2017 IEEE 25th International Requirements Engineering Conference Workshops

Toward Learning Realizable Scenario-based, Formal Requirements Specifications

David Schmelter*, Joel Greenyer† and Jörg Holtmann*
*Software Engineering Department, Fraunhofer IEM
Zukunftsmeile 1, 33102 Paderborn, Germany
Email: [david.schmelter | joerg.holtmann]@iem.fraunhofer.de
†Software Engineering Group, Leibniz Universität Hannover
Welfengarten 1, 30167 Hannover, Germany
Email: joel.greenyer@inf.uni-hannover.de

***Abstract*—Distributed, software-intensive systems such as fully automated cars have to handle various situations employing message-based coordination. The growing complexity of such systems results in an increasing difficulty to achieve a high quality of the systems' requirements specifications, particularly w.r.t. the realizability of the specifications. Scenario-based requirements engineering addresses the message-based coordination of such systems and enables, if underpinned with formal languages, automatic requirements validation techniques for proving the realizability of a requirements specification. However, formal requirements modeling languages require a deep knowledge of requirements engineers and typically require many manual iterations until they find a realizable specification. In order to support requirements engineers in the stepwise development of scenario-based requirements specifications, we propose to evolve a high-quality specification from a (presumably unrealizable) manually created specification employing an evolutionary algorithm. In this paper, we show our results on automatically evolving new assumptions on the systems' environment behavior that guarantee a realizable requirements specification. Based on this contribution, we outline our research roadmap toward our long-term goal of automatically supporting requirements engineers in finding high-quality requirements specifications.**

I. Introduction

Distributed, software-intensive systems such as fully automated cars have to handle various, often safety-critical situations employing message-based coordination (e.g., via vehicle-to-X communication). The growing complexity of such systems also results in an increased size of their requirements specifications. Accompanied by that, a high quality of such specifications is more difficult to achieve if the requirements engineers rely on manual requirements validation techniques. A high quality of a requirements specification particularly means that the specification is realizable at all. That is, the specification defines a system that can meet its guarantees under all assumptions about the environment.

Scenario-based requirements engineering addresses the message-based coordination of such systems and enables, if underpinned with formal languages, automatic requirements validation techniques for proving the realizability of a requirements specification. In previous work, we developed a formal and scenario-based requirements engineering approach based on a recent textual Live Sequence Chart [1] variant, the so-called *Scenario Modeling Language (SML)*. The formal semantics of SML enable automatic requirements validation techniques like simulation and particularly a formal realizability check [2]. Based on game theory, the latter feature allows checking whether the specification is realizable or not, that is, if a systems' environment that satisfies certain assumptions can force the system into a safety or liveness violation which would render the specification unrealizable.

However, formal requirements modeling languages like SML require a deep knowledge by the requirements engineers and typically require many manual iterations until they find a realizable specification. That is, the requirements engineers initially conceive an underspecified and unrealizable requirements specification. Afterward, they iteratively apply the realizability check and evolve the underspecified parts of the specification until a realizable solution is found.

Since this approach can be naturally seen as an evolutionary problem, we propose to employ a Multi-objective Evolutionary Algorithm (MOEA) [3, chapter 6] in the early phase of requirements engineering. This allows evolving realizable requirements specifications as candidate solutions from a manually created and presumably unrealizable specification.

In this paper, we show our results on automatically evolving new assumptions on the environment behavior that guarantee a realizable scenario-based requirements specification. The main contributions are the encoding of SML Requirements Specifications (SML Specifications) as a MOEA problem, the incorporation of the automatic realizability check into the MOEAs' fitness function, and the corresponding proof of concept. We illustrate the approach and perform the proof of concept with an example of an automotive vehicle-to-X driver assistance system, the so-called *Emergency Braking and Evasion Assistance System (EBEAS)*, cf. [4, chapter 4]. Based on these results, we outline our research roadmap toward our long-term goal of automatically supporting the requirements engineers in finding realizable as well as high-quality requirements specifications—particularly also including evolving the functional requirements on the system under development.

We present foundations on SML and problem encoding for MOEAs in Sect. II. Afterward, we illustrate our main contributions (overview of our approach, problem encoding of SML Specifications, and the conceived fitness function) in Sect. III. We present our proof of concept, related work, as well as our research roadmap in Sect. IV, Sect. V, and Sect. VI, respectively and conclude our work in Sect. VII.

978-1-5386-3488-2/17 $31.00 © 2017 IEEE
DOI 10.1109/REW.2017.14

II. Foundations

We divide the foundations in two parts: Sect. II-A introduces relevant concepts of SML, in particular the most important language elements as well as automatic analysis techniques like realizability checking. Sect. II-B introduces Grammatical Evolution which is a suitable approach for encoding SML Specifications as a learning problem for MOEAs.

