Active Learning of User Requirements Specifications in Dynamic Software Service Markets

Master Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

MARCEL DOMINIK WEVER
Friedrich-Ebert-Str. 46
33102 Paderborn

Thesis supervisor
Jun-Prof. Dr. Heiko Hamann
and
Dr. Lorijn van Rooijen

Paderborn, March 2017

Declaration

(Translation from German)

I hereby declare that I prepared this thesis entirely on my own and have not used outside sources without declaration in the text. Any concepts or quotations applicable to these sources are clearly attributed to them. This thesis has not been submitted in the same or substantially similar version, not even in part, to any other authority for grading and has not been published elsewhere.

Original Declaration Text in German:

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

City, Date	Signature

Abstract

In software engineering approaches, the first step of the software development process is dedicated to the elicitation of requirements usually created by a software engineering expert. Customers are usually supported by a requirements engineer who compiles the customers' vague and imprecise notions to a formal requirements specification. We try to substitute the software engineering expert by a domain expert capable of providing sequence diagrams exemplarily describing the behavior of the desired software. The sequence diagrams are partitioned into two sets, one describing desired behavior and one describing prohibited behavior. We search for a protocol in the form of a deterministic finite automaton formalizing the behavioral description also known as grammatical inference. Following Bongard and Lipson [BL05], we extend the approach by Rooijen and Hamann [RH16] to an active co-evolutionary approach using multi-objective optimization in both phases. One evolutionary process proposes additional training examples to be labeled by the domain expert, and the other one evolves suitable automata with respect to the training examples. Hence, supporting the domain expert in refining the provided examples and in providing more useful examples, we try to substitute the human requirements engineer by an automatic requirements engineer. Considering this approach in the context of dynamic software service markets, we propose additional concepts of refining the inferred automata by guards known from UML state machines and In order to benchmark the approach and compare it to [RH16] and [BL05], we introduce a Java framework called Requirements Elicitation by Active Learning framework. We find our approach significantly outperforming the approach by [RH16]. Furthermore, we outperform the state-of-the-art technique [BL05] for input alphabet sizes of more than three.

Contents

List of Figures

List of Tables

1 Introduction

A development process for a software system starts with the specification of requirements for this system. However, the customers often only have a vague and imprecise notion of their requirements to the software system. To this end, customers are usually supported by requirements engineers. Since the requirements specification forms a fundamental basis for the subsequent steps in the development process, the imprecise notions have to be refined to a more formal description of the software system. By interviewing the customers systematically, the requirements engineer compiles the customer's needs and notions to a precise and formal requirements specification. The latter may be accompanied by formal models as defined in the Unified Modeling Language.

This thesis deals with the vision of semi-automating the process of requirements elicitation and substituting the human requirements engineer by an automatic requirements engineer. To this end, we develop an interactive evolutionary algorithm, formalizing examples provided by the customer and supporting the customer to provide more useful data. More precisely, we use search-based software engineering for generalizing exemplary models to a formal requirements specification.

In search-based software engineering (SBSE), metaheuristic search approaches, such as evolutionary algorithms, are applied to address problems in software engineering. These problems range from requirements and organizational tasks to operations and maintenance [HMZ12]. Involving multiple competing objectives, these problems are usually associated with a complex problem space. By using SBSE, adaptive solutions to these problems are obtained in an automated or semi-automated way. For instance, search algorithms can be applied to the problem of model transformation. Defining abstract rules for transforming an instance of one model into an instance of another model is an exhaustive and complex task. In [HKP05; Kes+12; Kap+12; Küh+16], SBSE is used to deduce transformation rules from observing example model transformations.

Transforming sequence diagrams into deterministic finite automata, van Rooijen and Hamann [RH16] proposed an approach addressing the problem of requirements elicitation by SBSE. Directed at customers who are considered to be domain experts and who are able to specify their requirements to a software system in the form of exemplary sequence diagrams, these sequence diagrams are generalized to a deterministic finite automaton (DFA). The DFA represents a simplified version of a protocol state machine, constituting a requirements specification which is learned from observing examples. The transformation rules and dependencies leading from the sequence diagrams to the generalized protocol need not be understood to obtain good solutions. Through the nature of the problem, such approaches are called black-box optimizers.

In the Collaborative Research Center 901 “On-the-fly Computing”, we pursue the vision of a dynamic software service market, in which users are offered automatically assembled software services on demand [Col11]. A software service encapsulates a set of operations, described by inputs and outputs, as well as preconditions and effects in the form of first-order predicate logic [Moh16]. In order to enable automatic configuration and composition of services, a formal description of services is a crucial requirement. Corresponding to the Service Specification Language [Pla16], the formal description is structured in two major parts: a functional and a non-functional description. The functional description contains a declaration of operation interfaces by inputs, outputs, preconditions, and effects. Additionally, a protocol in the form of a deterministic finite automaton is stated, describing the usage of the declared operations to accomplish the task of the service. Furthermore, non-functional requirements such as the price or the reputation of the declared operations belong to the service description.

Conversely, this means in order to make use of the services of automated service composition and configuration, a formal requirements specification for the desired service is inevitable. Hence, a customer is required to state a formal service description of the desired service. To this end, the customer needs to have a technical background for creating such a service description. Additionally and more importantly, knowledge of available operations is required to declare these in the service description.

However, we hope to address this problem, providing a more user-friendly way of specifying such service descriptions for customers who are domain experts. We assume domain experts to be capable of expressing their requirements in the form of sequence diagrams, which describe the interaction between the user and the desired service in terms of examples. Within the sequence diagrams, the domain expert is allowed to use arbitrary operation names. Applying the SBSE approach by van Rooijen and Hamann [RH16], a protocol is generalized from the provided examples. In a further step, the arbitrarily chosen operation names of the domain expert are mapped to operations that exist in the dynamic software service market. The mapping of the operation names is still an open task and does not form a part of this thesis. Taken together, this yields the functional description of the desired service.

In order to generalize a protocol from the provided sequence diagrams, van Rooijen and Hamann [RH16] reduce this problem to the problem of inferring a deterministic finite automaton for some language from exemplary words. Generally, this problem is also known as *grammatical inference*. The Evidence-Driven State Merging (EDSM) algorithm is one of the most well-known and popular algorithms to address the problem of grammatical inference [LPP98]. Deterministically, EDSM builds a prefix tree for the obtained training data as an initial DFA and merges states depending on how much evidence the training data contains concerning this decision. However, Lucas and Reynolds [LR03; LR05] have shown that evolutionary approaches lead to promising results and even outperform EDSM for target models of less than 32 states.

Unfortunately, the amount of data that is needed to obtain good results is too extensive, since the domain expert is required to provide this amount of examples in the form of sequence diagrams. Moreover, customers are usually supported by requirements engineers in the elicitation of their requirements. As already mentioned before, requirements elicitation is an iterative process involving a systematic refinement from imprecise and abstract notions to explicit functional and non-functional requirements. Therefore, the customer needs to be supported by a type of automatic requirements engineer, which asks the customer for feedback and supports the customer in providing more useful examples.

Empowering a learning algorithm to choose training examples autonomously, in active learning approaches, the algorithm is given access to an oracle for labeling data points in order to form new training examples. In this way, the training data can be augmented strategically. For instance, Bongard and Lipson [BL05] combine active learning and an evolutionary algorithm for grammatical inference to an interactive approach, where queries to the oracle are chosen according to how much disagreement they produce among already evolved models. In experiments the authors showed their active approach to outperform other approaches, providing the same total amount of training data according to a uniform distribution. In particular, compared to the other approaches, the active approach by [BL05] needs less training examples to obtain a solution.

In the course of this thesis, we combine the approach of [RH16] and [BL05] to an active evolutionary learning approach for requirements elicitation. Considering the domain expert as an oracle, we hope to reduce the number of training examples, which need to be provided by the domain expert. Furthermore, by delegating the choice of which training examples are provided to the algorithm, the domain expert is supported by the algorithm in refining the already provided examples. In this way, the algorithm takes over the roles of an automatic requirements engineer.

1.1 Structure of Work

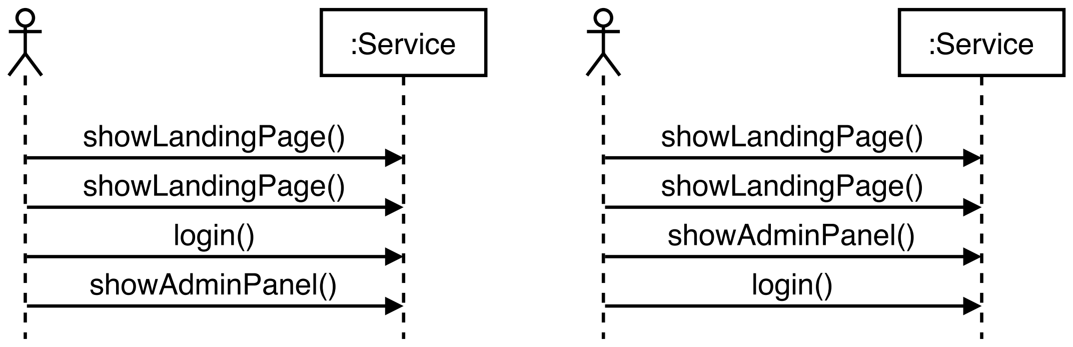
In the remainder of this chapter, we briefly introduce a running example which is used for illustrating definitions and concepts in the discourse of this thesis. Chapter 2 is dedicated to the fundamental basics regarding models, dynamic software service markets, evolutionary algorithms, and active learning. Related work is outlined in Chapter 3 on automated requirements engineering and learning deterministic finite automata from exemplary words. The core of this thesis is subsequently given in Chapter 4, where we present approaches for interactively learning requirements specifications from examples. Chapter 5 is subject to the implementation part of this thesis. In Chapter 6 we experimentally evaluate the approaches presented in Chapter 4, including a comparison between the state-of-the-art approach and our approach. Finally, this thesis is concluded by Chapter 7 with a summary of this thesis and an outlook.

1.2 Running Example: Shop Management Service

Throughout this thesis, we use a running example that we use to illustrate the concepts and approaches we discuss. For this purpose, we consider the case that a domain expert wants to specify the requirements for a shop management system.

The service we want to describe initially shows a landing page that can be refreshed infinitely often. To enable more features such as showing an overview panel for the shop administrator, a login is required. On a successful login, the shop administration panel is shown. We also demand that the administration panel is only accessible after the login. The domain expert may exemplarily model these requirements by sequence diagrams as depicted in Figure 1.1a and 1.1a.

A deterministic finite automaton representing a possible outcome of the requirements elicitation process is shown in 1.2. The automaton defines a protocol for applying operations according to the above-stated requirements.



(a) Example sequence diagram for desired behavior

(b) Example sequence diagram for prohibited behavior

Figure 1.1: Example: Sequence diagrams exemplarily modeling the behavior of a shop management system

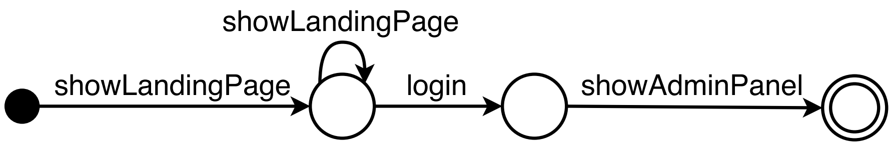


Figure 1.2: Example: protocol for the shop management service

In the course of this thesis, we extend the example above introducing an alternative path. Depending on the type of the logged in user, either an administration panel or a customer panel is shown. The extended deterministic finite automaton is depicted in 1.3.

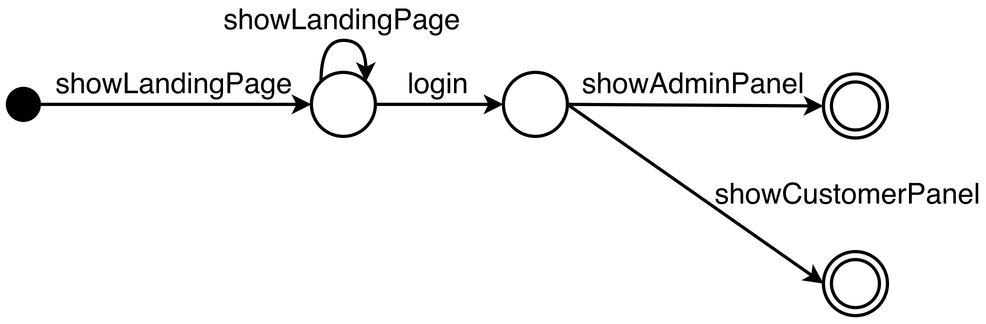


Figure 1.3: Example: extended protocol for the shop management service

2 Preliminaries

A software engineering process starts with the requirements elicitation phase, in which requirements for the software are formalized. The Unified Modeling Language (UML) is a standardized collection of such formalisms. Therefore, in Section 2.1, a brief description of the UML and two of its formalisms are provided, i.a. an enhanced implementation of deterministic finite automata is described. These cannot only be used for specifying the behavior of entire software systems, but also software services as building blocks for software systems. For instance, in the on-the-fly market (OTF market), these are used to describe the semantics of software services as well as to specify requests for services. The central concepts of the OTF market are explained in Section 2.2.

In order to facilitate the request specification for users without involving a requirements engineer, the user is only asked to describe exemplary behavior. For generalizing these examples to a formal description of a service, this task is reduced to the problem of grammatical inference, which is presented in Section 2.3. Section 2.4 provides an overview of evolutionary algorithms, which can be applied to solve problems of grammatical inference. Extending evolutionary algorithms to interactive algorithms by using concepts of the field of active learning, which is outlined in Section 2.5, the algorithm is given the opportunity to refine the generalization of the examples iteratively, while at the same time the algorithm supports the user in providing more useful data.

2.1 Unified Modeling Language

The Unified Modeling Language (UML) is a standard for modeling software, which is maintained by the Object Management Group (OMG). In the UML standard (see [Obj15]), two groups of models are defined: one for describing the behavior and one for modeling the structure of software. Due to the explicit and clearly defined semantics, models of the UML can be used to find a consensus between the notions of the client and the contractor. Usually, the requirements specification, which forms the contractual basis for engaging a software provider, is accompanied by such models. On top of that, the models represent a blueprint of the desired software for the software developer.

In the following, we consider two examples of model types for describing the behavior of software, standardized in the UML specification. Section 2.1.1 deals with state machines, which can be used for modeling the behavior of software systems in terms of states and interactions resulting in a change of state. In Section 2.1.2, we consider another model type, which can be used to exemplify sequences of interaction between the user, the software system, external software systems, as well as entities within these software systems.

2.1.1 State Machines

A state machine (SM), also known as statechart, is dedicated to model behavior of a software component in terms of states and transitions, leading from one state to another. Based on the concepts of statecharts by Harel [Har87], UML state machines extend the formalism of deterministic finite automata by introducing new concepts as hierarchical states and additional annotations for transitions.

For instance, the usage of a transition may be refined by annotating it with a so-called guard. A guard is a boolean expression, which is required to evaluate to `true` before the transition is allowed to be used. Therefore, by the usage of guards, it is possible to refine the semantics of transitions and states.

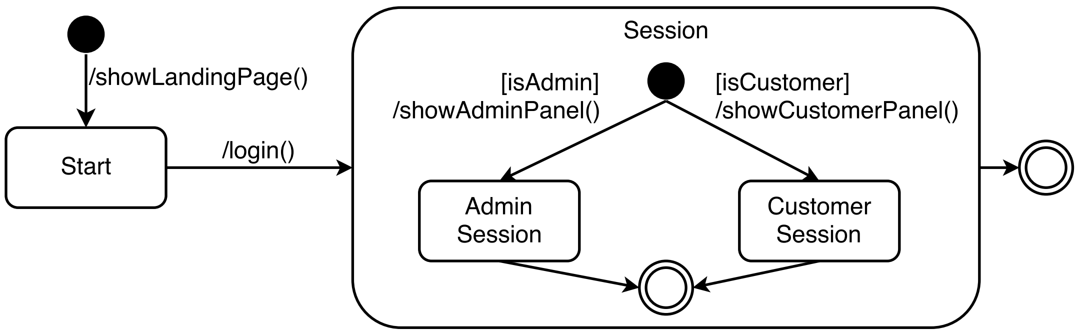


Figure 2.1: Example statechart with a hierarchical state, guards, and actions

In Figure 2.1, an example for a statechart is shown, modeling an extended version of our running example. The example statechart shows transitions which are labeled with so-called actions. These actions may represent operation calls or other events. In our scenario, we use operation calls, of which the `/login()` operation call leads to a hierarchical state encapsulating two further states. The transitions to the sub-states are refined by guards `[isAdmin]` and `[isCustomer]`.

2.1.2 Sequence Diagrams

Sequence diagrams model the interaction between the user and the software system, as well as between different entities of the software system. A sequence diagram consists of several vertical lines, so-called lifelines, each corresponding to one object, which is either a user or an instance of some class of the software system. Arrows which are drawn between those lifelines model messages, which are sent from the object that the arrows starts. The end of the arrow marks the recipient of that message. Messages are also referred to as events or operation calls. In order to model that a certain operation call or event is sent, the corresponding arrow is annotated with the name of the operation or event.

Moreover, the lifelines also denote the flow of time such that the arrangement of arrows also denotes their temporal order. The temporal order is interpreted from top to bottom. Therefore, an event or operation call occurs later iff its arrow is below the arrow of another event or operation call.

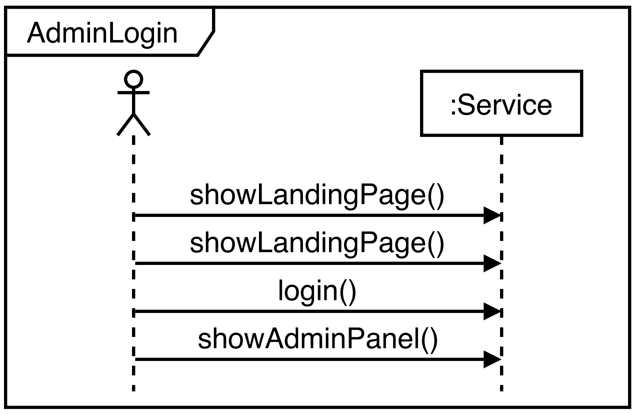


Figure 2.2: Example of a simplified sequence diagram

Sequence diagrams allow for more complex structures like loops and conditions involving different entities. However, in the course of this thesis, we only consider a simplified version of sequence diagrams as defined in [RH16]. This simplified version limits the number of involved objects to two, viz. the user and the software system. An example for a simplified sequence diagram is depicted in Figure 2.2, showing two lifelines. One lifeline belongs to the user, and the other lifeline belongs to the desired service. As it is illustrated, messages in the form of operation calls are sent from the user to the desired service.

2.2 On-The-Fly Computing

On-The-Fly Computing is an on-going research project at the University of Paderborn accomplished by the Collaborative Research Center 901 (CRC 901). Within the CRC 901, methods and technologies are investigated for dynamic software service markets, enabling participants of this market to request services, which are made available by so-called providers [Col11]. These dynamic software service markets are referred to as On-The-Fly Computing markets (OTF markets). Participants of the OTF market can be summarized in two major groups: end-users, which are referred to as customers, and providers. The various roles are outlined in Section 2.2.1.

The vision of the OTF market is to establish worldwide markets, where customers are provided individual services on demand, automatically composed of base services. Meaning, if a requested service as such is not available in the OTF market, providers compose the requested service out of other available services in order to fulfill the requirements of the request.

As an example consider the following scenario taken from [Pla+16]: A customer wants to ask for a service, computing a bus route from place A to place B. As an input the customer provides the GPS coordinates for the start location and the destination. The customer sends a request for a service to the OTF market, which calculates a bus route between two GPS coordinates. However, within the OTF market, there exists no service matching these particular requirements. Instead, there exists a service `getStation` that, given a GPS coordinate as an input, outputs the closest bus station and another service `getBusRoute`, calculating a bus route between two bus stations. Out of these two services, a service can be composed, obviously fulfilling the requirements of the customer. By applying the `getStation` twice, once for place A and another time for place B, we obtain the input data needed for the application of the `getBusRoute` service. In this way, although there exists no service in the OTF market that fulfills the requirements on its own, a service can be composed, using other services as building blocks.

In order to automate the process of service composition, the semantics of services have to be formalized to be comprehensible by machines. Hence, a formal description of services and requests is needed. In Section 2.2.2, this formal description of services and requests is outlined.

2.2.1 Roles

Within the OTF market, market participants may take over various roles. Generally, we can summarize the roles in two major groups: customers and providers.

The role of the provider is divided into two roles, which collaborate in order to provide the customer with individual services: on-the-fly providers (OTF providers) and service providers. While the OTF provider is responsible for answering requests of the user and configures services from base services, the base services are provided by the service providers. Therefore, the problem of composing services out of base services is distributed on different roles. While the OTF provider is responsible for configuring the blueprint for the requested service and requests the building blocks from the service provider, the service provider, in turn, is responsible for matching the requests for building blocks of OTF providers to its base services. In collaboration, the OTF provider and the service provider supply the user with the desired services on demand.

As service configuration is performed on demand, this has to be initiated by a request on behalf of a customer. The way this request is stated depends on the type of customer. In general, we distinguish between customers with a technical background, domain experts, and end users. While users with technical background bring the ability to state requests in the formalisms directly in the so-called Service Specification Language (SSL), domain experts and end users need a more user-friendly way of specifying services. The SSL is explained in the subsequent Section 2.2.2.

Other than specifying a service in terms of an SSL description, a specification in terms of a natural-language description provided by the end user represents one possibility of a user-friendly specification. This natural-language description is processed in order to extract the desired features and to formalize these requirements.

In contrast to that, we assume the domain expert to have the ability to model the behavior of the desired service in the form of exemplary models. Providing two sets of exemplary sequence diagrams, on the one hand, the user models desired behavior and prohibited behavior, on the other hand. Within the sequence diagrams, the domain expert may use arbitrary operation names. The formalized requirements specification is synthesized in two steps. First, a protocol for the desired service is generalized from the examples. The arbitrary operation names used in the sequence diagrams are mapped to actually existing operations and services in the OTF market, in a second step.

