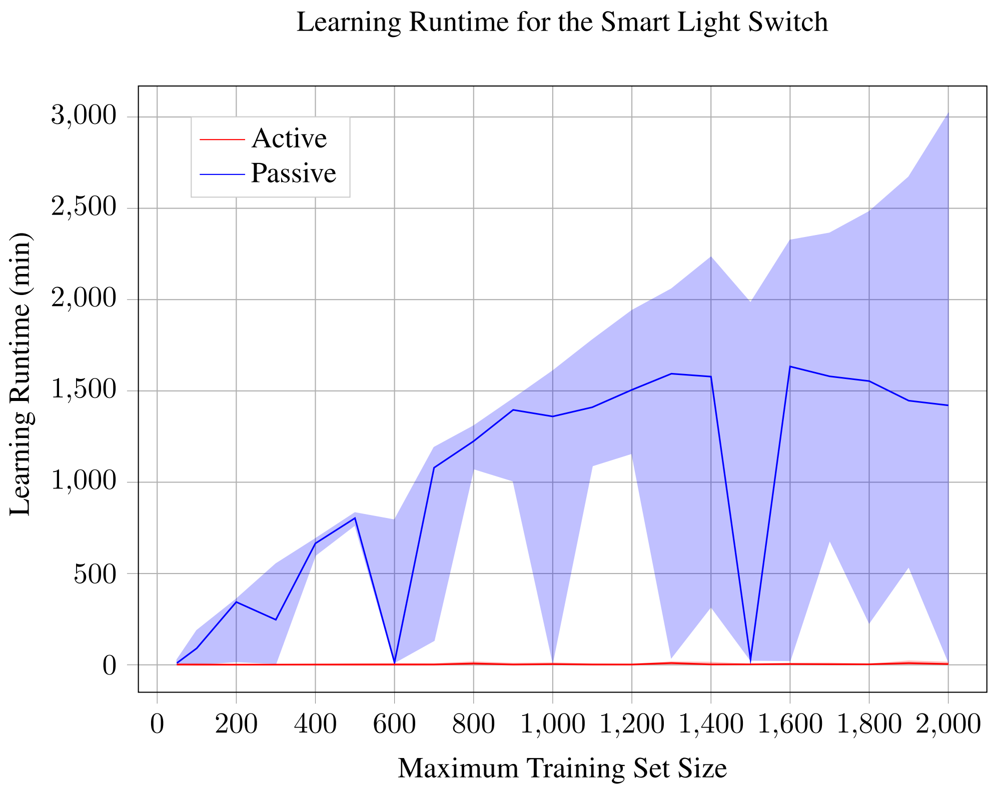


Figure 5.5: Learning runtime for the Smart Light Switch using hints

5.2.2 Evaluation without Hints

We evaluated the Smart Light Switch a second time, but this time we did not provide any hints. Therefore, the delay generator only selects delays based on a largest constant, which we set to $c_{\max} = 30$. All other settings for these experiments remain unchanged compared to the experiments with hints. In Appendix C we provide the median results of the performed evaluation.

We depict the correctness in Figure 5.6. Like in the evaluation with hints the active approach learns the Smart Light Switch correctly for any training set size. The correctness of the passive approach, which first grows and then decreases is more interesting. The drop of the median starts at a training set size of 100 and stops falling at a training set size of 500. Afterwards, the median correctness of the passive approach stays at 87.6 %. We assume that this drop is also due to a local maximum of the fitness value as explained in the evaluation with hints. Furthermore, we observe that this example without hints is more complex, since the median correctness of the passive approach never reaches 100%. Additionally, we see again that in the active approach local maxima are not a problem.

Figure 5.7 shows that the active approach requires a relatively small set of timed traces to infer a correct solution. Like in the evaluation with hints the active approach can improve the set size by up to 895%. These smaller training sets also reduce the time to execute the timed traces on the system, which is depicted by Figure 5.8. The median execution time can be improved by 475.4% by using the active approach.

The most significant difference between the active and the passive approach in this evaluation is the learning runtime as shown in Figure 5.9. The active approach is able to learn a correct TA already within 1.8 minutes, whereas the passive approach requires at least 20.2 minutes. However, the passive approach never reaches a median correctness of 100%. As explained in the evaluation with hints, the runtime increases significantly if the GP approach cannot find a solution where all traces pass. Therefore, we observe this significant difference between the active and the passive approach, where the improvement of the median runtime for the active approach is at least 369.8% and increases up to 9 386.2%.

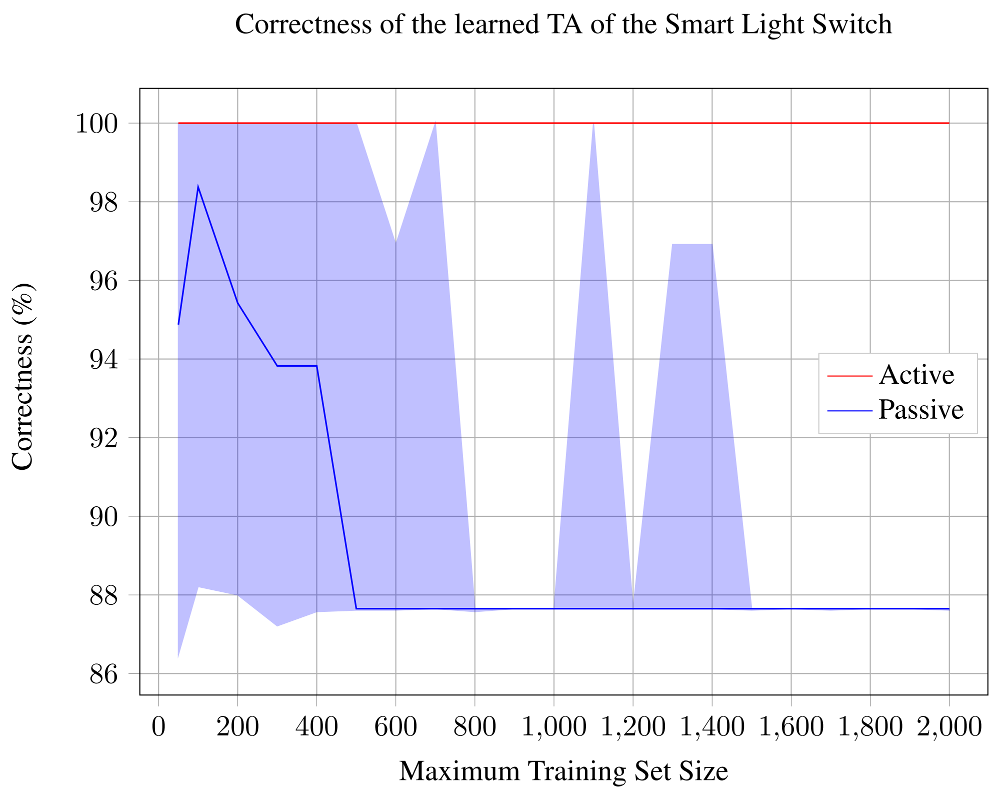


Figure 5.6: Correctness of the learned TAs of the Smart Light Switch without using hints

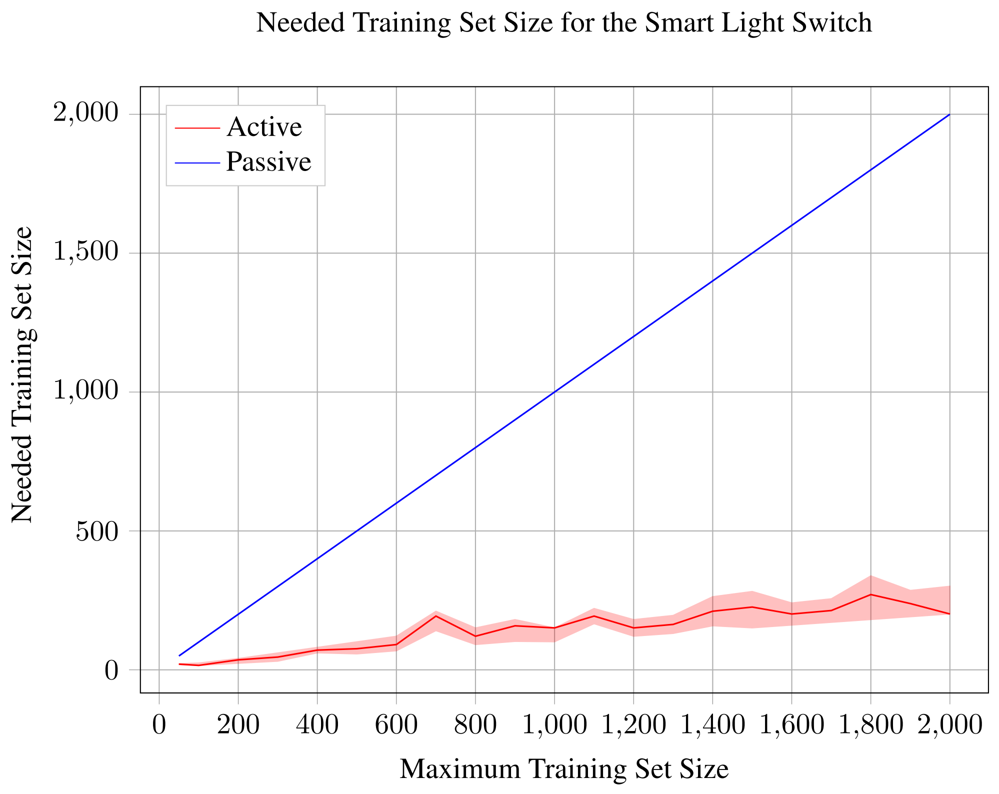


Figure 5.7: Needed training set size for the Smart Light Switch without using hints

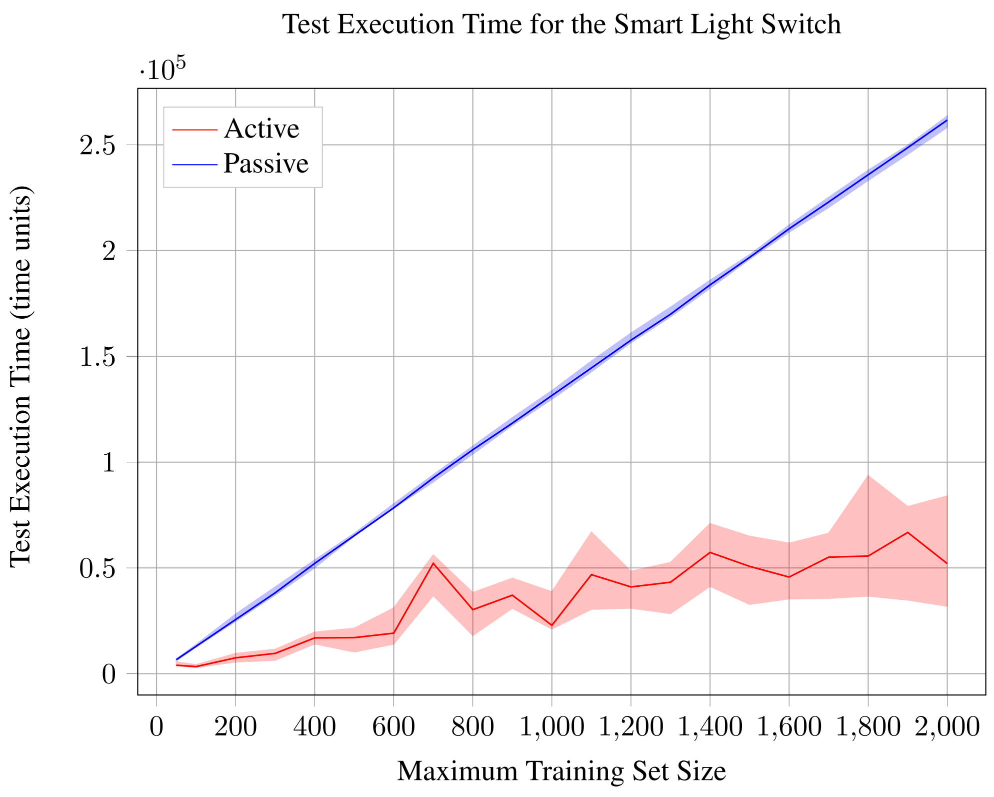


Figure 5.8: Execution time of training set for the Smart Light Switch without using hints

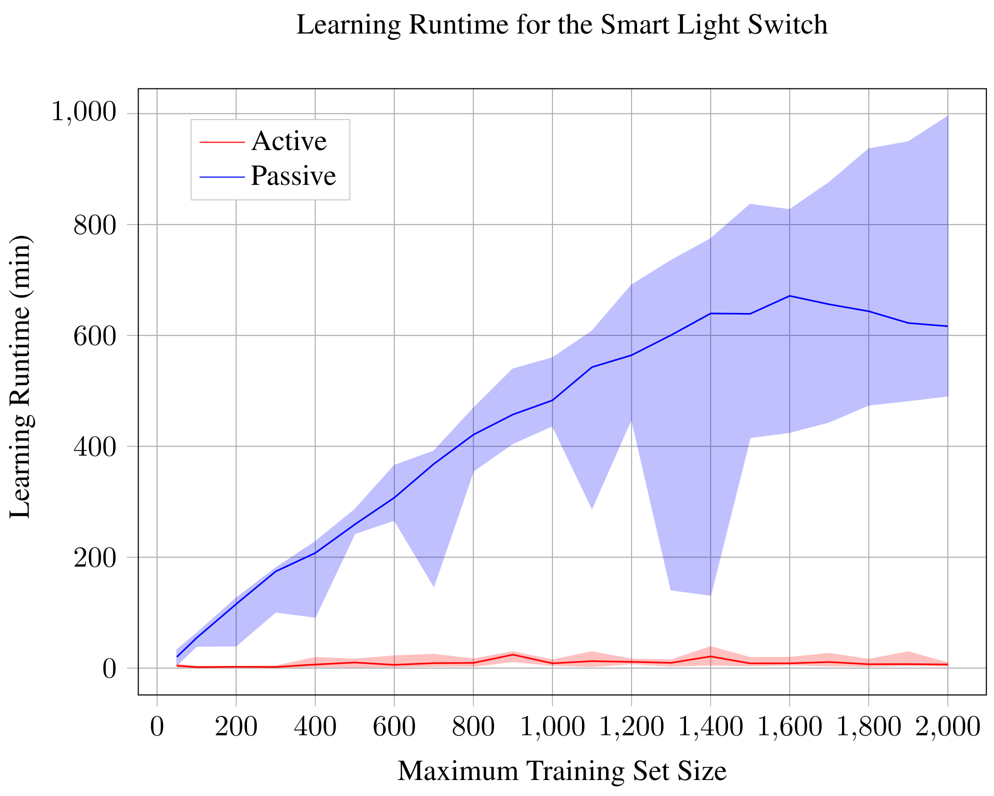


Figure 5.9: Learning runtime of the Smart Light Switch without using hints

In summary, the evaluations of the Smart Light Switch with hints and without hints show akin results. In both evaluations we see that the active approach can correctly learn the timed system. Overall, the active approach with hints performs slightly better than the active approach without hints. The provision of hints improves the needed training set size and the test execution time. Additionally, the active approach with hints can learn a correct TA with a median runtime between 0.9 minutes and 9.8 minutes, whereas the active approach without hints has a median runtime between 1.8 minutes and 24.2 minutes.

5.3 Car Alarm System

The Car Alarm System (CAS) is a model from the automotive industry, which describes the behavior of an alarm system of a car. In more detail, this system models when the car alarm should be armed as well as the behavior in the case of an attempted access to a locked car. In the literature we find various applications [2, 3] where this example is used for the evaluation of testing approaches.

Figure 5.10 shows the TA model of the CAS, which is modeled with one clock c. The CAS has four different inputs $\Sigma_I = \{open?, close?, lock?, unlock?\}$. At the beginning the car is open and unlocked. If we close and lock the car it gets armed after two time units. In Location l_4 the car is locked, closed and armed. We can now either disable the alarm by unlocking the car or trigger the alarm by opening the car. In the case that we unlock the car, we go back to the Location l_2 where the car is closed and unlocked. Opening the car without unlocking disables the armed state and enables the flash and the sound. We can stop the flash and sound by unlocking the car. According to governmental regulations the sound has to be stopped after three time units and the flash after additional 27 time units. Like the Smart Light Switch, we evaluate the CAS according to two different settings: with hints and without hints.

5.3.1 Evaluation with Hints

In this evaluation of the CAS we provide hints and a largest constant $c_{\max}$. The hints are $\{2, 3, 27\}$ and we set $c_{\max} = 33$. We state the medians of the results for all training set sizes in Appendix C.

Figure 5.11 depicts the correctness of the passive and the active approach. We see that the active approach can correctly learn the TA of the CAS from a training set containing 50 timed traces, whereas the passive approach requires at least 200 timed traces. In summary, both approaches can correctly learn the CAS and we cannot observe issues like getting stuck at a local maximum.

Furthermore, Figure 5.12 shows that the active approach never requires the maximum training set size. The active approach can improve the training set size by up to 341.8%. Since we reduce the set of executed tests significantly, the overall execution time of the timed traces also decreases for all evaluated training set sizes except 50 as shown in Figure 5.13. The active approach can improve the execution time of the set of timed traces by up to 190%. However, for the maximum training set size of 50 the active approach has a longer median execution time than the passive approach since the needed training set size of the active approach is closer to n_{test} than for other training set sizes.

We see a larger difference between the active and the passive approach in the required time to learn the automaton. We depict the evaluation of the learning runtime in Figure 5.14. The active approach requires more time to learn the CAS for the training set sizes 50 and 100. However, if we look at the respective correctness results in Figure 5.11 of these set sizes, we see that the active approach learns correctly, whereas the median results of the passive approach are not 100% correct. For all other training set sizes, the median of the required learning time is below the passive approach, with an improvement of up to 467.4%.

