Towards Automated Defect Analysis using Execution Traces of Scenario-based Models

Joel Greenyer[1], Daniel Gritzner[1], David Harel[2] and Assaf Marron[2]
[1]*Leibniz Universität Hannover, Hannover, Germany*
[2]*The Weizmann Institute of Science, Rehovot, Israel*

Keywords: Software Engineering, System Engineering, Scenario-based Programming, Behavioral Programming, Abstraction, Debugging, Program Repair, Execution Trace, Event Log

Abstract: Scenario-based specification approaches offer system engineering advantages with their intuitiveness, executability, and amenability to formal verification and synthesis. However, many engineering tasks such as debugging or maintenance are still far from trivial even when using such specifications. Specifically, it is hard to find out why a complex system behaves as it does, or how it would behave under certain conditions. Here, we present work in progress towards the (semi-)automatic analysis of event traces emanating from simulation runs and actual executions. These traces may be large, yet developers are often interested only in specific properties thereof, like is any specification property violated? are particular properties demonstrated? is there a smaller sub-sequence of events that violates or demonstrates the same properties? which trace properties are common to multiple traces and which are unique? etc. Our approach includes automatic techniques for discovering and distilling relevant properties of traces, analyzing properties of sets of traces, using (sets of) execution traces for understanding specified and actual system behavior and problems therein, planning system enhancement and repair, and more. Our work leverages and extends existing work on trace summarization, formal methods for model analysis, specification mining from execution traces, and others, in the context of scenario-based specifications. A key guiding perspective for this research is that interesting properties of a trace often can be associated with one or very few concise scenarios, depicting desired or forbidden behavior, which are already in the specification, or should be added to it.

1 INTRODUCTION

Execution logs of complex systems often contain thousands if not millions of events. Depending on the task at hand, say, debugging an apparent problem, studying existing behavior in preparing for new developments, or making a management decision, extracting from such logs, or traces, just the relevant items can be a difficult and error-prone task. Much work has been done on trace summarization, mining, and more, towards simplifying and accelerating tasks in software and system engineering (SE) that require, or that can take advantage of, execution traces. In this paper we extend this work by observing that the properties that one finds relevant in a given trace, may change depending on the task one is working on, be it helping a customer, debugging a problem, designing a new feature, validation and verification, detecting cyber intrusions, or, demonstrating the capabilities and limitations of a system to new audiences. More generally, we propose to create a systematic arsenal of algorithms, tools, and development methodologies for using event traces in SE.

Consider, for example, the case of a model of a city-wide road system, with many autonomous and human-driven cars, and with automated traffic lights and other controls. Then, during a model-based simulation a human observer looking at a video of the system behavior notes several near-collision situations. The system's event trace, will likely contain a large number of events, including of course all car movements, traffic light changes, raw and event-based sensor data coming in from cameras, range finders and other instruments, as well as high level abstract ones such as cars reaching their intended destinations, cars having negotiated busy intersections successfully, and, sudden queues having been handled successfully. However, in analyzing each of the near-collision situations, especially for the first time, one has to filter out the vast majority of the events in the trace. Moreover, a human may be able to describe the relevant portion of the video, or the trace, which may still be quite large, with very few terms and implicit abstractions, such as: "car C_1 stopped abruptly

because bicycle B_1 was quite fast, and was about to cross in front of C_1 without slowing down; and, car C_2 driving behind C_1 was barely able to brake in time and nearly collided with C_1; further, not only did C_2 not keep a safe distance at that moment, but it has been driving aggressively for some time now; this is interesting because car C_2 seems to be autonomous...".

Our context is the *scenario-based programming* approach (SBP), in which models and even final systems can be developed from components representing different aspects of desired and undesired system behavior. Here, our goal is to assist engineers working on development, debugging or maintenance of SBP models by automating the handling of simulation and execution traces, specifically, the extraction, and subsequent use of succinct sub-traces and relevant abstractions thereof.

In Section 2 we first present a small running example to be used as context for the rest of the paper; in Section 3 we introduce scenario-based modeling and programming; in Section 4 we discuss existing relevant research and tools; in Section 5 we elaborate on the desired capabilities of the proposed tools and methods and, via a few examples and preliminary results, show their application in Section 6; finally, in Section 7 we conclude with a discussion of the results and of the next steps in this research.

2 A RUNNING EXAMPLE

As a running example we use an advanced driver-assistance system using automated car-to-x communication to replace classic traffic control mechanisms such as traffic lights, towards safer and more efficient traffic flow. Fig. 1 shows an example situation in such a system as well as a scenario that would appear in a scenario-based specification or model of that system. Roadworks block one lane of a two-lane road. Cars approach on either lane and need to communicate with the obstacle's controller in order to know what signal (either *Go* or *Stop*) to show to their driver on their dashboards. An example scenario from the system's specification could be that: (1) when a car's sensors register an obstacle coming up ahead (2) the car's driver must be shown a *Go* or a *Stop* signal (3) before the car actually reaches the obstacle.

Even experienced engineers usually need many iterations until a specification is feature-complete and defect-free. Understanding the behavior induced by a specification, including an intuitive scenario-based one, is difficult. Simple mistakes, e.g., forgetting to specify the assumption that drivers obey the signals on the dashboard, can lead to formal methods reporting that violations, e.g., car collisions, are still possible despite the expected outcome being different.

