Early Verification and Validation According to ISO 26262 by Combining Fault Injection and Mutation Testing

Rakesh Rana[1(✉)], Miroslaw Staron[1], Christian Berger[1], Jörgen Hansson[1], Martin Nilsson[2], and Fredrik Törner[2]

[1] Computer Science and Engineering, Chalmers/University of Gothenburg, Gothenburg, Sweden
rakesh.rana@gu.se
[2] Volvo Car Corporation, Göteborg, Sweden

Abstract. Today software is core part of modern automobiles. The amount, complexity and importance of software components within Electrical/Electronics (E/E) systems of modern cars is only increasing with time. Several automotive functions carrying software provide or interact with safety critical systems such as systems steering and braking and thus assuring functional safety for such systems is of high importance. Requirements for the safety assurance are specified partially by such functional safety standards as ISO 26262. The standard provides the framework and guidelines for the development of hardware and software for components deemed to be safety critical. In this chapter we argue that traditional approaches for safety assurance such as fault injection and mutation testing can be adapted and applied to functional models to enable early verification and validation according to the requirements of ISO 26262. We show how to use fault injection in combination with mutation based testing to identify defects early in the development process - both theoretically and on a case of self-driving miniature vehicles. The argument is grounded upon the current best practices within the industry, a study of ISO 26262 standard, and academic and industrial case studies using fault injection and mutation based testing applied to the functional model level. In this paper we also provide the initial validation of this approach using software of a self-driving miniature vehicle.

Keywords: Fault injection · Mutation testing · ISO 26262 · Simulink · Model based development · Automotive domain · Safety critical software

1 Introduction

Nowadays, a typical premium car has up to 70 ECUs, which are connected by several system buses to realize over 2,000 functions [1]. As around 90 % of all innovations today are driven by electronics and software the complexity of cars embedded software is expected to grow. The growth is fuelled by cars beginning

J. Cordeiro and M. van Sinderen (Eds.): ICSOFT 2013, CCIS 457, pp. 164–179, 2014.
DOI: 10.1007/978-3-662-44920-2_11

to act more proactively and more assistive to its drivers, which requires software to interact with hardware more efficiently and making more decisions automatically (e.g. collision avoidance by braking, brake-by-wire or similar functions). In total with about 100 million lines of code (SLOC), premium segment vehicles carry more software code than in modern fighter jets and airliners [2]. Software for custom functionality in modern cars is usually developed by multiple suppliers although it is designed by a single OEM (Original Equipment Manufacturer) like Volvo Cars. The distributed development and use of standards like AUTOSAR aims to facilitate reuse of software and hardware components between different vehicle platforms, OEMs and suppliers [3]. However, testing of such systems is more complex and today testing of software generally accounts for almost 50 % of overall development costs [4].

ISO-26262 in automotive domain poses stringent requirements for development of safety critical applications and in particular on the testing processes for this software. These requirements are intended to increase the safety of modern cars, although they also increase the cost of modern cars with complex software functions influencing safety or car passengers.

The position for which we argue in this paper is that *efficient verification and validation of safety functions requires combining Model Based Development (MBD) with fault injection into models with mutation testing.* This position is based on the studies of the ISO 26262 standard (mainly Chap. 6 that describes requirements on software development but also Chap. 4, which poses requirements on product development [5]). It is also based on previous case studies of the impact of late defects on the software development practices in the automotive section [6].

The requirements from the ISO 26262 standard on using fault injection techniques is challenging since it relates to the development of complete functions rather than components of sub-components of software. The current situation in the automotive sector is that fault injection is used, but it is used at the level of one electronic component (ECU) or one software system, rarely at the function level [7,8].

The current state of art testing is not enough for detecting safety defects early in the automotive software development process since fault injection is done late in the development (when ECUs are being developed), which usually makes the detection of specification-related defects difficult and costly [6]. This detection should be done in the model level when the ECUs functionality is still under design and thus, it is relatively cheap to redesign. The evidence from literature on successful use of fault injection shows that the technique indeed is efficient in finding dependability problems of hardware and software systems when applied to computer systems [9]. To be able to increase the effectiveness of the fault injection strategies and identify whether the faults should be injected at the model, software or ECU level - mutation testing should be applied to verify the adequacy of test cases. And finally we need to assess how to combine these approaches and apply them at the model level that will enhance our ability to detect safety related defects right at the design stage.

In this paper we provide a roadmap, which shows how to introduce fault injection and mutation testing to modelling of automotive software in order to avoid costly defects and increase the safety of modern and future cars. This paper is the extended version of our previous work [10] where we presented the theoretical approach. In this paper we include a validation of this framework on a set of software components of self-driving miniature vehicles. The system used for initial validation is developed using a code-centric approach which makes the framework more generic as the initial evaluation in [10] was conducted on model-based development.

The remaining of the paper is structured as follows: In the next Sect. 2 we provide an overview of software development in automotive domain and associated concepts. This is followed by brief discussion on related work in Sect. 3 and our position is presented and discussed in Sect. 4. Section 5 presents the initial validation case for the framework and Sect. 6 provides conclusions.

2 Background

In this section we take a brief overview on the current state of automotive software development process and environment, how safety is important in safety critical applications and overview of theoretical background on fault injection techniques and mutation testing.

