15 Years of Reengineering Embedded Control Software at Bosch

Jochen Quante
Robert Bosch GmbH, Corporate Research
Renningen, Germany
Jochen.Quante@de.bosch.com

When the Workshop Software Reengineering celebrated it's 10th birthday in 2008, I joined Bosch Corporate Research to work in a team of Reengineering pioneers. In this paper, I report from the experiences and advances during the last 15 years regarding Reengineering activities within Bosch.

The Early Years

When the Bosch research department started the first Reengineering project around 2003, it was not yet clear whether it would be a success. The need for "Reengineering" implied that the software was not optimal, which was initially hard to accept by our organization. Also, other software properties such as code size and runtime were considered much more important than maintainability. Naturally, both are quite limited resources in embedded control software. Therefore, the first piloting activities were largely manual work to prove that Reengineering can also be applied to embedded realtime software without negative impact on code size or runtime. However, it quickly turned out that Reengineering can even improve the above-mentioned properties. For example, developers who used early versions of ASCET[1] (a model based tool for embedded software development) *had* to create clones in models because there was no support in the modelling software to reuse any code (i. e., no support for classes). Removing these clones using new features of the meanwhile available ASCET version lead to reduction of flash usage by up to 75%. Another problem in these early days was the lack of Reengineering tools that could be directly applied to embedded C code, or even ASCET or Simulink models. Also, a large share of the methods from academia was not applicable to our kind of code [12]. We therefore started to develop our own methods and model analysis tools.

Maintainability Index and Phase Model

One result from these activities was the adaptation of the well-known Maintainability Index [9]. We applied it to model-based development and calibrated it with the notion of understandability that our developers have (empirical assessment). The introduction of the Maintainability Index was quite a success: It is nowadays used for complexity assessment, prioritization of Reengineering activities, and tracking improvement of these activities. This is also the first stage of our Reengineering phase model [4], which is an extension of SEI's horseshoe model. The identification of overly complex modules is followed by program understanding and reconstruction of requirements. The third phase is construction of a new solution, and the fourth phase makes sure that the new implementation is the same as the old one.

Program Understanding

One of the most laborious activities in Reengineering is program understanding. Specially for software that has aged for many years, it is often a huge effort to reconstruct its requirements. Therefore, tool support is urgently required. We hoped for support from the research community and therefore started a public program understanding challenge[2], which was conducted as part of ICPC 2011. We published an embedded control C function along with a task description, which was to find, explain, and fix a bug that was exhibited in a number of test cases. The participants mostly focused on the bug finding aspect, which meant application of bug localization techniques – instead of really supporting program understanding. However, the most successful participant used abstract interpretation to understand the code and the bug [5]. This encouraged us to follow this direction also for our own analysis tool for models, which is meanwhile available [7].

In parallel to that, we investigated how different views on models can give better insights and support in certain program understanding tasks [6], since a model always only focuses on one particular aspect. The result was that there are only few generally helpful views. You often need specific views for specific questions. This means that a kind of query language is required to extract the relevant information from the program. Another result was that analysis results must be presented in a visualization that is close to the input artifact (e. g., ASCET model). A more recent and ongoing activity is the semi-automatic extraction of high-level models from code. We call this activity *Model Mining* [11]. First results for extracting state machines from code look very promising.

[1] https://www.etas.com/en/products/ascet-developer.php

[2] http://icpc2011.cs.usask.ca/conf_site/IndustrialTrack.html

Model Level Software Analysis

Efficient Reengineering also requires adequate tools. In the absence of such tools for model-based development with ASCET, we started developing our own tools. For example, clone detection on block diagrams was not available at all. Only two publications reported from applicable approaches for Simulink models. We adapted one of these approaches for ASCET [1]. Today, the tool is used to optimize flash usage and find Reengineering opportunities in models.

We furthermore developed an extensive framework for analysis of model-based control software [8]. It supports different kinds of input models and provides standard control and data dependency analyses, as well as an abstract interpreter [7]. The framework is the basis for many internally used tools and use cases, such as consistency checks, quality assurance, model metrics, and documentation generation, and it is used for prototyping new analysis ideas, such as concolic testing on models [3] or partial evaluation for program understanding [6].