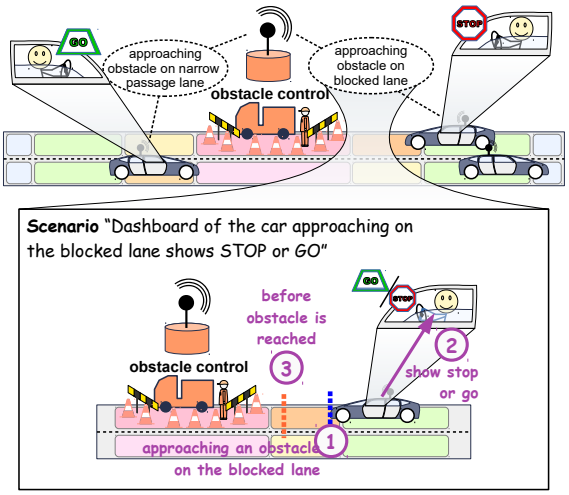

Figure 1: Car-to-X example overview

3 SCENARIO-BASED MODELING

Scenario-based Modeling (and Programming), also termed *behavioral programming*, offers an intuitive approach for writing formal specifications. Short scenarios specify sequences of events that involve multiple objects and that define how objects/components *may*, *must*, or *must not* behave. A collection of these scenarios is a specification which, through the interplay of the contained scenarios, defines the overall behavior of an entire system. Visual and textual formalisms and languages for writing scenarios include Live Sequence Charts (LSCs) (Damm and Harel, 2001; Harel and Marelly, 2003), the Scenario Modeling Language (SML) (Greenyer et al., 2015; Greenyer et al., 2016; Gritzner and Greenyer, 2017), and behavioral programming in general-purpose procedural languages like C++ or Java (Harel et al., 2012). Fig. 2 shows an LSC of the scenario depicted in Fig. 1.

Key to the scenario-based approach is that execution of the specification can be done intuitively using *play-out*, namely concurrent execution of all scenarios, while complying with the constraints and possibilities defined by the entire specification and yielding cohesive system behavior. Another execution method is by synthesizing a composite automaton that reflects the desired behavior of the system under all environment behaviors; in fact, this synthesis can be seen as creating a strategy that guides event selection during play-out. Yet another approach is execution with lookahead, termed *smart play-out*, where the event se-

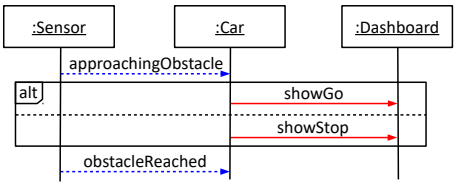

Figure 2: LSC1: The dashboard of car approaching the obstacle must display either "go" or "stop" before the car reaches the obstacle.

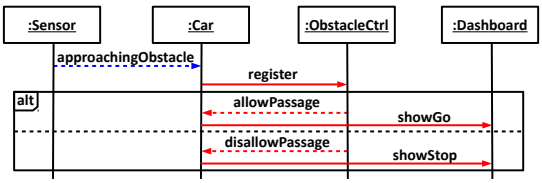

Figure 3: LSC2: A car approaching an obstacle must first register and then wait for a go or stop signal from its dashboard.

lection is subject to run-time assessment of all possible upcoming execution paths, to some limited depth or horizon.

Scenarios consist of events, representing system or environment actions. Scenarios define a partial order of events and modalities encoding what events may, must, or must not occur in each system state. An event may be *requested*, *waited for*, or *blocked*. During play-out, at each state, an event that is requested by some scenario and is not blocked by any scenario is selected for triggering. All scenarios either requesting or waiting for this event are notified and can change their state and optionally change their declarations of requested, blocked, and waited-for events.

Playing-out the scenarios in Figures 2 and 3, after the event *approachingObstacle* both LSCs are active, but the dashboard events *showGo* and *showStop* are blocked due to the order enforced by LSC2. Thus, *register* will be executed next. Depending on the obstacle controller's reply, the car will then update its dashboard appropriately. If a car is able to reach the obstacle before the dashboard shows either *Go* or *Stop* the specification is violated.

The amenability of SBP specifications to incremental refinement is accompanied by their often being under-specified and non-deterministic: depending on the specification, multiple events may be candidates in a given state some of which may be undesirable or even lead to violations. The opposite, not all desirable events are enabled in a given state, may also be true. These situations are indicators for missing features or defects and are vital for engineers to notice and to understand their cause. However, finding and reasoning about such situations is often difficult, especially in large systems.

4 RELATED WORK

Below we give brief examples of the kind of existing research that can be applied ad-hoc in the use of execution traces in the desired SE activities. In Section 5 we explain how our contribution aims to extend these capabilities.

Acting upon emergent properties. Much of the development process, and in particular in agile, incremental methodologies, revolve around observing desired and undesired properties in an existing model, and refining the specification accordingly. Returning to the example in the city-wide traffic automation in the introduction, clearly the human intuition that not only collisions are violations, but near-collisions should be reported and analyzed should be manifested as part of the specification. External sensors, as well as programmed analysis of known and predicted car movements can be used to alert about such risky conditions. The specification should then be enhanced with scenarios that forbid such events from occurring. At run time, these will thus be automatically avoided where possible, and when they nevertheless occur, a violation will be reported. The detection of near-collisions in general traces (depending on velocities and locations) can be specified by engineers and regulators, or can be automatically inferred using machine learning techniques. In (Harel et al., 2016) the authors present an automated approach for detecting emergent properties in sets of execution traces of scenario-based models, and allowing the programmer to determine if they are desired (perhaps so that they should be formally proven), or undesired, in which case the specification should be repaired (manually or automatically).

