Multi-Oracle Coevolutionary Learning of Requirements Specifications from Examples in On-The-Fly Markets

Marcel Wever
Heinz Nixdorf Institut,
Paderborn University, Germany

Lorijn van Rooijen
Department of Computer Science,
Paderborn University, Germany

Heiko Hamann
Institute of Computer Engineering,
University of Lübeck, Germany

Summer 2020

Abstract

In software engineering, the imprecise requirements of a user are transformed to a formal requirements specification during the requirements elicitation process. This process is usually guided by requirements engineers interviewing the user. We want to partially automate this first step of the software engineering process in order to enable users to specify a desired software system on their own. With our approach, users are only asked to provide exemplary behavioral descriptions. The problem of synthesizing a requirements specification from examples can partially be reduced to the problem of grammatical inference, to which we apply an active coevolutionary learning approach. However, this approach would usually require many feedback queries to be sent to the user. In this work, we extend and generalize our active learning approach to receive knowledge from multiple oracles, also known as proactive learning. The 'user oracle' represents input received from the user and the 'knowledge oracle' represents available, formalized domain knowledge. We call our two-oracle approach the 'first apply knowledge then query' (FAKT/Q) algorithm. We compare FAKT/Q to the active learning approach and provide an extensive benchmark evaluation. As result we find that the number of required user queries is reduced and the inference process is sped up significantly. Finally, with so-called On-The-Fly Markets, we present a motivation and an application of our approach where such knowledge is available.

∗ This paper was accepted to be published in the Evoluationary Computation journal: https://doi.org/10.1162/evco_a_00266

1 Introduction and Related Work

The task of partially automating the requirements specification process comes with two intriguing and mutually dependent challenges: sparse data and a vaguely defined objective. The human user can only define a few examples of the desired behavior of the software, due to limited resources. For this reason, our algorithm initially receives only little data. In contrast to standard optimization problems, the process of requirements specification is characterized by an incompletely defined objective. Often, the user is not able to give a complete description of the desired behavior of the software, which would allow to immediately formalize the requirements. The user consciously assumes certain generalizations while unconsciously also misses to specify relevant features. The requirements specification process is a journey started from a too roughly defined objective followed by iterated interactive refinement until the complete requirements specification is achieved. We approach these challenges with an active, multi-objective, and coevolutionary approach extending the work of Bongard and Lipson (2005a).

Our approach relies on a carefully orchestrated set of methods. Active learning is used to obtain additional data in an interactive way. The algorithm initiates queries sent to the user, who acts as an oracle. Coevolution is used to do this efficiently and to ask the right questions with maximal gain of information. Coevolution is known to generate complex evolutionary dynamics and can be challenging to be governed (Wiegand et al., 2001; Popovici et al., 2012). Typically, one distinguishes competitive coevolution from cooperative coevolution (cf. game theory). In competitive coevolution, at least two species directly compete in a task and may develop more and more complex solutions in an arms race. Cooperative coevolution is arguably a technique of modularization with several evolutionary runs generating components of solutions (Wiegand et al., 2001). Our approach may be understood as competitive coevolution in the form of a dialogue between two evolutionary algorithms that iteratively refine both the actual task and its solution. In our case, the coevolutionary dynamics is manageable because it is inherently biased towards more sophisticated solutions. Pathologies of coevolution, such as circular behavior, are not observed here (Wiegand, 2003).

In the following, we first motivate and describe the application, which we then transfer to the more general problem class of grammatical inference, and finally, we provide a description of our methodology.

1.1 Requirements specification and Search-Based Software Engineering

In order to develop a software system, the requirements on the system need to be specified first. Typically, users are supported by a requirements engineer in the elicitation of their requirements. A challenge is that the requirements defined by users are usually incomplete and imprecise. These vague user expectations need to be transformed into a formal requirements specification of the desired system to allow for an appropriate implementation. In the following, we describe our search-based software engineering approach using a genetic algorithm (Holland, 1975). We develop formalized behavioral requirements specifications, in the form of protocols, semi-automatically. Our vision is to use these approaches in future dynamic software service markets where software

services are searched and composed (semi-)automatically on a large scale. In addition, outside the context of these future dynamic software service markets, our method can also provide useful support for requirements engineers during the requirements specification process.

Search-based software engineering follows the idea of interpreting software engineering problems as search and optimization problems, which are then solved using metaheuristic search and optimization techniques (Harman et al., 2012). Research on search-based software engineering includes requirements engineering (Zhang and Harman, 2010), testing, optimization of code (Langdon and Harman, 2015), and debugging (Arcuri and Yao, 2008). The main idea of search-based software engineering is to support the software development process either by providing additional helpful engineering tools or by (partially) automating the development process. Automatic synthesis, for example of specifications, software models, or code, would empower users to (partially) construct software independently without support by human experts.

Synthesizing model transformations from example transformations of model instances has been studied before (Kappel et al., 2012; Kessentini et al., 2012; Faunes et al., 2013; Kühne et al., 2016). Evolutionary approaches to synthesizing model transformations were, for example, proposed by Faunes et al. (2013) and by Kühne et al. (2016). Evolutionary computation is a well-chosen method here, because the search space of such *by-example* problems is usually complex. In addition, the synthesis problems do not need to be fully understood due to the black-box-optimization approach.

Arcuri and Yao (2008) apply coevolution in a search-based software testing approach to debugging. In a competitive approach, both programs and test cases are evolved. In the test-case evolution, rewards are given for finding bugs, while in the program evolution, the fitness is defined by a distance between program outputs and expected results. Some, but not all, bugs can be fixed using this approach.

1.2 Dynamic Software Service Markets

Our main motivation for this work is the vision of a dynamic software service market, in which software services are built automatically and on-demand. The idea is to use the requirements specification to automatically search such a market for a service that complies with the requirements. In the case that such a service does not exist, the specification is analyzed and decomposed, in order to search for appropriate services that may be used as building blocks to automatically compose the required software service. So one can view these markets as more advanced versions of traditional app stores. We study this idea of a dynamic software service market in the Collaborative Research Center "On-The-Fly Computing"[1]. See Section 5 for more details.

With this application in mind, sound formal requirements specifications are necessary in order to be able to automatically build the corresponding software services. Just as in an app store, a user may expect to obtain a software service right away. This means that we cannot rely on a requirements engineer, hence the requirements specification process itself has to be automatic as well. That is, users should be empowered to quickly and autonomously create their specifications without the help of experts. We thus face the challenge of how to create formal requirements specifications semi-automatically and in a user-friendly way.

In the following, we assume users who are able to define example behaviors of their desired software service in the form of simplified sequence diagrams. These simplified

[1] `https://sfb901.uni-paderborn.de/`

sequence diagrams give examples of the user–service interaction in the form of simple sequences of operations (see Table 1 for an example). The low level of expertise needed to use our approach distinguishes it from other approaches to synthesizing behavioral descriptions, such as the work of Mäkinen and Systä (2000); Harel et al. (2005); Lambeau et al. (2008).

1.3 Grammatical Inference

Our specification-by-example approach requires the user to provide examples of desired and of prohibited behavior of the software service. These example behaviors are represented in the form of simplified sequence diagrams, from which we extract sequences of operations. The goal is to generalize from these examples in order to find a complete formal specification that describes the behavior of a service that is as close as possible to what the user originally had in mind. In the following, we restrict our approach to finite automata as formal behavioral specifications. Therefore, we translate creating behavioral requirements specifications into a search problem in the space of deterministic finite automata (DFA).

We apply methods of grammatical inference, which is the problem of learning a representation of a (typically regular) language based on a given set of words, with each word labeled as member or non-member of that language. The theoretical fundamentals of grammatical inference go back to Gold (1967), and for active learning, to Angluin (1987). The surveys by de la Higuera (2005, 2010) provide an overview of the various approaches to many variants of this inference problem. These rather theoretical approaches typically simplify the setting, for example, by restricting the alphabet size of the automata to two. In our case, the alphabet represents the set of operation names that are used to describe the example behavior of the desired service and are thus usually much larger.

A popular deterministic approach to grammatical inference is the so-called Evidence Driven State Merging (EDSM) algorithm, which was introduced by Lang et al. (1998). Evolutionary approaches have been used before to solve the grammatical inference problem by evolving deterministic finite automata (DFAs), see (Lucas and Reynolds, 2003, 2005; Bongard and Lipson, 2005b; Gómez, 2006). Tsarev and Egorov (2011) evolve finite state input-output machines. Lucas and Reynolds (2003) found that evolutionary methods can outperform EDSM for target DFAs consisting of less than 32 states.

In the following, we basically interpret the problem of synthesizing behavioral requirements specifications as a problem instance of grammatical inference (i.e., learning an automaton from a set of input examples). However, the techniques that were developed within this field cannot resolve all of our challenges. In the standard problem of grammatical inference, the input examples are easily generated from the target automaton that serves as oracle. In our scenario, such a formalized oracle does not exist and the user needs to serve as the oracle instead. Given that our oracle is a human being, we potentially deal with vague or even contradicting statements and with fatigue. Hence, our approach needs to be robust and efficient. In addition, we introduce a second oracle that represents the feasible compositions of software services on our dynamic software service market. In summary, viewing the challenging problem of synthesizing formal behavioral requirements specifications from examples as a grammatical inference problem is helpful but does not suffice: we have to extend the available methods considerably.

1.4 Active Learning, Coevolution, and Multi-Objective Optimization

In the requirements specification process, only the user can provide information about the desired software. So, only the user can provide us with training data. Hence, we face two challenges. First, a user can only provide a limited amount of examples and would be overcharged if we ask for too much input. Second, we solve a well-defined optimization problem where we have just one clearly defined optimal target DFA but the input usually allows for a variety of possible, technically correct DFAs that fit to the user's specification. We need to find a good balance between generalizing from the user's input while not diluting the user's specification. Hence, we want to have several appropriate automata as result, from which the user may select the one fitting his expectations best.

To address the first challenge (i.e., limited amount of input data), we try to obtain the *right* input data. This is why we use active learning. Our coevolutionary approach actively searches for the right question to ask the user. The right question is the question with the biggest information gain. During the evolutionary process, our approach comes up with a currently relevant example description of the program behavior (i.e., a word) and asks the user to label this example. In this interactive evolutionary approach, it is essential to be as efficient as possible and to minimize the queries to the user. Otherwise, the user could become fatigued and may lose interest. Following the approaches of Bongard and Lipson (2005b) and Ly and Lipson (2014), we use competitive coevolution to find the potentially most informative next query. One population consists of DFAs, which formalize the requirements specifications, and the other population consists of candidate words, which an oracle may receive as queries.

