Chapter 14
Open Dependable Power Computing Platform for Automated Driving

Andrea Leitner, Tilman Ochs, Lukas Bulwahn, and Daniel Watzenig

14.1 Introduction

Future automated driving systems need to be able to handle any possible traffic situation safely without relying on human supervision. These systems have two seemingly contradictory characteristics: first, they must be verifiably highly reliable because malfunctions could endanger passengers and other traffic participants, and second, they consist of complex and computer-intensive application software components. Figure 14.1 shows that an automated vehicle will operate in an ecosystem of infrastructure, cloud, and back-end systems, such as traffic lights, authority services, software distribution services, or map services. The vehicle will connect to those services via wide-area wireless internet links or vehicle-to-vehicle communication media defined by ETSI.1 Such an automated driving system is composed of a network of redundant electronic control units (ECUs)

1www.etsi.org/

A. Leitner (✉)
Virtual Vehicle Research Center, Graz, Austria
e-mail: andrea.leitner@v2c2.at

T. Ochs
BMW AG, Munich, Germany
e-mail: Tilmann.Ochs@bmw.de

L. Bulwahn
Lukas Bulwahn, BMW AG, Munich, Germany
e-mail: Lukas.Bulwahn@bmw.de

D. Watzenig
Virtual Vehicle Research Center and Graz University of Technology, Institute of Electrical Measurement and Measurement Signal Processing, Graz, Austria

© Springer International Publishing Switzerland 2017
D. Watzenig, M. Horn (eds.), Automated Driving,
DOI 10.1007/978-3-319-31895-0_14
that are connected to the vehicle’s sensors and actuators through a highly reliable, adequately responsive vehicle network.

Cars are becoming cyber-physical systems that need to respond in real time to dynamic and complex environmental situations. Given the current and future road infrastructure, it is mandatory to reach a safe vehicle state, i.e., pulling over a car at a situation-dependent speed to a safe stop on every road in every condition in the event of a single safety-critical system malfunction. This is especially challenging because of the high complexity and the partially probabilistic nature of intermediate results. Especially for active driving functions, shutdown of the system will not be a safe state; instead, the system must tolerate certain failures and support a fail-operational mode.

Automated driving therefore requires the introduction of a new type of automotive systems called cognitive systems (marked as “C” in Fig. 14.1). Figure 14.2 illustrates the differences between this new type of system and well-known control systems (marked as “D” in Fig. 14.1). A control system processes information from inside the vehicle, while cognitive systems in addition receive and process environmental information (Table 14.1). The basic elements of a cognitive system are thus environment perception, decision making, and initiation of actions based on machine learning and information fusion. In contrast to traditional deeply embedded control systems, which are generally perceived as limited in scope, cognitive systems require much more computational resources, e.g., to solve complex optimization problems in real time. Furthermore, software architectures for such systems are significantly more sophisticated than for traditional control systems. This paradigm shift necessitates the development of new platforms that support these applications in an efficient and effective manner.
The current state-of-the-art software platform for deeply embedded control ECUs and early cognitive computing ECUs is AUTOSAR. The main concern with the current AUTOSAR platform in this context is the static runtime environment together with the signal-oriented communication infrastructure (i.e., restricted types and data formats). In particular, fixed scheduling and fixed communication relationships are not ideal in a multi-core environment because they restrict the efficiency of these systems. A future dependable power-computing software platform should therefore provide a more efficient solution. Due to its monolithic and highly optimized design, a partial update of the AUTOSAR specification is difficult. This means that the future of AUTOSAR will be an additional, new AUTOSAR Adaptive Platform supporting the needs for automated driving. Activities in this direction have already been started but are of course currently only in the initial stages.

The software architecture for a cognitive computing ECU can be decomposed into two layers: the application software layer and the software platform layer. The higher-level application software layer contains the main cognitive software functions (e.g., high-level recognition of traffic situations, prediction of other traffic participants’ behavior, planning of the vehicle’s maneuvers) for automated driving.

