State-of-the-Art Tools and Methods Used in the Automotive Industry

Harald Altinger

Abstract In recent times, the number of features within a modern-day premium automobile has significantly increased. The majority of them are realized by software, leading to more than 1,000,000 LOC ranging from keeping the vehicle on the track to displaying a movie for rear seat entertainment. The majority of software modules need to be executed on embedded systems, some of them fulfilling mission-critical task, where a failure might lead to a fatal accident. Software development within the automotive industry is different from other industries or open source, as there are more restrictions upon development guidelines and rather strict testing definitions to meet the quality and reliability requirements or even ensure traceability on defect liability. To meet these requirements, various tools and processes have been integrated into the development process, delivering document metadata which can be used for further insights, for example, Software Fault Prediction (SFP).

1 When Reading This Chapter

This chapter represents a short introduction to current tools and development procedures used within the automotive (software) industry. It is a compilation of multiple antecedent publications. Some parts will be extended by knowledge of practitioners to give insights into common processes. As the author is related to one Original Equipment Manufacturer (OEM), some results might be influenced.

H. Altinger (✉)
Audi AG, Ingolstadt, Germany
e-mail: harald.altinger@audi.de

Y. Dajsuren, M. van den Brand (eds.), *Automotive Systems and Software Engineering*, https://doi.org/10.1007/978-3-030-12157-0_4

2 A Short Introduction upon Software within Cars

A modern-day premium car, for example, the 2018 Audi A8 [6], if fully equipped, can contain more than a 100 Electronic Control Units (ECUs), 14 networks and up to 2 SIM cards, and multiple kilometers of cable (compare Fig. 1). According to Charette [13], the software within a modern-day premium car might claim up to 1,000,000 Lines Of Code (LOC). Software modules need to fulfill various tasks ranging from simple comfort features, for example, controlling the stepper motor within an electric seat, up to complex tasks like a predictive chassis. Some features only require a single ECU with a low-power processor, for example, remote garage door control, while others might require a powerful processor with multiple cores and a Graphics Processing Unit (GPU) to encode multimedia content for entertainment. Some features require multiple sensor values connected to different ECUs and influence more than one actuator or even require online and map data, for example, the dynamic matrix headlights (see [6]). The majority of software components need to run under embedded conditions and deliver their result within real time, for example, power steering; others can operate on soft real time, for example, navigation systems. Some features need to meet rather strict development guidelines like Motor Industry Software Reliability Association (MISRA) or need to satisfy safety requirements as Automotive Safety Integrity Level (ASIL). Therefore, one can see a car as a heterogeneous network of multiple distributed computers running a high number of software functions, either

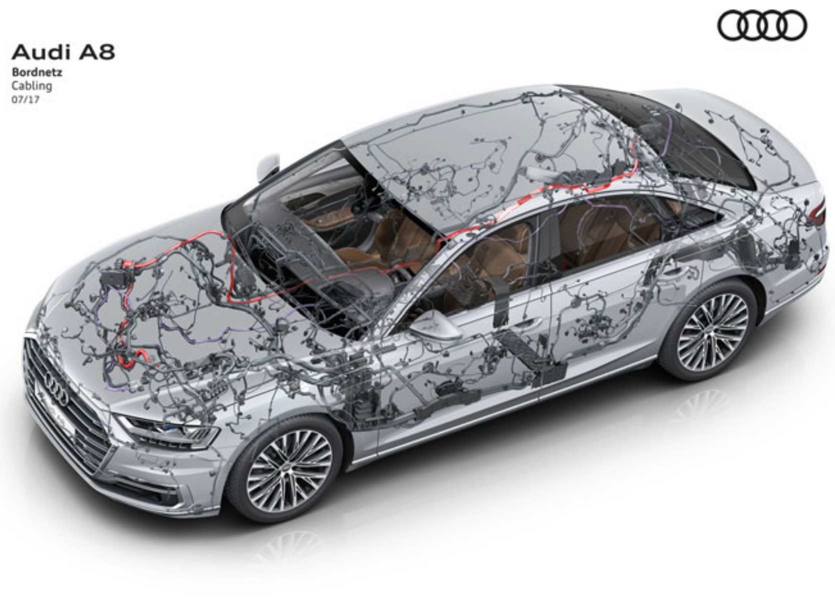

Fig. 1 The cabling topology of the 2018 Audi A8; note the actual ECUs are not shown

stand-alone, distributed asynchronous, or synchronous, which represents a highly complex system.