A. SML Specifications

An SML specification defines how objects in an *object model* shall interact by sending messages. SML is a textual variant of Live Sequence Charts (LSCs) [5], extended with concepts for modeling environment assumptions.

We consider *synchronous* communication where the sending and receiving of a message is a single *message event*. A message has one sending and one receiving object and it refers to an operation defined for the receiving object. A *run* of a system is an infinite sequence of message events.

The object model is partitioned into *controllable* (*system*) objects and *uncontrollable* (*environment*) objects. A message event is a (*controllable*) *system event* if it is sent by a system object and it is an (*uncontrollable*) *environment event* if it is sent by an environment object.

An SML specification is a scenario-based *assume/guarantee specification* that defines the valid runs of a system by a set of *assumption-* and *guarantee scenarios*. Guarantee scenarios describe how system objects may, must, or must not react to environment events; assumption scenarios describe what may, will, or will not happen in the environment, or how the environment may, will, or will not, react to system events.

Listing 1 shows an SML specification of a simplified EBEAS example consisting of three guarantee scenarios within a *collaboration*. Collaborations are used to structure a specification and define *roles* that represent objects in the system. The scenarios refer to the roles in order to define valid sequences of message events. The first scenario HandleObstacle, for example, says that whenever the (Adaptive Cruise Control) (acc) notifies the EBEAS of an obstacle, the EBEAS must send an emergency break command (emcyBrake) to the Vehicle Control (vc).

Scenario messages can have two modalities *strict* and *requested*, which model safety resp. liveness properties. In brief, when a strict message becomes enabled, the order of messages as described by the scenario must not be violated; when a requested message becomes enabled, it means that the scenario must eventually progress.

For example, when acc→ebeas.obstacle() occurs, then ebeas→vc.emcyBrake() is enabled in HandleObstacle, and eventually the message event ebeas→vc.emcyBrake() must occur (requested→liveness), and no second acc→ebeas.obstacle() must occur before ebeas→vc.emcyBrake() (strict→safety).

The guarantee scenario HandleLastPointToBrake says that when the car passed a last point to break (before reaching an obstacle), the EBEAS must trigger an emergency evading procedure; ForbidBrakingAfterEvade says that when after triggering an emergency evading procedure, the EBEAS must not trigger an emergency braking procedure.

Listing 1: Unrealizable SML Specification

```
import "ebeas.ecore"
specification EbeasSpecification {
 domain ebeas
 controllable { EBEAS }
 collaboration ObstacleDetection {
  static role AdaptiveCruiseControl acc
  static role EBEAS ebeas
  static role VehicleControl vc
  guarantee scenario HandleObstacle {
   acc->ebeas.obstacle()
   strict requested ebeas->vc.emcyBrake()
  }
  guarantee scenario HandleLastPointToBrake {
   acc->ebeas.lastPointToBrake()
   strict requested ebeas->vc.emcyEvade()
  }
  guarantee scenario ForbidBrakingAfterEvade {
   ebeas->vc.emcyEvade()
   ebeas->vc.emcyBrake()
   violation [true]
}}}
```

Listing 2: Learning Goal: Realizable SML Specification

```
... assumption scenario
  ↪  ForbidObstacleAfterLastPointToBrake {
 acc->ebeas.lastPointToBrake()
 acc->ebeas.obstacle()
 violation [true]
}}}
```

Now the problem occurs that when the (uncontrollable) event acc→ebeas.lastPointToBrake() is followed by the (uncontrollable) event acc→ebeas.obstacle(), a violation of either guarantee scenario becomes inevitable. We can fix this problem by adding an assumption scenario to the specification that says that this succession of uncontrollable events does not occur in the environment (cf. Lst. 2).

This is an example of how an *unrealizable* specification can be fixed to be *realizable*. A specification is *realizable* if there exists a strategy for the system to choose system events in such a way that it can react to any sequence of environment events in ways that when all assumption scenarios are satisfied, also all guarantee scenarios are satisfied. A specification is unrealizable if such a strategy does not exist. The expressive power of SML specifications is comparable with that of GR(1) (Generalized Reactivity of rank 1) [6], and checking the realizability of an SML specification can be mapped to a GR(1)-game [7]. In fact, ScenarioTools[1], an Eclipse-based[2] tool suite that supports the modeling, simulation, and analysis of SML specifications, implements the algorithm described in [7] for realizability checking SML specifications [8].

