LONDON'S GLOBAL UNIVERSITY

Specification Mining with Active Automata Learning

Raquel Fernandes da Silva[1]

MEng Computer Science

Supervisors: Tiago Ferreira, Alexandra Silva

Submission date: 24 May 2024

[1]**Disclaimer:** This report is submitted as part requirement for the MEng Computer Science degree at UCL. It is substantially the result of my own work except where explicitly indicated in the text.
The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

Constant fast-paced technological advancements are a major catalyst for software complexity, which becomes concerning as software systems spread to all areas of society and become safety-critical. Formal methods provide tools to mitigate this issue by targeting bugs and faults, however they require formal specifications for determining the intended behaviour of the system. This is problematic as specification-making is a time-consuming, laborious, and error-prone process, and is often avoided unless strictly necessary. This thesis addresses this problem by introducing a new approach to specification creation based on formal inference of system behaviour alongside user-based manual adjustments. Active automata learning is used to learn the behaviour of the target system in the form of deterministic finite automata (DFA) and register automata (RA). Then through an extension of the automata learning framework C$\mathcal{E}$AL, the generated specification can be manually refined through user-provided counterexamples. A proof-of-concept specification mining tool is also implemented which showcases the ability to generate a specification of the Java Iterator interface tailored to the user's preference. This work lays the foundations for a more reliable and efficient specification creation process that aims to make formal methods more accessible to real-world systems.

Acknowledgements

This dissertation would not have been possible without the help of some remarkable people. First and foremost, I would like to thank my supervisor and friend Tiago Ferreira for his incredible support over the past three years. His mentorship has been an incredible blessing in my life. He has guided me tirelessly through the beautiful world of research in formal methods and I am forever grateful for the incredible opportunities he has provided me with. I would also like to thank Prof. Alexandra Silva for her equal commitment to mentoring me in the world of academic research. I owe them both my confidence as a researcher, and without them I would not be pursuing a PhD in the coming years. My gratitude is beyond words.

I want to extend my appreciation to my beautiful friends without whom I would have not been able to achieve the success I have. To my incredibly talented friends from home Luís Ferreira, Filipe Laíns, and João Lourenço, thank you for guiding me through my first dabs at Computer Science. To my dearest João Viana, your immeasurable and unwavering support through my most difficult times is unrepayable and I look forward to owing you for the rest of my life. To my kindest and smartest of friends Sidonie Coker, our beautiful friendship brings so much joy to my life, and with you I could never doubt that women in STEM can do anything. To my dad in London Paul Byrne, your care and mentorship has made me certain that I should never settle for anything less than excelence, and I am always certain that it is achievable when I am with you. I would also like to thank my beautiful caring partner David Loughlin for the unbelievably incredible meals and the happiest of moments that sustained me through my most difficult work, I am so lucky to be able to share life with you. And to all the many others I have not mentioned but have enriched my life in beautiful ways.

I must also mention the cutest furballs that have been with me throughout my academic path. To Paganini Stradivarius, Fénix, Mr. Snifflers, and Enzo, I wish you were all here to celebrate this accomplishment with me.

Last but not least, I would like to thank my family. This includes but is by no means limited to my grandparents, aunts, uncles, and step-mother. Their love and care is what keeps me stiving to make them proud every single day. My parents São Fernandes and António Silva never stopped believing in me and ensured I always had everything I needed to achieve my dreams. Mom, Dad, I owe my success to your unconditional support, I love you dearly.

Contents

1 Introduction **2**

2 Motivation **4**
2.1 Contributions . . . 5

3 State of the Art **6**

4 Preliminaries **8**
4.1 Formal Specification . . . 8
4.2 Deterministic Finite-State Automata (DFA) . . . 8
4.2.1 String Acceptance Example . . . 9
4.3 Register Automata (RA) . . . 10
4.3.1 Data Word Acceptance Example . . . 11
4.3.2 Expressiveness of RAs . . . 11
4.4 Active Automata Learning . . . 12
4.4.1 MAT Framework . . . 12
4.4.2 L* Algorithm . . . 12
4.4.3 C$\mathcal{E}$AL Framework . . . 15

5 Solution **18**
5.1 From Specifications to the Iterator Interface . . . 18
5.2 Mining with DFAs . . . 19
5.2.1 Initial Implementation . . . 19
5.2.2 Incorporating the Method `remove` . . . 20
5.2.3 Incorporating the Method `add` . . . 22
5.3 Mining with Register Automata . . . 25
5.3.1 Bounded Iterators . . . 25
5.3.2 Unbounded Iterators . . . 26
5.4 Extending C$\mathcal{E}$AL and Using It to Accept User Input . . . 28
5.5 Introduction of the User Interface . . . 30
5.6 Specification Miner Example Run . . . 31

6 Conclusion **35**
6.1 Achievements . . . 35
6.2 Evaluation . . . 35
6.3 Future Work . . . 36
6.4 Final Remarks . . . 36

Chapter 1

Introduction

In a world riddled with extremely complex software systems, there is a major need for easy and clear descriptions of the behaviour of such systems - these are specifications. A common example of this is user manuals, which describe how to interact with the system to obtain the desired outcome and the precautions to have to prevent bad outcomes from happening. These specifications vary in the level of detail and abstraction to better suit the needs of their target audience. One key area that requires specifications is formal verification, a tool that provides correctness guarantees on specific system properties, which is particularly important for safety-critical systems.

Nonetheless, developing specifications requires a full in-depth understanding of the system and the intricacies of all possible interactions available. Even if such a person exists, they would have to go through a mostly manual and extremely time-consuming process to develop the specification, which in the end would still be prone to human error. It is therefore understandable why most developers and architects steer away from creating detailed system specifications unless necessary.

A question now arises: how to improve the specification-making process. One clear direction is to reduce the amount of manual work necessary by automating part of the specification inference. This is an area that has been mainly explored in the last two decades, however the produced work has several practical limitations, one of them being the inability to ensure that the obtained specification accurately describes the *intent* of the system. This mainly comes from the use of machine learning for the specification mining. As such, formal methods, due to their mathematical rigour, are a promising method for overcoming this issue.

By making specification creation more automated, the flexibility to alter the generated specification becomes more complicated. Since formal methods are based on logical reasoning and as such require consistency and determinism, contradicting learned behaviour is generally not supported by the current algorithms. This extension is nonetheless necessary, as providing software-development-like tools for creating specifications improves the experience of generating system specifications and makes all tools that require this type of system representation much more accessible.

This project aims to explore state-of-the-art formal methods for specification inference and to extend them to support the manual tweaking of the resulting specification. It also has the goal of producing a proof-of-concept tool for specification inference that overcomes the aforementioned limitations of current tools. The first stage of the project involved a literature review of the evolution of specification mining methods and the most current technology. Then the state-of-the-art active automata learning methods were applied to an exemplary specification mining instance. The subsequent stage involved the extension of these state-of-the-art methods to support counterexamples that overwrite part of the learned behaviour. Lastly, the user interface was implemented

to illustrate the possible use of the introduced work in a practical setting.

Outline: Firstly we motivate the problem and proposed solution in depth and list the project contributions. We then describe the state-of-the-art, explain its current limitations, and further expand on how this project distinguishes itself from previous work. We follow this by delineating the preliminary knowledge to provide the necessary background to the project details. Next we explain the implementation of the proposed solution and the chosen design decisions, finalising with a running example of the tool built. Finally we provide an overview of the project, highlight achievements and detail future work.

Chapter 2

Motivation

Software is expanding to almost every part of society, from entertainment to transportation and even healthcare. As these software systems are applied more widely and, consequently, become more complex, it becomes harder to ensure they behave as expected. Real-world systems employ features that are difficult to formally reason about, and the tools that analyse their correctness are inherently hard to develop and often under-resourced [Alg+11]. In spite of these difficulties, ensuring correctness of behaviour is important, particularly when it comes to safety-critical systems [Kli96]. A key example of this is Therac-25, a radiation machine which contained, among a variety of issues, bugs that majorly contributed to the fatalities of the patients using it [LT93]. This example has been an important cautionary tale for computer scientists since then, having brought to light the importance of spending enough resources in verifying critical software and motivating a lot of the research in the area of verification [Gar05; Tho94; BZK21].

Non-critical systems, on the other hand, do not demand such a restrictive perspective on their correctness, as their malfunction does not often carry such grave consequences. Nevertheless, software engineers still seek the most guarantees for the least effort [Lem+17]. The prevalent industry strategy is to use testing, but even though testing proves that a system behaves as expected for a finite amount of scenarios, it provides little guarantees on the absence of incorrect behaviour [Ber07]. The consequence of this is evident in the breakdown of software development costs, of which debugging accounts for half [Bri+20]. As such, the interest in practical easy-to-use tools that rapidly and effectively detect bugs has dramatically increased [HP18].

Formal methods have been able to offer various solutions to this problem through techniques such as automata theory, program semantics, and type systems [MB03]. Their ability to systematically prove that a given system does not violate specific properties has been successfully applied to critical systems, such as aerospace operating systems [Mar+07] and hardware chips [NM98]. These tools require specifications to formally define which behaviours are "right" or "wrong" [Hie+09]. Nonetheless, with the constant pressure in the current software development setting for as much production in as little time as possible, the time-consuming task of manually generating accurate representations of systems becomes unfeasible [ABL02]. As such, the problem of how to remove the barriers to specification creation is posed.

When developers implement a system, the details of its overarching architecture are often just part of the knowledge base of the person who designs it and those involved with its overall implementation. The only available specifications tend to be the documentation accompanying the codebase and the official user helper manual, and sometimes a diagram of the system structure. However, these are rarely formal and are often prone to errors. Developers and system architects

need to spend a lot of time dealing with the technical issues and, as such, it makes it hard for them to spend enough time creating simple and correct representations of their system. Often, their view of the system is biased towards the envisioned version of their project and does not accurately reflect real behaviour [Leg+19]. An example of this is the presence of bugs in software - it is not envisioned during development that a system will be buggy, however this does not guarantee that the system is indeed bug-free. This is also an important consideration when reasoning about how to improve the accuracy of specification creation.

There have been attempts at automating the process of creating formal specifications [ABL02; HMS03; LL18]. However, they struggle to find a good balance between customisability and automation, often leading to tools that either require too many and too trivial details or are too inflexible to user input. They also usually relied on either the user fully specifying the system from scratch, where the only automated part was the generation of diagrams and documentation, or on the use of machine learning, which often produces inaccurate results.

To first address the inaccuracy of specifications, one current research area that has produced promising practical results is active automata learning [Vaa17]. This is a method that infers formal models of systems through interaction. Their main benefit is the production of specifications that accurately represent the observed behaviour of the system. However, this methodology only produces accurate models if the target system is correct. As such, we need to allow manual user guidance for producing accurate specifications even if the system is not fully correct. This is particularly useful for leveraging current implementations of a system for specification inference even if the system is still under development.

Most state-of-the-art automata learning frameworks struggle to consider information that contradicts previously learned behaviour. For specification inference, this means that specific learned behaviours are hard, if not impossible, to be overridden by the user. However, the recent framework C$\mathcal{E}$AL provides a more flexible learning model which reasons about how to handle contradictory behaviours [Fer+23]. This framework was implemented for the use-case of learning black-box systems with noise and/or persistent changes during the learning process. However, it has the potential to be adapted to instead fit the setting of guided specification inference.

2.1 Contributions

Firstly, we explored the use of current state-of-the-art active automata learning methods for specification inference. We have showcased the potential of these tools for generating initial specification models that accurately portray the systems' implemented behaviour, enhancing the specification-making process by automating one of the most time-consuming parts of it.

We have also explored the differences between the different automata learning algorithms and frameworks and highlighted the key differences in the types of specification representation via different automata types. In particular, we have showcased the limitations of some models and showed how others are able to infer the specification of a strictly bigger set of systems, as well as presenting them more compactly.

Subsequently, we extended the state-of-the-art C$\mathcal{E}$AL framework to produce deterministic finite automata and register automata models. We also introduced a new type of counterexample origin, the user, which, alongside some adaptations to the original C$\mathcal{E}$AL algorithm, allows for manually overwriting specific learned behaviour.

Lastly, we produced a prototype of a specification mining tool that showcases the potential future of specification creation in a way that better suits the needs of the industry.

Chapter 3

State of the Art

Automata-based specification inference is first motivated by Cook and Wolf [CW95]. They propose some theoretical solutions to infer automata models from event sequences using three contrasting methods: algorithmic grammar inference, Markov models, and neural networks. Ammons, Bodík, and Larus [ABL02] then applied this work to software to generate an FSA that describes segments of execution traces via probabilistic grammar inference. This paper pioneered the use of the expression "specification mining" for the area of extrapolating software specifications from system traces and set a precedent for subsequent work on specification inference. One of the main drawbacks of their original approach was the need for each program trace to be manually labelled by some software expert as erroneous or correct, which not only would require extreme technical expertise but also would likely be subject to human error, particularly for complex systems. In subsequent work, Ammons et al. [Amm+03] introduce clustering of traces with equivalent behaviour to minimise the amount of labelling needed. However, manual classification is still required.

