# Requirements Specification-by-Example Using a Multi-Objective Evolutionary Algorithm

Lorijn van Rooijen and Heiko Hamann, Paderborn University

***Abstract*—A task at the beginning of the software development process is the creation of a requirements specification. The requirements specification is usually created by a software engineering expert. We try to substitute this expert by a domain expert (user) and formulate the problem of creating requirements specifications as a search-based software engineering problem. The domain expert provides only examples of event sequences that describe the behavior of the required software program. These examples are represented by simple sequence diagrams and are divided into two subsets: positive examples of required program behavior and negative examples of prohibited program behavior. The task is then to synthesize a generalized requirements specification that usefully describes the required software. We approach this problem by applying a genetic algorithm and evolve deterministic finite automata (DFAs). These DFAs take the sequence diagrams as input (input words) that should be either accepted (positive example) or rejected (negative example). The problem is neither to find the minimal nor the most general automaton. Instead, the user should be provided with several appropriate automata from which the user can select, or which help the user to refine the examples given initially. We present the context of our research ("On-The-Fly Computing"), present our approach, report results indicating its feasibility, and conclude with a discussion.**

## I. Introduction

An important part of software development is the specification of requirements. Usually, a software engineer in close collaboration with a domain expert (i.e., a person with knowledge of the specific context and business processes) would develop requirements specifications. In this paper we consider the vision of a computer-aided development of requirements specifications, which enables the domain expert to solve the task independently.

Different motivations for semi-automatizing the process of developing requirements specifications exist. Besides the mere potential for cost reduction, enabling a domain expert to complete this task independently also allows for an increased flexibility and a more rapid process. Furthermore, our approach can also be valuable for software engineers involved in the process of requirements specification for complex software. Our approach allows to construct requirements specifications semi-automatically from input examples. This may decrease the workload for software engineers that otherwise would have to formalize the specifications manually – a potentially costly and error-prone process. Moreover, a software engineer could later reuse the input examples he gave, as test cases for the constructed software.

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

The context of our research provides additional motivation. Within the Collaborative Research Center "On-The-Fly Computing" (CRC 901)[1], we investigate dynamic, online software markets (also called OTF markets) with maximal flexibility. For example, users are allowed to search for software services which possibly do not yet exist, but can be tailor-made by automatically composing appropriate existing services. In such a dynamic software market, it is essential that the requirements specification of services is not only flexible, rapid, and independent of support by experts, but also user-friendly. Our approach will be fully embedded into an existing tool-chain of software tools called SeSAME[2]. This tool-chain was developed to provide support for trading software services on an OTF market and consists of modules for comprehensive requirements specification, model transformation, searching services, composition, analysis and verification of services [AWBP14].

In the scenario of OTF markets, we distinguish two types of users: naive users and domain experts. Naive users prefer to specify their requirements in natural language. The specific challenges of dealing with such, often ambiguous and incomplete, requirements specifications in the context of an OTF market are, for example, described in [GB16]. In the following, we consider domain experts who have knowledge about the context of the requested software. They are capable of providing examples of desired and undesired behaviors of the requested software. However, we neither expect them to have a complete understanding of that software nor to be capable of independently constructing state machines that specify the software.

We address the problem of making requirements specification more accessible and user-friendly for domain experts. We take a by-example approach, that is, a user specifies positive examples of desired program behavior and negative examples of forbidden program behavior. Our task is then to synthesize a generalized requirements specification that usefully describes the required software. This specification-by-example approach is inspired by ideas from the field of DFA learning and promising developments from the field of Search-Based Software Engineering (see also Sec. II-A and II-D).

The UML formalism of sequence diagrams[3] is particularly well-suited to describe such example behaviors of systems. At this early stage of our work, we use simplified sequence

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

[2] The tool SeSAME is available under https://sfb901.uni-paderborn.de/sfb-901/projects/tools-demonstration-systems/sesame.html

[3] see for example http://www.omg.org/spec/UML/2.5/PDF/

diagrams, which only describe interactions between a user and the system. We do not allow any more sophisticated features of sequence diagrams, such as loops, alternative behaviors, time constraints, modal operators, or different kinds of messages. Let us define simplified sequence diagrams.

*Definition 1:* A (simplified version of a) *sequence diagram* consists of two parallel vertical lines, corresponding to two objects: the user and the system. Horizontal arrows can be drawn between these so-called *lifelines*, which represent messages (also called operation calls or events) issued by the object corresponding to the lifeline from which the arrow is leaving. Such an arrow is labeled with the name of the message. The lifelines represent the flow of time, from top to bottom: a message is issued later than another message iff its arrow is below the arrow corresponding to the other message.

