Evaluating a Formal Scenario-Based Method for the Requirements Analysis in Automotive Software Engineering

Joel Greenyer and Maximilian Haase
Software Engineering Group
Leibniz Universität Hannover
Welfengarten 1, 30167 Hannover, Germany
greenyer@inf.uni-hannover.de,
maximilian.haase@stud.uni-hannover.de

Jörg Marhenke and Rene Bellmer
IAV GmbH Gifhorn
Rockwellstraße 16,
38518 Gifhorn, Germany
dr.joerg.marhenke@iav.de,
rene.bellmer@iav.de

ABSTRACT

Automotive software systems often consist of multiple reactive components that must satisfy complex and safety-critical requirements. In automotive projects, the requirements are usually documented informally and are reviewed manually; this regularly causes inconsistencies to remain hidden until the integration phase, where their repair requires costly iterations. We therefore seek methods for the early automated requirement analysis and evaluated the scenario-based specification approach based on LSCs/MSDs; it promises to support an incremental and precise specification of requirements, and offers automated analysis through scenario execution and formal realizability checking. In a case study, we used ScenarioTools to model and analyze the requirements of a software to control a high-voltage coupling for electric vehicles. Our example contained 36 requirements and assumptions that we could successfully formalize, and we could successfully find specification defects by automated realizability checking. In this paper, we report on lessons learned, tool and method extensions we have introduced, and open challenges.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Design, Languages, Verification

Keywords

Automotive Software, Reactive Systems, Requirements Analysis, Modal Sequence Diagrams, Realizability

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2804432

1. INTRODUCTION

The increasingly advanced functions in today's cars are mainly realized by software. Automotive software systems are complex embedded reactive systems with multiple software components, sometimes distributed over several hardware nodes, that must often handle concurrent sensor inputs and user requests. Often, the software functions are safety-critical and, moreover, the software usually controls electromechanical components, which requires interdisciplinary expertise during its development.

The software development in the automotive industry follows the Automotive SPICE reference process model, an automotive variant of the ISO/IEC 15504. It prescribes a software requirements analysis where software requirements are analyzed for technical feasibility and testability before implementation. In automotive projects, the software requirements are typically documented informally or semi-formally (for example by state diagrams). The requirements are then analyzed by manual reviews. In this process, however, requirement specification inconsistencies regularly remain hidden and are then only discovered in later development phases, where their repair requires costly iterations.

At IAV, we therefore seek automated techniques that support us in the requirements analysis. In a case study in collaboration with the software engineering group in Hanover, we tested and evaluated the scenario-based specification approach based on Live Sequence Charts (LSCs) [3, 8] resp. Modal Sequence Diagrams (MSDs) [7], a recent variant of LSCs. LSCs/MSDs allow us to model formally how a system may, must, or must not react to external events.

Within the Eclipse-based ScenarioTools tool suite [16], MSDs can be modeled as lightweight extension of UML, using the Papyrus UML editor. ScenarioTools supports the modeling of software requirements as well as the modeling of environment assumptions [2]. For analysis, ScenarioTools implements the play-out algorithm for executing the scenarios [8, 2] and it can perform a thorough realizability checking of the MSD specification [5].

Realizability checking, intuitively, analyzes whether there exists a software controller that can react to all possible sequences of external events in a way that the specification is satisfied. It does so by viewing the interaction of the software and its environment as a two-player game where the software tries to satisfy the specification while the environ-

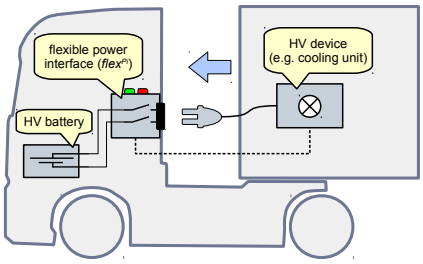

Figure 1: *flex*^{Pi} connects a cooling trailer to a hybrid truck

ment tries everything to violate it. If there exists a strategy for the system to win the game, this strategy can serve as an implementation of the specification, which is thereby proven realizable. If the software cannot keep the environment from winning the game, the specification is unrealizable; it means that the software can be forced into a state where it will consequently violate contradicting requirements.