Some modern-day features might only be realized by software. Most of the required actuators and sensors are already within the car to fulfill various functionalities. Central software components might use these to realize enhanced or additional features, for example, Start/Stop automatic. In this SoftWare Component (SWC) case, it needs to read back the battery capacity level, the climate control state, if the driver is steady on the break, etc. If all these conditions are met, the system instructs the engine control ECU to switch off the engine. Even more complex systems, for example, Advanced Driver Assistance Systems (ADAS), require information from multiple sensors, perform calculation upon more than one ECU, and may control several actuators which sum up to highly complex systems. To visualize this, we are using the Adaptive Cruise Control (ACC) as an example. The system consists of two main sensors, a radar and a camera. Both deliver object information (position, distance, speed, etc.) regarding vehicles ahead of the ego car. Those object information will be calculated on the ECU connected to the sensor; the ego car's motion will be acquired on the Inertial Measurement Unit (IMU) (which is already a part of the car as Electronic Stability Control (ESC) requires precise acceleration information to fulfill its task). The SWC realizing the ACC might be hosted on another ECU, or included on one of the sensors' ECU. It needs to be connected via a bus (CAN, FlexRay) network. This software will calculate acceleration and deceleration upon the driver's presets (distance to vehicle ahead, driving profile, etc.) and send the values to break and engine control ECU. Those components decide upon actual speed and requested (de)acceleration if the engines carrier gas (in case of an Electriv Vehicle (EV) maybe recuperation abilities) is enough or if the break system needs to act. Further details have been released by Duba and Bock [14]; a visualization can be seen in Fig. 2.

Some software are used to compensate mechanical deficits; for example, adding ESC[1] to the 1997 Mercedes A-Class after failing the Elk Test to enhance the car's stability.

These trends lead to an increasing amount of software being indispensable parts of the car. Raising the amount of software might cause the number of software-related bugs to increase too. Figure 3 shows vehicle recalls extracted from the NHTSA database [27]. This database is publicly available and contains all defects which lead to an official recall by the NHTSA on US roads. Similar to Altinger et al. [2], we query the up-to-date database grouping by vehicle models and model years. A software-related recall is counted if the defect or the repair description contains "software," "update," or "program."

One can see that a huge number of vehicles need to be recalled due to software issues. This data is only valid for the USA. The graph uses the term "model year" as a base, meaning the year where the car (or a sub release) has been introduced to the market. This is not necessarily the year where the car has been manufactured or

[1]Nowadays ESC is mandatory by law in EU since 11/2014.

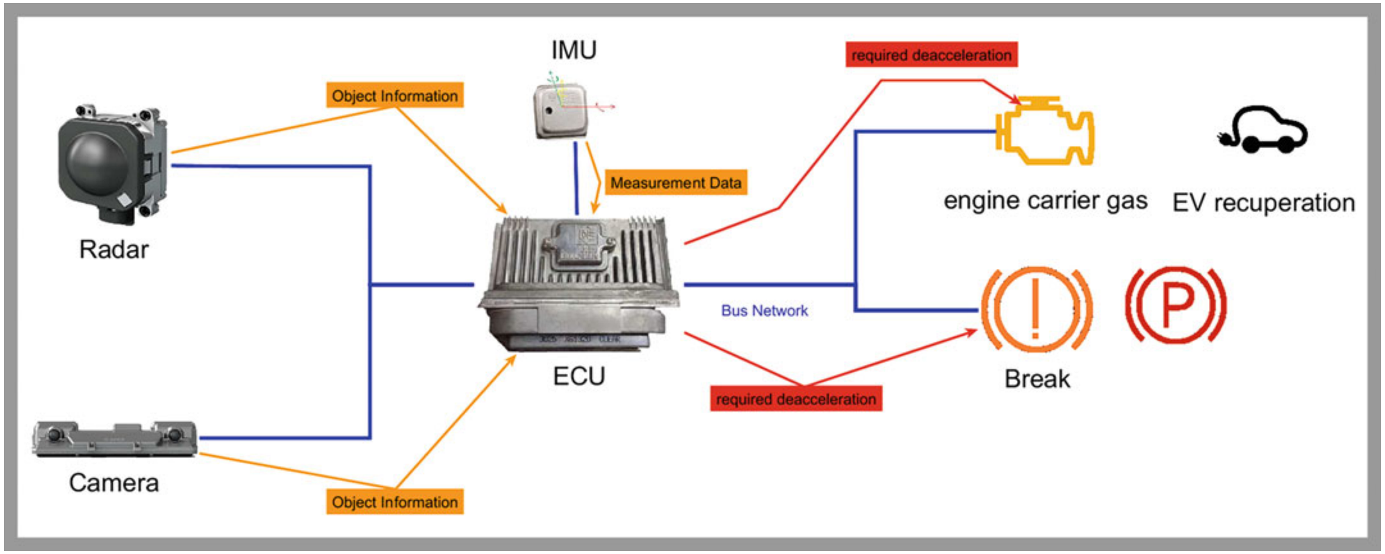

Fig. 2 The ACC workflow composed of three sensors: radar, camera, and IMU. The ECU is symbolic; the ACC SWC might be hosted on one of the sensor ECU. All systems are connected via a bus network