Trace summarization and analysis. A large variety of techniques for summarizing and abstracting execution traces, especially logs of method calls, has been researched. E.g., in (Hamou-Lhadj and Lethbridge, 2006) the authors present a technique to identify low importance utility method calls by a fan-in/fan-out metric. In (Braun et al., 2015) execution traces are used to automatically generate system documentation via use case maps. The authors describe eight algorithms (some emerging from prior works on the topic) for assigning relevance or importance of methods calls. These algorithms look at call patterns, method size, etc. In other papers, such as (Noda et al., 2017), filtering of events is based on pre-designated or inferred importance of the events themselves or of the objects involved.

While the structured data of a trace can be processed using many classical techniques, including storing in databases and subjecting the information

to database queries, another approach (Bertero et al., 2017), treats the log data as free text and applies natural language processing techniques to summarize the raw data and distill relevant properties thereof.

Causality analysis. In the present context of SBP we relate to causality, especially that of undesired events, as the sequence of events preceding the undesired one, where each one could occur only after one of several explicitly-specified events have occurred (triggered either by the system or the environment). This chain of events can be readily examined in a trace in which the states of all scenarios is known in addition to the identity of the events that occurred. Automated tools for problem detection (and repair) analyze traces that violate the specification or cause a crash. The tools then attempt to detect the unexpected environment event, or the undesired system decision that are the root cause for the violation, and the sequence of events leading from that root cause to the observed failure. The traces containing the problem may emanate from, e.g., execution failures (in the field or during testing) (Weimer et al., 2010), and from counterexamples generated by formal verification (Clarke et al., 2003). In incremental SBP development, when an added specification scenario reflecting a valid user requirement, causes the specification to become non-realizable, the engineers then search for the unrealizable core of the specification. In this context the new scenario can be viewed both as part of the specification and as test run that violates it.

5 PROPOSED METHOD

Below we discuss in detail key elements of a proposed method for working with execution traces in the development and verification of scenario-based models. The method is outlined in Figure 4.

The methodology is to contain the following tool capabilities and human-controlled steps:

5.1 State-graph Generation and Analysis

Our approach assumes that the formal specification is such that one can automatically generate from it a composite state graph. The graph can be comprehensive and cover the entire behavior of the system, or, when violations are found during the generation of the state graph, a partial one can be used. Most commonly the graph will be created by a BFS or DFS play-out of all scenarios in the specification, exploring all possible event selections in each composite state. When a violation or a deadlock occurs during the play-out, the state is marked as bad, or violating. When a cycle is detected, that branch of the search is abandoned. If, in addition, during that cycle there is at least one scenario that requests an event that *must* occur, and that event *does not* occur during the cycle, a safety violation is indicated in the state entering the cycle.

The goal at this stage is limited to finding *at least one* failure which the environment could force on the system. The findings of this step will be the input to the subsequent steps, whose goal is to pinpoint and *understand* the problem towards manual repair by a human engineer (for automated repair of scenario-based programs see, e.g.(Harel et al., 2014). Once such a problem is repaired or bypassed, the process can repeat, starting again with formal analysis.

In a variation on the formal methods approach for obtaining failed executions, one can identify failures by creating a collection of traces using multiple random (possibly parameterized) runs, in which undesired emergent properties are observed automatically (see, e.g. (Harel et al., 2016)), or by a human.

5.2 Generating Sets of Failing Traces

Developers commonly work with one trace at a time, reflecting one way to reach the problem state. The methods we propose enhance this kind of work, and also augment it with tools for working with sets of traces. This adds features that are common across multiple failing traces and behaviors that are unique to certain failing traces to the analysis. Naturally, many failing execution traces come from failed test runs and from problem reports. To these we suggest to add the following. Once a bad system state is identified as described in the formal analysis step, do not suffice with a single counterexample run that violates the specification or manifests some desired behavior. Instead collect *multiple* such paths in the model's state graph.

Depending on the nature of the portion of the state graph leading to the bad state, it may be possible to generate *all* bad execution traces (this is the case in our preliminary results). Alternatively, the number of traces can be reduced systematically using heuristics, such as,

- Partial order reduction: One specifies sets of events the order of which the engineer believes will not affect their understanding of the problem.
- Event filtering: Eliminate from the trace events that are deemed irrelevant. E.g., environment events that are listed and logged but do not affect the outcome. Some of these can be detected automatically, based on the logic of the scenarios