We evolve a population of DFAs in such a way, that evolution is biased towards more accurate and efficient formalizations of the growing input data. With the coevolved population of potential queries, we try to find the optimal query, that is, a query that creates the biggest disagreement among the current best DFAs. The fitness of potential queries is defined as the amount of caused disagreement in good candidate DFAs, while the fitness of DFAs is based on their accuracy with respect to the input data seen so far.

For small input examples, we face the challenge that even the most carefully picked and informative input data triggers the same behavior in the DFAs present in the population. There are many DFAs with a given maximal number of states that react in precisely the same way to the input data (i.e., all of them accept and reject the same words from the input data). This would create large plateaus in the respective fitness landscapes and would thus collapse our evolutionary approach into a mere random search. We fix this problem and at the same time deal with the second challenge (balance of generalization and respecting the user's specification) by using multi-objective optimization to evolve DFAs. In contrast to previous evolutionary approaches to grammatical inference, we use multi-objective evolution here. We use NSGA-II (Deb et al., 2002) that outputs a set of solutions, all being Pareto-optimal with respect to the objectives. The multi-objective approach allows us to fine-tune several quality aspects of the candidate DFAs.

For the evolution of potential queries, we also make use of multi-objective optimization. We get better performance by also minimizing the length of a query instead of focusing only on creating disagreement with a query. An additional advantage of using multi-objective optimization for both populations is that one can easily extend our approach with additional objectives.

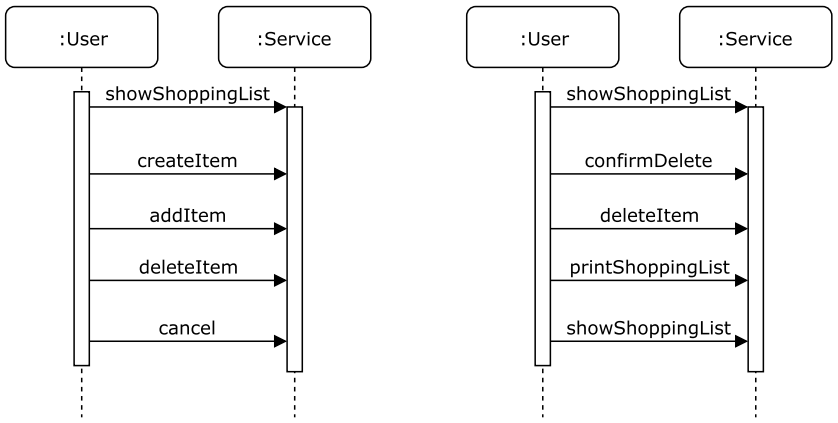


Figure 1: Two examples of user input in the form of Simplified Sequence Diagrams (SSDs). The left SSD shows behavior specified as desired and the right SSD shows forbidden behavior.

In this work, we extend our previous paper by Wever et al. (2017). An essential extension is the introduction of a second oracle that represents information we would get from the domain, i.e. the dynamic software service market, concerning which services can be configured and composed. Our previous work, like the work of Bongard and Lipson (2005b), relied on the fact that the sets of training data were balanced in terms of their labels. In our present work, we propose and evaluate a method to deal with unbalanced sets of training data. In addition, we report more results and provide more information about the motivation and background of our research.

The evaluation of our approach is given in Section 4. In this section, we compare the performance of different variations of our approach to that of the approach of Bongard and Lipson (2005b) and to the single-oracle approach of Wever et al. (2017), for different combinations of alphabet size and DFA size.

2 Generating Protocols from Examples

Starting from exemplary simplified UML sequence diagrams, we aim to infer, from these examples, a protocol describing the complete behavior of an entire software system. In the following, we refer to such a software system as a *service*. With Simplified Sequence Diagrams (SSDs) we denote UML Sequence Diagrams (OMG, 2011) restricting the number of involved objects to two: the user and the desired service. Moreover, we only allow operation call messages to be exchanged between the two objects. In Figure 1, two examples of SSDs are shown, where a user interacts with a service that represents an application for managing a shopping list. As can be seen from the SSDs, the application allows the user to create and delete items, and the user may show or print this list. The SSD on the left represents desired behavior, whereas the SSD on the right describes forbidden behavior.

The synthesis process is triggered by a user providing two sets of SSDs: one set describing desired behavior and a second one describing prohibited behavior. In these SSDs, the user is allowed to use arbitrary operation names in order to describe operation

call messages, but each operation name that will be used in the resulting service should occur at least once in one of the provided examples. Furthermore, the user is free to choose the number of examples that he wants to provide.

In order to reduce the problem of inferring a protocol from the provided examples to the problem of grammatical inference, we first need to extract the alphabet of the language to be learned. This alphabet simply is the set of all operation names used in the input data. Thus, we extract the operation names from each SSD in the input data, and let each operation name define a letter of the alphabet. For instance, consider the two example SSDs from Figure 1. From the left SSD, we extract the subalphabet $\Sigma_1 = \{\texttt{showShoppingList}, \texttt{createItem}, \texttt{addItem}, \texttt{deleteItem}, \texttt{cancel}\}$. Similarly, the right SSD yields the subalphabet $\Sigma_2 = \{\texttt{showShoppingList}, \texttt{confirmDelete}, \texttt{deleteItem}, \texttt{printShoppingList}\}$. We derive the alphabet Σ by taking the union over all SSDs from the input data, that is, $\Sigma := \bigcup_i \Sigma_i$. Thus, in the example above, supposing that these two SSDs represent the complete input data, the alphabet would be $\Sigma = \{\texttt{showShoppingList}, \texttt{createItem}, \texttt{addItem}, \texttt{deleteItem}, \texttt{cancel}, \texttt{confirmDelete}, \texttt{printShoppingList}\}$.

Since we restrict SSDs to involve only two objects, the entire communication between these objects is a sequence of operation call messages. Assuming that the direction of communication is inherent in the operation call message itself (e.g., the service may not invoke showShoppingList on the user), we can write an SSD as a sequence of operation names without any loss of information. Hence, with regard to the example, we obtain the following sequences:

`showShoppingList createItem addItem deleteItem cancel`

and

`showShoppingList confirmDelete deleteItem printShoppingList showShoppingList`.

Continuing the problem reduction to grammatical inference, note that we defined each operation name to be a symbol of the extracted alphabet Σ. Therefore, we can interpret each sequence of operation names as a word $w \in \Sigma^*$. We associate a label l from the set $L := \{\text{ACCEPT}, \text{REJECT}\}$ to each such word by setting

$$l := \begin{cases} \text{ACCEPT, if the SSD corresponding to } w \text{ represents desired behavior} \\ \text{REJECT, if the SSD corresponding to } w \text{ represents prohibited behavior} \end{cases}$$

In this way, we obtain a set of training examples $S \subseteq (\Sigma^* \times L)$. As we want to infer a protocol describing the behavior of the desired service, applying grammatical inference yields a deterministic finite automaton using the operation names as edge labels. We define a deterministic finite automaton as follows.

2.1 Genetic Representation of DFAs

From the two input sets of SSDs, we aim to infer a complete description of the behavioral requirements on the system. For these descriptions, we use the formalism of deterministic finite automata. Such an automaton describes how the system should behave: at each point in time, the system is in one of the states, starting in the initial state. Upon an operation call, the state of the system may change. A state is an abstraction of all information about past operation calls, which suffices to decide how the system will behave upon the next operation call.

Formally, a 5-tuple $A = (\Sigma, Q, \delta, q_0, F)$, where Σ is an alphabet, Q a finite set of states, $\delta : Q \times \Sigma \to Q$ a transition function, $q_0 \in Q$ the initial state and $F \subseteq Q$ a set of accepting states, is called a *deterministic finite automaton* (DFA) over the alphabet Σ.

$$\begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 2 \\ 2 & 2 & 2 \end{pmatrix}$$

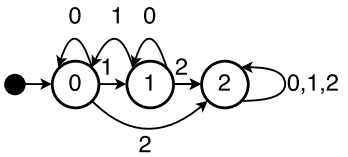


Figure 2: Example of the matrix representation of a DFA, with the corresponding graph.

A word $w = i_1 \cdots i_n$ of length n denotes a sequence of n *input symbols* $i_j \in \Sigma$, where $1 \leq j \leq n$. It is *accepted* by a DFA A if there exists a sequence $q_0 q_1 \cdots q_n$ of states such that $q_n \in F$ is an accepting state, and $\delta(q_{j-1}, i_j) = q_j$ for $j \in \{1, \ldots, n\}$. In all other cases, we say that the word is *rejected*. The *size* of a DFA A is defined as the number of states $|Q|$.

A total function is a function $f : X \to Y$ for which holds that for all $x \in X$, there exists $y \in Y$ such that $f(x) = y$. If the transition function δ is a total function, then we call A a *complete* DFA. In this case, the transition function δ is defined for each state and input symbol combination. As every DFA can be extended to a complete DFA accepting the same language in a canonical way, we only consider complete DFAs in the following. This allows us to represent a DFA as $|Q| \times |\Sigma|$-matrix.

More precisely, we construct bijective mappings for both the elements in Q and in Σ. Let $I_Q : Q \to \{0, \ldots, |Q| - 1\}$ and $I_\Sigma : \Sigma \to \{0, \ldots, |\Sigma| - 1\}$ be bijective functions, such that the image $I_Q(q_0)$ of the initial state q_0 is 0. We call I_Q respectively I_Σ the *index of* Q respectively the *index of* Σ. We use the images of I_Q as row index and the images of I_Σ as column index of the matrix. Note that this implies that, against convention, our matrix coordinates start from 0. The entries of the matrix also take values from the set $\{0, \ldots, |Q| - 1\}$.

For $i \in \{0, \ldots, |Q|-1\}$ and $j \in \{0, \ldots, |\Sigma|-1\}$, the entry $T_{i,j}$ has value $I_Q(\delta(q_i, a_j))$. Here, q_i denotes the state $I_Q^{-1}(i)$, that is, the state $q \in Q$ for which $I_Q(q) = i$, and a_j denotes $I_\Sigma^{-1}(j)$, that is, the input symbol $a \in \Sigma$ for which $I_\Sigma(a) = j$. The entry $T_{i,j}$ contains the index of the state that is reached when being in state q_i and reading input symbol a_j.

$$T = \begin{pmatrix} I_Q(\delta(q_0, a_0)) & I_Q(\delta(q_0, a_1)) & I_Q(\delta(q_0, a_{|\Sigma|-1})) \\ I_Q(\delta(q_1, a_0)) & \ddots & \vdots \\ \vdots & & \\ I_Q(\delta(q_{|Q|-1}, a_0)) & \ldots & I_Q(\delta(q_{|Q|-1}, a_{|\Sigma|-1})) \end{pmatrix}$$

We divide the representation of a DFA $A = (\Sigma, Q, \delta, q_0, F)$ into two parts. The first part (Σ, Q, δ, q_0) is represented by a $|Q| \times |\Sigma|$-matrix T as explained above. A concrete example of such a matrix and its corresponding graphical representation are shown in Figure 2. Note that this representation does not yet contain any accepting states.