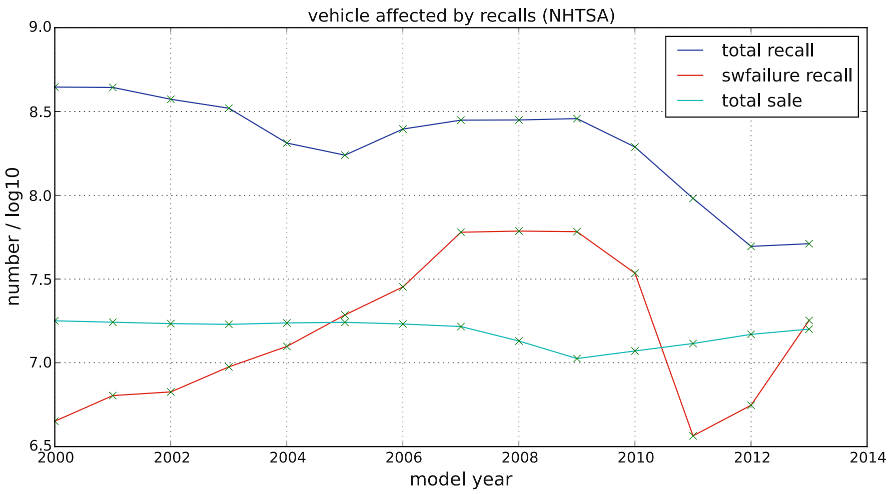

Fig. 3 The software caused recall statistics based on an NHTSA recall database query, [27]. The sale data is originated by Ward's Auto [32]. The graph has been generated using the same settings as Altinger et al. [2]

sold nor the year when the recall has been filed. At high peak, this data shows more recalls than sales which might be caused by vehicles affected by multiple recalls during their lifetime.

Fixing a software bug will be performed by updating the software. If a recall is filed, an OEM needs to pay for contacting the driver, an engineer at a workshop to update the software, maybe a rental car for the customer, etc. There might be lower costs upon using Over The Air (OTA) updates. However, the OEM needs to ensure that all affected vehicles would install the updates. Multiplied with the number of effected vehicles, easily one recall can sum up to millions of euros to fix a simple software bug (compare Capers [12]). This leads even to a high economic interest to prevent bugs in the field and invest a rather high share from the vehicle development costs into testing.

There are no official numbers on the development costs, but Shea [30] did interview John Wolkonowicz, a senior auto analyst for North America at IHS Global, claiming development costs between one (setting up on existing model) and six (new model, new platform, new engines, etc.) billion USD. Broy et al. [11] analyze the development costs for a vehicle electronic system at 300 million EUR, estimating two thirds of this being software development costs.

Developing software to be used within cars is subjected to various restrictions. Coding standards, as the 2004 MISRA-C [23], modeling guidelines as the MAAB [20], limit the usage of programming language features, such as no pointers or no dynamic memory usage or no unlimited depth of function calls, etc. A recent study by Altinger et al. [4] demonstrated the effectiveness of such standard upon preventing common programming bugs. Furthermore, standards like ISO

Table 1 Operators' overview from three automotive projects

Preprocessor		Flow control		Data type		Mathematical		Logic/comp.	
%	Operator	%	Operator	%	Type	%	Type	%	Type
19.41	#define	8.92	break	1.48	int	54.97	+	8.66	!
15.36	#ifndef	6.79	case	96.32	static	13.75	−	3.00	! =
65.24	#include	2.13	default	0.72	struct	1.47	%	7.05	&
		29.24	else	1.48	unsigned	21.85	*	13.06	&&
		0.47	for			7.96	/	49.30	=
		49.46	if					9.08	==
		0.72	return					2.05	|
		2.14	switch					4.89	||
		0.02	while					0.84	>=
		0.12	?					0.15	<=
								0.75	<
								0.80	>
								0.32	~
								0.03	^

26262 [17] demand various testing methods addictive to the risk of failure. The ASIL rating represents a software failure impact upon human lives. If a systems failure puts human life in danger, it will be rated *ASIL D*. This level strongly recommends formal verification of the program code. Even in less critical systems, the norm demands testing goals, for example, branch/statement coverage upon test cases. A good overview has been presented by Reactis software [28].

Altinger et al. [4] performed a study on three different automotive software projects, analyzing the usage of operators within those three projects. Table 1 shows an overall summary. All these projects use C as a programming language, limited by the MISRA 2004 coding guidelines. Table 1 demonstrates a strong tendency of using decision-based code elements (if/else, logic operators, etc.) and no usage of pointers or dynamic memory in the three automotive software projects. The whole software has been developed using model-driven approaches realized with Matlab Simulink and TargetLink to generate the actual C code. The workflow is presented within Fig. 6. A more detailed description on the development process has been presented by Altinger et al. [3].