Our case study was the software for controlling a high-voltage power interface for electric or hybrid vehicles. Electrified vehicles are equipped with a high-voltage system (400-600 Volts) that could serve as an energy supply for high power devices. For example, as shown in Fig. 1, customers would like to plug the cooling unit of a cooling trailer to the high-voltage system of a hybrid truck. Therefore IAV's High-Voltage & Components department invented the *flex*^{Pi} (flexible Power interface) [9]. Connecting and disconnecting high voltage devices is not as harmless as it is with 12/24V devices; disconnecting the plug under load causes an electric arc that can cause deadly injuries. Therefore, a software system is required for ensuring a safe operation of the interface. The software must for example lock the plug in its socket before the relays are closed.

The *flex*^{Pi} system is an IAV internal prototype project. A specification of that project consisted of 120 requirements. A student involved in this project, who had extensive knowledge of the electromechanical and software parts, was then asked to model 36 requirements[1] that made up the core functionality of the system as MSDs with ScenarioTools.

This study showed a high practical potential of the LSC/MSD-based specification approach. Especially, using the realizability checking feature, we could successfully uncover specification defects.

In this paper, we show a brief example and reflect on the lessons learned. Specifically, we will summarize where we see the main benefits of the approach, how automated analysis techniques helped, and what obstacles we think keep this approach from being applied in practice.

Structure: We first report on related work in Sect. 2. We then show by a small example how requirements from the *flex*^{Pi} project were formalized using MSDs in Sect. 3. In Sect. 4 we show two of the specification defects found. Last, we reflect on the lessons learned in Sect. 5.

[1]The remaining requirements were related to dealing with faults of relays, the interlock system, or overheating. We did not cover those in the limited time of the study, which was conducted as part of the Bachelor thesis project of the second author [6]. We are, unfortunately, not allowed to disclose all the requirements; a subset of requirements can be found in the thesis document [6] (in German).

2. RELATED WORK

The application of formal methods in automotive software engineering has been studied previously (for example [10, 4, 13]), mostly focusing on formal verification of designs *against* specifications. These modeling and verification techniques can also uncover specification defects: if verification fails, but the design looks good, the flaw may after all be in the specification. In this paper, however, we instead focus on analyzing specifications for inconsistencies *before* attempting to conceive a design. We can therefore detect problems earlier in the development process.

Post et al. [15] report a study at BOSCH on formalizing requirements in the automotive domain using natural language patterns [11] that can be mapped to temporal logics. They also propose a tool for finding unsatisfiable time constraints [14]. They, however, do not check the specification realizability in an open system setting as in our case.

Löffler et al. [12] evaluate formal scenario-based requirements specification in healthcare applications. In this work, however, a variant of Sequence Diagrams is used where explicit control-flow constructs determine the flow of events through the scenarios. MSDs, by contrast, are activated, progress, and synchronize through common events, which allows for a more flexible modeling of concurrent tasks.

3. REQUIREMENTS FORMALIZATION

In the following we describe a set of requirements from the *flex*^{Pi} project and their formalization using MSDs.

Unplugging the *flex*^{Pi} under load can cause fatal injuries. The connection can be under load only if the relays are closed. So the interface must prevent the plug from being pulled out of the socket if the relays are closed. This is formulated by the requirement unplug forbidden shown in Table 1. The socket is equipped with a locking mechanism that can prevent the plug from being unplugged. This plug locks automatically when the plug is plugged in (lock plug).

If the plug is not locked, the relays must not make contact (make contact forbidden); this is to prevent a current from flowing if a user inserts a screwdriver, for example.