B. Problem Encoding with Grammatical Evolution

Evolutionary algorithms employ genetic operators such as crossover and mutation in order to solve optimization problems and they apply these operators on the *genotype*. The genotype is an encoded representation of the *phenotype*, i.e., an individual in the population of candidate solutions. Accordingly, problem

[1]http://scenariotools.org/
[2]https://www.eclipse.org/

TABLE I: "The chromosome of an individual. Each gene represents a random number which can be used in the translation from genotype to phenotype." [9]

220	203	17	3	109	215	104	30

encoding plays an important role in the design of evolutionary algorithms for a particular problem. A *genotype* consists of one or several *chromosomes* which by themselves are, inter alia, represented by integer strings (*the genes*).

We regard the incremental learning of assumption scenarios for formal requirements languages like SML as a problem of automatic program generation and apply *Grammatical Evolution* [9] in order to represent and evolve SML Specifications. In Grammatical Evolution, the genotype of an individual consists of a chromosome with a variable number of pseudo random integer genes, aka *codon*. The phenotype is then derived based on this codon by means of evaluating a Backus-Naur-Form grammar (BNF grammar): Whenever a production rule in the BNF grammar for the target language contains more than one alternative, the next integer gene is selected from the codon in order to decide on the rule to be chosen by applying a modulo operation with the current production rules' number of alternatives. The resulting number represents the rule index which refers to the rule to be chosen.

Consider the following example: Table I shows a codon containing 8 pseudo-random integer genes. Listing 3 depicts a BNF grammar snippet for deriving simple mathematical expressions with a production rule consisting of four terminals (Line 3). Lets assume we are in the process of translating the genotype to a phenotype, have to decide on an alternative for the production rule <*pre-op*>, and the last codon used is 203. Then, Grammatical evolution would select the terminal *"Cos"* by applying a modulo 4 ($\widehat{=}$ number of alternatives) operation on the next selected codon (= 17). We get 17 MOD $4 = 1$ which is the rule index corresponding to *"Cos"*.

Listing 3: BNF grammar snippet for simple expressions (cf. [9])

```
2  ...
3  <pre-op> ::= "Sin" | "Cos" | "Tan" | "Log"
4  ;Rule index:   0       1       2       3
5  ...
```

By applying Grammatical Evolution as a problem encoding for SML Specifications, we achieve a very simple genotype representation applicable for standard genetic operators like crossover and mutation and have a very flexible way for phenotype derivation by modifying the underlying BNF grammar.

III. Learning SML Specifications employing a Multi-objective Evolutionary Algorithm

In order to learn a realizable specifications from a manually created one, we employ a MOEA that comprises NSGA-II [10] as the genetic algorithm, a single decision variable realized by means of Grammatical Evolution, and a custom fitness function. In an overview in the next section, we illustrate our approach which starts by creating an SML Specification manually, incorporates the MOEA and concludes with a manual evaluation of the evolved pareto optimal solutions. In Sect. III-B, we explain our encoding of SML Specifications as input for the MOEA. We conclude this main chapter with a description of the conceived MOEAs' fitness function in Sect. III-C where we use realizability checking to calculate an objective value as one optimization objective. We refer to our EBEAS running example for the following explanations (cf. Lst. 1).

A. Overview

Figure 1 depicts the overview of our approach in BPMN notation. Up to now, as elucidated in the introduction, we focus on learning assumption scenarios as one problem class for unrealizable SML Specifications.

First, the requirements engineer creates an initial specification in Task 1. Create initial SML Specification. The resulting output of this task is the *Manually Created SML Specification* from which we want to learn a realizable SML Specification (cf. Lst. 1). Next, the requirements engineer checks the created specification for realizability by means of realizability checking (cf. Sect. II-A). If the manually created SML Specification is not realizable, we prepare our MOEA in Task 3. Generate BNF grammar for EA. Otherwise, the software engineering can continue as sketched by Task 6. Continue with MDSD and/or Quality Control. In Task 3, we generate a *BNF grammar* for our MOEA that is tailored for the given *Manually Created SML Specification*. This grammar is used for the MOEAs' encoding and is explained in detail in Sect. III-B.

Afterward, the MOEA, comprised of three tasks, is executed (Subprocess 4. Run Multi-objective Evolutionary Algorithm):

4.1. Evolve Assumption Scenario Candidates: First, the populations of assumption scenario candidates are calculated by the MOEA based on the generated *BNF grammar* of Task 3. We provide details like optimization objectives, population size, maximum evaluations, etc. in Sect. III-C and Sect. IV.

4.2. Merge Evolved Scenarios with Initial Specification: Second, we merge each evolved set of *Assumption Scenarios* from Task 4.1 with the *Manually Created SML Specification*. Our motivation behind this task is to make sure that the MOEA does not modify the specification provided by the requirements engineers in order to avoid unwanted changes that they might not recognize (this is subject to future work, cf. Sect. VI). The result of this task is a *candidate SML Specification*.