3 Development Process and Available Documents

The VW corporation employed 45,742 people within their Research and Development (RD) according to their annual fiscal report [31]. This amount of engineers working together demands clear development processes and descriptions to handle such huge projects as developing a car. Figure 6 shows an exemplary development process used by three projects we did analyze. A core component is a clear separation between requirements, development, and testing which is realized by using different specialized tools working together with an adopted toolchain as presented by Kiffe et al. [19]. They present their work on linking different tools from various vendors called ENPROVE. This toolchain enables exporting requirements written in DOORS into a Matlab Simulink model. Thus the engineer designing the model can trace if subsystems accord to requirements.

Altinger et al. [2] performed a questionnaire survey receiving 68 responds from Series Development (SD), Pre Development (PD), and Research (Re). The authors presented IBM DOORS as the most common tool to write specifications. Further, they presented a typical work split between dedicated engineers writing specifications and test cases and developers realizing the software. Within their work, they presented Mathworks Matlab Simulink as the most common tool to design models and generate code.

Figure 2 shows the W-development model as a test-enhanced extension to the well-known V-model as presented by Jin-Hua et al. [18]. One can clearly see a related testing stage to every definition and implementation stage. As defined by the MISRA 2004 standard, every module needs to pass code review stages, which are a core part of the W model (Fig. 4).

Bock et al. [10] present the results from a study on various tools used during different development stages. The authors are presenting a rather good overview and short introduction to all tools they present. Their ranking is based on a questionnaire survey asking whether a tool is in use or a tool/method is familiar, targeting engineers working at SD, PD, and Re. Their work developed a taxonomy to guide engineers on the selection of new tools and methods.

Altinger et al. [2] draft commonly used tools for automotive software engineering. The majority of requirements and specifications are written using IBM DOORS. An engineer exports the related subset to his software module and establishes a link between the DOORS database and a Matlab Simulink model. As stated in Altinger et al. [2], software specifier, developer, and tester are dedicated personnel. Thus, another engineer exports the test requirements and generates test cases associated with Software in the Loop (SiL), Hardware in the Loop (HiL), etc. Following this process, every tool will get updates if requirements change. Scripting interfaces are used to ensure requirements are linked with code parts.

In contrast to classic software development, there is a strict milestone plan for every SWC. Figure 5 visualizes them. The automotive industry is dominated by Start Of Production (SOP), which means there is a fixed date where every module, no matter if mechanical, electrical, or software, has to be available to be fit into the new produced model.

Software development has derived various sub-milestones:

- *Interface freeze* – all software interfaces (including network messages on the CAN and FlexRay), similar to an Application Programming Interface (API), have to be defined and are not allowed to be changed afterward.
- *Feature freeze* – all functionality has to be defined and implemented; rapid prototypes are still allowed; code optimization might not be completed.

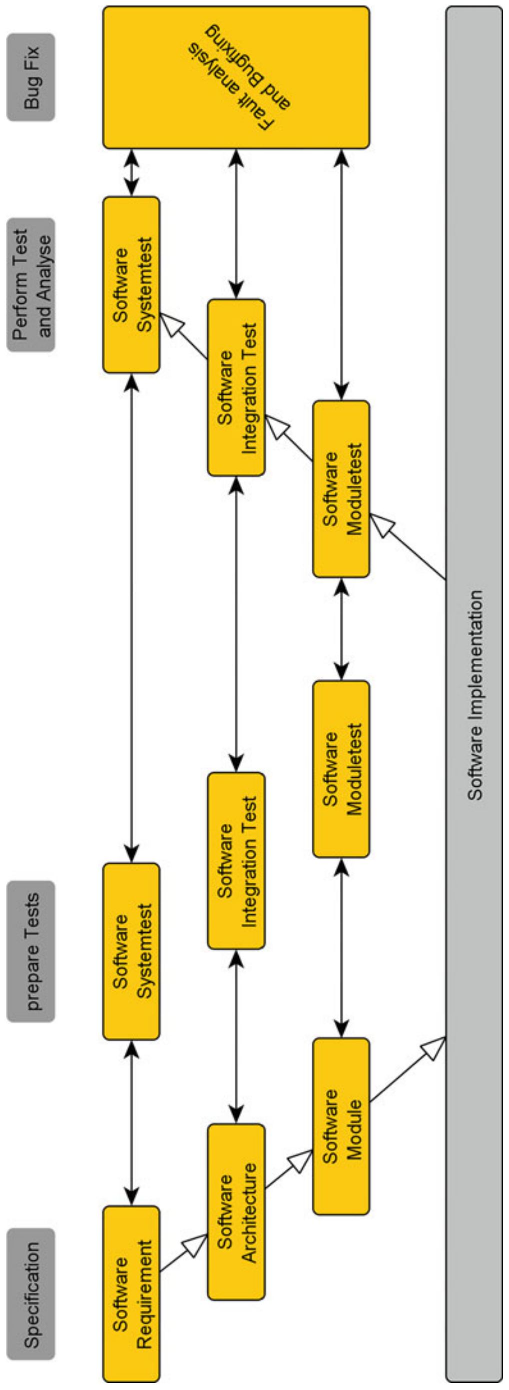

Fig. 4 The W-process as presented by Jin-Hua et al. [18]