Some of the subsequent work in specification mining focused on heuristics improvement. Lo and Khoo [LK06] explore the use of trace clustering and filtering techniques to increase the accuracy, precision and recall of the learned specifications. Alternatively, Whaley, Martin, and Lam [WML02] propose a new inference strategy by combining both static and dynamic analysis to extract component-based FSA specifications for object-oriented languages. More recent work involves exploring recent machine learning technologies such as deep learning for specification mining and training data test-case generation [LL18].

One of the main limitations in the current state-of-the-art tools concerns the correctness of the inferred specifications. In particular, the described algorithms use machine learning methods to obtain a specification that has a high likelihood of matching the observed system behaviour. This approach, however, provides limited guarantees, and most of the recent work has been focused on mitigating the learning of incorrect behaviour rather than ensuring correctness from the start of the specification mining process. As such, formal methods emerge as a good alternative.

Active automata learning was initially explored for specification mining by Hungar, Niese, and Steffen [HNS03] and Hungar, Margaria, and Steffen [HMS03], where the authors successfully used it to generate FSAs that explain the observed system behaviour. This was further explored in conjunction with grammar inference by Walkinshaw et al. [Wal+07]. These two works, however, are not only outdated compared to the current state-of-the-art automata learning algorithms but also unable to allow for the manual tweaking of the learned models. Performing this type of alterations to specifications inferred manually is also not straightforward, as often the change to the labelling of a single behaviour requires substantial changes to the original specification structure.

The unsoundness of most specification mining tools and the need for manually relabeling specific specification behaviours motivates our previously described pursuit of exploring the use of current state-of-the-art active automata learning techniques to produce flexible specification mining tools.

Chapter 4

Preliminaries

This chapter provides the necessary technical background for understanding the work described in the following chapters. First we define *formal specification*, then introduce *deterministic finite automata* and *register automata* alongside some useful properties and examples, and lastly introduce and compare the two main *active learning frameworks* and detail an exemplary *learning algorithm*.

4.1 Formal Specification

The expression *formal specification* does not have a straightforward definition. At a conceptual level, it is the expression, in a formal language and at some level of abstraction, of a collection of properties a system should satisfy [Lam00]. Some of the purposes of a specification are to aid the software development process, to describe a system, and to help analyse system behaviour [Hie+09]. For the purpose of this project, a formal specification is defined as a formal set of properties that form a necessary and sufficient condition for implementation correctness.

Formal specifications have a variety of possible representations: algebraic structures, logics, finite-state automata (FSA), among others. These are often equivalent, even though they can each be more adequate for a particular context. For example, a logic-based representation is useful when the properties we want to check about the system are in the form of logical formulas, whereas an algebraic representation is harder to utilise for this particular purpose. This project focuses on FSAs, as the chosen inference tools produce automaton representations of systems.

4.2 Deterministic Finite-State Automata (DFA)

Before formally defining DFAs, we first fix the relevant notation. An *alphabet* Σ is a finite set of symbols. A *string* or *word* is a finite sequence of symbols in Σ. The empty string is denoted by ε. The set of all possible strings is denoted by Σ^*. $u \cdot v$ is the *concatenation* operation which appends v to u. A *formal language* over Σ is a subset of Σ^* where all its strings conform to the formal rules defined by the language. From here on we refer to them only as languages.

Example: Let Σ be the alphabet $\{a, b\}$. The string "aba" is an example of a string in Σ^*. This string concatenated with the string "a" is denoted by "aba" $\cdot$ "a" = "$abaa$". We can define a formal language A as a language over Σ where there is always an even number of both as and bs. The string "$abaaba$" is in A as it contains an even number of both as and bs, but "aba" is not as it contains an odd number of bs.

DEFINITION 1 *A deterministic finite automaton M is a tuple $(Q, q_0, \Sigma, \delta, F)$, where:*

- *Q is a finite set of states*
- *$q_0 \in Q$ is an initial state*
- *Σ is an alphabet*
- *$\delta : Q \times \Sigma \to Q$ is a transition function*
- *$F \subseteq Q$ is a set of final/accepting states*

For conciseness, we can define the additional function $\delta^* : \Sigma^* \to Q$ which maps each string to the state reached by M after processing the given string. Formally:

$$\delta^*(w) = \begin{cases} q_0 & \text{if } w = \varepsilon \\ \delta(\delta^*(u), a) & \text{if } w = u \cdot a, \text{ with } u \in \Sigma^* \text{ and } a \in \Sigma \end{cases}$$

A DFA M implements a function $A : \Sigma^* \to \mathbb{2}$ with $\mathbb{2} = \{\top, \bot\}$, which defines if a string is accepted by M. Formally:

$$A(w) = \begin{cases} \top & \text{if } \delta^*(w) \in F \\ \bot & \textit{otherwise} \end{cases}$$

The language of a DFA M is defined as $\mathcal{L}(M) = \{w \in \Sigma^* \mid A(w) = \top\}$, which is the set of strings that the DFA accepts.

4.2.1 String Acceptance Example

Let M be a DFA over $\Sigma = \{a, b\}$ that implements the language that accepts strings with odd numbers of both as and bs. Figure 4.1 shows a graphical representation of this DFA. Each of the states corresponds to a different combination of parity values as described in Table 4.1. The state l_3 is the only accepting state (characterised by the double circle) and it indeed matches with the only one for which the parity is odd for both alphabet characters.

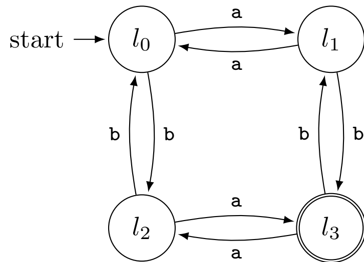

Figure 4.1: DFA that accept an odd number of as and bs

	a	**b**
$\boldsymbol{l_0}$	even	even
$\boldsymbol{l_1}$	odd	even
$\boldsymbol{l_2}$	even	odd
$\boldsymbol{l_3}$	odd	odd

Table 4.1: Parity of each state

Let us now consider the example strings u = "*ab*" and v = "*bab*". To verify the acceptance of

a string by M we need to check if $\delta^*(w) \in F$. For the string u we have:

$$
\begin{aligned}
&\delta^*(ab) = \\
&\delta(\delta^*(a), b) = \\
&\delta(\delta(\delta^*(\varepsilon), a), b) = \\
&\delta(\delta(l_0, a), b) = \\
&\delta(l_1, b) = \\
&l_3
\end{aligned}
$$

Since $l_3 \in F$, we can conclude that u is accepted by M, i.e. $A(u) = \top$. Figure 4.2 shows in bold the transitions taken in a graphical way. Contrastingly, for string v, $\delta^*(v) = l_1 \notin F$, therefore $A(v) = \bot$ and the string is not accepted by the DFA M. This run is graphically depicted in Figure 4.3.

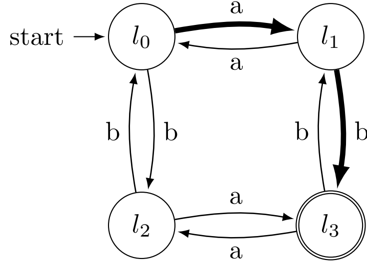

Figure 4.2: Transitions for input ab

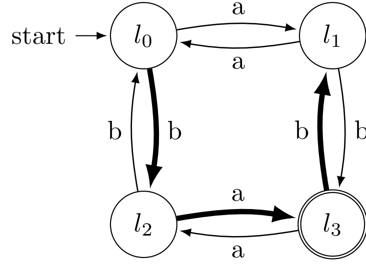

Figure 4.3: Transitions for input bab

4.3 Register Automata (RA)

Before formally defining register automata we detail relevant concepts. Let $\mathcal{D}$ be a domain of data values and $\mathcal{A}$ be a set of action symbols, each with a corresponding arity of elements from $\mathcal{D}$. A *parameterised input* is a symbol of the form $\alpha(p_1, p_2, \ldots, p_n)$, where $\alpha \in \mathcal{A}$ with arity n and $p_1, p_2, \ldots, p_n \in \mathcal{D}$. The simplified notation $\overline{p}$ is used to refer to $p_1, p_2, \ldots, p_n$. A finite sequence of parameterised inputs is a *data word.* A *data language* is a set of data words closed under permutations on $\mathcal{D}$. Data languages are strictly more expressive than regular languages as their alphabet set can be infinite if $\mathcal{D}$ is infinite. This allows for a better expression of systems that accept parameterised inputs, such as function calls with arguments. We additionally define some important RA components. A *register* is a variable which stores values from $\mathcal{D}$. A *guard* g is a propositional formula over registers and values from $\mathcal{D}$.

DEFINITION 2 *A register automaton M is a tuple $(L, l_0, \Sigma, X, \Gamma, F)$, where:*

- *L is a finite set of locations*
- *$l_0 \in L$ is an initial location*
- *Σ is an alphabet of parameterised inputs*
- *X a function that maps each location to a finite set of registers, with $X(l_0) = \emptyset$*

- Γ *is a set of transitions of the form* $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle$*, where* $l, l' \in L$*,* $\alpha(\overline{p}) \in \Sigma$*,* g *is a guard over* $\overline{p}$ *and* $X(l)$*, and* π *is a mapping from* $X(l')$ *to* $X(l) \cup \overline{p}$ *(every register in* $X(l')$ *is assigned a value from* $X(l) \cup \overline{p}$*)*
- $F \subseteq L$ *is a set of final/accepting locations*

Let us now define the formal semantics of RAs. We start by defining a *state* as the pair $\langle l, \nu \rangle$, where $l \in L$ and $\nu : \mathcal{X}(l) \to \mathcal{D}$ is a valuation of the set of registers at location l. The initial state of an RA is the pair $\langle l_0, \emptyset \rangle$. A *step* has the form $\langle l, \nu \rangle \xrightarrow{\alpha(\overline{p}), g, \pi} \langle l', \nu' \rangle$ and transfers the RA from state $\langle l, \nu \rangle$ to $\langle l', \nu' \rangle$ if $\langle l, \alpha(\overline{p}), g, \pi, l' \rangle \in \Gamma$ and g is true for $\langle l, \nu \rangle$. Let $w = \alpha_1(\overline{p}_1)\alpha_2(\overline{p}_2) \ldots \alpha_n(\overline{p}_n)$ be a data word over Σ. A run of the register automaton M over the data word w is a sequence:

$$\langle l_0, \nu_0 \rangle \xrightarrow{\alpha_1(\overline{p}_1), g_1, \pi_1} \langle l_1, \nu_1 \rangle \ldots \langle l_{n-1}, \nu_{n-1} \rangle \xrightarrow{\alpha_n(\overline{p}_n), g_n, \pi_n} \langle l_n, \nu_n \rangle,$$

such that $\forall_{i \in [1,n]} : \langle l_{i-1}, \alpha_i(\overline{p}_i), g_i, \pi_i, l_i \rangle \in \Gamma$. A run is accepting if $l_n \in F$.

4.3.1 Data Word Acceptance Example

Let M be a RA with alphabet $\Sigma = \{in(p), out(p)\}$ with all parameterised inputs with arity 1 and over the domain of the natural numbers $\mathbb{N}$. M implements the language that accepts any prefix of the infinite data word comprising of alternating the parameterised symbols $in(p)$ and $out(p)$ such that the data value p in $out(p)$ matches the data value of the preceding input $in(p)$. Figure 4.4 shows a graphical representation of this RA that omits any non-accepted transitions for conciseness. As such, $F = \{l_0, l_1\}$.

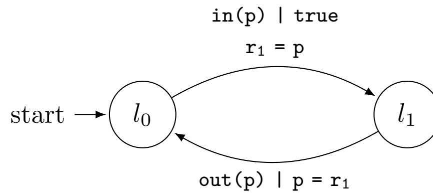

Figure 4.4: RA that accepts alternating $in(p)$ and $out(p)$ with matching p values

Let us now consider the example data words a = "$in(1)\,out(1)$", b = "$out(1)$", and c = "$in(1)\,out(2)$". To verify the acceptance of a data word by M we need to check if the run over that data word ends in a state $l \in F$. For data word a we have:

$$\langle l_0, \emptyset \rangle \xrightarrow{in(1), true, r_1 = 1} \langle l_1, \{r_1 = 1\} \rangle \xrightarrow{out(1), r_1 = 1, \varnothing} \langle l_0, \{r_1 = 1\} \rangle$$

Since all above steps are valid and $l_0 \in F$, we can conclude that a is accepted by M. Contrastingly, for data word b, there is no step starting at state $\langle l_0, \varnothing \rangle$ for the parameterised input $out(1)$. As such, this data word is not accepted by M. Lastly, for data word c the step $\langle l_0, \emptyset \rangle \xrightarrow{in(1), true, r_1 = 1} \langle l_1, \{r_1 = 1\} \rangle$ exists, however the step $\langle l_1, \{r_1 = 1\} \rangle \xrightarrow{out(2), r_1 = 2, \varnothing} \langle l_0, \{r_1 = 1\} \rangle$ is not valid as the guard $r_1 = 2$ is not satisfied since the initial state contains the valuation $r_1 = 1$. As such, this data word is also not accepted by M.

