Reengineering Embedded Automotive Software

Andreas Thums, Jochen Quante
Robert Bosch GmbH, Stuttgart, Germany
Corporate Sector Research and Advance Engineering Software
E-mail: {Andreas.Thums|Jochen.Quante}@de.bosch.com

Abstract

The fact that software ages holds for embedded automotive software as well as for any other kind of software. In comparison to IT software, the automotive domain has to deal with different kinds of requirements, such as real time properties, feedback control, and constrained resources. Therefore, used programming languages are C – to meet resource constraints – and data flow oriented graphical languages – to meet the used engineering method and notation of feedback control engineers. This makes the software quite different from what the software maintenance and reengineering community is usually working on, and their results are seldom directly applicable.

In this paper, we describe results of a Bosch-internal research project that focused on the adaption of existing reengineering techniques and methods to embedded automotive software development. The goal was to make software maintenance more efficient by a) preventing software ageing and b) supporting program comprehension. Our approach was to make existing reengineering techniques usable for series development in an effective and efficient way. The result is a set of reengineering tools and practices that are specialized for the needs of the automotive domain and usable in practice.

Keywords: Embedded software, reengineering, maintainability, program comprehension, industrial application.

I. Introduction

It is a well-known fact that software ages [1]. Embedded automotive development is based on software that has been developed decades ago and evolved over the years, too. This software has a big value in form of implemented knowledge and experience. However, requirements change over time, functionally as well as technologically. The consequence is that the original software structure is no longer adequate. This observation has already been reported by Lehman and Belady in the 1980’s [2].

Besides these facts, which are common with IT software development, there are some special characteristics in the development of embedded automotive software [3]:

C1 Real-time software has to fulfill very tight timing constraints. For example, an airbag has to be fired within milliseconds after collision detection to be effective. This has a number of further consequences.

C2 The software is either implemented using a restricted subset of C (such as MISRA C [4]), or model based with tools like ASCET\(^1\) or Matlab/Simulink\(^2\). The latter two provide data flow oriented modeling languages, from which efficient C code can be generated.

C3 The architecture is based on a black board and time slices. This means that communication between functions is mostly done through global variables (instead of function calls), and all functions are called cyclically in defined intervals.

C4 The existing code is hardly structured with functions or classes, but has flat call structures. This is done to maximize execution speed.

C5 The code cannot be instrumented for dynamic analysis, because this would break the hard real-time constraints (multimedia devices are excluded here).

C6 Everything is done statically: Dynamic memory allocation is not used to prevent runtime errors. Also, dynamic binding is not used. This eliminates quite a number of problems of classical static analyses and makes alias analysis much easier.

C7 Program understanding is mostly necessary on function level. Small and detailed physical concepts are hard to understand in their software representation.

C8 The software is usually developed using a product line approach. It has a lot of variability with variance determined both at compile time and at calibration time. Calibration means setting certain parameters, such as curves, maps, timeouts, etc., for a certain variant. This is done after the software is finished.

\(^1\)http://www.etas.com/de/products/ascet_software_products.php
\(^2\)http://www.mathworks.de/products/simulink/
i.e., when the system is already running, e.g., in a car.

There are 10 times more calibration engineers than function developers. For calibration, these engineers must understand the dependencies within the software: Which parameter do I have to change to attain a certain effect? They look at the models for that, which means that these models are far more often read than changed.

The results that we present in this paper were developed within a project that had the goal to make software maintenance more efficient by adapting known reengineering methods and techniques to the automotive domain with regard to these specialties.

The structure of the paper is as follows. In Chapter II, the data flow oriented modeling language ASCET is introduced, which is widely used for implementing embedded automotive software. Chapter III introduces the reengineering phase model that was developed within our project with a focus on identification and understanding of complex automotive software. In Chapter IV, our experiences with reengineering in a large organization are discussed, followed by a presentation of future directions and important research topics in reengineering for the automotive domain in Chapter V. We summarize the paper in Chapter VI.