4.3. Evaluate Merged Specification: Third, we evaluate each *Candidate Specification* from Task 4.2 by means of the MOEAs' fitness function. The outcome of Task 4.3 is a set of pareto optimal solutions as candidates for a realizable SML Specification (cf. Lst. 2).

Finally, we allow for a semi-automatic evaluation of the calculated pareto optimal solutions by the requirements engineer in Task 5. Evaluate Pareto Optimal Solutions. Here, the requirements engineer can use simulation techniques like Play-Out [2] to comprehend the evolved candidates, potentially tweak them further, and select the solution that match his engineering task best. If it is necessary to apply manual changes to the chosen solution, the overall process is re-iterated at Task 2. Check Realizabilty. Otherwise, subsequent software engineering

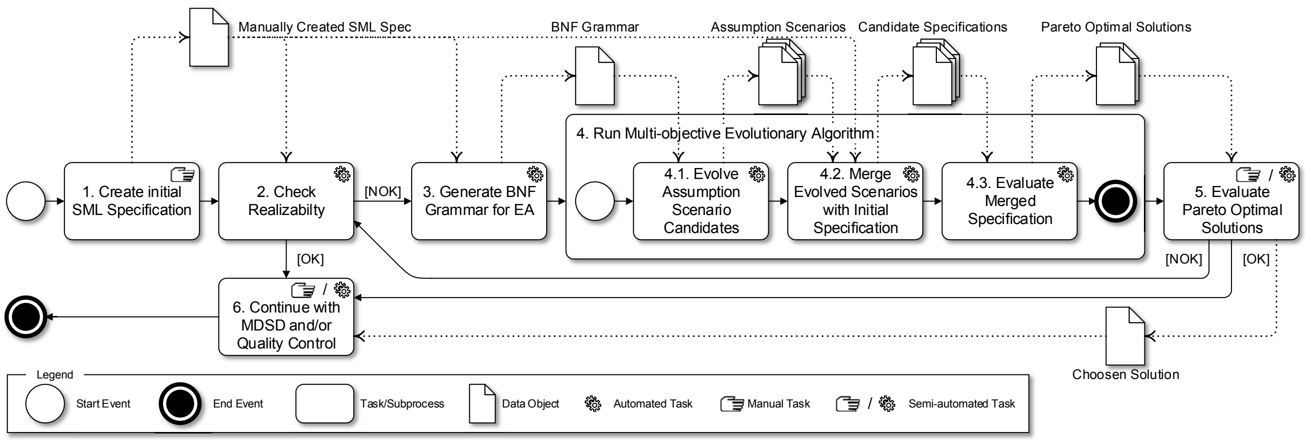
Manually Created SML Spec
BNF Grammar
Assumption Scenarios
Candidate Specifications
Pareto Optimal Solutions
1. Create initial SML Specification
2. Check Realizabilty
[NOK]
3. Generate BNF Grammar for EA
4. Run Multi-objective Evolutionary Algorithm
4.1. Evolve Assumption Scenario Candidates
4.2. Merge Evolved Scenarios with Initial Specification
4.3. Evaluate Merged Specification
5. Evaluate Pareto Optimal Solutions
[OK]
[NOK]
[OK]
6. Continue with MDSD and/or Quality Control
Choosen Solution
Legend
Start Event
End Event
Task/Subprocess
Data Object
Automated Task
Manual Task
Semi-automated Task

Fig. 1: Overview of Learning Realizable SML Specifications

can continue based on the *Choosen Solution* as sketched by Task 6. Continue with MDSD and/or Quality Control.

B. Encoding SML Specifications

We apply Grammatical Evolution in order to encode SML Specifications for evolution by means of a MOEA (cf. Sect. II-B). Since we want to learn assumption scenarios in the first place in order to evolve realizable SML Specifications we do not provide a BNF grammar for the complete language of SML (at least in this early stage of research). Instead, we focused on production rules dedicated to evolving assumption scenarios in order to achieve a rather small grammar for improved comprehensibility and efficient evolution. In order to evolve only syntactically correct candidate scenarios, we have to encode some information from the domain model provided by the requirements engineer as terminals, namely role and message names. Therefore, we generate a dedicated BNF grammar for each SML Specification as input for our approach from the domain model provided by the requirements engineer (cf. Task 3. in Fig. 1).

Consider the BNF grammar of the EBEAS example shown in Lst. 4: Line 1 contains the start symbol of the grammar and allows to produce one or more assumption scenarios recursively (cf. Line 2). Each evolved scenario has the fixed name $Scenario\_\_$ (cf. Line 2) which is numbered consecutively in ascending order in a post-processing step. Lines 3–12 contain the production rules for system and environment messages. Our grammar supports three interaction patterns between roles: messages from an environment role to a system role (Line 5), from a system role to an environment role (Line 6), and from a system role to another system role (this enables self messages for system roles, Line 7). Lines 8–12 contain production rules for different modalities we allow up to date for both environment and system messages. Lines 13–14 contain production rules for sender roles. Lines 15–16 contain production rules for receiver role messages. Note that the coupling between receiver roles and messages is hard-coded in the generated grammar in order to evolve only valid assumption scenarios. Finally, Line 17 contains the production rule which allows recursive evolution of scenario messages as well as specifying a negative scenario with the concluding fragment $violation\ [true]$.