The remainder of the DFA representation consists of the set of accepting states. In a naive way, the set of accepting states F could be represented as an array of $|Q|$ booleans, where an entry k is true if and only if $I_Q^{-1}(k)$ (that is, the state $q \in Q$ for which $I_Q(q) = k$) is an accepting state. Note that explicitly evolving this array contributes a factor of $2^{|Q|}$ to the search space complexity. Instead, we apply the so-called Smart State Labeling (SSL) algorithm by Lucas and Reynolds (2003, 2005) and consider

Desired Behavior	showShoppingList createItem addItem deleteItem cancel showShoppingList showShoppingList showShoppingList showShoppingList createItem addItem showShoppingList deleteItem confirmDelete showShoppingList deleteItem confirmDelete showShoppingList
Prohibited Behavior	showShoppingList confirmDelete deleteItem printShoppingList showShoppingList createItem confirmDelete createItem deleteItem cancel deleteItem createItem showShoppingList createItem deleteItem addItem cancel showShoppingList createItem addItem addItem

Table 1: Example user input as sequences of operation names describing the desired and prohibited behavior of the desired software.

only those DFAs with F optimal w.r.t. the set of provided training examples S.

The SSL algorithm computes the set of accepting states F from the set of training examples S, by considering for each state $q \in Q$ all the training examples ending in this state. If at least half of the training examples ending in a state q are labeled with ACCEPT, then the state q is considered to be an accepting state. Otherwise, we declare it a rejecting state. As an illustration, consider the following example. If there are exactly two training examples ending in a state q, and at least one of them is labeled ACCEPT, then q is added to F. If none of the two training examples is labeled ACCEPT, q is considered to be a rejecting state, and thus, q does not become an element of F.

As the second part of the DFA is computed from the first part and is therefore represented implicitly, we only need matrix T for the genetic representation of a DFA. We concatenate the rows of T to obtain a sequence of integers. Using this genotype enables a straight-forward application of standard recombination techniques, such as single-point crossover. The resulting search space complexity in our setup is $|Q|^{|Q|\cdot|\Sigma|}$.

Furthermore, our approach benefits from the SSL algorithm in a second manner. Since we are interested in finding DFAs for which the accepting states are optimal w.r.t. the provided training examples, the SSL algorithm prevents proposing inferior DFAs to the user. Furthermore, experiments (Lucas and Reynolds, 2005; Gómez, 2006) showed that evolutionary algorithms applying the SSL algorithm outperform approaches in which both the transition matrix and the set of accepting states are evolved.

2.2 Example

As already outlined in the beginning of this section, we require the user to provide example SSDs, as shown in Figure 1, which are then generalized to a protocol. In general, it is beneficial for the subsequent inference process to provide as many examples as possible per set of SSDs (desired/prohibited). In Table 1, some examples of desired resp. prohibited behavior are presented. Note that the examples are already compiled to sequences of operation names, i.e. words over the alphabet Σ which contains all distinct operation names.

In Figure 3, an example protocol of a simple shopping list service is illustrated as a DFA, representing a possible generalization of the examples in Figure 1 and Table 1. In this DFA, state 0 is the initial state. The desired service enables a user to create items and add them to the shopping list as well as to delete these items again. Moreover, the

user can show the shopping list or print it. Adding and deleting items from the list are two step procedures that may be canceled by the user. Before adding a new item, the user first has to create it. In the case the user wants to delete an item, this action has to be confirmed in order to prevent the user from deleting items accidentally.

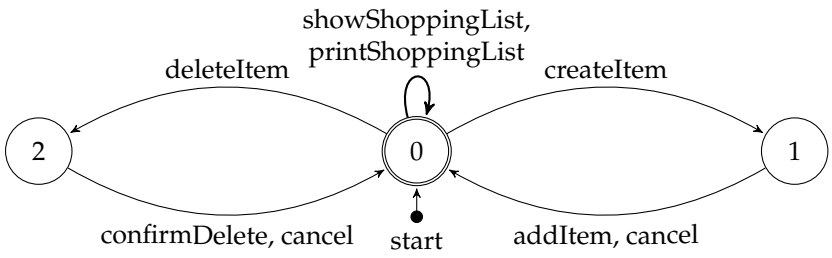


Figure 3: Example DFA of a simple shopping list application.

For clarity, we refrain from presenting a complete DFA and only show those edges being part of an accepting word. That is, reading an input symbol not shown for a state immediately results in rejecting the respective word.

2.3 Multi-Objective Evolution of DFAs

The problem of grammatical inference amounts to finding a DFA that accepts the positive input examples and rejects the negative input examples. In previous approaches using evolutionary algorithms to solve grammatical inference, the only objective is the accuracy on the input examples (Lucas and Reynolds, 2003, 2005; Bongard and Lipson, 2005b; Gómez, 2006). That is, these approaches try to maximize the fitness function

$$f_{\text{all}}(A) = \frac{\left|\{(w,l) \in S \mid A(w) = l\}\right|}{|S|}. \tag{1}$$

Here, A is a candidate DFA, S is the set of training data, which consists of tuples (w, l), where w is a word and l is its correct label (either ACCEPT or REJECT). The label returned by the DFA A when reading w is denoted by $A(w)$.

In principle, there are infinitely many DFAs that have fitness value 1 for the f_{all}-objective, for a given (finite) set of training data. As explained in Section 2.1, we bound the size of DFAs in our approach. This ensures that our search space is finite. However, for a given set of training data, there may still be many DFAs with the same value for f_{all}. That is, there may be large fitness plateaus in the search space. It roughly holds that a smaller set of training data yields a larger number of DFAs having the same value for the objective f_{all}.

In the application scenario that we describe in Section 5, the training data comes from two sources: a user who wants to buy a service on a software service market, and a second oracle that contains knowledge about the feasibility of service compositions in this market. In this setup (cf. Section 3.1), only the user provides positive examples to the set of training data. The number of such examples is thus limited. In order to provide the evolutionary algorithm with more information, we make use of an additional objective in our approach.

Note that we are not only looking for a DFA that behaves accurately on the training data, but that the DFA should furthermore *generalize* from the training data. Roughly

speaking, a DFA that uses a smaller number of states to recognize the input examples will generalize more from this training data. One reason for this is that a sufficiently small DFA is forced to have loops in order to recognize the input examples.

Our second objective for DFAs measures the proportion of states of the automaton that are reachable from the initial state. This is a measure for the size of a DFA, since states that are not reachable are irrelevant. Note that since the size of the matrix encoding a DFA is fixed, the unreachable states are still part of the genotype. In a DFA that accurately classifies the input examples, the set of reachable states contains the set of states needed to read the input examples. Using this objective allows us to focus on the structure of the DFA. The second objective, which we try to minimize, is

$$f_{\text{reach}}(A) = \frac{\left|\{q \in Q \mid \exists w \in \Sigma^*.\ \delta(q_0, w) = q\}\right|}{|Q|}, \tag{2}$$

that is, we divide the number of states for which there exists some word that leads from the initial state q_0 to that state by the total number of states. We use the NSGA-II algorithm (Deb et al., 2002) for multi-objective evolution in order to evolve DFAs on the objectives f_{all} and f_{reach}. We maximize the first objective, while we minimize the second one. Besides the advantages of stimulating the generalization behavior mentioned above and of avoiding plateaus in the fitness landscape, applying a multi-objective approach to the grammatical inference problem furthermore allows us to maintain a diverse population of candidate DFAs. In NSGA-II, Pareto domination of solutions is used to sort the individuals and preserve a diverse population.

The concept of Pareto-optimality is particularly useful in our application scenario (cf. Section 5), since it allows us to present the set of Pareto-optimal DFAs, obtained at the end of the evolution, to the user. The user may then decide for himself, which of these candidate solutions fits his wishes best. This adds to the flexibility of our approach and makes it well-suited for dealing with users that have individual preferences. To avoid overwhelming the user, we may choose to only present the performance values of solutions on the different objectives, together with typical example words, instead of showing the automata.

We use NSGA-II with a tournament based selection, in which two individuals are compared using Pareto dominance as by Deb (1998). If the result for two individuals is inconclusive, the individuals are furthermore compared using the crowding distance, following (Deb et al., 2002). This distance provides a measure to see how isolated a solution in the objectives space is. As one wants to maintain a diverse population, solutions that have a less crowded neighborhood are preferred.

The genetic operators we use are single-point crossover and mutation. As explained in Section 2.1, we represent DFAs as strings of integers. These strings are projected to binay encodings, on which we apply single-point crossover with a probability of 1. We mutate individuals by flipping a single bit in the binary encoding of an entry, with probability $p_M = 1/(|Q| \times |\Sigma|)$. In average we then have one changed entry in the corresponding matrix.

In the following, we extend our multi-objective evolutionary approach to an active coevolutionary learning approach. We actively seek for the most informative queries about which we can ask our oracles. Asking the right queries to augment the training data allows in particular to limit the needed amount of interaction with the user.

2.4 Active Learning and Coevolution

In active learning, the learning algorithm may actively query an oracle about the labels of new data points. That is, the set of training data can be augmented strategically. For the problem of grammatical inference, active approaches were shown to outperform passive approaches. Bongard and Lipson (2005b) give an active algorithm that can strategically decide which query to ask, whereas this is decided at random for the passive algorithm. The results imply that asking strategic queries decreases the total amount of queries needed in order to infer the target DFA.

As we explain in Section 2.6 and 3.1, there are two oracles present in our scenario. Both of these oracles may be queried to label words. In the aforementioned sections, it is explained how the interaction with the two different oracles is designed. In Section 3.2, we explain how we alternate between these two oracles. This is done in such a way that the user (the first oracle) is bothered as little as possible. That is, we only ask the user for information if this information cannot be found anywhere else.

Bongard and Lipson (2005b) introduced an active coevolutionary learning algorithm for grammatical inference, called the Estimation Exploration Algorithm (EEA). This is a single-objective coevolutionary algorithm that interacts with one oracle. The EEA algorithm consists of two different phases, between which is alternated until a stopping condition holds. While the estimation phase evolves automata, words are evolved in the exploration phase. The authors refer to individuals of the estimation phase as *candidate models* and to individuals of the exploration phase as *candidate tests*.