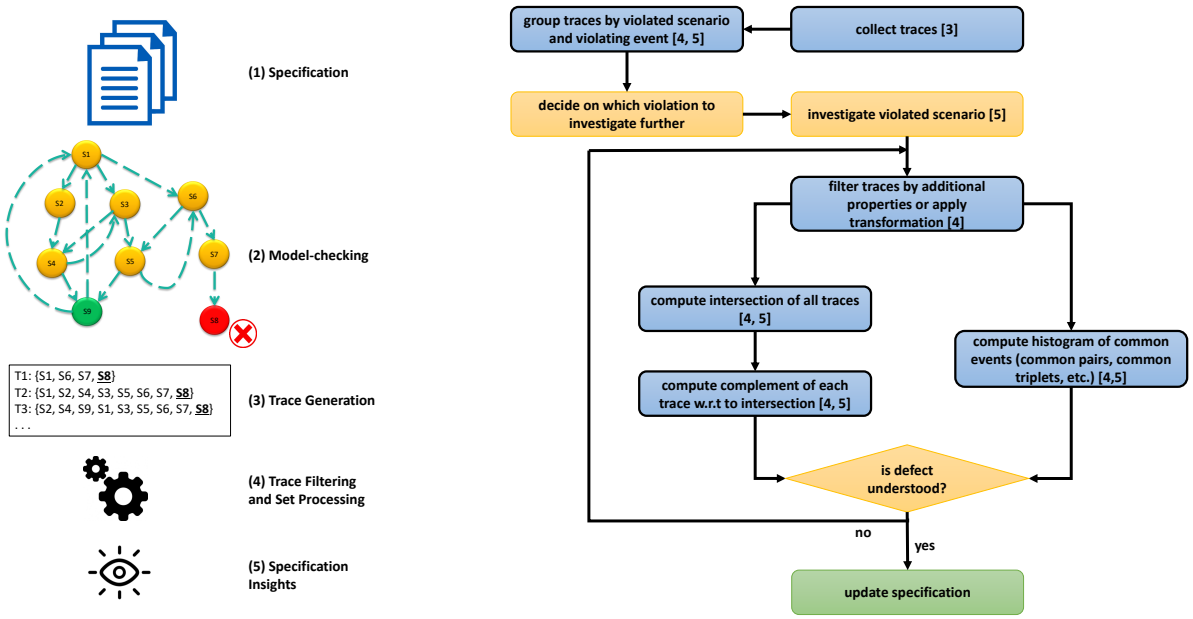

Figure 4: Overview over our proposed method (left-hand side) and example of its systematic application to fix a single defect (right-hand side). The numbers in parenthesis in the example refer to the steps of the overview they represent. In particular steps 4 and 5 are applied iteratively.

involved. Or, one may filter events based on the scenarios that requested them.

- Random trace selection: Create the traces using random sampling of the possible branches in the state graph.

5.3 Enhancing the Traces

Whether in the development lab or in the field, we propose that classical event traces be augmented. In our experiments, we enhanced the classical trace of state labels and transition events with an extensive snapshot including:

- a list of active scenarios (ideally, this would include their respective local states)
- the enabled events (metaphorically, the 'roads not chosen', at any given state), and,
- selected objects (e.g., cars) and their states (i.e., property values).

While such richly-labeled traces can become unwieldy in large systems, we observe and propose that extensive logging can be a game-changer in SE in general, regarding the ability of a system to be adaptive in real time to changes in the environment and to the results of its own behavior (see also (Marron, 2017)). Hence, developing fast automated offline and run-time techniques for compressing and filtering extensive traces would be an important enabler not only for problem detection, as in the present research, but for other purposes as well.

5.4 Ad-hoc Tool Validation

Whenever one relies on a tool to diagnose problem (especially in the case of a new tool, or a first time use by a novice user) there is a need to confirm, time and again, that the tool does not miss important problems, and does not generate false alarms, as may happen due to incorrect application of the tool, major errors in the original specification (e.g., a logically bad assertion that makes all runs into good ones), or bugs in the tool itself.

While SBP offers advantages in incremental development, our preliminary experiments show that it is also advantageous in doing the opposite: incremental removal of features, or isolated insertion of well-specified undesired behaviors. These can then be used to make sure that the tool indeed works as expected.

In the car-to-x SBP model described in Section 2 we have experimentally modified (or have removed altogether) individual specification scenarios (both individually and several together), and checked whether the proposed techniques can help identify the root cause of problems. We propose that when analyzing the root cause of a particular behavior (e.g., a hard-to-solve, hard-to-recreate customer-reported problem), we also modify the specification intentionally to generate similar external symptoms, and keep enhancing

our tools until they are able to automatically detect the new known (synthetic) root cause. Then, we can more safely apply the same tools to the traces from the customer problem at hand.

5.5 A Rich Trace-processing API

In our experiments we externalized a rich and growing library of filtering and validation functions to end-users and to higher-level scripts. This enables the easy creation and modification of chains of trace-processing functions suited to the task at hand. Often, engineers gain additional insights while closely examining a defect, thus they need to be able to easily incorporate these new insights into the trace analysis, both via ad-hoc queries regarding a problem being studies and via automating future trace-set analysis.

The trace-processing tools should also allow engineers to readily incorporate any heuristics they develop, as a method to be readily accessible in all future analyses, for the entire community.

For example, our proof-of-concept APIs include, among others:

- generating all traces induced by a model or specification that exhibit interesting properties, such as:
  - traces which lead from one particular state to another particular state (our experiments are a special case of this: the traces lead from the initial state of the system to a violating state),
  - traces that visit a particular state
  - traces that have certain length, or
  - traces during which a certain scenario is active