2.2.2 Requests and Services

In order to enable automated service composition and processing of requests by customers, it is inevitable to formally specify interfaces and semantics for services as well as for requests. For describing services and requests, the Service Specification Language is used, as defined in [Pla16; Pla+16]. The Service Specification Language (SSL), which extends the Palladio Component Model [BKR09], divides the functional description of a service respectively request in two parts: (1) a definition of operations by interface specifications and (2) a deterministic finite automaton describing the order in which the specified operations are used. Furthermore, non-functional requirements, such as for instance the price or the reputation, of a service can be defined (3). An example is shown in Figure 2.3

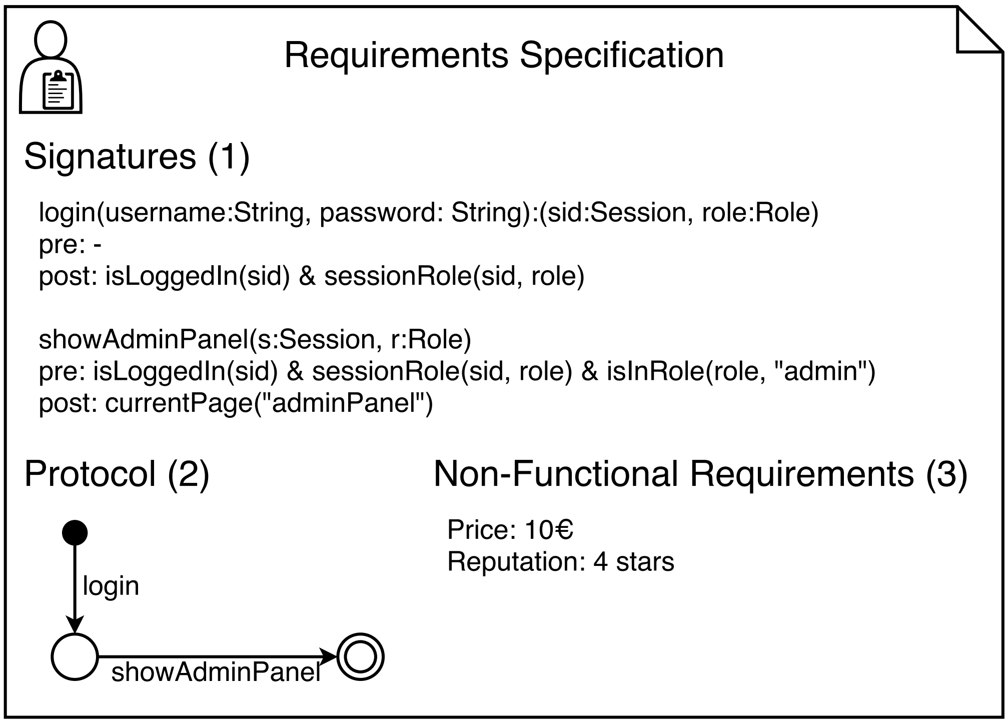

Requirements Specification
Signatures (1)
login(username:String, password: String):(sid:Session, role:Role)
pre: -
post: isLoggedIn(sid) & sessionRole(sid, role)
showAdminPanel(s:Session, r:Role)
pre: isLoggedIn(sid) & sessionRole(sid, role) & isInRole(role, "admin")
post: currentPage("adminPanel")
Protocol (2)
login
showAdminPanel
Non-Functional Requirements (3)
Price: 10€
Reputation: 4 stars

Figure 2.3: Example requirements specification described with SSL

The first part (1) of the functional specification contains a declaration of the operations that are used within the second part, i.e. the protocol. Traditionally, a signature of an operation is specified by the name of the operation, its input parameters, and the returned values respectively its output parameters. In addition, the semantics of the operation have to be specified by precondition and effect. While the precondition determines, which state is required to hold for enabling an operation call, the effect describes, as the name already suggests, the effect of an application of the operation. The description of an operation by **i**nput parameters, **o**utput parameters, **p**reconditions, and **e**ffects is referred to as IOPE description.

In the second part (2), a protocol in the form of a deterministic finite automaton describes in which order the declared operations are intended to be used. For that, the transitions of the automaton are labeled with the operation names.

Finally, the third part (3) is dedicated to the description of non-function properties. These include for instance the price for the desired service or a particular reputation in the market. Moreover, properties concerning technical non-functional requirement, such as throughput, the average response time, or execution time can be stated in this part of the requirements specification.

2.3 Grammatical Inference

Grammatical Inference refers to the process of learning formal languages from exemplary data. Usually, the inferred grammar is represented by production rules, finite state machines, or simply by deterministic finite automata. Grammatical Inference can be applied to a variety of problems such as natural language processing [DFG11], inductive logical programming [Fer+00], or sequential data mining [PS04].

In the case of learning a deterministic finite automaton (DFA), the learning algorithm is provided with example words of some language L. The aim of the machine learning process is to find an automaton which accepts the same language. Usually, the alphabet is known to the learner, while the remaining parameters have to be inferred from the training examples. Therefore, the task of the learner is to find an appropriate set of states, a set of accepting states and a transition function.

Deterministic as well as heuristic approaches exist for learning DFAs. The probably most popular deterministic grammatical inference algorithm is the so-called Evidence-Driven State Merging (EDSM) algorithm [LPP98]. It was shown in [LR03] that evolutionary approaches outperform EDSM for DFAs, having less than 32 states. There are even methods of training an artificial neural network (ANN) with the example data and extracting the DFA from that ANN after the training process [Gil+92]. In this thesis, we apply evolutionary algorithms to the problem of grammatical inference. Therefore, the main idea and basic concepts of evolutionary algorithms are presented in the subsequent Section 2.4.

2.4 Evolutionary Algorithms

In 1858, Charles Darwin published a theory towards evolution of species by natural selection that is based on four pillars: population, diversity, heredity and selection. A population is a group of several individuals with different characteristics and diversity refers to these differences among the individuals. The characteristics have their roots in the genes, which are also transmitted over generations by heredity. Selection refers to reducing the reproduction rate of certain individuals due to characteristics, which do not fit the environmental conditions. As a consequence, other individuals comparatively produce more offspring ("Survival of the fittest") [Dar].

Evolutionary algorithms (EAs) are learning or optimization algorithms that are inspired by the concepts of natural evolution [YG12]. Hereby, an individual represents a solution to a problem, and it can be specified in the form of some genetic representation. For instance, such a genetic representation might be a set of real values $(x_1, \dots, x_n)$ as parameters of some function $f : \mathbb{R}^n \to \mathbb{R}$, where the value of f has to be minimized.

Similar to natural evolution, the individuals in EAs are exposed to different selection mechanisms, and solution candidates are varied or recombined to obtain new solution candidates. In order to search for an optimal solution, the steps of evaluation, selection, and variation are repeated. These steps can be summarized in a basic evolutionary algorithm, which is presented in Section 2.4.1. Within this context, co-evolution refers to the parallel evolution of species that do not interbreed but may affect each other concerning for instance the selection. Section 2.4.2 is dedicated to explaining the concept of co-evolution, its abilities, and its limitations.

In real world applications, the fitness of an individual may be composed of different fitness functions, so that there are multiple objectives to be optimized. Due to maintaining multiple solutions at once, evolutionary algorithms prove a popular approach for solving multi-objective optimization problems. These so-called multi-objective evolutionary algorithms (MOEA) are presented in Section 2.4.3, and an exemplary instance of MOEA called NSGA-II is introduced in Section 2.4.4.

2.4.1 Basic Evolutionary Algorithm

Typically, evolutionary algorithms are oriented towards a certain pattern, which is illustrated in Figure 2.4. The overall process can be divided into four sub-routines: initialization of the population, fitness scoring, selection, and variation.

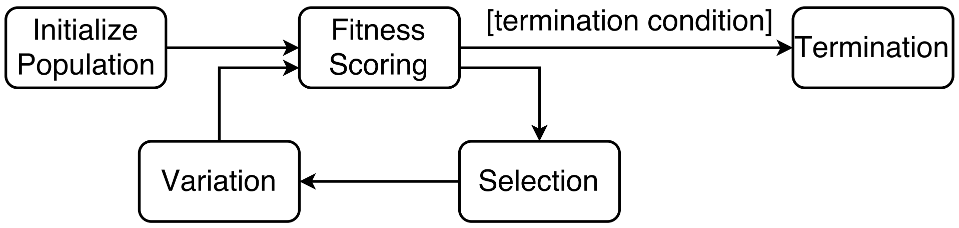


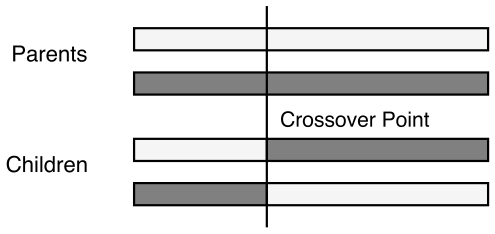
Figure 2.4: Basic evolutionary algorithm pattern

In the first step, a population is initialized with random individuals distributed according to some probability distribution. Typically, the distribution is a uniform distribution. After the population is initialized, the so-called *fitness value* is evaluated for each individual in the step *fitness scoring*. The fitness value is the outcome of some fitness function that has to be defined in advance. In general, the fitness function determines how well an individual performs as a candidate solution to the problem. In the example case of finding global optima of a function, this would simply be the function value for the parameters encoded by the individual.

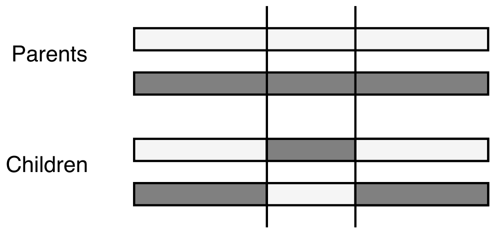
Once the fitness value is computed, individuals are selected for producing offspring. There are several different approaches on how to select individuals. As an example, a simple selection operator draws two individuals from the population and chooses the fitter individual. This kind of selection is known as *tournament selection* since two or more individuals compete against each other and the one with better fitness is considered the winner of the tournament.

In the variation step, selected individuals are used for reproduction, i.e. solution candidates are duplicated and modified in order to obtain new solution candidates. One possibility of altering individuals is referred to as mutation. A mutation operator makes with a certain probability a minimal change to the genome of an individual, e.g. a single bit of a genome in the form of a bit string is flipped.

Another way of modifying individuals is to recombine two individuals, e.g. by a so-called single-point crossover. As sketched in Figure 2.5a, a single-point crossover takes two individuals and chooses a random index to cut the genes. By switching the cut-off pieces, two new individuals are obtained which represent combinations of the parent individuals and may inherit the properties of the parents. Analogously, a two-point crossover is defined as cutting the gene in 3 parts and switching the center part (see Figure 2.5b). However, recombination techniques require the interpretation of genotypes to be decomposable, i.e. parts of the genotype represent sub-solutions, in order to work properly.



(a) Single-point crossover



(b) Two-point crossover

Figure 2.5: Crossover as recombination methods to produce offspring

By integrating the new individuals into the current population results in a new generation of the population. There are different strategies on how the offspring is integrated, e.g. replacing the whole population or keep the k best individuals of the previous generation and fill up the population with offspring.

The routines for fitness scoring, selection and variation are repeated until a termination condition holds. A termination condition may be, for instance, a certain threshold on the fitness value or a certain number of generations. While the number of generations is guaranteed to terminate eventually, a threshold on the fitness value might not be reached. Therefore, the choice concerning the termination condition depends on the application.

Altogether, evolutionary algorithms refer to a class of heuristic optimization approaches, which are inspired by natural evolution according to Charles Darwin. Taking into account the four pillars of evolution population, diversity, heredity and selection, evolutionary algorithms can be used to address complex problems. The concept of different species can be adopted, as well. Evolving multiple species in parallel is referred to as co-evolution and is presented in the subsequent section.

2.4.2 Co-evolution

The parallel evolution of different species that may affect each other but without interbreeding is known as co-evolution. In nature, there are many examples of co-evolution like predator and prey, host and parasite, or host and symbiont. Species living in a co-evolutionary context are exposed to a permanent competition, such that the development of the one species has a direct impact on the fitness of another species.

For instance, in the case of predator and prey, if the predator chases its prey and succeeds, this leads to a selection of prey individuals with better abilities in escaping from the predator. Hence, these individuals produce more offspring, so hunting becomes more difficult for the predator. As a consequence, faster or smarter predators are selected for producing more offspring. The process of affecting the other species to improve its fitness is called “arms race”.

In the context of evolutionary algorithms, co-evolution can be considered as two separate evolutionary processes running in parallel. However, the fitness scoring within these processes depends on the other evolutionary processes. Apart from that, evolution is performed as described in the previous sections.

2.4.3 Multi-Objective Evolutionary Algorithms

In the field of multiple objective decision making (cf. [HM79]), multi-objective optimization refers to the mathematical optimization problem involving multiple objective functions. The involved objective functions are subject to be optimized simultaneously. If it is not possible for two objective functions to obtain an optimal value at the same time, the two objective functions are called conflicting. In the presence of conflicting objectives, trade-offs have to be made.

Multi-objective evolutionary algorithms (MOEA) are a special instance of evolutionary algorithms, which are, usually, based on the principles of Pareto optimality. Instead of a single fitness value, individuals are assigned a fitness vector, where each entry of this vector corresponds to one fitness function. While fitness scoring simply iterates over the fitness functions for successively assigning the fitness values, selection becomes more complex.

In multi-objective optimization, usually, all the objective functions have the same weight. This means that, in the case of two fitness vectors $(0, 1)$ and $(1, 0)$, none of these can be found as better or worse. On the contrary, another fitness vector $(1, 1)$ is considered to be better than the other two. In particular, $(0, 1)$ and $(1, 0)$ are called *dominated solutions*, which are dominated by $(1, 1)$. Formally, dominated solutions can be defined as follows.

Let C be a set of solutions and let $f_1, \ldots, f_n$ be objective functions, where $f_i : C \to [0, 1]$ for all $1 \leq i \leq n$. A solution $c \in C$ is called *dominated* if there exists another solution $c' \in C$, such that $\forall i \in \{1, \ldots, n\} : f_i(c) \leq f_i(c')$. Otherwise, it is called *non-dominated*.

In the presence of multiple non-dominated solutions, these solutions form a non-dominated front as shown in Figure 2.6. A non-dominated front is defined as the set of all non-dominated solutions. If we take into account the entire search space

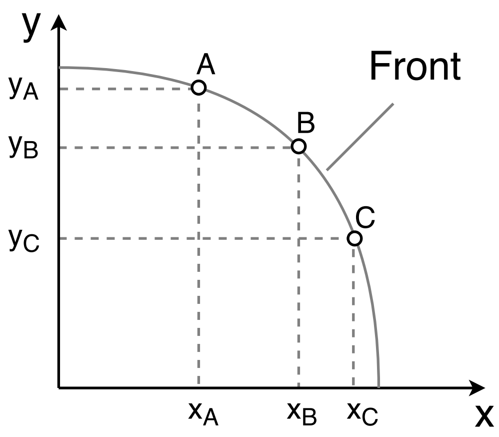


Figure 2.6: Candidate solutions arranged on a Pareto front

as the set C, the front of non-dominated solutions is called *Pareto front.* Furthermore, elements of the Pareto front are called *Pareto optimal.* Formally, Pareto optimal is defined as that there exists no other solution which is better in at least one objective without decreasing the value of another objective.

MOEAs based on the principle of Pareto optimality use the domination relation for the selection of individuals. Usually, a secondary selection operator needs to be taken into account since two individuals of the same front are incomparable. The following section deals with an exemplary instance of multi-objective evolutionary algorithms, which is called NSGA-II.

2.4.4 NSGA-II Algorithm

The NSGA-II initially presented in [Deb+02] is a special instance of an elitist multi-objective evolutionary algorithm based on the principle of Pareto optimality. NSGA-II is standardly instantiated involving a chained selection operator, i.e. if the first selection operator does not yield a conclusive result, a secondary selection operator is applied. Primarily, it is compared, whether one solution dominates the other one. If this is not the case, secondarily, the so-called crowding distance operator is applied.

The crowding distance takes into account, how different the considered individual from other individuals is. The crowding distance is defined as the space around the considered individual when drawing a hyperrectangle around this individual. The direct neighbors of the considered individual are used as reference points. A two-dimensional example for the crowding distance of two points A and B is shown in Figure 2.7.

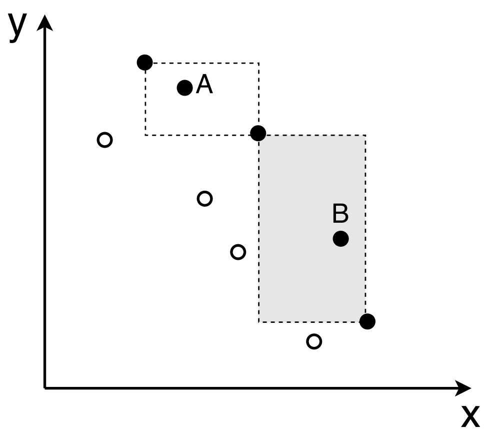


Figure 2.7: Example for crowding distance

In the figure, the black-colored points belong to the same non-dominated front. The closest neighbors are respectively taken as reference points for drawing a rectangular enclosing the point A respectively B. While this rectangular is colored white for A, the rectangular belonging to B is colored gray. Obviously, the size of the area of B is larger than the area of A, i.e. B has a greater crowding distance. When comparing two individuals, individuals with greater crowding distance are preferred since for individuals with smaller crowding distance there are still more similar solutions within the population. Moreover, the crowding distance operator proves beneficial for maintaining a more diverse population.

2.5 Active Learning

Active learning is a sub-field of machine learning, more specifically, semi-supervised learning, where the learning algorithm is given access to an oracle, which is able to label data points. The key hypothesis is that an active learning algorithm can achieve better performance with fewer training examples if it is allowed to choose the training data it learns from [Set12].

During the learning process, the learning algorithm maintains a set of considered data points, which can be divided into three sets. First, a set of data points for which the label is already known, second, a set of data points for which the label is unknown, and third, a set of data points that is chosen to be sent to the oracle in order to obtain labels for these data points.

The oracle is responsible for providing the labels of data points to the active learning algorithm. In an interactive scenario, the role of the oracle may be played by a human. Alternatively, the oracle might also be a database or another program.

Most likely, the biggest challenge in the field of active learning is to find an optimal strategy for choosing data points to be labeled by the oracle. One of these strategies is called *query-by-committee*. This strategy maintains a so-called committee of models, which are trained on the currently labeled data points but ideally represent different hypotheses. The most informative query is considered to be the data point for which the committee disagrees the most. The main idea behind this strategy is to minimize the version space, i.e. to minimize the number of hypotheses which are consistent with the set of labeled data points.

3 Related Work

Applying evolutionary algorithms to the problem of grammatical inference became really interesting through the introduction of the so-called Smart State Labeling algorithm by Lucas and Reynolds [LR03; LR05]. This approach, which is discussed in Section 3.1, allows evolving deterministic finite automata using only the transition matrix as a genome. Based on this algorithm, Bongard and Lipson [BL05] introduced an interactive co-evolutionary approach, reducing the amount of required training data by applying active learning techniques. Section 3.2 gives an overview of this approach.

Other than that, in Section 3.3, we outline an approach synthesizing statecharts from sequence diagrams. In Section 3.4, we discuss an approach by van Rooijen and Hamann [RH16], reformulating the problem of the synthesis of deterministic finite automata from sequence diagrams as a grammatical inference problem.

3.1 Evolving Automata Using Smart State Labeling

Lucas and Reynolds [LR03; LR05] introduced a new approach in the field of grammatical inference referred to as Smart State Labeling (SSL) algorithm. Since for the alphabet and the number of states fixed values are assumed and q_0 can be fixed w.l.o.g., it remains to evolve δ and F. Hence, the search space complexity is $|Q|^{|\Sigma| \cdot |Q|}$ for searching $\delta : Q \times \Sigma \to Q$ and $2^{|Q|}$ for F, as for every state a boolean value is needed, determining whether a state belongs to F. This yields an overall search space complexity of $|Q|^{|\Sigma| \cdot |Q|} 2^{|Q|}$.

In order to reduce the search space complexity, Lucas and Reynolds [LR05] propose the SSL algorithm for calculating F from the training examples, such that the evolved automata have an optimal labeling with respect to the training examples. To this end, the algorithm counts for each state how many training examples of each label end in the respective state. The state is added to F if at least half of the training examples which end in this state are labeled accepting. In this way, the search space complexity for evolving automata is reduced by the factor $2^{|Q|}$. Therefore, the resulting search space complexity is $|Q|^{|\Sigma| \cdot |Q|}$.

Experiments showed that a simple evolutionary algorithm, using the SSL algorithm, outperforms the deterministic, former state-of-the-art algorithm Evidence-Driven State Merging for target automata with less than 32 states.

3.2 Active Co-Evolutionary Learning

Traditional approaches, applying evolutionary algorithms to the problem of grammatical inference, need a large number of training examples in order to infer the target language. As outlined in Section 2.5, in active learning, the algorithm is given access to an oracle, labeling new data points. In this way, the algorithm is allowed to strategically augment the training data, requesting data points needed in order to increase the quality of the solutions.

Bongard and Lipson [BL05] combine active learning, the SSL algorithm (cf. Section 3.1), and two evolutionary algorithms to an active co-evolutionary learning approach for deterministic finite automata referred to as Estimation-Exploration Algorithm (EEA). The EEA alternates between evolving deterministic finite automata (DFAs) with respect to the training examples and evolving words which produce disagreement among a set of DFAs. Disagreement of a word is measured by counting how many DFAs accept respectively reject this word. The disagreement is considered to be maximal if one half of the DFAs accepts and the other half rejects the word. The word with the maximum disagreement is chosen as a query to the labeling oracle.

In [BL05], experiments showed that EEA outperforms its passive equivalent, i.e. queries to the oracle are chosen uniformly at random, and other passive approaches for less than 32 states. Furthermore, the results show that EEA needs less training examples than passive approaches to reach a certain test set accuracy.

3.3 Synthesis of Statecharts from Sequence Diagrams

Statecharts are useful artifacts for the specification and formalization of requirements. Especially in model-drive software development and systems engineering, statecharts represent an indispensable model class [SLV14; BGS05] enabling code generation.

However, the task of specifying and refining statecharts is a laborious task. Therefore, Harel, Kugler, and Pnueli [HKP05] present an approach for the synthesis of statecharts from examples of so-called Live Sequence Diagrams (LSDs). LSDs refer to an extension of standard message sequence diagrams (MSDs) or UML sequence diagrams, allowing for new expressions such as quantifiers.

The synthesis approach, first, encodes the LSDs as a transition system. In a second step model checking techniques are applied to this transition system in order to verify the soundness of the specified behavior. Lastly, for each of the objects evolved in the LSDs, a statechart is generated. However, the overall approach is time-consuming even though it is correct with respect to the provided LSDs. The requirements specified in the form of LSDs have to be precise and complete for synthesizing the statechart. Therefore, the task of thinking through the whole software system is already performed when creating the LSDs.

3.4 Requirements Specification-By-Example Using Multi-Objective Optimization

Van Rooijen and Hamann present an approach of inferring deterministic finite automata from a simple version of sequence diagrams in [RH16]. The number of involved objects in the simple version of sequence diagrams is limited to two, viz. an actor and the system. Therefore, sequence diagrams can be represented by sequences of operation calls. Taking an evolutionary multi-objective optimization (based on the principles of Pareto optimality) approach for grammatical inference, the authors interpret the sequences of operations as words over an alphabet, consisting of operation names as input symbols. These words are either labeled by the user as desired or prohibited behavior.

Based on the provided examples, a protocol represented by a deterministic finite automaton is inferred, optimizing the models with respect to the examples subject to different objectives. The authors propose to maximize the accuracy on positive examples, the accuracy on negative examples, and the number of sink states as well as minimize the proportion of states which is needed for processing all the training examples. A sink state refers to a state of which all outgoing transitions are self-loops. The result of this approach yields a Pareto front of deterministic finite automata, which are returned to the user.

4 Approach

The automated service composition algorithms within an OTF market involving users, service providers, and OTF providers, are based on formal descriptions for both services and requests. These formal descriptions, e.g. in the form of predicate logic, require expert knowledge of the demanded formalisms as well as an understanding of the services themselves. In order to also enable non-expert users, such as domain experts and even naive users, to state requests for the OTF market a more intuitive way of creating requests is mandatory.

In [RH16], van Rooijen and Hamann proposed a requirements specification-by-example approach (cf. Section 3.4). Requiring the user to provide sets of simplified sequence diagram representing examples of desired and prohibited interactions between the user and the desired service, a formal requirements specification in the form of a deterministic finite automaton is generalized from these examples. Using this concept and this way of interpreting the problem as a basis, in Section 4.1 we extend this approach by asking the user (since the user is the only one who is able to reveal more information about the desired service) for feedback following Bongard and Lipson [BL05].