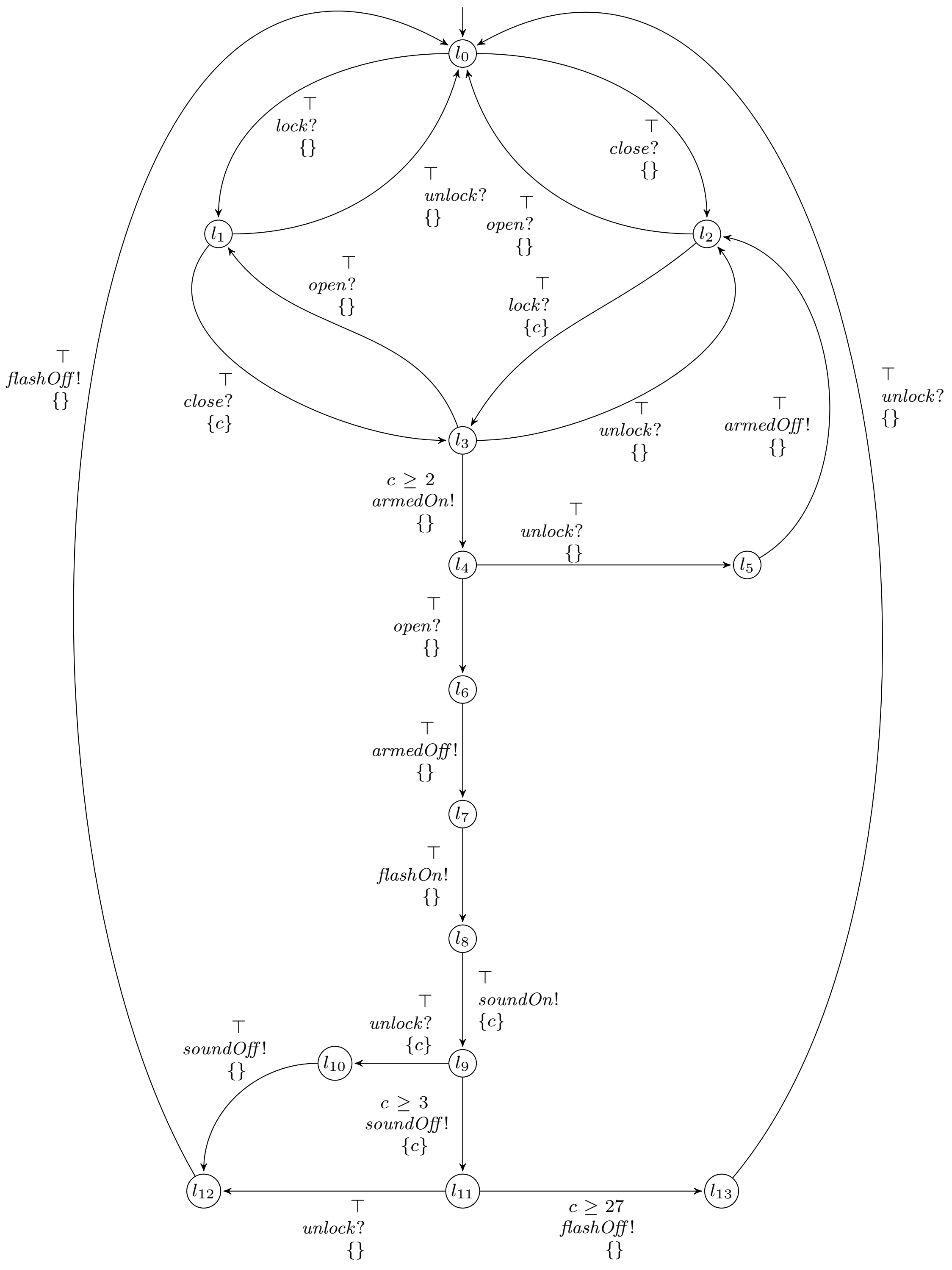


Figure 5.10: TA of the CAS

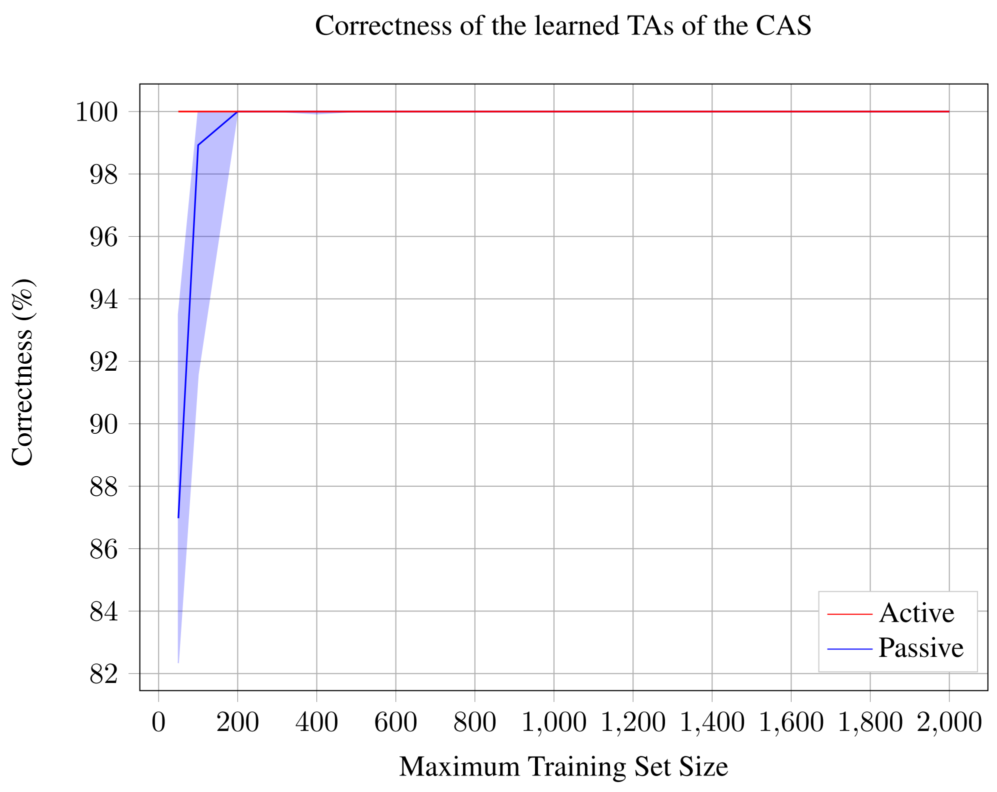


Figure 5.11: Correctness of the learned TAs of the CAS using hints

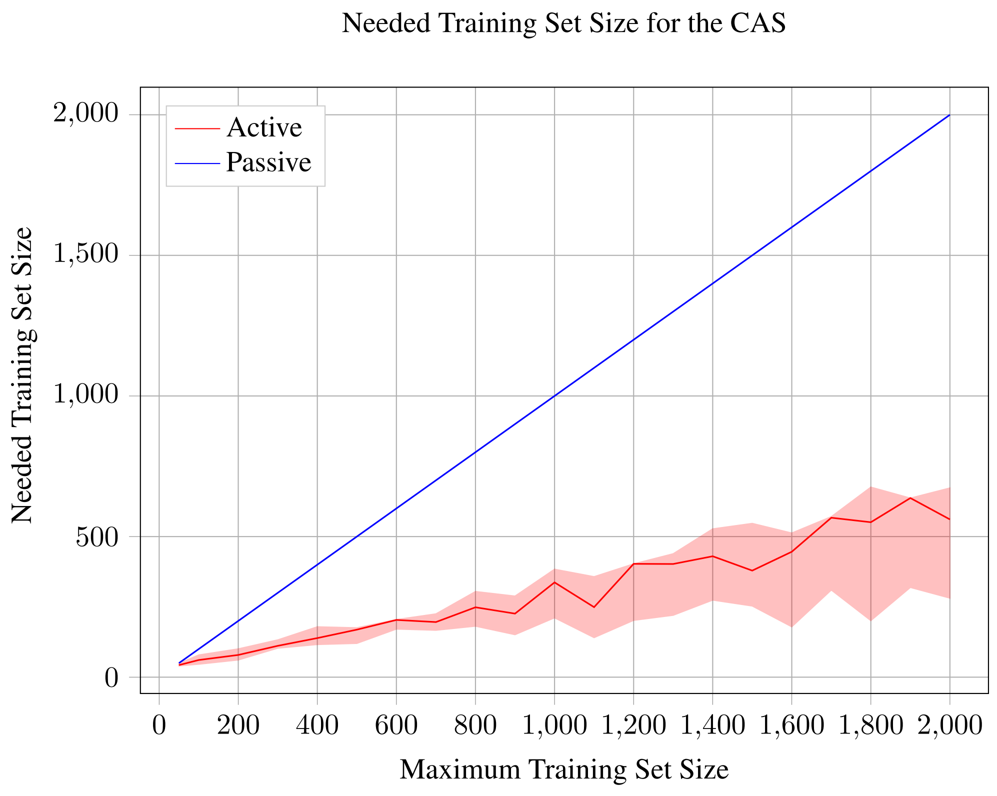


Figure 5.12: Needed training set size for the CAS using hints

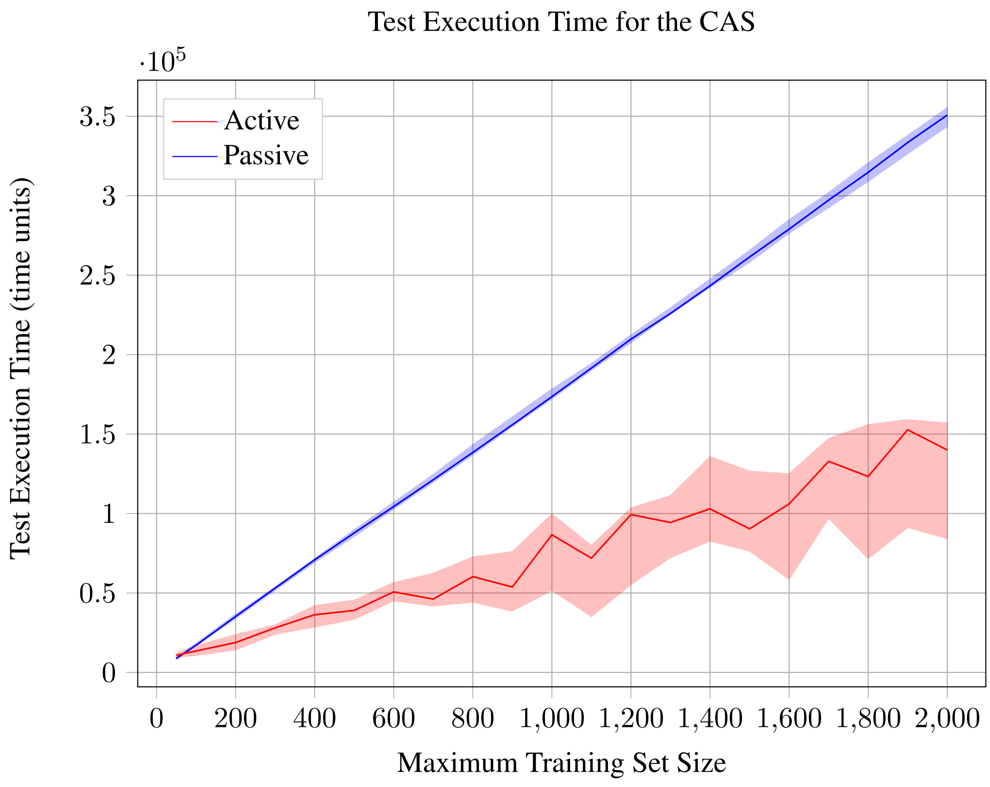


Figure 5.13: Execution time of training set for the CAS using hints

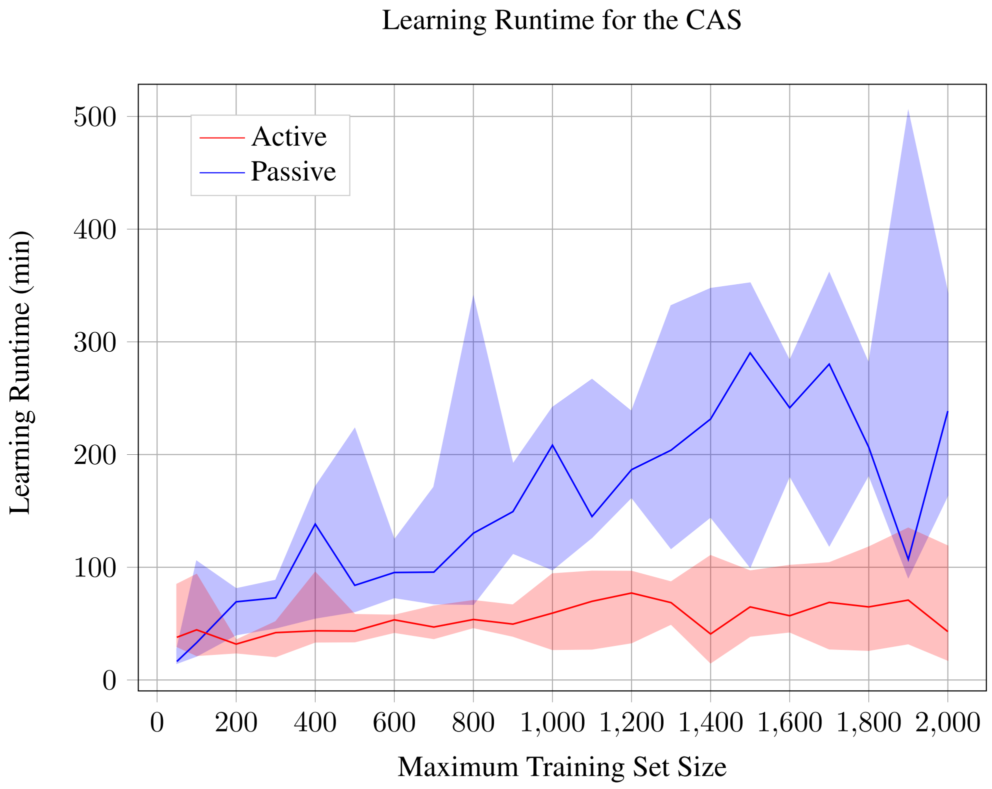


Figure 5.14: Learning runtime of the CAS using hints

5.3.2 Evaluation without Hints

As in the evaluation of the Smart Light Switch, we evaluate the CAS also with a random delay generator that selects delays without the usage of hints. We set the largest constant for the random delay generator $c_{\max} = 33$, which is equal to the evaluation with hints. In contrast to the case study of the Smart Light Switch, the results of this evaluation are more akin to the evaluation of the CAS which uses hints. We assume that this behavior is observable due to the characteristics of the CAS. All edges which are restricted by a clock guard are labeled with an output action. Since we assume that outputs are performed immediately when an output is enabled, the delay in the timed trace is equal to the guard. In the Smart Light Switch example, we also find guards on the edges which are labeled with inputs, therefore, these edges may induce various delays in the timed trace. Thus, we assume that guards of inputs are harder to learn. In Appendix C we show the median values of the evaluation of the CAS.

Regarding the median correctness in Figure 5.15, we see that the active approach learns for all training set sizes a correct TA of the CAS, whereas the passive approach requires at least a training set size of 400. Furthermore, Figure 5.15 shows that the first quartile of the passive approach is for certain training set sizes below $100\ \%$ correctness.

Figure 5.16 and Figure 5.17 show a similar picture as the evaluation of the CAS which uses hints. Using the active approach improves the median needed training set size up to 343.2%, which is equal to the improvement of the evaluation with hints. For the test execution time, which is depicted in Figure 5.17, we observe an akin behavior. The improvement of the overall time to execute is slightly lower compared to the evaluation that uses hints. However, we can still improve the execution time up to 174.3%.

Completely different to the evaluation of the Smart Light Switch, we observe that the overall runtime for learning the CAS decreases for the passive approach on the evaluation when we do not provide hints. In the evaluation with hints we observe a maximum runtime of 290.2 minutes for the passive approach, whereas the maximum runtime for the evaluation that uses no hints is at most 156.9 minutes. The runtime of the active approach does not vary as much compared to the runtime with hints. In Figure 5.18 we can see for all training set sizes that the gap between the active and the passive approach decreases. Regarding the median learning runtime, the passive approach outperforms the active approach for the training sets smaller than 600. However, in Figure 5.15 we observe that the passive approach can only achieve $100\ \%$ correctness starting from a training set size of 400. Even though the gap between the passive and the active approach decreases, the active approach achieves an improvement of up to 176.4%.

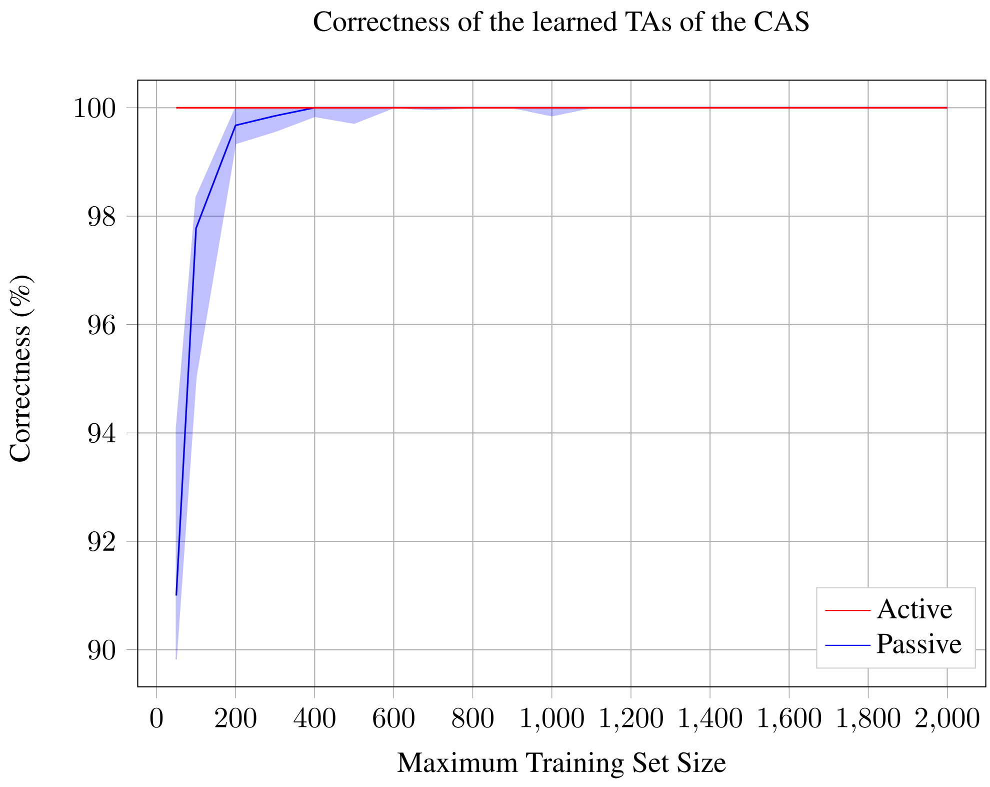


Figure 5.15: Correctness of the learned TAs of the CAS without using hints

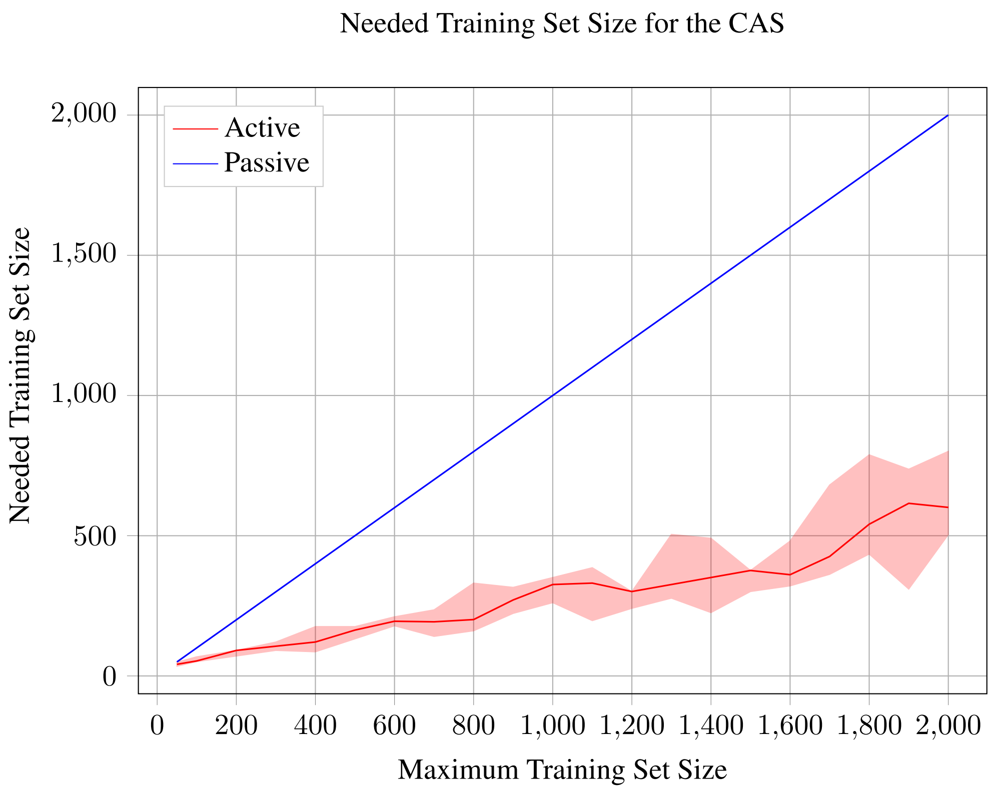


Figure 5.16: Needed training set size for the CASs without using hints

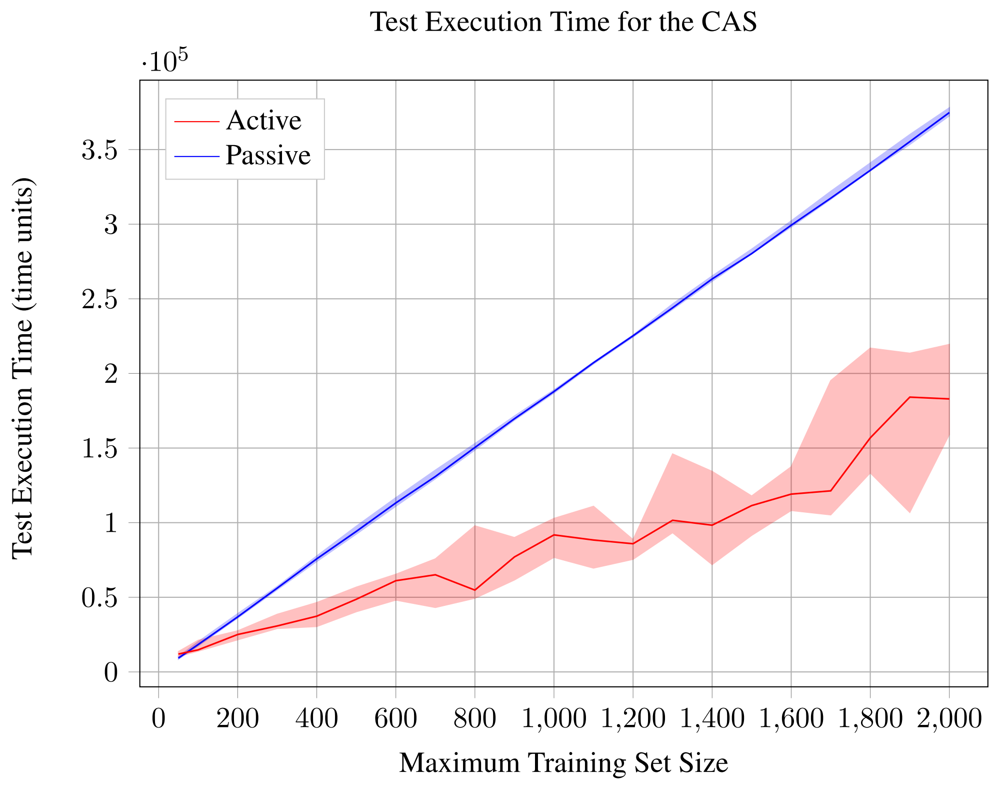


Figure 5.17: Execution time of training set for the CASs

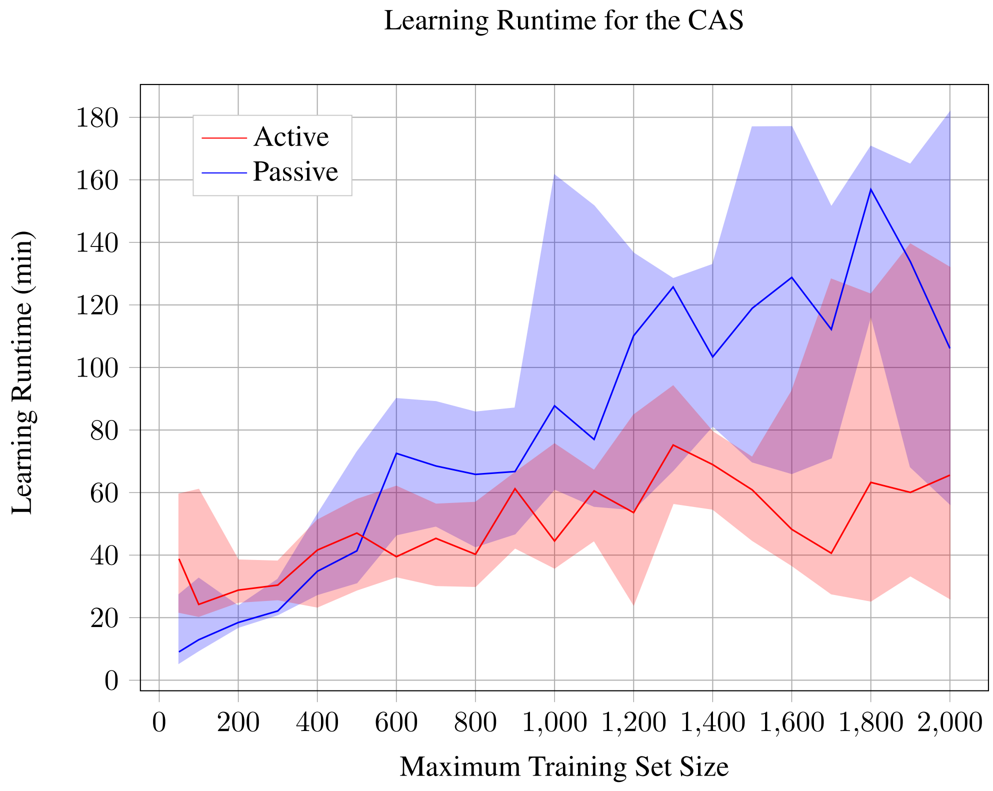


Figure 5.18: Learning runtime of the CASs without using hints

5.4 Particle Counter

The third manually created example is the Particle Counter. In the literature, Aichernig et al. [1] use this Particle Counter to evaluate a model-based mutation testing approach that uses a Unified Modeling Language state machine. This system models parts of a device that counts particles in exhaust gas. The Particle Counter has three different states: *pause*, *standby* and *active*. We can switch between the states by performing inputs. The entry to a new state and in some cases also the exit from a state trigger an output. Furthermore, the active state allows different tasks to be executed which we trigger by performing different inputs. All tasks in the active state return to the previous state either by themselves after 10 time units or by an input that enables the pause or the standby state. The initial state is the pause state.

Figure 5.19 shows the TA of the Particle Counter. In Location l_0 we start with an enabled output entering the pause state in Location l_1. In Location l_1 we can either switch to the standby state by performing the input $SetStandby?$ or perform $SetPurge?$ which triggers a task of the active state. If we perform $SetPurge?$ we return to the pause state after 10 time units. In Location l_7 we are in the standby state. The standby state provides the possibility to perform three further tasks of the active state which can be executed via the inputs $\{LeakageTest?, SetPurge?, ResponseCheck?\}$. After 10 time units these active tasks return to Location l_7 after performing the corresponding output actions. However, in the active state we can always switch to the pause and the standby state before 10 time units pass.

For the Particle Counter we performed only one evaluation. We evaluated the Particle Counter providing hints and a largest constant $c_{\max} = 20$. For this case study, only evaluation with hints is appropriate, since only one hint is provided. The provided hint in this example is 10. Furthermore, like the CAS, the Particle Counter only has clock guards on the edges labeled with outputs. Therefore, we assume that we may learn the guards easier than for other timed systems since the guard constants are observable in the timed trace. For this, we assume that the evaluation with hints produces akin results as the evaluation without hints. An interesting factor is the complex structure of the Particle Counter. We assume that this example is complex due to the fact that long traces are required to enter every location. Additionally, in Location l_7 we have several possible inputs that lead to different paths in the TA.