II. Data Flow Oriented Programming with ASCET

ASCET is a graphical modeling language that is suited to the needs of control engineers (C2). Therefore, ASCET supports block diagrams, state machines, and a textual language called ESDL\(^3\). In the context of this paper, we focus on block diagrams, which are ASCET’s main concept. A more comprehensive description of all concepts provided by ASCET can be found elsewhere [5].

A block diagram consists of elements – representing variables or operators – and solid lines that connect these elements – representing data flow. Figure 1 shows a simple example. When \(c\) is read, its value is implicitly computed as \(a + b\). The data flows from left to right and is modified by operators.

This notion of data flow is very intuitive for simulation and prototyping. Nevertheless, for reliable implementation of embedded automotive software, sometimes the explicit annotation of execution order is necessary. ASCET uses sequence numbers to specify the order of execution for each method.

Figure 2 shows a more complex example with sequence numbers, conditional control flow, and grouping in hierarchies. All statements (assignments, if clauses, etc.) are assigned to a method, here `compute`, and numbers specify the sequence of operations within this method. At first, the value of `result` (i.e., the result of the last call of this method) is stored in `mem` (\(/1/compute\)). Then, the condition of the `if-then-else` operator is evaluated (\(/2/compute\)). The `if-then-else` operator works as follows: The left input port provides the value of the boolean condition (boolean information is depicted through a dashed line). The right output port leads to those statements that should next be executed in the `true` case. In the `false` case, the bottom port is used. These dash-dot lines depict control flow. In the example, if `input == mem`, then `input` is assigned to `result`. Otherwise, the result of a computation which is done in the hierarchy `Average` is assigned to `result`. In summary, the diagram in Figure 2 corresponds to the following C function:

```c
void compute() {
    mem = result;
    if (input == mem)
        result = input;
    else
        result = Average(input, mem);
}
```

A hierarchy in ASCET is a purely graphical grouping of elements, connections, and sequencing. A hierarchy provides no kind of encapsulation: Every element of the whole block diagram can be referenced, not only those connected to the hierarchy. Also, sequence numbers need not be continuous in a hierarchy: The control flow may repeatedly jump into and out of the hierarchy, also through arbitrary levels of nesting.

Another specialty of ASCET is that there can be multiple occurrences of the same variable. In the example, there

\(^3\)Embedded Software Description Language
are multiple occurrences of both \textit{input} and \textit{result}. The latter is even written to in different places. In combination with conditional control flow, the real underlying data flow can therefore become very complex and is no longer obvious from the data flow model. The model is suggestive of providing a complete view of data flows, although the visual model may in fact \textit{not} be complete.

These specialties of ASCET (sequencing, hierarchies, multiple occurrences) impose severe problems for understanding and maintenance. They are addressed in Section III-C by different views.

On the other hand, ASCET fulfills the specific requirements for embedded automotive software development in multiple ways:

- It provides a large library of domain specific blocks. Among others, there are \textit{filters}, \textit{PID elements}, ramps, and \textit{integrators}. A specification based on the described elements can be simulated within the tool for a very fast validation step.
- ASCET generates efficient C code. This code can easily be integrated with other handwritten C code, which is for example necessary for hardware drivers. The code generator uses only a restricted subset of C (C2).
- For the efficient usage of memory and CPU, it allows to specify an integer representation for the values of continuous variables (C1). This task is separated from modelling.
- The support for black board communication and cyclic tasks is directly built into ASCET (C3).
- It supports variance points within the model: When a variable is declared as a \textit{system constant} and connected to a conditional, the code generator produces variant C code (using \#if preprocessor directives, C8).
- A variable can be declared as a \textit{parameter} which can be modified \textit{(calibrated)} after compilation to adjust the behavior to specific needs of the software, e.g., specific characteristics of an engine (C9). ASCET also supports \textit{curves} and \textit{maps} as collections of parameters that belong together, e.g., to specify specific acceleration curves.
- The diagrams provide a look that electrical engineers and control engineers are used to. This reduces the conceptual distance to traditional program code for them.
- Finally, it supports classes to allow for reuse, but only allows static instantiation and does not know inheritance (C6).

\begin{figure}[h]
\centering
\includegraphics[width=\columnwidth]{figure3.png}
\caption{Phase Model for Reengineering.}
\end{figure}

\section{Reengineering Applied to Embedded Automotive Software}

Embedded automotive software is usually built based on a product line approach (C8). A specific customer variant is derived from a product line platform, but it is also based on the preceding project of the same customer. Usually, the customer does not want to get rid of any functionality. Therefore, the product line has a long lifetime and is always growing.