The algorithm starts in the estimation phase. In this phase, candidate models are evolved in two distinct subpopulations (with no interbreeding) for g generations. The evolution takes place in two distinct subpopulations in order to maintain more diversity within the overall population. After this, the algorithm moves to the exploration phase. In this phase, again for g generations, candidate tests are evolved. The fitness of candidate tests is determined by the *disagreement* of some candidate models regarding the label of the word. A candidate test that causes most disagreement among these models has a good chance to entail most new information about these models: after an oracle has labeled the test, a potentially large part of the models no longer has perfect accuracy on the input data, which would give a new impulse to the subsequent evolution of models. The disagreement is expressed as

$$f_{\text{dis}}(t) = 1 - 2\left|0.5 - \frac{\sum_{j=1}^{|C|} \text{label}(c_j, t)}{|C|}\right|. \tag{3}$$

Here, t is a candidate test, C is a set of candidate models and $\texttt{label} : \text{Model} \times \text{Test} \to \{0, 1\}$ is a function that returns 1 if the candidate model accepts the test and 0 otherwise.

The strategy of letting a set of models vote on a query by disagreement is known, in the field of active learning, as query-by-committee (Settles, 2012). In EEA, this committee consists of the best individual of each subpopulation evolved in the preceding estimation phase. After the candidate tests have been evolved, the best candidate test is taken as a query to the labeling oracle.

The coevolutionary approach taken by Bongard and Lipson (2005b) turns out to be beneficial for the problem of grammatical inference. An evolved candidate test with maximum disagreement is the most informative query at that moment in the evolution. Labeling this word and adding it to the training data leads to the evolution of better suited candidate models. After this, a new test, which causes maximum disagreement among the new models, is evolved. Altogether, the competitive coevolution of candidate models and candidate tests leads to an arms race (Bongard and Lipson, 2005b).

In our approach, we follow the idea of EEA and adapt it to our multi-objective approach with two oracles. Similar to EEA, we divide the population of candidate DFAs into two subpopulations that are evolved individually. We apply NSGA-II with the objectives introduced in Section 2.3 to each of these subpopulations.

From the multi-objective evolution of DFAs, we obtain a Pareto front of candidate DFAs. The candidate DFAs of the Pareto fronts from both subpopulations form the committee (i.e., the set C) from which the disagreement for the candidate words is calculated, as described above. The EEA algorithm by Bongard and Lipson (2005b) uses the two best individuals to form the committee. Our approach uses Pareto fronts to form the committee. In this way, the committee may consist of more than two candidate models, which makes fractional disagreement values possible in addition to the extremes of 0 and 1. This is particularly advantageous, since this causes a gradient in the fitness landscape.

In EEA, the population of tests is, analogous to the population of models, evolved according to a single objective. We have found, however, that we obtain better results if we, again, add another objective. This second objective minimizes the length of a word. In preliminary experiments, it turned out that adding this objective is useful, whereas adding the opposite objective (maximizing the length of a word) is not. An explanation for this finding may be that a short word that causes maximum disagreement is more informative than a longer word causing the same amount of disagreement. If some DFAs already disagree on a small word, this means that there are differences close to the initial states of the DFAs. Directing candidate DFAs to improve at this location may have a bigger positive impact than directing them to improve at a point further away from the initial state.

We therefore evolve words with respect to two objectives: the amount of disagreement caused among the committee of DFAs (to be maximized) and the length of the word (to be minimized). After the evolution of words with respect to these two objectives, we select that word from the Pareto-front that causes maximum disagreement and query one of our oracles for this word.

2.5 Genetic Representation and Multi-Objective Evolution of Words

As explained before, we represent SSDs as words over the extracted alphabet of operation names Σ. In our genetic representation of words, the length of a word is bounded by $\ell_{\max}$. In this way, the search space complexity for words is $\Sigma^{\leq \ell_{\max}}$.

We represent a word by a tuple $t = (\ell, w)$. The first coordinate, ℓ, is a number between 0 and $\ell_{\max}$ and defines the length of the word represented by t. The second coordinate, w, is a word over Σ of length $\ell_{\max}$. The word represented by t corresponds to the first ℓ letters of the word w. The remaining letters of w are not taken into account. As an example, consider the tuple $t = (5, acabacb)$ over the alphabet $\{a, b, c\}$. This tuple represents the word formed by the first 5 symbols of $acabacb$, that is, the word $acaba$.

For the genetic representation of a tuple t, we use the bijective mapping $I_\Sigma : \Sigma \to \{0, \ldots, |\Sigma| - 1\}$ defined in Section 2.1. This mapping allows us to represent t as an integer string of length $\ell_{\max} + 1$. The first integer in this string denotes the value of ℓ, whereas the other integers denote the indices of the letters that form w. For the mapping $a \mapsto 0, b \mapsto 1, c \mapsto 2$, our example tuple $t = (5, acabacb)$ would correspond to the genotype $[5, 0, 2, 0, 1, 0, 2, 1]$.

A tuple may thus carry a lot of redundant information (namely, all symbols of w that

occur after the ℓ'th symbol). However, note that our genetic representation of DFAs, as explained in Section 2.1, is also a string of integers. We can thus apply the same genetic operators to both kinds of individuals.

We apply mutation to each integer of the genetic representation of a tuple t with a probability of $1/(\ell_{\max} + 1)$. We furthermore use single-point crossover to recombine individuals with a probability of 1. This extreme probability is put into perspective when considering that we interpret the genotypes such that we ignore all integers occurring after the ℓ-th integer. This reduces the impact of the crossover operation.

Similar to the population of DFAs, the population of words is also evolved using NSGA-II. Two objectives are used here. The first objective maximizes the disagreement a word causes among the candidate DFAs, and the second objective minimizes the length of a word. Applying NSGA-II yields a Pareto-optimal set of words. From this set, the word that causes maximum disagreement is selected to serve as a query in the subsequent step.

2.6 The User Oracle

Active learning approaches provide the learner with access to an omniscient oracle. The learner may ask the oracle to label training examples which the learner chooses on its own. Transferred to our scenario where we want to generalize the user's requirements to a desired service, this oracle must be embodied by the user himself, because only the user may disclose additional information about the desired behavior of the service.

Therefore, in addition to the genetic representation and the transformation from SSDs to words, we need a reverse transformation from words back to SSDs such that the latter can be presented to the user as sketched in Figure 4. As can be seen in this figure, the SSD of concern is shown to the user. The user has the usual two options of classifying the presented SSD as prohibited or as desired behavior. The SSD, together with the label provided by the user, is fed back to the algorithm as a new training example.

3 First Apply Knowledge Then Query (FAKT/Q)

As already pointed out in the previous section, the user is the only one who can reveal more knowledge about the desired functionality of the service. Unfortunately, answering tens, hundreds or even thousands of queries is a tedious and cumbersome task. So far, we have reduced the problem of generalizing a protocol from SSDs to grammatical inference using multi-objective optimization and active learning, and by regarding SSDs as words over an alphabet of operation names. While doing this, we have ignored the particular meaning of these operation names.

However, there might be some sequences of operation names that are infeasible with respect to the semantics of the corresponding operations. For instance, an item cannot be added if it has not been created first. If such background knowledge is available in a formalized way, this can be used to exclude sequences of operation names, thereby reducing the load on the user. In particular, this would avoid asking the user to label some SSD for which it is not possible (according to the background knowledge) to implement the described behavior.

The most prominent phrase of Sherlock Holmes:

> "[...] when you have excluded the impossible, whatever remains, however improbable, must be the truth."

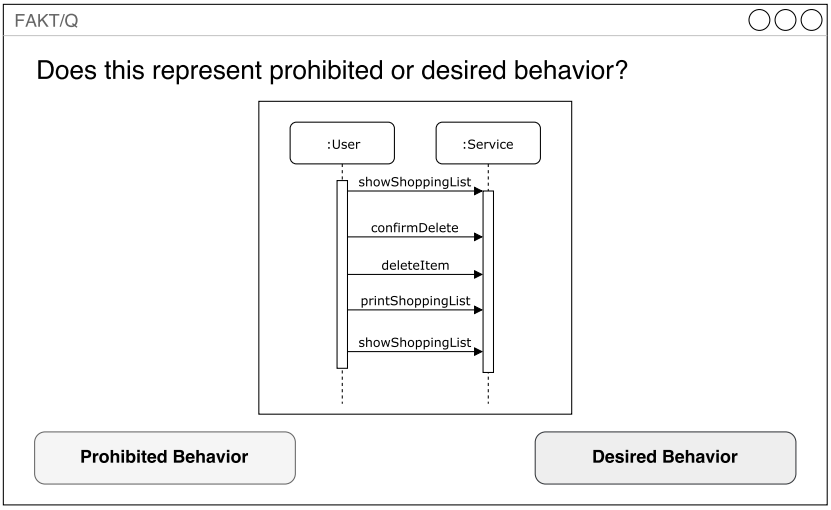


Figure 4: Mockup of a user interface asking the user to label a word that has been transformed into a Simplified Sequence Diagram.

by Sir Arthur Conan Doyle (*The Adventure of the Beryl Coronet*, p. 315) can be adapted to our scenario and can be rephrased to "*when you have excluded the technically infeasible, whatever remains may be the wish of a potential user.*"

That is, we take the semantics of operations into account in order to avoid bothering the user with queries that correspond to infeasible behavior. Knowledge about this feasibility can be made accessible to the grammatical inference process via an additional oracle without increasing the complexity of neither the genetic representation nor the genetic operators. The introduction of this additional oracle, to which we refer as *knowledge oracle*, extends the prior active learning approach to a proactive learning approach (Donmez and Carbonell, 2008).

3.1 The Knowledge Oracle

Up to now, we have regarded the operation names occurring in SSDs as abstract terms. Taking the semantics of these operation names into account, however, leads to more senseful specifications. To this end, we introduce the so-called knowledge oracle (KO) that provides such information to our learner.

We assume that domain knowledge about the operations is already available in a formalized way and we assume that the KO is capable of using this knowledge. We make no further assumptions to the formalized domain knowledge so that, for instance, it could be represented in terms of temporal logic as known from the field of model checking. Another possibility would be an ontology that assigns each operation name to a certain interface description (e.g., input and output parameters together with pre- and postconditions), which is used in combination with a matcher testing whether the interfaces can be plugged together. In the most abstract way, this domain knowledge could be modeled in terms of a super language of the language to be learned.