The last requirement (start-button pressed) says that if the start button is pressed, the relays must close the contact.

Table 1: Example requirements of the case study

name	requirement.
unplug forbidden	If a relay is closed, the plug must not be unplugged.
lock plug	If the plug is plugged into the socket, then the socket must lock the plug.
make contact forbidden	If the plug is not plugged or not locked then the relays must not make contact.
start-button pressed	If the start-button is pressed, then the relays have to make contact.

Underlying the requirements is a description of the system component architecture, naming the software components of the system, their state variables and events they can exchange with the environment. This is also the starting point for the formalization that is explained in the following.

We model the system structure using a composite structure diagram (top right of Fig. 2). The controller, socket, and relay are software objects; the two latter being software representatives of the physical socket and relay. The start button and environment are environment objects.

First, we model the requirement lock plug by the MSDs PlugIn and IfPluggedThenLock (see Fig. 2). The MSD PlugIn says that if an external `plugIn` event is detected by the socket, then the controller must be informed. Messages in MSDs have different modalities. The message `setPlugged(true)`, for example, is *hot* and *executed*, which means that the message must be sent. For details on the MSD semantics, see [7, 2, 5]. Messages of the form `set`⟨attribute-name⟩ messages also change the corresponding attributes of the receiving objects; `setPlugged(true)` therefore sets the attribute c.plugged to true. Upon plugging in, the socket must be locked. This is modeled by the MSD IfPluggedThenLock. Upon a successful locking of the socket, the attribute s.locked will be set to true; we omit the corresponding part of the specification for brevity.

Following a similar pattern, the requirement start-button pressed is modeled by the MSD IfStartPressedThenCloseContact. Again, we omit the part of the specification that sets the attribute r.contactClosed to true if the relay's contact was successfully closed.

The requirement unplug forbidden is modeled by an *anti-scenario*: IfRelayClosedUnplugForbbidden says that if `unplug` occurs and the relay's contact is closed, then this will be a forbidden violation of the specification.

Similarly, we model the requirement make contact forbidden by the MSD IfNotPluggedAndLockedThenCloseRelayForbidden; if `closeContact` occurs but the plug is not plugged in or the socket is not locked, this is a violation.

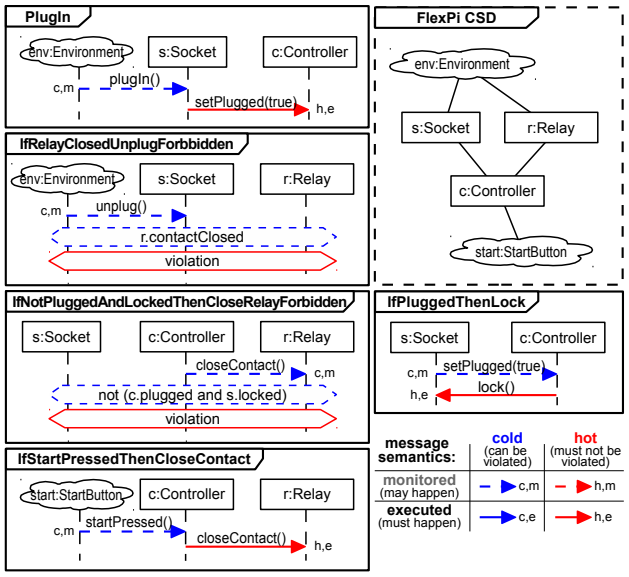

Figure 2: Some MSDs from the specification of the high-voltage coupling system

4. ANALYZING THE SPECIFICATION

Based on a specification as shown above, we can use the ScenarioTools' realizability checking feature [5] to check whether a software implementation exists that can react to all environment events in a way that the specification is satisfied, i.e., that no violations ever occur and all events required to occur do in fact eventually occur.