The motivation for this simplified version of sequence diagrams is twofold. Firstly, our prospective user does not have to be an expert in specifying sequence diagrams: it is our goal to make requirements specifications accessible to non-expert users. Keeping the formalism as basic as possible contributes to this goal. Secondly, restricting sequence diagrams to two lifelines with one kind of messages that are exchanged in a consecutive, non-branching, manner, allows us to view sequence diagrams as finite words. For a given set of input sequence diagrams, we view the set of all operations used in these diagrams as the alphabet, and the diagrams themselves as finite words over this alphabet. Note that the set of operations issued by the user is normally disjoint from the set of operations issued by the system. This means that information about the direction of the arrows is implicitly present in the operations and does not have to be added to the finite words. From a conceptual point of view, it is much easier to deal with these structures than with sequence diagrams. It is advantageous to handle sequence diagrams in this way because we can use the vast and well-developed theory of (regular) languages of finite words and approaches from the field of Automata Learning. We discuss this field in relation to our approach in Sec. II-A.

Sequence diagrams, however, only provide a very partial and exemplary view on the required software. Traditionally, these sequence diagrams are supplemented (by specialists) with other models of the software. Next to sequence diagrams, UML provides the formalism of state machines[4] (also called UML statecharts) in order to model the behavior of an existing system, or to specify behavioral requirements for a system. UML state machines differ from traditional state machines as they permit hierarchical nesting of states. As explained for example in [Yan00], however, a hierarchical state machine can always be flattened into an equivalent traditional state machine.

We will use state machines to describe allowed and forbidden operation sequences. Even though the formalism of UML state machines provides more complicated features, such as entry/exit actions and guard conditions, we require only the basic properties of state machines at this early stage of our research. We want to synthesize a finite automaton that completely specifies desired and forbidden behavior of a service, based on examples of such behavior provided by the user. That is, the synthesized automaton should generalize the examples given by the user, in such a way that the behavior described by the automaton corresponds to the idea of a service that the user has in mind. The behavior described by an automaton is the set of sequences of operations (or *language*) that the automaton accepts. The sequences of operations that are rejected by the automaton form the forbidden behavior (see also Sec. III-A).

[4]See e.g. http://www.omg.org/spec/UML/2.5/PDF/

## II. Related work

### A. Automata learning

Research in this field, also known as grammatical inference or grammar induction, started with the seminal work of Gold [Gol67]. The studied problem is to learn an automaton from a set of examples, that is, from a subset of the language that the automaton should recognize. Optionally, there may also be a set of words that are not accepted by the language. Many variants of this problem exist and have been studied extensively: learning from positive examples only, learning from both positive and negative examples, active learning (with an oracle), learning a language from a specific class, etc. We refer to the surveys [dlH05] and [dlH10] for an overview of the different approaches taken in this field. Deterministic approaches to automata learning start with a tree-like automaton (called prefix tree acceptor (PTA)), built from the input examples and precisely recognizing these. For each prefix present in this set of examples, there is a state present in the PTA (see Fig. 1 for an example of a PTA).

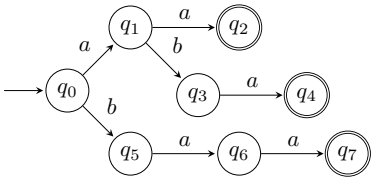


Fig. 1: The PTA for $\{aa, aba, baa\}$.

There exist many different algorithms to transform this PTA into an automaton that generalizes the input examples by merging states of the PTA. A popular one is the Evidence Driven State Merging (EDSM) algorithm, originally introduced in [LPP98], of which different variants exist.

Among the heuristic approaches to the problem of automata learning, different evolutionary algorithms have been applied. Most notable are [LR03], [LR05], [Góm06], [BL05] for deterministic finite automata, and [TE11] for finite state input-output machines. These papers show that evolutionary approaches can compete with state-merging approaches. For example, it was found in [LR03] that evolutionary methods can outperform EDSM when target DFAs consist of less than 32 states. Our problem differs from the above investigations

in two ways. Firstly, our set of training data (i.e. the examples given as user input) is very small compared to the sets of training data used in these studies, where it is not uncommon to have around 3000 input examples. In our scenario, the problem will be highly underspecified. Secondly, in the approaches mentioned above, there exists a target DFA, from which training data is generated. The goal is to construct a DFA equivalent to this target DFA. To which extent this goal is reached can be tested, by generating more test data from the target DFA. In our situation, there is no such target DFA. We should synthesize a DFA that is consistent with the examples and generalizes these in a way that the user wants. Therefore, we produce a set of good DFAs as output, together with information about their characteristics. The user can then make an informed choice and select one of them.