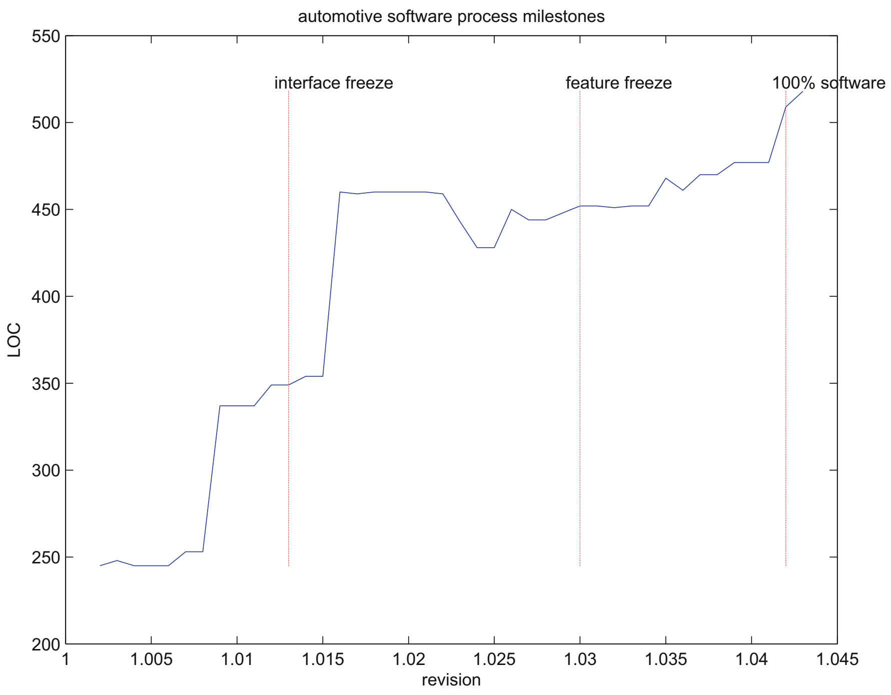

Fig. 5 Sample of software milestones within the automotive domain

- *100 % software* – all implementation shall be done; only bug fixes are allowed after this point.
- *SOP* – Version 1.0 which will be shipped with the first customer cars.

4 Tool Usage

Altinger et al. [2] and Bock et al. [10] yield a list of well-established tools to specify requirements and to perform software tests. Common to both studies, Matlab Simulink is the dominating programming environment.

As outlined by Altinger et al. [3], there exists a wide range of tools supporting the development process. Figure 6 presents their interaction. During the analysis of projects, the following tools served:

- **IBM DOORS**: writes and traces requirements.
- **Matlab Simulink**: develops models and performs basic tests during development.
- **dSpace TargetLink**: generates C code based on the Simulink models.

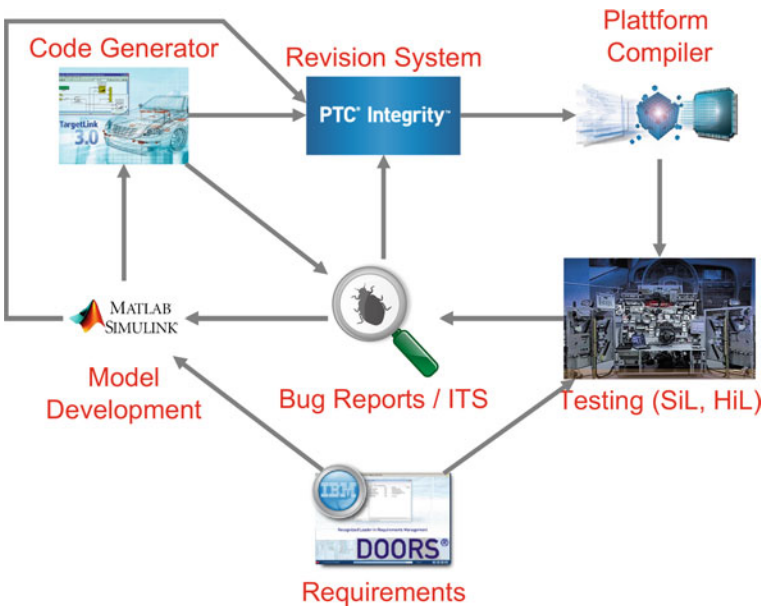

Fig. 6 Development workflow, adopted from Altinger et al. [3]

- **PTC Integrity**: revision system to store each development artifact (model files, configurations, etc.). This tool offers an Integrated Ticketing System (ITS) to handle bug reports and their fixes.
- **Target Compiler**: a compiler specific to the ECUs platform.
- **Testing**: various scripting interfaces using standard automotive tools such as Vector CanOe or CanAp or ADTF, . . .