In proactive learning, the learner is given access to multiple heterogeneous oracles

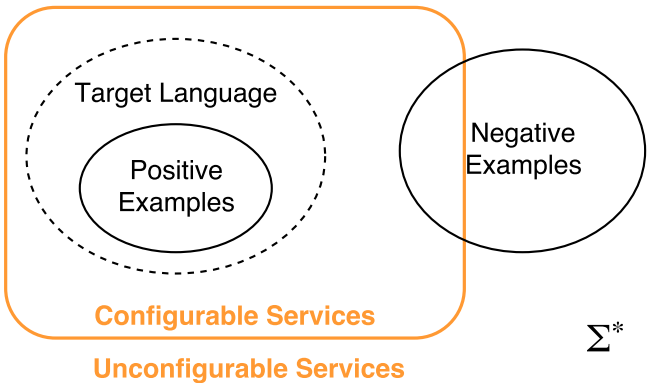


Figure 5: Illustration of the sets involved when learning with a User Oracle (black) and a Knowledge Oracle (orange).

that may be cost-sensitive and imperfect (Donmez and Carbonell, 2008). Comparing the user oracle to the KO, we find substantial differences in the definitions and respective assumptions on these oracles. On one hand, we assume that the user oracle answers queries perfectly, but is also associated with high costs, as we do not want to overburden the user. On the other hand, the KO is an automated oracle and thus much cheaper to query, at least in terms of time. Note that this does not mean that the set of all rejected words can be computed in reasonable time, but a specific word can be "verified." However, the KO has no information about the individual wishes of the user. Hence, the KO can only be used to "knock out" words on the ground of non-realizability or technical infeasibility, lending the abbreviation KO a dual meaning.

The use of domain knowledge via the KO in order to exclude words can be motivated easily. Figure 5 shows Σ^*, the set of all possible words over an alphabet Σ. Within this set, there is a subset representing the target language. From the latter, the user may provide a subset of positive examples, whereas the negative examples are taken from the complement of the target language. While the positive examples are contained in the target language, and thus, as we assume, in the set of configurable services, each negative example could either lie in the set of configurable services or in its complement.

The words outside the set of configurable services can be excluded by the KO. This means that the user will no longer be asked to label such words. In particular, unconfigurable negative examples about which the user would have been queried in our previous approach are now labeled by the KO. Clearly, the information that a word is contained in the set of configurable services is not sufficient to deduce that the word would be part of the target language, that is, that it would represent the requirements of the user. Therefore, queries that are answered positively by the KO should still be answered by the user.

3.2 FAKT/Q Algorithm

Previously, we have proposed a multi-objective version of the Estimation-Exploration Algorithm (EEA) that includes the user as an oracle (Wever et al., 2017). As we want to decrease the workload on the user as an oracle, we first query the knowledge oracle (and

thus, apply the available knowledge) before querying the user on a particular word. If the information obtained from the knowledge oracle implies that the queried word is not realizable or is technically infeasible, we do not need to bother the user with this query. Otherwise, the user is subsequently queried for the label of the word. The entire procedure ranging from initially receiving two sets of SSDs to returning the resulting set of DFAs to the user can be summarized as follows.

1. The initially provided, labeled SSDs are transformed into labeled words forming the set of training examples S.
2. A population of DFAs is randomly initialized and evolved for g generations. The set S of training data is used in order to assess the accuracy of the DFAs with respect to the training examples.
3. Repeat until the user has been queried the specified number of allowed queries:
   3.1. Randomly initialize a population of words and evolve them for g generations. The Pareto front of the evolved DFAs forms the committee for computing the disagreement fitness.
   3.2. Query the knowledge oracle for the word of the Pareto front with highest fitness in disagreement. If the knowledge oracle labels the word as to be rejected, add the word together with the label REJECT as a new training example to S. Otherwise, query the user about the word, and augment S with the additional training example.
   3.3. Reinitialize the population of DFAs keeping the Pareto front and evolve the reinitialized population for g generations.
4. Return the Pareto front of DFAs to the user and terminate.

Instead of requesting only a single word, we could try to leverage the knowledge oracle even more and query for more words. We could consider querying the knowledge oracle for the Pareto front or even the entire population of words in step 3.2. This may easily lead to a highly imbalanced set of training examples, since the knowledge oracle can only determine whether words should be rejected because of technical infeasibility. Determining whether a word should be accepted can only be done by the user oracle. Therefore, querying the knowledge oracle for more words, can lead to words with a REJECT label dominating the training data.

However, since words with the ACCEPT label describe the actually desired and thus required behavior, another objective function that can cope with such imbalanced data is required. To this end, we let the training data that carries the label ACCEPT define the positive class and compute, for a given candidate DFA, the values true positives (TP), false negatives (FN), and false positives (FP). For example, the value FP is the number of words from the training data that the candidate DFA classifies as ACCEPT, while their label is REJECT. From these values, we compute the precision

$$\text{PRECISION} = \frac{\text{TP}}{\text{TP} + \text{FP}}\,, \tag{4}$$

which accounts for the number of words correctly being classified as ACCEPT out of all the words from the training data that were classified as ACCEPT. Furthermore, we compute the recall

$$\text{RECALL} = \frac{\text{TP}}{\text{TP} + \text{FN}}\,, \tag{5}$$

which considers the portion of words that have been accepted out of all words from the training data that should be accepted. Precision and recall can then be combined into the F1 score (also known as harmonic mean)

$$f_{F1} = \frac{2 \times \text{PRECISION} \times \text{RECALL}}{\text{PRECISION} + \text{RECALL}}. \tag{6}$$

In order to obtain a good F1-score, the set of correctly classified training examples with label ACCEPT must not be 0, because otherwise the F1-score would be 0 as well. In contrast to f_{F1}, the fitness function f_{all} may be misleading on unbalanced training data, since a model that always returns REJECT seems to perform well on the training examples if the majority of training examples has label REJECT.

4 Evaluation

Our evaluation is twofold. First, we evaluate the effect of leveraging the second oracle across different problem sizes. Second, a more detailed analysis varying the objective functions, the amount of requested words, and the potential of the knowledge oracle for a fixed problem size is provided.

In the first part, we assess how the second oracle (the knowledge oracle proposed in Section 3.1) affects the test accuracy of the evolved DFAs. To this end, we compare to a slightly adapted version of our active coevolutionary multi-objective optimization approach (FAKT/Q-0) as proposed by Wever et al. (2017) and the Estimation-Exploration Algorithm (EEA) introduced by Bongard and Lipson (2005b) as baselines.

Subsequently, in the second part, we analyze the effect of exploiting the knowledge oracle. For this, we allow the different knowledge oracles to be queried either for a single word, the Pareto front, or the entire population of words. Furthermore, we compare the usage of f_{F1} to the previously proposed f_{all} and discuss the limitations of the study.

4.1 Experiment Setup and Execution

We start our evaluation assessing the influence of including additional knowledge via a knowledge oracle in settings of different search space complexity. To this end, we compare the results to the former version of our approach (referred to as FAKT/Q-0) and to EEA. In the second part, we proceed by only comparing the different variations of FAKT/Q to each other.

In order to evaluate our approach, we have to generate a DFA that is to be learned. This DFA, called the target model, is also used as the user oracle. For given values of $|Q|$ and $|\Sigma|$, the DFA is constructed as follows. Initially, set $Q = \{0, \ldots, |Q| - 1\}$, $\Sigma = \{0, \ldots, |\Sigma| - 1\}$, $q_0 = 0$ and construct δ by drawing states uniformly at random from Q for each pair in $Q \times \Sigma$. If there is a state in Q that is not reachable from q_0, we construct a new δ from scratch until every state is reachable. Otherwise we proceed and add every state $q \in Q$ to F with probability 0.5. Finally, we get $A_{\text{target}} := (\Sigma, Q, \delta, q_0, F)$.

Note that the generation routine does not yield a uniform distribution over the space of DFAs and it does not guarantee minimality of the returned DFAs. So, a DFA A' may exist with less states than A_{target} which recognizes the same language. Therefore, $|Q|$ is to be understood as an upper bound on the number of states that are required to represent the target model.

In the next step, we generate a model for the knowledge oracle, by constructing a super language from the generated target model A_{target}. We do this in the simplest

way possible, by adding states of A_{target} to the set of accepting states. Intuitively, this means that we evaluate the effect of the knowledge oracle (KO) with respect to different overlaps of the set of *negative examples* and the space of *unconfigurable services* as depicted in Figure 5. Given A_{target} and a number r that says how many states should remain rejecting states, we create a copy $A_{\text{valid}} := (\Sigma, Q, \delta, q_0, F_{\text{valid}}) \leftarrow A_{\text{target}}$, and we define $R := Q \setminus F_{\text{valid}}$ to be the set of rejecting states. As long as $|R| > r$, we iteratively choose one of the states from R uniformly at random and add it to F_{valid}.

The value $|Q|$ is known to the learning algorithms. The algorithms are initially provided with 10 training examples that are drawn uniformly at random from the set of all words which have a maximum length of $\ell_{\text{max}} = 20$. Provided these training examples, the different algorithms are run with access to the respective oracles.

We prepare the conduction of the experiments in two steps. First, for all combinations of $|Q| \in \{3, \ldots, 10\}$, $|\Sigma| \in \{3, \ldots, 10\}$ a target model A_{target} is generated for 50 samples with the generation routine described above. Second, from the generated target models, we generate knowledge oracles using the routine for constructing a super language model A_{valid} as explained above for all values of the number of rejecting states $r \in \{0, 1, 2, 3\}$.

The first part of our evaluation is carried out by applying the algorithms EEA and FAKT/Q. Both are provided with access to the user oracle, and FAKT/Q also gets access to a knowledge oracle. For FAKT/Q combined with a knowledge oracle with r rejecting states, we write FAKT/Q-r. For this part, we carry out $16,000$ experiments.

For the second part of our evaluation, we fix both the number of states and the number of input symbols to 8. Moreover, to assess the quality of models, we use either f_{all} or f_{F1} as an objective function. The KO is queried either for a single word (denoted by 'single'), the Pareto front (denoted by 'pareto') or the entire population of words (denoted by 'all'). Evaluating this for FAKT/Q-r knowledge oracles with $r \in \{0, 1, 2, 3\}$ and repeating each setting 50 times, we do another $1,200$ experiments which sums up to $17,200$ experiments in total.

All the algorithms obtain the same set of initial training examples and $|Q|$. The population size of all the approaches is limited to 100 individuals and in each phase, the individuals are evolved for only 5 generations to avoid overspecialized individuals.