2.1 Automotive Software Development and ISO 26262

Various software functions/applications developed within the automotive industry today are classed as safety critical for example Volvo's City Safety consists of components that are safety critical (Fig. 1).

Broy [1] gives examples of functions/areas within automotive domain of recent development which includes crash prevention, crash safety, advanced energy management, adaptable man-machine interface, advanced driver assistance, programmable car, car networking etc., much of these fall within the safety critical functionality and demands high quality and reliability. Also a number of on-going projects are directed towards the goal of self-driving cars.

Software development in automotive sector in general follows the 'V' process, where OEMs take the responsibility of requirement specification, system design, and integration/acceptance test. This is followed by the suppliers, where the actual code that runs on ECUs is developed. Although the code is tested at the supplier level (mainly unit testing), the OEMs are responsible for the final integration, system and acceptance testing to ensure that the given implementation of a software (SW) meets its intended functional and safety goals/demands.

In this model of software/product development (see Fig. 2) testing is usually concentrated in the late stages of development, which also implies that most of the defects are discovered late in the development process. In a recent study using real defect data from an automotive software project from the industry showed that late detection of defects is still a relevant problem and challenge yet

Fig. 1. Volvo Cars city safety function, image provided by Volvo Car Corporation.

to overcome [6]. The defect inflow profile presented in this study is presented in Fig. 3 for reference, which exhibits a clear peak in number of open defects in the late stages of function development/testing.

Testing the software is an important tool of ensuring correct functionality and reliability of systems but it is also a very resource intensive activity accounting for up to 50 % of total software development costs [11] and even more for safety/mission critical software systems. Thus having a good testing strategy is critical for any industry with high software development costs. It has also been shown that most of the defects detected during testing do not depend on actual implementation of code, about 50 % of defects detected during testing in

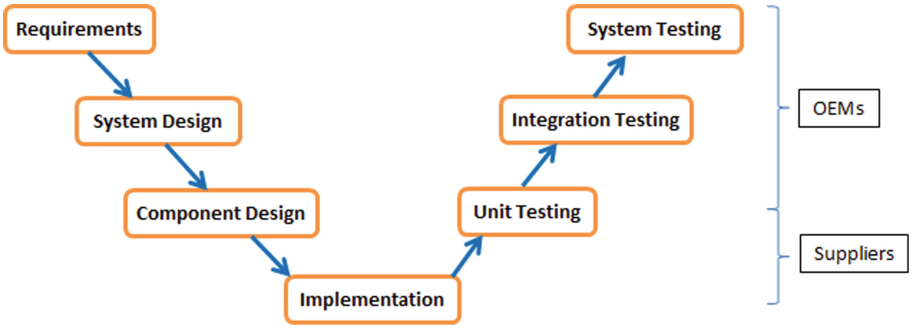

Fig. 2. The V-model in the automotive industry with distinction between the OEM and supplier contributions.

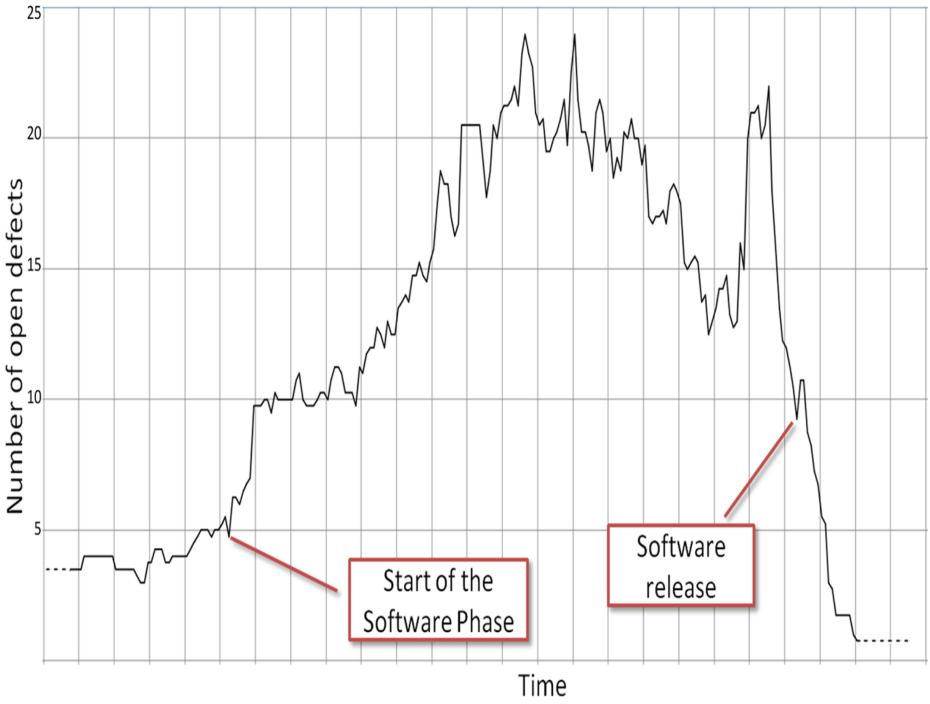

Fig. 3. Defect inflow profile for automotive software project, as given in [6].

the study by Megen and Meyerhoff [12] were found during the test preparation, an activity independent of the executable code. And since automotive sector has already widely adopted MBD for the software development of embedded systems, a high potential exists for using the behavioural modes developed at the early stages of software development for performing some of the effort spent on V&V (Verification & Validation). Early V&V by helping to detect defects early will potentially save significant amount of cost for the projects.