Variance

Another important aspect of Bosch software is variance: We have many different customers, and each customer has a lot of different products [12]. Variance is usually implemented using the C preprocessor. This makes it hard or impossible to analyze. Therefore, we also looked into analysis of such variant C code [2]. Later, we extended this with a transformation to equivalent valid C code by replacing preprocessor construct by corresponding C constructs [10]. This allows application of standard analysis tools also to product line code. Furthermore, it allows testing of all variants with a single executable.

Reengineering in the Large

For quite some time, we performed Reengineering on individual modules that were considered overly complex. Only in the last few years, Reengineering was also applied to entire subsystems. This required new approaches and methodologies. For example, for certain subsystems, only stepwise Reengineering was possible. This meant Reengineering one module after the other towards an intended target architecture, while preserving functionality at all intermediate stages. Our experience was that the effort for adjusting the existing system to the reengineered modules cannot be underestimated. In other cases, a "big bang" Reengineering was possible – but had other disadvantages, such as the long lasting parallel development.

Furthermore, large Reengineering projects were set up and leveraged the techniques and experiences from our pioneering work. A recent project that reengineered hundreds of modules estimated that it will save about 25% of maintenance and calibration effort and thus pays off quickly. There also is a special internal Reengineering course that tries to give the condensed learnings from our Reengineering endeavors to other developers. Since 2010, more than 200 developers have attended that training, and with the increasing importance of software, the demand is continuously increasing.

The Future

It was a long way from the early days with fundamental doubts about the necessity and applicability of Reengineering to an established technique at Bosch. Our experience suggests that it pays off quickly in most cases. Management is meanwhile aware of the specific challenges of aging software – and what to do against it. Also, the automotive safety standard ISO 26262 enforces limited complexity in software and thus drives the demand for Reengineering. Our research projects set the stage for this development, and we continue to develop advanced methods, techniques, and tools for making Reengineering more efficient.

References

[1] F. Gerardi and J. Quante. Type 2 clone detection on ASCET models. *Softwaretechnik-Trends*, 32(2), 2012.

[2] R. Heumüller, J. Quante, and A. Thums. Parsing variant C code: An evaluation on automotive software. *Softwaretechnik-Trends*, 34(2), 2014.

[3] A. Hoffmann, J. Quante, and M. Woehrle. Experience report: White box test case generation for automotive embedded software. In *Proc. 9th Int'l Conf. on Software Testing, Verification and Validation (Workshops)*, pages 269–274, 2016.

[4] J. Quante. Reengineering automotive software at Bosch. *Softwaretechnik-Trends*, 31(2), 2011.

[5] J. Quante. When program comprehension met bug fixing. *Softwaretechnik-Trends*, 32(2), 2012.

[6] J. Quante. Views for efficient program understanding of automotive software. *Softwaretechnik-Trends*, 33(2), 2013.

[7] J. Quante. A program interpreter framework for arbitrary abstractions. In *Proc. 16th Int'l Working Conf. on Source Code Analysis and Manipulation*, pages 91–96, 2016.

[8] J. Quante. A software analysis framework for automotive embedded software. *Softwaretechnik-Trends*, 36(2), 2016.

[9] J. Quante, T. Grundler, and A. Thums. Maintainability index revisited: Adaption and evaluation for Bosch automotive software. In *Proc. 3. Workshop Software-Qualitätsmodellierung und -bewertung (SQMB 2010)*, pages 67–70, Feb. 2010.

[10] J. Quante, A. Thums, and O. Pikuleva. Transformation of preprocessor variance to post-build variance. *Softwaretechnik-Trends*, 37(2), 2017.

[11] W. Said and J. Quante. Towards interactive model mining from embedded software. *Softwaretechnik-Trends*, 37(2), 2017.

[12] V. Schulte-Coerne, A. Thums, and J. Quante. Automotive software: Characteristics and reengineering challenges. *Softwaretechnik-Trends*, 29(2), 2009.