We can however, to some extent, make use of the methods developed in these evolutionary approaches. Most notably, we use an algorithm from [LR05] to determine an optimal set of final states (see Sec. III-A for an explanation).

### B. Generation of statecharts

Many different deterministic forms of automated support for constructing UML state machines from sequence diagrams exist. These approaches guide the user during the construction process and ask for more involvement from the user than we can expect. For example, in [MS00] an interactive algorithm deterministically builds a statechart, asking the user which merges of states are allowed along the way. In [HKP05], a statechart is generated from extended sequence diagrams, called live sequence charts.

### C. Process mining

In dynamic software analysis, the goal is to find rules about the behavior of existing software by analyzing its execution traces. This type of process mining is also called protocol mining. Process mining could be used to automatically deduce a behavioral specification for services in order to estimate whether a service can be reused or composed (e.g., [BIPT09]).

The goal of protocol mining is similar to ours, namely, to synthesize a state machine from sequences of operations (execution traces, similar to our sequence diagrams). A difference is that we are dealing with software that does not yet exist, while in protocol mining one assumes operating software and, potentially, big amounts of execution traces can be collected. Protocol mining is particularly appropriate to apply classical DFA learning techniques because a lot of data is available. A study of related applications can be found in [WBHS08]. An approach that is well-known in this respect is the $k$-tails algorithm, a state-merging approach introduced in [BF72]. In [WBHS08], an overview of different extensions of this algorithm is given. A framework to compare applications of different DFA learning approaches to this field was described in [BBA+13].

### D. Model transformation by example (MTBE)

Research on model transformation by example (MTBE) [KLR+12] is related to this work. In MTBE one needs to transfer instances of one meta-model $A$ into instances of another meta-model $B$. As input, one gets a set of example pairs, each with one model $a$ of type $A$ and the corresponding transformed model $b$ of type $B$. The essential difference to our problem here is that an example of the expected output is provided. Hence, a transformation candidate can be tested by comparing its result $b'$ to the expected output $b$ during the learning phase.

Faunes et al. [FSB13] apply genetic programming to MTBE. The output of the evolutionary approach is a transformation. Differences between desired ($b$) and actual output ($b'$) for each provided pair define the fitness. A similar approach also based on genetic programming by Kuehne et al. [KHAE16] focuses on maintainability and extensibility.

Another approach by Kessentini et al. [KSBBO12], [KSB08] applies particle swarm optimization (PSO). In difference to the above, the PSO algorithm implements a transformation directly. Similarly, we directly synthesize a generalized requirements specification instead of generating an algorithm that implements such a synthesizer. While we face similar challenges as in MTBE, an additional challenge in our problem is that the fitness function is not easily defined as difference to an expected output (see Sec. III-C and III-D for a discussion of a fitness function for our problem).

## III. Our approach

The domain expert provides two sets of sequence diagrams (see Def. 1) as input for our tool (we have implemented a sequence diagram editor to this end, see Sec. III-E). One set contains examples of desired behaviors of the service and the other set describes forbidden behaviors. The user may provide any number of examples and may use any operation name.

We process the sequence diagram files obtained from the user input and extract the information that is relevant for our evolutionary algorithm. We record all used operation names, which form the alphabet $A$ of the candidate DFAs. We extract the sequences of operations from the diagrams (desired resp. forbidden behavior) which form the training data (positive resp. negative) for the algorithm.

### A. Representing DFAs

A *deterministic finite automaton* over an alphabet $A$ is denoted by a tuple $\mathcal{A} = (A, Q, \delta, q_0, F)$, where $Q$ is the set of states, $\delta : Q \times A \to Q$ is the transition function, $q_0 \in Q$ is the unique initial state, and $F \subseteq Q$ is the set of final states (also called accepting states). A DFA $\mathcal{A}$ *accepts* a finite word $w = a_1 \cdots a_n$ iff there exist states $q_0, \ldots, q_n \in Q$, such that for all $i \in \{0, \ldots, n\}$, $\delta(q_i, a_{i+1}) = q_{i+1}$ and $q_n \in F$. If a word is not accepted, it is *rejected*. If $\delta$ is a total function, $\mathcal{A}$ is a *complete* DFA. We define the *size* of a DFA as its number of states $|Q|$. We only work with complete DFAs, as this allows us to represent DFAs as matrices over the set of states. This does not constrain our approach, because every DFA can be extended to a complete DFA, without changing the language.