2http://www.autosar.org/
This part is the differentiating part for car manufacturers and suppliers. The software platform layer provides basic services, e.g., communication between the software functions, and abstraction of the concrete computing hardware—this is the non-differentiating part. Collaboration at this level will accelerate the development of advanced and innovative functions. The focus of this article is on this lower-level software platform layer.

Key issues addressed in this article are as follows: Sect. 14.2 addresses a set of initial requirements for a computing platform, including functional capabilities, reliability, safety, and security demands.

Section 14.3 discusses why we think openness, and in particular open-source development, is the most promising solution. Section 14.4 describes the basic ideas of the envisaged in-vehicle dependable power-computing software platform for reliable cognitive data processing. Section 14.5 describes the steps that need to be taken in order to realize such a platform, and Sect. 14.6 sketches the requirements for an open-source development process in a safety-critical context. Section 14.7 concludes the paper.

14.2 Requirements for an Open Dependable Power Computing Platform

Cognitive systems and reliable cognitive data processing for advanced assisted and automated driving functions have significantly different requirements compared to those for systems using traditional automotive control algorithms. The most important requirements are (i) very large and highly dynamic resource demands with respect to communication bandwidth, memory and mass storage, and raw CPU power; (ii) in-vehicle update of parameter data, models, and algorithms; and (iii) the probabilistic nature of some basic algorithms.

Here we can only present a preliminary set of initial requirements for such a platform, since the specific requirements will only become available in the course of product development. We have extrapolated resource demands and use cases from existing systems and research projects [3].

Openness in the context of a dependable power-computing software platform can be understood in two ways. First, it must be open from a production point of view (pushing toward an open-source approach), and second, the system must be open to deal with the fast evolution of cognitive software, because continuous learning from field experience and short iteration cycles with frequent rollouts into vehicles are essential for usability, robustness, safety, and security.

We strongly believe that future market differentiation will come from up-to-date information and advanced functional software (application software) rather than vehicle E/E architectures and software platforms. The latter are becoming increasingly non-differentiating, and therefore joint development will help to share significant development efforts.
The benefits of an open-source approach furthermore include the possibility to create a community able to contribute to the evolution of the platform as well as the development of applications without depending too much on tool-chain providers. Using an open-source approach enables research institutions and SMEs to develop and test their applications on existing state-of-practice platforms. Currently, research projects seem to be slowed down by focusing too much on the software architecture [4, 5] instead of the more innovative part: the application. An open-source platform would thus reduce time to market for new innovative functions.

**Complexity** Software for automated driving functions has tens of thousands of lines of code and makes use of complex and dynamic data structures, such as graphs or sparse matrices. In order to support the development of this kind of software, the platform needs to support (i) the integration of loosely coupled application software components to minimize covert interference, (ii) a rich interface definition language to specify application software component interfaces unambiguously, (iii) high-level programming languages to reduce code complexity, and (iv) mainstream software engineering methods and technology. The latter fosters the use of mature and state-of-the-art technology in order to leverage experience and progress in the general IT industry (or other domains) while minimizing friction for application development due to immature or unfamiliar niche products.

**Computing Performance** Research projects for automated driving use modern personal computers to their full capacity. This means memory usage in the gigabyte range, CPU usage in the gigaflop range, and heavy use of hardware acceleration and parallel data processing. Hence, the platform should provide (i) a hardware-independent programming interface for acceleration hardware to enable easy porting of application software components, (ii) symmetric multiprocessing to leverage component level parallelism on mainstream processors, and (iii) performance-optimized processor architectures to maximize computational throughput.

Typical control software utilizes predefined fixed scheduling in order to ensure correct timing behavior. This is not meaningful for cognitive software with its high computational power requirements. These systems require multi-core systems, which are very inefficient with fixed scheduling. A possible solution could be the specification of dedicated timing properties (e.g., deadlines, criticality, etc.) in order to support dynamic scheduling and thus more efficient utilization of computing power.