4.3.2 Expressiveness of RAs

RAs are strictly more expressive than DFAs. An example of this is representing languages that require counting. Let us consider the language over $\Sigma = \{a, b\}$ of strings with exactly the same

amount of as and bs. Due to the pumping lemma, DFAs would not be able to represent this language as they would require an infinite amount of states to keep track of the number of as and bs seen. RAs, on the other hand, are able to save these values in their registers. Figure 4.5 shows an example RA that accepts such a language.

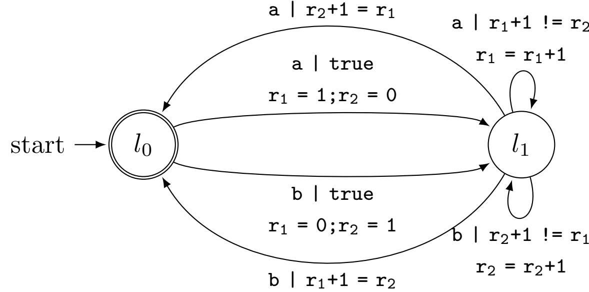

Figure 4.5: RA that accepts strings with the same amount of as and bs

4.4 Active Automata Learning

Active Automata Learning is a collection of algorithms that infer the behaviour of a system through interactions. In particular, they do so while providing minimality and correctness guarantees about the formal model inferred. We now explore the relevant active learning frameworks and algorithms.

4.4.1 MAT Framework

The Minimally Adequate Teacher framework [Ang87] is an active automata learning framework where a learner queries a teacher that is omniscient in regards to the system being learned. Formally, interactions are modelled as queries between a learner and a teacher where The learner generates their model by asking queries of the following types:

- **Membership:** The learner gives the teacher an input word and the teacher responds with the output word of the system for the provided input.
- **Equivalence:** The learner gives the teacher a hypothesis model representing the system and the teacher either confirms that the model is correct (it is equivalent to the system) or, otherwise, provides a counterexample input such that the output of the hypothesis for that input is different than the output of the system for that same input.

A visual representation of this framework can be seen in Figure 4.6.

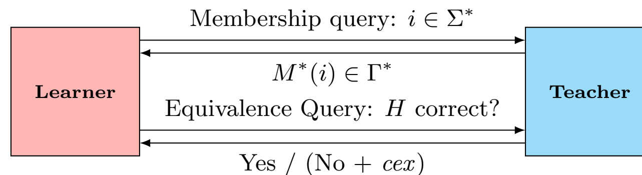

Figure 4.6: The Minimally Adequate Teacher framework.

4.4.2 L* Algorithm

The L* Algorithm [Ang87] is an algorithm under the MAT framework for learning a minimal DFA. It uses an observation table (S, E, T), where S, E are non-empty finite sets of strings and T is a

mapping $((S \cup S \cdot \Sigma) \cdot E) \rightarrow \{\top, \bot\}$, where $\top$ and $\bot$ represent the acceptance or rejection of the string by the system under learning (SUL), respectively. The table structure can be represented visually as shown in Table 4.2.

	E
S	
$S \cdot \Sigma$	

Table 4.2: Structure of an observation table

We use the notation $row(a)$ to mean the sequence of T values for all strings $a \cdot e$, where $e \in E$. This corresponds to the row indexed with string a in the visual representation of the table. We allow ourselves the abuse of notation $row(a) \in \Sigma$ to mean that the row indexed by a is present in the Σ section of the table.

We are able to convert an observation table into a corresponding model if the table is in a format compatible with the syntax and semantics of a DFA. This is guaranteed through the closure and consistency properties.

An observation table is *closed* when $\forall s_1 \in S \cdot \Sigma : \exists s_2 \in S : row(s_1) = row(s_2)$. Informally, this means that for every row in $S \cdot \Sigma$, there must be a row in S with the same cell values. To make a table closed, the rows in $S \cdot \Sigma$ not in S are added to S and the table is extended as appropriate. The pseudocode algorithm is described in Algorithm 1.

Algorithm 1: Make observation table closed

```
for row in S·Σ do
    if row not in S then
        S ← S ∪ {row};
    update S·Σ with new S;
```

An observation table is *consistent* when $\forall s_1, s_2 \in S, a \in \Sigma : row(s_1) = row(s_2) \rightarrow row(s_1 \cdot a) = row(s_2 \cdot a)$. To make a table consistent, the $a \in \Sigma$ for which $row(s_1 \cdot a) \neq row(s_2 \cdot a)$ and the $e \in E$ such that $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$ are found, the string $a \cdot e$ is added to E and the table is extended as appropriate. The pseudocode algorithm is described in Algorithm 2.

Algorithm 2: Make observation table consistent

```
for s1 ∈ S, s2 ∈ S, a ∈ Σ do
    if row(s1) = row(s2) and row(s1 · a) ≠ row(s2 · a) then
        let X be the set of elements from E for which rows s1 and s2 differ;
        for e ∈ X do
            E ← E ∪ {a · e};
        extend S and S·Σ with new values of E;
```

When an observation table is closed and consistent, we can define a minimal DFA where:

$$Q = \{row(s) : s \in S\} \qquad F\{row(s) : s \in S \wedge T(s) = 1\}$$
$$q_0 = row(\varepsilon) \qquad \delta(row(s), a) = row(s \cdot a)$$

The algorithm then works as follows. First, S and E are initialised to ε, membership queries are made for ε and all $a \in \Sigma$, and an initial observation table is constructed. The table should then be made closed and consistent (if it is not already) and the hypothesis representing the table should be provided to the teacher for an equivalence query. If a counterexample is provided, the table is extended by adding the counterexample to S along with all of its prefixes. The algorithm loops back to making the new table close and consistent and terminates when the equivalence query accepts the hypothesis provided.

Let us now see an example run of this algorithm. The DFA to be learned is the one shown in Figure 4.7.

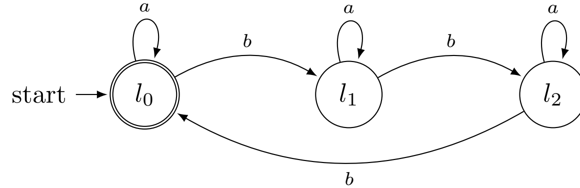

Figure 4.7: DFA that accepts strings with a number of b multiple of 3

We first start by initialising our observation table (Table 4.3). We can see that the table is not closed as $row(b) \notin S$. As such, we add b to S and update $S \cdot \Sigma$ to ensure the table is consistent (Table 4.4). A hypothesis is then generated from this table (Figure 4.8) which is provided to the teacher for an equivalence query.

	ε
ε	1
a	1
b	0

Table 4.3: Initial table

	ε
ε	1
b	0
a	1
ba	0
bb	0

Table 4.4: Closed and consistent

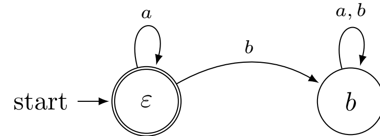

Figure 4.8: First hypothesis

The teacher then replies with the counterexample bbb, which is accepted by the system but rejected by our hypothesis. We now extend S with bbb and its prefixes bb and b (Table 4.5). The new observation table is closed but not consistent, as $row(b) = row(bb)$ but $row(b \cdot b) \neq row(bb \cdot b)$. In our case, we have that $T(b \cdot b \cdot \varepsilon) \neq T(b \cdot b \cdot \varepsilon)$ As such, we add $b \cdot \varepsilon = b$ to E and extend the table (Table 4.6). The new table is both closed and consistent, and the new hypothesis can be created (Figure 4.9). This hypothesis now matches our SUL, the equivalence query is accepted and our learning algorithm terminates.

	ε
ε	1
b	0
bb	0
bbb	1
a	1
ba	0
bba	0
bbba	1
bbbb	0

Table 4.5: Extended with cex

	ε	b
ε	1	0
b	0	0
bb	0	1
bbb	1	0
a	1	0
ba	0	0
bba	0	1
bbba	1	0
bbbb	0	0

Table 4.6: Consistent

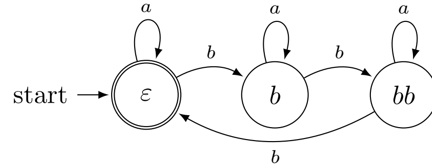

Figure 4.9: Final hypothesis

Let us now discuss correctness, minimality, and termination. If termination is guaranteed, the final hypothesis will be correct as the teacher is assumed to respond correctly to both membership and equivalence queries. Let M be a minimal DFA representing the SUL. The number of unique rows in S must always be less or equal to the number of states in M, as each row in S describes a necessary distinguishing state in the system. As such, if the algorithm terminates, the final hypothesis is guaranteed to be minimal. Every time the teacher rejects a hypothesis in an equivalence query, the provided counterexample must reach a state not present in the current hypothesis, and the analysis of such counterexample always leads to the addition of a row to S and therefore the increase of the number of states in our hypothesis by at least 1. Since every iteration of the algorithm generates at least one more state and the algorithm will always produce a final hypothesis with a finite number of states as shown by the minimality property, we guarantee termination.

The L* algorithm was the first formalised automata learning algorithm, however subsequent learning algorithms (KV [KV94a], TTT [IHS14], ...) achieve the same goal through more efficient tree-based data structures instead of an observation table. The L* algorithm was also extended to RAs, leading to the algorithm SL* [Cas+14] which is able to learn RAs parameterised on a particular theory (a set of guard operations over the data domain). This year, the new algorithm SLλ [Die+24] was introduced, which was inspired in the tree-based data structures used in the KV and TTT algorithms.

4.4.3 C$\mathcal{E}$AL Framework

The MAT framework has some practical limitations. The first one is the assumption of the existence of an omniscient teacher, which often does not hold in practice. This is not an issue in particular in regard to membership queries, as they can be answered directly by the SUL, however equivalence queries are much harder to implement. Most algorithms implement equivalence queries as a series of generated membership queries answered directly by the SUL until some level of confidence or a maximum number of queries is reached. However, system queries are often the bottleneck of performance and one of the main limitations in learning large real-world systems.

Another issue with the MAT framework concerns the inability to handle conflicts, which are formally defined as query answers which contradict previous queries in a way that cannot be expressed by a model of the target class. An example of this is the system initially accepting an input word but later rejecting it. This often occurs either due to unpredictable noise or purposeful system updates. Because there is no formal way of deciding which answer to choose, most MAT algorithms simply crash in the presence of a conflict, leading to all learning progress needing to

be restarted from scratch. Often practical tools will repeat the same query a certain amount of times and select the most common answer, but not only does this increase the learning time to often unfeasible amounts, it also does not fully prevent conflicts from occurring. Additionally, the necessary number of repeated queries is not a straightforward value to predict as it is intrinsically connected to the SUL and its environment, and both under and overestimating can lead to very time-consuming learning runs that make no progress.

As such, the C$\mathcal{E}$AL framework is presented as an alternative to MAT. C$\mathcal{E}$AL completely removes the teacher abstraction and elevates the SUL to a first-class citizen in the learning process. It then introduces a Reviser entity with whom both the Learner and the SUL solely interact. The Reviser's main responsibilities are to store the SUL's observed responses like a cache and to manage the conflict-handling strategy for the Learner. A visual representation of this framework can be seen in Figure 4.10.

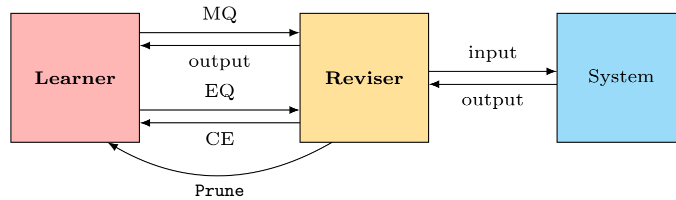

Figure 4.10: The C$\mathcal{E}$AL framework.

The fundamental knowledge-base of the Reviser is in the format of an *observation tree*. This data structure has the format of a tree whose root node corresponds to the empty string ε and each transition out of a node is labeled with an alphabet input symbol and ends in a node corresponding to the concatenation of the origin node string and the transition label. Each node then stores the response of the system to its corresponding string. Figure 4.11 shows an example of an observation tree for the DFA described in Figure 4.7.

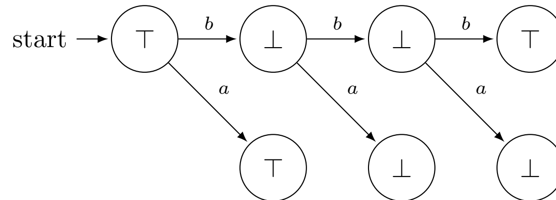

Figure 4.11: Example of Observation Tree

An example of a conflict for the above observation tree is the input-output pair ("bb", $\top$). Figure 4.12 highlights the conflicting nodes and transitions in red. To update the tree to be conflict-free and reflect the new input response, the highlighted nodes and transitions are removed and a new node is inserted reflecting the changed behaviour while preserving non-affected knowledge.