Also, we keep the number of states constant for all candidate DFAs in the population. This allows us to use standard

variation operators. Note that fixing the size of the DFAs does not prevent us from finding smaller DFAs, since every DFA of smaller size can be extended to a DFA of the given size. In Sec. III-F, we discuss how we process the complete DFAs of fixed size that occur in the output of our program, in order to retrieve possibly smaller and incomplete DFAs.

In our approach, a representation of a complete DFA $\mathcal{A} = (A, Q, \delta, q_0, F)$ has two parts. The first part, which represents $(A, Q, \delta, q_0)$, is a $|Q| \times |A|$ matrix $M$ over $\{0, \ldots, |Q| - 1\}$. When numbering the rows and columns of $M$, we start from 0, i.e., the top-left element of $M$ is denoted by $M_{0,0}$. The entry $M_{i,j}$ encodes the state $\delta(q_i, a_j)$, that is, the state that is reached when reading the letter $a_j$, while being in state $q_i$ (we use the notation $a_j$ to denote the $j$-th operation name).

The second part of the representation needs to encode the set $F$ of final states of $\mathcal{A}$. A natural way to do this, would be to use an array of size $|Q|$, where the $i$-th entry is 1 if $q_i$ is a final state, and 0 otherwise. This would, however, increase the size of the search space with a factor $2^{|Q|}$.

Along the lines of an algorithm of Lucas and Reynolds (called Smart State Labeling (SSL) [LR03], [LR05]) we only represent those DFAs in our search space, whose sets of final states are optimal with respect to the input examples. This allows us to evolve just $|Q| \times |A|$ matrices, which represent only the $(A, Q, \delta, q_0)$-part of a DFA. The corresponding set of optimal final states is then deterministically constructed for each candidate solution, in the following way.

For each state $q_i$ of a candidate solution, We compare the number $p_i$ of positive input examples ending in state $q_i$, and the number $n_i$ of negative input examples ending in it. We say $q_i$ is final iff $p_i \geq n_i$. Note that we still allow all possible $(A, Q, \delta, q_0)$-parts of DFAs, such that the restriction of the search space does not cause any unwanted specialization.

Besides the advantage of having a smaller search space, there is another reason for using the SSL procedure. We are actually interested in DFAs whose set of final states is optimal with respect to the input examples, and finding such a DFA through evolution could be quite involved. Indeed, as explained in [LR03], the set $F$ and the $(A, Q, \delta, q_0)$-part are rather dependent: to improve a candidate DFA, changes in the representations of both parts simultaneously may be needed. Finding the optimal set of final states deterministically allows us to work around this difficulty.

It was empirically shown in [LR05] that an evolutionary algorithm that uses the SSL procedure outperforms a plain evolutionary algorithm, in which both the matrix and a representation of the set of final states are evolved. This was supported by experiments in [Góm06].

### B. Multi-objective optimization

Traditionally, evolutionary approaches to DFA learning use a single objective to evolve candidate solutions. This is, for example, the case in [LR03], [LR05], [BL05], and [Góm06]. In these approaches, the fitness function measures the proportion of input words classified correctly by candidate solutions, that is, positive and negative input words are represented in one objective. In our case, it is advantageous to separate this into two objectives, viz. the proportion of positive examples classified correctly, and the proportion of negative examples classified correctly. This information will help the user to select a solution according to his personal preferences. An additional objective is to generalize the input examples. As often observed in multi-objective optimization, there is not one correct solution to be found. Instead, we want to obtain a varied set of solutions that are excellent with respect to different objectives. Special in our scenario is that only the user can decide which of the synthesized specifications corresponds best to his request. He gets the possibility to make an informed choice and to select an appropriate solution from this set.