To mitigate the risks of a long-living product line platform, we do not perform one “big bang” reengineering, but prefer smaller incremental ones. The advantage of this approach is that only small parts of the software are changed, and new technologies and methods can be evaluated in the small. Also, a follow-up reengineering can build on the experiences from previous activities.

To describe the incremental reengineering process and to correlate the different activities, we defined the \textit{phase model for reengineering of embedded automotive software} that is shown in Figure 3. It is similar to SEI’s “horseshoe model” [6], but we added a preceding identification phase and a succeeding validation phase. The horseshoe model describes the three levels \textit{source}, \textit{function}, and \textit{architecture level} on which transformations or reengineerings are performed. In our setting, the focus of reengineering is on source and function level, due to the requirement that we have to do reengineering in the small (C7).

First of all, the activities are based on those artifacts that the function developer is working on. In our case, these are usually C code modules and ASCET models. We have to identify the artifacts with the highest benefit for reengineering. The most complex ones are good candidates for that. Therefore, we developed indicators for detecting complex parts (see Section III-A). After identifying those, the reasons for their complexity have to be understood. We also need to provide solutions how to resolve this complexity. Furthermore, the functionality has to be understood in
detail before a component with equal behavior (and reduced complexity) can be constructed. This activity is very time consuming — it accounts for about 40% of the entire software lifecycle costs \cite{7}. To make this activity more efficient, we focused on using different views to support program understanding. In particular, we developed views for ASCET and C code (see Section III-C). Of course, these views can also be used to improve “normal” software maintenance.

To construct a new solution, design principles and patterns guide the development, which leads to comprehensible structures in the software. This eases both construction and understanding of the new solution. The final step in the phase model is to verify that the newly constructed solution fulfills the requirements, which means that it is functionally equal to the original solution.

Besides developing methods for the topics mentioned above, another main focus was to evaluate the techniques and to establish reengineering in the business units. The following sections describe our approaches and results for the first two phases in more detail.

A. Bosch Maintainability Index

The first step in our reengineering phase model is to identify those functions or modules with the worst maintainability. Therefore, we need to somehow assess maintainability of all the functions in a system.

As we did not want to define what makes up the complexity of automotive software (using metrics thresholds), we followed the “maintainability index” (MI) approach by Oman and Hagemeister \cite{8} to derive a complexity index from expert assessment.

First of all, we tried to identify which metrics are relevant for our domain. As remarked in characteristic C2, we have to deal with C code and ASCET models. Because we wanted to have a common basis for both, we decided to rely on pure code metrics and not to incorporate visual metrics (like numbers of crossing lines) into the indicator. On the other hand, the necessity of regarding product line code with high variance \cite{8} introduced metrics like degree of variance (both compile time and calibration time variance). We chose as a starting set for metrics the one measured by QA-C\cite{4}, enriched with domain specific metrics like the ones mentioned above.

As subject systems, we used different gasoline and diesel engine control unit (ECU) software projects and applied principal component analysis. This technique basically reduces this initial set of possibly correlated metrics to a smaller set of linearly uncorrelated metrics clusters (based on a number of observations). One can then reduce the set to one representative per cluster. In our case, the chosen representatives were: Lines of code (most metrics belong to this cluster), number of different function calls, number of local variables, maximal nesting level, and number of logical operators.

In parallel, we asked various function developers to assess the complexity of functions with the help of a questionnaire. The intent was to assess the complexity of the software solution and not of the inherent complexity given by its functionality. Only this accidental complexity can be reduced by reengineering, not the inherent complexity of the function. Unfortunately, it is not decidable whether the inherent or the accidental complexity was assessed. However, we set up the questionnaire in a way to give the developers a feeling for what makes up accidental complexity and thus guide them towards relevant aspects.