We could automatically detect two defects present in the above specification:

Defect 1: The MSD IfStartPressedThenCloseContact specifies that if the user presses the start button, the relay should close the contact. This is true also if the plug is not inserted or the socket is not locked. This again leads to a violation of the MSD IfNotPluggedAndLockedThenCloseRelayForbidden.

Raising this issue to the author of the requirements specification revealed that, implicitly, the requirement start-button pressed was meant to have a lower priority, and should only hold when none of the safety-related requirements demand otherwise. To correct the defect in the MSD specification, we added the condition "c.plugged and s.locked" between the two messages in the MSD IfStartPressedThenCloseContact.

Defect 2: In our example, the plug can be inserted into the socket and then the socket will be locked as well. Then, upon pressing the start button, the relays can be closed safely. Now, however, the specification allows the external event `unplug` to occur, leading to a violation of the MSD IfRelayClosedUnplugForbbidden. Here we have a case of an *implicit assumption* made by the requirements engineer, namely that the plug cannot be unplugged as long as the lock is engaged. To capture this assumption, we added an *assumption MSD*, modeled as an anti-scenario, that `unplug` while the socket is closed will lead to the violation of the assumptions, effectively meaning that such a behavior will not occur.

5. LESSONS LEARNED

What were the main benefits of the scenario-based specification and analysis approach? We could successfully model the core requirements of the *flex*^{Pi} system using MSDs. Where the requirements were specified atomically and following the "if ⟨precondition⟩ then (not) ⟨post-condition⟩" pattern, usually a one-to-one mapping of the informal requirements to MSDs was possible. The MSD specification contains more detail (extra MSDs with set- messages) where the informal requirements made implicit use of state variables, such as remembering that the socket is locked.

Some requirements that we encountered were not atomic and ambiguous. Here, the student, with his knowledge of the system, could successfully refine the specification and add the necessary precision.—This is where we see one of the main benefits of a formal approach: it forces requirements engineers to specify precisely.

The other main benefit of the approach we see in the possibility of automatic realizability checking. It successfully supported us in detecting specification defects as we introduced them. As discussed in Sect. 4, we could successfully detect inconsistencies (implicit priorities) in the specification. The realizability checking procedure pessimistically analyzes every possible sequence of environment events, which is impossible in a manual review even of small specifications.

Furthermore, we could successfully detect implicit assumptions in the specification, as discussed in Sect. 4. Often, there is not sufficient knowledge and communication about such hidden assumptions between experts from different disciplines. Especially dangerous is over-optimistically disregarding a possible environment behavior that can lead to a safety-critical violation.

We also see the potential for using the formal specification for example for automatically generating tests from the specification, but we didn't elaborate this direction yet.

How did automated analysis techniques help? During specification, we applied the realizability checking function of ScenarioTools after every specification increment. If the specification is unrealizable, the tool generates a labeled transition system that contains the information on how a particular sequence of environment events inevitably leads to a violation of the specification. This information was very helpful for understanding and resolving the specification defects that we had introduced. However, the transition system can become very large; in our case, it contained over 400 states. ScenarioTools can generate a visual output, but it becomes hard to read already when it contains 100 states or more.

We therefore introduced an extension to ScenarioTools. First, we reduced the complexity of the graph by only showing the environment events. This way, the engineer can see which sequence of environment events leads to a violation, but does not see the system steps. The system steps between environment events are then extracted to further, separate diagrams that the engineer can open on demand to understand the behavior including the system steps in more detail.

We also made use of the play-out algorithm to execute the scenarios in a step-by-step fashion. This helped especially in understanding specification defects and demonstrate them to others (requirements engineers or customers)

What are the main obstacles keeping this approach from being applied in practice? First, we must often deal with timing requirements, which can be specified and played-out with ScenarioTools [1], but currently cannot be checked for realizability.

Furthermore, in our experience, editing MSDs with the Eclipse Papyrus UML editor was very time consuming. For an industrial application, we would favor a textual specification language, which has benefits in version control and easier overall editing. Also, ScenarioTools is an academic tool that is not sufficiently user friendly and self-explanatory for industrial application.