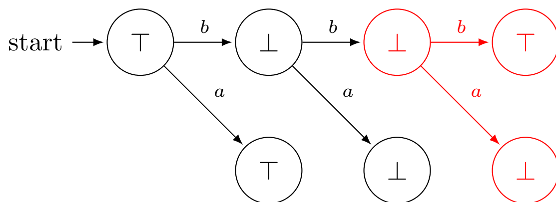

Figure 4.12: Nodes and transitions affected

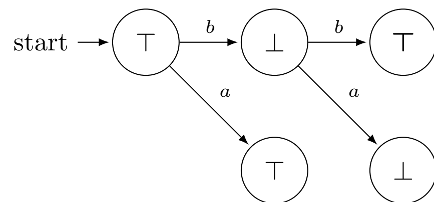

Figure 4.13: Updated observation tree

Once the observation tree is updated, the learner needs to be reset to a conflict-free state. This is done by sending a `Prune` signal to the Learner to restart the learning process. Because the Learner interacts solely with the Reviser, all the queries unaffected by the conflict are cached in the observation tree and there is no need to query the system directly for these answers. This is key as system queries are one of the main bottlenecks during automata learning.

The user can also adapt the Reviser to decide when a query answer can be labelled as a conflict. In use cases where the system frequently evolves, the *most recent* update strategy is preferred. If instead the main source of conflicts is due to noise, the *most frequent* strategy is better suited.

One key achievement of C$\mathcal{E}$AL is the fact that the Learner never needs to face any conflicts as they are all caught by the Reviser and handled there directly. As such, this framework is compatible with all MAT state-of-the-art learning algorithms. However, one limitation of the C$\mathcal{E}$AL framework was that it only supported reactive systems, in particular Mealy Machines, hence the need to extend the framework for compatibility with this project's use case. This extension is a continuation of my previous research at UCL, as the C$\mathcal{E}$AL framework was introduced by our group, work which I contributed to for the past 2 years.

Chapter 5

Solution

In this chapter, the implementation of the proof-of-concept tool for specification mining is detailed[1]. This is done in multiple sequential stages corresponding to the iteration cycles of the project development. Firstly we describe and motivate the chosen interface for learning, the iterator. The subsequent section focuses on inferring a DFA representation of a bounded iterator at different levels of abstraction. Next, we talk about the necessary changes to extend the tool to additionally support RA inference. Then, the learning of unbounded iterators is explored. We then move on to explain how C$\mathcal{E}$AL was extended and modified to support manual changes to the learned model. Lastly, a brief description of the UI and a running example of the tool in practice is provided.

5.1 From Specifications to the Iterator Interface

One of the main difficulties in inferring specifications through system interactions is the inability to distinguish the "intent" of the system developer from the actual implementation. There is no way of knowing if the implemented behaviour accurately represents the true requirements of the system, leading to a lack of a faithful source of truth for input queries. To circumvent this, we make the assumption that any execution trace that does not trigger any errors or exceptions is correct unless proven otherwise by the user directly. As such, the first generated specification needs only to match the behaviour of the system implementation.

The system selected for specification mining was the Java `Iterator` interface[2], as it provides a concise but rich collection of methods with interesting relationships. An `Iterator` is an object that can be used to sequentially loop through a collection of elements. It supports four methods: `hasNext`, `next`, `remove`, and `forEachRemaining`. For simplicity, the behaviour of the last method (`forEachRemaining`) was not considered. The method `hasNext` returns true if there are any elements in the collection that have not yet been iterated through and false otherwise. The method `next` returns the next element in the iteration if it exists, otherwise it throws an exception. Finally, `remove` deletes the element returned by the last call to `next` from the underlying collection or throws an exception if the iterator is not in the correct state: either `next` has not yet been called or `remove` has already been called since the last call to `next`.

The descriptions of the methods allow us to determine some correct and incorrect behaviour. Calling `hasNext` should always be allowed as it only checks the current state of the iterator without altering it. Calling `next` should only be allowed if there is a next element, i.e. if `hasNext` returns

[1] Link to repository: `https://github.com/rasofema/fyp-specification-miner`
[2] `https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html`

true. Lastly, calling `remove` should only be allowed once after a call to `next`.

To now be able to learn a specification of an Iterator, we need to define an oracle - a mechanism to convert abstract input sequences into concrete system interactions and then determine if the observed behaviour is correct or incorrect. A simple way of abstracting the method calling into input symbols is by simply using the names of the methods as our alphabet. Determining the correctness of the behaviour is, however, less straightforward. Because the Iterator interface throws an exception for most incorrect traces, these can be used as our labelling mechanism for traces. This, in general, is a good approach, as developers use exceptions to warn callers that the execution did not happen as *intended*. Nonetheless, the labelling of traces should be tailored to each particular use-case as necessary. With the target system selected and a strategy for classifying behaviour, we are now ready to implement our oracle and start learning.

5.2 Mining with DFAs

5.2.1 Initial Implementation

The first version of the project focused on implementing a simple program that was able to learn some of the behaviour of an iterator. The first methods selected for learning were `hasNext` and `next`. Because the iterator interface is an object of fixed size based on an underlying iterable that should not be changed during iteration, the emulation of iterators of different sizes had to be done based on information about their state. In particular, the function `hasNext` was treated as an indicator of the current state of the iterator, which would allow us to observe if `next` was an allowed operation or not. An overarching oracle was then used to mimic this “mutable” iterator.

To allow the state of the iterator to be determined by the return value of `hasNext`, the two return options - `hasNextTrue` and `hasNextFalse` - were added as possible values to our learning alphabet. That is, if `hasNextTrue` was provided as an input, the iterator would be changed to one which had the return value `true` for the function `hasNext`, but if `hasNextFalse`, the iterator would be changed to one which had the return value `false` for that same function. In practice, this was implemented by, on a `hasNextTrue`, creating a new iterator from a singleton list, and on a `hasNextFalse`, calling `next` while `hasNext` returned `true`. For both these inputs, the oracle's response was always `true` If the input `next` was provided, then the current iterator would have the respective function called and the oracle would respond with `true` except if an exception was thrown, in which case the oracle would respond with `false`. Algorithm 3 shows the pseudocode for the implementation of the oracle.

Algorithm 3: Initial implementation of the iterator oracle

```
switch input symbol do
    case hasNextTrue do
        if not iterator.hasNext() then  iterator = new SingletonIterator(elem) ;
    case hasNextFalse do
        while iterator.hasNext() do  iterator.next() ;
    case next do
        try:
            iterator.next()
        catch Exception:
            return False
return True
```

The main learning function used LearnLib to create an L* learner object that took as input the newly implemented iterator oracle and an alphabet with the symbols `hasNextTrue`, `hasNextFalse`, and `next`, and a simple equivalence oracle. The function then looped in generating hypotheses and finding counterexamples until no more counterexamples were found. The final learned DFA is shown in Figure 5.1. We can see that l_0 is the state associated with `hasNextFalse` and l_1 is associated with `hasNextTrue`. We can also see that `next` is only accepted when in the `hasNextTrue` state, as taking the `next` transition when in the `hasNextFalse` state leads to the non-accepting sink state. This therefore constitutes our unwanted behaviour for the iterator. The documentation website for the Java `Iterator` interface confirms this is the intended behaviour: "***hasNext()*** *- returns true if the iteration has more elements. (In other words, returns true if* ***next()*** *would return an element rather than throwing an exception.)*"[3].

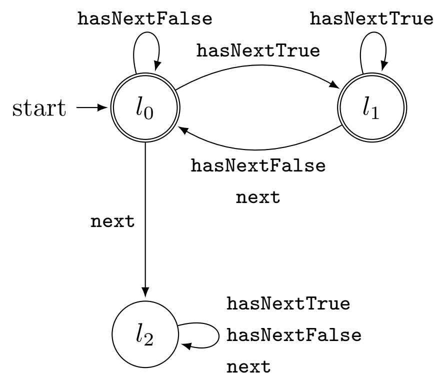

Figure 5.1: First DFA learned using `hasNextTrue`, `hasNextFalse`, and `next`

5.2.2 Incorporating the Method `remove`

The subsequent version focused on adding the iterator method `remove`, as this is the only remaining iterator method left.

The previous implementation of `hasNextTrue` used a singleton list from the `Collections` util. However, this type of list generates a `SingletonIterator` [4], which is immutable and therefore does not support the `remove` method. As such, the oracle's behaviour for `hasNextTrue` was

[3] `https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html#hasNext--`

[4] `https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/iterators/SingletonIterator.html`

changed to instead use an ArrayList containing a dummy element and generate an iterator. In addition, the previous behaviour of `hasNextFalse` used the method `next`, which was a problem since the behaviour of `remove` depends on `next` having been previously called. This is listed in the Java `Iterator` documentation: "*`remove()` - removes from the underlying collection the last element returned by this iterator (optional operation). This method can be called only once per call to `next()`.*"[5]. As such, the `hasNextFalse` behaviour was changed to instead create an empty `ArrayList` and generate a fresh empty iterator. The corresponding pseudocode for the oracle is shown in Algorithm 4.

Algorithm 4: Incorporation of the method `remove`

```
switch input symbol do
    case hasNextTrue do
        if not iterator.hasNext() then
            List lst = new List();
            lst.add(elem);
            iterator = lst.iterator();
    case hasNextFalse do iterator = new List().iterator();
    case next do
        try:
            iterator.next();
        catch Exception:
            return False
    case remove do
        try:
            iterator.remove();
        catch Exception:
            return False
return True
```

The newly obtained DFA is shown in Figure 5.2. This DFA mainly differs from Figure 5.1 by having an extra state, l_2, which corresponds to the case where the method `next` has been called on an iterator in a `hasNextTrue` state, but `remove` has not yet been called. By calling the method `remove`, the DFA transitions to the initial state. This is the only `remove` transition that does not end in the sink state l_3, therefore describing the only acceptable use of the `remove` method.

[5] https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html#remove--

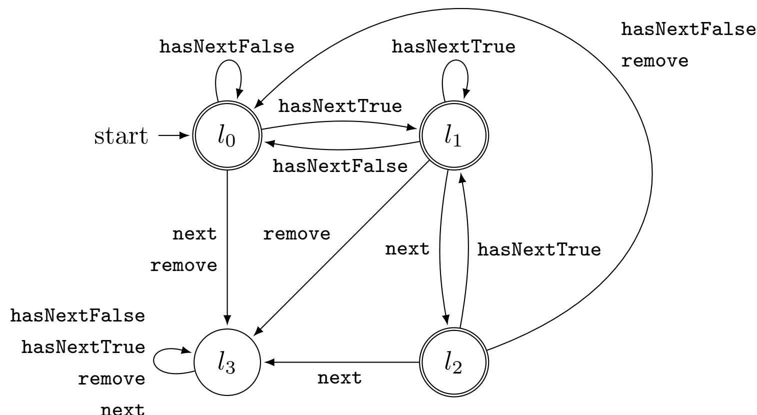

Figure 5.2: First DFA learned using `hasNextTrue`, `hasNextFalse`, and `next`

5.2.3 Incorporating the Method `add`

To fully show the behaviour of iterators of different sizes, a final method was necessary: `add`. This method is not available for iterators but instead for iterables. As such, a list-like class which implemented the interface `Iterable` was created. Introducing this new method meant that now the iterable could have any size and that the number of `next` method calls allowed would depend on the number of elements added, which would require counting. This means that this type of behaviour is not possible to be represented by a DFA, as shown by the pumping lemma. As such, the iterable class created was bounded such that, when calling the method `add`, if the number of elements in the bounded list matched the predefined bound (therefore being full), the bounded list would throw an exception. The target class then becomes tractable to be learned as a DFA, as the boundedness allows for a finite state representation.

The implementation of the oracle behaviour for `add` is the following: first, the method `add` is called on the bounded list with a dummy element, and then a new iterator is generated. It is necessary to use a new iterator because changing the underlying iterable during iteration is marked as undefined behaviour[6]. This newly generated iterator is reset, meaning that it does not know of the previous iterator's pointer position obtained via the calls to `next`. As such, an additional integer class field which tracks the current position in the iterator was added and the behaviour for `next` and `remove` adjusted: for every call to `next` the iterator position increases by one, while every call to `remove` decreases it. With now a tracker for the latest iterator position, whenever generating a new iterator, the method `next` is called until the previous pointer position is reinstated.

The addition of the method `add` required a few additional changes to the `IterOracle`. Since the size of the iterator was now variable (being controlled by `add` and `remove`), the previous implementations for `hasNextTrue` and `hasNextFalse` no longer applied and instead these methods now matched their initial intuition: returning if the method `hasNext` called on the iterator returned `True` or `False`, respectively. The final oracle is described by the pseudocode in Algorithm 5.