However, the user can only provide a strongly limited number of such exemplary sequence diagrams in order to describe the interaction with the desired service. Usually, thousands of examples are necessary in order to obtain solutions of good quality. Furthermore, the simplified version of sequence diagrams is restricting the complexity of the inferred models. With a view to remain practicable for naive users as well, it is more favorable to request the required data from another source of knowledge. Alternatively, other than the user, the OTF market contains detailed information about the semantics of the services which are about to be used as building blocks for the desired service, which is described by the user. Therefore, we consider two concepts in Section 4.2 on how to use some information, which is available within the OTF market or which can be learned by observing the communication between participants of the OTF market.

4.1 Interactive Coevolutionary Requirements Specification-by-Example

Initially, a user, who wants to query the OTF market for a desired service, may provide an exemplary description of that service in the form of simplified UML sequence diagrams. The simplified version of a sequence diagram corresponds to standard UML sequence diagrams (see [Obj15]) but restricts the number of involved objects to two objects: the user and the desired service [RH16].

The user provides two sets of simplified sequence diagrams. One set describes in terms of examples desired behavior and the other one prohibited behavior. This way, the user is able to describe the behavior, which is expected to be implemented by the desired service on the one hand, and on the other hand, other behavior can be excluded explicitly. Within the sequence diagrams, the user may use arbitrary operation names in order to describe the behavior of the service. Operation names used in the sequence diagrams are mapped to existing operation names in a later step. This way, the user does not need any knowledge about operations and services that are available within the OTF market. However, the mapping of those arbitrary operation names to actually existing services is out of the scope of this thesis.

The provided sequence diagrams can be understood as exemplary snippets of a protocol describing the interaction between the user and the desired service. Hence, the behavior of the service is also implicitly described. One possible way of representing such a protocol is a deterministic finite automaton, which is defined in the next Section 4.1.1.

In Section 4.1.1, we first define the genetic representation of deterministic finite automata. The following Section 4.1.2 is dedicated to presenting the requirements specification-by-example approach by van Rooijen and Hamann [RH16] in all details. As a contribution of this thesis, this approach is extended in Section 4.1.5 to an interactive co-evolutionary approach by following the approach by Bongard and Lipson [BL05], which is recalled in Section 4.1.4 and adapted to deal with non-binary alphabets as well. In these approaches, words for automata are evolved in a parallel evolutionary process, which requires a genetic representation for words. Therefore, Section 4.1.3 is dedicated to the definition of the genetic representation of words. Finally, we give an overview of the presented approaches in Section 4.1.6.

4.1.1 Genetic Representation of DFAs

A deterministic finite automaton (DFA) over an alphabet Σ can be defined by a 5-tuple $A = (\Sigma, Q, \delta, q_0, F)$, where Q is a finite set of states, $q_0 \in Q$ the initial state, $F \subseteq Q$ a set of accepting states and $\delta : Q \times \Sigma \to Q$ a transition function. A word $w = a_1 a_2 \dots a_n$ of length n is a sequence of n input symbols $a_i \in \Sigma$. If for such a word w there exists a sequence of states $q_0 q_1 \dots q_n$ such that $\delta(q_{i-1}, a_i) = q_i$ and $q_n \in F$, then w is *accepted* by A. Otherwise, the word is *rejected*. The set of all words that are *accepted* by A form the *language* of A denoted as $L(A)$, i.e. $L(A) = \{w \in \Sigma^* | w \text{ is accepted by } A\}$.

If the transition function δ is a total function, then A is called a *complete* deterministic finite automaton. In the following, we consider only complete DFAs, since they can be simply represented by a $|Q| \times |\Sigma|$ matrix over Q as explained below. This does not constrain the presented approaches as every DFA can be extended to a complete DFA still accepting the same language. The *size* of a DFA is defined as its number of states *reachable* from the initial state q_0. A state q is *reachable* if there exists a sequence of input symbols $a_1 \dots a_k$ such that $\delta(\delta(\dots \delta(q_0, a_1) \dots, a_{k-1}), a_k) = q$ for some $k \in \mathbb{N}$.

Since simplified sequence diagrams, as provided by the user, involve only interactions between the user and the desired service, the sequence diagrams can also be represented by a sequence of operation names. Considering the operation names as symbols a_i of an alphabet Σ, a sequence of operation names forms a word $w = a_1 a_2 \dots a_n$, for $a_i \in \Sigma$. We extract the alphabet Σ from all the distinct operation names occurring in the sequences of operation names. Additionally, the user classifies the sequences of operations as desired or prohibited behavior. This classification is translated to *accepted* respectively *rejected* labels. Hence, the user describes a language L by examples, accepted by some DFA. A tuple $x = (w, l)$ where w is a word and l its correct label is called a *training example*. Altogether the training examples form a set referred to as training data S.

In order to define the genetic representation of a DFA $A = (\Sigma, Q, \delta, q_0, F)$, we split this representation in two parts (Σ, Q, δ, q_0) and (F). Without loss of generality, we assume Q to be of the form $\{0, \dots, |Q| - 1\}$, Σ of the form $\{0, \dots, |\Sigma| - 1\}$ and q_0 is always the state 0. By this means we can represent the first part (Σ, Q, δ, q_0) by a $|Q| \times |\Sigma|$ matrix T over Q. As shown in Equation (4.1), the matrix coordinates of T range from 0 to $|Q| - 1$ and from 0 to $|\Sigma| - 1$, such that all the states and input symbols are implicitly represented by row and column numbers. Each entry $T_{i,j}$ is assigned the state returned by $\delta(i, j)$ and therefore contains the state reached when A is in the state i reading the input symbol j.

$$T = \begin{array}{c} \\ 0 \\ 1 \\ \vdots \\ |Q|-1 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & \dots & |\Sigma|-1 \end{array} \\ \begin{pmatrix} \delta(0,0) & \delta(0,1) & \dots & \delta(0,|\Sigma|-1) \\ \delta(1,0) & \delta(1,1) & & \vdots \\ \vdots & & \ddots & \vdots \\ \delta(|Q|-1,0) & \dots & \dots & \delta(|Q|-1,|\Sigma|-1) \end{pmatrix} \end{array} \tag{4.1}$$

The second part, F, is calculated from the first part and the provided training examples, as initially proposed by Lucas and Reynolds [LR05]. Instead of representing F intuitively by an array of size $|Q|$ setting the i-th value to 1 iff q_i is an accepting state, we only consider DFAs which have an optimal labeling with respect to the set of training examples. With the so-called Smart State Labeling (SSL) algorithm F is calculated as follows. In order to decide whether a state q belongs to F, the set of all training examples $x_i = (w_i, l_i)$ ending in q (i.e., when A reads w_i, in the end, A is in state q) are considered. If the number of input examples labeled with *accept* is greater or equal to the number of input examples with a *reject* label, q is considered to be an *accepting* state, and thus, it is added to F. Otherwise, q is considered to be a *rejecting* state.

In [LR05; LR03] it was shown, that approaches using the SSL algorithm outperform other approaches evolving both a matrix T and the set of accepting states F. Using the SSL algorithm has two major advantages. First, as already mentioned above, we want to consider only those DFAs having an optimal labeling with respect to the training examples, as this maximizes the number of correct classification of training examples. Second, the implicit representation reduces the search space complexity by a factor of $2^{|Q|}$.

As a result, a DFA can be completely represented by the matrix T as defined above. Concatenating the rows of T to a sequence of integers, standard recombination techniques such as single-point crossover and mutation can be applied straightforward. Therefore, T is used as a genome in the evolutionary approaches presented below. Furthermore, individuals representing a DFA are subsequently referred to as *candidate models*.

4.1.2 Grammatical Inference by Multi-Objective Optimization

Conventionally, applied to the problem of grammatical inference, heuristic approaches use only the accuracy of a candidate model regarding the training data S

as an objective [LR05; LR03; Gom06]. The accuracy of an automaton A regarding a set of training examples can be expressed as

$$f_{\text{all}}(A) = \frac{|\{(w,l) \in S \mid A(w) = l\}|}{|S|}, \tag{4.2}$$

where $A(w)$ denotes the label output by A when reading w. However, these approaches require thousands of training examples. In our setting, where the user is required to provide this amount of training examples, this is impracticable. Furthermore, it has to be expected that users are expected to make mistakes labeling the training examples or even provide contradictory training data. Meaning some words might appear twice, once labeled with *accept* and another time with *reject*.

Addressing these problems, Van Rooijen and Hamann [RH16] propose a heuristic approach using multi-objective optimization in order to present a diverse set of solutions to the user. Instead of measuring the fitness of all training examples within one fitness function, as in Equation 4.2, the overall accuracy is split up into fitness functions measuring the accuracy for each label (*accept* and *reject*). Hence, the authors consider one fitness function for training examples labeled with *accept* and another one for training examples labeled with *reject* as defined in Equations (4.3) and (4.4).

$$f_{\text{pos}} = \frac{|\{(w,l) \in S \mid A(w) = l \land l = \text{accept}\}|}{|\{(w,l) \in S \mid l = \text{accept}\}|} \tag{4.3}$$

$$f_{\text{neg}} = \frac{|\{(w,l) \in S \mid A(w) = l \land l = \text{reject}\}|}{|\{(w,l) \in S \mid l = \text{reject}\}|} \tag{4.4}$$

In the case of one training example provided twice by the user, once as desired behavior and another time as prohibited behavior, the solution could possibly contain one candidate model accepting the example and one candidate model rejecting it. Hence, the decision regarding the correct classification of the training example might be delayed to the user.

Additionally, structural aspects of the candidate models are considered in [RH16]. In order to evaluate the generalization performance, the proportion of states needed to process all the training examples is minimized. This objective is referred to as *relevant part* and is defined as follows.

$$f_{\text{rel}}(A) = \frac{\left|\bigcup_{(w,l)\in S}\{q_i \mid q_i \text{ is visited by } A \text{ reading } w\}\right|}{|Q|} \tag{4.5}$$

The authors argue, that the fewer states are needed for processing all the training examples, the better the generalization performance of a candidate model is. Moreover, this objective does not constrain the size of the DFA such that candidate models do not become too specialized.

Another structural objective is introduced, which is counting and maximizing the number of sink states, i.e. states of which all outgoing transitions are self-loops. This is intended to guide the algorithm to better discriminate between desired and prohibited behavior. Formally, this objective can be defined as in Equation 4.6.

$$f_{\text{sink}}(A) = \frac{|\{q \in Q \mid \forall a \in \Sigma : \delta(q, a) = q\}|}{|Q|} \tag{4.6}$$

The approach was implemented as an instance of the multi-objective optimization algorithm NSGA-II, which was originally introduced in [Deb+02]. In order to deal with the multiple objectives, NSGA-II is based on the concept of Pareto optimality and sorts individuals by calculating Pareto fronts recursively on the respectively remaining individuals. If there are two individuals belonging to the same Pareto front, these individuals are compared using the so-called *crowding distance*, as proposed in [Deb+02]. Candidate models with a larger crowding distance, i.e. of which less individuals with fitness values close to the considered candidate model exist in the population, are preferred. Selection is performed by a tournament of two individuals comparing first the membership of the Pareto front and secondarily the crowding distance.

As already mentioned above, due to the representation of T as a sequence of integers, standard recombination techniques can be applied straightforwardly. As genetic operators mutation and a single-point crossover are applied. Within the implementation, for both operators, the sequences of integers are converted to a binary string in the first place. With probability

$$p_M = \frac{1}{|Q| \times |\Sigma|} \tag{4.7}$$

one bit is flipped in the binary string representation of each entry in T. This ensures one change in the matrix T on average. Additionally, a single-point crossover is applied to the binary string representation of T with probability 1.

In our setting, with the aim of inferring a protocol from provided sequence diagrams, the usage of multi-objective optimization has several advantages over the traditional approaches. First, because of the sparse data, there are many models with high fitness with respect to the accuracy on the set of training examples S. Appropriately chosen objectives allow for a more sophisticated and a more robust heuristic. Instead of only a single solution, the user is shown a set of solutions such that the user can set preferences to objectives which might be more important than other ones. Finally, multiple objectives can have a positive impact on the search itself preserving a more diverse population and enabling local search in different locations of the fitness landscape.

Nevertheless, the quality of the inferred models highly depends on the usefulness of the provided training examples. Without automation of requirements elicitation, the user would collaborate with a requirements engineer in order to define requirements and refine these in an iterative process. In order to support the user, the algorithm needs to interact with the user, making suggestions, recommendations and asking for feedback. The multi-objective approach by van Rooijen and Hamann [RH16] still needs too much training data that is already required at the very beginning of the requirements elicitation process. On the one hand providing the amount of data needed for evolving well-performing candidate models would possibly demotivate and fatigue the user, on the other hand, the user is required to run the iterative process on his own in advance. Therefore, the amount of required training data has to be further diminished and the algorithm should interact with the user. The interaction with the user should support the user in providing more useful training examples and refine the initially provided training examples iteratively. In this way, the algorithm takes over the role of an "automatic requirements engineer".

In [BL05], Bongard and Lipson combine active learning techniques with a heuristic approach for grammatical inference. The algorithm is given access to a labeling oracle, which knows the target model and answers the algorithm's queries whether a word is accepted or rejected by the target model. For that, alternating candidate models and words are evolved. The evolved words are used for querying the oracle to label the respective words according to the target model. This way, the algorithm has the opportunity to strategically augment the training data with the help of a heuristic.

Experiments showed that this active co-evolutionary learning approach for grammatical inference outperforms passive approaches, i.e. approaches which obtain randomly drawn training data. In particular, considerably less training data is needed for achieving the same quality of candidate models compared to passive approaches. We transfer this approach to our context, where the user takes over the role of the oracle since only he is able to reveal more information about the desired service. In addition, we adapt the approach in order to deal with non-binary alphabets. Using this approach, the user is supported by providing training examples, which the algorithm considers to be useful. For enabling the evolution of words as queries to the user/oracle, we need a genetic representation of words. The genetic representation is presented in the subsequent Section 4.1.3.

4.1.3 Genetic Representation of Words

Evolving words requires a definition for the genetic representation of words. As a recall, a *word* $w = a_1 a_2 \dots a_n$ is a sequence of symbols a_i from an alphabet Σ. We assume for Σ to be of the form $\Sigma = \{0, 1, \dots, |\Sigma| - 1\}$. Thus, a word is represented by a sequence of integers ranging from 0 to $|\Sigma| - 1$.

Due to technical reasons (concerning the implementation), we restrict the maximum length of a word by $\ell_{\max}$. We define the genome for evolving words to be a tuple $t = (\ell, w)$, where $0 \leq \ell \leq \ell_{\max}$ denotes the length of the word and $w \in \Sigma^{\ell_{\max}}$ is a word of length $\ell_{\max}$. The variable ℓ determines to which extent the symbols in w are considered at all. More precisely, this means only the first ℓ symbols of w are considered. The remaining symbols are ignored. In the following, we refer to an individual representing a word as *candidate test*. Note that restricting the maximum length of words also constrains the search space complexity for the evolution of candidate tests to $|\Sigma|^{\ell_{\max}}$.

4.1.4 Estimation-Exploration Algorithm

The mentioned approach by Bongard and Lipson [BL05] called Estimation-Exploration Algorithm (EEA) is an active co-evolutionary learning algorithm for grammatical inference. As depicted in Figure 4.1, the algorithm operates in two phases: (A) an estimation phase for evolving candidate models, and (B) an exploration phase, in which candidate tests are evolved. In the following, we first present the original approach before we transfer it to the OTF context.

Initially, the algorithm obtains a set of training examples, i.e. a set of labeled words over a binary alphabet, and randomly initializes a population of p candidate models. In phase (A), the estimation phase, two distinct populations of candidate models are evolved with no interbreeding between these populations for g generations. Separating the two subpopulations maintains more diversity among the entire population. The overall accuracy on the training examples

$$f_{\text{all}}(A) = \frac{|\{(w, l) \in S \mid A(w) = l\}|}{|S|} \tag{4.8}$$

is used as a primary fitness measure. If candidate models have the same fitness for f_{all}, then the candidate model with the smaller relevant part

$$f_{\text{rel}}(A) = \frac{\left|\bigcup_{(w,l)\in S}\{q_i \mid q_i \text{ is visited by } A \text{ reading } w\}\right|}{|Q|} \tag{4.9}$$

is preferred over the larger one.

After that, in phase (B) candidate tests are evolved for g generations using a set of candidate models measuring the *disagreement* concerning how to label a respective candidate test. The disagreement is expressed as

$$g_{\text{dis}}(t, C) = 1 - 2\left|0.5 - \frac{\sum_{j=1}^{|C|} \texttt{label}(C(j), t)}{|C|}\right|, \tag{4.10}$$

where t is a candidate test, C a set of candidate models, and $\texttt{label}$: Model$\times$Test $\to \{0, 1\}$ is a function returning 1 if the candidate model accepts the candidate test and 0 otherwise [BL05].

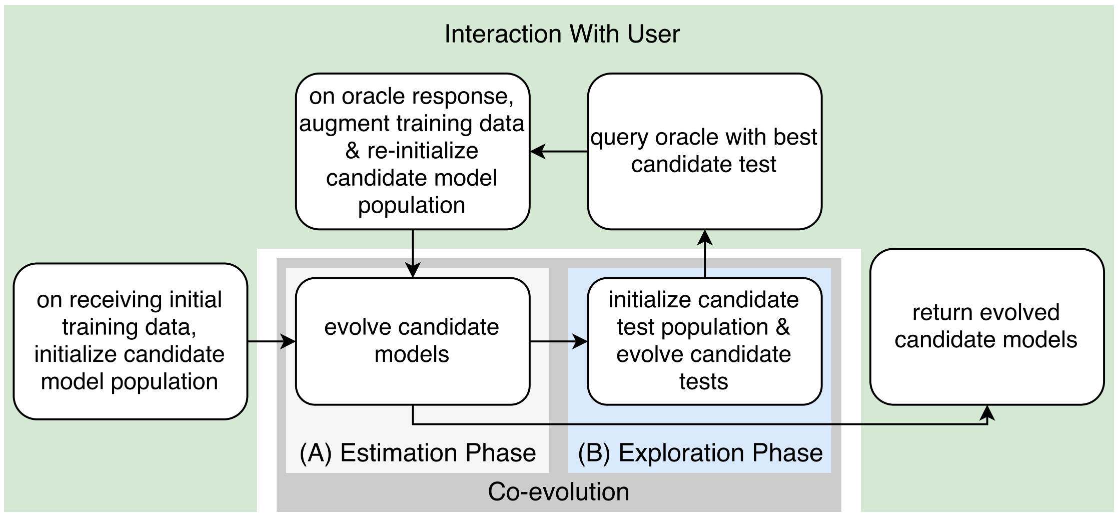

Interaction With User
on oracle response, augment training data & re-initialize candidate model population
query oracle with best candidate test
on receiving initial training data, initialize candidate model population
evolve candidate models
initialize candidate test population & evolve candidate tests
return evolved candidate models
(A) Estimation Phase
(B) Exploration Phase
Co-evolution

Figure 4.1: Estimation-Exploration Algorithm schematic process. (A) Estimation Phase: evolution of candidate models (B) Exploration Phase: evolving candidate tests as a query to the oracle.

The concept of giving the learning algorithm access to an oracle in order to get labels for additional data points originates from the field of active learning. Using a so-called *committee* of candidate models for measuring the disagreement on a query is known as the *query-by-committee* strategy.

In the EEA, the committee consists of the best individuals of each subpopulation from the preceded estimation phase. Since the population is split into halves, the size of this committee is 2. Hence, the disagreement for a candidate test is either 0 or 1. After evolving the candidate tests for g generations, the best individual is taken as a query to the oracle. When the oracle returns the true label of the requested candidate test, the new training example is added to S, and the population of candidate models is re-initialized. Re-initialization seeds the subpopulation with the corresponding best individual from the last generation and fills up the remaining population with randomly generated candidate models.

In both phases, each generation selects $0.75 \cdot p$ many pairs of individuals and compares their fitness. The worse individual of a pair is overwritten with a copy of the better one, and the copy is then mutated. The mutation for candidate models chooses uniformly at random one entry $T_{i,j}$ of T and changes its value to another state again uniformly at random. Since the transition matrix T has size $2 \cdot |Q|$ the mutation rate is denoted as

$$p_M = \frac{1}{2 \cdot |Q|} \tag{4.11}$$

For candidate tests, mutation is a little different. Since a candidate test is a tuple $t = (\ell, w)$ with probability 0.5 either ℓ or w is mutated. If ℓ is chosen to be mutated, a new value ℓ is drawn uniformly at random. Otherwise, one symbol in w is mutated uniformly at random.

Bongard and Lipson [BL05] introduce the EEA operating on binary alphabets only. In order to transfer the approach to the OTF context, we extend the alphabet to a non-binary since supporting only two different operation names would be infeasible. We assume the alphabet to be of the form $\{0, \dots, |\Sigma| - 1\}$. Keeping the definition of mutation by Bongard and Lipson [BL05] changing one entry of the transition matrix at a time, the mutation rate can be expressed by

$$\frac{1}{|Q| \cdot |\Sigma|}. \tag{4.12}$$

Furthermore, we substitute the provider of the initial training examples and the labeling oracle by the user. Analog to [RH16], we expect the user to provide simplified sequence diagrams which we interpret as words over an alphabet of operation names. For presenting the queries to the user, the candidate tests have to be retranslated to sequence diagrams.

In the case of grammatical inference, Bongard and Lipson [BL05] showed in experiments that co-evolution proves beneficial in order to improve the quality of candidate models by augmenting the training data with the right training examples. Over time candidate models become more sophisticated, such that candidate tests with high disagreement become harder to find. In Section 4.1.5, we combine the multi-objective optimization approach from Section 4.1.2 with this active co-evolutionary learning approach.

4.1.5 Interactive Co-Evolutionary Multi-Objective Optimization

Analog to [BL05], we extend the multi-objective optimization approach presented in Section 4.1.2 to an active co-evolutionary learning algorithm using multi-objective optimization for both evolving candidate models and candidate tests. Since we can expect the user to provide only very few training examples, we need a more sophisticated heuristic in order to distinguish more easily between superior and inferior candidate models.

Just like in the previous section, we split the population for candidate models into halves. In each subpopulation, we apply NSGA-II using the selection methods and genetic operators as specified in Section 4.1.2. Since NSGA-II is based on the concept of Pareto optimality, it returns a Pareto front as a result. After new training examples are obtained by the oracle/user, the subpopulations of candidate models are initialized with the corresponding Pareto front.

As another fitness measure, we define a function counting the number of reachable states from the initial state. In order to stress the generalization behavior, we want to minimize the number of reachable states and define the fitness function as

$$f_{\text{reach}}(A) = 1 - \frac{|\{q \in Q \setminus \{q_0\} \mid q \text{ is reachable from } q_0\}|}{|Q| - 1}, \tag{4.13}$$

where A is DFA and Q its set of states. Note, that this fitness function has no direct correlation with the training data obtained from the user. Hence, this fitness function might perform worse than f_{rel} (cf. Equation (4.5)) as a heuristic. Nevertheless, in f_{rel} we assume a better generalization performance if fewer states are used in order to process the training examples. Some states of a candidate model might never be visited for processing the training examples. This means that we have no knowledge about the behavior. By using f_{reach}, we cut off these obsolete states and focus more on the part of the automaton learned from the training data.

