Automotive Software: Characteristics and Reengineering Challenges

Vincent Schulte-Coerne, Andreas Thums, Jochen Quante

Robert Bosch GmbH
Corporate Sector Research and Advance Engineering Software
P. O. Box 30 02 40, 70442 Stuttgart, Germany

{Vincent.Schulte-Coerne|Andreas.Thums|Jochen.Quante}@de.bosch.com

Abstract

Automotive software is different from the kind of software that is usually addressed by current reengineering research. This paper gives an overview of the particular conditions and challenges that we face in the automotive domain.

1 Introduction

It is a well-known fact that software ages [7]. Nearly 30 years ago, Lehman and Belady noted that software has to change continuously in order to remain useful, and that a software's complexity increases as the software is changed [4]. The combination of these "Lehman's laws" means that complexity is steadily increasing – if no counteractive measures are taken.

Lehman's laws are valid as well for automotive software. Software is becoming more and more important in this domain. It enables new security, comfort, and economic functions which allow manufacturers to set themselves apart from competitors. For example, a typical engine control system contains about 500,000 lines of C code. However, the increasing amount of software in automobiles comes along with an increasing complexity, rising time and cost pressure, and high quality demands. This necessitates measures for controlling and limiting this ever-increasing complexity.

Software Reengineering is the discipline that tries to address these problems. Unfortunately, a large portion of published reengineering techniques and methods do not seem to be applicable to automotive software, or even to embedded software in general – despite the fact that a large and increasing share of software is developed for the embedded domain. Also, automotive software faces a number of special challenges that have not been addressed by the community so far. Although challenges for software engineering (i. e., mostly forward engineering) in this particular domain have been addressed by a number of publications [1, 8, 3], this is not the case for reengineering. Therefore, in this paper, we point out the special conditions and challenges for reengineering automotive software. We want to show up new research directions and encourage an increased interest for these particularities.

2 Requirements and Implications

Automotive software faces a number of special requirements that have to be considered. In this section, we present those and discuss their implications for reengineering.

Hard real-time requirements. In functions such as airbag control, a millisecond can be decisive. If the airbag is not triggered at the specified time, this can have severe consequences for the passenger's health. Therefore, programs have to have predictable behavior under all circumstances. To achieve this, everything (e. g., memory allocation, process assignment, typing) is done statically. This eases testing; nevertheless, testing real-time systems is a complex task. Also, using dynamic analysis techniques such as debugging or tracing is hardly possible due to technical reasons.

High reliability and safety requirements. While software faults are quite common in desktop software, system failures are not acceptable in the embedded world. Systems in an automobile must be reliable and safe in all situations. Software is a part of these systems and has a high impact on overall system behavior. This means that any software change – even a refactoring – must be succeeded by intensive system tests, which in turn is complex and costly, as the previous point highlighted.

Limited resources. The need for a larger memory chip in a controller means additional costs of millions of Euros when it is built into a large number of cars or devices. For this reason, storage has to be kept as small as possible. A similar argument is valid for computational power. These requirements are contrary to the trend in workstations, where nearly arbitrary amounts of memory and fast processors are available. They result in an optimized mapping of data to memory, optimized code, and limitation of stack usage.

Heterogeneity of domain knowledge. Experts from different core disciplines – such as electrical, control, and mechanical engineering – are involved when developing an automotive control device. A lot of their domain knowledge is integrated into the software, but is often not explicitly available. It would be

beneficial if it could be recovered from the code automatically. The high degree of connectivity between electronic control units makes design, integration and testing very complex [3].

Short development cycles under cost pressure. The high time and cost pressure in the automotive market encourages reuse of components and software. Product lines are the technology of choice for reuse in the large. They are intensively used in automotive software and lead to a high variability. An efficient and commonly used implementation mechanism for variability is the C preprocessor. Unfortunately, static code analysis has its problems with that [2].