  Such APIs enable engineers to describe the case-specific properties that the selected traces should have, e.g., only traces which lead to a collision of two cars, and to collect traces for any interesting observed behavior, even one which does not yet violate the specification.
- filtering a given sets of traces based on the same properties and conditions which can also be used for collecting traces in the first place. This feature enables engineers to quickly modify the set of traces they work with offline without the need to to run full system simulations.
- computing the intersection of traces Given that all traces in the entire set fail in the same or similar way, this may help pinpoint the elements that are essential to the failure.
- compute the complements of such sets (intersections), in search for properties that are unique to individual traces or to particular (sub)sets of traces. By contrast to the previous element, if analyzing commonalities does not help pinpointing the problem, perhaps analyzing the way by which each trace is different from all others (or many others) may provide the desired insights (Tolstoy's Anna Karenina opening sentence "All happy families are alike; every unhappy family is unhappy in its own way." may explain some of the rationale for this approach, though in our case all traces in the set are failed ones).
- retrieving or filtering sets of traces according to properties of the trace itself. A key property which we found useful is the scenario (or scenarios) being violated, and the identity of the violating event. Other properties can include order and correlation of events or properties of individual trace entries.
- quantitative analysis (e.g., producing histograms) of trace properties (within a set of traces) and of entry properties (within a trace or set of traces). This can be helpful for identifying properties that may not occur in all traces but may still occur in many traces and point towards the cause of a problem.
- trace transformation, especially according to specification properties. Traces may contain superfluous information which the engineer may want to remove, e.g., in our car-2-x example, information about a third and fourth car when investigating the collision of two cars, or removing the most frequent or most infrequent events in preparation for quantitative analysis or for anomaly detection.
- specification manipulation API. When narrowing down a problem, it is often nexessary to modify the original specification, such as adding simplifying assumption, removing irrelevant scenarios, etc. One may also mark certain 'bad' states as 'good' in order to bypass certain problems and allow otherwise hard-to-obtain traces to proceed past that point for further analysis. On the one hand, the scenario-based approach simplifies this kind of analysis, allowing the engineers to test a variety of combination of scenarios, adding and removing them to create different specification configuration, in a way that is much easier than is commonly possible with ordinary procedural code, using, say, special test scaffolding, method-call stubs and mock objects. On the other hand, managing manual modification, even when done in a well-controlled source/version management system would be risky, especially as it would be combined by the actual incremental repair. Hence

a specification-control system with allowance of temporary addition and removal of scenario is desired. Such an API can be seen as a partial manifestation of the concept of reactive specification (Marron, 2017), where the specification itself is a reactive system in its own right, adapting, in this particular case, to changes in the requirements, as opposed to changes in the runtime environment.

The right-hand side of Fig. 4 illustrates a systematic process how an engineer could use the methodology and the proposed API to find and fix a defect in a specification. All steps in blue boxes with black borders are steps directly supported by our proposed API whereas all other steps require manual, creative effort by a human engineer. First, the engineers collect all traces from the state graph induced by a scenario-based specification that exhibit some undesirable behavior, e.g., all simple paths leading from the initial state to a state in which any safety violation occurs. As there may be multiple defects in the specification, they then group these traces by violated scenario and violating event (i.e., the last event in each path). After choosing a group to investigate next, based, e.g., on severity of the symptoms) they take a closer look at the violated scenario to find out whether this scenario itself may be wrong. If, as is often the case, the violation is only the symptom of a defect, with the actual cause being somewhere else in the specification, the engineers enter an iterative process in order to gain understanding of the actual cause of the defect. In each iteration they filter and transform the traces from the group being investigated, based on insights on the defect's cause they have already gained so far. For example, irrelevant events such as ones representing the movement of a far-away car when investigating a collision that does not involve it all. Additionally, they filter and query the trace (using the above API), and modify the original specification in various ways to gain additional information and insight.

6 PRELIMINARY RESULTS

6.1 The Specification

In this section we describe an experiment we conducted to develop and exercise the proposed methodology. We also describe additional reflections and insights gained in the process, which have methodological implications for software development and associated tools, that are applicable to a broader context, beyond debugging with sets of traces.

In the experiment, we created an SML specification (Greenyer et al., 2015; Greenyer et al., 2016; Gritzner and Greenyer, 2017) of the example from Section 2. This specification defined the behavior of an obstacle controller which tells approaching cars when they must wait and when they may enter the single available lane, so that cars approaching from opposite directions can pass the obstacle without colliding with each other.

The specification contained three conceptual kinds, or categories, of scenarios, each kind modeling a different aspect of the system, or of the design process. A similar distinction is, of course, applicable in other contexts as well, but the scenario-based approach allows tying it to executable parts of the code. The categories are:

(A) Assumptions about the environment's behavior. E.g., in our case, how the cars and drivers behave, the layout of the road, etc., which are the environment in which the obstacle controller system exists and behaves.

(B) Rules guiding the behavior of the system. E.g., in our case, what the obstacle controller must or must not do following certain events or when certain conditions hold.

(C) Requirements that the final behavior of the obstacle controller must fulfill given its environment. E.g., in our case, that collisions are not allowed. This reflects the purpose of the system and overall constraints not covered by the first two categories.

Note: The general problem in formal program synthesis is to produce category (B) specifications (in the form of an automaton, a procedural program or a scenario-based program) given only category (A) and category (C) specifications. Further the boundary between categories (B) and (C) is not crisply defined, and keeps moving with the development of the ever higher abstractions available in programming languages and in synthesis capabilities.

In the present specific context of scenario-based programs, and in particular, as the execution is often driven by *play-out* rather than by running synthesized programs, the scenarios of category (C) serve multiple purposes: First, they enable underspecification in scenarios of category (B), leaving the system more options to choose from at run time. The category (C) scenarios actively participate in the play-out, and are consulted as guards and constraints during event selection. Second, category (C) scenarios can guide smart play-out (i.e., execution with lookahead) or program synthesis so that the proper choices can be made. Finally, when the category (B) scenarios define the system's behavior fully and deterministically, the

category (C) scenarios serve as a separate, nearly independent alternative specification of the properties that the system should be tested against.