For the evolution of candidate tests, we again apply NSGA-II. We use the disagreement as specified in Equation (4.10) as one objective for assessing the fitness of a candidate. The Pareto fronts of all subpopulations form the committee for calculating the disagreement values. Since the population is split into halves, it is ensured that the committee consists of at least two candidate models. However, the more different fitness functions are used, the more candidate models possibly belong to the committee. Therefore, in contrast to the approach by Bongard and Lipson [BL05] presented in Section 4.1.4, disagreement values different from 0 or 1 are possible. On top of this, we introduce another objective, minimizing the length of such a candidate test. The idea behind this objective is to postpone consequential errors that occur due to mistaken transitions close to the beginning. The fitness function for minimizing the length of a candidate test $t = (\ell, w)$ is given in Equation (4.14).

$$g_{\text{len}}(t) = 1 - \frac{\ell}{\ell_{\max}} \tag{4.14}$$

Except for the probability of applying mutation to a gene, the genetic operators for candidate tests function in an equal manner as explained in Section 4.1.2. The genome of a candidate test $t = (\ell, w)$ consists of $\ell_{\max} + 1$ integers ($\ell_{\max}$ symbols from Σ and the number of considered symbols ℓ). A mutation rate of

$$p_M = \frac{1}{\ell_{\max} + 1} \tag{4.15}$$

is used to ensure one change in the genome of a selected individual on average. Due to implementation details for the genetic encoding of candidate tests in NSGA-II, we use a lower permutation rate for the length variable than in the EEA, but the permutation rate for w is approximately the same.

Altogether, this extends the approach presented in Section 4.1.2 following the concepts by Bongard and Lipson [BL05] to an interactive co-evolutionary algorithm which is strictly based on multi-objective optimization. We fully leverage the advantages of multi-objective optimization and present the whole Pareto front of evolved candidate models to the user. Due to the very small amount of data, the usage of multiple objectives provides a benefit for differentiating the quality of candidate models in a more sophisticated way. In the following, the strictly multi-objective optimization approach presented here is referred to as Estimation-Exploration Multi-Objective Optimization (eeMOO) algorithm, the initial multi-objective optimization approach presented in Section 4.1.2 is referred to as passive multi-objective optimization (pMOO) algorithm.

4.1.6 Overview of Approaches

Table 4.1 summarizes the fitness functions used in the different approaches presented in the previous sections. The multi-objective optimization approach by van Rooijen and Hamann [RH16] (cf. Section 4.1.2) is denoted as pMOO. The Estimation-Exploration Algorithm, originally introduced by Bongard and Lipson [BL05] (cf. 4.1.4), in its slightly adapted version in order to deal with non-binary alphabets, is referred to as EEA. Finally, the approach introduced in Section 4.1.5 is instantiated with two different sets of fitness functions. While the first version is instantiated with the same objectives as pMOO and is identified by eeMOO, the second version uses only the overall accuracy on the training data (cf. f_{all} in Equation (4.2)) and the reachable states fitness function (cf. f_{reach} in Equation (4.13)) for the evolution of candidate models.

Table 4.1: Overview of the different approaches and their fitness functions.

	pMOO	EEA	eeMOO	eeMOO2
Model Fitness	f_{pos}-f_{neg}-f_{rel}-f_{sink}	$f_{\text{all}}\|\|f_{\text{rel}}$	f_{pos}-f_{neg}-f_{rel}-f_{sink}	f_{all}-f_{reach}
Test Fitness	/	g_{dis}	g_{dis}-g_{len}	g_{dis}-g_{len}

Legend: A dash denotes that the fitness functions are evaluated independently from each other and, thus, have the same precedence. The double-pipe operator stands for a chained application of the fitness functions, i.e. if the first values are equal, then the next fitness function is taken into account. If no fitness function is used at all, this cell is marked by an oblique stroke (/).

As the table shows, pMOO is the only passive algorithm, meaning that the algorithm only processes an initially provided set of training examples. Hence, there is also no evolution of candidate tests and thus no objective for candidate tests' fitness. While EEA principally is a single objective approach using the relevant part fitness function only as a tie-break rule, eeMOO and eeMOO2 represent strict multi-objective approaches.

4.2 Integration Into OTF Markets

In our scenario, semi-automated requirements elicitation is done within the OTF market, when specifying the requirements for the desired service. This process of requirements elicitation involves at least two roles, viz. the user describing the desired service and some OTF provider generalizing a formal model from the provided examples and supporting the user to provide relevant data.

Using one of the interactive approaches presented above implies communication between the user and the OTF provider. In particular, for each oracle query to the user and the respective response, a distinct message is sent. This constitutes not only high traffic overhead querying only for single words, but it also increases the overall time of the process. Addressing this problem by bundling multiple queries into a single message might lead to less effective training data augmentation. Since a committee of evolved candidate models partly assesses the fitness values of candidate tests, these estimations are highly dependent on the quality of the candidate models. Vice versa, the quality of the candidate models depends on the training data which is extended by selected candidate tests. Therefore, it is intuitively expected for five candidate tests to ask for feedback query-by-query, while intermediately evolving new committees of candidate models, leads to a better performance increase than querying a bundle of five candidate tests assessed with the very first committee of candidate models. This might be due to select candidate tests that are too similar or the label of one candidate test implying another one.

To avoid bundling too similar candidate tests, we want to bundle preferably diverse queries. But, due to unknown stochastic dependencies among different words, diversity is rather hard to define. In Section 4.2.1 two concepts are presented which are intended to make differences of candidate tests measurable.

So far, we only devoted attention to the user as a source of information considering the desired service, but the OTF market has knowledge about semantics at its disposal. The user provides sequence diagrams using arbitrary operation names. Mapping these operation names to actually existing services makes formal descriptions of the services used available. This description covers semantics of these services. In Section 4.2.2, we consider a concept on how to use the knowledge within the OTF market to also introduce more complex modeling structures for candidate models.

4.2.1 Bundling Queries

Sending a single message for each oracle query which is again responded to via another message produces large amounts of overhead regarding network traffic. Furthermore, the overall response time (regarding the time from initiating the requirements elicitation process to the final presentation of solutions after a certain number of feedback requests) suffers from the transfer times which are needed to deliver messages between the user and the algorithm. Therefore, it is useful to reduce the number of messages while keeping the number of requested candidate tests the same.

Candidate models of the committee for assessing the fitness of candidate tests become more sophisticated with every additional training example. Thus, it has to be expected for performance to drop regarding the quality of finally returned candidate models for bundling queries because of too similar queries bundled within one message. Hence, we aim for preferably different candidate tests. To measure the diversity within a bundle of queries we define the following fitness functions:

Transition Matrix Coverage

When a DFA A reads a word $w = a_1 a_2 \dots a_n$, the transition function determines the transitions to follow. In Section 4.1.1 we represent this transition function by a matrix T. While reading w, certain entries of T are read to determine target states $q_i = T_{q_{i-1},a_i}$ for $i \in \{1, \dots, n\}$.

In order to reward the algorithm exploring more different structures, we count the number of different matrix entries of T used while processing a set of candidate tests M with a committee of candidate models C. As a fitness function this can be expressed by

$$h_{\text{cov}}(M, C) = \frac{\sum_{A \in C} \mid \bigcup_{t=(\ell,w) \in M} \{T_{x,y} \mid T_{x,y} \text{ is used for processing } A(w)\} \mid}{|C| \times |Q| \times |\Sigma|}, \quad (4.16)$$

where M denotes a set of candidate tests, C a set of DFAs A with a matrix T of dimension $|Q| \times |\Sigma|$ as genetic representation and $T_{x,y}$ the entry in row x and column y of T. Note that the maximum value of 1 is not always achievable, since there might be states for which no path exists starting from the initial state q_0 ending in those states.

Maximizing the h_{cov} objective guides the algorithm to explore more within the matrices of the committee's candidate models. If the sum of disagreement

$$h_{\text{setDis}}(M, C) = \sum_{t \in M} g_{\text{dis}}(t, C) \quad (4.17)$$

is maximized simultaneously, we obtain sets of candidate tests which the committee of candidate model disagrees on while preferably using a large share of the matrices T, and therefore of different parts of the DFA.

Using h_{cov}, we ensure exploring sub-structures of DFAs to be rewarded. However, it is still possible to obtain sets of candidate tests which might consist of very similar up to multiple times the same candidate test with maximum coverage proportion. Therefore, in addition to the matrix coverage, we consider the different fractions of candidate models within the committee voting for disagreement.

Fraction Patterns

When assessing the fitness of some candidate test t, a committee of candidate models C votes on the label for t and the disagreement among the candidate models is measured. Hence, the committee of candidate models can be sub-divided into two fractions. First, one fraction $F_1 \subseteq C$ voting on *accept* for t and, second, one fraction $F_0 \subseteq C$ voting with *reject*. Enumerating the candidate models from 1 to $|C|$, we can define a bit vector $V = (v_1, v_2, \dots, v_{|C|}) \in \{0,1\}^{|C|}$, where $v_i := 1$ if the i-th candidate model is in F_1 and $v_i := 0$ if the i-th candidate model is in F_0.

By using the Hamming distance as a diversity measure, we can calculate the average Hamming distance of the bit vectors as a fitness function

$$g_{\text{frac}}(M, C) = \frac{\sum_{i=1}^{|M|-1} \sum_{j=i+1}^{|M|} \Delta(t_i, t_j, C)}{(\sum_{i=1}^{|M|} |M| - i)} = \frac{2 \sum_{i=1}^{|M|-1} \sum_{j=i+1}^{|M|} \Delta(t_i, t_j, C)}{|M|^2 - |M|}, \quad (4.18)$$

where M is a set of candidate tests, C a set of candidate models and Δ a function computing for both candidate tests t_i, t_j the respective fraction pattern vectors and returning the Hamming distance of these two vectors.

By g_{frac} we track the fractions voting for the disagreement and reward for different constellations of those fractions, i.e. that the F_1 respectively F_0 fractions consist of different candidate models for different candidate tests. Implicitly, we consider candidate tests as more diverse if the labels of the states, where they end in the candidate models, are different.

4.2.2 On-the-fly Supply

Currently, the user is required to mention each operation name relevant for the behavior description of the desired service at least once in the initial training examples. Furthermore, the presented approaches consider the user to be the only one who is able to reveal more information about the desired service. The training examples provided by the user involve operation names which are arbitrarily chosen by the user. At this stage, these identifiers might contain semantic information about the behavior of this operation, but this semantics are not understood by the algorithms nor employed to advantage. The current vision of the OTF market contains another step after generalizing the formal model from the provided training examples for mapping the arbitrary operation names to services that actually exist in the OTF market.

Indeed, the step of mapping the operation names contained in the training examples to real services reveals information about the semantics of these services. This information is provided in the form of the Inputs-Outputs-Preconditions-Effects (IOPE) description as explained in Section 2.2.2. Taking this information as a basis, for instance, we are able to recommend services which are frequently combined with an already used service and thereby augment the alphabet as described in Section 4.2.2.1. Furthermore, the details about the semantics of the services may be used for introducing more complex model structures such as branching structures. In Section 4.2.2.2, the idea of how to introduce branching structures to inferred models is described in more detail. Due to the scope of this thesis, we consider only a conceptual sketch of the ideas. Concrete approaches might be covered in future work.

4.2.2.1 Augmenting the Alphabet

For the current approaches, the user is required to mention each operation name at least once within the initial set of training examples. Especially, if the user has only a vague notion of what the service is meant to behave like, this requirement is rather impracticable. In a scenario involving only the user and the algorithm, the algorithm strongly relies upon the data the user provides. Fortunately, in our scenario, the algorithm is located in an OTF market consisting of different participants such as service providers and OTF providers.

Service providers and OTF providers might offer an oracle interface providing recommendations about which operations are often used in combination with a given set of operations. These recommendations may be learned by observing requested and successfully executed service compositions. Another way might be to consider only the similarity of some operations based on the preconditions and effects of operations. For clarification, we consider the running example of describing a shop management service again.

The user wants to request a service that is able to show an admin panel and a customer panel. However, the user provides sequence diagrams using the operation names `showLandingPage`, `login`, and `showAdminPanel`. Hence, the corresponding alphabet extracted from the provided sequence diagrams consists only of these three operation names.

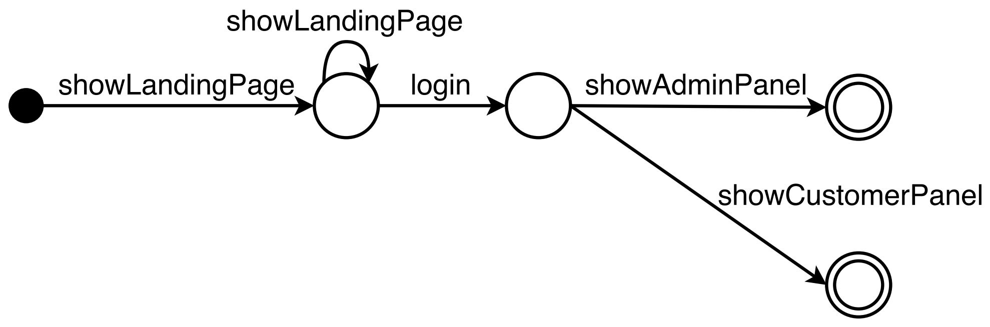


Figure 4.2: Example extended shop management service

A candidate model inferred from the training examples might look like the automaton in Figure 4.3 (for convenience transitions of the complete DFA leading to rejecting states are omitted). This candidate model is already very similar to the model the user has in mind but, obviously, a transition labeled with `showCustomerPanel` is missing. Moreover, since the algorithm is not aware of any operation name `showCustomerPanel`, no additional training examples involving this operation name are requested from the user.

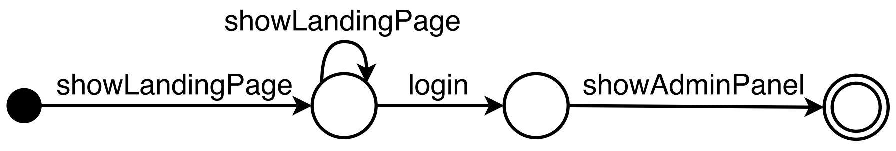


Figure 4.3: Running example model representing the shop management system as DFA

Giving the algorithm access to some recommendation system as an oracle enables the algorithm to identify related operations and forward these recommendations to the user. As shown in Figure 4.4, the user provides a set of sequence diagrams from which the distinct operation names are extracted by the algorithm in order to form an alphabet as usual. The extracted operation names are handed to the black box “Operation Mapper” which maps the operation names to actually existing operations and queries the second black box “Recommendation System” on these operations for other, recommended operations. The recommendations are returned to the algorithm via the “Operation Mapper”, where the recommended operation names are mapped to an operation name corresponding to the naming strategy of the user. Subsequently, the algorithm may propose the obtained recommendations to the user. Finally, the user has to be asked for feedback on the recommended operation names.

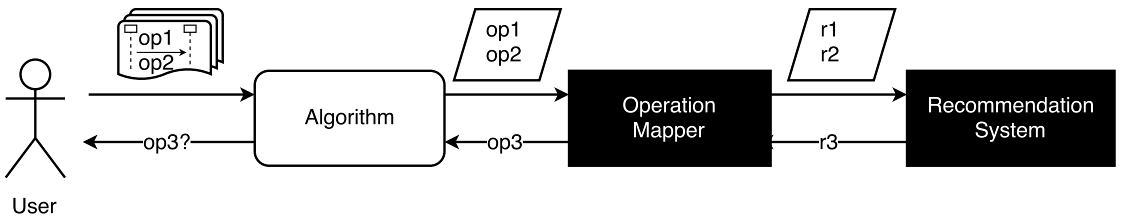


Figure 4.4: Recommendation of operation names for augmenting the alphabet

In terms of “how” to ask for feedback, there are at least two possible solutions: First, the algorithm generates a word containing the recommended operation, and the user is shown the corresponding sequence diagram in order to classify this as desired or prohibited behavior, as usual. If the user indeed classifies this example as prohibited the interpretation of this classification is ambiguous. On the one hand, this could mean that the user might not want to use the recommended operation at all. On the other hand, the user might want to accept the recommended operation, but the presented sequence diagrams simply describe no desired behavior.

Second, the user is asked whether the recommended operation name should be part of the behavior description of the desired service. If so, the user is asked to give an example for desired behavior. Otherwise, the user declines the operation. As a result, the algorithm is able to decide whether to add the name of the recommended operation to the alphabet or not.

In the case of the running example, the recommendation system might propose an operation showing a customer panel, which might be a desired feature for a shop management service. Hence, the algorithm may propose some operation name *showCustomerPanel* to the user. In the case of the shop management service, the user may decide that the proposed operation is needed to describe the desired service and the user provides a training example including the new operation name. After further recommendations, the desired service is finally described by the DFA is depicted in Figure 4.5.

To conclude, considering the OTF market and its possibilities of learning from service composition behaviors enables the requirements elicitation algorithm to propose unmentioned, possibly interesting operation names to the user. On the one hand, these recommendations support the user to refine vague notions of the desired service into a more precise concept. Furthermore, the user is not required to mention all the relevant operation names in the initial set of training examples. From a technical perspective, the recommendation oracle provides the opportunity

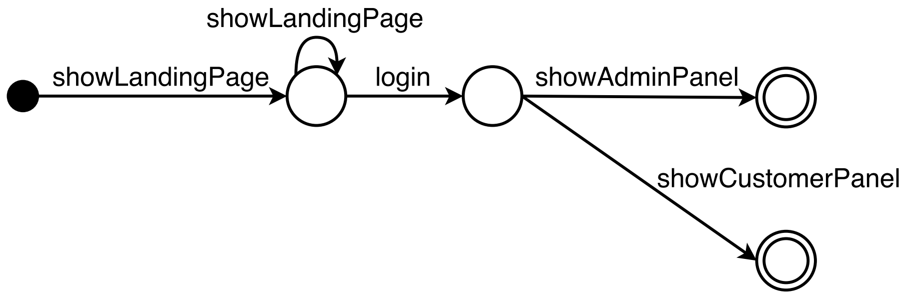


Figure 4.5: Refined shop management system extended by showCustomerPanel operation

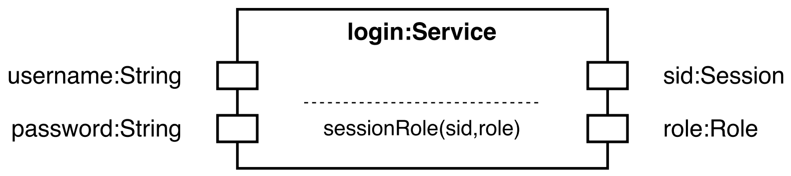
to augment the alphabet, while the user is considered as an oracle for augmenting the training data.

4.2.2.2 Branching Structures

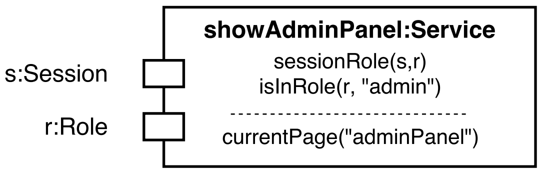
Another way of using the information provided by the OTF market is to use the previously described "Operation Mapper" in order to gather more detailed information about the operation names used in the training examples. Within the OTF market, the behavior of services is described via an Input-Output-Precondition-Effect (IOPE) description. In particular, within a service description, the operations are declared by an IOPE description. By mapping the user's operation names to actually existing operation names, more information about the used operation names and thus about the behavior of the desired service as a whole is revealed.

The formal model for the desired service shown in Figure 4.5 describes a simple shop management service, providing different panels. From the plain operation names, it is difficult to see coherences between the operation names. Mapping the operation names `Login`, `showAdminPanel` and `showCustomerPanel` to actually existing operations provides more information. Let in Figures 4.6a, 4.6b and 4.6c be the corresponding IOPE descriptions for these operation names.

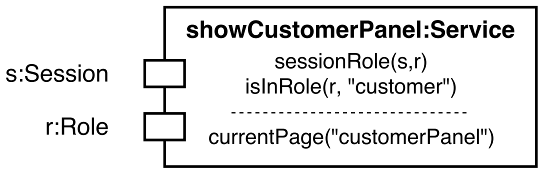
The login operation takes two input parameters `username` and `password` of type `String` and outputs one `Session` object and one `Role` object. After applying the operation, the effect `sessionRole(·,·)` holds. The outputs of `login` are required as input parameters by the two operations `showAdminPanel` and `showCustomerPanel`, and the effect of login is used as a precondition. Additionally, the two operations require a predicate `isInRole`. This predicate ensures the logged in user to have the correct role for accessing the respective panel. However, assuming the predicate to be evaluated given the `Role` object and the respective



(a) Service taking a username and a password as input parameters and returning a Session object together with a Role object



(b) Service requiring a logged in user, and that the respective user havin the role "admin"



(c) Service requiring a logged in user, and that the respective user has the role "customer"

Figure 4.6: Example IOPE descriptions of the services login, showAdminPanel, and showCustomerPanel

constant “admin” or “customer”, the `isInRole` predicate is interpreted equivalent to a guard in a DFA. We refer to this kind of predicate as *interpreted predicate* in the following.

Tracing the interpreted predicate back into the model description (by some appropriate mapping), we can use the interpreted predicate for further restrictions in the model by refining it with guards. Guards are conditions required to hold for enabling a transition to be used. As a user in the role of a customer may not be able to show the administration panel, the DFA modeling the shop management service should also restrict the access to the admin panel for user with an appropriate role. Vice versa, the admin is not a customer and therefore should have no access to the customer panel. By introducing guards for the interpreted predicate `isInRole`, it is possible to model these constraints, and we obtain the model depicted in Figure 4.7. To support this refinement, the formalism defined as Service Specification Language (cf. Section 2.2.2) has to be adapted.

To sum up, by mapping the arbitrarily chosen operation names provided by the user to operations actually existing in the OTF market, additional information about the semantics of these operations is provided. As outlined above, this information can be used to introduce elements of statecharts (cf. Section 2.1.1) for more complex model structures such as branching structures with guards.

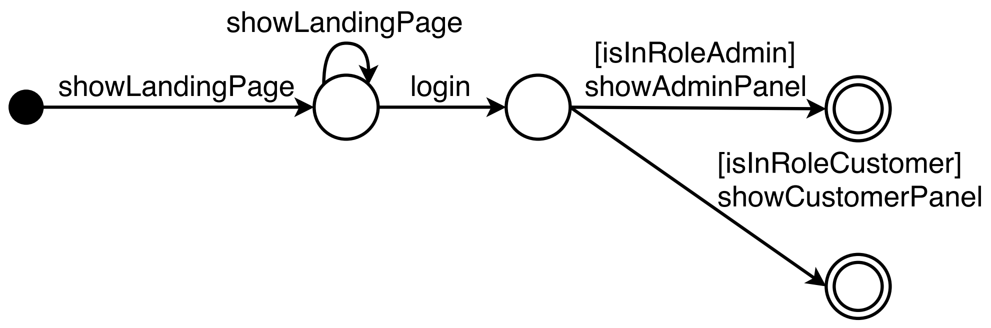


Figure 4.7: Example model refining the shop management system using guards

In general, the OTF market provides abundant information about semantics, which is not considered within the current approaches. The concepts presented above outline a way of how to make use of this information. However, the presented concepts make assumptions on the existence of two black boxes, namely “Recommendation System” and “Operation Mapper”. A learner observing executed service compositions might be one possible way of implementing such a recommendation system. Another possibility is to provide recommendations according to the descriptions of available services and encapsulated operations. However, the “Operation Mapper” is a highly non-trivial task to implement since the “Operation Mapper” is only provided a set of operation names that are chosen arbitrarily by the user. The conception and realization of both black boxes is beyond the scope of this thesis and, thus, left for future work.