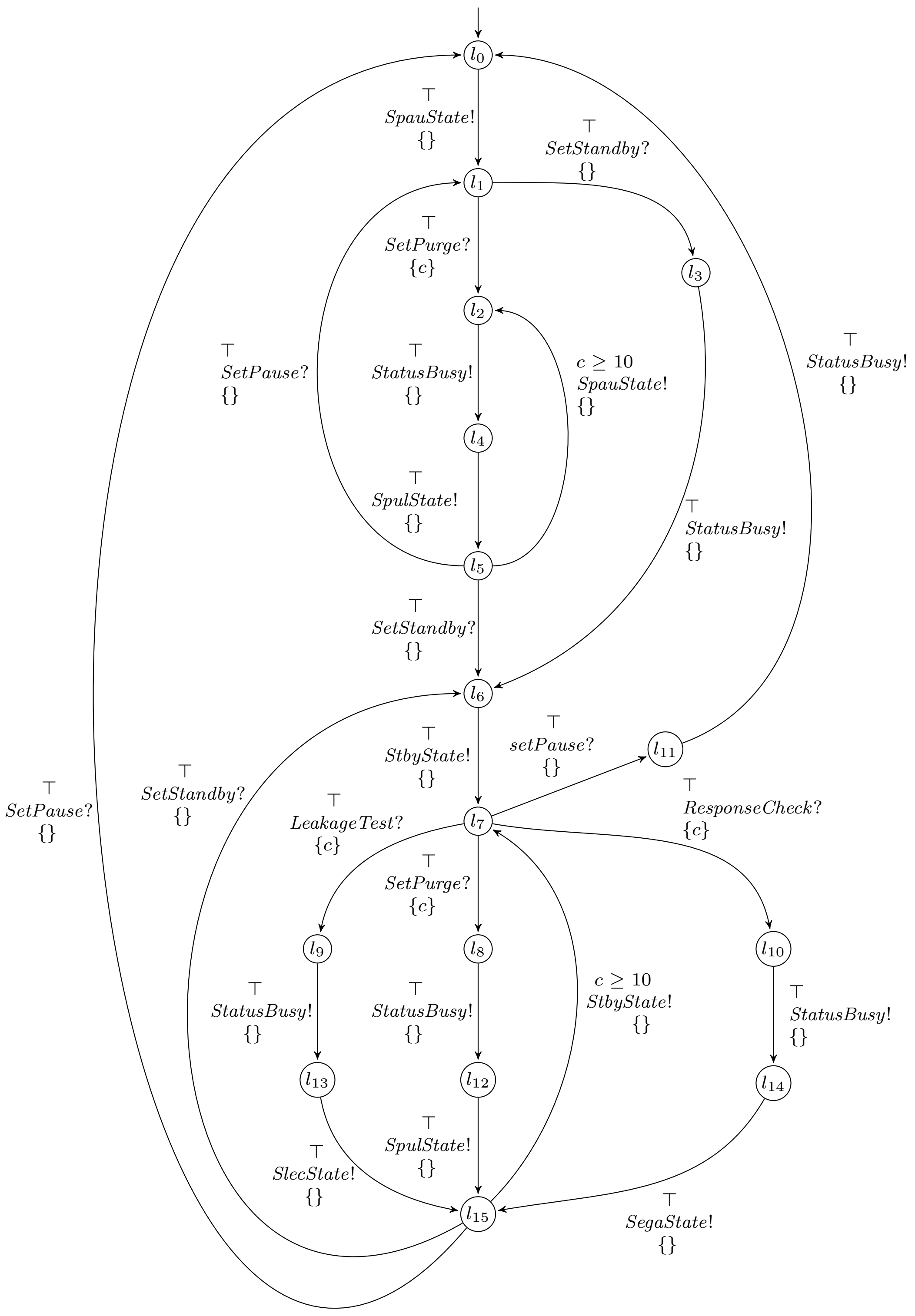

l_0
l_1
l_2
l_3
l_4
l_5
l_6
l_7
l_8
l_9
l_{10}
l_{11}
l_{12}
l_{13}
l_{14}
l_{15}
$\top$ $SpauState!$ $\{\}$
$\top$ $SetStandby?$ $\{\}$
$\top$ $SetPurge?$ $\{c\}$
$\top$ $SetPause?$ $\{\}$
$\top$ $StatusBusy!$ $\{\}$
$c \geq 10$ $SpauState!$ $\{\}$
$\top$ $StatusBusy!$ $\{\}$
$\top$ $SpulState!$ $\{\}$
$\top$ $StatusBusy!$ $\{\}$
$\top$ $SetStandby?$ $\{\}$
$\top$ $StbyState!$ $\{\}$
$\top$ $setPause?$ $\{\}$
$\top$ $SetPause?$ $\{\}$
$\top$ $SetStandby?$ $\{\}$
$\top$ $LeakageTest?$ $\{c\}$
$\top$ $SetPurge?$ $\{c\}$
$\top$ $ResponseCheck?$ $\{c\}$
$\top$ $StatusBusy!$ $\{\}$
$\top$ $StatusBusy!$ $\{\}$
$c \geq 10$ $StbyState!$ $\{\}$
$\top$ $StatusBusy!$ $\{\}$
$\top$ $SlecState!$ $\{\}$
$\top$ $SpulState!$ $\{\}$
$\top$ $SegaState!$ $\{\}$

Figure 5.19: TA that models parts of the Particle Counter

The median results of the evaluation with hints are shown in Appendix C. Figure 5.20 depicts the correctness evaluation. At a training set size of 50 we see that the active approach achieves a lower median correctness than the passive approach. However, we also observe in Figure 5.21 that the needed training set size of the active approach is exactly 50, without achieving correctness, which we assume is an indicator that we need a larger training set size to correctly learn the TA. Beginning at a training set size of 100, the median correctness of the active approach is 100%, whereas the passive approach

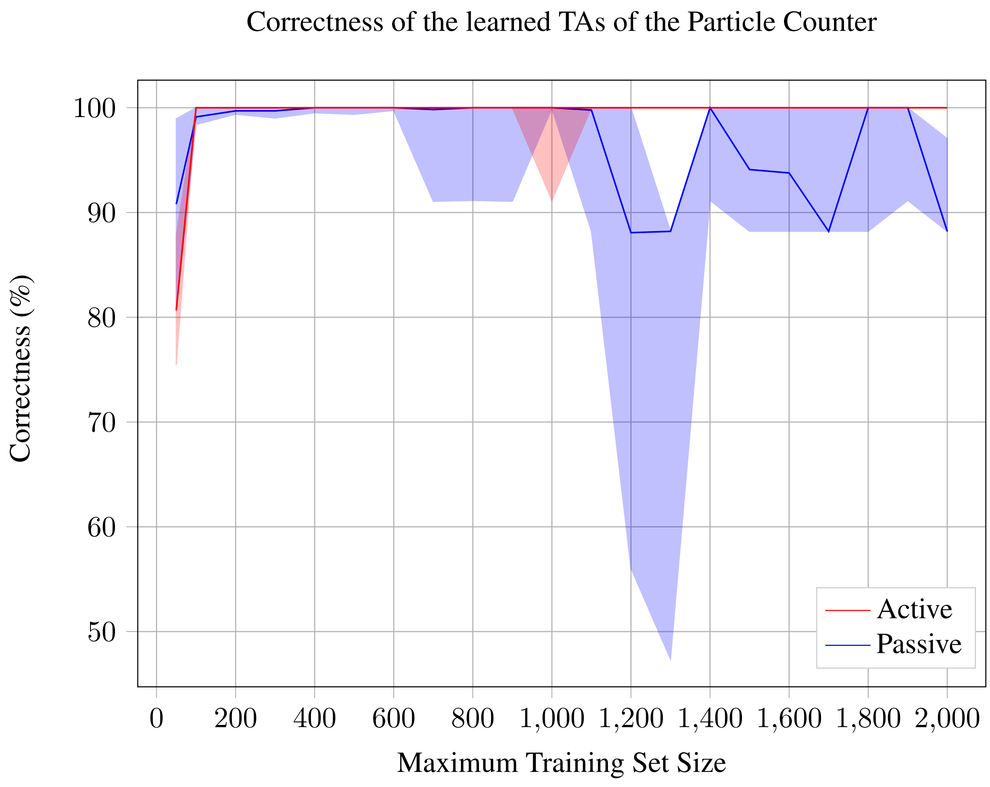


Figure 5.20: Correctness of the learned Particle Counter using hints

fluctuates between 88.1% and 100%. At a training set size of $1\,200$ the median correctness of the passive approach reaches a low with 88.1%. Furthermore, the values of the first quartiles show that both the active and the passive approach cannot always learn the correct TA. The active approach has a low in the first quartile at a training set size of $1\,000$, where the correctness is 91.2%. The lowest point in the first quartile of the passive approach is at the training set size of $1\,300$, where only 47.4% of the test set traces pass. Additionally, the third quartile of the passive approach for the same training set size is only 88.2%. We assume that these problems occur due to the previously mentioned complexity of the Particle Counter.

Our assumption that the Particle Counter is harder to learn can be observed in the measurements of the needed training set size and the test execution time. Compared to the other industry relevant examples we need larger training sets and, therefore, also the test execution times are higher. The needed training set sizes are depicted in Figure 5.21 and the execution times in Figure 5.22. Even though the results are higher than in the first two examples, we still can improve the median needed training set size of the Particle Counter by up to 122.1% and the execution time by up to 92.0%.

Although the improvements of the needed training set size and the execution time are smaller than those of the other industry examples, the improvement of the required runtime to learn the particle counter is as large as in the previous examples. Figure 5.23 depicts the runtime of the passive and the active approach. The improvement of the runtime is at least 139.8% and increases up to 2001.5%.

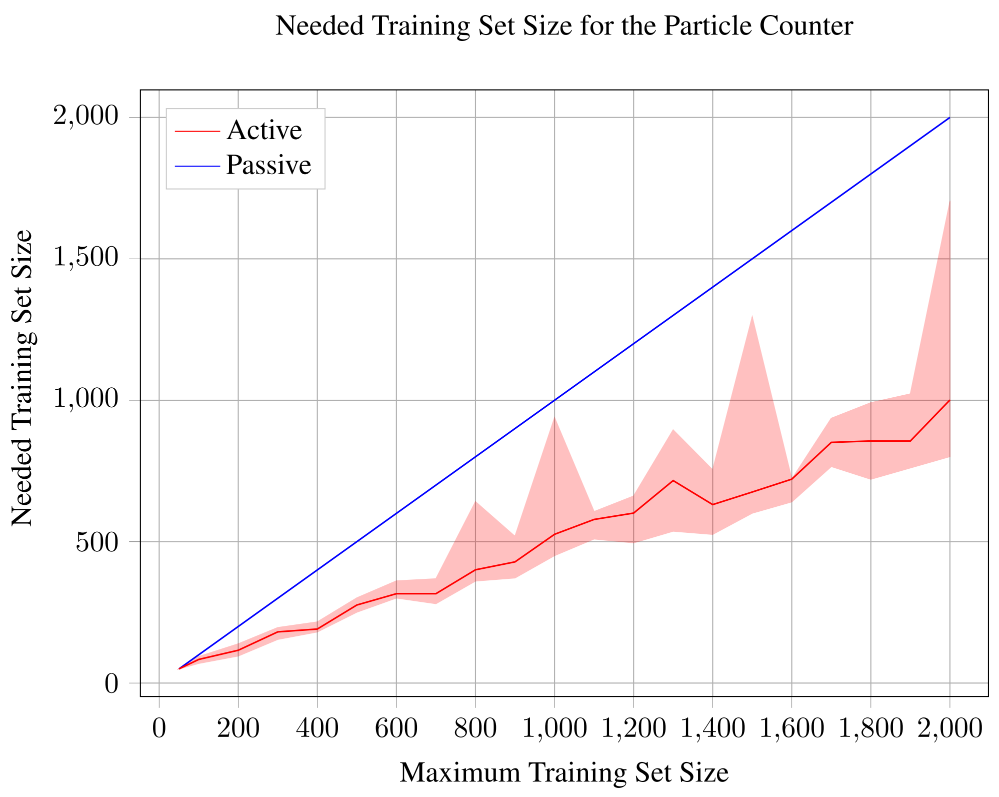


Figure 5.21: Needed training set size of the Particle Counter using hints

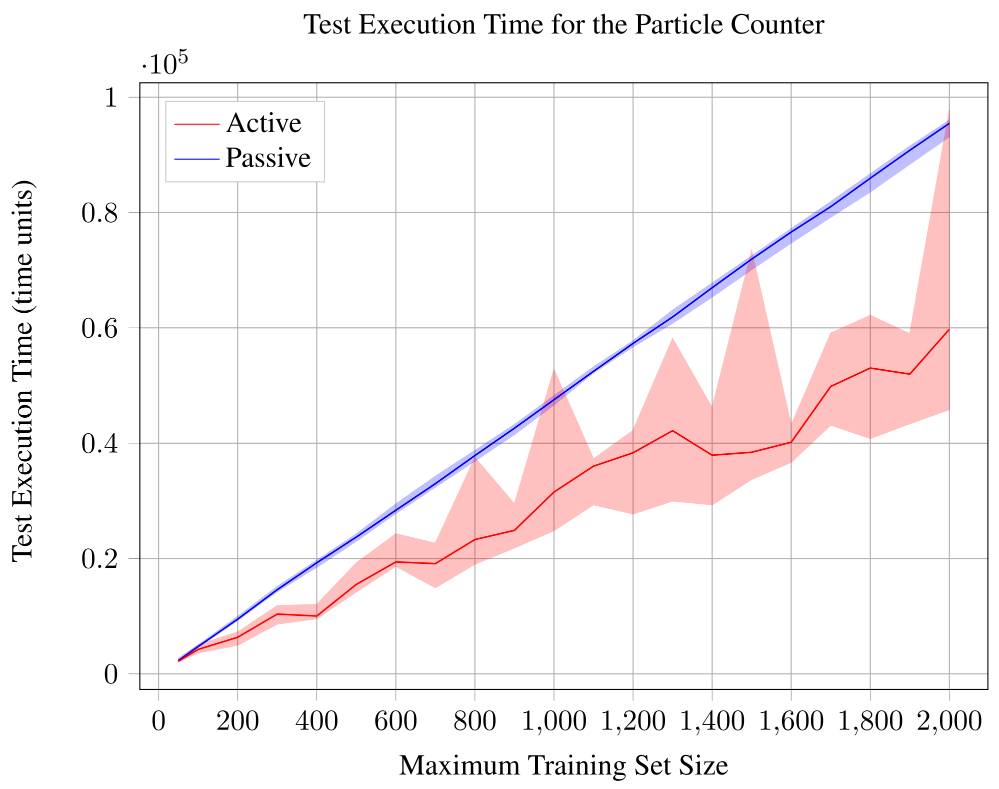


Figure 5.22: Execution time of training set for the Particle Counter using hints

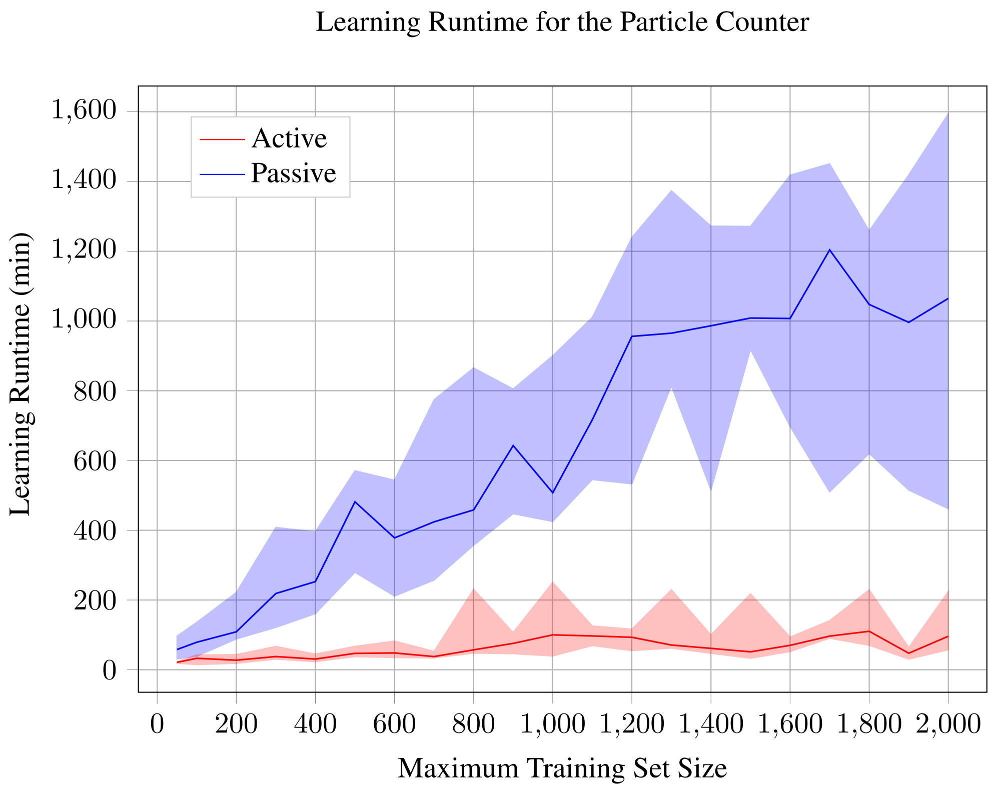


Figure 5.23: Learning runtime of the Particle Counter without using hints

5.5 Random Timed Automata

This section presents the evaluation based on 40 randomly generated TAs. These randomly generated TAs are similarly created as the random TAs used in the evaluation of Tappler et al. [47]. We divide these random TAs into four categories which are based on the number of locations and the number of clocks. Each category consists of ten different TAs. The results of each category are combined in one evaluation. The statistics shown in the plots are computed from the median values of the individual TAs.

The configuration of the parameters is equal to the previous examples from the industry. We performed every evaluation with hints and set the largest constant $c_{\max} = 20$ for every randomly generated TA.

5.5.1 Random Timed Automata with 10 Locations

In this section we present the evaluation based on random TAs that contain 10 locations and one clock. We provide the median results for each evaluation criterion and the corresponding improvement for the ten evaluated automata in Appendix C.

Figure 5.24 shows that the median correctness of the active approach is 100% for all training set sizes. The passive approach is below 100% median correctness for the training set sizes 50, 100 and 200. For all other training set sizes the passive approach can learn the TAs correctly.

Assessing the needed training set size in Figure 5.25 we see that the active approach requires at the maximum the whole training set size. This is an indicator that at least one TA of these ten randomly generated automata is more complex to learn than the others. This assumption is based on the fact that the active approach always tries to find counterexamples and thus increases the training set size in each iteration until the maximum number of training timed traces is reached. The larger the needed training set is the harder it is to learn a TA. However, for the median and even for the third quartile, the active

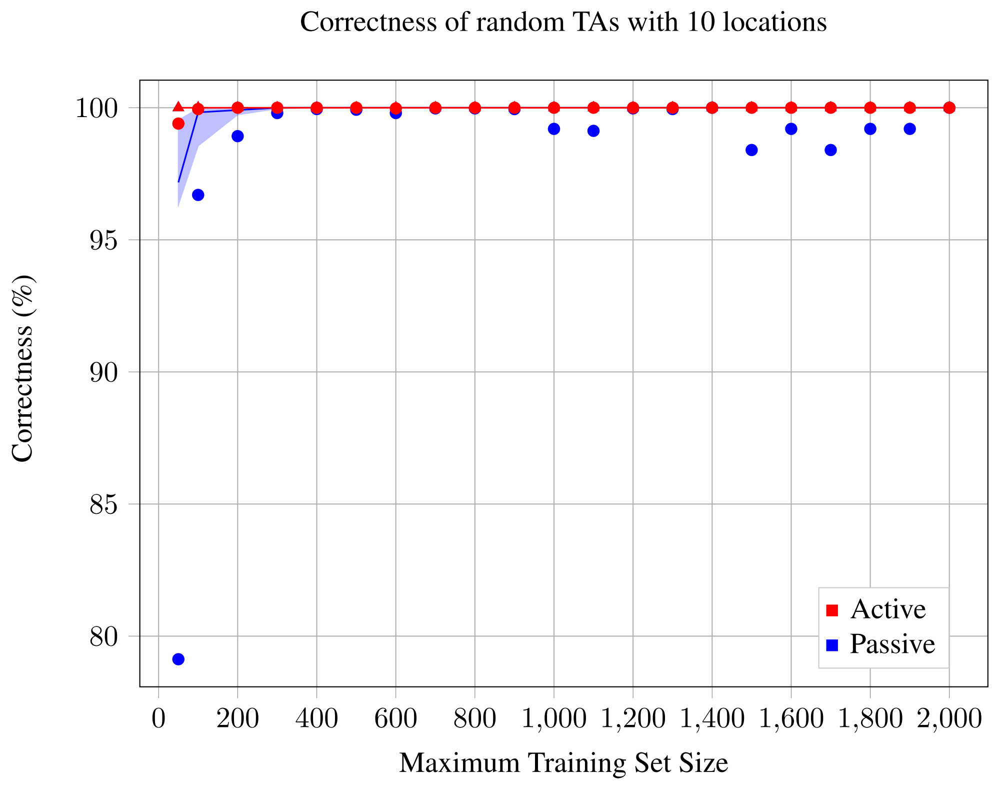


Figure 5.24: Correctness of the random TAs with 10 locations without using hints

approach is clearly below the maximum training set size. We can improve the median training set size by up to 298.9%.

Since the active approach requires at maximum the entire available training set size and the timed traces of the active approach are longer than the passive traces, the maximum execution time of the active approach is also at the maximum higher than the passive approach. Figure 5.26 depicts the execution times of the active and the passive approach. Except for the maximum value, the execution time of the active approach is shorter than the passive approach. In addition, for all training set sizes larger than 100 the minimum test execution time of the passive approach is higher than the third quartile of the active execution time. Comparing the median execution times of both approaches, the active approach improves the execution time by at least 32.2% and up to 283.1%. To provide a better visualization of the distribution of the execution times, we depict the time values for four training set sizes as box plots (see Figure 5.27). In Figure 5.27 we see that the active approach has two outliers, which are always higher than the third quartile of the passive approach. All other execution times are shorter than the passive approach. Furthermore, the minimum execution times of the passive approach, which are outliers, are also higher than most of the execution times of the active approach.

The learning runtime in Figure 5.28 reveals again that the active approach has difficulties with the inference of at least one TA. Furthermore, the passive approach is faster for the first three training sets. However, considering all training set sizes, the median runtime can be improved by 378.7% when using the active approach.