Based on this data, regression analysis was performed to determine the optimal weight of each metrics cluster representative with respect to the complexity assessment of the function developer. The overall result is a weighted sum of the five representative metrics. We call this formula the “Bosch Maintainability Index”.

Evaluation of the index on a different set of functions showed that both precision and recall are about 60%. Although this may not seem very high, the feedback from the business units was that this is good enough. They are happy to get some hint where to look first when aiming at maintainability improvement. Instead of further improving the index (for example, by including bad smell patterns, see Section III-B), we therefore focused on providing a tooling to make the index usable in practice. The tooling used QA-C\cite{5} as a metrics suite, and ConQAT\cite{6} for visualization of the results for C code. Since the QA-C tool was already there anyway, it was very easy and cheap to roll out this index. These are factors that should not be underestimated for industrial application. The index is meanwhile used and accepted in the business units to prioritize and justify their reengineering activities.

Apart from the practical relevance, applicability and acceptance aspects, we found a number of other noteworthy issues in our studies on which we will elaborate in the following (cf. \cite{9}).

Code size and maintainability.: The length of a function or module (LOC) alone is already a strong indicator for maintainability. However, we found that the LOC criterion alone also results in a precision of 50%, but in a recall of only 43%. This means that the additional terms in the index help to improve recall, and this improvement comes at quite low cost.

Comparability.: Another finding concerns different programming styles. Different organizational units may

\textsuperscript{4}Static code analysis tool from QA Systems, http://www.qasystems.de/qac

\textsuperscript{5}http://www.qa-systems.com/

\textsuperscript{6}http://www.conqat.org/
use different programming guidelines and implementation styles, which makes results hard to transfer from one system to another. This means that the index has to be calibrated for each unit individually. There cannot be a single universal or even company-wide maintainability index.

*Individual differences.* Also, different experts judged about the same module in quite inconsistent ways. For example, the same module was rated “good” by one expert and “very bad” by another. Of course, this is not a good basis for maintainability assessment: How can an automatism assess maintainability when even the experts do not agree? And how could an objective assessment look like?

*Outliers.* Finally, the index ignores outliers, which may contain valuable hints about maintainability problems in practice. In a case study we performed, an approach solely based on outliers delivered quite good results for assessing maintainability. If precision shall be improved, it could be a good idea to include outliers into the index.

### B. Bad Smell Patterns

One disadvantage of the maintainability index approach is that it does not allow root cause analysis, i.e., you know that something is bad about your code or model, but not how to improve that. Another approach to hot spot detection is the use of bad smell patterns. These are recurring patterns in the code that often indicate opportunities for refactoring [10]. Similar to the index, these smells only cover selected aspects of maintainability, but along with the indicated spot, they deliver concrete hints what should be improved. Such approaches are as well successfully applied for quality assessment of software in industry [11]. Bad smell patterns can be used for other purposes as well. For example, they can be used to teach developers good programming style.

Quite a number of patterns are described in the literature, but these are usually for object-oriented systems. Therefore, we adapted a subset of them to our development paradigm. During application of our reengineering methodology in the business units, we extended this catalog of bad smell patterns by recurring examples that we found in our code, in particular by negative patterns for model-based development.

The following are some of the categories that are covered by our bad smell catalog:

- Control structures that are unnecessarily complex, for example because the control logic is spread everywhere.
- Model aesthetics, in particular concerning the layout of graphs.

<table>
<thead>
<tr>
<th>Name</th>
<th>Name of the smell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>What is the smell?</td>
</tr>
<tr>
<td>Problem</td>
<td>Which problem is caused by the smell?</td>
</tr>
<tr>
<td>Solution</td>
<td>How can the smell be removed?</td>
</tr>
<tr>
<td>Indicators</td>
<td>What are the indicators for the presence of this smell?</td>
</tr>
<tr>
<td>Source</td>
<td>Where was the smell found? (i.e., in a project or in literature)</td>
</tr>
<tr>
<td>Evaluation</td>
<td>How often does the smell occur in Bosch software?</td>
</tr>
</tbody>
</table>

**Table I. Bad smell pattern form.**

- State machine usage, where a lot can go wrong. For example, state machines may grow too large which makes them unusable. Or they lack abstractions, e.g., when conditions are called “condition1”, “condition2”, etc.
- Clones, which can occur in textual programming as well as in model-based development.
- Variance issues, e.g., distributed variance points.