C. Fitness Function and Incorporated Analysis Techniques

The fitness function of our MOEA calculates six optimization objectives which we describe in the following. First of all, we execute a realizability check for each merged candidate specification (cf. tasks 4.2 and 4.3 in Fig. 1) in order to calculate the first objective O_1. If the check is successful, i.e., the specification is realizable, we set $O_1 = 1$. Otherwise, we set $O_1 = 0$. Thereby, this objective represents a binary objective we want to maximize.

Optimization objectives O_2 and O_3 are objectives to address the quality, particularly the understandability, of SML Specifications and are to be minimized. We argue that specifications with a small amount of scenarios as well as a s-mall amount of messages per scenario are more comprehensible than specifications with a lot of scenarios and a high amount of messages per scenario. Let A be the set of assumption scenarios in a merged candidate specification, $a_i \in A$ a single assumption scenario in A with $i = 1..n$ and $n = |A|$, and $M(a_i)$ a function which returns the ordered set of messages in a_i (up to now, we do not consider additional SML language elements like nested fragments). Consequently, we define O_2 and O_3 as follows:

$$\text{Minimize} \left\{ \quad O_2 = |A| \quad O_3 = (\sum_{i..1}^{n} |M(a_i)|) \ / \ |A| \right.$$

Optimization objectives O_4, O_5 and O_6 are used to optimize for *meaningful* SML Specifications. Since we are evolving assumption scenarios, we want to maximize the environment-related information, i.e., the amount of environment messages per assumption scenario. Let $M_e = \{m_e \in \bigcup M(a_1)..M(a_n) : m_e \text{ is uncontrollable}\}$ be the set of all environment messages in A. Consequently, we define O_4 as an objective to maximize as follows:

$$\text{Maximize} \{ \quad O_4 = |M_e| \ / \ |A|$$

Listing 4: BNF grammar for Learning Syntactically Correct EBEAS Assumption Scenarios

```
<scenarios> ::= <scenarios> <scenario> | <scenario>
<scenario> ::= "assumption scenario Scenario__ " <root_interaction> "\n"
<root_interaction> ::= "{\n" <message> "\n" <interaction_fragments> "\n}\n"
<message> ::= <env_message> | <sys_message>
<env_message> ::= <env_sender_role> "->" <sys_receiver_role_message> "\n"
<sys_message> ::= <sys_sender_role> "->" <env_receiver_role_message> "\n"
<sys_self_message> ::= <sys_sender_role> "->" <sys_receiver_role_message> "\n"
<modal_message> ::= <modal_env_message> | <modal_sys_message>
<modal_env_message> ::= <env_modality> <env_message>
<modal_sys_message> ::= <sys_modality> <sys_message>
<env_modality> ::= "" | "eventually " | "monitored eventually "
<sys_modality> ::= "" | "eventually " | "monitored eventually "
<env_sender_role> ::= acc | vc
<sys_sender_role> ::= ebeas
<env_receiver_role_message> ::= vc.emcyBrake() | vc.emcyEvade()
<sys_receiver_role_message> ::= ebeas.obstacle() | ebeas.lastPointToBrake()
<interaction_fragments> ::= <interaction_fragments> <modal_message> | <modal_message> | <modal_message>
  ↪  "violation [true] \n"
```

Moreover, we want to avoid duplicate subsequent messages. We define that two messages *match* when they have the same sender role, receiver role, and message name. Let $T_i = \{\bigcup\{(m_a, m_{a+1})\} : m_a, m_{a+1} \in M(a_i);\ m_a, m_{a+1} \text{ match}\}$ be the set of matching message tuples in a_i. Consequently, we define O_5 as an objective to minimize as follows:

$$\text{Minimize} \left\{ \quad O_5 = \sum_{i..1}^{n} |T_i| \right.$$

Finally, we want to avoid trivial solutions for realizability that are not *well-separated* (cf. [11]), i.e., we want to avoid assumption scenarios that are just (in part) copies of guarantee scenarios. Hereto, we identify the number of *matching* message sequences between guarantee and assumption scenarios and define O_6 as an objective to minimize as follows: Let G be the set of guarantee scenarios in a merged candidate specification and $g_j \in G$ a single guarantee scenario in G with $j = 1..m$ and $m = |G|$. Let