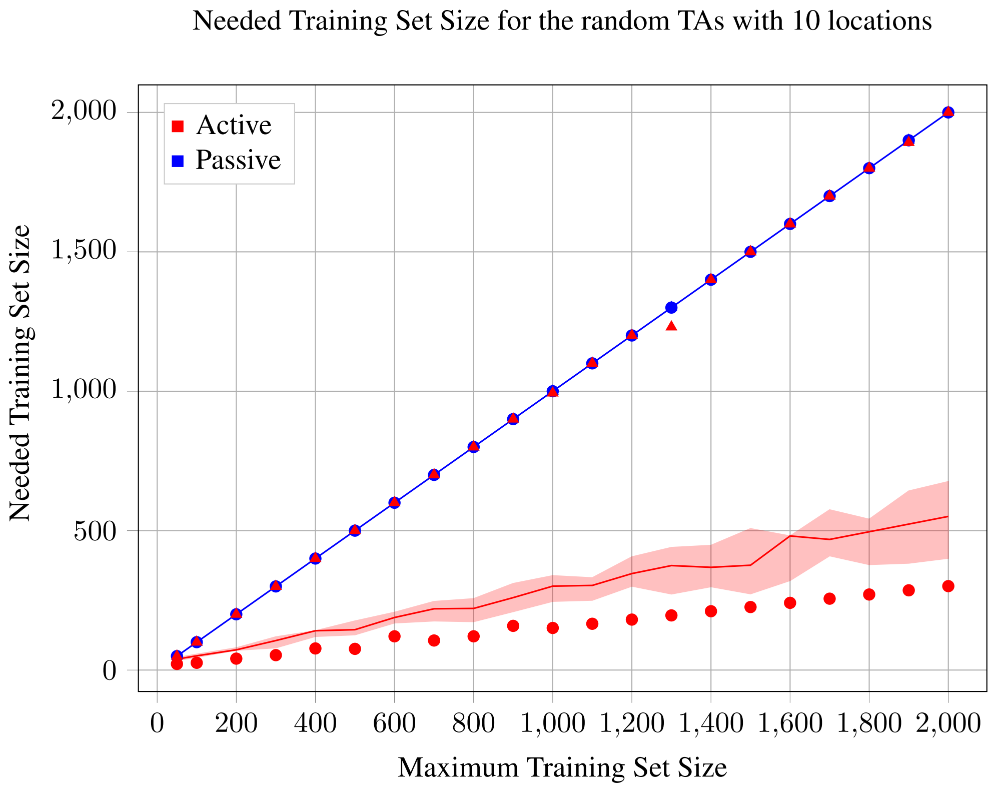


Figure 5.25: Needed training set size for the random TAs with 10 locations using hints

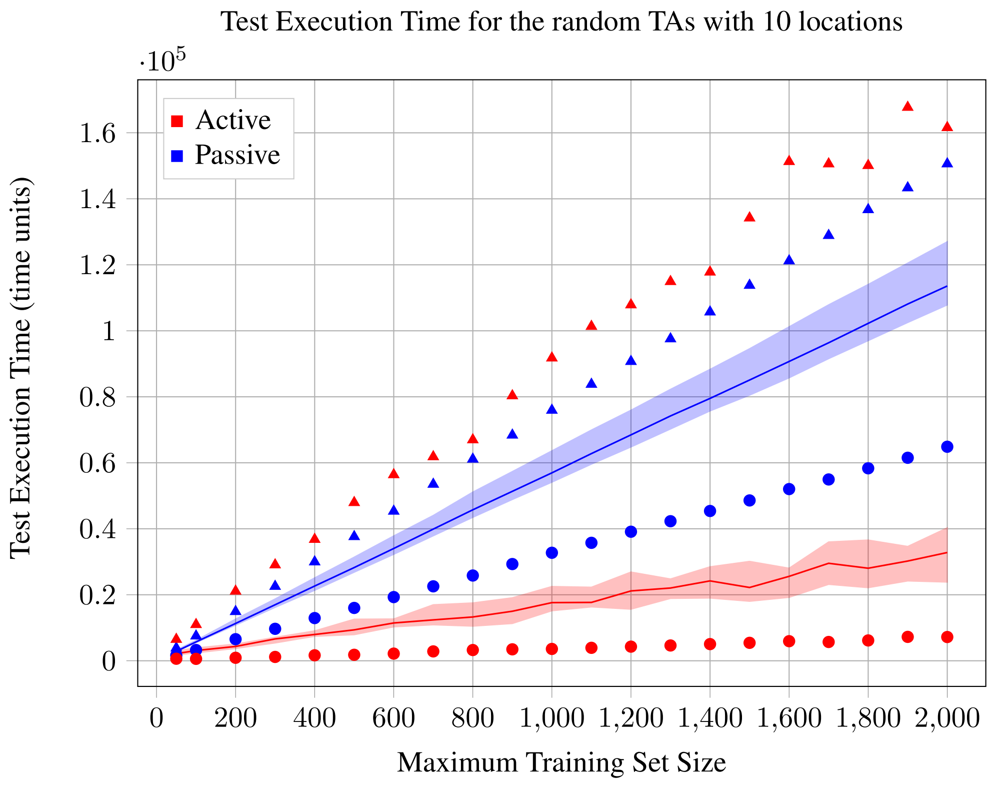


Figure 5.26: Execution time of training sets for the random TA with 10 locations using hints

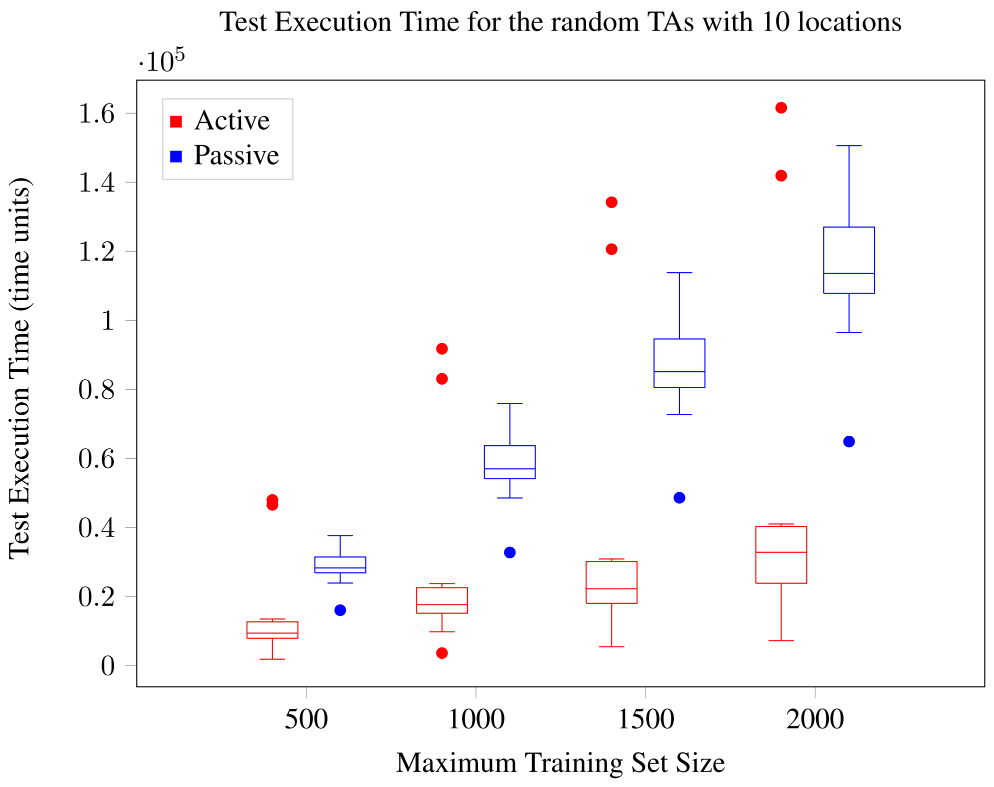


Figure 5.27: Execution time of training set, which is used to learn the TA of the random TAs with 10 locations using hints

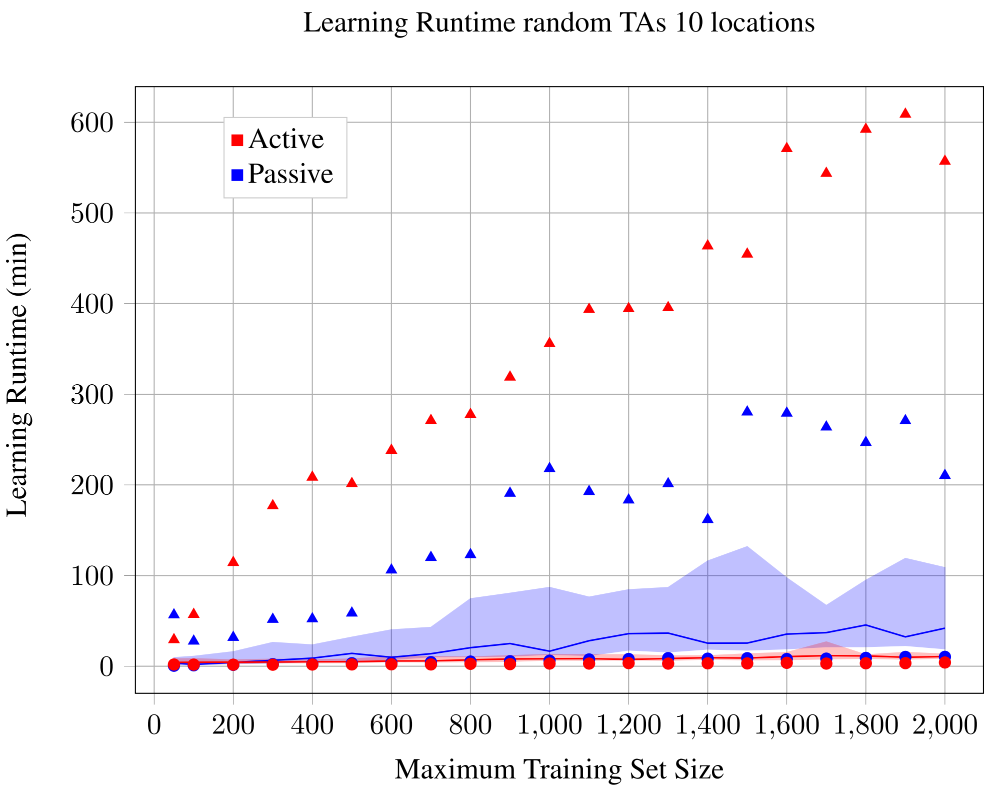


Figure 5.28: Learning runtime of random TAs with 10 locations using hints

5.5.2 Random Timed Automata with 15 Locations

Our next category of random TAs has 15 locations and one clock. In this category of random timed systems we increase the complexity by adding more locations to model the timed systems.

We observe this increase in complexity already in the correctness evaluation depicted in Figure 5.29. The passive approach never reaches a 100% median correctness. In addition, the passive approach has a constant minimum at approximately 94% for training set sizes larger than 100. However, the median correctness of the active approach is only below 100% for the maximum training set size of 50. For all other training set sizes the median correctness is 100%.

Figure 5.30 depicts the needed training set sizes. As in the evaluation of random TAs with 10 locations, the active approach requires, in the worst case, for almost all training set sizes the maximum set size. Learning a correct TA, requires a median training set size of 100. For larger training set sizes we can decrease the set size up to 179.1% compared to the passive approach.

The evaluation of the test execution time in Figure 5.31 shows that the maximum test execution time of the active approach fluctuates and for most training set sizes it is higher than the maximum of the passive approach. Overall, the median execution times of the active and the passive approach are closer to each other than in the previous examples. Additionally, we observe that the test execution times of the active approach have a larger variation than before and also have outliers for larger training set sizes (see Figure 5.32). For smaller training set sizes (50, 100 and 200) the median execution time of the passive approach is smaller. The median execution time of the active approach is better starting at a training set size of 300. We can improve the median test execution time by up to 81.2%.

Figure 5.33 shows the learning runtime for the random TAs with 15 locations. In contrast to the evaluation of the TAs with 10 locations, the maximum runtime of the active approach is not only below the maximum runtime of the passive approach, but also close to the runtime of the first quartile of the passive approach for most training set sizes. In addition, the median runtime of the active approach is below the passive approach for all training set sizes, except 100, where the passive approach is only 1.08 times faster than the active approach. In summary, we can improve the median learning runtime with the active approach by up to 1119.5%.

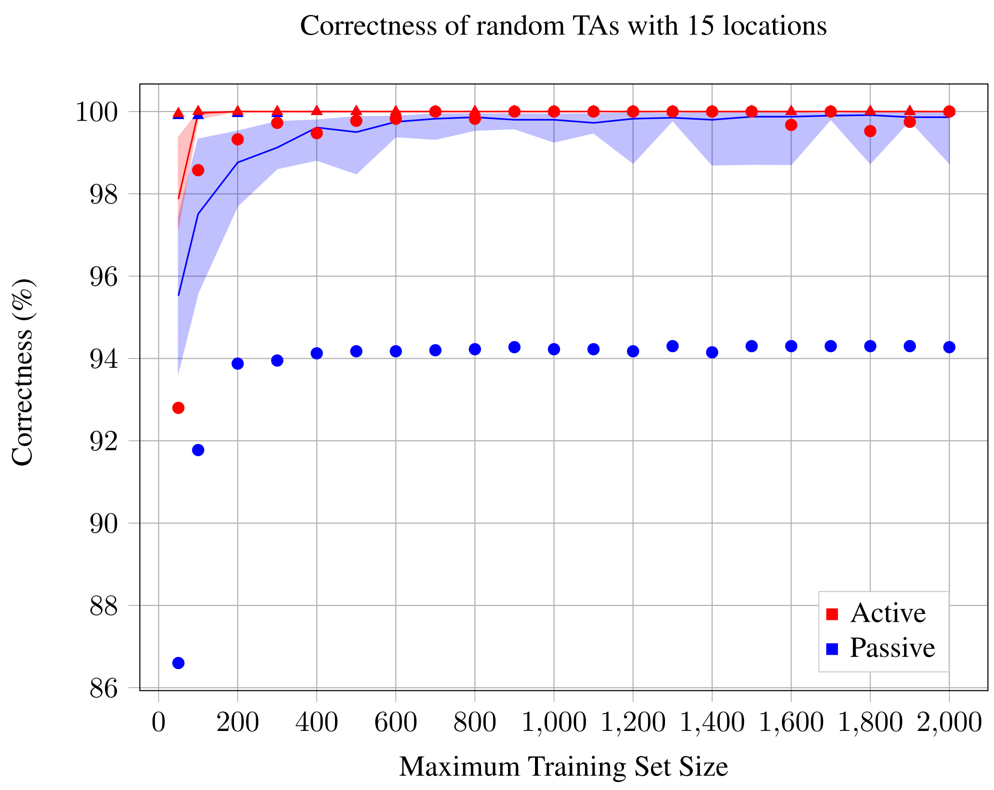


Figure 5.29: Correctness of the random TAs with 15 locations using hints

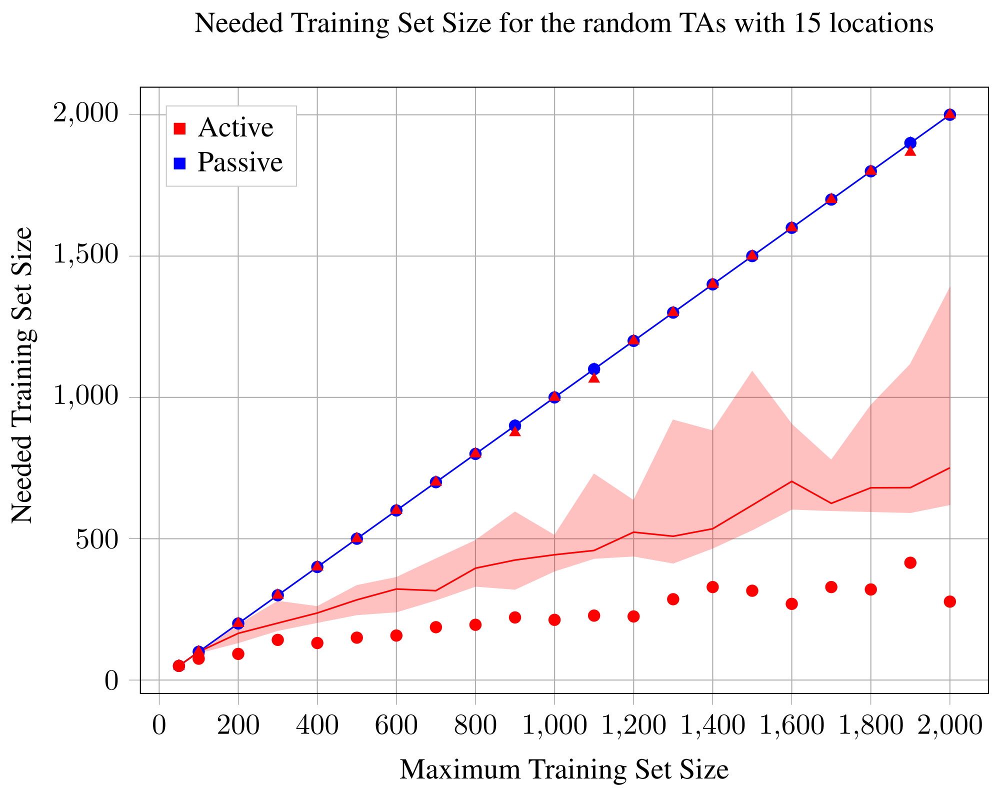


Figure 5.30: Needed training set sizes for the random TAs with 15 locations using hints

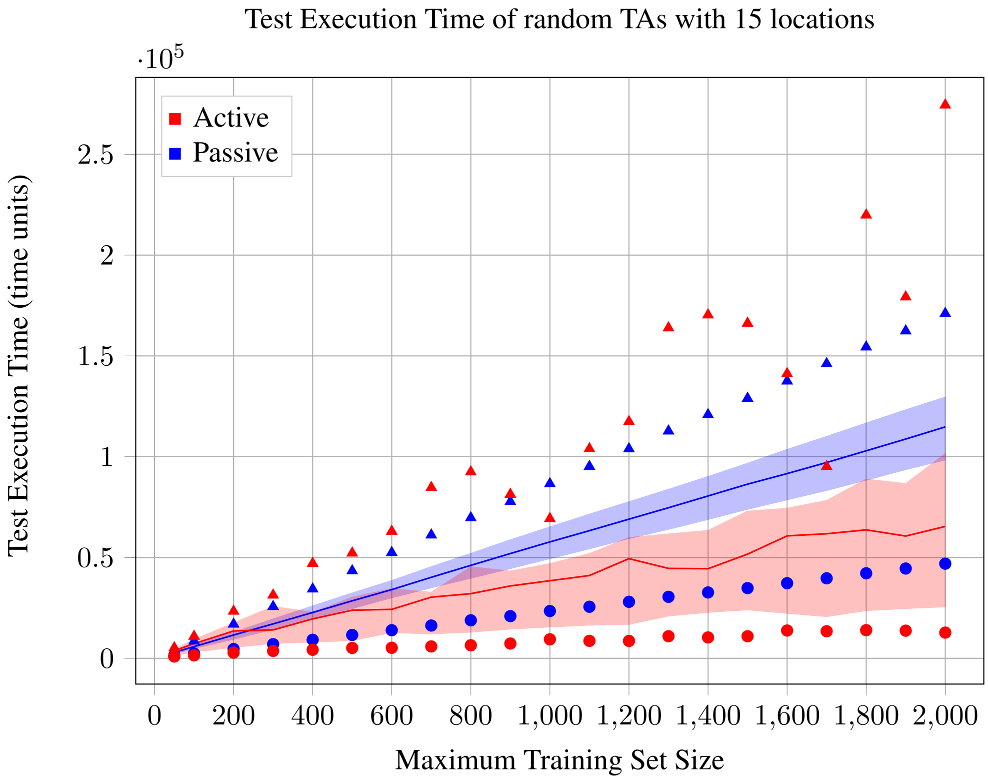


Figure 5.31: Execution time of training sets for the random TAs with 15 locations using hints

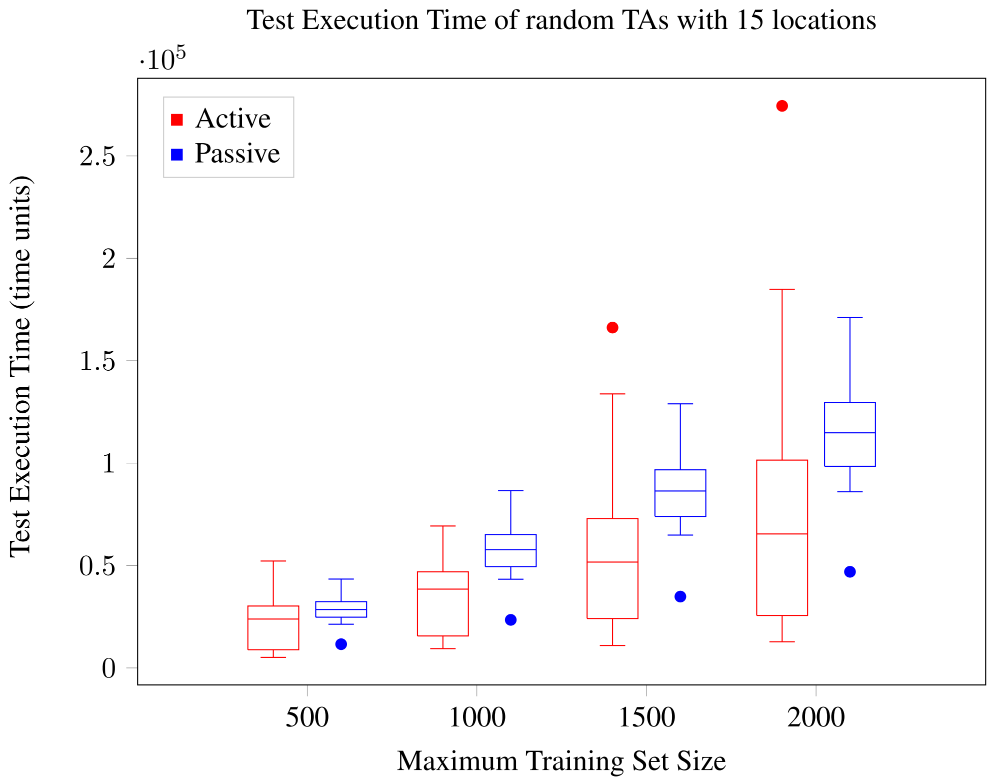


Figure 5.32: Execution time of training set for the random TAs with 15 locations using hints

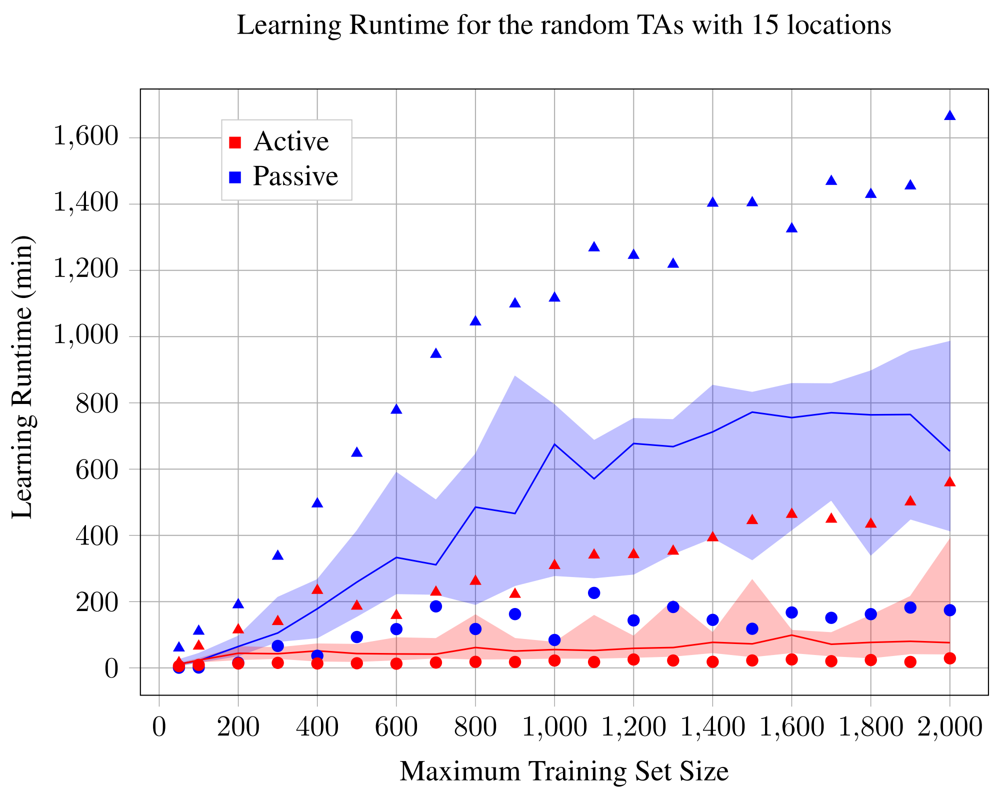


Figure 5.33: Learning runtime of random TAs with 15 locations using hints

5.5.3 Random Timed Automata with 10 Locations and 2 Clocks

The random TAs in this evaluation have again 10 locations, but this time we add a second clock to the system. All median results of this case study can be found in Appendix C.