6.2 Inserting Defects

We made the following modifications to the specifications in our experiments:

1. We changed an obstacle controller scenario to have an "off-by-one" error - where when only one car is passing in the narrow area, cars arriving from the other direction are not signaled to stop. When two or more cars occupy the narrow area, the signal works correctly.
2. We removed the (often forgotten) environment assumption that drivers obey the stop/go signal on their dashboard.
3. We omitted the scenario that, as soon as the narrow area becomes free, allows the passing of cars that were previously told to wait.
4. We introduced a copy&paste error in the scenario describing the requirement that car collisions must be avoided.

These modifications reflect commonly made mistakes such as "off-by-one" errors, errors resulting from using copy&paste incorrectly, or forgetting to specify important aspects of the system. The modifications in our experiments also covered all three categories of scenarios.

6.3 Systematic Analysis of Traces to Identify Defects.

We used a prototype of our aforementioned trace-processing tools and API to analyze traces induced by our modified specifications. Initially, we created a version of the specification that contained the first three modifications and collected all traces leading from the initial state of the system to a state violating the specification. This initial set of traces occupied 78MB. It contained about 5000 traces of about 20 events each. The three modifications we picked all represented defects in (A) or (B) kind scenarios.

Clearly one or few of these small traces could have been analyzed manually using traditional techniques, but in our initial experimentation we were able to execute the following partially automated analysis of the entire set as follows. In the first batch of steps:

1. We extracted all traces that lead to a safety violation of the specification.
2. We created a list of all events that trigger a violation.
3. We (manually for now) observed in this list that violations occur upon the event of a car reaching the obstacle or the event of a car passing the obstacle.
4. We used this observation to narrow our set of traces to all those in which the event `carB1.obstacleReached` is the cause of a violation. (such choice can emanate from, say, a customer complaint — that after certain actions certain undesired conditions emerged). This yielded 670 traces, all with the same violated scenario, the one with the self-explanatory name of `CarReceivesAnswerBeforeReachingObstacle`.
5. We checked a failed trace against this scenario and saw that the above event occurred out of order and the expected event (of reaching the obstacle) has not arrived yet at that point.
6. We checked all scenarios which can emit this event. This yielded (in this case) just a single scenario. Finding the "off-by-one" bug in this small scenario was then straightforward.

While some of these steps are similar to classical debugging, one should note that some of the answers apply to a multitude of test runs and not just to one, providing a greater generality to the analysis and to the proposed solution.

We then pursued a second batch of steps as follows:

1. We fixed the bug we have found above.
2. We collected new traces from the specification now presumably containing only defects 2 and 3.
3. We verified that the first problem was indeed fixed, i.e., the previously observed undesirable behavior of having cars reach the obstacle without being signaled whether they should wait or may pass was indeed gone. The events triggering a safety violation now only included events of cars that are trying to pass the obstacle.
4. Again, we narrowed down the set of traces under investigations to those in which a particular event, `carB1.passingObstacle`, triggers the violation. This resulted in a set of 3640 traces.
5. We investigated which scenarios were violated, and found 4200 violations of the scenario specifying that car collisions must be avoided.
6. We inspected these violations and noticed that `carB1` can collide with two other cars, `carA1` and `carA2`. Sometimes even with both at the same time, thus triggering two violations with one event.

7. We we went back to the set of traces under investigation and used our API to filter it some more. We removed all traces that include `carA2` to focus on collisions between the other two cars.
8. We used the intersection feature of our API to find the common behavior among all these remaining traces and found the following:

```
env -> carA1.approachingObstacle()
carA1 -> obstacle.register()
env -> carB1.approachingObstacle()
carB1 -> obstacle.register()
env -> carA1.passingObstacle()
env -> carB1.passingObstacle()
```

   and two kinds of complements of the intersection, namely six traces containing

```
obstacle -> carA1.allowPassage()
                          // may pass
obstacle -> carB1.disallowPassage()
                          // must wait
```

   and four traces containing

```
obstacle -> carB1.allowPassage()
obstacle -> carA1.disallowPassage()
```

   This suggested that the cars simply ignored the signals they were sent. Both cars tried to pass the obstacle at the same time despite one car having received a wait signal.
9. We examined the entire specification looking for a specification that forbids events associated with a car's progression between getting events associated with disallowing passage and events associated with allowing passage. We could not find any such scenarios (granted, one does not always know *exactly* what one is looking for in such cases as some of the desired effect may be only indirectly implied from well-thought-out specifications). We concluded that a specification of the assumption that cars obey the signals that are sent to them was missing.

We were able to reach the same conclusion about the root cause of the problem (cars ignore the signals from the obstacle controller) using our API also with different analysis steps, this time quantitative ones, as follows:

1. We created a histogram of the number of occurrences of all pairs and triplets of consecutive events in all the traces.
2. We observed that the triplet (and its analogue for the other cars)

```
obstacle->carB1.disallowPassage()
env->carB1.obstacleReached()
env->carB1.passingObstacle()
```

   occurred many times.
3. In a slightly different quantitative analysis, using our API to create three transformations of each trace, each transformation only containing the events involving one particular car, we could show that each trace contained at least one offending triplet, i.e., at least one car ignored its wait signal.