As we only want to compare the performance of the different algorithms, we allow them to query the user oracle for $1,000$ additional and self-chosen training examples. In order to assess and compare the quality of the solutions returned by EEA and FAKT/Q, a post-evaluation is performed, testing the returned solutions on $20,000$ examples that have not been used in the evolutionary run and to which we refer as *test data* in the following.

The performance of the best individuals of the last generation for each setting was tested for significance using the Wilcoxon Signed Rank Test with a p-value of 0.05.

4.2 Results for the Effect of Leveraging the Knowledge Oracle

Our evaluation starts with comparing our approach, as presented in Section 3, with different knowledge oracle configurations to the slightly adapted version of the Estimation-Exploration Algorithm as introduced by Bongard and Lipson (2005b). One of these knowledge oracle configurations (namely FAKT/Q-0, in which the KO does not entail any extra information) corresponds to our approach without KO from (Wever et al., 2017). The heatmaps in Figure 6 illustrate the difference in the accuracy (performance difference) with respect to the test data after 100 (Figures 6a-6d) and after 1000 user queries (Figures 6e-6h) of FAKT/Q and EEA. We plot the alphabet size on the horizon-

tal axis against the number of states on the vertical axis. A darker red indicates a greater difference between the two accuracies measured for the test data. Here, a higher difference denotes an advantage of FAKT/Q-r (r is the number of rejecting states) over EEA. Note that the color scales are only fixed for heatmaps with the same number of user queries.

Considering the coloring in the Figures 6a to 6d, we observe that the performance difference between EEA and FAKT/Q is larger if the knowledge oracle has more rejecting states. That is, if the domain knowledge allows to exclude many words as these are not realizable, the algorithm is able to take the domain knowledge provided via the knowledge oracle into account, in such a way that the quality of returned solutions can be improved notably. The settings in the light area starting from the bottom left are the easiest scenarios, for which 100 user queries already suffice to infer the complete target DFA. In these cases, additional domain knowledge is not needed. Attached to this area is an area taking a slightly darker red. The red intensifies when more domain knowledge can be applied so that the performance difference reaches up to 18% for some settings in Figure 6d. In the upper right corner the intense red fades again since the complexity of the settings is higher, and expectedly, the performance difference should increase as the number of user queries would be increased.

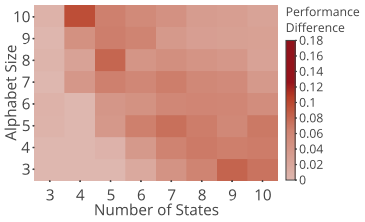
This claim is confirmed in Figures 6e to 6h showing that the performance difference even raises up to a margin of 25%, while the light area starting from the bottom left corner extends to more settings, which indicates that the DFAs in these settings are learned almost completely as the performance difference is close to 0 here.

In Figure 7, we showcase the course of the accuracy on the test data (Test Accuracy) along the vertical axis against the number of queries sent to the user on the horizontal axis for six different combinations of the number of states and the size of the alphabet of the target model. These settings only represent a subset of the evaluated settings, but these plots are representative as the shape of the fitness curves are quite similar for different settings. In all evaluated settings, we observe, as expected, a boost in the test accuracy when additional knowledge is supplied by the knowledge oracle as it is the case for FAKT/Q-1, FAKT/Q-2, and FAKT/Q-3. Even for simpler target models, such as the case of $|Q| = 4$, $|\Sigma| = 7$, we notice steeper learning curves for these configurations of the FAKT/Q algorithm. From the steeper curves, we also conclude that we require less training examples provided by the user in order to achieve the same test accuracy compared to the case when the knowledge oracle provides no additional information (FAKT/Q-0) and compared to EEA. A similar observation is made for the $|Q| = 6$, $|\Sigma| = 6$ setting. This trend continues in the other settings, although the test accuracy does not reach value 1 anymore for $|Q| = 7$, $|\Sigma| = 7$ and the following settings. Another observation is that across the different settings, the shape of the curves is quite similar despite some factors stretching or clinching the course. The question whether this insight can be leveraged for improving the process or providing an estimate for the test accuracy of the returned solution represents interesting future work.

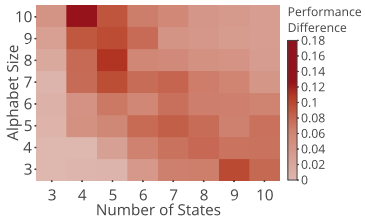
Furthermore, for settings where $|Q| \geq 6$ and $|\Sigma| \geq 6$, we noted that FAKT/Q-r for $r \in \{1, 2, 3\}$ yields significant improvements over the baselines EEA and FAKT/Q-0.

Lastly, we assess the impact of the knowledge oracle on the concrete number of user queries that need to be answered in order to achieve a certain test accuracy. These numbers are illustrated in the form of heat maps in Figure 8, where the number of user queries that is needed to achieve a test accuracy of at least 0.95 is counted. A darker red indicates that more queries were necessary, while white areas represent settings for which a test accuracy of at least 0.95 was not reached.

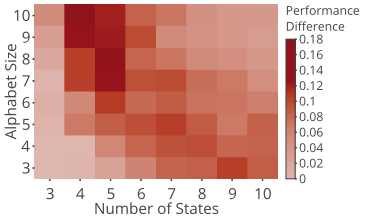
In these heat maps, the differences between the algorithms and knowledge oracle



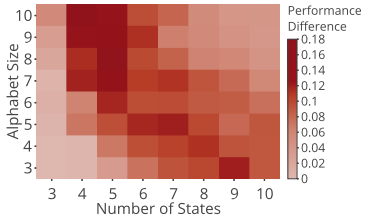
(a) EEA VS FAKT/Q-0, 100 user queries



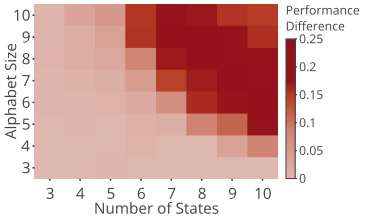
(b) EEA VS FAKT/Q-1, 100 user queries



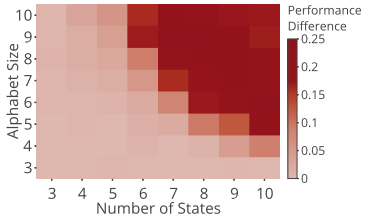
(c) EEA VS FAKT/Q-2, 100 user queries



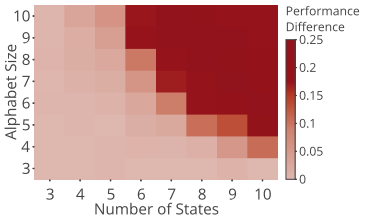
(d) EEA VS FAKT/Q-3, 100 user queries



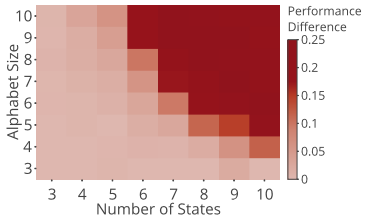
(e) EEA VS FAKT/Q-0, 1,000 user queries



(f) EEA VS FAKT/Q-1, 1,000 user queries

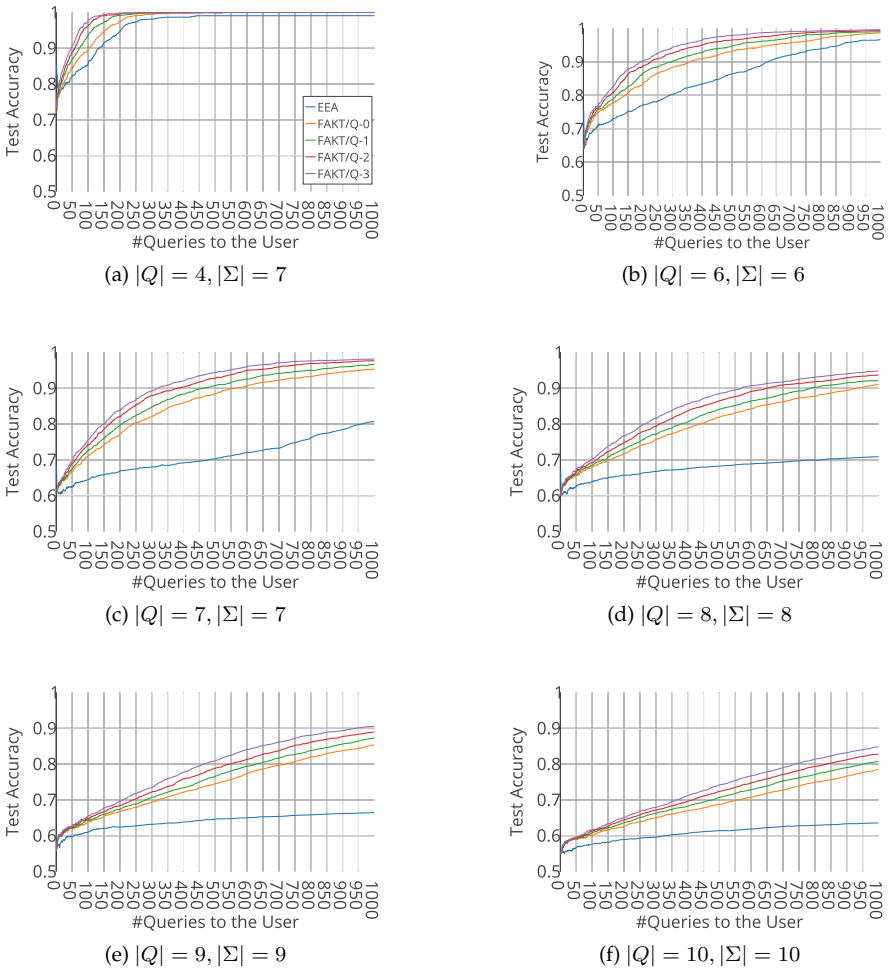


(g) EEA VS FAKT/Q-2, 1,000 user queries



(h) EEA VS FAKT/Q-3, 1,000 user queries

Figure 6: Heat maps showing the Performance Difference (i.e., difference of the test accuracies) of FAKT/Q-r minus EEA after 100 user queries (a-d) and after 1,000 user queries (e-h). A positive difference represents an advantage of FAKT/Q-r over EEA.