The correctness measurement in Figure 5.34 shows that the minimum correctness of the active approach is below 100% for certain training set sizes. The median correctness of the active approach is for the smallest training set size 99.4% and for all other training set sizes 100%. The passive approach again achieves a lower correctness. The minimum correctness of the passive approach is for most set sizes only slightly higher than 85%. Furthermore, the passive approach reaches 100% median correctness only four times.

In Figure 5.35 we observe that the active approach requires for some training set sizes the entire available training set size. For the maximum training set sizes 50 and 100 the training set size is equal to the maximal available set size. However, the median needed set size is always below 1 000. Overall we can improve the needed set size by up to 139.4%.

For the test execution time, which is depicted in Figure 5.36, we see an akin picture as in the evaluation of random TAs with 15 locations. The maximum test execution times of both approaches are nearly at the same level. The active approach has a higher median test execution time for the first three set sizes. For all other set sizes the median execution time of the active approach is lower than the median execution time of the passive approach. We can improve the median execution time by up to 89.9%. Furthermore, Figure 5.37 depicts that the passive approach has two outliers with a higher execution time than the other random TAs of this category. The results of the active approach show a larger variation than in the other examples. The median execution test time of the active approach is indeed lower than that of the passive approach, but we also see that more examples require a higher execution time.

Figure 5.38 shows the required runtime to learn a random TA with 10 locations and 2 clocks. We observe that the median learning runtime of the active approach is higher than that of the passive approach for the training set sizes where we require the maximum training set size. As a result, the median runtimes

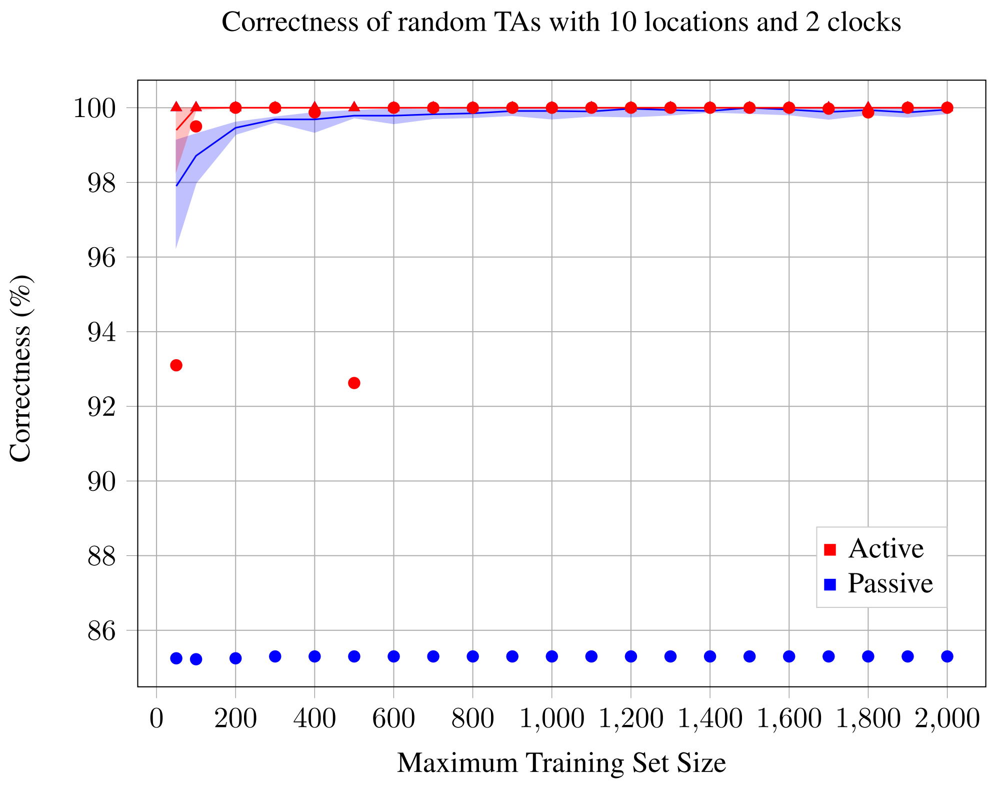


Figure 5.34: Correctness of the learned TAs of the random TAs with 10 locations and 2 clocks without using hints

for the training set sizes 50 and 100 of the active approach are higher than the runtimes of the passive approach. For all other training set sizes the median runtime of the active approach is below that of the passive approach and we can improve the median runtime by up to 691.9%. In addition, we observe that the maximum runtime of the active approach is, for almost all training set sizes, between the first and third quartile of the passive approach.

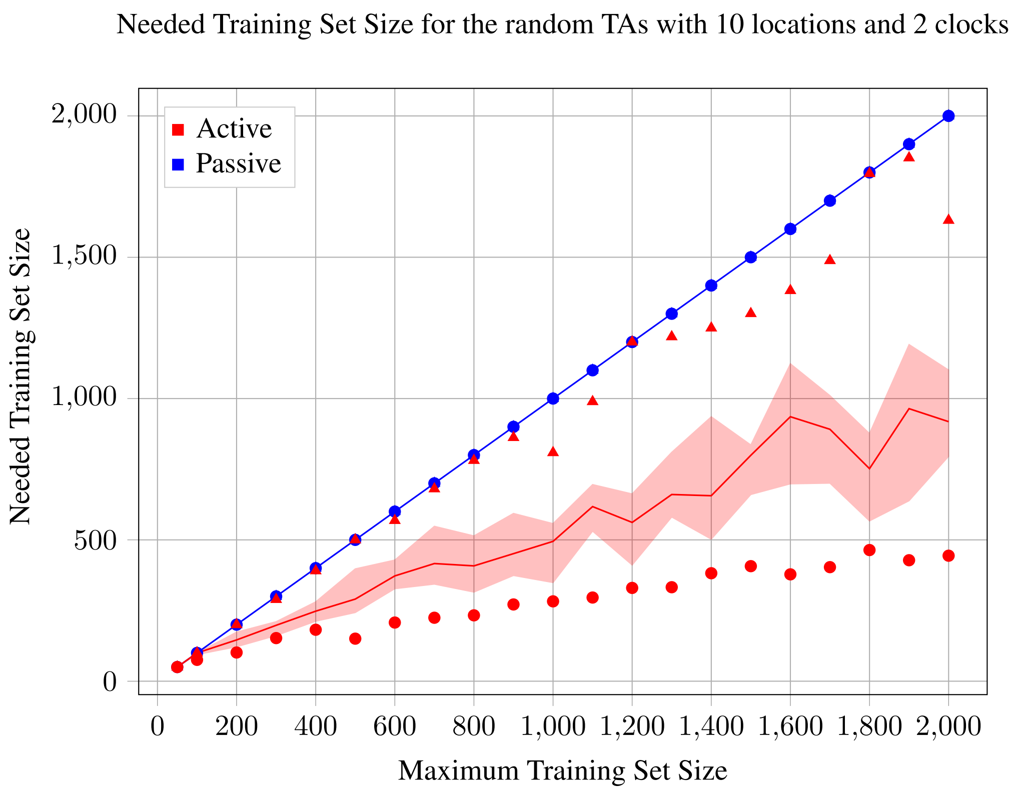


Figure 5.35: Needed training set sizes for the random TAs with 10 locations and 2 clocks using hints

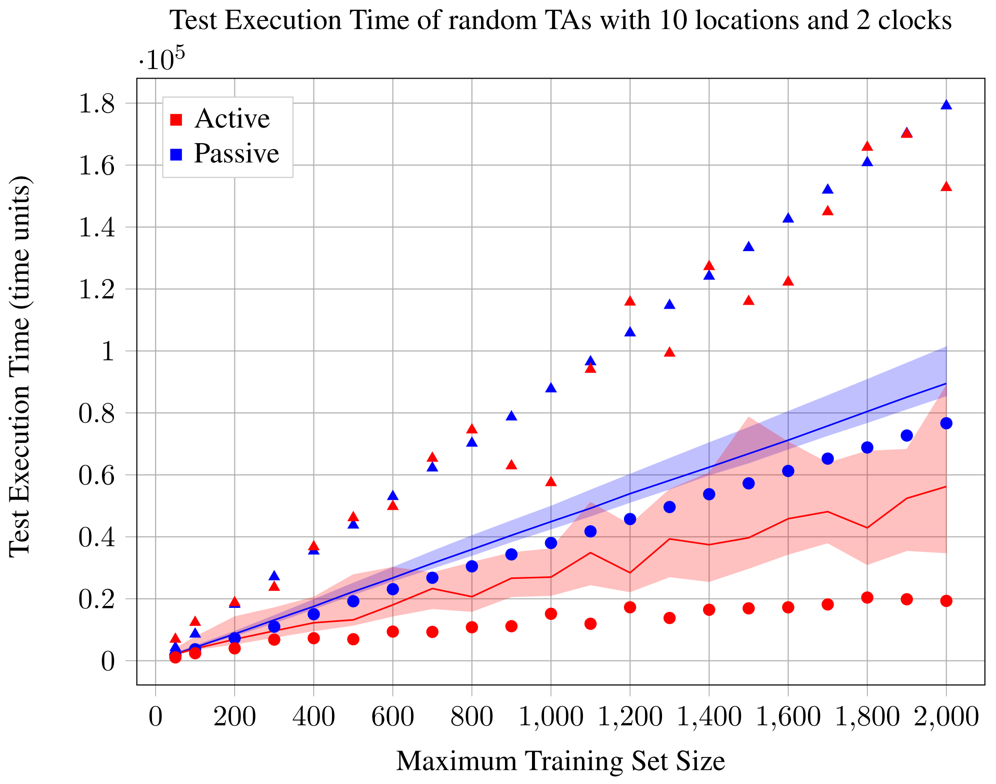


Figure 5.36: Execution time of the training sets for the random TAs with 10 locations and 2 clocks without using hints

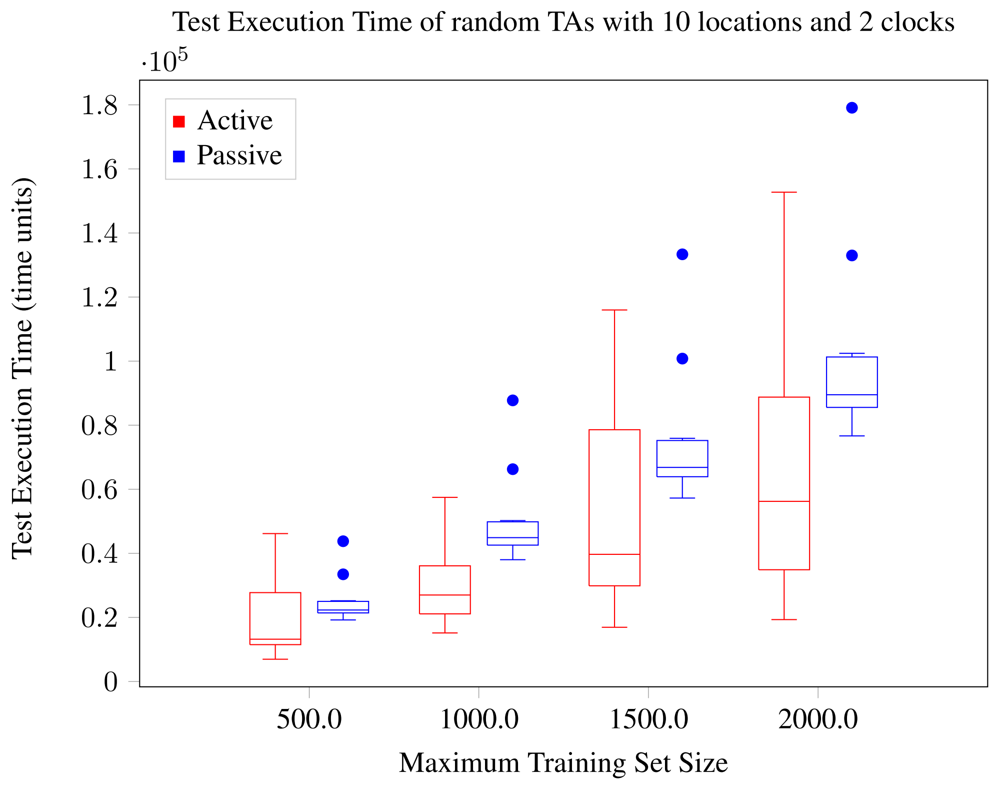


Figure 5.37: Execution time of training sets, which is used to learn the TA of the random TAs with 10 locations and 2 clocks using hints

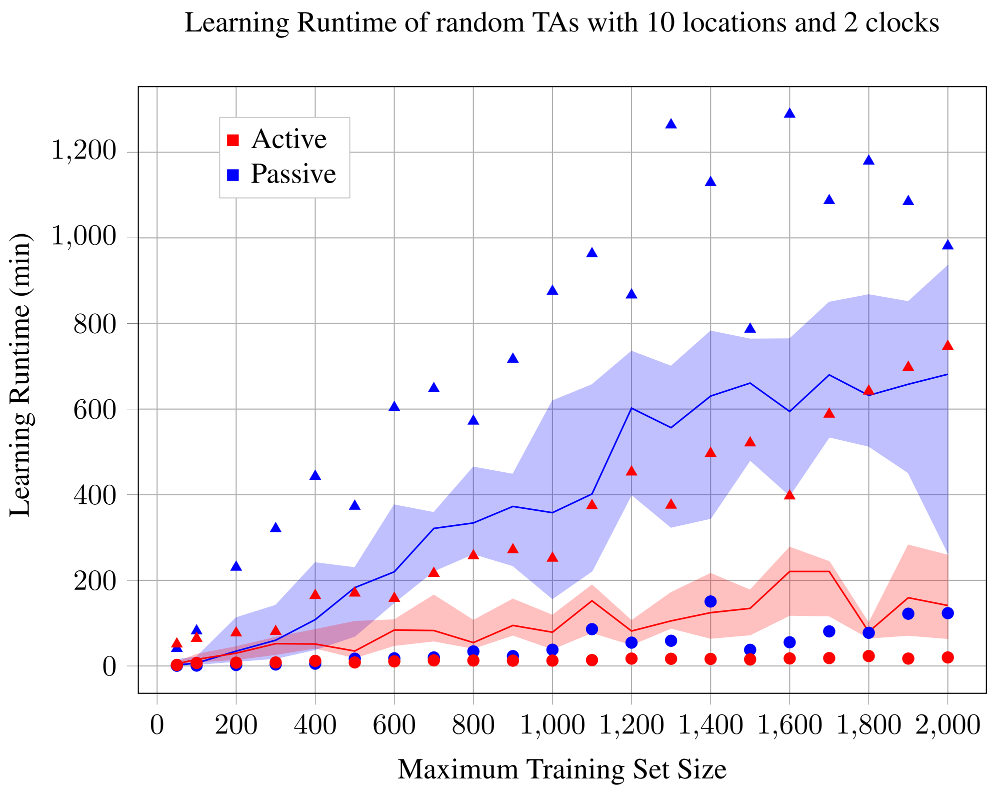


Figure 5.38: Learning runtime of random TAs with 10 locations and 2 clocks using hints

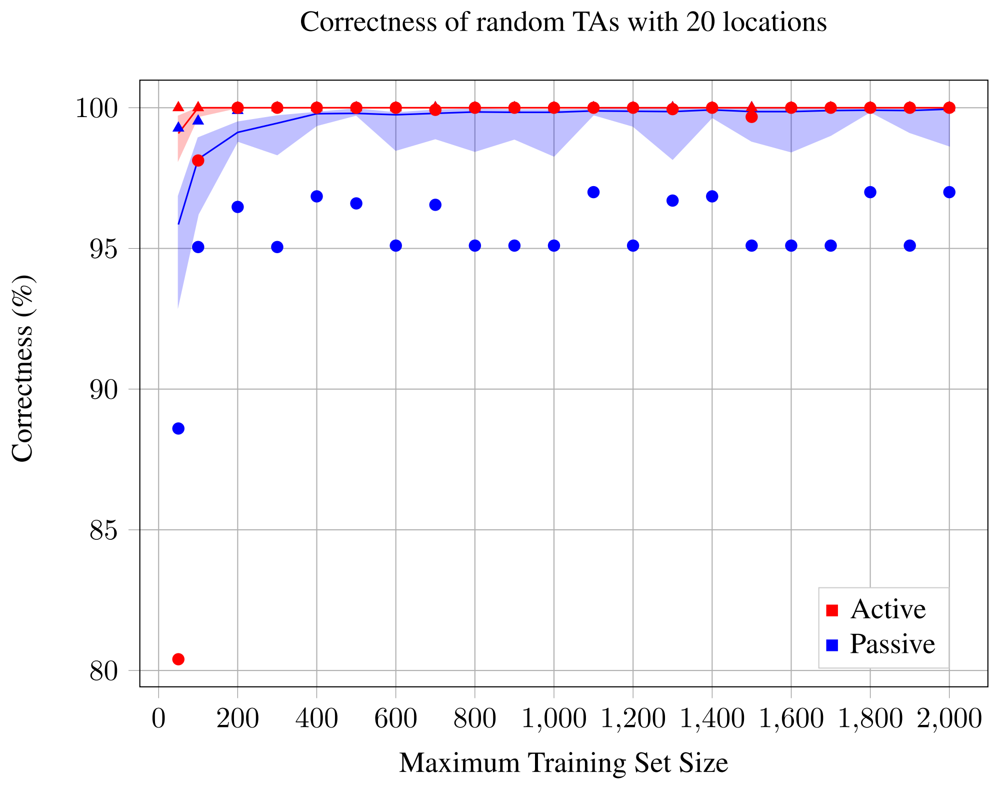


Figure 5.39: Correctness of the learned TAs of the random TAs with 20 locations using hints

5.5.4 Random Timed Automata with 20 Locations

Our last category of random TAs has 20 locations and one clock. We state all median results of the active and the passive approach with the corresponding improvement in Appendix C. Figure 5.39 shows that the active approach can correctly also learn timed systems with 20 locations. Starting from a training set size of at least 100 the active approach achieves median correctness of 100%. The passive approach reaches a median correctness of 100% with a training set size of $2\,000$ only. As a result, we achieve an improvement of the median correctness of up to 3.3% with the active approach.

Figure 5.40 shows the needed training set sizes. Comparing the needed training set sizes with the previously discussed case studies, we observe that we need larger training set sizes to learn a TA with more locations. Regarding the median results, we require for the maximum training set sizes 50 and 100 the entire available number of traces. For larger maximum training set sizes we can improve the median needed training set size by up to 77.1%.

In Figure 5.41 we observe that also the median test execution times of the active and the passive approach are closer together than in the previous examples. In addition, we see that the maximum test execution time of the active approach is significantly higher than the maximum execution time of the passive approach. However, Figure 5.42 shows that this maximum execution time is an outlier. Furthermore, the active results in Figure 5.42 have a lower variation than those in the evaluation of the random TAs with 10 locations and 2 clocks. The median execution time of the active approach is for all training set sizes below the median execution time of the passive approach. The improvement of the median test execution time is at least 10.2% and at the maximum at 48.1%.

Figure 5.43 depicts the results of the runtime evaluation. We see that the passive approach has a significantly higher maximum runtime than the active approach. The maximum runtime of the active approach is approximately in the range between the first and third quartile of the passive approach. Additionally, most of the active maximum runtimes are below the median runtime of the passive approach. In summary, we can improve the median runtime by at least 73.5% and up to 309.9%.

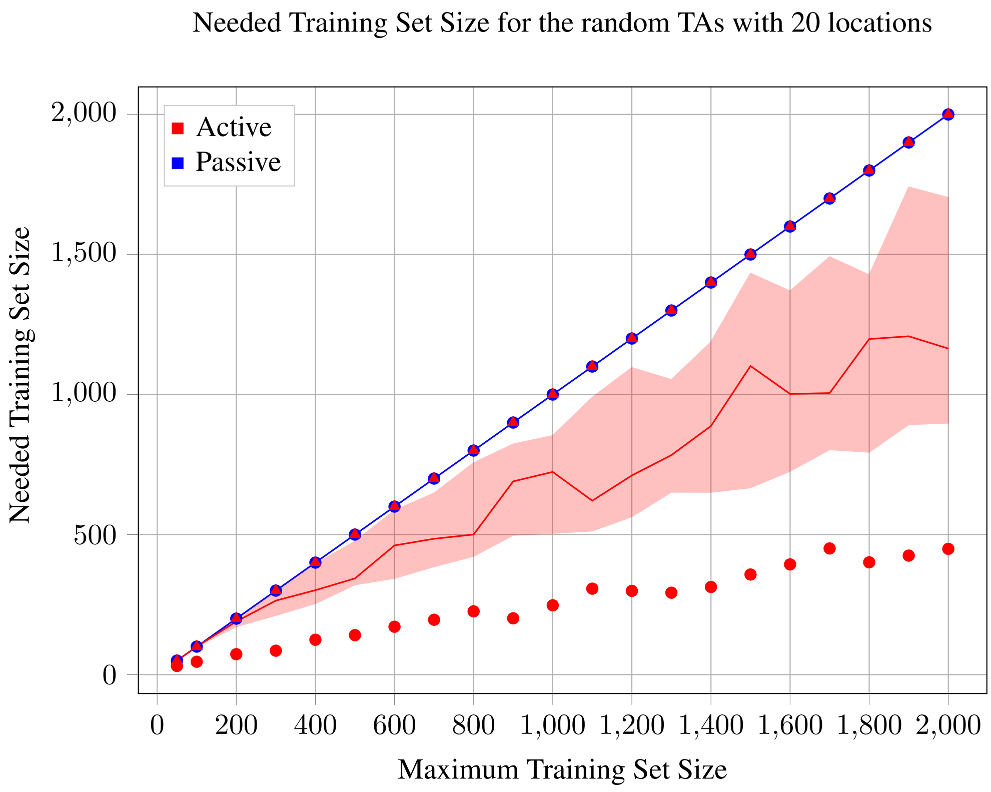


Figure 5.40: Needed training set sizes for the random TAs with 20 locations using hints

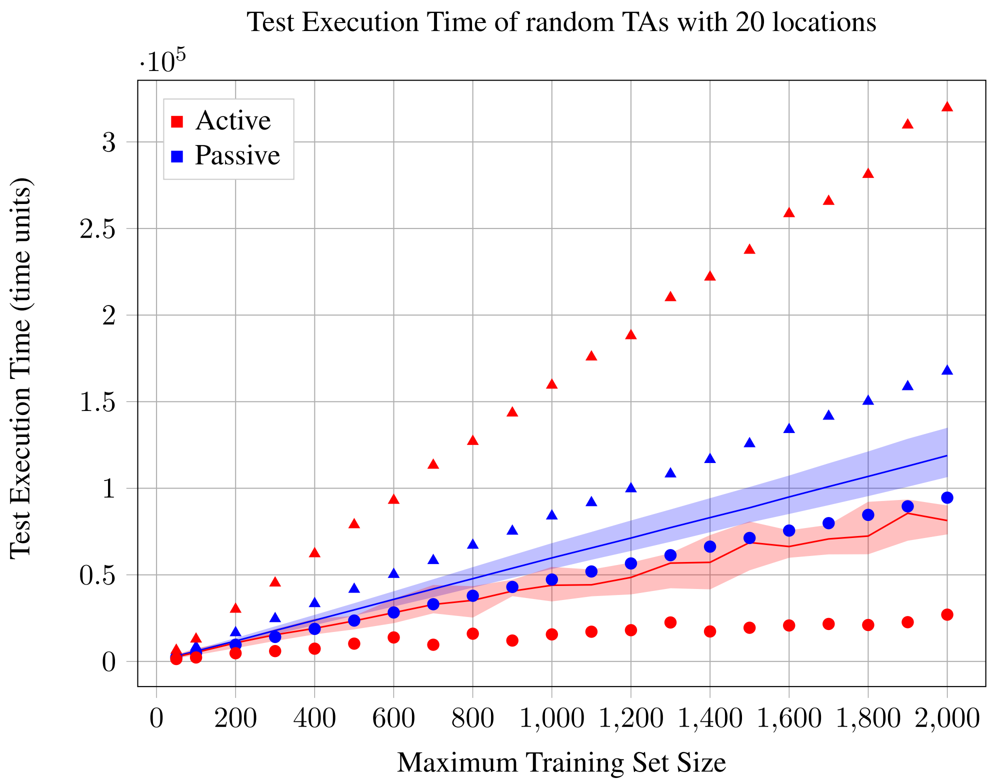


Figure 5.41: Execution time of training sets, which are used to learn the TA of the random TAs with 20 locations using hints