$$\begin{aligned} C = \bigcup_{j..1}^{m}\bigcup_{i..1}^{n}\{\{(m_{a\_1}, .., m_{a\_k}), (m_{g\_1}, .., m_{g\_k})\} : \\ m_{a\_1}, .., m_{a\_k} \in M(a_i), m_{g\_1}, .., m_{g\_k} \in M(g_j), \\ m_{a\_1}, .., m_{a\_k} \text{ match } m_{g\_1}, .., m_{g\_k} \text{ pairwise}, \\ 1 \le k \le min(|M(a_i)|, |M(g_j)|)\} \end{aligned}$$

be the set of message tuples in assumption and guarantee scenario that match pairwise. Consequently, we define O_6 as an objective to minimize as follows:

$$\text{Minimize} \{ \quad O_6 = |C|$$

D. Summary

In conclusion, our novel approach is the combination of evolving specifications by means of a MOEA and the inclusion of automatic analysis techniques in its fitness function early in the development process. First, we combine different quality objectives like realizability and comprehensibility in a single algorithm. This takes the burden from the requirements engineer in finding realizable specifications manually—which is typically a tedious process since even small manual changes to an specification might result in its unrealizability. Second, the presentation of different Pareto-optimal candidate solutions helps the requirements engineer in understanding the reasons for unrealizability and finding the best solution for his engineering task. Third, we enable a smooth transition to quality assurance (by providing specifications in a high quality as input for test case generation) as well as Model-driven Software Design (MDSD) (by providing controllers for the system under development which serve as an initial software architecture including behavioral models).

IV. Proof of Concept

In order to show the principle viability of our approach, we implemented a proof of concept. We give details about our implementation in Sect. IV-A and discuss experimental results in Sect. IV-B.

A. Implementation

We use the *MOEA Framework*[3] as the basis for our proof of concept and use NSGA-II [10]—a widely used MOEA that enables us to find a Pareto-optimal frontier of solutions—as the underlying algorithm. This frontier gives the requirements engineer a broad range of SML Specification candidates to choose from. In our experiments, we use the following parameters for the MOEA: *maximum number of evaluations* is 250, *population size* is 50, and *crossover* and *mutation rates* are kept at the default value 1.0. The codon length is set to the maximum value of 256. Further optimization of these parameters is subject to future work. Our implementation is realized as a plugin for the Eclipse platform and we use Xtend[4] to generate the BNF grammar [12].

B. Discussion of Results

Our proof of concept evolves a Pareto-optimal frontier of candidate solutions for the EBEAS example. This frontier contains a near-optimal realizable SML Specification continuously,

[3]http://moeaframework.org/
[4]https://www.eclipse.org/xtend/

i.e., across different runs of the algorithm. One of the derived solutions that comes very close to the learning goal depicted in Lst. 2 is shown in Lst. 5. The only difference in this evolved specification is the additional *eventually* modifier in Line 24.

Listing 5: Best Evolved Realizable SML Specification Candidate

```
22 ... assumption scenario Scenario_1 {
23  acc->ebeas.lastPointToBrake()
24  eventually acc->ebeas.obstacle()
25  violation [true]
26 }}}
```

The most important threats to validity are as follows: First, our experiments are based on a very small example that is probably not representative for larger SML Specifications. Second, our BNF grammar covers only a rather small subset of the SML language to date. Third, a majority of evolved candidate SML Specifications are not meaningful, i.e., semantically correct, and/or helpful. Consider Lst. 6 as an example: The evolved candidate solution basically specifies that the last point to brake must not be indicated by the Adaptive Cruise Control (ACC) after an obstacle has been detected, which would be a direct contradiction to our learning goal. Moreover, the frontier of Pareto-optimal solutions typically contains candidates with up to eight assumption scenarios that are hard to comprehend.

Listing 6: Suboptimal Evolved Realizable SML Specification

```
22 ... assumption scenario Scenario_2 {
23  acc->ebeas.obstacle()
24  eventually acc->ebeas.lastPointToBrake()
25  violation [true]
26 }
```

Despite the stage of our approach represents ongoing research, our experiments show its principle viability. Hence, we argue that it has the potential in helping the requirements engineer finding realizable specifications in a high quality by providing concrete solution candidates after following our research roadmap presented in Sect. VI.

V. Related Work

Rooijen and Hamann describe an approach to learn generalized requirements specifications (represented as Deterministic Finite Automatons (DFAs)) from positive and negative input examples (represented as sets of simplified UML Sequence Diagrams) in [13]. They want to enable *domain experts* to create requirements in a user-friendly way—independently of the *requirements engineer*—by means of positive and negative input examples and derive a requirements specification that is a good generalization from these examples. The main differences to our work are the targeted roles (domain expert vs. requirements engineer), input examples (simplified UML Sequence Diagrams vs. formal SML Specifications), learning goals (DFA vs *realizable* SML Specifications), applied encoding (fixed size adjacency matrix vs. flexible BNF grammar), and the design of the fitness function where we apply realizability checking to calculate one of six optimization objectives.