5 Implementation of the REAL Framework

This chapter addresses the implementation of a framework for performing requirements engineering, using active learning techniques. To this end, we abstract the semi-automated requirements process itself and the objects involved in that process to form the Requirements Elicitation by Active Learning (REAL) framework. The primary goal of this framework is to enable the comparison of different semi-automated requirement elicitation algorithms. Besides, due to an event-based architecture, classes within the framework are highly decoupled, and therefore they can be easily exchanged by other implementations. Furthermore, the framework provides a central registry keeping track of registered objects. Extending the framework with further implementations only requires the developer to register these implementations at the central registry. After the registration, the registered implementations are ready to be used.

From the approaches and concepts presented in the previous chapter, we can abstract three roles in total. First, the role of the user, who wants to request a desired service from the OTF market and provides examples of desired and prohibited behavior for this service. Second, the requirements elicitation algorithm, which generalizes a formal model describing the desired service from the examples provided by the user. Third, a recommendation system providing supplementary information to the requirements elicitation algorithm.

Additionally, further entities and listeners need to be included for benchmarking, e.g. generating problem instances and performing post-evaluation of the returned solutions. Section 5.1 gives details concerning the responsibilities and capabilities of the roles, entities, listeners and their standard implementations.

The interaction between these objects is carried out according to a fixed protocol. This protocol is documented in Section 5.2, and the interaction between roles and entities is described.

5.1 Roles, Entities, and Listeners

Within the REAL framework, we distinguish clearly between objects relevant to a real world application and those only needed for benchmarking or logging purposes. In this way, the framework can also be integrated into real world applications and already existing toolchains, such as SeSAME[1].

According to the three different roles involved in the requirements elicitation process, which we identified as users, requirements elicitation algorithm, and a recommendation system, we design corresponding classes in the framework. While the user is abstracted to an oracle able to reveal more information about the target model, the requirements elicitation algorithm is abstracted to an automatic requirements engineer (ARE). Finally, the supplier typifies the recommendation system, which provides any kind of information available in the OTF market. In order to enable benchmarking different implementations for the three roles, we need additional entities within the framework for controlling the execution of different experiments (control), generating problem instances (generator) and for running a post-evaluation comparing the solution of the ARE with the target model (validator).

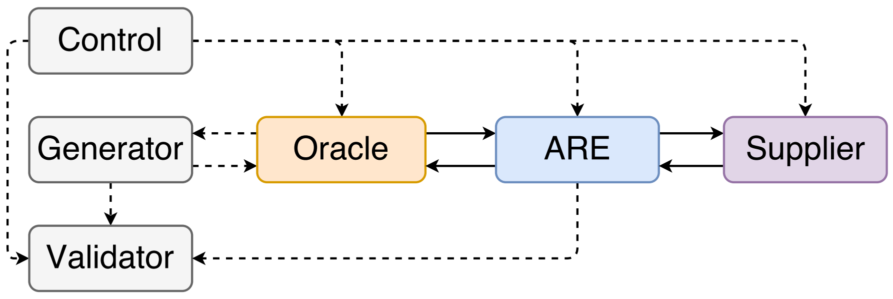


Figure 5.1: Roles and entities within the REAL framework. Arrows denote communication between the instances, where the direction of the arrow describes the direction of the sent messages

In Figure 5.1 the interaction between the roles and entities is illustrated. While the interaction relevant for the requirements elicitation process is depicted by solid arrows, the interaction that is only needed for benchmarking purposes is denoted by dashed arrows. The direction of the arrows corresponds to the direction of the communication, i.e. a leaving arrow from oracle to ARE indicates that the oracle sends events to the ARE. Furthermore, while the roles are highlighted by colored

[1]https://sfb901.uni-paderborn.de/sfb-901/projects/tools-demonstration-systems/sesame.html - Accessed: 2017-02-23

boxes, benchmark entities are shown in gray boxes.

The main idea of the interaction between the roles and entities is that the control initiates a benchmarking process and provides the parameters of the benchmark setting to the relevant objects. Initially, the oracle requests a new problem instance from the generator, which is also handed to the validator for performing post-evaluation. This is followed by the requirements elicitation process which is carried out by the roles oracle, ARE and supplier. During the requirements elicitation process, the validator is provided with current population data in order to perform a post-evaluation.

In the following, we consider the responsibilities and capabilities of the roles and entities in more detail. Section 5.1.1 gives more insights into the oracle itself. How the oracle makes use of the generator is explained in Section 5.1.2. Subsequently, we elaborate the role of the ARE in Section 5.1.3. Section 5.1.4 deals with the entity validator, which obtains data from the requirements elicitation process in order to perform a post-evaluation. In Section 5.1.6, we consider the control entity. Finally, Section 5.1.7 gives details about listeners who can subscribe to the framework in order to observe all the communication between roles and entities.

5.1.1 Oracle

The *oracle* represents the role of the user within the REAL framework. Hence, the oracle is responsible for initiating the requirements elicitation process. This is done by sending a message to the *ARE* providing an initial set of training data and a number of states ($|Q|$). Here, the number of states might be even only an estimate, such as a maximum number of necessary states to represent this model. Furthermore, the *oracle* is responsible for answering oracle queries received from the *ARE*.

While performing a benchmark, the oracle simulates the behavior of the user. In a real scenario, the user has a target model in mind for which the user provides examples in order to manifest these notions as a formal model. This is a little different in the case of benchmarking. The target model has to be made explicit in order to enable the oracle to answer the oracle queries automatically. Therefore, the oracle is given access to the *generator* entity in order to generate an explicit target model, which can then be used for answering the oracle queries.

Since in a real application it might also happen that the user makes mistakes classifying sequence diagrams, the REAL framework provides the *pHonestOracle* as a standard implementation. The pHonestOracle is configured via a parameter p, which determines the probability of classifying a word correctly. Initial training data is sampled uniformly at random from the set of all words with a given maximum length $\ell_{\max}$. The initial training data, as well as subsequent oracle queries, are classified correctly with a probability of p, i.e. if for instance $p = 0.5$, on average, only half of the words are labeled correctly.

5.1.2 Generator

The *generator* belongs to the entities of the REAL framework, which are only used in a benchmark scenario. It is responsible for generating a target model ,and therefore, for providing an explicit target model to the oracle. Furthermore, the target model is also provided to the validator for post-evaluation.

In order to request a generated target model, the generator can be used in two different ways: First, the generator is provided a number of states and an alphabet which has to be used for labeling transitions. Second, instead of an alphabet, the generator is only given the size of the alphabet, such that p.r.n. semantics for the symbols of a self-generated alphabet can be taken into account. In the latter case, the oracle and the validator are also provided the generated alphabet.

As a standard implementation, we use a simple random generator, which is provided an alphabet and a number of states. Outgoing transitions for each state are drawn uniformly at random, such that there is no semantics covered in the generated target models. In Section 6.1.2, the generation routine is described in more detail.

5.1.3 Automatic Requirements Engineer

The role of the *automatic requirements engineer* (ARE) is responsible for generalizing the training examples obtained initially from the oracle to a deterministic finite automaton. Interacting with the oracle and supplier via messages, the ARE is able to gather more information about the target model and its semantics. Furthermore, for monitoring the requirements elicitation process intermediate as well as the final results are sent to the validator. Finally, after a fixed number of oracle queries, the ARE returns its solutions to the oracle respectively the user.

A general version of an active ARE is standardly implemented, summarizing the common process for ARE's as sketched in Figure 5.2. Each requirements elicitation process starts with obtaining initial training data, followed by evolving candidate models for g generations. Then, for r many allowed oracle queries, candidate tests and candidate models are evolved in an alternating order. After the evolution of candidate tests, the best individual is sent to the oracle. On receiving the label for the requested candidate test and augmenting the training data with this new training example, candidate models are evolved again. Finally, the resulting candidate models are presented to the oracle as the solution.

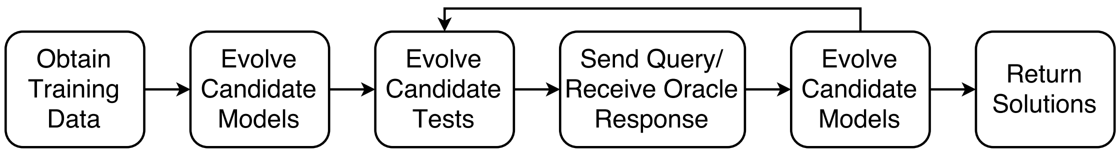


Figure 5.2: Sketch of common process for ARE's

Since the overall structure is the same for the approaches presented in the previous chapter, we use a general active ARE with different algorithm setups for evolving candidate models respectively candidate tests. An algorithm setup consists of parameters for the population, an evolutionary algorithm and a set of fitness functions. The general ARE obtains one set of algorithm setups for the evolution of candidate models and one set for the evolution of candidate tests. This way, we are able to split up the populations, combining different algorithms to hybrid evolutionary algorithms.

In contrast to active ARE's, passive ARE's are required to pass a set of algorithm setups for the evolution of candidate models only. Unless otherwise stated, an instance of the general version of a passive ARE replaces the step of evolving candidate tests by drawing a word uniformly at random from the set of all words with a given maximum length.

The algorithms listed in Table 4.1 (EEA, eeMOO, eeMOO2, pMOO) are preconfigured and available per default.

5.1.4 Validator

The *validator* is another entity of the framework, which is solely involved in the benchmark processes. Within a benchmark process, the validator is responsible for performing a post-evaluation of candidate models, i.e. determining to which extent the language of the evolved candidate models coincides with the language of the target model. To this end, the validator is provided the target model generated by the generator.

In order to perform the post-evaluation, the generator uses a set of unseen words (test set), i.e. words which are neither part of the initial training data nor are requested in any oracle query by the ARE. By calculating the proportion of words labeled the same by the target model and the candidate model, the validator determines a performance value for the respective candidate model. The performance value is considered to be an indicator of the quality of the candidate model, i.e. how well the candidate model fits the language of the target model.

Within the REAL framework, two different validators are available, which only differ in the way the test examples are created. The `ExplorationValidator` performs a breadth-first-search traversing the target model adding all visited paths of unseen words to the test set until the test set has a certain size. Hence, the test set is created in a deterministic way. In contrast to that, the `SampleValidator` draws words uniformly at random from the set of all words with a given maximum length and adds these words to the test set if they are not used in the training data or oracle queries.

5.1.5 Supplier

The role of the *supplier* is a representative role for all kinds of knowledge sources located in the OTF market. In particular, meaning the supplier represents an interface to the OTF market for the ARE. As proposed in Section 4.2.2 the supplier is provided to the ARE in terms of an oracle. In this way, the ARE is able to access information contained in the OTF market via a simple event-based interface.

Since the concepts presented in Section 4.2.2 are still immature and rely on black boxes, which are located in the OTF market, a supplier instance is out of the scope of this thesis. Before realizing a supplier instance, these black boxes have to be implemented. Hence, the REAL framework provides no instance for a supplier, yet.

5.1.6 Control

The control entity is responsible for initiating the benchmark process. Therefore, the control is provided some benchmark task. The benchmark task contains information about which roles and entities are supposed to be involved in the benchmark process and further parameters, concerning the parameterization of the involved algorithms. After the setup of the involved algorithms, the oracle is signaled to start the requirements elicitation process.

In the REAL framework, two different implementations of control entities are provided as standard implementations. First, the `ChunkControl` is given a set of tasks (chunk) containing all the information relevant for defining an experiment, i.e. which algorithms are evolved, which parameterization is used for these algorithms and what data is post-evaluated. One by one, the tasks contained in the given chunk are issued to the roles of the framework and executed. Second, for running the `ArrayControl`, lists of parameters (e.g. size of the population, size of the alphabet, etc.) are stated in a properties file. From these lists of parameters the cross product is taken, each element of that cross product defining a single task. Subsequently, these tasks are bundled and executed using the `ChunkControl`.

5.1.7 External Listeners

In order to observe the requirements elicitation process and to extract data, which is exchanged between the roles and entities, external listeners can be registered to the REAL framework. As the name already indicates, external listeners obtain read-access only. Therefore, listeners are not able to participate actively in the requirements elicitation process. The listener entity can be used for logging events sent during a benchmark process, serializing post-evaluation data, etc. For instance, in order to observe the evolutionary runs, the best and average fitness of the population may be serialized generation by generation with the `BestAvgStatsSerializer`.

5.2 Benchmark Protocol

Before starting a benchmark, a set of tasks has to be defined as input for the control entity. A task contains various information about the experiment setup, such as which implementations of roles and entities are involved in the processing, parameters describing the problem instance (alphabet size, the number of states), and parameters for the setup of the algorithms.

As shown in Figure 5.3, the control entity iterates over the set of tasks defined in advance and activates the instances listed in the task description and deploys the setup parameterization to the particular algorithms. Subsequently, the oracle is instructed about the task and in particular about the properties of the target model. This information is forwarded to the generator entity, which returns a target model. The generated target model is also provided to the validator entity, enabling post-evaluation by comparing candidate models to the target model.

After the target model is returned to the oracle, an initial set of training examples (training data) is created by the oracle and provided to the ARE. Simultaneously, this initiates the requirements elicitation process between the oracle and the ARE.

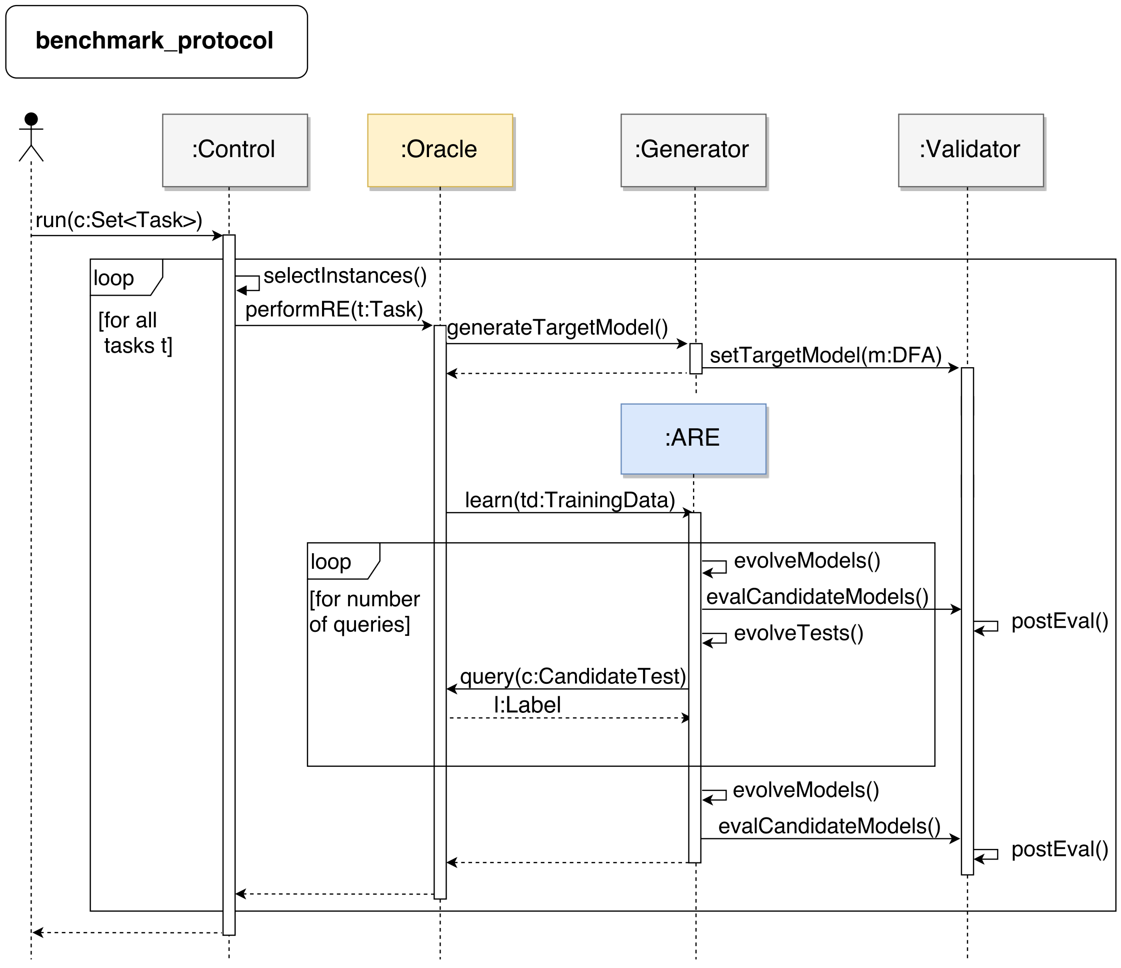


Figure 5.3: Protocol run for benchmarking algorithms

Within the task, it is also specified how many queries the ARE is allowed to send to the oracle. On receiving the initial training data, the ARE alternates between evolving candidate models and candidate tests. After evolving candidate tests, the ARE sends an oracle query for the best individual in the population of candidate tests. The oracle returns a label for the queried candidate test and the new training example obtained this way is added to the training data. This sequence is repeated until the ARE queried for as many candidate tests as it is specified in the task description. Finally, the ARE again evolves candidate models for the last time and returns the solutions to the oracle as a result of the overall requirements elicitation process.

Furthermore, after each generation, the candidate models are provided to the validator. The validator, being in possession of the target model, compares all the candidate models against the target model. In this way, the validator provides information about the progress of the requirements elicitation process. Furthermore, the validator provides the calculated information to listeners, e.g. for statistical observation of the process. Note that the information calculated by the validator is not provided to the ARE at all, so the ARE does not get to know the target model except the training data obtained from the oracle.

On receiving the solutions returned by the ARE, the oracle notifies the control, that the execution of the current task is finished. The whole procedure is repeated until all specified tasks are executed.

6 Evaluation

This chapter deals with the benchmark and comparison of the requirements elicitation approaches presented in Chapter 4. To this end, the implementation of the REAL framework is used as introduced in Chapter 4. In Section 6.1, we state the setup for the evaluation. Results of the evaluation comparing pMOO by van Rooijen and Hamann [RH16] and eeMOO and eeMOO2 as designed in the course of this thesis are presented in Section 6.2. The comparison of the state-of-the-art algorithm EEA by Bongard and Lipson [BL05] with eeMOO2 is presented in Section 6.3.

6.1 Evaluation Setup

The configuration of the different algorithms and the setup of the remaining framework depends on many different variables. For the following evaluation, some of these variables are fixed to certain values, which are listed in Section 6.1.1. Traditional approaches for grammatical inference consider binary alphabets only. Therefore, Section 6.1.2 deals with an algorithm for generating problem instances with arbitrary alphabet sizes. In Section 6.1.3 the evaluation process is described. Finally, in Section 6.1.4 a self-developed management tool is presented, which was used to distribute the load for conducting the experiments over a distributed, heterogeneous cluster of workers.

6.1.1 Fixed Parameters

The first question regarding an evaluation task concerns the roles involved. In the following evaluations we use an oracle which labels all words correctly, and the Random Uniform Generator for generating target models (for details cf. 6.1.2). For post-evaluating candidate models evolved by the learner, we use the exploration validator.

Due to the very small set of training examples, preliminary experiments revealed that learners tend to evolve too specialized candidate models. Therefore, we set the number of generations to a value of 5 in order to avoid overfitting. The number

Parameter	**Value**
Oracle	PHonestOracle, p=1
Validator	Exploration Validator
Generator	Random Uniform Generator
Number of generations	5
Size of population	100
Maximum word length	20
Amount of initial training data	10
Number of oracle queries	100/1200
Evaluation cycle	after each generation
Evaluation target	whole population
Test set size	30,000

Table 6.1: Overview of fixed parameters for the evaluation

of individuals is fixed to 100, which results in 50 individuals per subpopulation using EEA, eeMOO or eeMOO2. All the populations are post-evaluated completely after each generation, using a test set of 30,000 unseen words in order to limit the evaluation time for a single generation to approximately 1 second. The maximum length of words ($\ell_{\max}$) is limited to 20. The learners are given an initial set of 10 training examples and are allowed to query for 100 resp. 1200 labels to augment the initial training data. An overview of the fixed parameters is given in Table 6.1.

6.1.2 Problem Generation

Traditionally, algorithms for grammatical inference are evaluated for problem instances with binary alphabet only. Emerged from scientific challenges, various generators for producing target models over binary alphabets exist [SCZ04; LP98]. However, the performance on larger alphabets is a crucial aspect of the usage in OTF markets, since a binary alphabet would restrict the number of used services to 2. Therefore, we construct a simple generation algorithm for problem instances with arbitrary alphabet size as shown in Listing 6.1.

The algorithm in Listing 6.1 obtains as an input a number of states ($|Q|$) and the size of the alphabet ($|\Sigma|$). In line 2, Q is initialized with $Q := \{0, \dots, |Q| - 1\}$ and Σ is set to $\{0, \dots, |\Sigma| - 1\}$, q_o is set to 0. After the initialization, the transition function δ is constructed by drawing uniformly at random target states q' from Q for all combinations of states $q \in Q$ and input symbols $a \in \Sigma$, i.e. if the DFA is

Listing 6.1: Random Uniform Generator

```
1  On input $(|Q|, |\Sigma|)$:
2  $Q := \{0, \ldots, |Q|-1\}$ , $\Sigma := \{0, \ldots, |\Sigma|-1\}$ , $q_0 := 0$
3  $\forall q \in Q \forall a \in \Sigma$: $q' \overset{\$}{\leftarrow} Q$ , $\delta(q,a) := q'$
4  $\forall q \in Q$: $b \overset{\$}{\leftarrow} \{0,1\}$
5    if $b = 1$ then
6      $F := F \cup \{q\}$
7    endif
8  if $\forall q \in Q$: $\exists n \in \mathbb{N}$ $\exists w = a_0 a_1 \ldots a_n \in \Sigma^n$ such that $q = \delta(\delta(\ldots\delta(q_0, a_0)\ldots, a_{n-1}), a_n)$
       then
9    return $(\Sigma, Q, \delta, q_0, F)$
10 else
11   go back to step 2.
12 endif
```

in state q and reads the input symbol a, then the next state is q'. By generating transitions for all combinations of q and a, the constructed δ is a total function. In lines 5 to 7, the set of accepting states F is constructed by flipping a coin for each state. Hence, each state is added to F with probability 0.5. Next, in line 8 it is checked whether each state $q \in Q$ is reachable from the initial state. If every state is reachable, the constructed automaton is returned. Otherwise, we start all over again with the construction of the transition function δ.

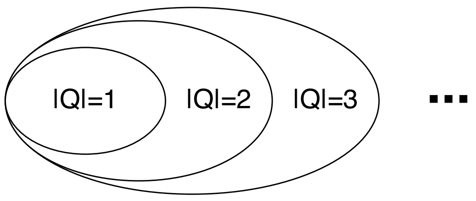


Figure 6.1: Target model spaces

Note that the generation routine does not ensure that there is no other model which has fewer states and accepts the same language as the generated model. In the worst case, it is possible that the generated model can be represented by an automaton with a single state. In order to make trivial solutions or solutions with fewer states less likely, the validation step in line 8 ensures that at least all states are reachable, and therefore also possibly used for labeling words. Figure 6.1 displays the spaces from which the target models are drawn in the generation algorithm presented above. If for example $|Q|$ is set to 3, it is also possible that the resulting automaton can be minimized to an automaton with 2 or even only 1 state.

Hence, $|Q|$ denotes an upper bound on the maximum number of states necessary to accept the language of the target model. However, in our scenario, in which the user has a model in mind, the user might give only an estimate of the complexity of the target model instead of a definite number of states that would be needed.