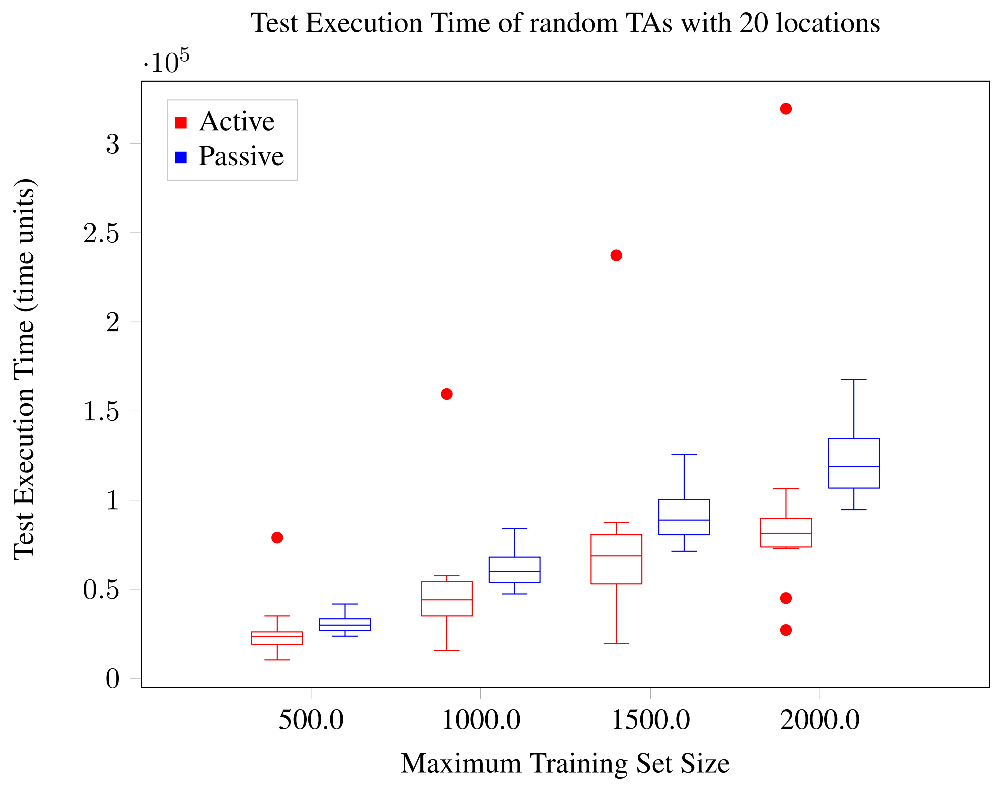


Figure 5.42: Execution time of training sets, which are used to learn the TA of the random TAs with 20 locations using hints

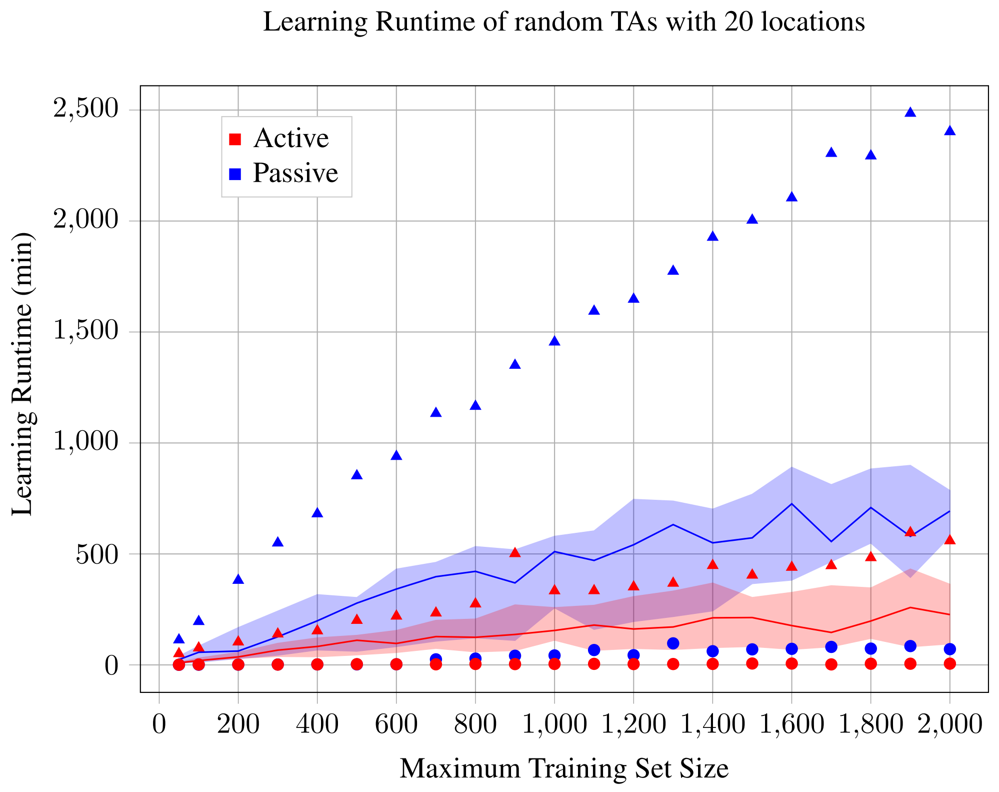


Figure 5.43: Learning runtime of random TAs with 20 locations using hints

6 Concluding Remarks

6.1 Related Work

In learning of software systems, we distinguish between *active* and *passive* learning approaches, according to De la Higuera [16]. Passive approaches learn automata models based on a given set of data, like monitored traces in log files. Active approaches, however, actively retrieve the data from the system under learning by asking queries. According to Hsu et al. [26] querying a system is not always possible due to the complex information that is required for input queries. In addition, they mention that the inquiry of an oracle, like it is done in Angluins' L^* [9], often performs inefficiently. To overcome these problems, Hsu et al. [26] prefers a passive approach based on monitored traces to detect security flaws in network protocols.

According to Aichernig et al. [4] one solution for passive learning are state-merging techniques. State-merging techniques generate from a given set of traces a prefix tree acceptor. In state-merging, we iteratively search for compatible nodes in the generated tree, which then are merged. For this procedure we use a given test set. Depending on the specific state-merging approach, this set of tests can contain tests that should be accepted by the learned system, but also tests that should be rejected. The tree is transformed to an automaton via this process of state merging. In their survey Aichernig et al. [4] remark the RPNI [42] and ALERGIA algorithm [13] as prominent state-merging approaches.

Verwer et al. [50] proposed a passive learning algorithm based on state-merging for timed systems. Using a given set of timed traces, their algorithm uses state-merging and state-splitting to learn TAs. Their merging strategy is the *red-blue fringe state-merging* approach, where they use colored nodes to mark locations that can be merged. Additionally, they can split transitions by splitting the time frame where the transition is enabled. All used time values in their framework are natural numbers. For the representation of their timed systems they use *real-time automata*. Real-time automata only use one clock which they reset in every transition. Furthermore, the guards of the transitions always have a lower and an upper bound. Since they assume that all considered time values are integers and the guards define a lower and upper bound, the number of possible transition-splitting operations is finite. Furthermore, the used automata are only labeled with one type of action. Therefore, inputs and outputs are not treated differently. They evaluate their passive learning approach using randomly generated real-time automata.

Verwer et al. [51] extend their state-merging approach by an approach that learns probabilistic deterministic real-time automata. First, they define probabilistic deterministic real-time automata. Therefore, they extend deterministic real-time automata with probability distributions. One of these extensions is that performing an action in a location has a probability. Additionally, observable time values have a probability distribution. This distribution of observable time intervals is represented by a histogram. To learn these probabilistic deterministic real-time automata, they propose a state-merging approach with a likelihood-ratio test. They evaluated their approach with automata consisting of up to eight locations and four actions, and restrict their time values to an upper bound of 100. Their evaluation shows that their approach has problems to learn the guards and some transitions of the probabilistic deterministic real-time automaton correctly. Improvements of this learning algorithm for probabilistic deterministic real-time automata are proposed by Mediouni et al. [39]. Using these improvements their approach achieves a higher correctness of the learned timed systems at the expense of learning runtime. Based on the ALERGIA algorithm [13], Mao et al. [38] developed an approach to learn deterministic probabilistic automata. Equal to the approach of Verwer et al. [51] they learn timed systems. In their work they learn labeled Markov decision processes, which contain input and output actions, and also continuous-time labeled Markov chains, which only contain outputs.

The results of passive learning approaches heavily depend on the given set of traces. Active approaches overcome this problem by actively retrieving the required information on demand. In Section 2.2 we discussed the active learning algorithm L^* proposed by Angluin [9]. In the field of learning and testing Angluin's L* was the base for several further works in this area of research.

Based on Angluin's L*, Grinchtein et al. [20] developed an algorithm to learn *deterministic event-recording automata*. Event-recording automata have for every action an own clock, which states the elapsed time since the corresponding action has been observed. Each time an action is performed the corresponding clock is reset. The learned automaton is represented by a multi-dimensional region graph, where the number of dimensions is equal to the number of clocks. This representation can lead to a state space explosion and the representation may be doubly exponential larger than the minimal automaton.

To overcome the complexity of learning region graphs, Grinchtein et al. [21] proposed a new approach for the inference of timed systems. In their new approach they use timed decision trees which are then transformed to event-recording automata. They denote a timed decision tree as a set of traces that are merged to a tree according to the prefixes of the traces. The leafs of the tree indicate if the path that leads to the leaf should pass or fail, i.e. be accepted or rejected. However, their proposed algorithm has high complexity. Nevertheless, they note that their algorithm can be modified to learn timed systems with one clock with a polynomial number of required queries. In any case, for timed systems with more than one clock the complexity of the algorithm increases significantly.

Jonsson and Vaandrager [29] also propose an active learning approach to learn timed systems. They also learn timed systems based on Angluin's L* approach. However, in contrast to the previously discussed learning approaches, they represent the learned timed systems with Mealy Machines with timers. To learn these Mealy Machines with timers, they translate timed sequences to untimed sequences. Furthermore, they define an equivalence relation between timed and untimed systems. They use two MAT frameworks, which interact with each other based on the equivalence relations between timed and untimed systems. They conclude that the required number of queries is polynomial in the number of states of the learned Mealy Machine with timers. However, regarding the number of timers of the original Mealy Machine the learned Mealy Machine is exponentially bigger.

Groz et al. [22] propose an active learning approach that learns non-resettable Mealy Machines. The proposed algorithm only requires the set of possible input actions. Furthermore, Groz et al. [22] stress that their approach can learn efficiently systems with hundred states.

L* requires an oracle that can check the equivalence between the hypothesis and the learned system. In practice, the availability of such an oracle is in many cases not achievable. Consequently, Peled et al. [43] introduced an algorithm to check the conformance between a learned automaton and a black-box system. To perform this conformance check, they used the conformances testing strategy by Vasilevskii and Chow [49, 14]. In literature this testing strategy is denoted as *W-Method*. With the W-Method we can check if the hypothesis is equal to the learned system if the hypothesis automaton and the system have the same input alphabet and an upper bound on the number of states of the system is known. Considering the assumption that the system is a black-box, the maximum number of states cannot always be assumed to be known. In addition, the complexity of the W-Method is exponential regarding the number of states of the learned system. Furthermore, Fujiwara et al. [19] suggest an optimized W-Method, which is denoted as *partial W-Method* or *WP-Method*. However, the W-Method as well as the WP-Method have in the worst case exponential complexity.

Since the W-Method is computationally expensive, active learning approaches use other methods. Aichernig and Tappler [5] show that they can learn Mealy Machines successfully by using a randomized test generation combined with mutation and transition coverage. For the testing of real-time systems, Schmaltz and Tretmans [45] introduced two conformance relations for timed systems based on the **ioco** relation proposed by Tretmans [48]. Hessel et al. [24] present an approach to perform conformance testing for timed systems using UPPAAL. In their appraoch they generate *time-optimal* tests. Time-optimal tests take the least possible time to execute. Springintveld et al. [46] adapt the W-method to generate a test suite for real-time systems.

Different theories exist in the literature regarding meta-heuristic learning approaches. To automatically solve problems, one example for meta-heuristic learning approaches are evolutionary algorithms. Such evolutionary algorithms include Genetic Algorithms (GAs) and Genetic Programming (GP). Lucas and Reynolds [37] compared in their work an evolutionary algorithm with a state-merging approach. Both approaches learn a deterministic finite automaton (DFA). Their evaluation is based on a given training set. Additionally, the number of states is fixed by an upper bound.

Lai et al. [32] proposes an active learning approach for deterministic finite automata (DFAs) using GAs. Their approach iteratively improves the learned automaton. In each iteration a counterexample is added which reveals that the learned automaton is incorrect. Their algorithm uses the W-Method for the conformance test between learned automaton and system. To apply their approach, the set of actions and the number of locations must be known.

Another field of application for learning approaches is software synthesis. According to Lefticaru et al. [36] the interest on meta-heuristic approaches for model-checking increases. In their work, they present a model-checking approach based on meta-heuristic approaches to synthesize software. Johnson [28] proposed an approach which uses evolutionary strategies. Evolutionary strategies require a fitness function to assess the quality of the learned solutions. Johnson's learning approach uses model checking to measure the fitness values. this model-checking approach is then used for software synthesis. Based on Johnson's approach Lefticaru et al. [36] develop a learning approach that uses an improved fitness function that is also based on model checking. For the application of the learning approach of Lefticaru et al. [36] the number of states must be known. Katz and Peled [30] suggest a GP approach to synthesize code from a formal specification. They suggest that their approach cannot only be used to develop program code, but also to find and correct errors in applications, and for code improvements.

6.2 Summary

In this thesis we presented an active learning algorithm which learns timed systems via GP. Tappler et al. [47] discuss a passive learning approach which also uses GP to learn timed systems. The aim of this work was to optimize the passive learning approach by an active learning extension. For this, the set of tests, which the GP process uses to learn the timed system, is iteratively improved.

Our active learning approach learns timed systems which are represented by a TA. Alur and Dill [7] present a definition of TAs: TAs are finite state machines that are equipped with real-valued variables, which store time values. Thus, these real-valued variables represent clocks. To make the learning of TAs accessible, we use to the assumptions for TAs presented by Hessel et al. [24] and Tappler et al. [47]. Thus, we use TAs that are deterministic, input enabled and have isolated outputs.

In each iteration of the active learning approach we perform a conformance check. Hence, our active learning approach requires a conformance relation between the SUT and the TA. Based on the assumption that the SUT can be represented by a TA, we define a conformance relation between the SUT and the hypothesis TA, based on trace equivalence. Trace equivalence is given if the compared systems behave equally for all traces. For timed systems we consider timed traces, which are sequences of actions where each action has a time stamp. However, since the number of possible traces is infinite, we limit our conformance check to a finite set of timed traces. Tretmans [48] stresses that the goal of model-based testing is to find a sound and exhaustive test suite. Therefore, we want to find a set of timed traces that is sound and as exhaustive as possible. Furthermore, to improve the performance of the GP procedure, we keep the set of required timed traces as small as possible.

The basic procedure of our active algorithm is based on the iterative refinement approach discussed by Walkinshaw et al. [53]. In their approach they refined the inferred hypothesis until no further counterexamples to the SUT could be found. Based on this idea, our active learning approach starts with an initial hypothesis. We then select timed traces that reveal that the hypothesis is wrong. Based on the found counterexamples, we use GP to model a new hypothesis. Afterwards, we check again if the hypothesis conforms to the SUT. If the conformance relation is satisfied, i.e. we do not find a counterexample, our approach outputs the learned hypothesis, otherwise the refinement of the hypothesis continues.

For the generation of timed traces we presented a random timed trace generator, that generates traces by randomly walking through the hypothesis. The aim of the random walk is to reflect the behavior of the hypothesis. To check certain properties of the hypothesis, we generate traces that invoke behavioral aspects of the hypothesis. For example, we explore new behavior by performing random inputs or checking constants of the learned guards. After the generation of the trace, our active approach executes the trace on the SUT and compares the trace of the SUT with the trace of the hypothesis. If the traces are different, we extend our set of timed traces with the found counterexample.

For our active learning approach we developed a Java application. Additionally, we offer a GUI which demonstrates the learning process of the active and the passive GP approach. This GUI offers a convenient way to backtrack the single steps of the active and the passive learning approach.

Finally, we performed seven case studies on our active approach. These case studies comprise altogether 18 900 learned TAs. In these case studies we compare the results of the active approach with those of the passive approach. For the evaluation, we used three manually created systems which are examples from the industry. We also evaluated another 40 randomly generated timed systems. We compare the active and the passive approach based on correctness, needed training set size, test execution time and learning runtime. The results of the case studies show that active learning of TAs via GP is a promising technique to automatically learn timed systems correctly and efficiently.

6.3 Conclusion

The research problem of this thesis was to develop an active learning approach for Timed Automata (TAs) via Genetic Programming (GP). The developed active approach is based on the passive GP approach presented by Tappler et al. [47]. The main idea of an active approach is to iteratively add tests on demand. These added tests should improve the learning process. One promising solution to turn the passive approach into an active approach was the iterative refinement approach proposed by Walkinshaw et al. [53]. To apply this approach, we required a method to generate tests. The challenge in this work was to find an appropriate test generation method which leads to a correct result and also performs efficiently. Timed systems have infinitely many states due to the time variables. Therefore, adding tests that cover all possible states is impossible. According to Aichernig and Tappler [5] one promising solution is the generation of tests by randomly walking through the automaton. We adapted this approach for randomly walking through TAs. Using this method, we showed that randomized approaches are able to generate an appropriate test set.

The goal of this thesis was to optimize the passive approach of Tappler et al. [47] with an active learning approach. To assess if we reached this goal with our active approach, we performed an evaluation based on 18 900 learned TAs. The learned TAs include three examples from the industry as well as 40 randomly generated timed systems. Our evaluation is based on timed systems with up to 20 locations and multiple clock variables. In this evaluation we compare the results of the active approach with the corresponding results of the passive approach. We quantified the optimization based on following four criteria: *Correctness*, *Training Set Size*, *Test Execution Time* and *Learning Runtime*. In all four criteria we see that the active approach achieves improvements.

We evaluated the correctness of the learned TAs to show that our approach is not only optimized regarding time, but also learns timed systems correctly. In our case studies we see that the active approach requires at most a median test set size of 100 to correctly learn an automaton. In contrast, the passive approach requires more traces and additionally has the problem that too many traces create local maxima. Local maxima lead to the problem that the learning approach does not perform further mutations, since the fitness decreases after the application of small mutations. The active approach does not have these problems since failing tests increase the need to perform further mutations. In addition, we see that the passive approach has problems to correctly learn the timed system if the user-provided maximum clock constant is selected imprecisely. The active approach does not depend on a precise largest constant and, therefore, can correctly learn the timed system with less information.

The active approach only adds as many tests as required to correctly learn the timed system. One goal of our approach is to find a test set that is sound and as complete as possible. Furthermore, to make the GP procedure efficient, we want to keep the test set as small as possible. Our evaluation shows that we can achieve a maximum improvement of the median test size by at least 77.1% and by up to 895.0%. This decrease of the test set size shows that the active approach requires fewer tests to learn a correct TA. Moreover, a smaller test set size also increases the performance of the learning approach.

Due to the fact that we can decrease the training set size, less time to execute the tests on the system is required. The maximum improvement of the median execution time is at least 48.1% and can be improved by up to 731.6%.

Our last optimization criterion is the learning runtime. This evaluation criterion achieves the largest improvements. The maximum improvement of the median learning runtime is at least 176.2% and increases up to 84953.7%. We observed this huge improvement due to the passive approach not being able to learn the automaton correctly with the given test set. Additionally, the active approach learns less complex systems relatively fast, thus showing this vast improvement. For example, we can learn the discussed Smart Light Switch with a median runtime of 1.8 minutes in the case of the active approach and require 1505.8 minutes in the case of the passive approach.

All evaluated experiments were performed with the same parameters, with the exception of some system specific settings like the largest constant and the hints about delays. This shows the success of our approach does not depend on the configuration of the parameters and is generically applicable. Furthermore, we evaluated our approach without the provision of hints about the delay and with less precise largest constants. Again, we were able to correctly learn the timed systems.

The performed evaluations show that our developed active approach achieves huge improvements compared to the passive approach. However, in some iterations we observed that the passive approach outperforms the active approach. Especially, the test execution time for smaller training set sizes is often lower than that of the active approach. We assume that the longer overall execution time is higher due to the higher execution time of the single tests in the active approach. However, in the case of larger maximum training set sizes the significant decrease of the necessary training sets leads to a decrease of the overall test execution time. In addition, in some experiments with smaller training set sizes the passive approach achieves a lower correctness than the active approach.

In summary, we can say that our proposed active learning approach for TAs via GP is a promising solution to learn timed systems. Our approach can learn timed system correctly and, compared to the passive approach proposed by Tappler et al. [47], more efficiently. The learning of systems is an increasing popular topic in the field of quality assurance. Our approach is open for extensions which may include, e.g., a more sophisticated test selection or a further improvement of the test execution time. Finally, we can say that we achieved our research goal by the improvement of the passive approach with an active approach based on heuristics.

6.4 Future Work

The results in Chapter 5 show that our presented active learning algorithm significantly improves learning of timed systems compared to a passive learning approach. We evaluated a wide variaty of different systems. However, further experiments could provide more insights. Evaluating more examples from the industry could demonstrate the applicability of the algorithm in practice. Additionally, experiments on systems with more locations or clock values would show the performance of the presented approach on even more complex systems.

In addition to learning other systems, we can evaluate the approach with different configurations. In Appendix B we list all possible parameters that can be configured. Since there are various different parameters, the number of possible configurations is large. In our case studies in Chapter 5 we use the same configuration for all experiments. This shows the general applicability of the algorithm, since no special configuration for each experiment is required to learn the SUT correctly. However, we assume that we would observe further improvements if the parameters are set individually for each experiment. Additionally, a further study could implement an approach to automatically learn the optimal parameters.

Besides a more detailed evaluation of the current learning approach, further work could include adaptions to the algorithm itself. Following adaptions of the algorithm may improve the active learning process:

Test Execution Time. In Chapter 5 we observe that the improvement of the test execution time is lower than the improvement of the needed training set size. The reason for this behavior is that an actively generated timed trace has a longer execution time than the randomly generated traces in the passive approach. In the active approach we select the delays in the timed traces either based on the values of the guards or randomly. We may decrease the overall execution time if we select the delays based on a more sophisticated approach. One possibility would be to generated *time-optimal* traces like in the conformance testing approach of Hessel et al. [24].

Timed Trace Selection. In our presented timed traces selection approach we always add the first found counterexamples to the set of traces. The only selection criterion we use is to check if the counterexample contains an output action. We may achieve a better performance of the algorithm if we select timed traces using advanced selection criteria. For example, we could select only the shortest traces from a pool of counterexamples. Shorter traces may be easier to learn, since they do not contain distracting contents that unnecessarily increase the size of the automaton. Furthermore, we could only add counterexamples that reveal the same error in the hypothesis. We assume that adding only traces which reveal the same error in the hypothesis is an especially promising technique to improve the performance of the learning algorithm. If we add several traces which reveal the same error, the error has more weight. Therefore, the fitness value increases significantly if an individual of the population learns to fix the found error.