**Mixed Criticality** Automated driving functions are highly safety critical because malfunctions can cause harm to passengers and other traffic participants. In order to assure the safe operation of an automated vehicle, a safety concept has to be defined at the vehicle level and decomposed to derive safety requirements for all relevant vehicle components. Following the ISO 26262 standard [6] for the development of safety-critical automotive E/E systems, vehicle components are classified according to four integrity levels from the lowest Automotive Safety Integrity Level (ASIL) A to the highest level ASIL D, depending on the consequences of failures. The platform should be able to host application software components with different
integrity levels (mixed criticality) and must ensure that those with lower integrity level do not interfere with higher-integrity components. This requirement is called “freedom from interference” in ISO 26262, and it means that it must be ensured that shared memory is protected from access violation, tasks are executed in the correct order, and functions are able to finish within their real-time boundaries.

To execute mixed-critical software functions, the platform should support (i) the coexistence of application software components with different integrity levels (mixed criticality), (ii) misbehavior detection for application software components and restart or shutdown of affected components to increase robustness, (iii) control flow monitoring in application software components and misbehavior detection, (iv) data flow monitoring between application software components and misbehavior detection, and (v) standardized software and error propagation models to support automated safety analyses.

**Decomposition** Based on personal communication with various hardware vendors, we assume that a platform will at most provide integrity levels up to ASIL B due to a lack of high-integrity processing hardware with the required performance. Furthermore, development of high-integrity software is very time-consuming and limited to a certain level of complexity. Therefore, cognitive software should be of limited ASIL, while hard safety functions should be shifted to more traditional and static high-integrity computer systems.

If required, higher-integrity levels can be achieved by vehicle-level redundancy. This concept is called ASIL decomposition in ISO 26262 and means that ASIL D can be attained by using two redundant, independent ASIL B components.

Of course, it would also be possible to qualify the platform itself, or the applications implemented on it, at a higher level. Techniques such as safety kernels [7], both hardware and software, allow failures of the cognitive processing software to be tolerated and hence a higher ASIL to be achieved. Using architectural patterns recommended by IEC 61508 [8] and other standards, an overall level of ASIL C can be supported. Even ASIL D would be possible with formal proof of the protection mechanisms (e.g., the safety kernel). Wika et al. [9] have demonstrated the feasibility of such a proof.

In order to support decomposition, the platform should support (i) the execution of application software components with integrity levels up to ASIL B and (ii) failure detection for hardware and platform software components and silent shutdown of affected partitions/containers to support vehicle-level redundancy.

**Time Sensitivity** The timing properties of automotive systems are among the most important properties. It is not only important to demonstrate that the system meets its timing requirements; the developers also need to show that activities are performed in the correct order, the system does not deadlock or live-lock, and the system degrades in case of a failure.

Automated driving systems have additional real-time requirements because outdated environment data or lagging maneuver planning can lead to oscillating dynamic behavior and collisions. From our experience, we assume that a sporadically occurring maximum jitter below 100 µs in the application components...
will not lead to a hazardous event. Recent work by the Open Source Automation Development Lab (OSADL)\(^3\) shows that interrupt latency below 100 \(\mu\)s can be achieved on mainstream processing hardware using Linux with the RT Preempt patch. Hence, the software platform must provide (i) deterministic timing behavior with a maximum jitter of 100 \(\mu\)s to ensure replicable behavior, (ii) real-time scheduling to meet the timing requirements of application software, and (iii) timing monitoring of application software components and detection of timing violations with 100 \(\mu\)s tolerance.

**Volatility** Research in computer vision and artificial intelligence will evolve rapidly in the near future, and several research projects are currently developing infrastructure for automated driving.\(^4\),\(^5\) As already argued in the paragraph on openness, we assume that car manufacturers must supply software updates through a remote connection to provide the latest functionality and integration of locally available infrastructure services. Hence, the platform should support (i) agile development practices for software components to mitigate low concept maturity, (ii) safe and secure remote update of application software components to maintain software continuously, (iii) a modular safety concept to integrate independently developed application software components, and (iv) safe and secure remote addition and removal of application software components to provide extensibility.