Each smell in the catalog is documented in a form as shown in Table I, along with a number of examples. One concrete instance of such a smell pattern is shown in Table II. The bad smell pattern catalog is now used as a checklist for maintainability assessment and for training.

### C. Program Comprehension with Views

In a first attempt to better support program comprehension, we asked the community to demonstrate their tools on a problem of our domain [12]. Unfortunately, the results were not convincing [13], so we had to come up with our own solutions which are introduced in the following.

In architecture descriptions, *views* [14] are used to focus on different concerns of the same system. The same concept can be used on implementation level. A given piece of software or model can only express a certain amount of aspects. Almost always, there are other aspects that stay hidden and are therefore hard to comprehend. If we change the software to emphasize these hidden aspects, others will be neglected. Views are a means to solve this problem.

This is also true for model based software development. A model only focuses on some aspects. For example, ASCET focuses on data flow and neglects control flow. This makes it very hard to understand the control flow of an ASCET model, although it is still important. The same is true for data flow in a C function.

When thinking about potential views, we have to consider the specialties of embedded automotive software that were mentioned in the introduction. A first restriction is the programming language: We deal with C code and ASCET models (C2). However, the results for ASCET can easily be transferred to other data flow based languages like Matlab/Simulink. A further restriction is that dynamic
A function behaves differently in different states, but the state information is distributed over several variables or even bits, and the actions and conditions that belong to states and transitions are distributed throughout the code. The state is explicitly checked at many different locations.

Such an implicit state machine can be extremely hard to comprehend. Things that are only done in a certain state may be distributed everywhere, and the conditions that make up the state are unclear. Also, transitions and associated actions are hard to identify. Often, this bad smell occurs together with other smells, such as long method, missing separation of concerns, or complex conditions and data flow.

Make the state machine explicit and introduce abstractions for the actions and conditions. Also let the state machine trigger all actions.

So far, this smell can only be identified manually, by intensive analysis of the function. Good indicators are the presence of flip flops (states) and edge rising/falling detectors (events causing transitions).

Table II. Concrete Bad Smell Pattern: Implicit State Machine.

<table>
<thead>
<tr>
<th>Name</th>
<th>Implicit State Machine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>A function behaves differently in different states, but the state information is distributed over several variables or even bits, and the actions and conditions that belong to states and transitions are distributed throughout the code. The state is explicitly checked at many different locations.</td>
</tr>
<tr>
<td>Problem</td>
<td>Such an implicit state machine can be extremely hard to comprehend. Things that are only done in a certain state may be distributed everywhere, and the conditions that make up the state are unclear. Also, transitions and associated actions are hard to identify. Often, this bad smell occurs together with other smells, such as long method, missing separation of concerns, or complex conditions and data flow.</td>
</tr>
<tr>
<td>Solution</td>
<td>Make the state machine explicit and introduce abstractions for the actions and conditions. Also let the state machine trigger all actions.</td>
</tr>
<tr>
<td>Indicators</td>
<td>So far, this smell can only be identified manually, by intensive analysis of the function. Good indicators are the presence of flip flops (states) and edge rising/falling detectors (events causing transitions).</td>
</tr>
</tbody>
</table>

Context aware views to simplify computations use calibration data to simplify code. For example, if a parameter is calibrated to zero, a following addition can be omitted, or a multiplication can be substituted with zero.

Control flow views: help to understand the sequence of execution, in particular in data flow oriented representations such as ASCET. Examples are control flow graphs, Nassi Shneiderman diagrams, and the ASCET sequence view (see below).

The complete list consists of about 20 views. Based on this list and corresponding examples, we questioned function developers, team leaders, and calibration engineers about their potential usefulness. Based on the results of this survey, we prioritized the view candidates and realized the two views with the highest expected benefit with respect to their specialties in the automotive domain. They are introduced in the following.