Further and more detailed tool and test strategies are presented by Müller et al. [24].

ISO 26262 requires tools to be qualified before being used for various levels of ASIL-rated software. The EntwicklungsProzess Verbesserung, German: Development Process Improvement (EnProVe) process as described by Kiffe et al. [19] performs such tool qualifications. In addition, this process develops scripting interfaces between the qualified tools to enable automation.

5 Testing Approaches

Traditional testing approaches two decades ago lead to setting up a test car and perform various road tests on a dedicated test track; if the car passes all of them, the phase is over. Modern-day cars consist of a huge amount of components which need to be tested separately. Common approaches are in the loop tests, such HiL, SiL, Model in the Loop (MiL), etc. As Bergmann et al. [7] state, real test drive will decrease, virtual test drives will increase.

A recent survey by Altinger et al. [2] presents the work distribution upon engineers when developing car software. There is a dedicated group designing test cases which might be executed on HiL or even real test drives. Even within early states of software development, a test case is linked to every stage (compare Jin-Hua et al. [18]) leading to the W-development process (see Fig. 4).

Within computer games, for example, "Need for Speed," one can drive a virtual car on digital reconstructed race tracks. Virtual Test Drive (VTD) (see Dupuis et al. [15]) takes this idea into realistic road simulation. This system uses a simulated world to generate the test stimuli. Based on road network, a car can drive along a virtual street, and other cars might be simulated based on driver model descriptions. The rendering engine can generate an image to be fed into a camera which will generate test input for computer vision algorithm. Sensor model descriptions might extract raw values, for example, LIght Detection And Ranging (LIDAR), based on the virtual world. Nentwig et al. [26] performed analysis upon performance of generated images for lane mark detection considering weather influences. Within their study, they compare generated data with real measurements. Using these toolsets, one might be able to define test cases using real-world assumptions and generate the stimuli for every ECU on a test bench. Müller et al. [24] use this method but extend the data generation. Within their work, they use VTD to calculate sensor input data, for example, video data which is captured by the car's ADAS camera using a video screen. Similar processes are used to generate input data for ultrasonic sensors or even radar. Thus, the complete system can be considered as an HiL.

SiL or MiL tests might be performed with stimuli defined within the test cases, but as systems get more complex, they cannot be tested independently. To cover more aggregated scenarios, one can use, for example, "Virtuelle Probefahrt 2.0." This method is a mixture between real hardware operated on HiL benches and simulated stimuli (sensors, road network, etc.). An in-depth explanation has been given by Miegler et al. [21]. Standardized interfaces to exchange modules, for example, a driver model or road networks, are mandatory. The system is capable of replacing/mixing stimuli with either simulated data (using, e.g., VTD) or recorded real drive data (e.g., captured at road test drives). The modular architecture enables engineers to replace sensor models with different simulation approximations. Using an HiL test bed, the simulated data can be fed into a real ECU or even a network of ECU.

Rather new technologies, such as Vehicle in the Loop (ViL), are using a real vehicle but virtual environment to perform initial test without harming real vehicles. This means a human driver might sit within a real car driving along an empty road. The driver will have to wear virtual reality glasses where he can see the simulated traffic. The simulation will generate sensor inputs to be fed into a System Under Test (SUT) or capture them by real objects on the road. The system will be able to generate real force feedback (via injecting the real vehicle systems) and record the driver's reaction. ADAS development can benefit due to early test chance.

According to Bock et al. [8], the ViL method is usable. Within their work, they performed user study with 36 persons followed by later work [9] where they verified the virtualization assumptions stating that ViL is able to generate realistic driving impressions within a very early stage of development.

Miegler et al. [22] present a new HiL approach where they use existing ECU and real-time rapid prototyping systems to integrate unavailable ECU; for example, vehicle dynamics might be computed on an HiL, and environmental data, for example, traffic participants, is simulated on a PC. Overall, scenario control is part of the simulation; Reset/Replay is possible. Rapid prototyping modules are possible using Automotive Data and Time-Triggered Framework (ADTF) and VTD, where ADTF ensures the connectivity to HiL systems and VTD realizes the environment simulation.

6 Software Fault Prediction (SFP): A New Idea to Be Integrated

As presented in Sect. 2, automotive software development follows restrictive settings in terms of coding guidelines and, for example, commit policies. As outlined by Altinger et al. [3], it is possible to gain advantages. The basic idea in brief is as follows: use code measurements (LOC, cyclomatic complexity by McCabe, etc. as analyzed by Herbold et al. [16]) for every commit; extract bug reports from the ITS and derive a ground truth for bug and fault-free commits; use this data to train a machine learning algorithm, for example, Support Vector Machine (SVM), Näive Bayes (NB), and Random Forest (RF); repeat the measurement step at the time of commit and use the trained machine learning system to derive a probability if the actual commit contains a bug. Figure 7 presents the involved tools and process steps to perform such measurements. Further details and sample projects with measurements are outlined by Altinger et al. [3]. The dataset is publicly available by Altinger [1]; further datasets, including industry grade, are outlined by Sayyad et al. [29].