**Automotive Specifics** Automated driving systems are integrated into vehicle networks, are managed by standardized diagnostic functions, and interact with vehicle state management and energy management. Hence, the platform should support (i) standardized diagnostic and vehicle management functions, (ii) integration of vendor-specific diagnostic and vehicle management functions, and (iii) automotive-specific communication protocols such as Automotive Ethernet [10], CAN, or FlexRay.

### 14.3 Why Qualifiable Open Source Is Appropriate

There are several possible approaches to provide software platforms. In the following, we discuss the advantages and disadvantages of the different approaches and explain why we think that open source is the best choice:

- **Custom**—The advantage of this approach is that the entire design and development approach can be tailored to the specific application, but most companies do not have the required specialist skills. Furthermore, it entails significant cost.

\(^3\)http://www.osadl.org  
\(^4\)http://kofas.de/  
\(^5\)http://www.simtd.de/
• **COTS**—Many COTS-based systems have the advantage of more “formal” software engineering methods behind them than open-source development, but this comes with significant upfront and recurring costs. Additionally, the user is dependent on the vendor to implement required changes.

• **Qualifiable open source**—Another approach is to build the platform from existing open-source software and further develop and maintain it using an open-source approach. With the qualifiable open-source approach, not only the delivered software but also the software lifecycle, including specification, documentation, and safety case, is provided using free/libre/open-source software (FLOSS) licensing concepts [11]. This approach has several advantages compared to proprietary solutions:
  
  – Higher quality: widely used software matures more quickly, since it incorporates experiences from various use cases.
  – Higher confidence: everybody, including educational institutions, can assess and rate risk classifications and the effectiveness of safety measures.
  – Higher agility: innovative car manufacturers, application software developers and integrators who need additional platform capabilities can collaborate on new features, implement them a dedicated branch, and use them in their product development before the changes are integrated into the main branch.
  – Lower cost: shared costs for development and qualification of this complex platform.

The most important and famous open-source operating system is the Linux operating system (OS). A major advantage of the Linux real-time OS (RTOS) is that it has been demonstrated as adequately (in terms of flexibility and reliability) supporting a wide range of different platforms and applications. In particular, it has been proposed and evaluated for use in critical systems [12, 13] and has been used in cognitive systems and reliable cognitive data processing applications [14]. From an economic point of view, the Linux operating system is the most successful open-source project and has been developed in a joint manner. It is hard to estimate the cost of its development and consequently its worth. Nevertheless, there have been studies that investigated what it would have cost to develop Linux in a typical company in the USA. David A. Wheeler [15] determined in 2002 a cost of more than $1.2 billion. An updated study in 2008 showed that it would cost $10.8 billion to develop the Fedora 9 Linux distribution at this time by traditional proprietary means. This shows the tremendous value which can be created by the open-source community and which could be exploited by the automotive domain. For these reasons, Linux RTOS is an efficient and cost-effective option.

Of course, the Linux RTOS currently has one big disadvantage: there is presently no appropriate qualification approach for the Linux RTOS itself and no certification strategy for safety-related systems using this operating system.

Open-source projects, such as the Linux operating system, have the big advantage that they employ stringent development processes [16, 17] and deliver software of very high quality, but they do not fulfill the requirements of current safety standards, such as ISO 26262, for software in automotive systems. These standards impose
strict demands on project management, developer qualification, risk management, requirements management, quality assurance, and documentation. Therefore, open-source software cannot be used in safety-critical systems without further activities. Nevertheless, a related project called SIL2LinuxMP [18] is attempting to show that open-source processes fulfill most of the requirements but by different means.

Still, the following issues need to be resolved: (i) a qualification strategy for Linux RTOS, which also supports the co-evolution of software and safety case, (ii) a configured and ASIL B qualified version of Linux RTOS suitable for use in safety-related cognitive systems, and (iii) guidance on how this configuration can be maintained.