2.2 ISO 26262

ISO/IEC 26262 is a standard describing safety requirements. It is applied to safety-related systems that include one or more electrical and/or electronic (E/E) systems. The overview of safety case and argumentation is represented in Fig. 4.

Written specifically for automotive domain, the ISO-26262 standard is adapted for the V-model of product development corresponding to the current practice in the industry. The guidelines are laid out for system design, hardware and software design and development and integration of components to realize the full product. ISO-26262 includes specifications for MBD and provides recommendations for using fault injection techniques for hardware integration and testing, software unit testing, software integration testing, hardware-software

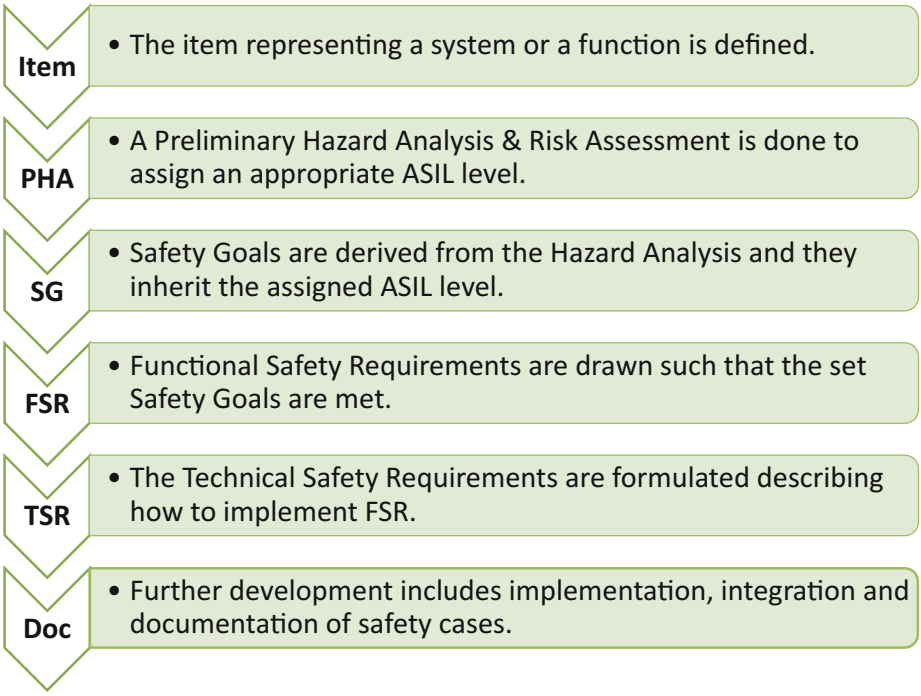

Fig. 4. Overview of ISO-26262 safety case & argumentation process.

integration testing, system integration testing and vehicle integration testing. Although the functional safety standard specifies clearly the recommendations for using fault injection during various stages of testing but does not recommend anything with respect to using mutation testing. This also reflects the current standard practice within the automotive industry where mutation testing is not widely adopted yet.

2.3 Fault Injection

Fault injection techniques are widely used for experimental dependability evaluation. Although these techniques have been used more widely for assessing the hardware/ prototypes, the techniques are now about to be applied at behavioural models of software systems [13], thus enabling early verification of intended functionality as well as enhancing communication between different stakeholders. Fault injection techniques applied at models level offer distinct advantages especially in an industry using MBD, but use of these techniques at model level in automotive industry is currently at its infancy. Figure 5 shows a mind map of classification of fault injection techniques based on how the technique is implemented; some of the tools which are developed based on given approach are also listed for reference. For a good overview of fault injection techniques readers are referred to [9,14].