3 Architecture and Implementation

These requirements lead to a design and implementation that also has its specialities. A number of recommendations and best-practices for automotive software is described in the MISRA guidelines [5, 6]. For example, MISRA-C forbids the use of recursion, dynamic memory allocation, or certain library functions such as those in `stdio.h` and `time.h`.

Programming paradigm and tools. For a long time, embedded software has been programmed in assembly language. This has mostly changed to C since the early 1990's. However, C was used in a rather assembly-like manner in the beginning. This has changed meanwhile, but there is still a lot of legacy code. Also, since the mid-1990's, visual modeling or programming tools[1] have been commonly used for creating control software (see Figure 1). To achieve high performance and a low memory footprint, a complex tool chain is then applied to the generated code. For reengineering, it could be beneficial to start on the level of the visual modeling tool instead of the generated code.

Time-triggered computation. Functionality that controls an automotive device is usually organized in different time slices. Actions are scheduled to be executed in a specified time raster. This also means that an action has to terminate within a certain period of time – otherwise, timing is disturbed, and other actions cannot be executed in time (cf. real-time requirements). The individual actions within these time slices are highly specialized and reflect the division of labor between the involved specialists from different domains.

Blackboard architecture. Communication, control and data flow between time slices and actions is usually realized through a blackboard architecture: values are written to and read from a shared memory area that is visible to all involved participants. Mechanisms to ensure data integrity between time slices are established. This architecture is sometimes carried to the extreme by having only global variables. Techniques for understanding blackboard-based systems could be very useful in the automotive domain.

[1] such as ASCET, `http://www.etas.de/`

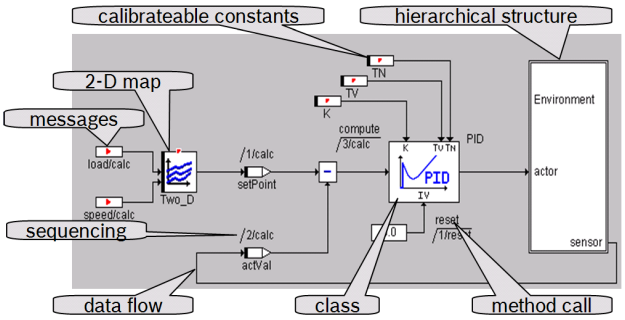

Figure 1: Sample ASCET model

Properties of control systems. Automotive software has an emphasis on data driven algorithms, parameter optimization, adaptive control/control loops, and on-board diagnostics [6]. It usually contains numerous characteristic maps and curves that represent physical dependencies and properties (Fig. 1). These properties require specialized analysis techniques.

4 Experiences and Future Directions

In the last years, reengineering was introduced to Robert Bosch software development. Within the existing programming paradigm, restructuring techniques were successfully applied and resulted in improved maintainability and reduced resource usage. However, this was largely done manually, which is too laborious as a developer's daily work. As the next steps, we will therefore investigate how "hot spots" as potential candidates for restructuring can be identified automatically and how understanding of typical automotive software can be supported by tools.

References

[1] M. Broy. Challenges in automotive software engineering. In *Proc. of 28th ICSE*, pages 33–42, 2006.

[2] J.-M. Favre. Preprocessors from an abstract point of view. In *Proc. of 12th ICSM*, pages 329–338, 1996.

[3] K. Grimm. Software technology in an automotive company. In *Proc. of 25th ICSE*, pages 498–505, 2003.

[4] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press, 1985.

[5] MISRA consortium. Guidelines for the use of the C language in vehicle based software, 1998.

[6] MISRA consortium. Road vehicles – development guidelines for vehicle based software, 2000. ISO 15497.

[7] D. L. Parnas. Software aging. In *Proc. of 16th ICSE*, pages 279–287, 1994.

[8] A. Pretschner, M. Broy, I. H. Kruger, and T. Stauner. Software engineering for automotive systems: A roadmap. In *Proc. of 29th ICSE, Future of Software Engineering*, pages 55–71, 2007.

[9] V. Schulte-Coerne, A. Thums, and J. Quante. Challenges in reengineering automotive software. In *Proc. of 13th CSMR*, 2009.