The learned DFA containing the additional method `add` for an iterable with bound 1 is shown

[6] `https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html#remove--`

in Figure 5.3. This model now more accurately depicts the expected behaviour of the iterator. Initially, `hasNextFalse` is accepted, as there are no elements in the list. When `add` is called, a new element is added to the list and, as expected, `hasNextTrue` is accepted. In this state l_1, calling `add` leads the DFA to transition to our sink state. This matches our expected behaviour according to the predefined bound of 1 for the list. As there is now one element to iterate over, `next` can be called, leading to transitioning into state l_2, where `hasNextFalse` is now accepted. In this state, we are able to call `remove`, which eliminates our just-iterated-through element from the list and opens up the space for another element to be added. We can see that, not considering calls to `hasNext`, this model shows that the iterator of bound 1 only accepts actions following the cycle `add - next - remove`.

Algorithm 5: Incorporation of the method `add`

```
switch input symbol do
    case hasNextTrue do  return iterator.hasNext() ;
    case hasNextFalse do  return not iterator.hasNext() ;
    case next do
        try:
            iterator.next();
        catch Exception:
            return False
        iteratorPosition += 1;
    case remove do
        try:
            iterator.remove();
        catch Exception:
            return False
        iteratorPosition -= 1;
    case add do
        try:
            lst.add(elem);
        catch OutOfMemoryException:
            return False
        updateIterator();
return True
```

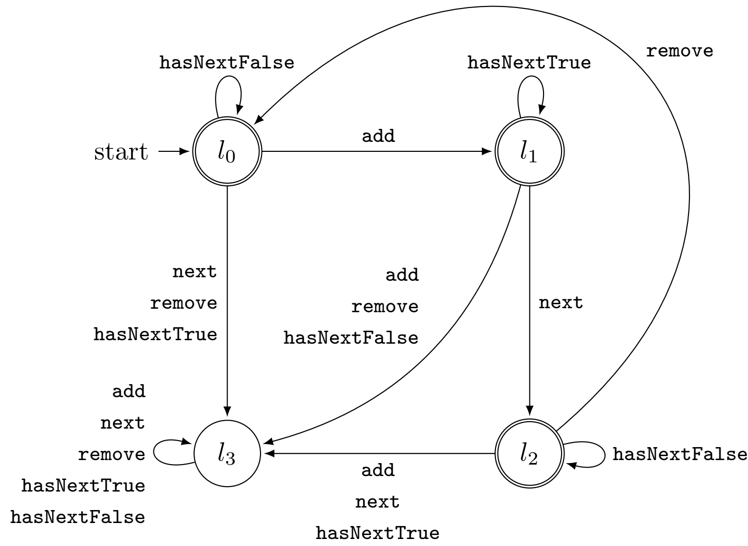

Figure 5.3: DFA learned for list of bound 1

An additional version of the iterable with bound 1 was learned where the list, instead of throwing an exception when trying to call `add` when full, silently fails, and the learned DFA is shown in Figure 5.4. The only differences are in states l_1 and l_2, where calling the method `add` is now accepted and does not change the state of the iterator, as expected.

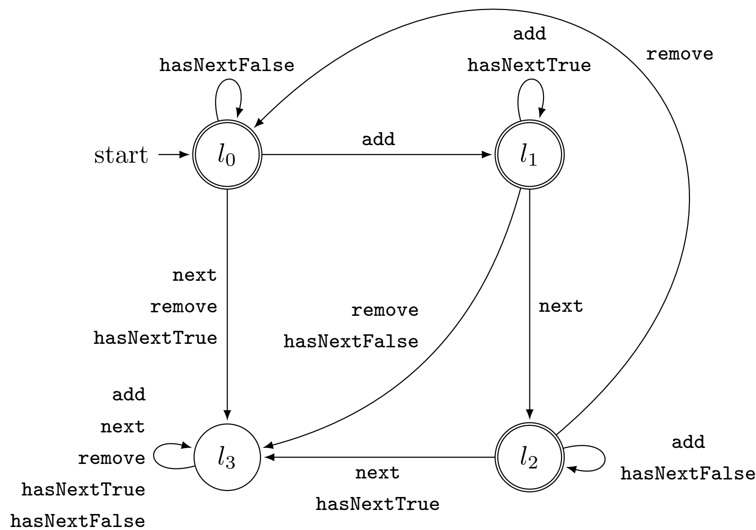

Figure 5.4: DFA learned for list of bound 1 and with silent failure for `add`

5.3 Mining with Register Automata

5.3.1 Bounded Iterators

Because of the limitations in representing the behaviour of unbounded list iterators with DFAs, learning with Register Automata (RA) was also implemented. In particular, RAs were selected as they are machines that allow for the learning of languages that require counting (ex. counting the number of elements in a list) and can also produce more compact representations of equivalent DFAs. Learning RA is available via the package `RALib`[7], which is an extension to `LearnLib`. However, the two packages had some incompatibilities. During the implementation of this project, RALib was using LearnLib's version `0.16.0`, which used AutomataLib's version `0.10.0`. However, the learning loop created previously for learning DFAs required at least version `0.17.0` of LearnLib and version `0.11.0` of AutomataLib, as the newer versions introduced a lot of refactoring and additional functionality not present in the previous versions and not compatible with their structure. As such, the first part of adding the extra functionality of learning RAs was refactoring RALib to use the latest version of LearnLib and AutomataLib: `0.18.0-SNAPSHOT` and `0.12.0-SNAPSHOT` respectively. RALib did later in the project update its version of LearnLib to `0.17.0`[8].

Once all three libraries were able to be imported without any conflicts, a first attempt at learning with RALib was made. The RALib instructions for learning the behaviour of a class suggested the use of their class `ClassAnalyser` via the command line and using a configuration file. The resulting model of this learning method using only the methods `add` and `hasNext` and a bound of 2 is shown in Figure 5.5. As can be seen by the model learned, ClassAnalyzer produces a verbose model where the output of each function is shown as a transition instead of being represented as part of the node acceptance. This is because the ClassAnalyzer tool was designed to take into account the input arguments and the output values for the represented behaviour (ex.: for checking if, when popping the elements of a stack, they match with the previously added). This way of representing the behaviour of the class is innate to the ClassAnalyzer and as such made this learning method inadequate for the intended purpose of the tool.

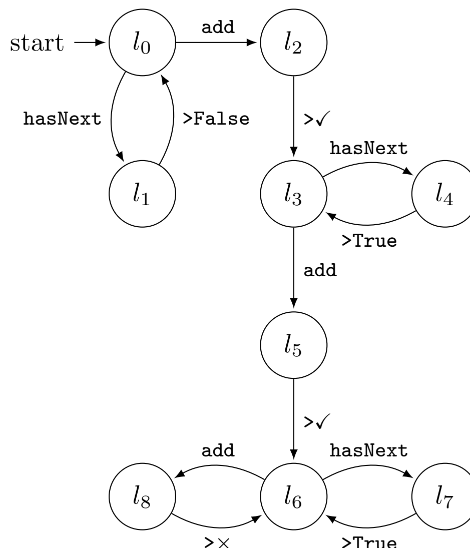

Figure 5.5: RA learned using ClassAnalyzer

[7] https://github.com/LearnLib/ralib
[8] https://github.com/LearnLib/ralib/commit/fe99734a4d1b26ca884aa781de13e947168b3f45

The subsequent RA learning method explored was to simply implement the learning loop from scratch as was done with DFAs. The RALib library was designed with the focus of learning alphabets with parameterised inputs, i.e. inputs that can take a finite number of arguments with a certain data type. An example of such type of inputs is the function call `push` for the Stack interface, which takes as argument an element of the type that the Stack allows. However, this use-case does not need to consider any input arguments and, as such, treats the parameterised input classes as simple alphabet constants. The rest of the learning loop follows the same strategy as for learning DFAs. The learned RA matches exactly with the DFA learned in Figure 5.4.

5.3.2 Unbounded Iterators

The ability to learn an unbounded iterator was then explored, and an additional type of iterator was defined: unbounded iterator with a bounded reading range. This type of iterator has the restriction of the pointer indicating the "next-to-read" element not being allowed to be at a distance greater than the defined bound from the end of the list. This means that the method `add` would only succeed if the difference between the size of the iterator and the "next-to-read" pointer is less than the predefined bound. In order for the RA to capture the values of these two variables, it makes use of the parameterised input arguments. These arguments are usually used for capturing the original input arguments to the method calls, however as these are not relevant to our use case we can use them as helper values that track the specified properties. The RALib is constrained to learning RAs with data languages that can be modelled by a restricted amount of theories [Cas+16]. One of these theories is the Integer Equality theory, in which the RA registers and input parameters can be compared equality-wise with each other and with integer constants.

The idealised compact model of an unbounded iterator with a reading range bound of 3 can be seen in Figure 5.6, where the only input symbols considered were `add` and `next`, each taking one argument, and the transitions to the failing sink state have been omitted for conciseness, meaning the depicted transitions are the only ones accepted. An additional input symbol was used, `init`, so that the registers r_2 and r_3 could be initialised with the value 0, as RALib only allows input parameters or the value of a register to be assigned to another register. The behaviour of this model is best understood by thinking of the registers as a stack, where $\mathtt{r_1}$ is at the top and $\mathtt{r_3}$ is at the bottom of the stack. The `add` transition from state l_1 to state l_2 should only be taken when there are no elements to be read, i.e. the iterator has no next element. It adds one element and records this by pushing a 1 into the register stack. Once in state l_2, adding more elements into the stack has the same behaviour, except there is now an additional guard to ensure that the bottom of the stack has not been reached with 1s (i.e. the reading bound has not been reached). If instead the function `next` is called, an element is popped from the register stack. Once only one element is left in the stack (i.e. the iterator is only able to call next once more), the transition to l_1 is taken. The depicted model shows the case of a reading bound of 3, however this is only an example, as equivalent models with different reading bounds can be obtained by simply adding/removing registers and updating the transitions accordingly.

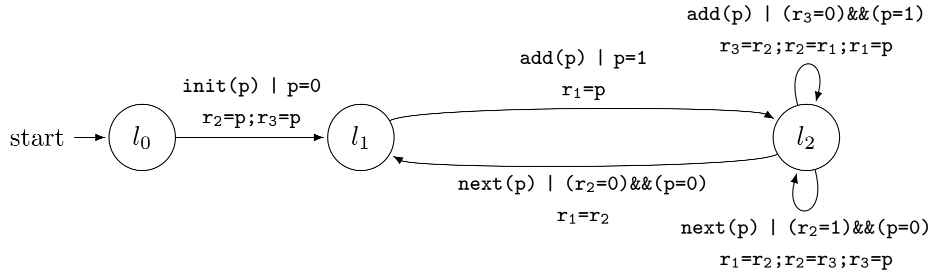

Figure 5.6: Idealised compact RA for an unbounded list with a reading bound of 3

To make a version of the idealised RA without a reading bound, a stronger theory is needed. In particular, this theory needs a way to keep track of the reading difference using the registers. Because calling `add` and `next` changes the reading difference in steps of 1, we need a theory that can track changes in exact steps. Such a theory is called an increment theory, which, in addition to the checks provided by the equality theory, can check for equality between a value and another value with a constant added to it. The full unbounded iterator RA that uses the increment theory can be seen in Figure 5.7, which, like for Figure 5.6, only shows the allowed transitions. This RA only requires two registers: $\texttt{r}_1$ tracks the number of elements in the iterator and $\texttt{r}_2$ the current "next-to-read" pointer. It ensures that next can never be called if either there are no elements in the iterator or if the "next-to-read" pointer is at the end.

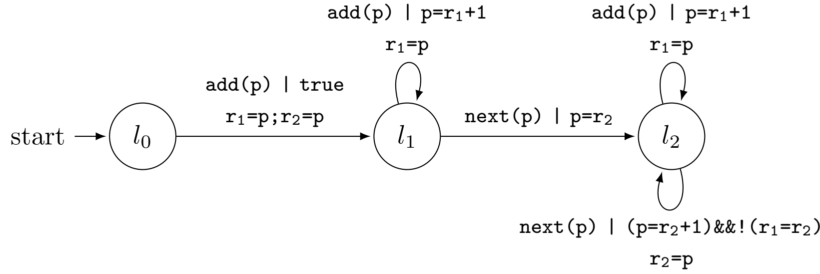

Figure 5.7: Idealised RA for an unbounded list no reading bound

Both these idealised RAs, however, were not able to be learned using RALib. The version with the reading bound only uses constant parameters, and as such the registers are not necessary and the library chooses instead to just create states. The RA learned with RALib for the idealised version of the unbounded iterator with a reading bound of 3 can be seen in Figure 5.8.