The idea of using an open-source platform in a safety-critical context is not new. An initiative in the railway domain has resulted in a project called OpenETCS, which aims to develop a software kernel for the European train control system based on open source and open proof. Another project called SIL2LinuxMP [18] plans to certify the base components, i.e., the boot loader, root file system, Linux kernel, and C library bindings, of an embedded GNU/Linux real-time operating system compliant with safety integrity level 2 (SIL2) according to safety standard IEC 61508 [8], which roughly corresponds to ISO 26262 integrity level ASIL B.

For the automotive domain, GENIVI is another example for an open-source initiative, though not in the context of a safety-critical system. Many industrial partners have joined forces to develop a reusable, open-source in-vehicle infotainment (IVI) platform.

### 14.4 Considerations for the Platform Architecture

Figure 14.3 shows an architectural overview of the dependable power-computing software platform. The dependable power-computing software platform presumes high-performance processor hardware that extends current mainstream microcontrollers for mobile devices by additional integrity mechanisms and reliability figures that are able to fulfill the requirements of the ISO 26262 safety standard for integrity levels up to ASIL B. The combination of this hardware and the dependable power-computing software platform delivers the overall dependable power-computing platform. An API built on the software platform abstracts the underlying implementation from the application which uses it.

A dependable power-computing software platform consists of two layers: (i) the platform foundation layer and (ii) the platform services layer. The platform foundation layer consists of system software modules that expose APIs to application software components and implement basic platform capabilities, such as hardware abstraction, mass storage, network communication, power management,
and process control. In addition, low-level safety and security mechanisms, such as temporal and spatial isolation, mandatory access control and runtime monitoring are provided here. A large part of this platform foundation layer can be implemented by a qualified open-source RTOS based on the Linux operating system. The platform service layer consists of software components that implement higher-level management and monitoring functions, such as state management, graceful degradation, update over the air, diagnostics, and real-time intrusion detection. This layer should also be implemented using an open-source approach and should reuse as much existing software as possible.

The differentiating part for OEMs will still be the application software. The dependable power-computing software platform also allows the partitioning of application software components and provides protection mechanisms for resilience against malicious attacks, design flaws, and hardware faults. Personalities allow same base platform to be adapted to different application domains (e.g., IMA for aerospace, ROS for robotics, etc.).

14.5 Steps Toward an Open Dependable Computing Platform

For a driverless vehicle, a core component such as an operating system along with its supportive environment (e.g., development tools) needs to be qualified. Any qualification is built around a well-specified set of traceable processes and procedures. The qualification of Linux as an operating system in a safety-critical
context requires two main things: first, selection of the kernel and core components to be qualified, and second, definition of a qualification strategy for open source that is compliant with the automotive safety standard ISO 26262.

Linux is an existing component and has not only a broad range of functionality but also varying maturity of concepts and implementations. The starting points for OS selection are well-defined selection criteria and evaluation of available operational data sources. While many of the criteria will need to remain qualitative criteria, quantification is important for managing the dynamics of open-source projects, which often exhibit change rates of multiple changes sets per hour. Definition of quantitative selection criteria and implementation of their automation can significantly ease the utilization of open-source components in safety-related and dependable systems. Based on these criteria, the Linux kernel and core OS components to be used as the qualification target can be selected.

Relevant functional safety standards, notably IEC 61508 and ISO 26262, are built around well-defined processes to mitigate risks related to systematic faults in complex systems. Selection of methods from the set of accepted state-of-the-art methods is a first step, but not all of the traditional methods developed for custom systems are suitable for open-source components. Thus, there might be a need for new methods, processes, and procedures for the formal qualification of a use-case-driven subset of functional and nonfunctional capabilities of Linux, open-source tools, and support libraries.