Timed Trace Generation. The generation of timed traces in our approach is done via a random walk through a TA. During this random walk, we follow edges of the TA randomly. We may achieve more informative counterexamples if we cover all the behavioral aspects of the TA. Higher coverage can be achieved, e.g., by visiting every location in the hypothesis or by visiting all edges. Aichernig and Tappler [5] present a randomized test-case generation which combines random walks with the coverage of edges. In their approach, they may perform a random walk through the automaton, before trying to reach certain edges of a Mealy Machine. However, to implement such an approach for TAs we require a reachability check. Implementing such a reachability check for TAs is not trivial since the state space is infinite. Furthermore, complex implementations for coverage or reachability checks may be more expensive regarding run time than a simple random walk.

Another field of application for our active learning approach could be software synthesis. Katz and Peled [30] showed that GP can be used to automatically synthesize programs based on a user specification. This approach requires an oracle to verify the synthesized program. To verify the synthesized program, they used model checking. Instead of using model checking based on a meta-heuristic approach, we cloud extend their framework by our active learning approach for TAs, which would create a possibility to synthesize timed systems.

Further work could also be done to extend the developed GUI. Currently, the GUI provides a fixed number of examples. A possibility to use own timed system for experiments would improve the applicability of the GUI. Additionally, an interface to UPPAAL, which exports learned systems in the `XML` format would be practical. Such an export functionality would provide the possibility to use learned timed systems directly in UPPAAL.

Appendices

A Random Walk Algorithm

Algorithm 7 Timed Trace Generation

Input: Fittest Timed Automaton $\mathcal{H}$ denoted as triple $\langle L, l_0, E \rangle$ over $(\mathcal{C}, \Sigma)$, and p_{input}
Output: Timed Trace tt_{h}

```
 1: function RandomWalk
 2:     $tt_h \leftarrow \{\}$
 3:     $l \leftarrow l_0$
 4:     $t \leftarrow 0$
 5:     $\nu \leftarrow \mathbf{0}_{\mathcal{C}}$
 6:     while $|tt_h| < n_{\text{len}}$ do
 7:         $edge \leftarrow Nil$
 8:         $infiniteOutputs \leftarrow \bot$
 9:         $E_{\text{O}} \leftarrow \{e \mid \exists\, l', g, r : l \xrightarrow{g,o,r} l', o \in \Sigma_{\text{O}}\}$
10:         $E_I \leftarrow \{e \mid \exists\, l', g, r : l \xrightarrow{g,i,r} l', i \in \Sigma_{\text{I}}\}$
11:         $L_{\text{urgent}} \leftarrow \{\}$
12:         for all $(l, g, o, r, l') \in E_{\text{O}}$ do
13:             if $\nu \models g$ then
14:                 if $L_{\text{urgent}}(l) = 2$ then
15:                     $infiniteOutputs \leftarrow \top$
16:                     break
17:                 $tt_h \leftarrow tt_h \cdot t \cdot o$
18:                 $\nu \leftarrow \nu[r]$
19:                 $L_{\text{urgent}} \leftarrow L_{\text{urgent}} \uplus \{l\}$
20:                 $l \leftarrow l'$
21:                 $E_{\text{O}} \leftarrow \{e \mid \exists\, l', g, r : l \xrightarrow{g,o,r} l', o \in \Sigma_{\text{O}}\}$
22:                 $E_I \leftarrow \{e \mid \exists\, l', g, r : l \xrightarrow{g,i,r} l', i \in \Sigma_{\text{I}}\}$
23:         $delay \leftarrow$ getRandomDelay()
```

```
    **if** $(\neg probChoice(p_{\text{input}})$ **then**
        **if** $probChoice(p_{\text{trans}})$ **then**
            $\mathcal{K} \leftarrow \{k \mid \exists\, l', a, r : l \xrightarrow{g,a,r} l', g = c \oplus k\}$
            **if** $|\mathcal{K}| > 0$ **then**
                $d \leftarrow selectRandom(\mathcal{K})$
            $\nu \leftarrow \nu + d$
        **for all** $(l, g, o, r, l') \in E_O$ **do**
            **if** $\nu \models g$ **then**
                $d \leftarrow minDelay(g, \nu)$
                $edge \leftarrow (l, g, o, r, l')$
                **break**
        **if** $edge = Nil$ **then**
            $E_{\text{I}_{\text{sat}}} \leftarrow \{\}$
            **for all** $(l, g, i, r, l') \in E_{\text{I}}$ **do**
                **if** $\nu \models g$ **then**
                    $E_{I_{\text{sat}}} \leftarrow E_{I_{\text{sat}}} \cup \{(l, g, i, r, l')\}$
            $edge \leftarrow selectRandom(E_{\text{I}_{\text{sat}}})$
    **if** $edge = Nil \wedge \neg infiniteOutput$ **then**
        **do**
            $outputSatisfied \leftarrow \bot$
            $\nu \leftarrow \nu + d$
            **for all** $(l, g, o, r, l') \in E_O$ **do**
                **if** $\nu \models g$ **then**
                    $outputSatisfied \leftarrow \top$
                    **break**
            **if** $\neg outputSatisfied$ **then**
                $\Sigma_{\text{I}_{\text{apply}}} \leftarrow \Sigma_{\text{I}}$
                **for all** $(l, g, i, r, l') \in E_{\text{I}}$ **do**
                    **if** $\nu \models g$ **then**
                        $\Sigma_{\text{I}_{\text{apply}}} \leftarrow \Sigma_{\text{I}_{\text{apply}}} \setminus \{(l, g, i, r, l')\}$
                **if** $|\Sigma_{\text{I}_{\text{sat}}}| > 0$ **then**
                    $i \leftarrow selectRandom(\Sigma_{\text{I}_{\text{apply}}})$
                    $tt_h \leftarrow tt_h \cdot t \cdot i$
                    **break**
            $\nu \leftarrow \nu - d$
            $d \leftarrow getRandomDelay()$
        **while** $\neg probChoice(p_{\text{stop}})$
```

```
61:     else
62:         $\nu \leftarrow \nu[r]$
63:         $t \leftarrow t + d$
64:         $tt_h \leftarrow tt_h \cdot t \cdot a$ with $edge = l \xrightarrow{g,a,r} l'$
65:         $l \leftarrow l'$
66:         $E_O \leftarrow \{e \mid \exists\, l', g, r : l \xrightarrow{g,o,r} l', o \in \Sigma_O\}$
67:         $isolatedOutput \leftarrow \bot$
68:         for all $(l, g, o, r, l') \in E_O$ do
69:             if $\nu \models g$ then
70:                 $isolatedOutput \leftarrow \top$
71:                 break
72:         if $|tt_h| = n_{\mathrm{len}} \;\wedge\; n_{\mathrm{len}} < n_{\mathrm{len}_{\max}} \;\wedge\; isolatedOutput$ then
73:             $n_{\mathrm{len}} \leftarrow n_{\mathrm{len}} + 1$
74:         if $infiniteOutputs$ then
75:             $d \leftarrow getDelay()$
76:             $t \leftarrow t + d$
77:             $i \leftarrow selectRandom(\Sigma_I)$
78:             $tt_h \leftarrow tt_h \cdot t \cdot i$
79:             return $tt_h$
80: return $tt_h$
```

B Parameter Glossary

c_{max} User defined largest constant in a timed system

g_{change} Number of generations after which the GP procedure is stopped, since the overall fitness value does not change

g_{max} Maximum number of generations

$g_{\text{max}_{\text{active}}}$ Maximum number of generations per each iteration in the active approach

g_{simp} Number of generations after which locations that do not have an effect on the semantic of the TA are removed from the TA

$|L|$ Number of locations of each individual in the initial population

n_{attempts} Number of attempts to perform a random walk, which should reveal a counterexample

n_{clock} Number of clocks in the TA

n_{fail} Number of timed traces that are added in one iteration in the active approach

n_{len} Number of actions in a timed trace. The number of actions is also denoted as length of a timed trace $|tt|$

$n_{\text{len}_{\text{max}}}$ Maximum number of actions in a timed trace

n_{pop} Number of individuals in the population

n_{sel} Number of individuals that are selected to perform mutations on it

n_{start} Number of tests for the generation of the initial hypothesis in the active approach

n_{test} Number of tests

p_{input} Probability to perform a non-location-changing input action

p_{stop} Probability to stop a procedure

p_{trans} Probability to choose a value from the constants used in the guards, which label a set of edges

$\mathit{reduceInput}$ Constant by which p_{input} is decreased in each iteration of the active approach

$w_{\texttt{NONDET}}$ Weight in the fitness function for the verdict of the execution of a timed trace that is non-deterministic

$w_{\texttt{OUT}}$ Weight in the fitness function for the number of outputs in a timed trace

$w_{\texttt{PASS}}$ Weight in the fitness function for the verdict of the execution of a timed trace that passes

$w_{\texttt{SIZE}}$ Weight in the fitness function for the number of edges in the TA

$w_{\texttt{STEPS}}$ Weight in the fitness function for the number of performed non-deterministic steps during the execution of a timed trace

95

C Results

Table C.1: Medians of the Smart Lights Switch evaluation using hints

# Maximum Training Set	Correctness			Training Set		Test Execution Time			Learning Runtime		
	Passive (%)	Active (%)	Improvement (%)	# Needed	Improvement (%)	Passive	Active	Improvement (%)	Passive (min)	Active (min)	Improvement (%)
50	100.0	100.0	0.0	17.5	185.7	2976.3	1589.9	87.2	10.0	1.6	543.2
100	100.0	100.0	0.0	23.5	325.5	6059.2	1637.5	270.0	90.6	1.0	9013.0
200	72.1	100.0	27.9	36.0	455.6	12195.7	3215.1	279.3	344.1	0.9	37883.3
300	86.3	100.0	13.7	46.0	552.2	18359.3	4000.3	358.9	246.9	1.2	19737.7
400	72.7	100.0	27.3	61.0	555.7	24461.9	5406.0	352.5	665.1	1.5	44521.3
500	72.7	100.0	27.3	63.5	687.4	30980.1	6290.8	392.5	803.3	1.4	57685.5
600	100.0	100.0	0.0	91.0	559.3	37472.3	7431.3	404.2	13.1	1.5	748.7
700	72.7	100.0	27.3	106.0	560.4	43630.7	8573.9	408.9	1079.4	1.8	60699.8
800	72.7	100.0	27.3	121.0	561.2	49921.9	13911.0	258.9	1224.7	7.4	16487.0
900	72.7	100.0	27.3	91.0	889.0	55928.8	8617.0	549.1	1396.1	1.8	77114.0
1000	72.7	100.0	27.3	151.0	562.3	62143.6	13880.1	347.7	1360.1	4.0	33863.1
1100	72.7	100.0	27.3	138.5	694.2	68057.8	10626.3	540.5	1410.6	1.8	78849.3
1200	72.7	100.0	27.3	121.0	891.7	74581.0	8968.9	731.6	1505.8	1.8	84953.7
1300	72.7	100.0	27.3	196.0	563.3	80817.3	23923.5	237.8	1594.2	9.8	16097.5
1400	72.7	100.0	27.3	211.0	563.5	87046.6	16066.6	441.8	1578.3	2.4	66182.5
1500	100.0	100.0	0.0	151.0	893.4	92637.1	11309.6	719.1	30.2	2.9	951.4
1600	72.7	100.0	27.3	201.0	696.0	98422.6	16506.3	496.3	1633.8	4.4	36959.0
1700	72.7	100.0	27.3	171.0	894.2	104419.1	15664.4	566.6	1579.8	3.3	47122.9
1800	72.7	100.0	27.3	181.0	894.5	110633.3	14142.9	682.3	1553.8	3.1	50299.7
1900	72.7	100.0	27.3	238.5	696.6	116743.1	23963.0	387.2	1446.5	9.4	15287.8
2000	72.7	100.0	27.3	201.0	895.0	122570.0	17574.7	597.4	1420.9	4.2	33372.8

Table C.2: Medians of the Smart Lights Switch evaluation without using hints

# Maximum Training Set	Correctness			Training Set		Test Execution Time			Learning Runtime		
	Passive (%)	Active (%)	Improvement (%)	# Needed	Improvement (%)	Passive	Active	Improvement (%)	Passive (min)	Active (min)	Improvement (%)
50	94.9	100.0	5.1	20.5	143.9	6627.6	4003.8	65.5	20.2	4.3	367.9
100	98.4	100.0	1.6	16.0	525.0	12941.1	3359.7	285.2	54.5	1.8	2949.4
200	95.4	100.0	4.6	36.0	455.6	25597.8	7503.1	241.2	115.5	2.4	4784.7
300	93.8	100.0	6.2	46.0	552.2	38221.9	9596.6	298.3	174.4	2.1	8381.6
400	93.8	100.0	6.2	71.0	463.4	52129.5	16922.0	208.1	207.6	6.5	3099.4
500	87.6	100.0	12.4	76.0	557.9	65257.9	17049.1	282.8	259.1	10.1	2466.8
600	87.6	100.0	12.4	91.0	559.3	78392.7	19161.6	309.1	307.4	5.9	5084.6
700	87.6	100.0	12.4	193.5	261.8	92539.6	52162.7	77.4	368.3	9.0	3995.4
800	87.6	100.0	12.4	121.0	561.2	105860.1	30258.7	249.8	421.0	9.4	4366.3
900	87.6	100.0	12.4	158.5	467.8	118434.1	37104.6	219.2	457.4	24.2	1786.7
1000	87.6	100.0	12.4	151.0	562.3	131496.4	22852.9	475.4	483.0	8.7	5464.0
1100	87.6	100.0	12.4	193.5	468.5	144534.7	46829.5	208.6	542.8	12.6	4212.1
1200	87.6	100.0	12.4	151.0	694.7	157663.1	41023.3	284.3	564.4	11.3	4891.1
1300	87.6	100.0	12.4	163.5	695.1	169927.4	43239.8	293.0	600.5	9.5	6196.6
1400	87.6	100.0	12.4	211.0	563.5	183794.6	57326.9	220.6	639.6	21.1	2931.4
1500	87.6	100.0	12.4	226.0	563.7	196779.2	50742.8	287.8	639.0	8.5	7378.7
1600	87.6	100.0	12.4	201.0	696.0	210318.4	45685.6	360.4	671.3	8.6	7663.9
1700	87.6	100.0	12.4	213.5	696.3	222976.8	55091.4	304.7	656.1	11.0	5854.3
1800	87.6	100.0	12.4	271.0	564.2	235850.9	55609.7	324.1	643.6	7.1	9011.9
1900	87.6	100.0	12.4	238.5	696.6	248663.6	66787.0	272.3	622.4	7.2	8517.0
2000	87.6	100.0	12.4	201.0	895.0	261702.4	52042.3	402.9	616.6	6.5	9321.6

Table C.3: Medians of the CAS evaluation using hints

# Maximum Training Set	Correctness			Training Set		Test Execution Time			Learning Runtime		
	Passive (%)	Active (%)	Improvement (%)	# Needed	Improvement (%)	Passive	Active	Improvement (%)	Passive (min)	Active (min)	Improvement (%)
50	87.0	100.0	13.0	43.0	16.3	8765.5	10759.8	-18.5	16.4	37.9	-56.6
100	98.9	100.0	1.1	61.0	63.9	17101.6	13522.8	26.5	33.1	44.5	-25.7
200	100.0	100.0	0.0	79.0	153.2	35184.2	18773.9	87.4	69.4	31.8	118.0
300	100.0	100.0	0.0	111.5	169.1	53028.8	28042.6	89.1	72.8	42.0	73.4
400	100.0	100.0	0.0	139.0	187.8	70856.6	36298.4	95.2	138.4	43.6	217.1
500	100.0	100.0	0.0	169.0	195.9	87789.4	39026.0	125.0	84.0	43.4	93.5
600	100.0	100.0	0.0	203.5	194.8	104382.9	50701.3	105.9	95.4	53.4	78.7
700	100.0	100.0	0.0	196.0	257.1	121257.3	46155.5	162.7	95.7	47.0	103.7
800	100.0	100.0	0.0	248.5	221.9	138435.1	60300.0	129.6	130.2	53.7	142.6
900	100.0	100.0	0.0	226.0	298.2	155848.6	53743.1	190.0	149.4	49.6	201.3
1000	100.0	100.0	0.0	337.0	196.7	173534.6	86611.3	100.4	208.3	59.3	251.2
1100	100.0	100.0	0.0	249.0	341.8	191529.6	71893.5	166.4	145.0	69.8	107.8
1200	100.0	100.0	0.0	403.0	197.8	209734.5	99344.5	111.1	186.6	77.2	141.8
1300	100.0	100.0	0.0	402.5	223.0	225933.6	94367.4	139.4	203.9	68.6	197.2
1400	100.0	100.0	0.0	430.0	225.6	243317.6	102971.4	136.3	231.5	40.8	467.4
1500	100.0	100.0	0.0	379.0	295.8	261446.4	90396.8	189.2	290.2	64.8	347.5
1600	100.0	100.0	0.0	446.0	258.7	278962.4	106005.1	163.2	241.5	57.0	323.8
1700	100.0	100.0	0.0	567.0	199.8	297229.3	132830.6	123.8	280.2	68.8	307.0
1800	100.0	100.0	0.0	551.0	226.7	314712.4	123324.9	155.2	206.5	64.8	218.6
1900	100.0	100.0	0.0	637.0	198.3	333474.1	152724.6	118.3	107.0	70.9	51.0
2000	100.0	100.0	0.0	561.0	256.5	350732.9	139973.4	150.6	238.6	42.9	455.8

Table C.4: Medians of the CAS evaluation without using hints

# Maximum Training Set	Correctness			Training Set		Test Execution Time			Learning Runtime		
	Passive (%)	Active (%)	Improvement (%)	# Needed	Improvement (%)	Passive	Active	Improvement (%)	Passive (min)	Active (min)	Improvement (%)
50	91.0	100.0	9.0	41.5	20.5	9230.4	11961.7	-22.8	9.1	38.8	-76.7
100	97.8	100.0	2.2	53.5	86.9	18325.1	14772.1	24.1	12.9	24.2	-46.6
200	99.7	100.0	0.3	91.0	119.8	36855.0	25067.7	47.0	18.4	28.8	-36.0
300	99.8	100.0	0.2	106.0	183.0	56149.5	30804.2	82.3	22.1	30.4	-27.2
400	100.0	100.0	0.0	121.0	230.6	75772.0	37342.6	102.9	34.8	41.6	-16.3
500	100.0	100.0	0.0	163.5	205.8	94151.4	48679.3	93.4	41.4	47.1	-12.1
600	100.0	100.0	0.0	195.0	207.7	113232.2	61114.8	85.3	72.5	39.5	83.7
700	100.0	100.0	0.0	193.0	262.7	130927.3	65089.7	101.1	68.5	45.4	51.0
800	100.0	100.0	0.0	201.0	298.0	150381.1	54816.2	174.3	65.8	40.3	63.5
900	100.0	100.0	0.0	271.0	232.1	169642.3	77035.2	120.2	66.7	61.2	9.0
1000	100.0	100.0	0.0	326.0	206.7	187885.8	91806.0	104.7	87.7	44.5	97.1
1100	100.0	100.0	0.0	331.0	232.3	207173.5	88371.6	134.4	77.0	60.5	27.2
1200	100.0	100.0	0.0	301.0	298.7	225210.9	85884.1	162.2	110.1	53.6	105.5
1300	100.0	100.0	0.0	326.0	298.8	243976.0	101583.0	140.2	125.7	75.2	67.2
1400	100.0	100.0	0.0	351.0	298.9	263299.7	98317.5	167.8	103.4	68.9	50.0
1500	100.0	100.0	0.0	376.0	298.9	280285.4	111448.8	151.5	118.9	60.9	95.4
1600	100.0	100.0	0.0	361.0	343.2	299314.5	119137.8	151.2	128.8	48.2	167.1
1700	100.0	100.0	0.0	426.0	299.1	317411.8	121323.6	161.6	112.2	40.6	176.2
1800	100.0	100.0	0.0	541.0	232.7	336095.9	156838.7	114.3	156.9	63.3	148.1
1900	100.0	100.0	0.0	615.5	208.7	355324.2	184122.3	93.0	133.8	60.1	122.8
2000	100.0	100.0	0.0	601.0	232.8	374812.8	182925.5	104.9	106.1	65.6	61.8

Table C.5: Medians of the Particle Counter using hints

# Maximum Training Set	Correctness			Training Set		Test Execution Time			Learning Runtime		
	Passive (%)	Active (%)	Improvement (%)	# Needed	Improvement (%)	Passive	Active	Improvement (%)	Passive (min)	Active (min)	Improvement (%)
50	90.8	80.7	-12.6	50.0	0.0	2301.7	2171.5	6.0	58.0	21.3	172.5
100	99.1	100.0	0.9	83.5	19.8	4734.6	4233.8	11.8	78.9	32.9	140.0
200	99.7	100.0	0.3	116.0	72.4	9483.3	6342.8	49.5	108.8	27.5	296.3
300	99.7	100.0	0.3	181.0	65.7	14590.8	10359.2	40.8	218.6	37.9	477.3
400	100.0	100.0	0.0	191.0	109.4	19257.1	10027.8	92.0	252.8	30.7	723.6
500	100.0	100.0	0.0	276.0	81.2	23720.3	15504.9	53.0	481.6	47.0	924.1
600	100.0	100.0	0.0	316.0	89.9	28367.6	19403.4	46.2	378.1	48.2	684.8
700	99.8	100.0	0.2	316.0	121.5	32976.2	19110.6	72.6	424.1	37.8	1021.0
800	100.0	100.0	0.0	400.5	99.8	37853.9	23286.4	62.6	458.3	57.1	702.4
900	100.0	100.0	0.0	428.5	110.0	42597.0	24881.0	71.2	642.8	75.7	749.1
1000	100.0	100.0	0.0	526.0	90.1	47532.3	31526.7	50.8	507.6	100.0	407.5
1100	99.8	100.0	0.2	578.5	90.1	52460.0	36008.8	45.7	716.8	97.0	639.0
1200	88.1	100.0	11.9	601.0	99.7	57300.3	38334.2	49.5	955.9	93.1	926.8
1300	88.2	100.0	11.8	716.0	81.6	61882.7	42165.0	46.8	965.1	70.8	1262.1
1400	100.0	100.0	0.0	631.0	121.9	66955.4	37913.8	76.6	986.3	61.3	1508.2
1500	94.1	100.0	5.9	675.5	122.1	71912.8	38423.2	87.2	1008.5	51.4	1861.6
1600	93.8	100.0	6.2	721.0	121.9	76614.4	40166.9	90.7	1007.2	70.0	1338.9
1700	88.2	100.0	11.8	851.0	99.8	81014.9	49840.4	62.5	1203.8	96.5	1147.4
1800	100.0	100.0	0.0	856.0	110.3	85949.1	53027.2	62.1	1047.4	110.3	849.5
1900	100.0	100.0	0.0	856.0	122.0	90793.5	51976.1	74.7	996.1	47.4	2000.8
2000	88.2	100.0	11.8	1001.0	99.8	95463.7	59757.2	59.8	1064.8	96.1	1008.0