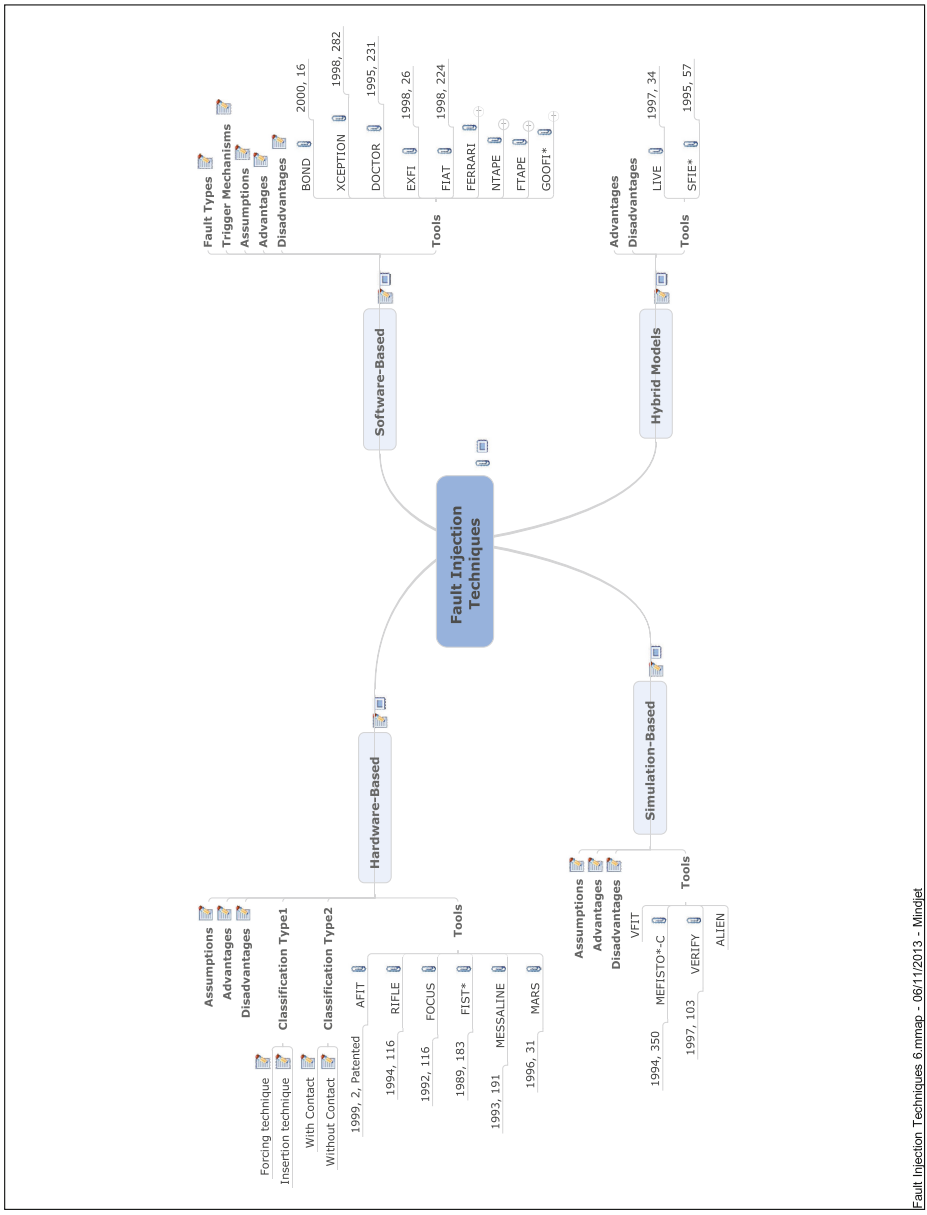

Fig. 5. Common classification of fault injection techniques and implementation tools, description available in [9,14].

2.4 Mutation Testing

Mutation testing is technique for assessing the adequacy of given test suite/set of test cases. Mutation testing includes injection of systematic, repeatable seeding of faults in large number thus generating number of copies of original software artefacts with artificial fault infestation (called a mutant). And on the basis of what percentage of these mutations are detected by the given test cases/suite gives a metrics (called "mutation adequacy score" [15]) which can be used for measuring the effectiveness of given test suite. Faults for mutation testing approach can be either hand written or auto-generated variants of original code. The effectiveness of this approach in mimicking the real faults has also been established [16] i.e. mutants do reflect characteristics of real faults. Mutation theory is based on two fundamental hypotheses namely Competent Programmer Hypothesis (CPH) and the Coupling Effect, both introduced by DeMillo et al. [17]. CPH at its core reflects the assumption that programmers are competent in their job and thus would develop programme close to correct version while coupling effect hypothesis according to Offutt is "Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex defects" [18].

3 Related Work

A number of European Union sponsored projects have within the area of embedded software development and safety critical systems have looked at and developed techniques to effectively use fault injection for safe and reliable software development. The examples include the ESACS [19] (Enhanced Safety Assessment for Complex Systems), the ISAAC [20] (Improvement of Safety Activities on Aeronautical Complex systems). These projects have used the SCADE (Safety-Critical Application Development Environment) modelling environment to simulate hardware failure scenarios to identify fault combinations that lead to safety case violations.

A model-implemented fault injection plug-in to SCADE called FISCADE is introduced in [21] which utilizes approach similar to mutation based testing and replaces the original model operators by equivalent fault injection nodes. The derived models are then used to inject the fault during execution and log the results which are analysed later. Dependability evaluation of automotive functions using model based software implemented fault injection techniques have also been studied in [22].

A generic tool capable of injecting various types of faults on the behavioural or functional Simulink models is also developed and introduced [13]. The tool called MODIFI (or MODel-Implemented Fault Injection tool) can be sued to inject single or multiple point faults on behavioural models, which can be used to study the effectiveness/properties of fault tolerant system and identify the faults leading to failure by studying the fault propagation properties of the models.