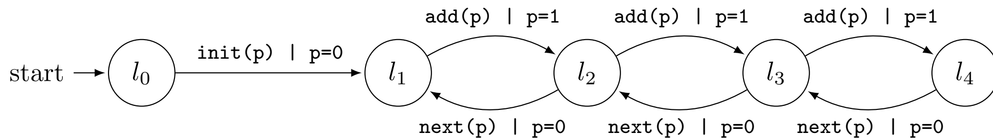

Figure 5.8: Learned "compact" RA for an unbounded list with a reading bound of 3

The learning of the unbounded version with no reading bound was also explored in practice, however the increment theory had only been implemented in an experimental branch[9], which diverged majorly from the main branch. The RA learned with RALib for the idealised version of the unbounded iterator with no reading bound can be seen in Figure 5.9. In this case, we can see

[9]`https://github.com/LearnLib/ralib/tree/succ-stable`

that the RA did not learn the full behaviour and did not generalise it either. Initially, this was believed to be due to RALib not allowing register-to-register comparison in the guard, preventing it from checking if the reading difference had been reached. However, an equivalent version of the ideal unbounded model with no reading bound was idealised that did not require such a constraint in the guard. This is shown in Figure 5.10. The unlearnability of the idealised models seems to be due to the limitations of the library, meaning that should be theoretically possible to obtain compact models of unbounded iterators. However, extending the current libraries to enable this aim would require quite a considerable amount of work.

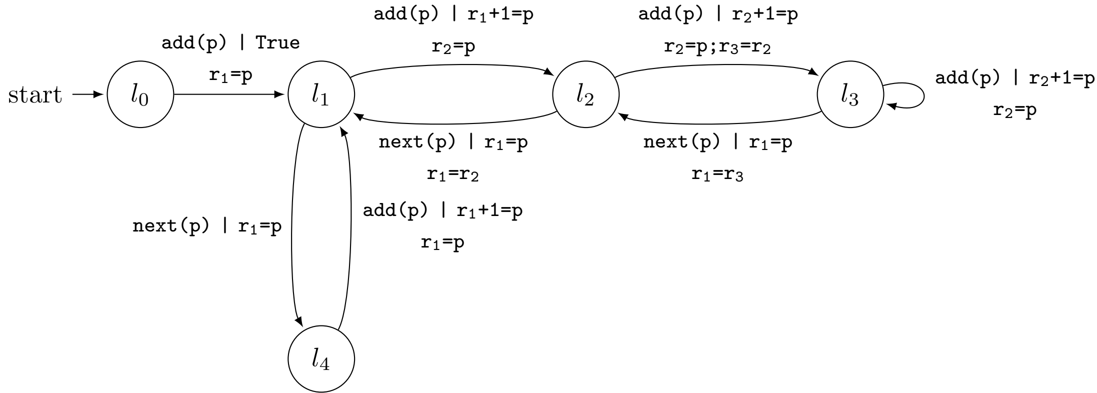

Figure 5.9: Learned RA for an unbounded list with no reading bound

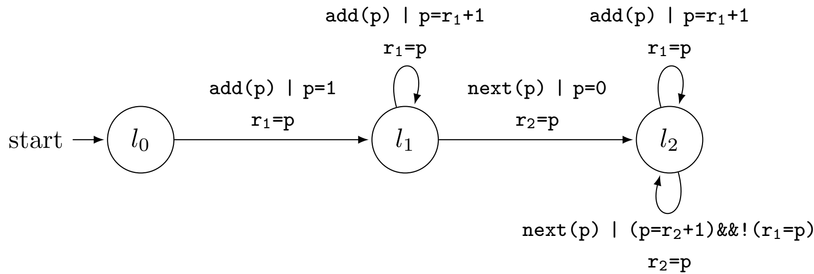

Figure 5.10: Second idealised unbounded with no reading bound, no register comparison in guards

5.4 Extending C$\mathcal{E}$AL and Using It to Accept User Input

So far, all the learning implementations have used the MAT framework. This traditional learning framework relies on determinism, where no system responses are able to contradict previously learned behaviour. However, for this tool to allow the user to overwrite previously learned behaviour, the learning algorithm must be able to accept counterexamples that conflict with previous system responses. The C$\mathcal{E}$AL framework is then useful, as it provides a framework for handling conflicts. However, C$\mathcal{E}$AL had been only implemented for Mealy Machines and not for DFAs or RAs. As such, the initial step in this extension was to enable the support of these two model types.

Firstly, the accepted automaton type was changed to any object implementing the AutomataLib `Output` interface. The `DFA` class already implemented this interface, so it was only necessary to add this interface to the `RegisterAutomaton` class and implement the relevant methods.

C$\mathcal{E}$AL takes as one of its arguments an `AdaptiveConstruction`, corresponding to the observation tree used by the Reviser to keep track of the observed system traces. The name adaptive

contrasts with incremental as this data structure is able to handle conflicts instead of simply recording information in an incremental way. The only implemented adaptive observation trees in the AutomataLib library were for Mealy Machines. As such, DFA and RA versions of both the incremental and the adaptive observation trees were added to AutomataLib and RALib. Once these changes were performed, two additional alterations were required. The first one involved slightly adjusting the input types of the `RegisterAutomata` class to allow for a common alphabet type with DFAs and allow the tool to support these two automata. Then, the RALib learning algorithms were made to extend the class `LearningAlgorithm` and C$\mathcal{E}$AL adapted to accept objects of type `MembershipOracle` to further enable the compatibility between libraries. After adding these classes, it was finally possible to use C$\mathcal{E}$AL to learn DFA and RA models of the iterator which, as expected, returned the same results as prior.

Both AutomataLib and RALib were now finally ready to be extended to accept user-provided counterexamples. Their goal is to overwrite the behaviour learned for a particular input by the algorithm, which for C$\mathcal{E}$AL is stored in the Reviser's observation tree. To therefore determine if a query response supersedes another for the same input, we need to keep track of the origin of each observation. As such, the enum `CexOrigin = {SUL, USER, UNKNOWN}` was introduced to label each tree node with the corresponding origin. When a node corresponds to a query answered directly by the system, the enum `SUL` is used. If however it comes from a manual counterexample provided by the user, `USER` is chosen. The enum `UNKNOWN` is only used when a node is created such that there is a path to the target input word but its acceptance is not yet known. For example, if an empty observation tree received the query ("ab", $True$), it would create two nodes from the root node, one for the input symbol a and another for b. The node associated with the word "ab" would be labelled as True, however it would not yet know the acceptance for the "a" node.

With the necessary tools for labelling observation origin set-up, the behaviour of both the DFA and RA adaptive trees was extended to implement the counterexample handling strategy. The behaviour for inserting a new observation is detailed in Algorithm 6. In particular, the logic for determining if an observation can be overwritten corresponds to the if condition in lines 7 and 8. Concisely, those lines ensure that overwriting is only allowed if the acceptance differs and either the node's answer came from the same origin or it did not come from the user. This ensures that only the user is able to overwrite a user-provided counterexample.

Algorithm 6: Inserting query answer into observation tree with origin label

```
1 Function insert(input, acceptance, origin):
2     currNode = root;
3     for symbol ∈ input do
4         if not currNode.hasChild(symbol) then currNode.create(symbol);
5         currNode = currNode.transition(symbol);
6     if currNode.acceptance == UNKNOWN then currNode.set(acceptance, origin);
7     else if currNode.acceptance != acceptance and
8     (currNode.origin == origin or currNode.origin != USER) then
9         currNode.set(acceptance, origin);
```

At this point, most functionality seemed to be working, but manually providing counterexamples was not working as expected. This was because, once the tree builder was updated and the `Reviser` restarted the learning, it was still using the system directly to check for counterexamples instead of the observation tree, therefore bypassing the user-provided counterexamples. As such,

the Reviser was adapted to instead always using the observation tree and only checking for system answers if the query was not in the tree. This could be done because the system's behaviour is non-changing. The original use case for C3AL was to consider scenarios where there is either noise or the system evolves. However, in our scenario, we can assume that the observation tree only contains correct system responses and, as such, does not need to check if the system responds with anything that contradicts previously learned behaviour.

All changes were then in place, and the tool was ready to accept manual counterexamples. When creating the `C3AL` learner it is necessary to provide a tree builder. The tool creates an `RaTreeBuilder` and a `DFATreeBuilder` on start and stores a reference to these objects as class attributes. These are then used when a new instance of `C3AL` is created, done when a new learning algorithm is created. By maintaining the previous tree builder we retain its cache and prevent the repetition of several queries, optimising the learning process. When a manual counterexample needs to be provided, it is passed onto both tree builders. By manually altering both tree builders we ensure the counterexamples provided are consistent for both DFA and RA learning algorithms.

5.5 Introduction of the User Interface

We used the Swing API for the user interface (UI) and restructured the code to follow a Model-View-Controller structure. The implemented classes were put in the `model` package, the main learning loop was put in the `controller` package alongside a main class that starts the application, and the UI was implemented in the `view` package.

The UI comprises a singular `JFrame`, which contains a main `JPanel` with all of the available functionality. An image of the UI can be seen in Figure 5.11. On the top left is the image of the currently learned model. Below the model is a drop-down menu of the available learning algorithms. On the bottom right are two buttons, one named `True` in a green background and one named `False` in a red background. These buttons work in conjunction with the drop-down menu above them, which allows the user to select an input from the available alphabet (in this case: `add`, `next`, `remove`, `hasNextTrue`, and `hasNextFalse`) and add it to the grey area above. The user can also remove elements from the grey area by simply clicking on them. Once the user is happy with their input sequence, they can click the `True` or `False` button to let the learning algorithm know they want this sequence to be considered as True or False, respectively. The learning algorithm will then add this user counterexample and update the learned model.

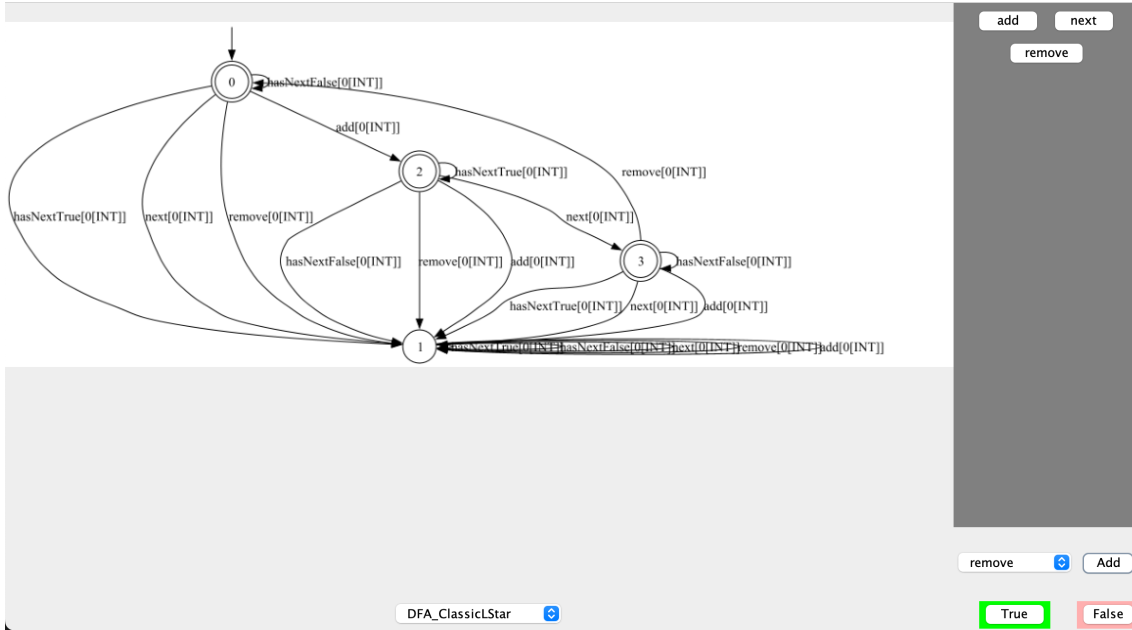

Figure 5.11: Screenshot of the UI

The following learning algorithms are supported:

- DFA Classic L*
- DFA Extensible L*
- DFA KV
- DFA TTT
- RaDT
- RaLambda
- RaStar

5.6 Specification Miner Example Run

This section will now describe an example run of the tool for an iterator of bound 1 using only the inputs `add`, `next`, and `remove`. Figure 5.12 shows the screen when the application first starts, which contains the first learned model of the iterator interface as shown in Figure 5.13.