A starting point for qualifying open-source is to develop a complete mapping/interpretation of standards. Because coverage of systematic faults relies on the development process, the main need is to develop documentation of the detailed procedures that are used and follow-up gap analysis of the current development life cycle. This should be based on the evaluation of available process data and assurance criteria for data and procedures.

As mentioned before, the platform foundation and the platform service layer should be built from existing and operationally proven open-source software wherever possible. This seems to be reasonable, because open-source is a vast resource pool of technology. Nevertheless, a key obstacle to utilizing it is the lack of an accepted generic procedure to integrate these components into qualified systems. Although key standards do give some high-level guidance, there is a lot of room for interpretation. Narrowing this range of interpretation is a key risk mitigation strategy, in order to simplify the use of open-source components in industrial projects in general. The open-source community has well-established methods and policies, some of which need minor adjustments to satisfy the qualification needs set out in relevant standards. One key aspect therefore is active feedback of modifications and procedural changes to the open-source community in order to achieve the long-term objective of a maintainable qualification methodology.
14.6 Open-Source Software Development Process

As mentioned before, open-source projects employ stringent development processes, which might need to be adapted for the use in a safety-critical context. Figure 14.4 describes the roles and interactions of an open-source development process and the actions necessary to transform a common open-source process into a qualifiable open-source process.

In any case, we distinguish between collaborative development, which is done by the open-source community, and product development, which is subsequently done within a company based on the jointly developed deliverables.

A central aspect of any open-source development is the management organization (e.g., the Linux foundation), which provides the infrastructure (e.g., repositories, etc.) and defines and monitors the processes.

The contributors of course play an important role because they do the actual work on the deliverables according to the defined process. Before they are able to contribute, they have to accept a Contributor License Agreement which governs intellectual property rights. The user (OEM or Tier 1) has to accept the open-source license and usually drives the development. The user therefore engages service providers (software experts) to contribute to the open-source project. This potentially leads to a huge business opportunity for service providers, who can offer

Fig. 14.4 From open-source software to qualifiable open-source software
their services to the users. The big advantage for the user is that required changes or adaptations can be implemented efficiently without depending on a single supplier.

A qualifiable open-source process needs to be enhanced by a more stringent safety development process. This needs to be driven by the management organization, which also needs to ensure compliance with the processes. The management organization is furthermore responsible for establishing a safety culture. This means that it must be clearly described and communicated to the contributors how functional safety will be part of the product and how it should be handled within the development process.

One important document in a qualifiable process is the Developer Interface Agreement (DIA) as defined in ISO 26262. The DIA is a multilateral contract that determines each party’s responsibilities. Especially in a safety-critical context, there is a need for additional deliverables which document the respective safety activities. This is especially important because the final responsibility lies with the user (in this case the OEM or Tier 1) who integrates the open-source development in the final product. At the end, there needs to be a safety case, which documents that all relevant risks have been identified and that the corresponding mitigation measures have been taken.

### 14.7 Conclusion

This article discusses the needs of an automated driving computing platform and highlights the advantages of open-source development. Nevertheless, we are fully aware that these goals are not easy to achieve without major efforts by various parties. There are several open research questions which need to be targeted for the successful implementation of complex cognitive systems. One of these questions is related to functional safety and how it can be addressed in this complex and highly dynamical context. The high degree of automation requires a transition from fail safe to fail operational, because it is nearly impossible to determine a safe state for an automated vehicle. This requires more advanced redundancy patterns for automotive software architectures, as for example shown in Fig. 14.5.

Another important issue is the agility of cognitive systems, which requires a modular safety model. Currently, a system has to be considered as a monolithic entity. In order to support system properties required for future highly innovative vehicle functions, for example, remote updates (or regular functional updates in general) modularization is a key feature.

Cognitive systems are essential for automated driving but are still a challenging topic. This new type of system requires completely new and more advanced software platforms, which are able to cope with newly emerging requirements.