Another work [23] with its root in the European CESAR (Cost-efficient methods and processes for safety relevant embedded systems) project provides a good

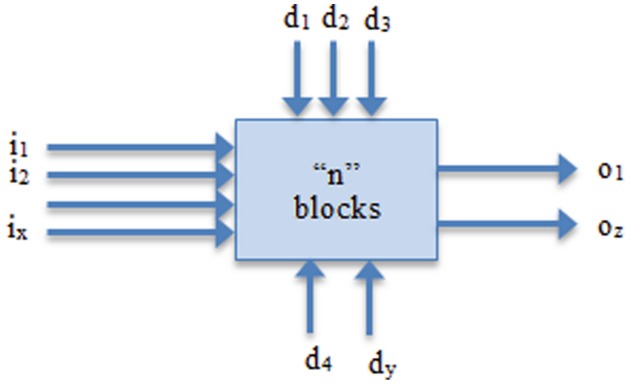

Fig. 6. MBD based representation of a general system with inputs, outputs and dependencies.

theoretical overview of how fault and mutation based test coverage can be used for automated test case generation for Simulink models. We provide a practical framework on how fault injection combined with mutation testing within an MDB environment can be used in the industry. And how will this practice enhance the verification and validation of software under development, its functional validation that would generates statistics for the effective argumentation of ISO 26262 compliance.

4 Framework for Early Verification and Validation According to ISO 26262

We contend that fault injection can be effectively used at the model level to verify and validate the attainment or violation of safety goals. By applying mutation testing approach at the model level enough statistical evidence will be provided for the coverage needed for argumentation of fulfilment of safety goals as per the ISO26262 safety standard requirements.

A major challenge in successful argumentation of ISO-26262 compliance is to provide statistical evidence that Safety Goals (SGs) would not be violated during operation and doing this within reasonable testing efforts.

If we are able to differentiate early between defects that will or not cause the violation of SGs, the amount of testing required will be manageable. With MBD the testing for functionality under these defect conditions could be modelled using fault injection techniques, while the possibility of implementation bugs in the actual code can be checked using the mutation testing approach. The framework on how this could be achieved in practice is as follows:

As illustrated in Fig. 6, a given system/function generally has following common features (in context of model based development): firstly it will have x inputs ($i_{1,2,...x}$); it would have dependencies to other y components/functions ($d_{1,2,...y}$);

it will have z outputs ($o_{1,2,...z}$); and it will have a number of sub-units/modules within it that implements the intended functionality, let us assume that this part contains n basic blocks in the modelling environment corresponding to n statements for a hand written code. To verify and validate the correct functionality and ISO 26262 compliance of this generic function using fault and mutation testing approach we can follow the steps as:

- Assign or define the Functional Safety Requirements (FSRs) and Technical Safety Requirements (TSRs) for the z outputs of the given system/function in accordance to ISO 26262.
- Use fault injection technique to inject common occurring defects and other theoretically possible fault conditions at the x inputs.
- By studying the fault propagation of different injected faults at inputs and their effect on outputs, the individual faults and combinations of it that violate the FSRs for given system can be noted.
- Steps (b) & (c) should also be done to test and validate the given system/function dependencies on other functions/components.
- Mutation approach is then used to inject faults (or cause mutations) to the n basic blocks of given functional model and assess the detection effectiveness of test suite/cases for possible implementation bugs.
- The mutants which are not killed by given set of test cases/suits are examined for their effect on given functions FSRs, if the given mutation violates the SGs/FSRs then a suitable test case will be created to detect/kill such mutants i.e. detect such bugs in actual code.

Thus by following the above mentioned steps we not only ensure that the given function works as intended, does not violate the SGs and FSR/TSRs under faulty inputs and/or due to dependencies on other functions, but we can also identify possible implementation defects using the mutation approach and ensure that we have test cases ready to catch such faults that can potentially violate the SGs/TSRs even before the code is implemented/generated.

Further to make this framework/approach more effective in industrial practice we identify some best practices that will have positive impact on detecting defects early in the development process and thus have effective V&V of ISO26262.

- Model evolution corresponding to different levels of software/product development.
- Specification and testing for SGs, FSRs and TSRs on the behavioural models.
- Identification of different types of defects/types of faults and at what stage they could be modelled/injected at models to ensure that models are build robust right from the start instead of adding fault tolerance in later stages of development.

5 Case Study: Validation

In this section we present the validation of proposed framework on a set of components for self-driving miniature vehicles. The software for the miniature

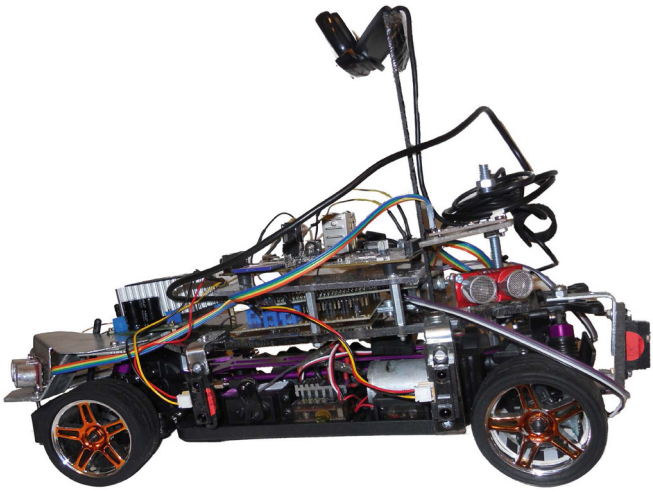

Fig. 7. Self-driving miniature vehicle [25].