The REAL framework presented in Chapter 5 provides an implementation of the algorithm for generating target models as described above. The instance of that generator is identified by the Random Uniform Generator.

6.1.3 Evaluation Process

The evaluation process covers two steps in total. In a first step, the generator routine produces a fixed number of samples of target models together with an initial set of training examples for each data point in the analysis space (as defined in the following). Second, the algorithms pMOO, EEA, eeMOO, and eeMOO2 are given access to a labeling oracle and the initial training data. Then, the different algorithms are applied to the same problem instances. We briefly describe the analysis space, the observation space, the hardware used for conduction of the experiments, and the software used to distribute the experiments dynamically to a heterogeneous cluster of computational resources.

The Analysis Space

As input for the generator, we use the maximum number of states ($|Q|$) required to represent the target model and the size of the alphabet ($|\Sigma|$). For $|Q|$ we defined a range from 3 to 12 and for $|\Sigma|$ from 3 to 10. In one case the learner is allowed to send 100 queries to the oracle, and in the other case, the number of allowed queries is set to 1200. While the experiments allowing for 100 queries were sampled 50 times, the scenarios allowing for 1200 queries were sampled only 20 times due to the computational costs of the post-evaluation.

In total, 19,360 experiments were conducted. The problem setup for the experiments allowing for 100 queries is defined by the set $\{3, \ldots, 12\} \times \{3, \ldots, 10\}$. We applied the algorithms pMOO, EEA, eeMOO, and eeMOO2 to each of the induced 80 points of evaluation for 100 queries. In the case of 1200 allowed queries the problem setup is the set $\{4, \ldots, 9\} \times \{3, \ldots, 9\}$. Altogether, this sums up to $80 \cdot 4 \cdot 50 + 42 \cdot 4 \cdot 20 = 19,360$ experiments. An overview of the experiments is given in Table 6.2.

Algorithm	$\|Q\|$	$\|\Sigma\|$	Queries	Samples
EEA, eeMOO, eeMOO2, pMOO	3,...,12	3,...,10	100	50
EEA, eeMOO, eeMOO2, pMOO	4,...,9	3,...,9	1200	20

Table 6.2: Overview of conducted experiments

The Observation Space

The data measured when conducting an experiment can be categorized by whether the data is actually available to the learner or not. More precisely, we track the fitness values calculated by the learner, and on top of that, we measure fitness values calculated by the validator doing a post-evaluation of the candidates.

However, in order to compare the different approaches, we consider the accuracy of the evolved candidate models on a set of unseen test examples. In the following, the *performance* denotes the accuracy of the best individual with respect to the test set.

Furthermore, we use the Mann-Whitney U test, also known as Rank-Sum Test, in order to determine, whether differences in performance between the algorithms are significant [MW47]. The Rank-Sum Test is a non-parametric statistical test, which is used to test whether two distributions of random samples belong to the same universe. In order to find significant differences in performance, we apply the Rank-Sum Test to the performance values of the sampled best individuals of the last generation returned by two algorithms. To this end, we assess whether the distributions induced by the algorithms coincide. As a result of the Rank-Sum Test, we obtain a *p-value*. If the *p-value* is smaller than 0.05, significant differences are observed between the two algorithms.

Experiment Execution

Calculations for the results presented here were performed on resources provided by the IRB University Paderborn[1] and resources by Contabo[2]. Since the experiments have no time dependency and can be reproduced on arbitrary computer systems because of seeded randomization, different nodes were involved in the evaluation process. In total, up to 634 logical CPU cores (40×4, 6×6, 51×8, 3×10) and 1,29TB main memory were used to process the experiments.

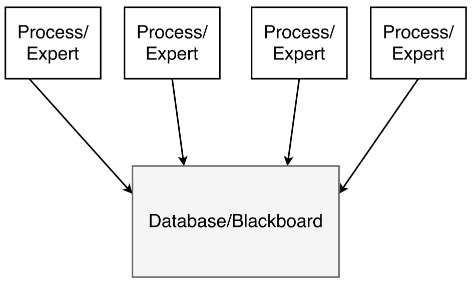
[1]https://cs.uni-paderborn.de/en/irb/ - Accessed 2017-02-08
[2]https://contabo.de/ - Accessed 2017-02-08

For automating the distribution of the experiments to this number of heterogeneous nodes, a software-based cluster management tool was developed, which is described in the following.

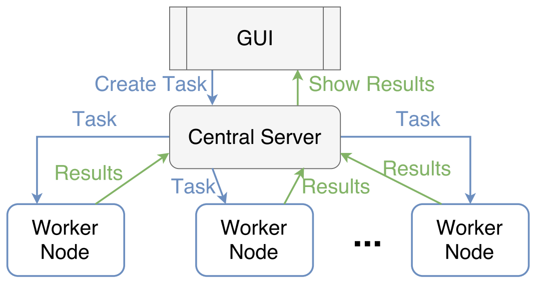
6.1.4 Software-Based Experiment Conduction Cluster

Following the blackboard architecture style, as illustrated in Figure 6.2a, a software-based dynamic cluster management tool was designed and implemented, enabling automated and distributed experiment conduction. According to the blackboard architecture style, a central instance, i.e. the *blackboard*, maintains the knowledge, for instance, the blackboard might be a database. *Experts* or, from a technical perspective, *processes* fetch tasks from the blackboard, solve these and write the results back to the blackboard.

In our scenario, the blackboard is implemented by a central server maintaining a database containing records specifying experiments and records containing the measured results when conducting the experiments. Each of the 100 nodes takes over the role of the expert fetching experiments from the central server, conducting these experiments using the REAL framework and sends reports of measured properties to the central server, which stores the data in a database. The architecture of the scenario is illustrated in Figure A.1f. Moreover, the central server provides a graphical user interface (GUI) in order to enable users to specify experiments and to visualize the measured data.



(a) Schema of Blackboard Architecture (see [GS94])



(b) Software-based Cluster Management Tool Architecture Sketch

Figure 6.2: Blackboard architecture pattern implemented in software-based dynamic cluster management tool

In order to increase the throughput and avoid too much network traffic overhead, tasks are bundled in chunks of for instance 10 tasks. For fetching the chunks from the central server, a client is deployed and run at each of the 100 nodes. The client software is responsible for fetching new chunks from the central server, run the REAL framework using the ChunkControl locally. Periodically, the client software sends a heartbeat to the central server, such that the central server notices when a node goes off-line. When the execution of the tasks contained in the fetched chunk is finished, the client reports the measured data to the central server.

In a further step, we average the measured data belonging to samples of the same experiment setup. The averaged data can be visualized in the GUI of the central server. By recording how many samples already have been used for averaging the collected data, additional samples can be retrospectively added if we desired to increase the sample size. Also, each measured data point is stored in a separate record which allows for an individual sample size.

6.2 Passive VS Active Multi-Objective Optimization

We start the evaluation by analyzing the effect of introducing active learning techniques (eeMOO, eeMOO2) to the passive requirements specification approach by van Rooijen and Hamann [RH16] (pMOO) (cf. Section 4.1.6). To this end, we carry out a first evaluation of the pMOO applying the algorithm to different settings. In order to ensure equal conditions and the same amount of computation time for the evolutionary runs, pMOO is provided additional training examples that are drawn uniformly at random from the set $\{w \in \Sigma^n \mid 0 \leq n \leq \ell_{\max}\}$. Hence, all the training data pMOO obtains, is drawn uniformly at random. We, thus, compare the effect of strategically augmenting the training data to randomly chosen additional training data.

The training data could be provided to pMOO directly from the very beginning, but for that, a different parameterization (number of generations, population size) would be necessary in order to remain fair with respect to computational resources. However, due to problems as for example overfitting, fairness among different parameterizations cannot be defined trivially. Therefore, we proceed as aforementioned by randomly choosing additional training examples.

In Section 6.2.1, the results for the comparison of pMOO and eeMOO are presented. Subsequently, pMOO is compared to the second active multi-objective optimization approach eeMOO2 in Section 6.2.2. Finally, we discuss the results in Section 6.2.3 and draw a conclusion. While for the comparison between pMOO and eeMOO we will notice no advantage of eeMOO over pMOO, we will see that choosing different objectives as in eeMOO2 leads to a significant improvement.

For both comparisons, we summarize the evaluation results in the form of two heat maps showing the difference in performance after 100 queries and after 1,200 queries. In addition, we choose six exemplary settings, lying on the diagonal of the analysis space (i.e. $|Q| = |\Sigma|$), in order to give an impression of how an increasing number of states and size of alphabet affects the performance of the presented approaches.

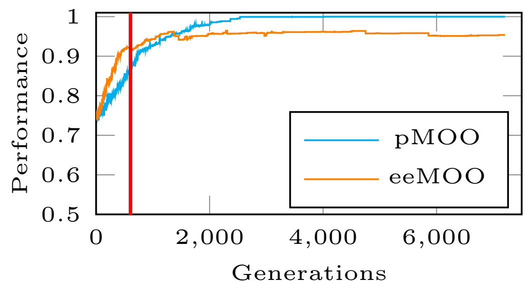
6.2.1 Results for pMOO VS eeMOO

To compare the quality of the returned candidate models, in Figures 6.3a to 6.3f the development of the test set performance for evolutionary runs allowing for 1,200 oracle queries is shown. While the performance of pMOO is represented by a cyan-colored line, the performance of eeMOO is depicted in orange.

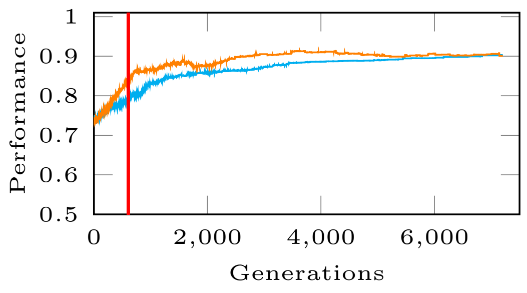
Furthermore, since we are mainly interested in whether the performance can be improved by applying the active learning techniques, the remaining settings are summarized in two heat maps. The first heat map in Figure 6.3g shows a profile of the difference in performance taken after 100 queries. In Figures 6.3a to 6.3f, a red vertical line denotes the point in time that this snapshot is taken. The other heat map in Figure 6.3h shows the performance difference after the full evolutionary runs, i.e. after 1,200 queries.

The *performance difference* (PD) is calculated by subtracting the test set performance of pMOO from the test set performance of eeMOO. Thus, a positive PD denotes that the solutions returned by eeMOO classify more examples of the test set correctly than the ones returned by pMOO.

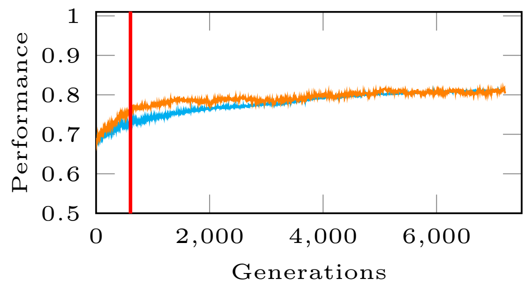
As it can be seen in Figures 6.3a and 6.3b for smaller target models, the performance of candidate models by eeMOO increases faster than for pMOO, in the first place, but pMOO catches up in performance after 1,500-2,000 generations. Especially noticeable is that eeMOO seems to lose good solutions frequently, while the performance of pMOO is nearly monotonically increasing. In the case of $|Q| = 4, |\Sigma| = 4$, pMOO even passes eeMOO and reaches a performance of 1. Considering the setting $|Q| = 5, |\Sigma| = 5$, the two algorithms perform about equal. The rather strong oscillations for both indicate that the good solutions are out of scope using the combination of objectives. In the remaining graphs shown in Figures 6.3d to 6.3f a small tendency for eeMOO over pMOO can be recognized.



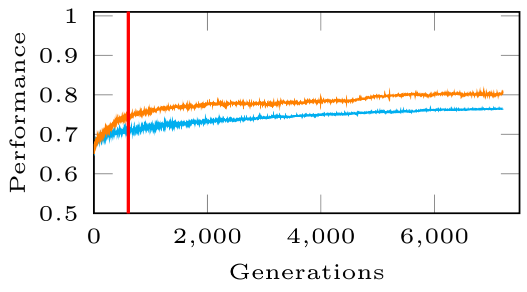
(a) $|Q| = 4$, $|\Sigma| = 4$ ($p \approx 0.0033$)



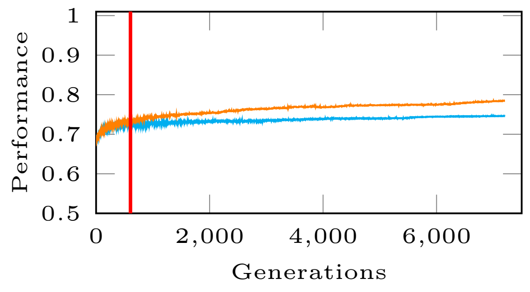
(b) $|Q| = 5$, $|\Sigma| = 5$ ($p \approx 0.741$)



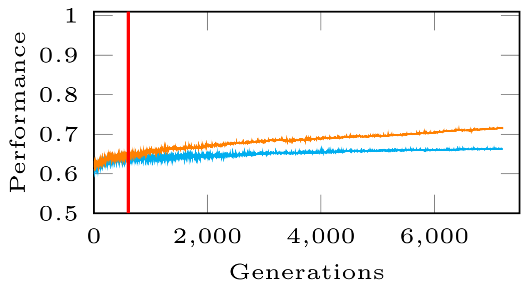
(c) $|Q| = 6$, $|\Sigma| = 6$ ($p \approx 0.7584$)



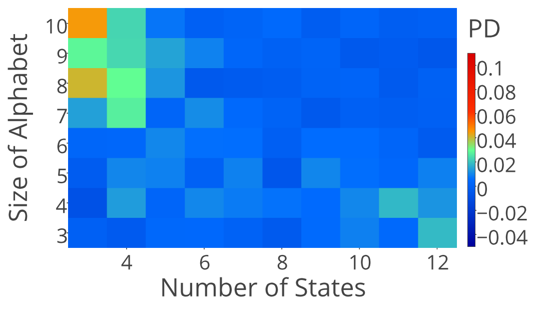
(d) $|Q| = 7$, $|\Sigma| = 7$ ($p \approx 0.0922$)



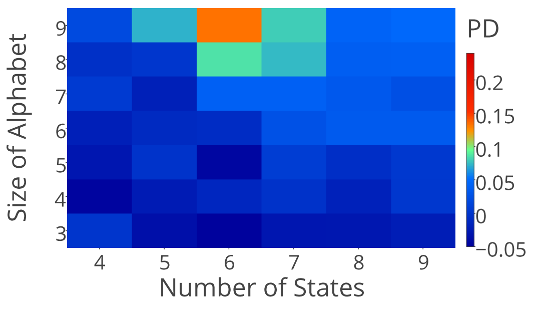
(e) $|Q| = 8$, $|\Sigma| = 8$ ($p \approx 0.1572$)



(f) $|Q| = 9$, $|\Sigma| = 9$ ($p \approx 0.0375$)



(g) Heat map over performance difference (PD) eeMOO−pMOO, 100 queries



(h) Heat map over performance difference (PD) eeMOO−pMOO, 1200 queries

Figure 6.3: Results for the comparison of pMOO and eeMOO

The heat maps in Figures 6.3g and 6.3h summarize the results of all settings. After 100 queries as well as after 1,200, apart from some exceptions, no significant improvements of eeMOO over pMOO can be observed. In some settings as for for instance $|Q| = 6, |\Sigma| = 5$ after 1,200 queries, the performance difference is negative, i.e. the performance of pMOO is greater than the performance of eeMOO.

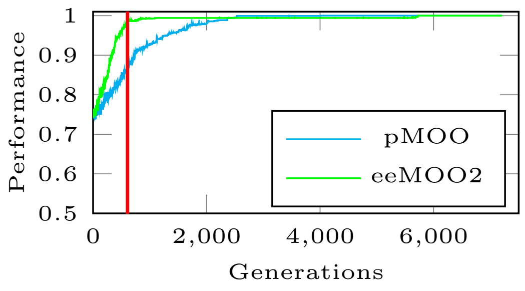
Summing up, eeMOO shows no advantage over pMOO, although eeMOO strategically chooses queries to the oracle by evolving candidate tests applying active learning techniques. Instead, in some settings, the results of the evaluation even indicate that choosing queries uniformly at random even leads to a better performance. Especially, in the settings with 1,200 allowed queries, the performance difference becomes negative in about half of the settings. Except for a few settings, the performance of the two algorithms is competitive only.

6.2.2 Results for pMOO VS eeMOO2

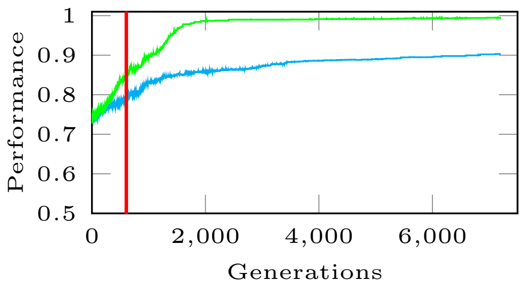
In the following, we consider the second version of the strictly co-evolutionary multi-objective optimization approach (eeMOO2). Unlike eeMOO, eeMOO2 uses only the two objectives f_{all} (cf. Equation 4.2 in Section 4.1.2) and f_{reach} (cf. Equation 4.13 in Section 4.13) for assessing the fitness of a candidate model, where only f_{all} directly relates to the training data. The results of the comparison between pMOO and eeMOO2 are presented in Figures 6.4a to 6.4h.

In the setting of $|Q| = 4$ and $|\Sigma| = 4$, we observe a faster saturation at a performance value of 1 for eeMOO2. Meaning eeMOO2 needs less additional training data than pMOO. For the remaining and more complex settings, the graphs indicate significant improvements by eeMOO2 over pMOO.

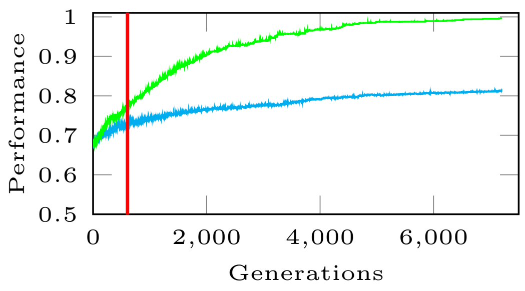
As it can be seen in the heat map in Figure 6.4g, after 100 queries, both algorithms perform about equal for smaller problems, e.g. $|Q| = 3, |\Sigma| = 3$ or $|Q| = 3, |\Sigma| = 4$, which is denoted by the blue colored cells. Next, to the two blue cells, the neighbors turn green and red. While the green cells indicate only a small advantage of eeMOO2 over pMOO, the red cells denote performance improvements of 0.05 up to 0.1. Beyond the red area, on the upper right-hand side of the heat map, the settings are more complex, and therefore more training data is needed to find candidate models of good quality. The profile shown in Figure 6.3g represents an early point in time at which there is no actual difference between the two algorithms.



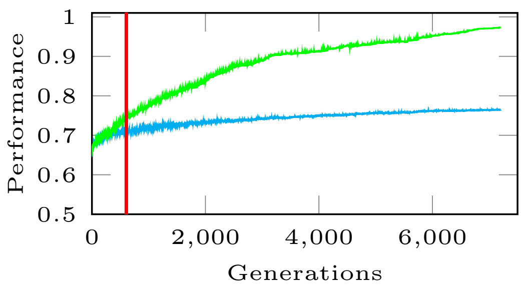
(a) $|Q| = 4$, $|\Sigma| = 4$ ($p = 1$)



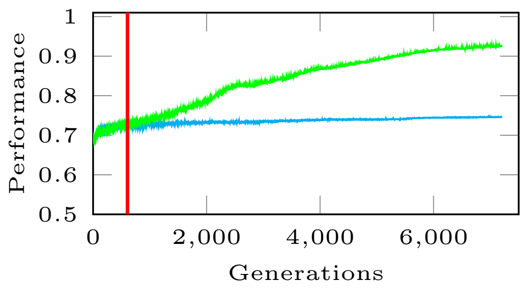
(b) $|Q| = 5$, $|\Sigma| = 5$ ($p < 10^{-4}$)



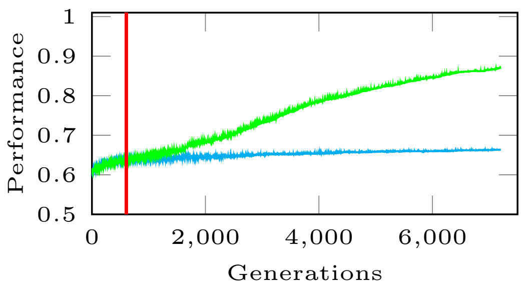
(c) $|Q| = 6$, $|\Sigma| = 6$ ($p < 10^{-10}$)



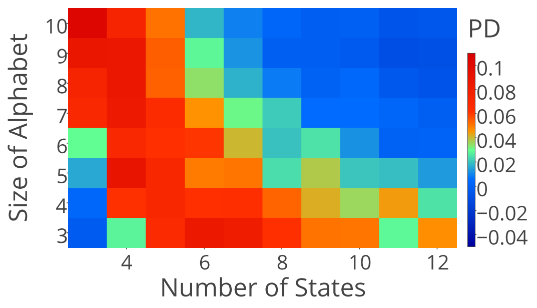
(d) $|Q| = 7$, $|\Sigma| = 7$ ($p < 10^{-7}$)



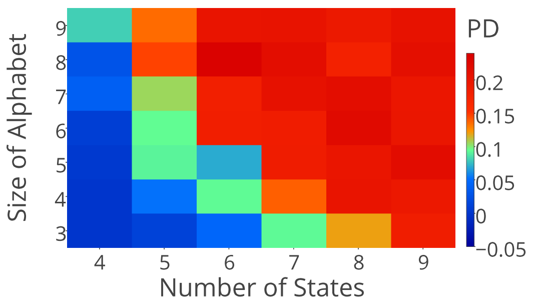
(e) $|Q| = 8$, $|\Sigma| = 8$ ($p < 10^{-8}$)



(f) $|Q| = 9$, $|\Sigma| = 9$ ($p < 10^{-9}$)



(g) Heat map over performance difference eeMOO2−pMOO, 100 queries



(h) Heat map over performance difference eeMOO2−pMOO, 1200 queries

Figure 6.4: Results for the comparison of pMOO and eeMOO2

For instance, this can be seen in Figures 6.4e and 6.4f. In these scenarios, pMOO and eeMOO2 perform about equally for a certain timespan up to approx. 1,500 generations. For longer evolutionary runs, the difference increases over time so that after 1,200 queries, the difference reaches about ≈ 0.18 for the setting $|Q| = 8, |\Sigma| = 8$ and ≈ 0.21 for the other setting.

In Figure 6.4h, the profile of this development can be seen as well. While in the bottom left corner of the heat map, the two algorithms both reach a performance of 1 for less complex scenarios with 1,200 allowed queries, for more complex tasks the difference between eeMOO2 and pMOO increases significantly.