We therefore use the NSGA-II algorithm, which was introduced in [DAPM02]. This multi-objective evolutionary algorithm allows to find a diverse Pareto-optimal front of solutions, by taking into account the fitness of solutions for each objective, and how they are spread with respect to these fitness values. Obtaining such a varied set of good solutions as output, gives the user a broad spectrum of DFAs to choose from. Two of our objectives are derived from the obvious thought that a good candidate DFA should be consistent with the input examples: it should accept all positive examples and reject all negative examples. Note, however, that we do not forbid contradictions among the sets of input examples (e.g., a sequence may occur both in the set of positive and negative input examples due to a mistake of the user).

In our scenario, the positive and negative examples are provided by a user, thus their number is limited. Hence, there are very many candidate solutions that are consistent with the input examples. If being consistent with the positive and negative input examples were to be our only objective, this would lead to large fitness plateaus. Therefore, different from other evolutionary approaches to DFA learning where many examples are present, the fitness function here has to address several different aspects.

The small amount of available examples introduces new challenges. For example, a certain operation could be allowed to be repeated arbitrarily often. The user could choose to indicate this by one example only where that operation is repeated three times. In other evolutionary approaches to DFA learning, many more examples could be provided with different numbers of repetitions. Hence, the correct transition loop in the respective automaton could be found. This is even more likely if there are input examples of length larger than the fixed size of the target DFA. Then, generalization is forced implicitly. Thus, we want our algorithm to be able to induce information about repetitions, and more in general, to induce information about transitions between states, from the few examples that we have. This leads to our third and fourth objective: generalizing the input examples (see Sec. III-D).

### C. Objectives 1 and 2: Consistency with the input examples

A positive example is classified correctly by a candidate DFA, if it ends in a state that is declared to be an accepting state by the SSL algorithm. Similarly, a negative example

should end in a rejecting state to be classified correctly. Reusing information collected during the SSL phase, we can compute the numbers of correctly classified positive resp. negative examples. We divide these by the size of the respective input example sets, obtaining the two fractions of correctly classified examples. Our first and second objective are to maximize both of these fractions. If no contradictions are present among the two input sets, clearly, the maximal value for both fractions is 1. Note that we could also maximize the numbers of correctly classified examples. This would give a different scaling of the objective values, but would not change the result of the selection mechanism, which only depends on the relative order and the relative vicinity of the candidate solutions for each fixed objective.

### D. Objective 3 and 4: Generalizing the input examples

As explained above, different from other evolutionary approaches to DFA learning, the first and second objective do not suffice. We introduce a third objective that measures the generalization of the input examples by a candidate solution. A DFA generalizes, if it accepts more than the given positive input examples and/or rejects more than the given negative input examples. In our scenario, a good form of generalization would detect patterns that the user had in mind when specifying his input examples, and allow for repetitions of such patterns. This can be done in many different ways and we do not want to bias our algorithm towards the detection of certain patterns. Instead of measuring a 'degree' of generalization, we measure a related property: the size of a candidate DFA in terms of states. Generalization and size are indeed related, for example, consider an input word $w = w_0 w_1 \cdots w_{k-1}$, where each $w_i$ is a letter from the alphabet. If $k+1$ different states of a DFA are used to read $w$, then the DFA contains a path as depicted in Fig. 2, for some $q_0, \ldots, q_k \in Q$.

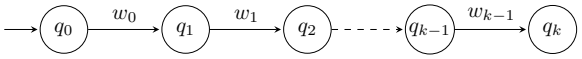


Fig. 2: A path visiting $k+1$ states.

If strictly less than $k+1$ states are used to read $w$, then some generalization must have occurred. For example, a path as in Fig. 3 could be part of the DFA.

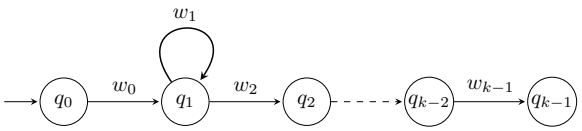


Fig. 3: A path visiting $k$ states.

In the extreme case, just one state is used to read $w$, and the word $w$ has been generalized as much as possible: the order of the operations in $w$ is ignored completely, and only the set of operations used in $w$ is taken into account (see Fig. 4).

Thus, we want to minimize the size of a DFA in order to maximize the extent of generalization of the input examples. In our implementation we actually keep the number of states

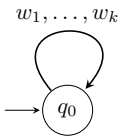


Fig. 4: A path visiting 1 state.

constant for all candidate DFAs. Therefore, we define the *relevant part* of a DFA. Given a DFA and a set of input examples, we say that those states that are visited when reading the examples form the relevant part of the DFA (wrt the examples). The smaller this part is, the more the examples have been generalized. In particular, if this part is smaller than the length of the longest input example, then the DFA has generalized from the input. The relevant part of a given DFA can easily be computed from information needed to perform the SSL procedure. An alternative definition could be to take the set of states that are reachable from the initial state and from which a final state can be reached. This is called *trimming*. However, this trim version is not related to the input examples, and is therefore less suited here as a measure of generalization of the input examples.