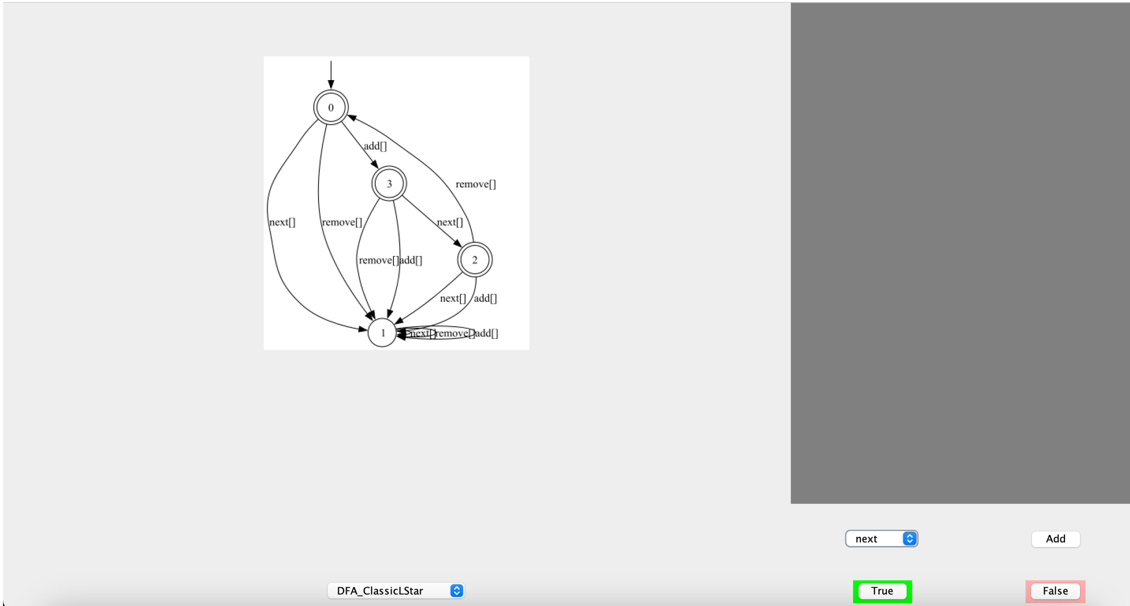

Figure 5.12: Initial screen

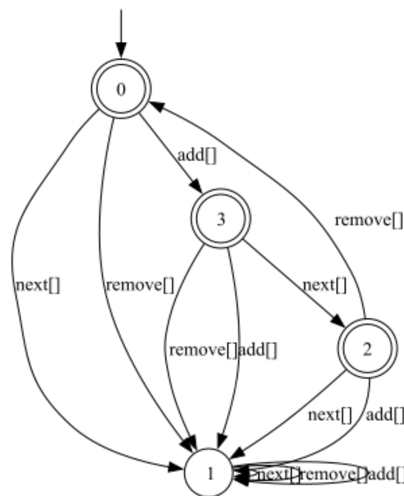

Figure 5.13: Initially learned DFA

The initial DFA matches exactly with the expected behaviour of the iterator, with a cycle for the sequence "`add next remove`". However, the user may decide that calling `remove` should only be accepted if, in an input sequence, it appears more than once. This would mean that "`add next remove`" should be rejected but nothing else (i.e. "`add`", "`add next`", "`add next remove add`" and others are still accepted). The user can therefore provide this counterexample by selecting the input symbols and adding them to the grey-box area using the dropdown menu and button shown in Figure 5.14. If a user, by mistake, inserts the wrong input symbol, they can just click on the inserted symbol in the grey-box area.

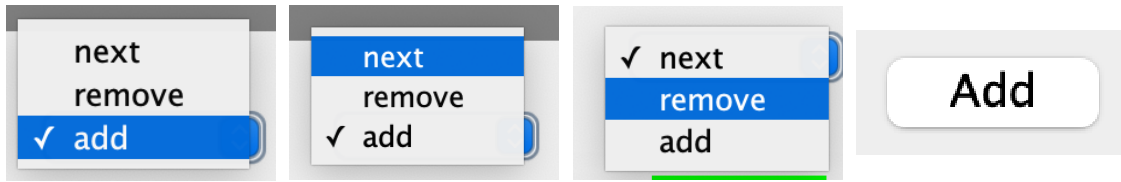

Figure 5.14: Buttons to manually create counterexample

The final counterexample for "`add next remove`" should then look like Figure 5.15. To provide the crafted input word as a counterexample to the tool, the user can simply click the button for the truth value they want the input word to have, in this case `False` (Figure 5.16).

Figure 5.15: Counterexample input word created

Figure 5.16: False counterexample button

The tool then processes the counterexample provided by the user and the model is updated, resulting in the DFA shown in Figure 5.17. As can be seen, indeed "`add next remove`" is now rejected by the DFA (state 4), with other remaining input words retaining their truth value.

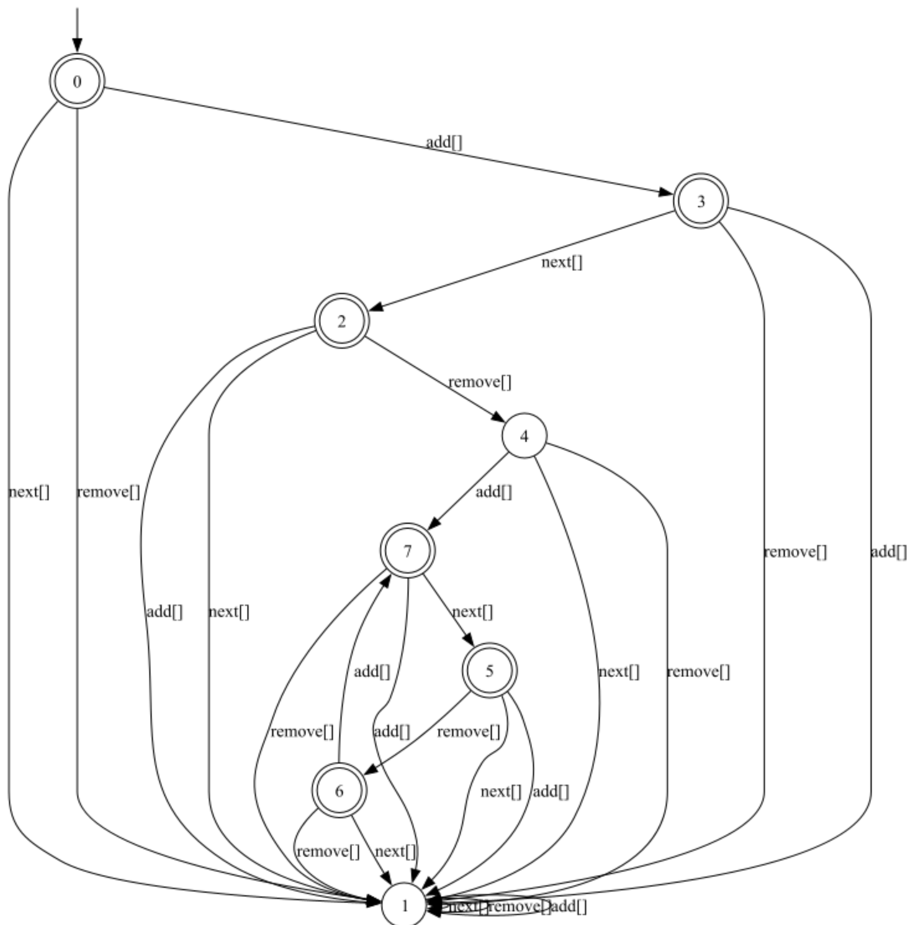

Figure 5.17: Newly learned DFA after providing ("add next remove", False) as a counterexample

If the user then decides they instead want, for example, to learn a Register Automata with the learning algorithm "RaDT", they can select it in the dropdown menu shown in Figure 5.18.

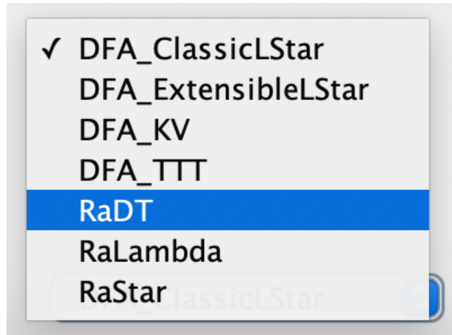

Figure 5.18: Changing learning algorithm

All the user's provided counterexamples are passed onto both the DFA and the RA learners and, as such, the user's progress is maintained independently of the learning algorithm and model used. We can confirm this by observing the learned RA depicted in Figure 5.19, which corresponds to the model shown right after selecting the new learning algorithm.

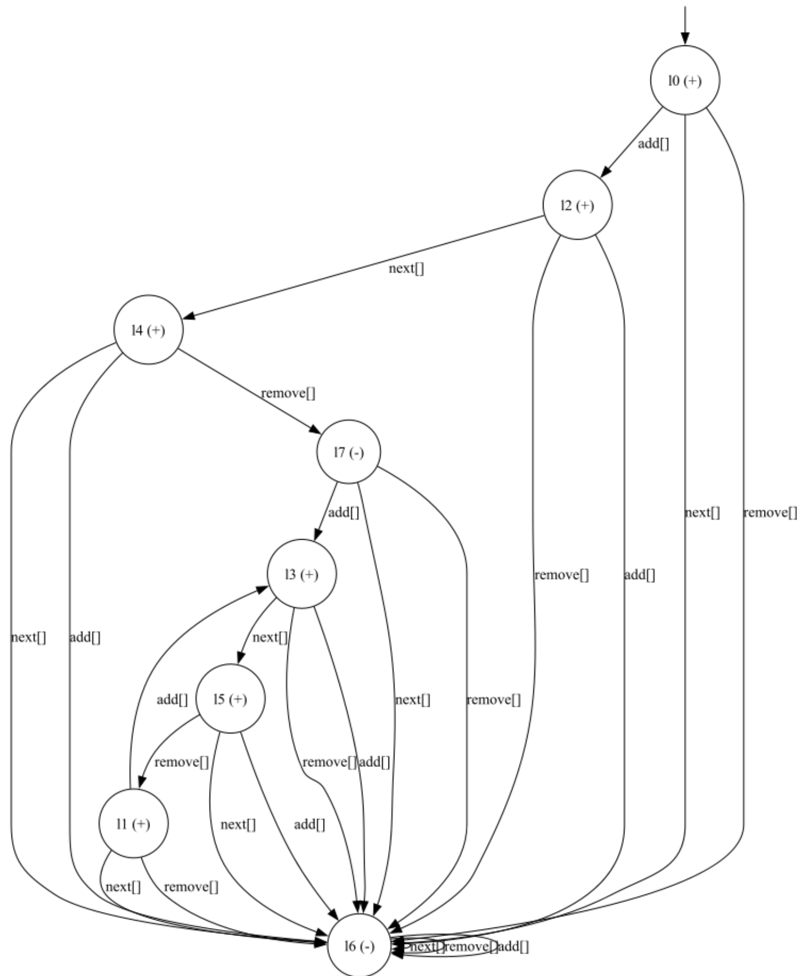

Figure 5.19: RA learned after changing learning algorithm

Chapter 6

Conclusion

6.1 Achievements

This project has achieved the goals it originally set out to do. A variety of state-of-the-art active automata learning methods have been employed to infer the specification of an Iterator class and successfully produced both DFAs and RAs that accurately explain the behaviour of the class in study. The active learning framework C$\mathcal{E}$AL has also been extended to provide the ability to manually change some of the behaviour described by the learned automata without requiring the restart of the learning process. Finally, a full-stack proof-of-concept tool was produced to demonstrate what future specification miners that exploit formal model inference and accept user-provided counterexamples could look like.

6.2 Evaluation

The outlined goals for this project have been achieved, however some limitations still remain. One key limitation was the inability to obtain a specification for an unbounded iterator. Although the iterator interface was an example target system, keeping track of particular property values is a common property of other systems, and, as such, not being able to currently learn these types of systems restricts significantly the scope of systems that the proposed tool can learn.

One other important consideration is the current need for a manual implementation of the SUL oracle that classifies the behaviour of the system. This was out of the scope of the project, as the main goal was to prove the usefulness and ability to apply active automata learning for a generic target system. However, the current tool requires the user to provide their own system oracle that converts abstract inputs into real system calls and classifies the correctness of the observed output.

One final issue is the restriction on the type of counterexamples accepted. The implemented extension to C$\mathcal{E}$AL only allows for providing finite-length input words, which prevents certain types of behaviour from being tweaked, which are important for ensuring safety and liveliness properties. For example, the current algorithm has no way of expressing that "calling `add` after `remove` should always be rejected", as this would require the user to label these as false for the infinite set of all possible input strings containing an `add` following a `remove`. In practice, it is possible that the user provides several finite traces that the algorithm extrapolates this behaviour to infinite sequences. However, it still requires that the user provides enough of these traces for this to happen.

Despite the evaluation concerns of the tool, this project succeeded in showing that active automata learning tools can be used to produce reliable specifications and that their rigour does

not have to be in the way of manual adjustments of the results obtained, a must-have for this tool to be applicable to real-world software development and system design pipelines.

6.3 Future Work

The primary sources of future work are the limitations described in the evaluation above. The restrictions in learning an unbounded iterator were caused by the limitations in the RALib learning library. To address this, it would be necessary to add missing functionality, such as register-with-register comparison for guards and the integration of the increment theory with the main branch.

The system oracle generation has the problem of being domain-specific. Nevertheless, programming systems are often similar in the way that they convert abstract input symbols to concrete method executions and label certain behaviours as correct or incorrect. An example of this is the RALib `ClassAnalyzer` tool, which requires only a simple configuration file for generating the system oracle. Similar approaches could be explored to aid in the generation of these oracles.

Lastly, the limitations on counterexample types can be addressed by introducing rule-based counterexamples. These would convert a simple counterexample into a set of counterexample inputs defined by a condition (ex.: all inputs containing `add` right after `remove` should be rejected).