Table C.6: Medians of the Random Timed Automata with 10 locations using hints

# Maximum Training Set	Correctness			Training Set		Test Execution Time			Learning Runtime		
	Passive (%)	Active (%)	Improvement (%)	# Needed	Improvement (%)	Passive	Active	Improvement (%)	Passive (min)	Active (min)	Improvement (%)
50	97.2	100.0	2.8	39.3	27.4	2861.0	2164.7	32.2	3.5	4.2	-18.2
100	99.8	100.0	0.2	51.0	96.1	5715.8	3100.9	84.3	2.0	4.4	-54.3
200	99.9	100.0	0.1	72.5	175.9	11312.4	4344.0	160.4	4.2	4.5	-5.7
300	100.0	100.0	0.0	105.8	183.7	16979.9	6624.3	156.3	6.5	4.7	39.9
400	100.0	100.0	0.0	141.0	183.7	22649.1	7967.7	184.3	8.9	4.9	80.7
500	100.0	100.0	0.0	144.8	245.4	28249.4	9370.8	201.5	14.2	5.0	182.7
600	100.0	100.0	0.0	188.5	218.3	34032.0	11457.1	197.0	9.9	5.8	70.4
700	100.0	100.0	0.0	219.8	218.5	39906.6	12391.7	222.0	13.8	5.9	134.5
800	100.0	100.0	0.0	221.0	262.0	45754.6	13273.2	244.7	20.5	7.1	189.8
900	100.0	100.0	0.0	259.8	246.5	51352.2	15012.5	242.1	25.0	8.0	212.3
1000	100.0	100.0	0.0	301.0	232.2	56949.3	17623.8	223.1	16.6	8.2	101.2
1100	100.0	100.0	0.0	303.5	262.4	62787.2	17689.4	254.9	28.1	8.4	235.9
1200	100.0	100.0	0.0	346.0	246.8	68462.1	21168.3	223.4	35.9	7.5	378.7
1300	100.0	100.0	0.0	374.8	246.9	74196.7	22048.2	236.5	36.5	8.5	330.0
1400	100.0	100.0	0.0	368.5	279.9	79512.3	24211.1	228.4	25.4	9.4	170.5
1500	100.0	100.0	0.0	376.0	298.9	85084.3	22211.7	283.1	25.6	9.0	183.6
1600	100.0	100.0	0.0	481.0	232.6	90708.1	25588.0	254.5	35.5	10.6	233.4
1700	100.0	100.0	0.0	468.5	262.9	96372.5	29544.2	226.2	37.0	11.7	217.6
1800	100.0	100.0	0.0	496.0	262.9	102255.5	28049.5	264.6	45.5	11.4	299.7
1900	100.0	100.0	0.0	523.5	262.9	108113.1	30223.6	257.7	32.4	9.9	226.5
2000	100.0	100.0	0.0	551.0	263.0	113579.5	32808.8	246.2	42.0	10.7	292.6

Table C.7: Medians of the Random Timed Automata with 15 locations using hints

# Maximum Training Set	Correctness			Training Set		Test Execution Time			Learning Runtime		
	Passive (%)	Active (%)	Improvement (%)	# Needed	Improvement (%)	Passive	Active	Improvement (%)	Passive (min)	Active (min)	Improvement (%)
50	95.5	97.9	2.4	50.0	0.0	2949.9	3831.6	-23.0	12.0	10.8	11.3
100	97.5	100.0	2.5	100.0	0.0	5811.8	7233.8	-19.7	21.6	23.4	-7.5
200	98.8	100.0	1.2	165.3	21.0	11561.6	13575.9	-14.8	64.6	44.2	46.3
300	99.1	100.0	0.9	201.5	48.9	17075.8	14080.7	21.3	105.4	42.7	146.7
400	99.6	100.0	0.4	237.3	68.6	22732.6	19552.3	16.3	178.5	51.5	246.9
500	99.5	100.0	0.5	283.8	76.2	28516.7	23863.5	19.5	259.0	43.2	498.9
600	99.8	100.0	0.3	322.0	86.3	34077.1	24283.2	40.3	333.3	42.1	691.5
700	99.8	100.0	0.2	316.3	121.3	40188.5	30336.8	32.5	311.3	41.6	647.8
800	99.9	100.0	0.1	395.8	102.1	46111.5	32075.5	43.8	485.3	61.5	689.3
900	99.8	100.0	0.2	424.5	112.0	51974.4	35905.9	44.8	466.1	50.8	816.8
1000	99.8	100.0	0.2	443.3	125.6	57739.5	38481.2	50.0	675.0	55.3	1119.5
1100	99.7	100.0	0.3	458.5	139.9	63320.3	41066.1	54.2	570.9	52.5	987.5
1200	99.8	100.0	0.2	523.3	129.3	69026.5	49436.9	39.6	677.3	59.1	1046.3
1300	99.8	100.0	0.2	508.8	155.5	74690.6	44595.1	67.5	668.0	61.5	987.0
1400	99.8	100.0	0.2	535.3	161.6	80527.6	44445.6	81.2	712.4	77.1	823.7
1500	99.9	100.0	0.1	618.8	142.4	86386.0	51684.2	67.1	772.1	72.6	963.5
1600	99.9	100.0	0.1	703.0	127.6	91633.3	60728.9	50.9	755.5	98.9	664.0
1700	99.9	100.0	0.1	625.3	171.9	97129.4	61782.4	57.2	770.3	71.4	979.0
1800	99.9	100.0	0.1	680.3	164.6	102928.6	63727.0	61.5	763.9	77.0	892.4
1900	99.9	100.0	0.1	680.8	179.1	108761.9	60644.6	79.3	764.9	80.3	852.9
2000	99.9	100.0	0.1	751.0	166.3	114789.0	65412.6	75.5	654.2	76.2	758.6

Table C.8: Medians of the Random Timed Automata with 10 locations and 2 clocks using hints

# Maximum Training Set	Correctness			Training Set		Test Execution Time			Learning Runtime		
	Passive (%)	Active (%)	Improvement (%)	# Needed	Improvement (%)	Passive	Active	Improvement (%)	Passive (min)	Active (min)	Improvement (%)
50	97.9	99.4	1.5	50.0	0.0	2190.0	2090.3	4.8	2.1	5.0	-58.0
100	98.7	100.0	1.3	100.0	0.0	4320.0	3887.3	11.1	6.8	15.3	-55.6
200	99.5	100.0	0.5	146.0	37.0	8645.8	6926.8	24.8	34.6	29.9	15.9
300	99.7	100.0	0.3	197.8	51.7	13160.8	9639.7	36.5	60.2	52.1	15.5
400	99.7	100.0	0.3	247.8	61.5	17537.8	12258.6	43.1	108.3	51.3	111.0
500	99.8	100.0	0.2	290.5	72.1	22312.9	13196.2	69.1	183.0	34.8	425.1
600	99.8	100.0	0.2	372.5	61.1	26738.7	18034.9	48.3	219.9	84.0	161.9
700	99.8	100.0	0.2	416.3	68.2	31449.6	23306.7	34.9	321.3	82.7	288.4
800	99.8	100.0	0.2	408.0	96.1	35936.6	20679.7	73.8	334.1	54.6	512.1
900	99.9	100.0	0.1	451.3	99.4	40483.5	26639.6	52.0	372.6	94.7	293.6
1000	99.9	100.0	0.1	494.8	102.1	44898.3	26995.6	66.3	358.0	78.8	354.2
1100	99.9	100.0	0.1	617.8	78.1	49223.1	34881.3	41.1	401.7	152.3	163.8
1200	100.0	100.0	0.0	561.8	113.6	53935.8	28403.4	89.9	602.2	81.6	637.6
1300	99.9	100.0	0.1	660.5	96.8	58210.7	39329.2	48.0	556.6	105.3	428.7
1400	99.9	100.0	0.1	656.3	113.3	62464.9	37459.5	66.8	630.4	124.5	406.2
1500	100.0	100.0	0.0	799.3	87.7	66829.5	39689.8	68.4	660.7	134.5	391.1
1600	100.0	100.0	0.0	935.8	71.0	71197.3	45843.9	55.3	594.4	220.7	169.3
1700	99.9	100.0	0.1	891.3	90.7	75806.6	48128.2	57.5	680.0	220.7	208.1
1800	99.9	100.0	0.1	752.0	139.4	80458.0	42910.7	87.5	632.0	79.8	691.9
1900	99.9	100.0	0.1	964.3	97.0	85094.1	52438.3	62.3	658.0	159.4	312.7
2000	100.0	100.0	0.0	918.3	117.8	89504.9	56232.2	59.2	681.4	141.3	382.3

Table C.9: Medians of the Random Timed Automata with 20 locations using hints

# Maximum Training Set	Correctness			Training Set		Test Execution Time			Learning Runtime		
	Passive (%)	Active (%)	Improvement (%)	# Needed	Improvement (%)	Passive	Active	Improvement (%)	Passive (min)	Active (min)	Improvement (%)
50	95.8	99.1	3.3	50.0	0.0	3126.9	2626.1	19.1	24.0	9.5	152.3
100	98.2	100.0	1.8	100.0	0.0	5932.7	5195.3	14.2	56.9	19.4	192.6
200	99.1	100.0	0.9	189.8	5.4	11819.3	10727.7	10.2	62.4	35.9	73.5
300	99.5	100.0	0.5	264.3	13.5	17866.4	15390.2	16.1	126.7	66.5	90.5
400	99.8	100.0	0.2	301.5	32.7	23771.2	19113.2	24.4	198.6	83.1	139.0
500	99.8	100.0	0.2	343.5	45.6	29775.3	23429.6	27.1	278.3	111.0	150.8
600	99.8	100.0	0.3	461.3	30.1	35854.4	28171.7	27.3	341.9	96.7	253.6
700	99.8	100.0	0.2	485.0	44.3	41842.8	32905.4	27.2	397.6	127.5	211.8
800	99.9	100.0	0.1	500.3	59.9	47799.8	35295.5	35.4	421.4	124.4	238.8
900	99.8	100.0	0.2	690.0	30.4	53818.6	40613.8	32.5	369.5	137.2	169.4
1000	99.8	100.0	0.2	723.8	38.2	59789.7	43983.4	35.9	510.4	155.2	228.9
1100	99.9	100.0	0.1	621.0	77.1	65529.4	44246.7	48.1	470.7	179.4	162.3
1200	99.9	100.0	0.1	711.3	68.7	71266.6	48522.4	46.9	541.3	161.8	234.5
1300	99.9	100.0	0.1	783.8	65.9	77276.0	56781.3	36.1	632.1	171.0	269.8
1400	99.9	100.0	0.1	887.8	57.7	83062.2	57225.7	45.1	550.0	212.1	159.3
1500	99.9	100.0	0.1	1102.0	36.1	88744.5	68676.1	29.2	572.5	213.3	168.5
1600	99.9	100.0	0.1	1002.3	59.6	94982.2	66352.7	43.1	725.9	177.1	309.9
1700	99.9	100.0	0.1	1005.3	69.1	100952.7	70736.3	42.7	555.8	146.2	280.1
1800	99.9	100.0	0.1	1198.3	50.2	106845.2	72410.1	47.6	708.9	197.8	258.4
1900	99.9	100.0	0.1	1208.0	57.3	112799.8	85562.3	31.8	579.3	258.7	123.9
2000	100.0	100.0	0.0	1164.3	71.8	118883.1	81361.9	46.1	693.3	226.7	205.8

Bibliography

[1] Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korosec, Willibald Krenn, Rupert Schlick, and Birgit Vera Schmidt. Model-based mutation testing of an industrial measurement device. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings*, volume 8570 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2014. (Cited on page 56.)

[2] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. UML in action: a two-layered interpretation for testing. *ACM SIGSOFT Software Engineering Notes*, 36(1):1–8, 2011. (Cited on page 49.)

[3] Bernhard K. Aichernig, Florian Lorber, and Dejan Nickovic. Time for mutants - model-based mutation testing with timed automata. In Margus Veanes and Luca Viganò, editors, *Tests and Proofs - 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer, 2013. (Cited on pages 6 and 49.)

[4] Bernhard K. Aichernig, Wojciech Mostowski, Mohammad Reza Mousavi, Martin Tappler, and Masoumeh Taromirad. Model learning and model-based testing. In Amel Bennaceur, Reiner Hähnle, and Karl Meinke, editors, *Machine Learning for Dynamic Software Analysis: Potentials and Limits - International Dagstuhl Seminar 16172, Dagstuhl Castle, Germany, April 24-27, 2016, Revised Papers*, volume 11026 of *Lecture Notes in Computer Science*, pages 74–100. Springer, 2018. (Cited on page 74.)

[5] Bernhard K. Aichernig and Martin Tappler. Learning from faults: Mutation testing in active automata learning. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, volume 10227 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2017. (Cited on pages 22, 75, 77 and 79.)

[6] Rajeev Alur. Timed automata. In Nicolas Halbwachs and Doron A. Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 8–22. Springer, 1999. (Cited on page 8.)

[7] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. (Cited on pages 1, 5 and 76.)

[8] Dana Angluin. A note on the number of queries needed to identify regular languages. *Information and Control*, 51(1):76–87, 1981. (Cited on page 4.)

[9] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987. (Cited on pages 1, 3 and 74.)

[10] Eugene Asarin, Oded Maler, Amir Pnueli, and Joseph Sifakis. Controller synthesis for timed automata. *IFAC Proceedings Volumes*, 31(18):447 – 452, 1998. 5th IFAC Conference on System Structure and Control 1998 (SSC'98), Nantes, France, 8-10 July. (Cited on page 24.)

[11] Wayne D. Blizard. The development of multiset theory. *Modern Logic*, 1(4):319–352, 1991. (Cited on page 3.)

[12] Sébastien Bornot, Joseph Sifakis, and Stavros Tripakis. Modeling urgency in timed systems. In Willem P. de Roever, Hans Langmaack, and Amir Pnueli, editors, *Compositionality: The Significant Difference, International Symposium, COMPOS'97, Bad Malente, Germany, September*

8-12, 1997. Revised Lectures, volume 1536 of *Lecture Notes in Computer Science*, pages 103–129. Springer, 1997. (Cited on page 7.)

[13] Rafael C. Carrasco and José Oncina. Learning stochastic regular grammars by means of a state merging method. In Rafael C. Carrasco and José Oncina, editors, *Grammatical Inference and Applications, Second International Colloquium, ICGI-94, Alicante, Spain, September 21-23, 1994, Proceedings*, volume 862 of *Lecture Notes in Computer Science*, pages 139–152. Springer, 1994. (Cited on page 74.)

[14] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, 1978. (Cited on page 75.)

[15] Charles Darwin. *The Origin of Species: By Means of Natural Selection Or the Preservation of Favored Races in the Struggle for Life*, volume 1. Modern library, 1872. (Cited on page 9.)

[16] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, New York, NY, 2010. (Cited on page 74.)

[17] Pierre Dupont, Bernard Lambeau, Christophe Damas, and Axel van Lamsweerde. The QSM algorithm and its application to software behavior model induction. *Applied Artificial Intelligence*, 22(1&2):77–115, 2008. (Cited on page 4.)

[18] Paul Fiterau-Brostean, Ramon Janssen, and Frits W. Vaandrager. Combining model learning and model checking to analyze TCP implementations. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 454–471. Springer, 2016. (Cited on page 15.)

[19] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991. (Cited on page 75.)

[20] Olga Grinchtein, Bengt Jonsson, and Martin Leucker. Learning of event-recording automata. *Theoretical Computer Science*, 411(47):4029–4054, 2010. (Cited on pages 2 and 75.)

[21] Olga Grinchtein, Bengt Jonsson, and Paul Pettersson. Inference of event-recording automata using timed decision trees. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 435–449. Springer, 2006. (Cited on pages 2 and 75.)

[22] Roland Groz, Nicolas Brémond, and Adenilso Simão. Using adaptive sequences for learning non-resettable FSMs. In Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek, editors, *Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018, Wrocław, Poland, September 5-7, 2018*, volume 93 of *Proceedings of Machine Learning Research*, pages 30–43. PMLR, 2018. (Cited on page 75.)

[23] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using UPPAAL. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 77–117. Springer, 2008. (Cited on page 8.)

[24] Anders Hessel, Kim Guldstrand Larsen, Brian Nielsen, Paul Pettersson, and Arne Skou. Time-optimal real-time test case generation using UPPAAL. In Alexandre Petrenko and Andreas Ulrich, editors, *Formal Approaches to Software Testing, Third International Workshop on Formal*

Approaches to Testing of Software, FATES 2003, Montreal, Quebec, Canada, October 6th, 2003, volume 2931 of *Lecture Notes in Computer Science*, pages 114–130. Springer, 2003. (Cited on pages viii, 5, 6, 7, 75, 76 and 79.)

[25] John H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* MIT Press, Cambridge, MA, USA, 1992. (Cited on page 9.)

[26] Yating Hsu, Guoqiang Shu, and David Lee. A model-based approach to security flaw detection of network protocol implementations. In *Proceedings of the 16th annual IEEE International Conference on Network Protocols, 2008. ICNP 2008, Orlando, Florida, USA, 19-22 October 2008*, pages 114–123. IEEE Computer Society, 2008. (Cited on page 74.)

[27] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source LearnLib - A framework for active automata learning. In Daniel Kroening and Corina S. Pasareanu, editors, *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 487–495. Springer, 2015. (Cited on page 1.)

[28] Colin G. Johnson. Genetic programming with fitness based on model checking. In Marc Ebner, Michael O'Neill, Anikó Ekárt, Leonardo Vanneschi, and Anna Esparcia-Alcázar, editors, *Genetic Programming, 10th European Conference, EuroGP 2007, Valencia, Spain, April 11-13, 2007, Proceedings*, volume 4445 of *Lecture Notes in Computer Science*, pages 114–124. Springer, 2007. (Cited on page 76.)

[29] Bengt Jonsson and Frits W. Vaandrager. Learning Mealy Machines with Timers. online preprint, 2019. available via `http://www.sws.cs.ru.nl/publications/papers/fvaan/MMT/`, accessed 12.05.2019. (Cited on pages 2 and 75.)

[30] Gal Katz and Doron Peled. Synthesizing, correcting and improving code, using model checking-based genetic programming. *STTT*, 19(4):449–464, 2017. (Cited on pages 76 and 79.)

[31] John R. Koza. *Genetic programming - on the programming of computers by means of natural selection.* Complex adaptive systems. MIT Press, 1993. (Cited on pages 1, 9, 11 and 12.)

[32] Zhifeng Lai, S. C. Cheung, and Yunfei Jiang. Dynamic model learning using genetic algorithm under adaptive model checking framework. In *Sixth International Conference on Quality Software (QSIC 2006), 26-28 October 2006, Beijing, China*, pages 410–417. IEEE Computer Society, 2006. (Cited on page 76.)

[33] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. Results of the Abbadingo One DFA learning competition and a new evidence-driven state merging algorithm. In Vasant G. Honavar and Giora Slutzki, editors, *Grammatical Inference, 4th International Colloquium, ICGI-98, Ames, Iowa, USA, July 12-14, 1998, Proceedings*, volume 1433 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998. (Cited on page 4.)

[34] Kim Guldstrand Larsen, Marius Mikucionis, and Brian Nielsen. Online testing of real-time systems using UPPAAL. In Jens Grabowski and Brian Nielsen, editors, *Formal Approaches to Software Testing, 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, volume 3395 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2004. (Cited on page 8.)

[35] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2):134–152, 1997. (Cited on page 8.)

[36] Raluca Lefticaru, Florentin Ipate, and Cristina Tudose. Automated model design using genetic algorithms and model checking. In Petros Kefalas, Demosthenes Stamatis, and Christos Douligeris, editors, *2009 Fourth Balkan Conference in Informatics, BCI 2009, Thessaloniki, Greece, 17-19 September 2009*, pages 79–84. IEEE Computer Society, 2009. (Cited on page 76.)

[37] Simon M. Lucas and T. Jeff Reynolds. Learning DFA: evolution versus evidence driven state merging. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2003, 8 - 12 December 2003, Canberra, Australia*, pages 351–358. IEEE, 2003. (Cited on page 76.)

[38] Hua Mao, Yingke Chen, Manfred Jaeger, Thomas D. Nielsen, Kim G. Larsen, and Brian Nielsen. Learning deterministic probabilistic automata from a model checking perspective. *Machine Learning*, 105(2):255–299, 2016. (Cited on page 74.)

[39] Braham Lotfi Mediouni, Ayoub Nouri, Marius Bozga, and Saddek Bensalem. Improved learning for stochastic timed models by state-merging algorithms. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods - 9th International Symposium, NFM 2017, Moffett Field, CA, USA, May 16-18, 2017, Proceedings*, volume 10227 of *Lecture Notes in Computer Science*, pages 178–193. Springer, 2017. (Cited on page 74.)

[40] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, 1998. (Cited on pages 9 and 10.)

[41] Mariusz Nowostawski and Riccardo Poli. Parallel genetic algorithm taxonomy. In Lakhmi C. Jain, editor, *Third International Conference on Knowledge-Based Intelligent Information Engineering Systems, KES 1999, Adelaide, South Australia, 31 August - 1 September 1999, Proceedings*, pages 88–92. IEEE, 1999. (Cited on page 10.)

[42] José Oncina and Pedro García. *Identifying Regular Languages In Polynomial Time*, pages 99–108. World Scientific, 11 2002. (Cited on page 74.)

[43] Doron A. Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. *Journal of Automata, Languages and Combinatorics*, 7(2):225–246, 2002. (Cited on pages 1 and 75.)

[44] Alexander Pretschner. Model-based testing in practice. In John S. Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*, pages 537–541. Springer, 2005. (Cited on page 4.)

[45] Julien Schmaltz and Jan Tretmans. On conformance testing for timed systems. In Franck Cassez and Claude Jard, editors, *Formal Modeling and Analysis of Timed Systems, 6th International Conference, FORMATS 2008, Saint Malo, France, September 15-17, 2008. Proceedings*, volume 5215 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2008. (Cited on pages 8 and 75.)

[46] Jan Springintveld, Frits W. Vaandrager, and Pedro R. D'Argenio. Testing timed automata. *Theor. Comput. Sci.*, 254(1-2):225–257, 2001. (Cited on pages 7 and 75.)

[47] Martin Tappler, Bernhard K. Aichernig, Kim Guldstrand Larsen, and Florian Lorber. Learning timed automata via genetic programming. *arXiv preprint arXiv:1808.07744*, abs/1808.07744, 2018. (Cited on pages viii, 1, 2, 5, 7, 11, 12, 13, 15, 16, 18, 19, 26, 32, 39, 40, 41, 42, 43, 60, 76, 77 and 78.)

[48] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer, 2008. (Cited on pages 8, 16, 75 and 76.)

[49] M. P. Vasilevskii. Failure diagnosis of automata. *Cybernetics*, 9(4):653–665, Jul 1973. (Cited on page 75.)

[50] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. An algorithm for learning real-time automata. In Maarten van Someren, Sophia Katrenko, and Pieter Adriaans, editors, *Proceedings of the Sixteenth Annual Machine Learning Conference of Belgium and the Netherlands (Benelearn)*, pages 128–135, 2007. (Cited on page 74.)

[51] Sicco Verwer, Mathijs de Weerdt, and Cees Witteveen. A likelihood-ratio test for identifying probabilistic deterministic real-time automata from positive data. In José M. Sempere and Pedro García, editors, *Grammatical Inference: Theoretical Results and Applications, 10th International Colloquium, ICGI 2010, Valencia, Spain, September 13-16, 2010. Proceedings*, volume 6339 of *Lecture Notes in Computer Science*, pages 203–216. Springer, 2010. (Cited on page 74.)

[52] Michele Volpato and Jan Tretmans. Approximate active learning of nondeterministic input output transition systems. *Electronic Communications of the EASST*, 72, 2015. (Cited on page 15.)

[53] Neil Walkinshaw, John Derrick, and Qiang Guo. Iterative refinement of reverse-engineered models by model-based testing. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 305–320. Springer, 2009. (Cited on pages viii, 2, 3, 4, 15, 16, 17, 19, 76 and 77.)