6.2.3 Discussion of Passive VS Active Multi-Objective Optimization

With the first comparison of eeMOO and pMOO, we cannot recognize any noteworthy improvements. For more complex settings. the experiments indicate a small improvement of eeMOO over pMOO. However, the observed effect is still not noteworthy. One of the reasons might be, that the objectives for the evolution of candidate models lead to an inappropriate combination of candidate models in the committee. Most likely, the objective counting the number of sink states is the cause for that, since there always exist individuals consisting of one state only having self-loops for all input symbols. Depending on whether this state is labeled *accepting* or *rejecting*, the objective f_{pos} respectively f_{neg} takes on the maximum fitness of 1 as well. Obviously, the labeling of the state only depends on the proportion of positive and negative training examples. Additionally, such a candidate model has an optimal value for the f_{rel} objective, since only one state is used for processing all the training data. Hence, each Pareto front returned by NSGA-II, which is optimal with respect to the training data containing one such candidate model. This strongly restricts the shape of the remaining Pareto front. In the case of training data, where all training examples are labeled the same, this candidate model dominates all other solutions.

Since the overall population is split into halves, each of the subpopulations is possibly dominated by such a candidate model. In the course of an evolutionary run, we observed in the experiments that this kind of candidate model is found very often. Hence, the committee usually contains two of these candidate models. Therefore, the disagreement of the committee is strongly biased by this type of candidate model. All in all, this leads to a worse candidate test selection. Furthermore, the diversity of the candidate models in the set of solutions returned to the user is decreased. Therefore, f_{sink}, as introduced in [RH16], turns out to be disadvantageous for the evolution of candidate models as well as for the evolution of candidate tests.

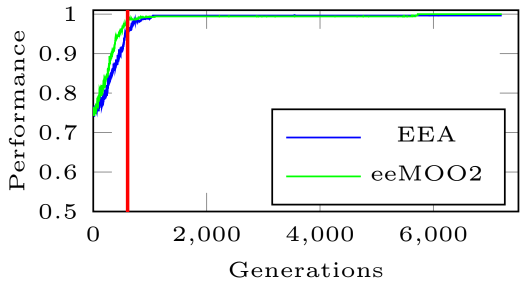
Considering the comparison of eeMOO2 and pMOO, eeMOO2 shows significant improvements. Unlike in the case of eeMOO, the committees evolved by eeMOO2 prove beneficial for the assessment of candidate tests. Hence, the chosen queries for augmenting the training data enhance the quality of the training data significantly. Since eeMOO2 does not make use of the f_{sink} objective, eeMOO2 does not intendedly evolve such trivial candidate models as described above leading to a less useful committee of candidate models. Especially, eeMOO2 shows a steeper learning curve which implies less training examples needed than in pMOO in order to obtain the same performance value for the best candidate models. Specific values concerning the question of how many queries are needed in order to obtain a candidate model with a performance of at least 0.95 are listed in Table 6.3. Moreover, this implies that less information needs to be provided by the user.

6.3 Single Objective VS Multi-Objective

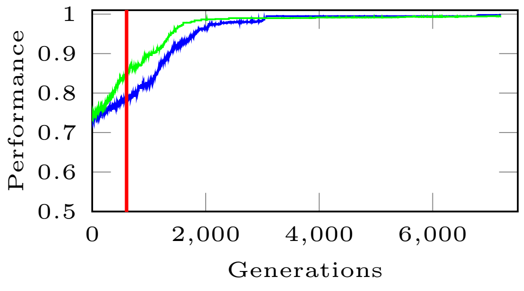
We conclude the evaluation chapter by comparing the Estimation-Exploration Algorithm (EEA) by Bongard and Lipson [BL05] to eeMOO2. The EEA is, to the best of our knowledge, the current state-of-the-art technique in heuristic approaches for grammatical inference. In Section 4.1.4 we made minor modifications in order to enable the algorithm to deal with non-binary alphabets as well. The results presented in the following compare this slightly modified version to eeMOO2.

In [WRH17], we already compared EEA to a similar version of eeMOO2, which is referred to as MOOA. Substituting f_{reach} (cf. Equation 4.13 in Section 4.1.5) by f_{rel} (cf. Equation 4.5 in Section 4.1.2), we started our evaluation for binary alphabets. While for target models of binary alphabets and less than 32 states EEA performed better than MOOA, we observed that MOOA deals better with the 32 states setting. Furthermore, we noted that MOOA outperforms EEA for more complex settings where $|\Sigma| > 3$ and for larger $|Q|$.

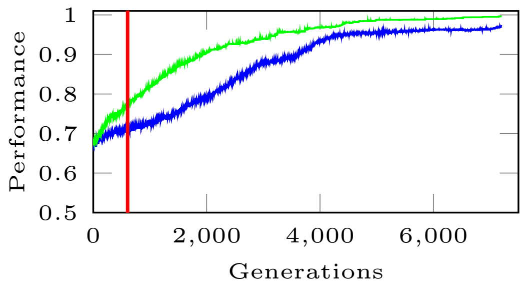
The performance data acquired in the course of this evaluation is presented in detail for some exemplary settings in Figures 6.5a to 6.5f, where the number of states equals the size of the alphabet. A summary of the performance for all the evaluated settings after 100 queries and 1,200 queries is given in the form of heat maps in Figures 6.5g and 6.5h.



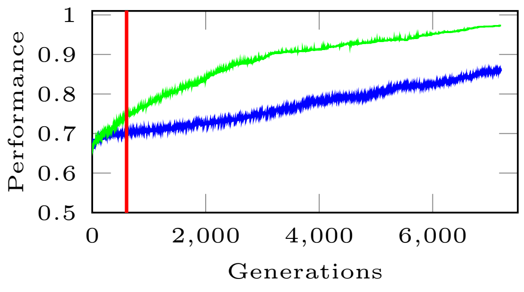
(a) $|Q| = 4$, $|\Sigma| = 4$ ($p = 1$)



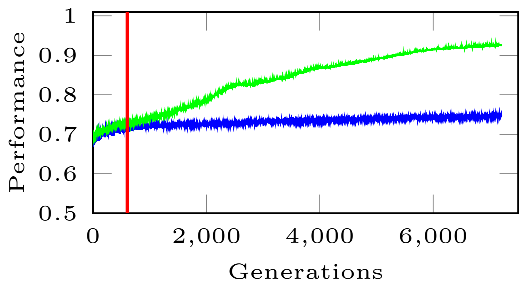
(b) $|Q| = 5$, $|\Sigma| = 5$ ($p \approx 0.4802$)



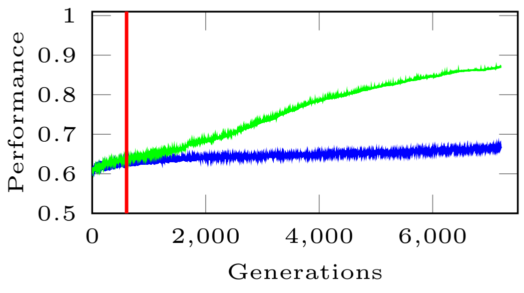
(c) $|Q| = 6$, $|\Sigma| = 6$ ($p \approx 0.2137$)



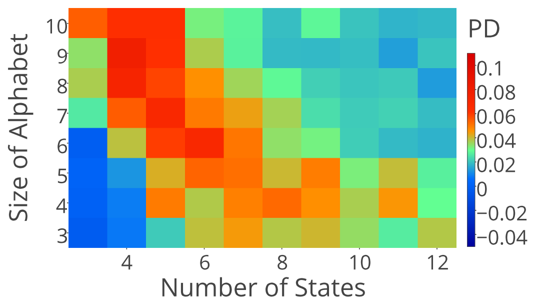
(d) $|Q| = 7$, $|\Sigma| = 7$ ($p \approx 0.0190$)



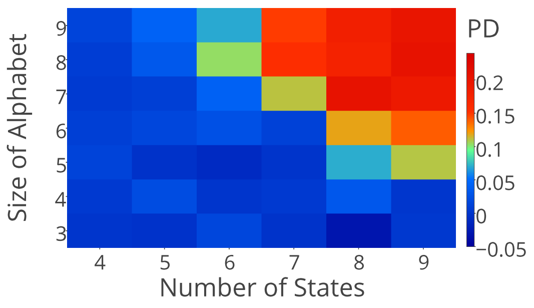
(e) $|Q| = 8$, $|\Sigma| = 8$ ($p < 10^{-8}$)



(f) $|Q| = 9$, $|\Sigma| = 9$ ($p < 10^{-9}$)



(g) Heat map over performance difference eeMOO2 − EEA, 100 queries



(h) Heat map over performance difference eeMOO2 − EEA, 1,200 queries

Figure 6.5: Results for the comparison of EEA and eeMOO2 in different setups

For the problem instances of $|Q| = 4, |\Sigma| = 4$ and $|Q| = 5, |\Sigma| = 5|$, we observe a competitive performance of the two algorithms as already found in the previous comparisons. This result is expected since less complex models also need less training examples and therefore these models can be inferred faster. However, eeMOO2 shows a steeper increase in performance, i.e. it also needs less training examples than EEA in order to reach some fixed performance value. As the number of states and the size of the alphabet increases the difference between the two algorithms becomes more and more visible.

In Figure 6.5c, eeMOO2 first expands its lead over EEA until EEA gets closer again around 4,000 generations. Nevertheless, EEA is not able to catch up completely, so that at the end of the run a small difference in performance still remains.

Figures 6.5d, 6.5e and 6.5f show, how eeMOO2 outperforms EEA in these more complex settings. While the graph representing the performance value of eeMOO2 gives a more purposive impression, the oscillations in the graph of EEA indicate that EEA loses good solutions repeatedly. This implies that EEA is not aware of these good solutions and drops these when the population is re-initialized after training data augmentation.

The heat map showing the profile for all the evaluated settings after 100 queries in Figure 6.5g confirms the observations for the exemplary settings. In the lower left corner, the blue cells denote that the two algorithms perform about equally for a number of states of 3 or 4 and $|Q| + |\Sigma| \leq 9$. For the other settings where $|Q| + |\Sigma| < 15$, a red coloration indicates a performance increase of 0.05 up to 0.1.

In more complex settings, i.e. scenarios in which $|Q| + |\Sigma| \geq 15$, the difference between the performance of eeMOO2 and EEA becomes insignificant. This can be seen for instance in Figures 6.5e and 6.5f, where the red vertical line highlights the point, where 100 queries were used to augment the training data. In these examples, the amount of training data at this point is not sufficient for revealing a real difference between the two algorithms. But, as it can be seen in the further course of the evolutionary run allowing for 1,200 queries, eeMOO2 clearly outperforms EEA. Therefore, we would expect the green cells to turn red for longer evolutionary runs.

Considering the heat map in Figure 6.5h, we can confirm the hypothesis that the performance increases for longer evolutionary runs in the more complex settings. For the settings where $|Q| + |\Sigma| < 15$, EEA catches up so that the performance difference ranges from close to 0 up to 0.1. In the remaining settings, we observe a bigger difference of up to 0.2. Mostly, eeMOO2 shows a steeper learning curve, implying that less training examples are needed.

In summary, eeMOO2 needs less training data than EEA in order to evolve candidate models of the same performance. Since the optimization for multiple objectives needs special treatment e.g. for selection regarding the concepts of Pareto optimality, we use another evolutionary algorithm for the evolution of candidate models and candidate tests. Hence, we do not only compare the usage of single objectives to multiple objectives but different algorithms at the same time. This might lead to the suspicion that the steeper learning curve could be due to implementation details of the algorithms. Actually, in preliminary experiments, the advantage was shown to be originating from using multiple objectives. For that, the exploration phase evolving candidate tests in EEA was extended to use g_{len} as a tie-breaker when two candidate tests in the population obtain the same disagreement value. Meaning if two candidate tests obtain the same disagreement value, the shorter candidate test is preferred over the longer one. On average, the extended version of EEA performed competitively to eeMOO2. However, eeMOO2 naturally allows for an easier extension when adding further objectives. Therefore, with eeMOO2 we chose a strict multi-objective approach over the extension of the EEA.

6.4 Overview and Conclusion

The comparison between EEA, eeMOO, eeMOO2 and pMOO shows, the performance of the algorithms highly to depend on the amount of training data and the heuristic used to explore the search space. Especially, in the case of eeMOO, we can recognize the importance of choosing the right heuristic. When performing multi-objective optimization, it is necessary to use either pair-wise independent objectives or competing objectives, i.e. objectives which cannot be fully satisfied at the same time, in order to maintain a diverse committee of candidate models.

In order to give a conclusive overview of the evaluated algorithms, in Table 6.3, it is listed how many queries are needed by the algorithms for learning the target model allowing an error of 0.05. Therefore, the table lists the number of queries which are needed for achieving a performance of at least 0.95 for all settings evaluated for 1,200 allowed queries. In brackets below, the final performance rounded to two decimal places is noted, i.e. the performance reached after 1,200 oracle queries.

$\|Q\|$	$\|\Sigma\|$	**EEA**	**eeMOO2**	**pMOO**	**eeMOO**
4	3	33 (1)	44 (1)	73 (1)	32 (1)
	4	93 (1)	74 (1)	218 (1)	184 (0.95)
	5	98 (0.99)	85 (1)	225 (1)	137 (0.97)
	6	303 (0.99)	129 (1)	244 (0.99)	200 (0.97)
	7	214 (1)	164 (1)	947 (0.96)	433 (0.97)
	8	311 (0.99)	183 (1)	646 (0.97)	312 (0.97)
	9	361 (0.99)	224 (1)	>1200 (0.92)	1007 (0.94)
5	3	105 (1)	113 (1)	238 (0.98)	>1200 (0.95)
	4	262 (0.98)	187 (1)	898 (0.95)	>1200 (0.92)
	5	300 (1)	230 (0.99)	>1200 (0.9)	>1200 (0.9)
	6	366 (0.98)	313 (1)	>1200 (0.91)	>1200 (0.89)
	7	590 (0.99)	328 (1)	>1200 (0.89)	>1200 (0.87)
	8	966 (0.97)	367 (1)	>1200 (0.85)	>1200 (0.85)
	9	836 (0.96)	459 (1)	>1200 (0.86)	>1200 (0.93)
6	3	149 (0.98)	169 (1)	905 (0.95)	>1200 (0.91)
	4	352 (1)	338 (1)	>1200 (0.91)	>1200 (0.89)
	5	652 (0.99)	398 (0.98)	>1200 (0.91)	>1200 (0.87)
	6	729 (0.97)	533 (1)	>1200 (0.81)	>1200 (0.8)
	7	1135 (0.95)	489 (0.99)	>1200 (0.81)	>1200 (0.85)
	8	>1200 (0.89)	709 (0.99)	>1200 (0.76)	>1200 (0.85)
	9	>1200 (0.92)	771 (0.99)	>1200 (0.78)	>1200 (0.92)
7	3	238 (1)	305 (1)	>1200 (0.9)	>1200 (0.87)
	4	631 (0.99)	444 (0.99)	>1200 (0.85)	>1200 (0.85)
	5	733 (1)	577 (1)	>1200 (0.81)	>1200 (0.82)
	6	1076 (0.96)	770 (0.97)	>1200 (0.78)	>1200 (0.81)
	7	>1200 (0.86)	991 (0.97)	>1200 (0.77)	>1200 (0.81)
	8	>1200 (0.81)	909 (0.97)	>1200 (0.75)	>1200 (0.83)
	9	>1200 (0.81)	996 (0.96)	>1200 (0.76)	>1200 (0.84)
8	3	388 (1)	604 (0.97)	>1200 (0.85)	>1200 (0.82)
	4	1069 (0.96)	603 (0.99)	>1200 (0.79)	>1200 (0.77)
	5	>1200 (0.89)	988 (0.96)	>1200 (0.76)	>1200 (0.75)
	6	>1200 (0.83)	1170 (0.95)	>1200 (0.73)	>1200 (0.76)
	7	>1200 (0.72)	>1200 (0.92)	>1200 (0.71)	>1200 (0.74)
	8	>1200 (0.75)	>1200 (0.93)	>1200 (0.75)	>1200 (0.78)
	9	>1200 (0.72)	>1200 (0.9)	>1200 (0.7)	>1200 (0.75)
9	3	778 (0.98)	657 (0.98)	>1200 (0.8)	>1200 (0.77)
	4	1171 (0.95)	1094 (0.95)	>1200 (0.76)	>1200 (0.76)

$\vert Q\vert$	$\vert\Sigma\vert$	**EEA**	**eeMOO2**	**pMOO**	**eeMOO**
9	5	>1200 (0.82)	>1200 (0.93)	>1200 (0.71)	>1200 (0.71)
	6	>1200 (0.79)	>1200 (0.93)	>1200 (0.73)	>1200 (0.76)
	7	>1200 (0.72)	>1200 (0.92)	>1200 (0.72)	>1200 (0.74)
	8	>1200 (0.7)	>1200 (0.91)	>1200 (0.7)	>1200 (0.74)
	9	>1200 (0.67)	>1200 (0.87)	>1200 (0.66)	>1200 (0.71)

Table 6.3: Overview of how many queries are needed to reach a performance of at least 0.95. The final performance values are disclosed in brackets. >1200 means that the algorithm does not reach a performance of at least 0.95 in this setting. (Performance values are rounded to two decimal places)

As it can be seen in Table 6.3, eeMOO2 clearly outperforms pMOO. On the one hand, this is due to using different objectives, and on the other hand, this is due to strategically augmenting the training data by evolved candidate tests. While pMOO reaches a performance of at least 0.95 only in 9 out of the 42 settings, eeMOO2 does not reach a performance of 0.95 in 8 settings. Moreover, in the 9 settings for which pMOO reaches a performance of at least 0.95, eeMOO2 needs only a fraction of the oracle queries compared to pMOO.

Except for settings in which $|\Sigma| = 3$, the table furthermore illustrates the advantage of eeMOO2 over EEA for more complex settings. Compared to eeMOO2, EEA does not reach a performance of at least 0.95 for 7 other settings. Consulting the table, it can be recognized that the advantage of eeMOO2 over EEA increases for an increasing number of states and size of the alphabet. This is indicated by reaching an accuracy of 0.95 earlier and a higher value for the performance value in settings, where an accuracy of 0.95 is not reached. By showing significant improvements in these settings, eeMOO2 outperforms the current state-of-the-art algorithm for grammatical inference. In particular, eeMOO2 needs less training data in order to reach the same performance value as EEA.

However, for our setting, the performance of the algorithms is still insufficient and the amount of training data, which needs to be provided by the user, is still infeasible. Even if the user only has to classify sequence diagrams either as desired or prohibited behavior instead of creating these diagrams on his own, the sheer number of queries would fatigue, demotivate or simply overtax the user. Nonetheless, optimizing for multiple objectives in order to get a more sophisticated view for the assessment of candidate models is a step in the right direction. It is inevitable to reduce further the amount of training data, which needs to be provided by the user in some way or other.

One possible solution might be to use the domain knowledge inherent in the OTF market and to learn from observing requirements specifications in order to reduce the amount of training data, which needs to be provided by the user. In Section 4.2, we already proposed some concepts to address the problem concerning how to integrate this knowledge into the automated requirements engineering process.

7 Conclusion

This final chapter is devoted to conclusions, insights, remaining work and visions. In Section 7.1 we recap the interactive multi-objective optimization approach for learning requirements specifications from examples. The results of the evaluation in Chapter 6 led to relevant insights summarized in Section 7.2.

7.1 Summary

In the context of OTF markets, we extended the requirements specification-by-example approach by van Rooijen and Hamann [RH16] to an automatic requirements engineer. This was done by transferring the concepts of Bongard and Lipson [BL05] to an interactive co-evolutionary approach strictly using multi-objective optimization for evolving both candidate models and candidate tests. A special challenge in this context is sparse data. In contrast to traditional approaches for grammatical inference, we can only request a few training examples, since all these training examples have to be provided by the user. Using multiple objectives helps the algorithm to discriminate superior and inferior candidate models better. This, in turn, helps to generate more useful queries and request the right examples from the user. Furthermore, the user is supported in providing more relevant training examples, so that the algorithm takes over the role of a requirements engineer. In this way, the algorithm also supports the user in refining imprecise or vague notions of the desired service to a precise formal requirements specification.

However, the automatic requirements engineer currently only uses the data provided by the user. Therefore, we proposed new concepts on how to relieve the user and make use of the knowledge, concerning the semantics of operations and services available in the OTF market. Another concept for extending the formalism to describe protocols of services was proposed. By abstracting predicates used to describe operation interfaces in the OTF market to guards, it is possible to introduce more complex structures such as branching structures to the protocol generalized from the user's training examples.

For the comparison and benchmarking of different requirements elicitation algorithms, the REAL framework was introduced. Due to its architecture and clear distinction between roles, which are involved in the actual requirements elicitation process, and entities, which are solely needed for benchmarking purposes, the framework can be integrated into other application and can be used as a standalone in order to test and benchmark different algorithms. Furthermore, since the roles and entities are highly decoupled, the framework is easily extensible and adaptable.

Exceeding the objectives of this thesis, we integrated the framework into a client-server application in order to distribute the load of conducting the evaluation experiments on up to 100 nodes. For the distribution, a central server was dedicated to assigning chunks to the nodes and stored the results of the experiments in a database. This enabled us to base the evaluation on a number of experiments, which would have taken 286 days of computation on a single computer.

7.2 Insights

The evaluation showed that in the scenario of very few training examples, multiple objectives can guide the algorithm to discriminate better between candidate models and candidate tests of higher or lower quality. Being more aware of how good candidate models are, the active learning process inferring the target model is accelerated. However, objectives need to be carefully chosen as demonstrated by the example of the objective maximizing the number of sink states (f_{sink}).

Furthermore, by extending the requirements specification-by-example approach by van Rooijen and Hamann [RH16] to an active learning approach, we could improve the performance of the requirements elicitation algorithm significantly. In particular, the active learning approach helps to ask the user the right questions and, thereby, reduces the number of training examples needed for inferring the target model. Although allowing an error of 5% for the test set performance, the number of training examples, which would have to be provided by the user, is still infeasible and would fatigue and demotivate the user.

Incidentally, the evaluation comparing our interactive multi-objective optimization approach to the state-of-the-art technique proposed by Bongard and Lipson [BL05] yielded gratifying results. In settings with bigger alphabets our approach, especially for bigger automata, outperforms the state-of-the-art approach ([BL05]) clearly. However, the experiment setup had a limitation, since we did not only evaluate the effect of using multiple objectives compared to a single objective but also compared two different evolutionary algorithms at the same time. Therefore, we cannot ascribe the significant improvements to the usage of multiple objectives alone. Nevertheless, these improvements are important in our setting, since we can only require a limited amount of data from the user. Hence, there is a clear advantage of our approach over the approach by Bongard and Lipson [BL05].

For all approaches, we observed oscillations in several settings. Especially, for more complex problem instances at the beginning, the algorithms lose better performing models repeatedly, since the algorithms are not aware of the real performance. We also observed that the usage of multiple objectives reduces this effect.

7.3 Future Work

The most relevant task for future work is to decrease further the amount of training data required to be provided by the user. To this end, more sophisticated heuristics are necessary, and additional knowledge has to be acquired from other knowledge sources. Finding additional objectives for assessing the fitness of candidate models and candidate tests, in order to let the algorithm be more aware of the quality of candidates, is one way of addressing this problem. For instance, objectives optimizing for common structures within protocols such as branches, loops, prefix or suffix might be a useful extension.

Moreover, in Section 4.2.2, we introduced two concepts of how to use knowledge about the services available in the OTF market to augment the alphabet and to introduce branching structures in the protocols. Due to the immaturity of these concepts, further work of research is needed to implement these concepts. Especially, the black boxes “Operation Mapper” and “Recommendation System” need to be designed and implemented.

In Section 4.2.1 heuristics for bundling multiple queries with the aim to decrease network traffic and to reduce the waiting time for both user and algorithm were proposed. It has to be evaluated whether bundling has a negative impact on the number of training examples needed in total. However, as outlined in Section 4.2.1, we would expect that more training examples are needed compared to sending queries individually.

