Towards Systematic and Automatic Handling of Execution Traces Associated with 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, via a few examples and preliminary results, we elaborate on the desired capabilities of the proposed tools and methods; and, in Section 6 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.

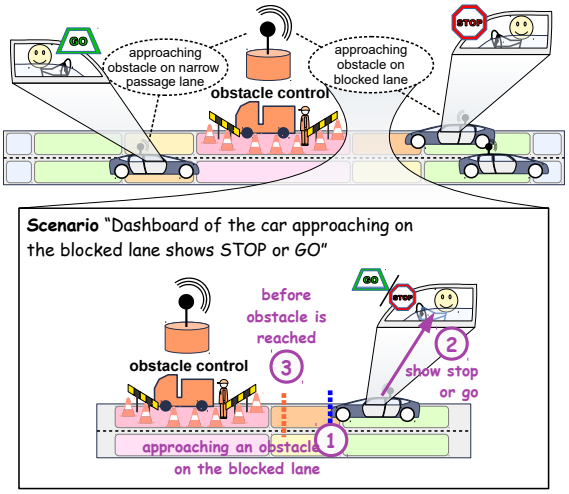

Figure 1: Car-to-X example overview

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 selection is subject to run-time assessment of all possi-

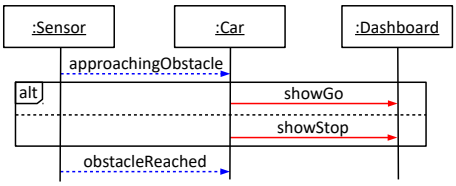

Figure 2: LSC1: The dashboard of car approaching the obstacle must display either "go" or "stop" before the car reaches the obstacle.

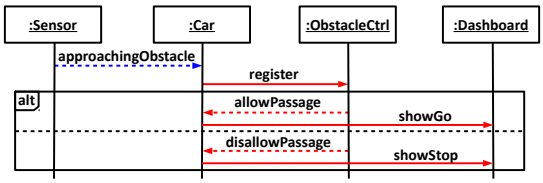

Figure 3: LSC2: A car approaching an obstacle must first register and then wait for a go or stop signal from its dashboard.

ble 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 AND PRELIMINARY RESULTS

The methodology we are developing for working with execution traces should contain the following elements:

Working with sets of traces. Developers commonly work with one trace at a time. The methods we propose enhance this kind of work, but also augment it with tools for working with sets of traces, adding to the considerations the analysis of common features and of behaviors that are unique to certain traces. As for generating these sets, naturally, many interesting execution traces come from test runs, especially failed ones, and from problem reports. To these we suggest to add at least two variations: (a) collection of traces emanating from random (possibly parameterized) runs (see, e.g. (Harel et al., 2016)), and (b) during model checking, do not suffice with a single counterexample run that violates the specification or manifests some desired behavior, but instead collect *all* such paths in the model's state graph (or a manageable subset thereof). In our current experiments we have enhanced one of the SBP synthesis algorithms to generate and collect all such paths.

Enhanced 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 states 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 traces can become unwieldy in large systems, we observe and propose that extensive logging can be a game-changer in system real time adaptivity a SE in general (see also (Marron, 2017)), and developing fast automated offline and run-time techniques for compressing and filtering such traces would be an important enabler.

Ad-hoc tool validation. 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. 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. Specifically, in our experiments we have modified the specification as follows:

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. In fact we experimented with affecting one, two, or all drivers in this manner.
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.

A rich, dynamic and open trace-processing

API. In our experiments we externalized to end-users and to higher-level scripts a rich and growing library of filtering and validation functions. The trace-processing tools should allow engineer 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:

- extracting (from a set of traces) all those with safety violations, and all those exhibiting liveness 'violation' within the trace, as well as the respective violated scenarios
- finding properties that are common to sets of traces or sub-traces, by computing their intersection; additionally, compute the complements of such sets, in search for properties that are unique to individual traces or to particular (sub)sets of traces
- filtering sets of traces according to trace properties
- filtering a trace according to entry properties
- a variety of queries on trace data
- trace transformation, especially according to specification properties
- finding a first or a last entry with a particular property in a trace, and
- 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).

For example in our analysis of the set of traces with all three defects, the initial set of traces occupied 78MB. It contained about 5000 traces of about 20 events each. Clearly one or few of these small traces could have been analyzed manually using traditional techniques, but in our initial experimentation (to be elaborated in future work) we were able to program the following automated analysis of the entire set as follows: we extracted all traces that lead to a safety violation of the specification; we create a list of all events which trigger a violation. 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. 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`. Checking a failed trace against this scenario we see that the above event occurred out of order and the expected event (of reaching the obstacle) has not arrived yet at that point. Checking all scenarios which can emit this event yielded (in this case) just a single one, and finding the bug in this small scenario was then straightforward. Again, 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 one, providing a greater generality to the analysis and to the proposed solution.

It should be noted that intersection of traces refers to event sequences and not just to event sets. Consider our analysis of the second defect we injected. This defect caused car collision in the narrow passage next to the obstacle to occur. As the intersection of violating traces we obtained the following sub-trace (shown here in text, with the sending and receiving lifelines and the event method name):

```
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()
```

Which suggested that indeed the drivers were not obeying the signals.

The analysis of the liveness violation in the third injected defect highlights the role of object data. After several filtering operations similar to the above, we observe that the last event received (earlier) by `carB1` is `carB1.disallowPassage()`, and that no `allowPassage()` was sent to it, despite all cars that drive in the opposite direction being conspicuously past the narrow area (e.g., the location of `carA1` is `BehindObstacle`).

Quantitative analysis showed its value as well. While we knew what we were looking for, it was still interesting to see certain suspicious pairs or triples of events occurring, in this order, but not necessarily purely consecutively, in large numbers of problematic traces. E.g., in traces where collisions occurred due to the second defect, the pair

```
obstacle->carB1.allowPassage();
env->carB1.passingObstacle()
```

and the triple

```
obstacle->carB1.disallowPassage();
```

```
env->carB1.obstacleReached();
env->carB1.passingObstacle()
```
occur thousands of times, indicating that `carB1` may not be obeying the signal sent to it.

Support for demonstrating relevant properties. We propose to give a particular emphasis to demonstrating desired properties in specifications and sets of traces. Consider for example the requirement that the obstacle signal approaching cars to wait whenever there are other cars in the opposite direction occupying the narrow area. Of course, a single test would be a nice, but insufficient demonstration. A straightforward formal verification of this specification property may 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. Results indicating that the following triples occurred thousands of times, repeatedly, and in distinct traces

```
obstacle->carA1.allowPassage();
env->carB1.obstacleReached();
obstacle->carB1.disallowPassage()
```
indeed contributes substantially to demonstrating the desired property. This 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.

6 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?".

In particular we would be interested in causality queries, such "starting with 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, 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 user could not complete his desired action of

pressing buttons B1, B2, B3, B4 in this order, because, always after one presses button B2, button B3 is 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) 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. 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.