For larger specifications, understanding specification defects will become harder. Here it must be investigated if there exist methods that support the engineer and which scale with the specification complexity.

We could very well imagine to use the approach in internal projects. In projects where customer requirements may not be in a shape that can be easily formalized with MSDs, the payoff of a formal specification approach must still be assessed. Being able to discuss with the customer defects in their specification on the basis of a scenario-based specification sounds intriguing. However, it could be that the effort of formalizing the specification is higher than finding defects in classical requirements analysis (by reviews) or the later development (even if this may require iterations).

6. REFERENCES

[1] C. Brenner, J. Greenyer, J. Holtmann, G. Liebel, G. Stieglbauer, and M. Tichy. ScenarioTools real-time play-out for test sequence validation in an automotive case study. In *Proc. 13th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2014)*, volume 67. EASST, 2014.

[2] C. Brenner, J. Greenyer, and V. Panzica La Manna. The ScenarioTools play-out of modal sequence diagram specifications with environment assumptions. In *Proc.12th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013)*, volume 58. EASST, 2013.

[3] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. In *Formal Methods in System Design*, volume 19, pages 45–80. Kluwer Academic Publishers, 2001.

[4] Q. Fang and C. Zhang. Use of formal method in construting safey-critical automotive software component. In *Software Engineering and Service Science (ICSESS), 2014 5th IEEE Int. Conf. on*, pages 70–76, June 2014.

[5] J. Greenyer, C. Brenner, M. Cordy, P. Heymans, and E. Gressi. Incrementally synthesizing controllers from scenario-based product line specifications. In *Proc. 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering 2013*, 2013.

[6] M. Haase. Erprobung einer formalen Methode zur Anforderungsanalyse in der Automobil-Softwareentwicklung am Beispiel einer Schnittstelle für Hochvolt-Nebenaggregate. Bachelor's Thesis, Leibniz Universität Hannover, 2015.

[7] D. Harel and S. Maoz. Assert and negate revisited: Modal semantics for UML sequence diagrams. *Software and Systems Modeling (SoSyM)*, 7(2):237–252, 2008.

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

[9] IAV GmbH. The 400-volt power socket for vehicles. *automotion*, 2:36–37, June 2014.

[10] J. Kim, K. Larsen, B. Nielsen, M. Mikučionis, and P. Olsen. Formal analysis and testing of real-time automotive systems using uppaal tools. In M. Núñez and M. Güdemann, editors, *Formal Methods for Industrial Critical Systems*, volume 9128 of *LNCS*, pages 47–61. Springer International Publishing, 2015.

[11] S. Konrad and B. Cheng. Real-time specification patterns. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 372–381, May 2005.

[12] R. Löffler, M. Meyer, and M. Gottschalk. Formal scenario-based requirements specification and test case generation in healthcare applications. In *Proc. 2010 ICSE Workshop on Software Engineering in Health Care*, pages 57–67, New York, NY, USA, 2010. ACM.

[13] K. Pohl, H. Hönninger, R. Achatz, and M. Broy, editors. *Model-Based Engineering of Embedded Systems – The SPES 2020 Methodology*. Springer Berlin Heidelberg, 2012.

[14] A. Post, J. Hoenicke, and A. Podelski. rt-inconsistency: A new property for real-time requirements. In D. Giannakopoulou and F. Orejas, editors, *Fundamental Approaches to Software Engineering*, volume 6603 of *LNCS*, pages 34–49. Springer Berlin Heidelberg, 2011.

[15] A. Post, I. Menzel, J. Hoenicke, and A. Podelski. Automotive behavioral requirements expressed in a specification pattern system: a case study at BOSCH. *Requirements Engineering*, 17(1):19–33, 2012.

[16] ScenarioTools website. `http://scenariotools.org`. online, accessed 26-March-2015.