In this article, we describe the main requirements for such a qualifiable platform and why we think that joint development using an open-source approach is the way forward.
Fig. 14.5  Sample architecture for fail-safe and fail-operational tasks

References

1. H. Martorell, J.-C. Fabre, M. Roy, R. Valentin, Improving Adaptiveness of AUTOSAR Embedded Applications, in Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC ’14) (ACM, New York, NY, 2014), pp. 384–390
2. M. Bechter, Softwareplattform für zukünftige automotive Anwendungen, http://www.informatik.tu-cottbus.de/ase2015/Abstract01.pdf. Accessed 12.10.2015
3. S. Holder, M. Hörwick, H. Gentner, Funktionsübergreifende Szeneninterpretation zur Vernetzung von Fahrerassistenzsystemen. Automatisierungssysteme, Assistenzsysteme und eingebettete Systeme für Transportmittel (AAET 2012) (2012)
4. S. Sommer, A. Camek, K. Becker, C. Buckl, A. Zirkler, L. Fiege, M. Armbruster, G. Spiegelberg, A. Knoll, Race: A Centralized Platform Computer Based Architecture for Automotive Applications (2013)
5. M. Goebl, M. Althoff, M. Buss, G. Faerber, F. Hecker, B. Heissing, S. Kraus, R. Nagel, F.P. Leon, F. Rattei, M. Russ, M. Schweitzer, M. Thuy, C. Wang, H.-J. Wuensche, Design and Capabilities of the Munich Cognitive Automobile, in Proceedings of IEEE Intelligent Vehicles Symposium, pp. 1101–1107, Eindhoven, The Netherlands, Jun 2008
6. ISO 26262:2011, Road vehicles—Functional safety International Standard (Parts 1–10). International Organization for Standardization, 1st edn (2011)
7. J. Rushby, Kernels for Safety? in T. Anderson (ed.), Safe and Secure Computing Systems (Blackwell Scientific Publications, Malden, 1989), Chapter 13, pp. 210–220
8. CENELEC, IEC 61508 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. International Electrotechnical Commission (IEC), 2nd edn (2010)
9. K. Wika, J. Knight, On the Enforcement of Software Safety Policies, in Proceedings of the 10th Annual IEEE Conference on Computer Assurance, Jun 1995
10. Broadcom Corporation, BroadR-Reach® Physical Layer Transceiver Specification for Automotive Applications, http://www.ieee802.org/3/1TPCESG/public/BroadR_Reach_Automotive_Spec_V3.0.pdf. Accessed 12.10.2015
11. K.-R. Hase, Open Proof” for Railway Safety Software—A Potential Way-Out of Vendor Lock-In Advancing to Standardization, Transparency, and Software Security, in FORMS/FORMAT 2010, ed. by E. Schnieder, G. Tarnai (Springer, Berlin, 2011), pp. 5–38
12. CSE International Limited, Preliminary Assessment of Linux for Safety Related Systems, HSE (2002)
13. R. Kammerer, *Linux in Safety-Critical Applications*. OSADL Academic Works (OSADL, Heidelberg, 2011)
14. M. Goebl, G. Farber, A Real-Time-Capable Hard-and Software Architecture for Joint Image and Knowledge Processing in Cognitive Automobiles, in *Intelligent Vehicles Symposium*, 2007 IEEE, pp. 734–740, Jun 2007
15. D.A. Wheeler, More Than a Gigabuck: Estimating GNU/Linux’s Size (2001), [http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html](http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html)
16. J. Corbet, *How the Development Process Works* (The Linux Foundation, San Francisco, 2011)
17. A. Mockus, R.T. Fielding, J.D. Herbsleb, Two Case Studies of Open Source Software Development: Apache and Mozilla. ACM Trans Softw Eng Methodol **11**(3), 309–346 (2002)
18. OSADL Project: SIL2LinuxMP, [http://www.osadl.org/SIL2LinuxMP.sil2-linux-project.0.html](http://www.osadl.org/SIL2LinuxMP.sil2-linux-project.0.html). Accessed 12.10.2015