We then fixed the problem we just identified, i.e., we added (back) the assumption that cars obey the signals, and proceeded with the partially-automated analysis as follows:

1. We used the API to generate the state graph and look for violations. This showed that there are no safety violations, only liveness violations. The specific property that was violated was that each car eventually goes past the narrow area.
2. We collected a set of traces leading to the liveness violation.
3. After several filtering operations similar to the above, we observed that the last event received by `carB1` prior to entering a cycle in which a car never goes past the narrow area, is `carB1.disallowPassage()`, and that no `allowPassage()` is sent to it after that event, despite all cars that drive in the opposite direction being conspicuously past the narrow area (e.g., the location of `carA1` is `BehindObstacle`).

In this experiment we showed that using multiple traces instead of just one single example, can significantly help the process of understanding of the cause of a problem. Note that while the observed violations were all in the requirements scenarios, (i.e., some property encoded in a category (C) scenario did not hold), the actual causes of the problems were elsewhere, and trace analysis helped to understand and pinpoint the causes, even though the problematic sequence of events may have occurred much earlier than the actual violation. Even in cases in which entire scenarios where missing this approach worked well and it works for both safety and liveness violations.

6.4 Defects in the Requirements

During this first experiment we noticed that all three mutations of our specifications relied on the presence of category (C) scenarios. The latter forced the defective category (A) and (B) scenarios to cause violations. This leads to two questions:

1. How should one handle user reports of an issue that does not cause a violation (e.g., because the respective category (C) scenario encoding the user's expectation is missing)?

2. What happens if there is a defect in one of the category (C) scenarios?

One answer to the first question is that the proposed API needs to support the collection of traces fulfilling arbitrary user-defined properties. Collecting all traces from the initial state to a violating state is a special case of an engineer wanting to inspect all traces which lead from one specific state in which the system is still known to be in a good state, to another specific state, where the engineers know the target state or paths between the two are undesirable. Our API now includes support for such trace collection request. Extending this to sets of source states and sets of target states, as defined by some properties is left as future work.

The answer to the second question has two parts. First, if the defective category (C) scenario is indeed violated by an otherwise valid trace, this can be discovered immediately when examining the violated scenario upon encountering such a violation. In scenario-based specifications, where program modules are usually short and refer to one sentence or a short paragraph in the requirements as conceived by stakeholders and engineers, it is often easy to notice the error in the way a requirement is specified which causes it to be unduly violated. This became apparent in second, smaller-scale iteration of our experiment in which we tried to apply our proposed methodology to a mutated version of our example specification which contained the fourth defect of those listed in Sect. 6.2. Throughout our experiments, when intended and unintended defects caused the defective scenario itself to manifest a violation, discovering and fixing the defect was straightforward.

In order to be able to discover problems in category (C) scenarios which are not manifested by their violation, we propose a somewhat different approach. For each category (C) scenario, one should prepare a setup that violates it. For example, such a setup can be a single test scenario that requests a sequence of events that should violate the property at hand. As part of initial testing (perhaps immediately when the category (C) scenario for the property is first written) any other existing scenarios which may prevent the violation should be disabled or removed and the test scenario be executed alongside with the tested property. Then one should confirm that either the property is violated, or that the property scenario causes the test scenario to be stuck, or to terminate prematurely.

Thus, the main strength of the trace analysis approach described in Section 5 lies in helping to identify defects in which the cause lies at a different point in time than the observed undesirable behavior.

6.5 Support for Demonstrating Relevant Properties

The debugging tools and methodology presented above, can also be used for a different purpose: demonstrating desired properties in specifications and sets of traces. Consider for example a request by a reviewer of the system to see a demonstration 'proving' that the obstacle controller sends a `disallowPassage` signal to approaching cars whenever there are other cars occupying the narrow area in the opposite direction. We assume also that this has not been and cannot be added as a separate category (C) scenario, and must be implied by other, existing scenarios. Note also that a straightforward formal verification that this property holds in the specification may even be misleading, e.g., if due to other modeling errors it turns out that cars rarely, or even never, arrive at the obstacle from opposite directions at the same time. Of course, a single test run would be a nice, but insufficient demonstration. This is what often happens in real system demos. However, a more powerful demonstration would be to show statistics indicating that in an extensive collection of runs, triples like

```
env->carA1.passingObstacle()
obstacle->carB1.disallowPassage()
env->carB1.obstacleReached()
```

occurred thousands of times, repeatedly, and in distinct traces. This feature of the entire approach also serves as a reminder that a particular trace or set thereof may possess multiple relevant properties, and engineers may be interested in different properties at different times. E.g., during our analysis of collisions in pursuing the second defect, the automated trace analysis informed us, without us asking explicitly, that the obstacle sent the required signals correctly in all possible runs.

7 DISCUSSION AND FUTURE WORK

We have presented our direction towards a systematic approach for management, summarization, analysis and querying of large sets of large execution traces of SBP models, and have shown preliminary results how such tools can accelerate causal analysis, debugging and maintenance.

A more systematic evaluation of the advantages of such tools over manual techniques can motivate and guide the particular areas that should be further developed.

For example, the approach can be enhanced via

richer queries on traces, scenarios and system states. E.g., "what are the scenarios which request other enabled events when event E1 was selected (in traces in the current set), and were these event requests ever granted, or did the scenarios transition out of that state due to other events that occurred?". Also, queries regarding strategies may help engineers: some undesirable behavior may be avoidable by the system what it follows a certain strategy (if it has enough degrees of freedom to do so). In such cases engineers may want to know whether some behavior is avoidable (i.e., if a variant of the strategy exists), and if it is, under what circumstances this will be possible.