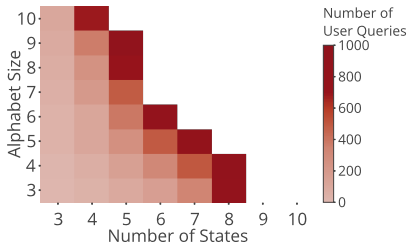


(a) $|Q| = 4, |\Sigma| = 7$ (b) $|Q| = 6, |\Sigma| = 6$

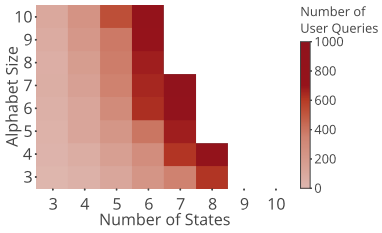
(c) $|Q| = 7, |\Sigma| = 7$ (d) $|Q| = 8, |\Sigma| = 8$

(e) $|Q| = 9, |\Sigma| = 9$ (f) $|Q| = 10, |\Sigma| = 10$

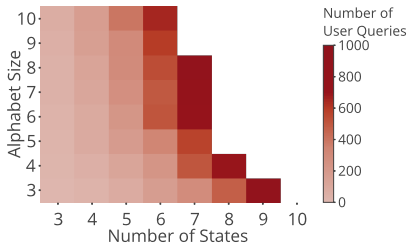
Figure 7: Evolutionary runs with $1,000$ allowed user queries. The averaged test accuracy of best individuals is plotted against the number of queries sent to the user from 50 independent evolutionary runs for different combinations of alphabet size $|\Sigma|$ and number of states $|Q|$.



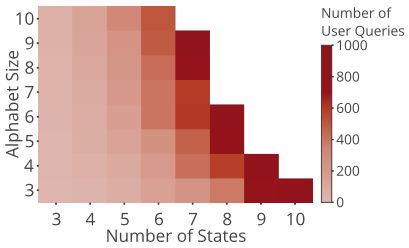
(a) EEA



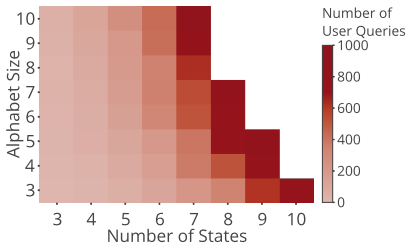
(b) FAKT/Q-0



(c) FAKT/Q-1



(d) FAKT/Q-2



(e) FAKT/Q-3

Figure 8: The heat maps show the number of user queries needed for each approach to reach a test accuracy of at least 0.95. A darker coloring denotes a higher number of queries needed, whereas white spaces show that a test accuracy of 0.95 could not be reached within $1,000$ user queries.

configurations become even more visible. Figure 8a shows that for EEA it is not possible to achieve the required test accuracy in exactly half of the 64 settings, and the boundary of the colored area takes a dark red, indicating that in these settings between 800 and $1,000$ queries to the user were required.

Considering Figure 8b, we notice that the dark red border becomes steeper and that FAKT/Q-0 (multi-objective approach without knowledge oracle) is able to deal with larger alphabet sizes compared to EEA. This can also be seen from the smoothness of the column-wise color gradient in the heat maps. Moreover, from this figure, we can also see that the larger the number of states, the greater the impact of increasing the alphabet size on increasing the difficulty of solving the problem. Moving on to Figure 8c, 8d, and 8e, we find that the boundary is shifted to the right each time we increase the available knowledge that can be provided by the knowledge oracle. In the case of a knowledge oracle with three rejecting states (FAKT/Q-3), the number of settings for which FAKT/Q does not achieve a test accuracy of at least 0.95 is decreased to 15 out of 64 settings.

From this evaluation, we conclude that incorporating additional knowledge via the knowledge oracle is possible and indeed makes a difference. Depending on how restrictive the respective domain knowledge is, the effectiveness of using the knowledge oracle varies. However, we did up to this point not exploit the full potential of the knowledge oracle as we only used it for validating single queries. In the following, we evaluate various settings, in which the knowledge oracle is queried more intensively.

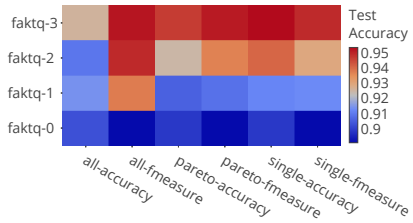
4.3 Detailed Analysis

The results of the evaluation for several variations of FAKT/Q with respect to their number of rejecting states of the knowledge oracle, the objective functions, and amount of requested words are summarized in Figures 9 and Figure 10. While in Figure 9a the final test accuracy after 1000 user queries is shown, Figure 9b illustrates the portion of positive training examples in the training data. The overall observation is that the distribution of positive and negative training examples plays a major role for the generalization behavior of the evolved models.

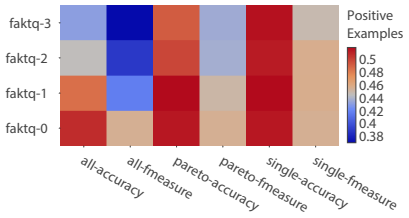
In particular, it can be observed that for single requested words and f_{all}, the training examples requested by FAKT/Q are nearly balanced having a positive training example portion of approximately 0.5, which also corresponds to the expected number of accepting states. Using f_{all} with an increased number of requested words (i.e. with settings 'pareto' and 'all') leads to decreased test accuracy on the one hand and a decreased portion of positive training examples on the other hand.

These results were expected, since we have seen in Section 3.2 that querying the knowledge oracle more leads to unbalanced training data, whereas f_{all} was designed for balanced sets of training data. Figure 9a shows that the additional negative examples are actually harmful when using f_{all}.

To overcome this issue, we introduced f_{F1} as a substitute objective function for f_{all}. Figure 9a shows that in particular for the settings in which a knowledge oracle is queried for the entire word population, FAKT/Q-r for $r > 0$ achieves better performance when using f_{F1} than with f_{all}. Interestingly, the training data tends to become even more imbalanced than in the case of f_{all}, clearly showing that f_{F1} indeed can deal with the highly imbalanced training data. More importantly, with f_{F1} we can make use of the additional amount of data and the knowledge oracle can be exploited effectively. However, in the case of querying the knowledge oracle for a single word only, f_{F1} yields slightly worse results than f_{all}, as for f_{all} the training examples are more or less bal-



(a) Test accuracies



(b) Portion of positive training examples

Figure 9: Left: The final test accuracy after 1000 user queries is shown for different settings. Right: The portion of positive training examples in the training data for these settings is shown. Here, 'accuracy' means f_{all} is used, 'fmeasure' means f_{F1} is used. 'Single', 'pareto' and 'all' refer to the number of words queried to the knowledge oracle.

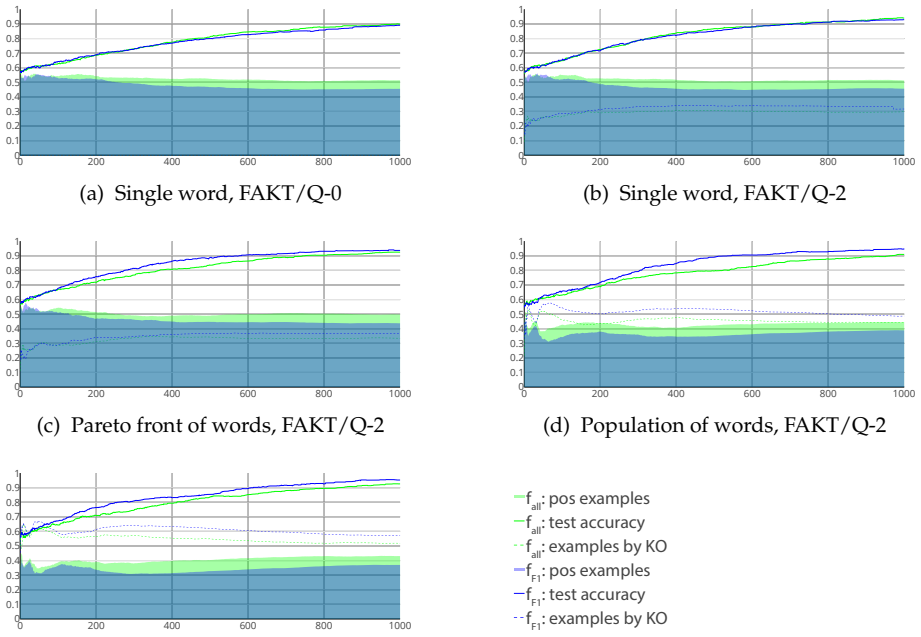
anced.

In Figure 10, a selection of the evaluated settings is presented showing the test accuracy, the portion of positive training examples, and the portion of training examples that are obtained from the knowledge oracle as a function of user requests. In these plots, we observe again that the portion of positive training data decreases as more examples are obtained from the knowledge oracle. Across the different settings, in the case of FAKT/Q with f_{F1}, the portion of positive training examples is substantially smaller than the one using f_{all}.

Interestingly, the portion of positive training examples converges to 0.5, which is also the expected portion of accepting states of our DFA generation routine. Another interesting observation is that in early iterations, the portion of positive training examples forms a wave shape where positive training examples and negative training examples are acquired in an alternating fashion. Comparing to f_{all}, we conclude that f_{F1} proves beneficial in settings where a large number of words are unfeasible or can be excluded due to technical reasons. However, f_{all} is still competitive when the knowledge oracle is queried using Pareto front words.

4.4 Limitations

It is important to note that the experimental evaluation has its limitations due to the way the target DFAs are generated. The target DFAs are generated randomly (cf. Section 4.1) and the generation routine does not give any guarantees on the minimum number of the states that is necessary to induce a DFA with the same language as the target model. More precisely, the generation routine does not necessarily return DFAs with minimum size, such that there might exist another DFA with less states accepting the same language. Hence, the target model might be significantly easier to induce in presumably more difficult settings. This becomes particularly clear when studying the data shown in Figure 11. The mean plus/minus the standard deviation of the test accuracy is plotted against the number of user queries for the setting of 10 states and 10 input symbols. While the mean indicates that FAKT/Q-3 performs clearly better than FAKT/Q-0, the respective areas illustrating the standard deviation around the mean have a huge over-



(a) Single word, FAKT/Q-0

(b) Single word, FAKT/Q-2

(c) Pareto front of words, FAKT/Q-2

(d) Population of words, FAKT/Q-2

(e) Population of words, FAKT/Q-3

Figure 10: Test accuracy (solid line), portion of positive training examples (area), and portion of training examples obtained from the knowledge oracle (dashed line) for f_{all} (green) and f_{F1} (blue) respectively.