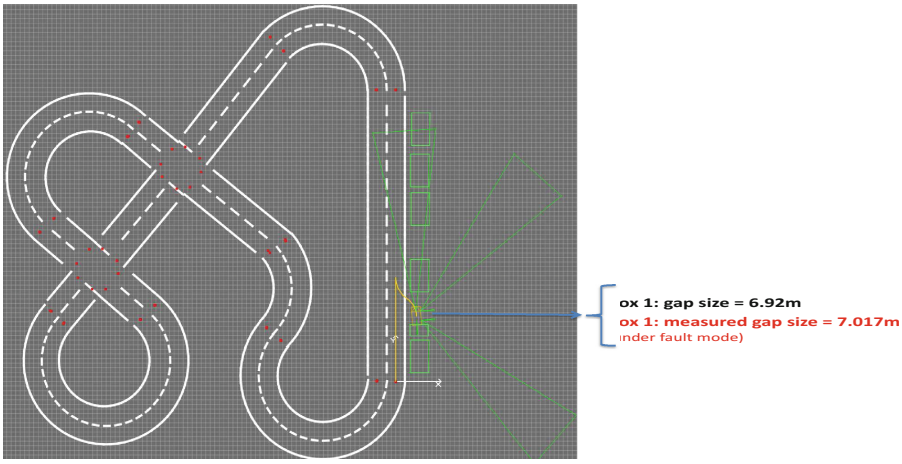

Fig. 8. Test track for the experiment with parking gap from our simulation environment.

vehicles is build using similar methods and tools as professional software in the automotive industry, although on a smaller scale. In the validation we use the self-parking function of a self-driving miniature vehicle [24]. The architecture of the software is described in detail in [25] and one of our miniature vehicles using the self-driving vehicle software and a scenario for a sideways parking realized in our simulation environment are illustrated in Figs. 7 and 8. The miniature vehicles are in the scale 1:10 compared to the normal cars.

For understanding the initial validation of this framework it is sufficient to note that the functionality we are dealing with is self-parking for on a sideways parking strip. The self-parking algorithm expects a gap size of at least 7 m to park in one turn without using an additional correction trajectory. This scenario is presented in Fig. 8.

We applied the framework for early verification and validation following the steps given in Sect. 4 as follows:

- Assign FSR/TSR: An example of obvious functional safety requirement (FSR) for self-parking functionality is parking without hitting any other object. The corresponding technical safety requirement (TSR) can thus be parking only when gap size exceeds 7 m (minimum gap size requirement).
- Using fault injection to simulate common fault scenario: A fault scenario is created by injecting a fault in the returned value for the travelled path by adding an error value of maximum 3.4 % for the relatively travelled path increment. Thus, the size for measured gaps (due to faulty sensor input) increases for example by 9.7 cm to 7.01678 m.
- Identify fault scenarios leading to FSR/TSR violations: Since in the experiment with fault injection, the parking algorithm depends on the travelled path; thus the algorithm parks the car in the lower gap which leads to a safety case violation because the cars collides with the obstacle at the rear side.
- Repeat steps (b) & (c) for all inputs: For this experiment, we focused on the fault injection for a single signal.
- Cause mutations: Single point mutations are caused by changing the logical operators in the self-parking function code, the standard test protocol to test the expected functionality was then applied to evaluate the generated mutants.
- Examine mutants & create new test cases: The mutants and the results whether they were successfully detected are provided in Table 1. In this simple case itself with only 24 mutations, to our surprise two mutations produced unexpected results and violated the assigned FSR. While previously the test protocol has been deemed being sufficient for this function, the experiment clearly demonstrated the need for adding further test cases to reliably spot these failures and to detect possible faults leading to FSR violations.

5.1 Lessons Learned

The initial validation experiment presented in this section for the proposed framework is the first step towards a complete validation of this framework in an industrial setting. Although the framework is focused on using fault injection and mutation testing at functional model level in model-based development to shift some of the verification and validation efforts to early stages of development, the example here demonstrated its applicability of given framework in a code-centric development environment as well.

The experiments using the software of a miniature vehicle provided a proof-of-concept for the framework and provide a frame of reference with respect to

Table 1. Mutation testing output, case with and without fault mode scenario.