Altinger et al. [3] could demonstrate a rather high true positive hit rate on fault predictions at more than 90%. They conclude that their data is better than others, for example, Zimmermann et al. [33], on an open source projects; and Nagappan et al. [25], on commercial software, as the changes between commits are smaller and the nature of generated code from model-driven development approaches results within more homogeneous code structure compared to text-based software developers. Further, they could rely on the data in the ITS, as there were policies to enforce correct commit logs by every single developer. Thus, the quality of the measured data is much higher than, for example, on Zimmermann et al.'s [33] analysis on the open source tool Eclipse, which suffers from no clear differentiation between feature commit and a bug fix or needs to deal with blank commit messages.

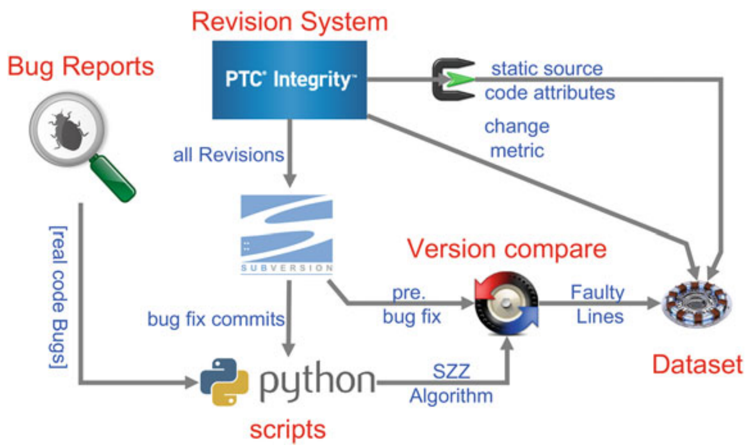

Fig. 7 Workflow to extract metric values for dataset; adopted from Altinger et al. [3]

Due to a low bug rate, as there are only 2.6–15% samples in the bug class, Altinger's [1] datasets suffer from imbalanced class distribution. To overcome this, Altinger et al. [5] suggest to use over- and undersampling to enhance predictive performance. They succeed in showcasing an improved performance using an NB. They state when using up- or downsampling, one has to choose between precision (all reported faults are true bugs) and recall (finding all bugs), both of which cannot be increased at the same time.

The obvious idea, to use a trained prediction model from one project to another, for example, at an early stage where there are too few metric data to train, does not seem to perform well. Altinger et al. [3] performed a cross-project prediction including well-known metric transfer methods but could not archive a prediction performance above the random hit rate. These findings are in line with other industry data as presented by Zimmermann et al. [34].

This method seems to be a promising extension to the W-development process (see Fig. 5), as it could be an early indicator to define the various software tests. As known from Altinger et al. [2], test engineers need to spend free testing budget based on their experience; thus, SFP might aid in these decisions. As of the more s̈tatisticn̈ature, SFP cannot replace a full test suite.

References

1. Altinger H (2015) Dataset on automotive software repository. http://www.ist.tugraz.at/_attach/Publish/AltingerHarald/MSR_2015_dataset_automotive.zip
2. Altinger H, Wotawa F, Schurius M (2014) Testing methods used in the automotive industry: results from a survey. In: Proceedings of JAMAICA. ACM, New York, pp 1–6