Another future task is to improve the generation routine to generate more realistic target models. The generation routine used in the evaluation chapter simply draws destination states uniformly at random for all transitions. Hence, the generated target models have no specific structure and semantics of operation names, i.e. symbols of the alphabet, are not taken into account. In order to involve suppliers into the benchmark, the target model generation could be combined with a service repository generator, as proposed in [Moh16].

Appendix A

Software-Based Experiment Conduction Cluster

In Figures A.1a to A.1h screenshots give an impression of the software-based experiment conduction cluster (SBECC) used to distribute the load of conducting all the experiments to up to 100 nodes.

System requirements

SBECC needs at least a single core CPU with 1GHz and 2GB RAM. Furthermore, the following software is required to be installed before:

- Apache 2.4 with mod_rewrite
- PHP 7
- MySQL 5.7

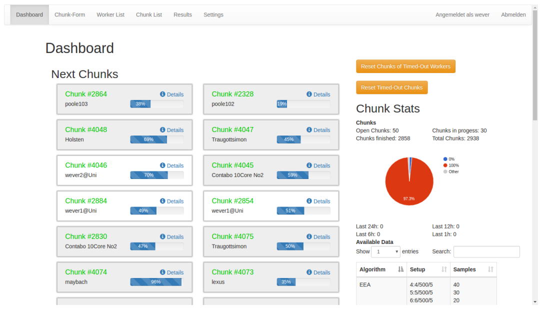
In order to get SBECC running as expected the configuration files of php and mysql have to be adapted. It is recommended to increase script running times for php, allowed file sizes, and the allowed memory sizes.

Setup

Copy the file from the `SBECCServer.zip` to the document root of the server and load a MySQL dump into the database. Update the ./config.php file with the correct credentials for accessing the MySQL database.

Notes

The more active nodes work on chunks and possibly send experiment results to the central server, the more resources should be allocated for the central server.



Dashboard

Next Chunks

Chunk	Worker	Progress
Chunk #2864	poole103	38%
Chunk #2328	poole102	19%
Chunk #4048	Holsten	69%
Chunk #4047	Traugottsimon	45%
Chunk #4046	wever2@Uni	70%
Chunk #4045	Contabo 10Core No2	59%
Chunk #2884	wever1@Uni	49%
Chunk #2854	wever1@Uni	51%
Chunk #2830	Contabo 10Core No2	47%
Chunk #4075	Traugottsimon	50%
Chunk #4074	maybach	96%
Chunk #4073	lexus	35%

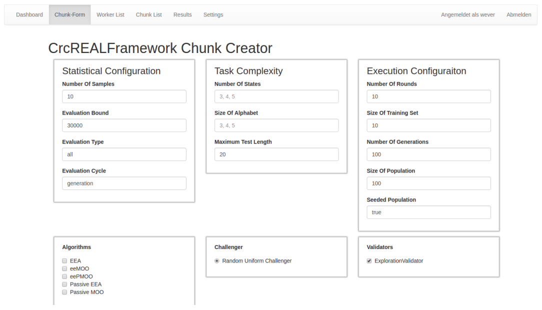
Chunk Stats

Chunks
Open Chunks: 50 Chunks in progess: 30
Chunks finished: 2858 Total Chunks: 2938

Last 24h: 0 Last 12h: 0
Last 6h: 0 Last 1h: 0
Available Data
Show 1 entries Search:

Algorithm	Setup	Samples
EEA	4:4/500/5 5:5/500/5 6:6/500/5	40 30 20

(a) SBECC dashboard



CrcREALFramework Chunk Creator

Statistical Configuration
Number Of Samples: 10
Evaluation Bound: 30000
Evaluation Type: all
Evaluation Cycle: generation

Task Complexity
Number Of States: 3, 4, 5
Size Of Alphabet: 3, 4, 5
Maximum Test Length: 20

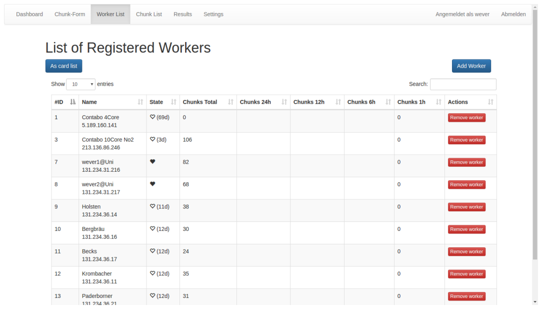
Execution Configuraiton
Number Of Rounds: 10
Size Of Training Set: 10
Number Of Generations: 100
Size Of Population: 100
Seeded Population: true

Algorithms
☐ EEA
☐ eeMOO
☐ eePMOO
☐ Passive EEA
☐ Passive MOO

Challenger
● Random Uniform Challenger

Validators
☑ ExplorationValidator

(b) SBECC chunk configuration

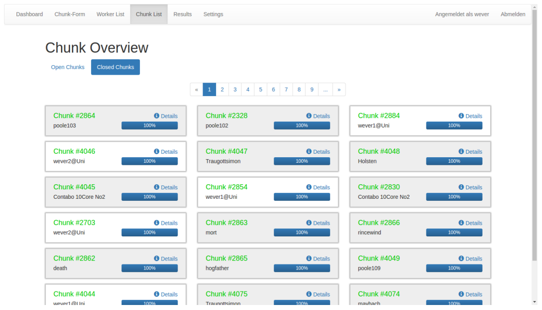


List of Registered Workers

Show 10 entries Search:

#ID	Name	State	Chunks Total	Chunks 24h	Chunks 12h	Chunks 6h	Chunks 1h	Actions
1	Contabo 4Core 5.189.160.141	♡ (69d)	0				0	Remove worker
3	Contabo 10Core No2 213.136.86.246	♡ (3d)	106				0	Remove worker
7	wever1@Uni 131.234.31.216	♥	82				0	Remove worker
8	wever2@Uni 131.234.31.217	♥	68				0	Remove worker
9	Holsten 131.234.36.14	♡ (11d)	38				0	Remove worker
10	Bergbräu 131.234.36.16	♡ (12d)	30				0	Remove worker
11	Becks 131.234.36.17	♡ (12d)	24				0	Remove worker
12	Krombacher 131.234.36.11	♡ (12d)	35				0	Remove worker
13	Paderborner 131.234.36.21	♡ (12d)	31				0	Remove worker

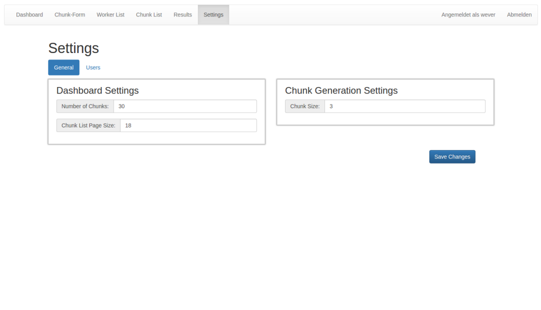
(c) SBECC list of worker clients



Chunk Overview

Chunk	Worker	Progress
Chunk #2864	poole103	100%
Chunk #2328	poole102	100%
Chunk #2884	wever1@Uni	100%
Chunk #4046	wever2@Uni	100%
Chunk #4047	Traugottsimon	100%
Chunk #4048	Holsten	100%
Chunk #4045	Contabo 10Core No2	100%
Chunk #2854	wever1@Uni	100%
Chunk #2830	Contabo 10Core No2	100%
Chunk #2703	wever2@Uni	100%
Chunk #2863	mort	100%
Chunk #2866	rincewind	100%
Chunk #2862	death	100%
Chunk #2865	hogfather	100%
Chunk #4049	poole109	100%
Chunk #4044	wever1@Uni	100%
Chunk #4075	Traugottsimon	100%
Chunk #4074	maybach	100%

(d) SBECC list of all chunks

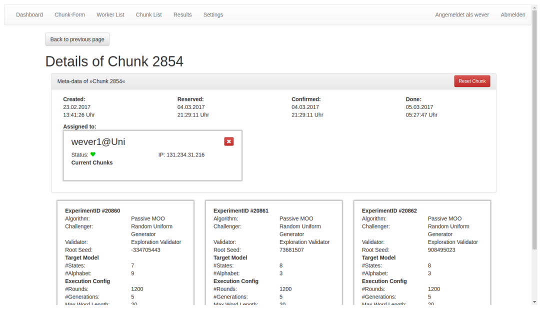


Settings

Dashboard Settings
Number of Chunks: 30
Chunk List Page Size: 18

Chunk Generation Settings
Chunk Size: 3

(e) SBECC general settings



Details of Chunk 2854

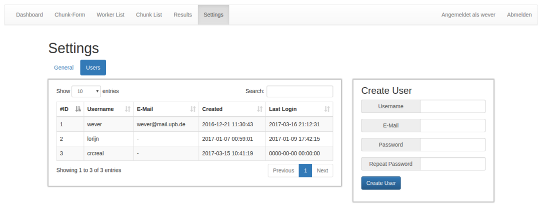
Meta-data of »Chunk 2854«

Created: 23.02.2017 13:41:26 Uhr
Reserved: 04.03.2017 21:29:11 Uhr
Confirmed: 04.03.2017 21:29:11 Uhr
Done: 05.03.2017 05:27:47 Uhr

Assigned to:
wever1@Uni
Status: ♥ IP: 131.234.31.216
Current Chunks

	ExperimentID #20860	ExperimentID #20861	ExperimentID #20862
Algorithm:	Passive MOO	Passive MOO	Passive MOO
Challenger:	Random Uniform Generator	Random Uniform Generator	Random Uniform Generator
Validator:	Exploration Validator	Exploration Validator	Exploration Validator
Root Seed:	-334705443	73681507	908495023
Target Model			
#States:	7	8	8
#Alphabet:	9	3	3
Execution Config			
#Rounds:	1200	1200	1200
#Generations:	5	5	5
Max Word Length:	20	20	20

(f) SBECC chunk details



Settings

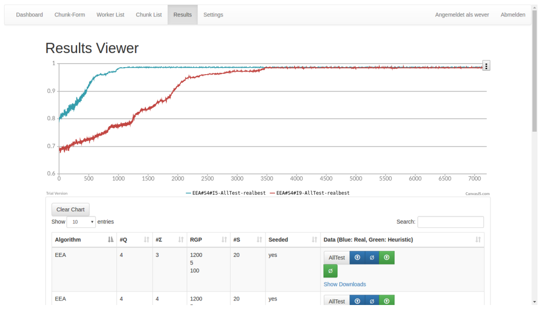
Show 10 entries Search:

#ID	Username	E-Mail	Created	Last Login
1	wever	wever@mail.upb.de	2016-12-21 11:30:43	2017-03-16 21:12:31
2	lorijn	-	2017-01-07 00:59:01	2017-01-09 17:42:15
3	crcreal	-	2017-03-15 10:41:19	0000-00-00 00:00:00

Showing 1 to 3 of 3 entries

Create User
Username
E-Mail
Password
Repeat Password

(g) SBECC user settings



Results Viewer

Show 10 entries Search:

Algorithm	#Q	#Σ	RGP	#S	Seeded	Data (Blue: Real, Green: Heuristic)
EEA	4	3	1200 5 100	20	yes	AllTest
EEA	4	4	1200	20	yes	AllTest

(h) SBECC results presentation

Figure A.1: Graphical User Interface of the Software-Based Experiment Conduction Cluster (SBECC) server

Appendix B

CD Contents

- PDF of this thesis
- Implementation of the REAL framework
- Implementation of a Java client for distributed evaluation
- Implementation of a PHP server for defining and assigning evaluation tasks to clients
- Initial MySQL dump
- MySQL dump containing all the measured data of the evaluated experiments
- Executable JARs of CRCREALExecutor and CrcREALWorkerClient
- Supplementary results material: Box plots over the samples for each setting and algorithm after 100 queries

The software, which was implemented as part of this thesis, can be divided in two major parts: a server in PHP for the administration of distributed experiment conduction and a Java implementation of the framework including a client in order to pull tasks from the PHP server. The Java implementation is separated into the following projects:

CrcREALFramework contains the framework as presented in Chapter 4

CrcREALUtils contains utility classes required by the framework, executor and client

CrcREALExecutor contains a standalone instantiation of the framework, which can be executed via the command line

CrcREALWorkerClient contains a simple client, which pulls chunks from a server which is specified by a URL

CrcREALWebInterpolator contains a simple client, which pulls chunks from a server which is specified by a URL

Concerning the Software-Based Experiment Conduction Cluster, two dumps for the database are provided: one dump containing all the calculated experiment results presented in Chapter 6 and another dump without any data. In the latter, a standard user is already registered with the username `crcreal` and the password `crcreal`. Worker clients have to be registered before they can pull chunks from the central server and send back evaluation results to the central server.

Furthermore, the Java projects require additional projects from the CRC901-B-Tools repository. These projects are located at the following URLs:

- https://svn-serv.cs.upb.de/CRC901-B-Tools/trunk/projects/configmate/gs-core-modded/
- https://svn-serv.cs.upb.de/CRC901-B-Tools/trunk/projects/sequenceDiagramEditor/MOEAFramework/

(Accessed 2017-03-17)

Executing Worker Client

The `CrcREALWorkerClient.jar` is executed and started directly by the following command:

```
java -jar CrcREALWorkerClient.jar run
```

The CrcREALWorkerClient implements a simple CLI that can be used with the following commands.

run Start a worker thread pulling chunks from the central server and sending back the results of the experiment

stop [-f] Stops the worker thread after the currently processed chunk is finished. Adding the parameter `-f` forces the worker thread to terminate directly.

exit stops the worker thread after the currently processed chunk is finished and quits the worker client completely.

Bibliography

[BGS05] Sven Burmester, Holger Giese, and Wilhelm Schäfer. "Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code." In: *Model Driven Architecture - Foundations and Applications, First European Conference, ECMDA-FA 2005, Nuremberg, Germany, November 7-10, 2005, Proceedings.* 2005, pp. 25–40. DOI: 10.1007/11581741_4. URL: http://dx.doi.org/10.1007/11581741_4.

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf H. Reussner. "The Palladio component model for model-driven performance prediction." In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22. DOI: 10.1016/j.jss.2008.03.066. URL: http://dx.doi.org/10.1016/j.jss.2008.03.066.

[BL05] Josh C. Bongard and Hod Lipson. "Active Coevolutionary Learning of Deterministic Finite Automata." In: *Journal of Machine Learning Research* 6 (2005), pp. 1651–1678. URL: http://www.jmlr.org/papers/v6/bongard05a.html.

[Col11] Collaborative Research Center 901 (CRC 901). *On-The-Fly Computing.* 2011. URL: http://sfb901.uni-paderborn.de/sfb-901/home.html (visited on 03/11/2017).

[Dar] Charles Darwin. *On the origin of species.* New York :D. Appleton and Co., p. 470. DOI: 110.5962/bhl.title.28875. URL: http://www.biodiversitylibrary.org/item/71804.

[Deb+02] Kalyanmoy Deb et al. "A fast and elitist multiobjective genetic algorithm: NSGA-II." In: *IEEE Trans. Evolutionary Computation* 6.2 (2002), pp. 182–197. DOI: 10.1109/4235.996017. URL: http://dx.doi.org/10.1109/4235.996017.

[DFG11] Arianna D'Ulizia, Fernando Ferri, and Patrizia Grifoni. "A survey of grammatical inference methods for natural language learning." In: *Artif. Intell. Rev.* 36.1 (2011), pp. 1–27. DOI: 10.1007/s10462-010-9199-1. URL: http://dx.doi.org/10.1007/s10462-010-9199-1.

[Fer+00] Francesc J. Ferri et al., eds. *Advances in Pattern Recognition, Joint IAPR International Workshops SSPR 2000 and SPR 2000, [8th International Workshop on Structural and Syntactic Pattern Recognition, 3rd International Workshop on Statistical Techniques in Pattern Recognition], Alicante, Spain, August 30 - September 1, 2000, Proceedings.* Vol. 1876. Lecture Notes in Computer Science. Springer, 2000. ISBN: 3-540-67946-4.

[Gil+92] C. Lee Giles et al. "Learning and Extracting Finite State Automata with Second-Order Recurrent Neural Networks." In: *Neural Computation* 4.3 (1992), pp. 393–405. DOI: `10.1162/neco.1992.4.3.393`. URL: `http://dx.doi.org/10.1162/neco.1992.4.3.393`.

[Gom06] Jonatan Gomez. "An Incremental-Evolutionary Approach for Learning Deterministic Finite Automata." In: (2006), pp. 362–369. DOI: `10.1109/CEC.2006.1688331`. URL: `http://dx.doi.org/10.1109/CEC.2006.1688331`.

[GS94] David Garlan and Mary Shaw. *An Introduction to Software Architecture.* Tech. rep. CMU-CS-94-166. Carnegie Mellon University, Jan. 1994.

[Har87] David Harel. "Statecharts: A Visual Formalism for Complex Systems." In: *Sci. Comput. Program.* 8.3 (1987), pp. 231–274. DOI: `10.1016/0167-6423(87)90035-9`. URL: `http://dx.doi.org/10.1016/0167-6423(87)90035-9`.

[HKP05] David Harel, Hillel Kugler, and Amir Pnueli. "Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements." In: *Formal Methods in Software and Systems Modeling, Essays Dedicated to Hartmut Ehrig, on the Occasion of His 60th Birthday.* 2005, pp. 309–324. DOI: `10.1007/978-3-540-31847-7_18`. URL: `http://dx.doi.org/10.1007/978-3-540-31847-7_18`.

[HM79] Ching-Lai Hwang and Abu Syed Md Masud. *Multiple objective decision making, methods and applications: a state-of-the-art survey.* 164th ed. Springer-Verlag Berlin Heidelberg, 1979. ISBN: 978-3-642-45511-7. DOI: `10.1007/978-3-642-45511-7`.

[HMZ12] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. "Search-based software engineering: Trends, techniques and applications." In: *ACM Comput. Surv.* 45.1 (2012), 11:1–11:61. DOI: `10.1145/2379776.2379787`. URL: `http://doi.acm.org/10.1145/2379776.2379787`.

[Kap+12] Gerti Kappel et al. "Model Transformation By-Example: A Survey of the First Wave." In: *Conceptual Modelling and Its Theoretical Foundations - Essays Dedicated to Bernhard Thalheim on the Occasion of His 60th Birthday.* 2012, pp. 197–215. DOI: 10.1007/978-3-642-28279-9_15. URL: http://dx.doi.org/10.1007/978-3-642-28279-9_15.

[Kes+12] Marouane Kessentini et al. "Search-based model transformation by example." In: *Software and System Modeling* 11.2 (2012), pp. 209–226. DOI: 10.1007/s10270-010-0175-7. URL: http://dx.doi.org/10.1007/s10270-010-0175-7.

[Küh+16] Thomas Kühne et al. "Patterns for Constructing Mutation Operators: Limiting the Search Space in a Software Engineering Application." In: *Genetic Programming - 19th European Conference, EuroGP 2016, Porto, Portugal, March 30 - April 1, 2016, Proceedings.* 2016, pp. 278–293. DOI: 10.1007/978-3-319-30668-1_18. URL: http://dx.doi.org/10.1007/978-3-319-30668-1_18.

[LP98] Kevin J. Lang and Barak A. Pearlmutter. *Abbadingo.* 1998. URL: http://abbadingo.cs.nuim.ie/ (visited on 03/11/2017).

[LPP98] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. "Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm." In: *Grammatical Inference, 4th International Colloquium, ICGI-98, Ames, Iowa, USA, July 12-14, 1998, Proceedings.* 1998, pp. 1–12. DOI: 10.1007/BFb0054059. URL: http://dx.doi.org/10.1007/BFb0054059.

[LR03] Simon M. Lucas and T. Jeff Reynolds. "Learning DFA: evolution versus evidence driven state merging." In: *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2003, 8 - 12 December 2003, Canberra, Australia.* 2003, pp. 351–358. DOI: 10.1109/CEC.2003.1299597. URL: http://dx.doi.org/10.1109/CEC.2003.1299597.

[LR05] Simon M. Lucas and T. Jeff Reynolds. "Learning Deterministic Finite Automata with a Smart State Labeling Evolutionary Algorithm." In: *IEEE Trans. Pattern Anal. Mach. Intell.* 27.7 (2005), pp. 1063–1074. DOI: 10.1109/TPAMI.2005.143. URL: http://dx.doi.org/10.1109/TPAMI.2005.143.

[Moh16] Felix Mohr. "Towards Automated Service Composition Under Quality Constraints." PhD thesis. University of Paderborn, Germany, 2016.

[MW47] H. B. Mann and D. R. Whitney. “On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other.” In: *The Annals of Mathematical Statistics* 18.1 (1947), pp. 50–60. DOI: 10.1214/aoms/1177730491.

[Obj15] Object Management Group (OMG). *Unified Modeling Language TM.* OMG Document Number formal/2006-01-01 (http://www.omg.org/spec/UML/2.5). 2015.

[Pla+16] Marie Christin Platenius et al. *An Overview of Service Specification Language and Matching in On-The-Fly Computing (v0.3).* Tech. rep. tr-ri-16-349. Software Engineering Group, Heinz Nixdorf Institute, University of Paderborn, Jan. 2016.

[Pla16] Marie Christin Platenius. “Fuzzy Matching of Comprehensive Service Specifications.” PhD thesis. University of Paderborn, Germany, 2016. URL: http://nbn-resolving.de/urn:nbn:de:hbz:466:2-26843.

[PS04] Georgios Paliouras and Yasubumi Sakakibara, eds. *Grammatical Inference: Algorithms and Applications, 7th International Colloquium, ICGI 2004, Athens, Greece, October 11-13, 2004, Proceedings.* Vol. 3264. Lecture Notes in Computer Science. Springer, 2004. ISBN: 3-540-23410-1.

[RH16] Lorijn van Rooijen and Heiko Hamann. “Requirements Specification-by-Example Using a Multi-objective Evolutionary Algorithm.” In: *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016.* 2016, pp. 3–9. DOI: 10.1109/REW.2016.015. URL: http://dx.doi.org/10.1109/REW.2016.015.

[SCZ04] Brad Starkie, Francois Coste, and Menno van Zaanen. *The Omphalos Competition.* 2004. URL: http://www.irisa.fr/Omphalos/ (visited on 03/11/2017).

[Set12] Burr Settles. “Active Learning.” In: Synthesis Lectures on Artificial Intelligence and Machine Learning (2012). URL: http://dx.doi.org/10.2200/S00429ED1V01Y201207AIM018.

[SLV14] Daniel Schütz, Christoph Legat, and Birgit Vogel-Heuser. “MDE of manufacturing automation software - Integrating SysML and standard development tools.” In: *12th IEEE International Conference on Industrial Informatics, INDIN 2014, Porto Alegre, RS, Brazil, July 27-30, 2014.* 2014, pp. 267–273. DOI: 10.1109/INDIN.2014.6945519. URL: http://dx.doi.org/10.1109/INDIN.2014.6945519.

[WRH17] Marcel D. Wever, Lorijn van Rooijen, and Heiko Hamann. “Active Coevolutionary Learning of Requirements Specifications from Examples.” In: SUBMITTED to Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15 - 19, 2017. 2017.

[YG12] Xinjie Yu and Mitsuo Gen. *Introduction to Evolutionary Algorithms.* 1st ed. Springer Publishing Company, Incorporated, 2012. ISBN: 144712569X, 9781447125693. DOI: 10.1007/978-1-84996-129-5.