In particular we would be interested in causality queries, such as "given a violation, find the sequence of events that directly caused the triggering of the last event". In other words, going backwards, for each triggered system event, what are all the scenarios that requested it at that state (system cut); what was the preceding event in each of these scenarios; and then, repeat the process for each of these events. In fact, this should be augmented with researching the events that were blocked in those states, and how the scenarios that blocked them have reached those particular states. While this chain of analysis may be large, recall that it filters out all the events that are not in this causal chain, and are merely the result of parallel processes.

One can automate certain aspects of liveness-property analysis in traces (when comprehensive formal verification is not possible), based on the fact that scenarios distinguish events that *must happen* from those that 'just' *may happen*, at a given state in a scenario. Hence the specification and traces can guide the discovery of situations where scenarios wait for an extended period of time for events that were marked as *must happen*, as well as the causality chains which may have been broken.

Another area of intriguing research opportunity is automating (or, at least, methodologically prescribing) the steps in the method that presently depend on human decision and intuition.

The enrichment of the log with object data can help analyze complex problems. For example, it seems that only a few additional details, like time and certain car properties, and a small amount of domain knowledge (to be captured as additional assumption scenarios), should be needed in further automating the analysis of near-collisions described in Section 1. We would expect the computer to be be able to reach complex observations like: (i) "Car C_2 was actually an ambulance on an emergency call with a siren and lights on" (hence its driving aggressively may be acceptable); (ii) "the event of car C_1 pulling over to the side to make way for C_2 is missing"; and (iii) "C_1 is not at fault as the ambulance has just turned into the street in which C_1 was driving and there was not enough time for C_1 to pull over before the bicycle crossed its path.".

Another dimension in which this work should be extended is to create generalized behavioral summaries which transcend specification scenarios and individual trace summaries. E.g. we would like to find a formal, concise representation for SE knowledge as contained in natural language sentences like: "presently, always, (as opposed to 'it happened once') when the user presses the green button the buzzer sounds, but instead, the green light should go on", or "the users could not complete their desired action of pressing buttons B1, B2, B3, B4 in this order, because, always after they pressed button B2, button B3 was disabled". Such formalization capabilities would enable deeper analysis and perhaps streamline the automation and complex development tasks such as feature analysis, problem determination, and professional interaction with customers.

ACKNOWLEDGEMENTS

This work has been funded in part by grants from the German-Israeli Foundation for Scientific Research and Development (GIF), from the Minerva foundation, and from the Israel Science Foundation (ISF).

REFERENCES

Bertero, C., Roy, M., Sauvanaud, C., and Trédan, G. (2017). Experience Report: Log Mining using Natural Language Processing and Application to Anomaly Detection. In *28th International Symposium on Software Reliability Engineering (ISSRE)*.

Braun, E., Amyot, D., and Lethbridge, T. (2015). Generating Software Documentation in Use Case Maps from Filtered Execution Traces. In *International SDL Forum*, pages 177–192. Springer.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2003). Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794.

Damm, W. and Harel, D. (2001). LSCs: Breathing life into message sequence charts. In *Formal Methods in System Design*, volume 19, pages 45–80.

Greenyer, J., Gritzner, D., Gutjahr, T., Duente, T., Dulle, S., Deppe, F.-D., Glade, N., Hilbich, M., Koenig, F., Luennemann, J., Prenner, N., Raetz, K., Schnelle, T., Singer, M., Tempelmeier, N., and Voges, R. (2015).

Scenarios@run.time – Distributed Execution of Specifications on IoT-Connected Robots. In *10th Int. Workshop on Models@Run.Time (MRT), co-located with MODELS 2015*, CEUR Workshop Proceedings.

Greenyer, J., Gritzner, D., Katz, G., and Marron, A. (2016). Scenario-Based Modeling and Synthesis for Reactive Systems with Dynamic System Structure in ScenarioTools. In *Proceedings of the MoDELS 2016 Demo and Poster Sessions, co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*. CEUR.

Gritzner, D. and Greenyer, J. (2017). Controller Synthesis and PCL Code Generation from Scenario-based GR(1) Robot Specifications. In *Proceedings of the 4th Workshop on Model-Driven Robot Software Engineering (MORSE 2017), co-located with Software Technologies: Applications and Foundations (STAF)*.

Hamou-Lhadj, A. and Lethbridge, T. (2006). Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *14th IEEE International Conference on Program Comprehension (ICPC)*, pages 181–190. IEEE.

Harel, D., Katz, G., Marelly, R., and Marron, A. (2016). An Initial Wise Development Environment for Behavioral Models. In *Proc. 4th Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 600–612.

Harel, D., Katz, G., Marron, A., and Weiss, G. (2014). Non-Intrusive Repair of Safety and Liveness Violations in Reactive Programs. *Transactions on Computational Collective Intelligence (TCCI)*, 16:1–33.

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

Harel, D., Marron, A., and Weiss, G. (2012). Behavioral Programming. *Comm. of the ACM*, 55(7).

Marron, A. (2017). A Reactive Specification Formalism for Enhancing System Development, Analysis and Adaptivity. In *15th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMCODE)*.

Noda, K., Kobayashi, T., Toda, T., and Atsumi, N. (2017). Identifying Core Objects for Trace Summarization Using Reference Relations and Access Analysis. In *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual*. IEEE.

Weimer, W., Forrest, S., Le Goues, C., and Nguyen, T. (2010). Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116.