3. Altinger H, Herbold S, Grabowski J, Wotawa F (2015) Novel insights on cross project fault prediction applied to automotive software. In: El-Fakih K, Barlas G, Yevtushenko N (eds) Testing software and systems, vol 9447. Springer, Berlin, pp 141–157. http://dx.doi.org/10.1007/978-3-319-25945-1_9
4. Altinger H, Dajsuren Y, Sieg S, Vinju JJ, Wotawa F (2016) On error-class distribution in automotive model-based software. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering, IEEE, Piscataway, pp 688–692. https://doi.org/10.1109/SANER.2016.81
5. Altinger H, Herbold S, Wotawa F, Schneemann F (2017) Performance tuning for automotive software fault prediction. In: 2017 IEEE 24th international conference on software analysis, evolution, and reengineering. IEEE, Piscataway. https://doi.org/10.1109/SANER.2017.7884667
6. Audi AG (2017) Audi A8 (Typ 4N) Selbststudienprogramm 662. Technical manual. AUDI AG
7. Bergmann R, Walesch R (2012) HiL strategie audi. In: 6. dSpace Anwender Konfernz 2012, dSpace. http://www.dspace.com/shared/data/pdf/ankon2013/tag1_pdf/2_audi_walesch_robert_bergmann_richard.pdf
8. Bock T, Maurer M, Farber G (2007) Validation of the vehicle in the loop (VIL); a milestone for the simulation of driver assistance systems. In: 2007 IEEE Intelligent vehicles symposium, pp 612–617
9. Bock T, Maurer M, Meel F, Müller T (2008) Vehicle in the loop. ATZ Automobiltech Z 110(1):10–16. http://dx.doi.org/10.1007/BF03221943
10. Bock F, Homm D, Siegl S, German R (2016) A taxonomy for tools, processes and languages in automotive software engineering abs/1601.03528. http://arxiv.org/abs/1601.03528
11. Broy M, Kruger I, Pretschner A, Salzmann C (2007) Engineering automotive software. Proc IEEE 95(2):356–373. https://doi.org/10.1109/JPROC.2006.888386
12. Capers J (2009) A short history of the cost per defect metric. www.semat.org
13. Charette RN (2009) This car runs on code. IEEE Spectr 46(3):3
14. Duba GP, Bock T (2008) ATZextra Worldw 13:56. https://doi.org/10.1365/s40111-008-0055-0
15. Dupuis M, von Neumann-Cosel K, Weiss C (2010) Virtual test drive vereinheitlichung der simulationsumgebung für SiL-, HiL-, DiL-und ViL-tests bei der entwicklung von fahrerassistenz- und aktiven sicherheitssystemen
16. Herbold S, Grabowski J, Waack S (2011) Calculation and optimization of thresholds for sets of software metrics. Empir Softw Eng 16(6):812–841. https://doi.org/10.1007/s10664-011-9162-z
17. ISO TC 22 SC 3 (2011) ISO 26262:2011:Road vehicles – Functional safety. International. www.iso.org
18. Jin-Hua L, Qiong L, Jing L (2008) The w-model for testing software product lines. In: International Symposium on Computer Science and Computational Technology, 2008. ISCSCT'08, vol 1, pp 690–693
19. Kiffe G, Bock T (2013) Standardisierte entwicklungsumgebung fuer die softwareeigenentwicklung bei audi. In: 7. cSapce User Conference, dSpace, https://www.dspace.com/de/gmb/home/company/events/dspace_events/archive_2013/ankon2013.cfm
20. Mathworks T (2014) Mathworks automotive advisory board checks (MAAB). http://de.mathworks.com/help/slvnv/ref/mathworks-automotive-advisory-board-checks.html
21. Miegler M, Nentwig M (2015) Testing of piloted driving on virtual streets. ATZ Worldw 117(9):16–21. http://dx.doi.org/10.1007/s38311-015-0044-7
22. Miegler M, Schieber R, Kern A, Ganslmeier T, Nentwig M (2009) Hardware-in-the-loop test of advanced driver assistance systems. ATZ Elektron Worldw 4(5):4–9
23. Motor Industry Software Reliability Association (2004) MISRA-C:2004 - Guidelines for the use of the C language in critical systems, 2nd edn. MISRA
24. Müller DIFSO, Brand IM, Wachendorf S, Schröder DIFH, Szot DIFT, Schwab DIS, Kremer B (2009) Integration vernetzter fahrerassistenz-funktionen mit HiL für den VW passat CC 14(4):60–65

25. Nagappan N, Ball T, Zeller A (2006) Mining metrics to predict component failures. In: Proceedings of the 28th international conference on software engineering, pp 452–461
26. Nentwig M, Stamminger M (2011) Hardware-in-the-loop testing of computer vision based driver assistance systems. In: 2011 IEEE Intelligent vehicles symposium (IV), pp 339–344. https://doi.org/10.1109/IVS.2011.5940567
27. NHTSA USDoT {and} USAgov (1997) Office of defects investigation (ODI) recalls database. www-odi.nhtsa.dot.gov
28. Reactis (2015) Achieving ISO 26262 compliance with reactis. http://www.reactive-systems.com/papers/iso-26262.pdf
29. Sayyad Shirabad J, Menzies T (2005) The PROMISE Repository of Software Engineering Databases. http://promise.site.uottawa.ca/SERepository, published: School of Information Technology and Engineering, University of Ottawa, Canada
30. Shea T (2010) Why does it cost so much for automakers to develop new models? http://www.autoblog.com/2010/07/27/why-does-it-cost-so-much-for-automakers-to-develop-new-models/
31. VW Aktiengesellschaft (2014) Geschaeftsbericht 2014. http://geschaeftsbericht2014.volkswagenag.com/konzernlagebericht/nachhaltige-wertsteigerung/forschung-und-entwicklung/f-e-kennzahlen.html
32. WARDS Auto (2013) U.S. car and truck sales, 1931–2013. www.wardsauto.com
33. Zimmermann T, Premraj R, Zeller A (2007) Predicting defects for eclipse. In: International Workshop on Predictor models in software engineering, 2007. PROMISE'07: ICSE Workshops 2007. IEEE, Piscataway, p 9
34. Zimmermann T, Nagappan N, Gall H, Giger E, Murphy B (2009) Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, pp 91–100