ID	Mutant	Change description	Test case result using regular vehicle simulation	Test case result using vehicle simulation with fault injection
1	Unmodified	Original self-parking algorithm	Passed as expected	Failed as not expected, vehicle took first gap, collided (expected: robust algorithm dealing with varying travelled distance data)
2	68	Changed == to >	Failed as expected, vehicle did not start	Failed as expected, vehicle did not start
3	73	Changed first > to ==	Failed as expected, vehicle started hardly noticeable (v < 0.009m/s)	Failed as expected, vehicle started hardly noticeable (v < 0.009m/s)
4	73	Changed && to ||	Failed as expected, vehicle moved forwards slightly and pull back while turning to left	Failed as expected, vehicle moved forwards slightly and pull back while turning to left
5	73	Changed second < to >	Failed as expected, vehicle moved to the end of the second parking spot but did not start parking	Failed as expected, vehicle moved to the end of the first parking spot but did not start parking
6	79	Changed first >= to <=	Failed as expected, vehicle did not start	Failed as expected, vehicle did not start
7	79	Changed && to ||	Failed as expected, vehicle moved backwards while turning to left	Failed as expected, vehicle moved backwards while turning to left
8	73	Changed second < to >	Failed as expected, vehicle moved to the end of the second parking spot but did not start parking	Failed as expected, vehicle moved to the end of the first parking spot but did not start parking
9	85	Changed first >= to <=	Failed as expected, vehicle moved backwards while turning to right	Failed as expected, vehicle moved backwards while turning to right
10	85	Changed && to ||	Failed as expected, vehicle moved backwards while turning first to right and then to left (S-shaped)	Failed as expected, vehicle moved backwards while turning first to right and then to left (S-shaped)
11	85	Changed second < to >	Failed as expected, vehicle moved to the end of the second parking spot but did not start parking	Failed as expected, vehicle moved to the end of the first parking spot but did not start parking
12	91	Changed first >= to <=	Failed as expected, vehicle moved backwards while turning to left	Failed as expected, vehicle moved backwards while turning to left
13	91	Changed && to ||	Failed as expected, vehicle moved backwards while turning to left	Failed as expected, vehicle moved backwards while turning to left
14	91	Changed second < to >	Failed as expected, vehicle moved to the end of the second parking spot, started parking, but stopped after the first right turn	Failed as expected, vehicle moved to the end of the first parking spot, started parking, but stopped after the first right turn
15	97	Changed >= to <=	Failed as expected, vehicle did not start	Failed as expected, vehicle did not start
16	115	Changed first > to <	Failed as expected, vehicle did not find the parking stop and continues driving	Passed as not expected, vehicle parked in the second parking spot because the noise added to the travelled distance resulted in a valid parking gap size
17	115	Changed && to ||	Failed as expected, stopped before the first parking gap, collided with parked car	Failed as expected, stopped before the first parking gap, collided with parked car
18	115	Changed second > to <	Failed as expected, stopped before the first parking gap, collided with parked car	Failed as expected, stopped before the first parking gap, collided with parked car
19	126	Changed first > to <	Failed as expected, vehicle took first gap, collided	Failed as expected, vehicle took first gap, collided
20	126	Changed && to ||	Failed as expected, vehicle did not find the parking stop and continues driving	Failed as expected, vehicle did not find the parking stop and continues driving
21	126	Changed second > to <	Failed as expected, vehicle did not find the parking stop and continues driving	Failed as expected, vehicle did not find the parking stop and continues driving
22	135	Changed first > to <	Failed as expected, vehicle did not find the parking stop and continues driving	Failed as expected, vehicle did not find the parking stop and continues driving
23	135	Changed && to ||	Failed as expected, stopped before the first parking gap, collided with parked car	Failed as expected, stopped before the first parking gap, collided with parked car
24	135	Changed second > to <	Failed as expected, stopped before the first parking gap, collided with parked car	Failed as expected, stopped before the first parking gap, collided with parked car

its possible effectiveness. While in full scale safety evaluations following the ISO 26262, a given function depending on its functionality may be subjected to tens of safety goals and even larger number of corresponding FSR/TSRs, we only evaluated one such scenario. Still with only a single fault scenario - we were able

to identify faults leading to safety case violation. Also the mutation approach applied to this exemplary scenario by using 24 mutations, 2 out of these 24 mutants produced unexpected results and exposed the deficiency of the current test protocol, which was considered as adequate for the given functionality.

Therefore while these are encouraging results pointing towards applicability and effectiveness of the proposed framework, we also learned that we need further validation on industrial scale projects to increase the external validity of these results. Further for this framework to be successful in any organization much of the steps of described framework will have to be automated and supported by appropriate tools. As explained in Sects. 2 and 3, a number of tools for fault injection and mutation testing based approaches are available for code-centric development making this framework practical for implementation on large scale with high automation. But corresponding tools to support fault injection and mutation based testing at functional model level in model-based development are not widely available and the few tools currently available are in their early stages of development where reliability of such tools will be an issue at least for some time in near future.

6 Conclusions

In this paper we have examined the growing importance of software in automotive domain. The development of software in automotive and other similar industries has widely adopted the paradigm of model based development and by the nature of application much of the functionality developed and implemented in these sectors is safety critical. Safety critical software/application development requires observation of stringent quality assessment and adherence to functional safety standards such as ISO 26262 in automotive and DO-173 in aerospace industry.

Development of behavioural models in MBD offers significant opportunity to do functional testing early in the development process. Fault injection and mutation testing approach in combination can be used to effectively verify and validate the functional properties of a software system/function. The approach will also provide the required statistics for the argumentation of safety standards compliance. In this paper the need for such validation and a framework on how this could be achieved in practice is discussed. More research and tools are needed to bring this approach into wider industrial adoption.

Initial validation of our proposed framework provided a proof-of-concept and produced encouraging results indicating its usefulness and effectiveness in practice. It is also noted that the framework will become much more effective and easy to use for model-based development as tools related to fault injection and mutation testing at model level matures over time. In the meantime, validation on industrial scale functions will provide further evidence to evaluate the applicability and effectiveness of the proposed framework in practice.