Previous work on generating environment assumptions for GR(1) specifications has been presented in [14] and [15]. Li et al. present a template-based approach for mining environment assumptions to correct unrealizable LTL specifications. Their approach utilizes counter examples from realizability checking as well as exemplary traces provided by the user to generate candidate assumptions. Alur et al. also apply a counter example guided approach for adding environment assumptions to an unrealizable specification in order to achieve realizability. Both approaches focus on generating environment assumptions based on counter-examples which stem from realizability checks. We argue, that our MOEA-based approach includes a generally bigger search space than their approaches based on counter-examples. Moreover, due to multi-objective optimization, we include additional quality objectives beyond realizability checking that enables optimization toward informal objectives like comprehensibility and makes our approach very flexible for ongoing research.

Further approaches for learning requirements with evolutionary algorithms are presented in [16] and [17]. Martorell et al. present an approach to optimize surveillance requirements for the use case of nuclear power plant specifications. Sutcliffe et al. present an approach to optimize requirements for the use case of naval command and control systems. In contrast to their work, we do not address single use cases but aim to support arbitrary specifications for distributed, software-intensive systems.

In conclusion, combining evolutionary learning of formal specifications by means of a multi-objective algorithm incorporating automatic analysis techniques early in the development process is a novelty of our approach.

VI. Research Roadmap

Based on our proof of concept, our research roadmap is diverse. We want to pursue the following research directions:

Learning meaningful specifications: Learning realizable but meaningless SML Specifications should be avoided. One idea toward learning meaningful specifications is to include the degree of test coverage in the fitness function: A merged specification that is comprised of manually created guarantee scenarios and automatically evolved assumption scenarios would be compared to accepted and rejected message sequences. These message sequences can be specified as a set of positive and negative test cases by the requirements engineer (e.g., in terms of *existential scenarios*).

Addressing additional problem classes: Up to now, we only address underspecified environment assumptions as a problem class for unrealizable specifications. However, this is only one possible cause for unrealizability. For example, MSD Specifications [4], another variant of Live Sequence Charts, enable the modeling of real-time requirements which can also lead to an unrealizable specification because of timing contradictions between different scenarios. In the future, we want to identify such problem classes and extend our approach to address them. In this paper, we explicitly do not allow modifications to the manually created parts of a specification and we want to investigate which kind of automatic modifications by a MOEA would be needed and/or desirable in order to achieve realizable specifications in a high quality in the future.

Improving the quality of specifications: Beside realizability, we only address two rather simple objectives in our fitness function in order to improve the quality of the evolved SML Specifications: Number of evolved scenarios and number of evolved messages per scenario (cf. O_2, O_3 in Sect. III-C). Today, it is unclear which quality objectives a specification should fulfill and which objectives are realizable as quantifiable objectives in the fitness function of our MOEA. Thus, we want to identify and incorporate additional objectives in order to evolve realizable specifications in a high quality.

Improving the MOEA: The design of or MOEA leaves a lot of room for further research: First, we want to conduct an in-depth evaluation of different MOEA parameterizations concerning population size, selection, crossover, and mutation strategies in order to optimize the effectiveness of our MOEA. Second, we will extend the BNF grammar in order to (1) support additional SML language elements and (2) support additional languages like MSD Specifications in order to enable the evolution of specifications with real-time requirements.

Integrating EAs in development processes: Ultimately, we want to embedded our approach in our previously conceived development processes (cf. [18]). This encompasses a process definition including automated and semi-automated/manual process steps as well as the definition of guidelines on how to apply our MOEA and interpret candidate solutions effectively.

VII. Conclusion

In this paper, we proposed to employ a MOEA to evolve high-quality SML specifications from initial, manually conceived ones. We presented an encoding of SML specifications as a MOEA problem by means of grammatical evolution, the incorporation of an automatic realizability check of SML specifications into the MOEAs' fitness function, and the corresponding proof of concept. These results showed for the problem class of underspecified environment assumptions how we automatically evolve new assumption scenarios such that safety violations of the guarantee scenarios are avoided. We outlined our research roadmap toward evolving complete, high quality scenario-based requirements specifications (i.e., particularly including evolution of guarantee scenarios).

Our results facilitate the conception of evolving a complete set of initially underspecified assumptions on the environment behavior by providing concrete candidate solutions. Furthermore, the proof of concept on our results shows the principle viability of the approach. Finally, our research roadmap paves the way for semi-automatically supporting the requirements engineers in conceiving realizable and high-quality specifications, thereby reducing their manual and cognitive effort on this sophisticated task.