6.4 Final Remarks

Specifications are essential to any real-world system: they guide us through the good and permissible behaviour and warn us of the consequences of misuse. They are also a necessity in formal contexts that require technical descriptions of the system to evaluate if the intended behaviour matches with the implemented one. Nevertheless, they are rare due to the large amount of time and effort required. This project tackled the main reasons making developers and architects steer away from formal specifications and aimed to showcase the potential of state-of-the-art formal methods to aid in the generation of such specifications.

References

[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. "Mining specifications". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* POPL '02. New York, NY, USA: Association for Computing Machinery, Jan. 1, 2002, pp. 4–16. ISBN: 978-1-58113-450-6. DOI: 10.1145/503272.503275. URL: https://doi.org/10.1145/503272.503275 (visited on 05/21/2024).

[Alg+11] Jade Alglave et al. "Making Software Verification Tools Really Work". en. In: *Automated Technology for Verification and Analysis.* Ed. by Tevfik Bultan and Pao-Ann Hsiung. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 28–42. ISBN: 978-3-642-24372-1. DOI: 10.1007/978-3-642-24372-1_3.

[Amm+03] Glenn Ammons et al. "Debugging temporal specifications with concept analysis". In: *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation.* PLDI '03. New York, NY, USA: Association for Computing Machinery, May 9, 2003, pp. 182–195. ISBN: 978-1-58113-662-3. DOI: 10.1145/781131.781152. URL: https://doi.org/10.1145/781131.781152 (visited on 05/21/2024).

[Ang87] Dana Angluin. "Learning regular sets from queries and counterexamples". In: *Information and Computation* 75.2 (Nov. 1, 1987), pp. 87–106. ISSN: 0890-5401. DOI: 10.1016/0890-5401(87)90052-6. URL: https://www.sciencedirect.com/science/article/pii/0890540187900526 (visited on 04/29/2024).

[Ber07] Antonia Bertolino. "Software Testing Research: Achievements, Challenges, Dreams". en. In: *Future of Software Engineering (FOSE '07).* Minneapolis, MN, USA: IEEE, May 2007, pp. 85–103. ISBN: 978-0-7695-2829-8. DOI: 10.1109/FOSE.2007.25. URL: http://ieeexplore.ieee.org/document/4221614/ (visited on 12/05/2023).

[Bri+20] Tom Britton et al. *Reversible Debugging Software "Quantify the time and cost saved using reversible debuggers".* Nov. 2020.

[BZK21] Matthew L. Bolton, Xi Zheng, and Eunsuk Kang. "A formal method for including the probability of erroneous human task behavior in system analyses". In: *Reliability Engineering & System Safety* 213 (2021), p. 107764. ISSN: 0951-8320. DOI: https://doi.org/10.1016/j.ress.2021.107764. URL: https://www.sciencedirect.com/science/article/pii/S0951832021002921.

[Cas+14] Sofia Cassel et al. "Learning Extended Finite State Machines". In: *Software Engineering and Formal Methods.* Ed. by Dimitra Giannakopoulou and Gwen Salaün. Cham: Springer International Publishing, 2014, pp. 250–264. ISBN: 978-3-319-10431-7. DOI: 10.1007/978-3-319-10431-7_18.

[Cas+16] Sofia Cassel et al. "Active learning for extended finite state machines". In: *Formal Aspects of Computing* 28.2 (Apr. 2016), pp. 233–263. ISSN: 0934-5043, 1433-299X. DOI: 10.1007/s00165-016-0355-5. URL: https://dl.acm.org/doi/10.1007/s00165-016-0355-5 (visited on 12/14/2023).

[CW95] Jonathan E. Cook and Alexander L. Wolf. "Automating process discovery through event-data analysis". In: *Proceedings of the 17th international conference on Software engineering*. ICSE '95. New York, NY, USA: Association for Computing Machinery, Apr. 23, 1995, pp. 73–82. ISBN: 978-0-89791-708-7. DOI: 10.1145/225014.225021. URL: https://dl.acm.org/doi/10.1145/225014.225021 (visited on 05/21/2024).

[Die+24] Simon Dierl et al. *Scalable Tree-based Register Automata Learning*. arXiv.org. Jan. 25, 2024. URL: https://arxiv.org/abs/2401.14324v1.

[Fer+23] Tiago Ferreira et al. *Conflict-Aware Active Automata Learning (Extended Version)*. Sept. 6, 2023. DOI: 10.48550/arXiv.2308.14781. arXiv: 2308.14781[cs]. URL: http://arxiv.org/abs/2308.14781 (visited on 05/23/2024).

[Gar05] Simson Garfinkel. "History's worst software bugs". In: *Wired News, Nov* (2005), p. 1.

[Hie+09] Robert M. Hierons et al. "Using formal specifications to support testing". In: *ACM Computing Surveys* 41.2 (Feb. 2009), pp. 1–76. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/1459352.1459354. URL: https://dl.acm.org/doi/10.1145/1459352.1459354 (visited on 02/14/2024).

[HMS03] H. Hungar, T. Margaria, and B. Steffen. "Test-based model generation for legacy systems". In: *International Test Conference, 2003. Proceedings. ITC 2003.* International Test Conference, 2003. Proceedings. ITC 2003. Vol. 1. ISSN: 1089-3539. Sept. 2003, pp. 971–980. DOI: 10.1109/TEST.2003.1271084. URL: https://ieeexplore.ieee.org/document/1271084 (visited on 05/22/2024).

[HNS03] Hardi Hungar, Oliver Niese, and Bernhard Steffen. "Domain-Specific Optimization in Automata Learning". In: *Computer Aided Verification*. Ed. by Warren A. Hunt and Fabio Somenzi. Berlin, Heidelberg: Springer, 2003, pp. 315–327. ISBN: 978-3-540-45069-6. DOI: 10.1007/978-3-540-45069-6_31.

[HP18] Andrew Habib and Michael Pradel. "How many of all bugs do we find? a study of static bug detectors". In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE '18. New York, NY, USA: Association for Computing Machinery, Sept. 2018, pp. 317–328. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238213. URL: https://dl.acm.org/doi/10.1145/3238147.3238213 (visited on 12/04/2023).

[IHS14] Malte Isberner, Falk Howar, and Bernhard Steffen. "The TTT Algorithm: A Redundancy-Free Approach to Active Automata Learning". In: *Runtime Verification*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Cham: Springer International Publishing, 2014, pp. 307–322. ISBN: 978-3-319-11164-3. DOI: 10.1007/978-3-319-11164-3_26.

[Kli96] Rob Kling. "A - Systems Safety, Normal Accidents, and Social Vulnerability". In: *Computerization and Controversy (Second Edition)*. Ed. by Rob Kling. Second Edition. Boston: Morgan Kaufmann, 1996, pp. 746–763. ISBN: 978-0-12-415040-9. DOI: https://doi.org/10.1016/B978-0-12-415040-9.50144-5. URL: https://www.sciencedirect.com/science/article/pii/B9780124150409501445.

[KV94a] Michael J. Kearns and Umesh Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, Aug. 15, 1994. ISBN: 978-0-262-27686-3. DOI: 10.7551/mitpress/3897.001.0001. URL: https://direct.mit.edu/books/book/2604/An-Introduction-to-Computational-Learning-Theory (visited on 05/23/2024).

[Lam00] Axel Van Lamsweerde. "Formal specification: a roadmap". In: *Proceedings of the Conference on The Future of Software Engineering*. ICSE00: 22nd International Conference on Software Engineering. Limerick Ireland: ACM, May 2000, pp. 147–159. ISBN: 978-1-58113-253-3. DOI: 10.1145/336512.336546. URL: https://dl.acm.org/doi/10.1145/336512.336546 (visited on 02/14/2024).

[Leg+19] Owolabi Legunsen et al. "How effective are existing Java API specifications for finding bugs during runtime verification?" In: *Automated Software Engineering* 26.4 (Dec. 1, 2019), pp. 795–837. ISSN: 1573-7535. DOI: 10.1007/s10515-019-00267-1. URL: https://doi.org/10.1007/s10515-019-00267-1 (visited on 05/23/2024).

[Lem+17] Rogério de Lemos et al. "Software Engineering for Self-Adaptive Systems: Research Challenges in the Provision of Assurances". en. In: *Software Engineering for Self-Adaptive Systems III. Assurances*. Ed. by Rogério de Lemos et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 3–30. ISBN: 978-3-319-74183-3. DOI: 10.1007/978-3-319-74183-3_1.

[LK06] David Lo and Siau-Cheng Khoo. "SMArTIC: towards building an accurate, robust and scalable specification miner". In: *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. SIGSOFT '06/FSE-14. New York, NY, USA: Association for Computing Machinery, Nov. 5, 2006, pp. 265–275. ISBN: 978-1-59593-468-0. DOI: 10.1145/1181775.1181808. URL: https://doi.org/10.1145/1181775.1181808 (visited on 05/21/2024).

[LL18] Tien-Duy B. Le and David Lo. "Deep specification mining". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. New York, NY, USA: Association for Computing Machinery, July 12, 2018, pp. 106–117. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213876. URL: https://doi.org/10.1145/3213846.3213876 (visited on 05/21/2024).

[LT93] N.G. Leveson and C.S. Turner. "An investigation of the Therac-25 accidents". en. In: *Computer* 26.7 (July 1993), pp. 18–41. ISSN: 0018-9162. DOI: 10.1109/MC.1993.274940. URL: http://ieeexplore.ieee.org/document/274940/ (visited on 12/01/2023).

[Mar+07] Lawrence Z. Markosian et al. "Program Model Checking Using Design-for-Verification: NASA Flight Software Case Study". en. In: *2007 IEEE Aerospace Conference*. Big Sky, MT, USA: IEEE, 2007, pp. 1–9. ISBN: 978-1-4244-0524-4. DOI: 10.1109/AERO.2007.352767. URL: http://ieeexplore.ieee.org/document/4161597/ (visited on 12/03/2023).

[MB03] Jean-François Monin and Michael BSc. "Understanding Formal Methods". In: Jan. 2003, pp. 1–10. ISBN: 978-1-85233-247-1. DOI: 10.1007/978-1-4471-0043-0_8.

[NM98] N. Nayani and M. Mollaghasemi. "Validation and verification of the simulation model of a photolithography process in semiconductor manufacturing". en. In: *1998 Winter Simulation Conference. Proceedings (Cat. No.98CH36274)*. Vol. 2. Washington, DC, USA: IEEE, 1998, pp. 1017–1022. ISBN: 978-0-7803-5133-2. DOI: 10.1109/WSC.

1998.745835. URL: http://ieeexplore.ieee.org/document/745835/ (visited on 12/03/2023).

[Tho94] M. Thomas. "The story of the Therac-25 in LOTOS". In: 1994. URL: https://www.semanticscholar.org/paper/The-story-of-the-Therac-25-in-LOTOS-Thomas/6c9c6024cf95aadae8b7edf1160e0e4500410eb9 (visited on 12/10/2023).

[Vaa17] Frits Vaandrager. "Model learning". In: *Communications of the ACM* 60.2 (Jan. 23, 2017), pp. 86–95. ISSN: 0001-0782. DOI: 10.1145/2967606. URL: https://dl.acm.org/doi/10.1145/2967606 (visited on 04/29/2024).

[Wal+07] Neil Walkinshaw et al. "Reverse Engineering State Machines by Interactive Grammar Inference". In: *14th Working Conference on Reverse Engineering (WCRE 2007)*. 14th Working Conference on Reverse Engineering (WCRE 2007). ISSN: 2375-5369. Oct. 2007, pp. 209–218. DOI: 10.1109/WCRE.2007.45. URL: https://ieeexplore.ieee.org/document/4400167 (visited on 05/22/2024).

[WML02] John Whaley, Michael C. Martin, and Monica S. Lam. "Automatic extraction of object-oriented component interfaces". In: *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. ISSTA '02. New York, NY, USA: Association for Computing Machinery, July 1, 2002, pp. 218–228. ISBN: 978-1-58113-562-6. DOI: 10.1145/566172.566212. URL: https://doi.org/10.1145/566172.566212 (visited on 05/21/2024).

Bibliography

Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, Apr. 25, 2008. 994 pp. ISBN: 978-0-262-30403-0.

Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge: Cambridge University Press, 2010. ISBN: 978-0-521-76316-5. DOI: 10.1017/CBO9781139194655.

Michael J. Kearns and Umesh Vazirani. *An Introduction to Computational Learning Theory*. The MIT Press, Aug. 15, 1994. ISBN: 978-0-262-27686-3. DOI: 10.7551/mitpress/3897.001.0001.

Wenchao Li. "Specification Mining: New Formalisms, Algorithms and Applications". PhD thesis. UC Berkeley, 2013.

David Lo et al., eds. *Mining Software Specifications: Methodologies and Applications*. Boca Raton: CRC Press, June 2, 2011. 460 pp. ISBN: 978-0-429-13126-4. DOI: 10.1201/b10928.