With our fourth objective, we let the algorithm favor generalization in a different way. We count the number of so-called sink states (states that have self-loops for every operation name, an example is the state in Fig. 4) and maximize this number. This objective guides the algorithm towards interpreting the input examples as forbidden resp. desired prefixes rather than forbidden resp. desired words. Note that there is a trade-off to be made between a high degree of generalization (fulfilling Objective 3 and 4) and still using enough states to be able to distinguish between the positive input examples and the negative ones (needed to fulfill Objectives 1 and 2).

### E. Implementation

Our sequence diagram editor was implemented in Eclipse, using the Graphical Modeling Framework (GMF) Tooling of the Eclipse Graphical Modeling Project[5]. Our editor enables the user to draw sequence diagrams that exemplify behavior of the service that he would want to acquire. See Fig. 5.

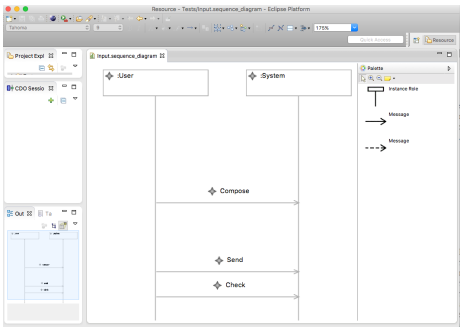


Fig. 5: User input in the sequence diagram editor.

From this input, the relevant information is extracted and passed to our evolutionary algorithm, which we have imple-

[5]http://www.eclipse.org/modeling/gmp/?project=gmf-tooling#gmf-tooling

mented using the MOEA Framework[6]. From this open source Java library, we used the implementation 'backbone' of the NSGA-II algorithm.

### F. Output of our approach

The output of our algorithm is a set of complete DFAs that are Pareto-optimal wrt the objectives described in Sec. III-C and III-D. Due to the fixed size of the DFAs, these may contain states that are not relevant for the specification. We therefore remove all states (and their transitions) that are not used to read any of the input examples. We endow each DFA with an optimal set of final states, computed with the aforementioned SSL algorithm. To avoid overcharging the user, we only present those DFAs from the set, that are reasonably consistent with the input examples (i.e., have value $\geq 0.8$ for objectives 1 and 2). These DFAs are presented together with information on their characteristics (i.e., their objective values). This allows the user to make an informed choice and to select, from this set, the DFA that fits his personal preferences best.

## IV. Results

As parameters, we have used a population size of 100 and a maximal number of evaluations of $2 \times 10^4$. As variation operators, we used the Uniform Mutation operator with parameter $1/(|Q| \cdot |A|)$ and One-Point Crossover operator with probability 1. We have found that our approach works very well for specifications with few operation names, but is at the moment less suited to synthesize useful specifications with many operation names. The problem for such larger alphabets lies in the fact that with just few examples as user input, there is no information present at all for many of the transitions in the DFAs. It is part of our future work, to see how more information from the user can be obtained in these cases.

Let us provide an example which uses the alphabet $A = \{\text{Compose, Send, Cancel, Check}\}$. Suppose a user wants to obtain a service, in which he can compose (an e-mail), then cancel or send this, and in which he can check (for new e-mail). He wants to be able to repeat these combinations of events at will. Note that checking while in the 'Compose-mode' is not allowed. The user provides the following examples as sequence diagrams.

| **Positive examples** | **Negative examples** |
|---|---|
| Check | Send |
| Check, Check | Cancel |
| Check, Check, Check | Compose |
| Compose, Send | Send, Compose |
| Compose, Send, Compose, Send | Send, Cancel |
| Compose, Send, Compose, Send, Compose, Send | Send, Check |
| Compose, Cancel | Cancel, Check |
| Compose, Cancel, Compose, Cancel, Compose, Cancel | Cancel, Cancel |
| Compose, Send, Check | Cancel, Compose |
| Compose, Send, Check, Compose, Send | Compose, Check |
| Compose, Send, Check, Compose, Cancel, Check | Compose, Compose |
| Check, Compose, Cancel, Check, Compose, Send | Compose, Check, Send |
| | Compose, Check, Cancel |