For future work, we will follow our research roadmap by focusing on bigger examples and the derivation of meaningful candidate specifications as next steps. To this end, we plan to conduct a user study and detailed evaluation in order to analyze the applicability and performance of our approach for realistic specifications. Afterward, we want to address additional problem classes and further optimization objectives.

Acknowledgment

We thank Marie Christin Platenius for valuable feedback on drafts of the paper. This research and development project is funded by the German Federal Ministry of Education and Research (BMBF) within the Leading-Edge Cluster "Intelligent Technical Systems OstWestfalenLippe" (it's OWL) (managed by the Project Management Agency Karlsruhe, PTKA), by the ITEA2 AMALTHEA4public project (managed by the Project Management Agency of the German Aerospace Center, PT-DLR), and by the German-Israeli Foundation for Scientific Research and Development (GIF, grant no. 1258).

References

[1] W. Damm and D. Harel, "LSCs: Breathing life into message sequence charts," *Formal Methods in System Design*, vol. 19, pp. 45–80, 2001.

[2] J. Greenyer, D. Gritzner, G. Katz, and A. Marron, "Scenario-based modeling and synthesis for reactive systems with dynamic system structure in ScenarioTools," in *MoDELS 2016 Demo and Poster Sessions*, vol. 1725. CEUR, 2016, pp. 16–32.

[3] X. Yu and M. Gen, *Introduction to Evolutionary Algorithms*. Springer, 2012.

[4] J. Holtmann, M. Fockel, T. Koch, D. Schmelter, C. Brenner, R. Bernijazov, and M. Sander, "The MechatronicUML requirements engineering method: Process and language," Fraunhofer IEM / Heinz Nixdorf Institute, Tech. Rep. tr-ri-16-351, 2016.

[5] D. Harel and R. Marelly, *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[6] N. Piterman, A. Pnueli, and Y. Sa'ar, *Synthesis of Reactive(1) Designs*, ser. LNCS. Springer, 2006, vol. 3855, pp. 364–380.

[7] K. Chatterjee, W. Dvorák, M. Henzinger, and V. Loitzenbauer, "Conditionally optimal algorithms for generalized Büchi games," in *41st Int. Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*, 2016, pp. 25:1–25:15.

[8] D. Gritzner and J. Greenyer, "Synthesizing executable PLC code for robots from scenario-based GR(1) specifications," in *Proc. of the 4th Int. Workshop on Model-driven Robot Software Engineering*, 2017.

[9] C. Ryan, J. J. Collins, and M. O. Neill, "Grammatical evolution: Evolving programs for an arbitrary language," in *Proc. Genetic Programming: First European Workshop (EuroGP 1998)*. Springer, 1998, pp. 83–96.

[10] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[11] S. Maoz and J. O. Ringert, "On well-separation of GR(1) specifications," in *Proc. FSE 2016*. ACM, 2016, pp. 362–372.

[12] D. Schmelter, "Supplementary material on learning realizable scenario-based, formal requirements specifications," 2017. [Online]. Available: https://doi.org/10.5281/zenodo.810043

[13] L. v. Rooijen and H. Hamann, "Requirements specification-by-example using a multi-objective evolutionary algorithm," in *2016 IEEE 24th Int. Requirements Engineering Conference Workshops (REW)*, 2016, pp. 3–9.

[14] W. Li, L. Dworkin, and S. A. Seshia, "Mining assumptions for synthesis," in *9th ACM/IEEE Int. Conference on Formal Methods and Models for Codesign (MEMPCODE2011)*, 2011, pp. 43–50.

[15] R. Alur, S. Moarref, and U. Topcu, "Counter-strategy guided refinement of GR(1) temporal logic specifications," in *2013 Formal Methods in Computer-Aided Design*.

[16] S. Martorell, S. Carlos, J. Villanueva, A. Sanchez, B. Galvan, D. Salazar, and M. Cepin, "Use of multiple objective evolutionary algorithms in optimizing surveillance requirements," *Reliability Engineering and System Safety*, vol. 91, no. 9, pp. 1027 – 1038, 2006, special Issue - Genetic Algorithms and Reliability.

[17] A. Sutcliffe, W.-C. Chang, and R. Neville, "Optimizing system requirements with evolutionary computation," in *Proc. Evolutionary Computation (CEC 2002)*, vol. 1, 2002, pp. 495–499.

[18] J. Holtmann, R. Bernijazov, M. Meyer, D. Schmelter, and C. Tschirner, "Integrated and iterative systems engineering and software requirements engineering for technical systems," *J. Softw. Evol. Process*, vol. 28, no. 9, pp. 722–743, 2016.