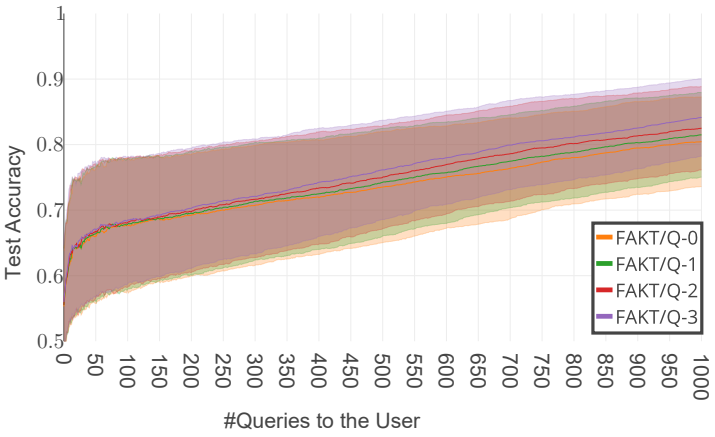


Figure 11: Mean $\pm$ std. deviation of the test accuracy for the setting of number of states $|Q| = 10$ and alphabet size $|\Sigma| = 10$

lap. This is because some models are comparatively easy to induce, that is, there exists a model for the same language with less states. In these cases, less information is provided by the knowledge oracle and less user queries are required to induce a model with high accuracy.

However, in the experiment-wise comparison, that is running the different FAKT/Q instantiations for the same target model, a more informative knowledge oracle still compares preferably over the less informative one and the significance of the average results is preserved due to the pair-wise comparison. Indeed, we suspect that the effect of the knowledge oracle would be even more significant because there is more information to reveal in the case of more complex target models. The design of a more sophisticated generation routine is an important future task.

Another important limitation is the distribution of final states that is uniform in our experiments. More precisely, a state is considered an accepting state with a probability of 0.5. This is a rather arbitrary assumption and not necessarily reasonable for all scenarios. Therefore, the sensitivity of the approaches to imbalanced distributions of final states needs further investigations in future work.

5 Application scenario: On-The-Fly Markets

In this section, we take a brief look at so-called On-The-Fly (OTF) markets[2] as a possible application scenario for our multi-oracle approach to learning behavioral requirements specifications from examples. An OTF market is a software service market of the future, in which users may request customized software services, which do not need to exist yet. Such a requested service may then be created automatically and on the fly using available services in the market as building blocks.

[2] `https://sfb901.uni-paderborn.de/`

In some existing software service markets, such as IFTTT and Stringify[3], services may be manually composed. The visionary idea of an OTF market, however, is that composition of services is done automatically (Mohr et al., 2018). Moreover, the different steps preceding and following this composition are automated as well.

The creation of a new service is initiated by a request from a user in the form of a requirements specification. This specification should represent the wishes of the user, and be formal enough to be processed automatically. In order to make the OTF market accessible to non-technical users, we only ask the user to write down exemplary desired and forbidden behavior. The approach proposed in this paper may then be applied to produce, in a user-friendly manner, a formal behavioral requirements specification.

In this application, the first oracle is impersonated by the user. As explained in Wever et al. (2017), asking the user smart queries about specific behavior of the service allows us to vastly improve the produced requirements specification. Coevolution enables us to find highly informative queries at each moment of the interaction process.

In Wever et al. (2017), however, many user queries were needed in order to produce a requirements specification that accurately represents the desired service. This of course limits the user-friendliness of the approach. The multi-oracle approach taken in this paper solves this problem, by taking into account knowledge from a second source.

As our second oracle, we use knowledge about the feasibility of service compositions. Note that such knowledge should be present in the OTF market, as only feasible compositions can be presented to the users. Using this oracle, we avoid the generation of a requirements specification corresponding to a service that cannot be built in a given OTF market. More importantly, we reduce the number of queries asked to the user, since many queries are already answered by the knowledge oracle.

6 Conclusion

We have shown how the coevolutionary active learning approach to requirements elicitation that we took (Wever et al., 2017) can be extended to a multi-oracle approach. Our previous approach did not take into account the semantics of operation names, but rather viewed these as abstract terms. In this paper, we include knowledge about the semantics of operation names in the form of an additional oracle - the knowledge oracle. This extension does not increase the complexity of the coevolutionary algorithm. Although this oracle can only be used to exclude some behavior due to being non-realizable within the particular domain, we found in Section 4 that involving the knowledge oracle leads to considerable improvements. Especially for greater automaton sizes, the knowledge oracle can be used to substantially reduce the number of queries that need to be sent to the user.

However, since the Smart State Labeling algorithm requires the set of training examples to be balanced in terms of the distribution of accepted words and rejected words, we could not fully leverage the potential of the knowledge oracle when using the canonical fitness function that measures the fraction of correctly classified examples. To resolve this problem, we introduced a new fitness function in the form of an F1-score. This fitness function is able to deal with unbalanced training data, as is confirmed by our observations in Section 4.

In future work, we want to embed the proposed multi-oracle approach and its concepts into an OTF market, and evaluate the effect of the available domain knowledge

[3] `http://ifttt.com`, `http://www.stringify.com`

provided by the knowledge oracle on data with realistic semantics, guiding the user with an appropriate interface.

Another point of transferring the proposed approach to the OTF market scenario is that the inclusion of non-functional requirements (e.g., computing time, reputation of a service, etc.) can be used as additional objectives in our multi-objective approach, which would yield a many-objective approach. Of special interest would then be the selection of the committee for assessing the fitness of the words that serve as queries to the oracles, since we expect the Pareto front to become extremely large in this case.

Acknowledgments

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901).

References

Angluin, D. (1987). Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106.

Arcuri, A. and Yao, X. (2008). A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 162–168. IEEE.

Bongard, J. and Lipson, H. (2005a). Active coevolutionary learning of deterministic finite automata. *ACM Trans. Program. Lang. Syst.*, 6:1651–1678.

Bongard, J. C. and Lipson, H. (2005b). Active coevolutionary learning of deterministic finite automata. *Journal of Machine Learning Research*, 6:1651–1678.

de la Higuera, C. (2005). A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348.

de la Higuera, C. (2010). *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA.

Deb, K. (1998). An efficient constraint handling method for genetic algorithms. In *Computer Methods in Applied Mechanics and Engineering*, pages 311–338.

Deb, K., Agrawal, S., Pratap, A., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation*, 6(2):182–197.

Donmez, P. and Carbonell, J. G. (2008). Proactive learning: cost-sensitive active learning with multiple imperfect oracles. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management, CIKM 2008, Napa Valley, California, USA, October 26-30, 2008*, pages 619–628.

Faunes, M., Sahraoui, H., and Boukadoum, M. (2013). Genetic-programming approach to learn model transformation rules from examples. In Duddy, K. and Kappel, G., editors, *Theory and Practice of Model Transformations: 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, pages 17–32. Springer, Berlin, Heidelberg.

Gold, E. M. (1967). Language identification in the limit. *Information and Control*, 10(5):447–474.

Gómez, J. (2006). An incremental-evolutionary approach for learning deterministic finite automata. In *IEEE International Conference on Evolutionary Computation, CEC 2006, part of WCCI 2006, Vancouver, BC, Canada, 16-21 July 2006*, pages 362–369.

Harel, D., Kugler, H., and Pnueli, A. (2005). Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNCS*, pages 309–324. Springer Berlin Heidelberg.

Harman, M., Mansouri, S. A., and Zhang, Y. (2012). Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.*, 45(1):11.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. Univ. Michigan Press, Ann Arbor, MI.

Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., and Wimmer, M. (2012). Model transformation by-example: A survey of the first wave. In Düsterhöft, A., Klettke, M., and Schewe, K.-D., editors, *Conceptual Modelling and Its Theoretical Foundations*, pages 197–215. Springer, Berlin, Heidelberg.

Kessentini, M., Sahraoui, H., Boukadoum, M., and Ben Omar, O. (2012). Search-based model transformation by example. *Software & Systems Modeling*, 11(2):209–226.

Kühne, T., Hamann, H., Arifulina, S., and Engels, G. (2016). Patterns for constructing mutation operators: Limiting the search space in a software engineering application. In *Applications of Evolutionary Computation (EvoApplications 2016)*, volume 9594 of *LNCS*, pages 278–293. Springer.

Lambeau, B., Damas, C., and Dupont, P. (2008). State-merging DFA induction algorithms with mandatory merge constraints. In *Grammatical Inference: Algorithms and Applications, 9th International Colloquium, ICGI 2008, Saint-Malo, France, September 22-24, 2008, Proceedings*, pages 139–153.

Lang, K. J., Pearlmutter, B. A., and Price, R. A. (1998). Results of the abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *Grammatical Inference, 4th International Colloquium, ICGI-98, Ames, Iowa, USA, July 12-14, 1998, Proceedings*, pages 1–12.

Langdon, W. B. and Harman, M. (2015). Optimizing existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135.

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

Lucas, S. M. and Reynolds, T. J. (2005). Learning deterministic finite automata with a smart state labeling evolutionary algorithm. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(7):1063–1074.

Ly, D. L. and Lipson, H. (2014). Optimal experiment design for coevolutionary active learning. *IEEE Trans. Evolutionary Computation*, 18(3):394–404.

Mäkinen, E. and Systä, T. (2000). An interactive approach for synthesizing UML statechart diagrams from sequence diagrams. In *Proceedings of OOPSLA 2000 Workshop: Scenario based round-trip engineering*, pages 7–12.

Mohr, F., Wever, M., and Hüllermeier, E. (2018). On-the-fly service construction with prototypes. In *2018 IEEE International Conference on Services Computing, SCC 2018, San Francisco, CA, USA, July 2-7, 2018*, pages 225–232.

OMG (2011). OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1.

Popovici, E., Bucci, A., Wiegand, R. P., and De Jong, E. D. (2012). *Coevolutionary Principles*, pages 987–1033. Springer Berlin Heidelberg, Berlin, Heidelberg.

Settles, B. (2012). *Active Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Tsarev, F. and Egorov, K. (2011). Finite state machine induction using genetic algorithm based on testing and model checking. In *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Companion Material Proceedings, Dublin, Ireland, July 12-16, 2011*, pages 759–762.

Wever, M., van Rooijen, L., and Hamann, H. (2017). Active coevolutionary learning of requirements specifications from examples. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO'17, pages 1327–1334, New York, NY, USA. ACM.

Wiegand, R. P. (2003). *An analysis of cooperative coevolutionary algorithms*. PhD thesis, George Mason University Virginia.

Wiegand, R. P., Liles, W. C., and Jong, K. A. D. (2001). An empirical analysis of collaboration methods in cooperative coevolutionary algorithms. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, pages 1235–1242. Morgan Kaufmann Publishers Inc.

Zhang, Y. and Harman, M. (2010). Search based optimization of requirements interaction management. In *Second International Symposium on Search Based Software Engineering (SSBSE)*, pages 47–56. IEEE.