Important information about the behavior of the service is not present in these examples and needs to be inferred by the algorithm. For example, it should be inferred that Check may be called arbitrarily often at any time, except when preceded directly by Compose. Likewise, it should be inferred that the combinations Compose Send and Compose Cancel may be called arbitrarily often at any time, and that Send and Cancel should always be directly preceded by Compose. Another difficulty for the algorithm is that part of the negative examples (e.g. Send and Send Check) are to be understood as forbidden words (but *not* as forbidden infixes), whereas other negative examples (e.g. Send Cancel and Compose Check) are forbidden even as infixes.

If we run the program on these sets of examples with number of states $|Q| = 6$, we obtain a set of 14 candidate DFAs that are Pareto-optimal with respect to our objectives. This set is diverse: one extreme DFA has only one state and rejects every word over the alphabet (thus scoring high on generalization and classifying negative examples correctly, but scoring 0 for classifying positive examples correctly). Other DFAs in the set score better with respect to consistency with the examples, but generalize less. On the other end of the range, there are two solutions, which are completely consistent with the input examples. In our current implementation, these are the only DFAs that are presented to the user. Since these are most consistent with the input examples, they are most relevant to the user. The post-processed versions of these DFAs are given in Fig. 6 and 7. States with loops labeled by the alphabet $A$ are sink states, that is, they have self-loops for every operation name.

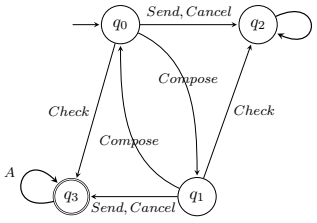


Fig. 6: A synthesized DFA.

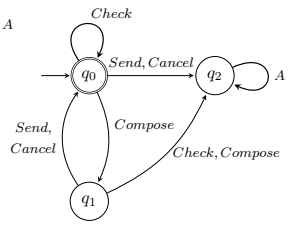


Fig. 7: Another synthesized DFA.

These DFAs generalize the examples in different ways, giving the user the opportunity to choose the direction of generalization. The DFA from Fig. 7 has only 3 states. It has one sink state ($q_2$), which makes sure the negative examples are interpreted as forbidden prefixes. The DFA from Fig. 6, on the other hand, has two sink states ($q_2$, $q_3$) and also interprets the positive examples as desired prefixes. It accepts any word that starts with Check, with Compose Send or Compose Cancel, and also allows extra Compose Compose-prefixes. In this case, the user would choose the DFA from Fig. 7, as it precisely recognizes the language he had in mind.

## V. Conclusion and Perspectives

We proposed a method to synthesize statecharts from example sequence diagrams using a multi-objective evolutionary algorithm. This method can support domain experts in the requirements specification for desired software. Our approach works well for specifications using few operation names, but

[6]http://moeaframework.org/index.html

can still be improved in order to better scale with increasing number of operation names.

Our future work focuses on improvements in two directions. Firstly, we will investigate how to fine-tune our objectives. We expect that adding (or adapting our current) objectives that aim at generalizing the input examples could be worthwhile. Secondly, we want to obtain more information about the desired specification from the user. To this end, we could switch to an approach of interactive evolutionary computation and ask the user whether particular sequences of operations should be allowed or not. These sequences should be chosen to cause maximal disagreement among the good candidate DFAs. One possible way to obtain such sequences, could be to let them co-evolve with the candidate DFAs, similar to [BL05]. There, co-evolution was applied to classical active DFA learning. Another way to obtain more information from the user could be to ask the user, whether the examples he provides should be interpreted as (forbidden resp. desired) words, prefixes, suffixes, infixes or subwords. This would yield a lot more information for the algorithm to work with.

Our overall objective is to fully integrate our work in the context of OTF markets within our project "On-The-Fly Computing". Within the rich context of this project, we will be able to relate candidate specifications to services that can actually be composed based on existing services available on the market. Hence, we aim for an integrated approach that interacts with the user and checks current market situations to iteratively find specifications that fit the user's desires and ensures that the market is able to deliver that software service.

### *Acknowledgments*

The authors would like to thank Klementina Josifovska for her contributions to the implementation of our approach.

## References

[AWBP14] S. Arifulina, S. Walther, M. Becker, and M. C. Platenius. SeSAME: Modeling and analyzing high-quality service compositions. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 839–842, New York, NY, USA, 2014. ACM.