By detecting defects early and being able to do much of verification and validation of intended functionality, robustness and compliance to safety standards on the models the quality and reliability of software in automotive domain

will be significantly enhanced. More effective approaches and tools support will also reduce the V&V costs and lead to shorter development times. High quality, reliable and dependable software in automobiles brings innovative functionality sooner, keeps product costs lower and most importantly ensures that automobiles are safer than ever before.

Acknowledgements. The work has been funded by Vinnova and Volvo Cars jointly under the FFI programme (VISEE, Project No: DIARIENR: 2011-04438).

References

1. Broy, M.: Challenges in automotive software engineering. In: Proceedings of the 28th International Conference on Software Engineering, pp. 33–42 (2006)
2. Charette, R.N.: This car runs on code. IEEE Spectr. **46**(3), 3 (2009)
3. Fennel, H., Bunzel, S., Heinecke, H., Bielefeld, J., Fürst, S., Schnelle, K.P., Grote, W., Maldener, N., Weber, T., Wohlgemuth, F., et al.: Achievements and exploitation of the autosar development partnership. In: Convergence 2006, October 2006
4. Boehm, B., Basili, V.: Defect reduction top 10 list. Computer **34**, 135–137 (2001)
5. ISO, C.: 26262, road vehicles-functional safety (2011)
6. Mellegård, N., Staron, M., Törner, F.: A light-weight defect classification scheme for embedded automotive software and its initial evaluation. In: 2012 IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE), pp. 261–270. IEEE (2012)
7. Hillenbrand, M., Heinz, M., Adler, N., Müller-Glaser, K.D., Matheis, J., Reichmann, C.: ISO/DIS 26262 in the context of electric and electronic architecture modeling. In: Giese, H. (ed.) ISARCS 2010. LNCS, vol. 6150, pp. 179–192. Springer, Heidelberg (2010)
8. Schätz, B.: Certification of embedded software – impact of ISO DIS 26262 in the automotive domain. In: Margaria, T., Steffen, B. (eds.) ISoLA 2010, Part I. LNCS, vol. 6415, p. 3. Springer, Heidelberg (2010)
9. Hsueh, M., Tsai, T., Iyer, R.: Fault injection techniques and tools. Computer **30**(4), 75–82 (1997)
10. Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M., Törner, F.: Increasing efficiency of iso 26262 verification and validation by combining fault injection and mutation testing with model based development. In: 8th International Joint Conference on Software Technologies-ICSOFT-EA, Reykjavík, Iceland, July 2013
11. Jones, E.L.: Integrating testing into the curriculumarsenic in small doses. In: ACM SIGCSE Bulletin, vol. 33, pp. 337–341
12. Megen, R., Meyerhoff, D.: Costs and benefits of early defect detection: experiences from developing client server and host applications. Software Qual. J. **4**(4), 247–256 (1995)
13. Svenningsson, R., Vinter, J., Eriksson, H., Törngren, M.: MODIFI: a MODel-implemented fault injection tool. In: Schoitsch, E. (ed.) SAFECOMP 2010. LNCS, vol. 6351, pp. 210–222. Springer, Heidelberg (2010)
14. Ziade, H., Ayoubi, R., Velazco, R., et al.: A survey on fault injection techniques. Int. Arab J. Inf. Technol. **1**(2), 171–186 (2004)
15. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)

16. Andrews, J., Briand, L., Labiche, Y.: Is mutation an appropriate tool for testing experiments? [software testing]. In: Proceedings of the 27th International Conference on Software Engineering, ICSE 2005, pp. 402–411 (2005)
17. DeMillo, R., Lipton, R., Sayward, F.: Hints on test data selection: help for the practicing programmer. Computer **11**(4), 34–41 (1978)
18. Offutt, A.: Investigations of the software testing coupling effect. ACM Trans. Softw. Eng. Methodol. (TOSEM) **1**(1), 5–20 (1992)
19. ESAC: Enhanced safety assessment for complex systems. FP5-GROWTH contract no. G4RDCT-2000-00361
20. ISAAC: Improvement of safety activities on aeronautical complex systems. FP6-AEROSPACE project reference 501848 (2007)
21. Vinter, J., Bromander, L., Raistrick, P., Edler, H.: Fiscade - a fault injection tool for scade models. In: 2007 3rd Institution of Engineering and Technology Conference on Automotive Electronics, pp. 1–9 (2007)
22. Plummer, A.: Model-in-the-loop testing. Proc. Inst. Mech. Eng. Part I: J. Syst. Control Eng. **220**(3), 183–199 (2006)
23. Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., Weissenbacher, G.: Mutation-based test case generation for simulink models. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 208–227. Springer, Heidelberg (2010)
24. Berger, C., Chaudron, M., Heldal, R., Landsiedel, O., Schiller, E.M.: Model-based, composable simulation for the development of autonomous miniature vehicles. In: Mod4Sim'13: 3rd International Workshop on Model-driven Approaches for Simulation Engineering at SCS/IEEE Symposium on Theory of Modeling and Simulation in Conjunction with SpringSim 2013 (2013)
25. Berger, C., Hansson, J., et al.: Cots-architecture with a real-time os for a self-driving miniature vehicle. In: Proceedings of Workshop ASCoMS (Architecting Safety in Collaborative Mobile Systems) of the 32nd International Conference on Computer Safety, Reliability and Security (2013)