[BBA+13] I. Beschastnikh, Y. Brun, J. Abrahamson, M. D. Ernst, and A. Krishnamurthy. Unifying FSM-inference algorithms through declarative specification. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 252–261, 2013.

[BF72] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Computers*, 21(6):592–597, 1972.

[BIPT09] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 141–150, 2009.

[BL05] J. C. Bongard and H. Lipson. Active coevolutionary learning of deterministic finite automata. *Journal of Machine Learning Research*, 6:1651–1678, 2005.

[DAPM02] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation*, 6(2):182–197, 2002.

[dlH05] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

[dlH10] C. de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, New York, NY, USA, 2010.

[FSB13] M. Faunes, H. Sahraoui, and M. Boukadoum. Genetic-programming approach to learn model transformation rules from examples. In K. Duddy and G. Kappel, editors, *Theory and Practice of Model Transformations: 6th International Conference, ICMT 2013, Budapest, Hungary, June 18-19, 2013. Proceedings*, pages 17–32. Springer, Berlin, Heidelberg, 2013.

[GB16] M. Geierhos and F. S. Bäumer. *How to Complete Customer Requirements*, pages 37–47. Springer International Publishing, Cham, 2016.

[Gol67] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[Góm06] J. Gómez. An incremental-evolutionary approach for learning deterministic finite automata. In *IEEE International Conference on Evolutionary Computation, CEC 2006, part of WCCI 2006, Vancouver, BC, Canada, 16-21 July 2006*, pages 362–369, 2006.

[HKP05] D. Harel, H. Kugler, and A. Pnueli. Synthesis revisited: Generating statechart models from scenario-based requirements. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNCS*, pages 309–324. Springer Berlin Heidelberg, 2005.

[KHAE16] T. Kühne, H. Hamann, S. Arifulina, and G. Engels. Patterns for constructing mutation operators: Limiting the search space in a software engineering application. In *Applications of Evolutionary Computation (EvoApplications 2016)*, volume 9594 of *LNCS*, pages 278–293. Springer, 2016.

[KLR+12] G. Kappel, P. Langer, W. Retschitzegger, W. Schwinger, and M. Wimmer. Model transformation by-example: A survey of the first wave. In A. Düsterhöft, M. Klettke, and K.-D. Schewe, editors, *Conceptual Modelling and Its Theoretical Foundations*, pages 197–215. Springer, Berlin, Heidelberg, 2012.

[KSB08] M. Kessentini, H. Sahraoui, and M. Boukadoum. Model transformation as an optimization problem. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems: 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, pages 159–173. Springer, Berlin, Heidelberg, 2008.

[KSBBO12] M. Kessentini, H. Sahraoui, M. Boukadoum, and O. Ben Omar. Search-based model transformation by example. *Software & Systems Modeling*, 11(2):209–226, 2012.

[LPP98] K. J. Lang, B. A. Pearlmutter, and R. 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*, pages 1–12, 1998.

[LR03] S. M. Lucas and T. J. Reynolds. Learning DFA: evolution versus evidence driven state merging. In *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2003, 8 - 12 December 2003, Canberra, Australia*, pages 351–358, 2003.

[LR05] S. M. Lucas and T. J. Reynolds. Learning deterministic finite automata with a smart state labeling evolutionary algorithm. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(7):1063–1074, 2005.

[MS00] E. Mäkinen and T. Systä. An interactive approach for synthesizing UML statechart diagrams from sequence diagrams. In *Proceedings of OOPSLA 2000 Workshop: Scenario based round-trip engineering*, pages 7–12, 2000.

[TE11] F. Tsarev and K. Egorov. Finite state machine induction using genetic algorithm based on testing and model checking. In *13th Annual Genetic and Evolutionary Computation Conference, GECCO 2011, Companion Material Proceedings, Dublin, Ireland, July 12-16, 2011*, pages 759–762, 2011.

[WBHS08] N. Walkinshaw, K. Bogdanov, M. Holcombe, and S. Salahuddin. Improving dynamic software analysis by applying grammar inference principles. *Journal of Software Maintenance*, 20(4):269–290, 2008.

[Yan00] M. Yannakakis. Hierarchical state machines. In *Theoretical Computer Science, Exploring New Frontiers of Theoretical Informatics, International Conference IFIP TCS 2000, Sendai, Japan, August 17-19, 2000, Proceedings*, pages 315–330, 2000.