Concrete View: Signal Flow

To comprehend the physical interrelationships between values at certain points in the software, the data flow between these points is very important. Therefore, a view that shows these dependencies is of great interest, especially for calibration engineers (C9). Calibration engineers have to set values for engine specific parameters in the software to derive a specific variant from a product line software (C3). Therefore, they have to understand the physical effect chain which is represented in the software (C7). For example, they want to know which of the parameters affect which output variable.

A technique that is intended for calculating the flows through a function is slicing [15], [16]. For a given variable at a given program point, slicing can find out which other parts of the program are affected by that variable, or which parts of the program influence that variable. It can also determine the dependencies between two program points, which is called chopping [17]. These techniques answer exactly those questions that application engineers have when working with our software models. Therefore, our
first view is a realization of slicing and chopping for ASCET.

In Figure 4, the result of this analysis on an ASCET model is presented. This example shows one specialty of ASCET. Because one can have multiple occurrences of variables in one picture, data flow analysis has to derive dependencies that were not present as data connections in the original picture. This means the data flow model may be incomplete prior to data flow analysis.

To realize this view, we needed to derive the complete data dependency graph for ASCET models. The internally used program dependency graph is the same for ASCET and C code in our framework. In the construction of the program dependency graph, the control flow and multiple occurrences have to be considered. Based on this dependency graph, the computation of the signal flow is reduced to the computation of a forward slice for the input variable and a backward slice for the output variable and then taking the intersection of these two slices (chopping). Just recently, the same approach has successfully been used for understanding signal flows through a component-based embedded system [18]. Another recent publication [19] shows how slicing can be applied to MATLAB Simulink models. Along with the positive feedback from our engineers, this is another indicator that this technique is well suited for easier understanding of models of embedded software.

Concrete View: Sequencing

The second view enriches the data flow oriented representation of ASCET with a control flow centric one. As ASCET focuses on data flow, the control flow is hard to follow. In particular, it is realized with sequence numbers. To understand the sequence of events, the developer must identify the sequence of successive sequence numbers. However, these may be distributed over several diagrams, and they are allowed to have gaps – which greatly impedes program understanding.

With the sequence view, it is easy to follow the sequence of numbers through different diagrams. Figure 5 shows a sample sequence view for an ASCET model. Below the ASCET model, you see a “time line” of the sequence numbers. A branch in the control flow is depicted as a branch in the time line. The time line basically corresponds to the control flow graph. Below the time line, each colored box represents an ASCET hierarchy (grouping). The dots mark the occurrence of the sequencing in the corresponding hierarchy.

You can easily see that there is a gap in the sequencing between the sequence numbers 11 and 13, and that there are consecutive sequence numbers that break the hierarchical grouping. This view makes the implicit control flow of ASCET explicit, and the “time line” is a representation of the corresponding control flow graph.

This view addresses special shortcomings of ASCET (gaps in sequences allowed, manual sequencing, sequence numbers distributed through hierarchies). However, it also addresses a general problem of data-flow oriented models: That the sequence of operations is hard to see. The view is an example of **understanding in the small** (C7): It helps to understand the details of a software solutions.
Remarks on Views

During the evaluation of these views, we experienced one practical issue that may limit their acceptance in practice if not considered: The different views and the ASCET model have to be interconnected. Ideally, the results of the analysis should be shown in the original model. This is very important, because the original code or model is always the reference and also a well-known orientation. For example, a click on a sequence number in the sequence viewer causes the view to jump to the corresponding representation in the ASCET model, and vice versa. Also, the signal flow viewer highlights the signal flows in the original model, and the sequence viewer is connected to it. We realized the views and this interaction concept in a tool prototype. This prototype is so convincing that currently a tool based on the concepts of this prototype is created for series production code.

While working on this topic, we realized that most use cases or questions to code or models need specialized analyses and views. There are only few views that help in many use cases. Therefore, we plan to extend our ASCET/C analysis framework such that it does not only support the above mentioned views, but additionally allows individual queries and visualizations. This way, the developers can create those individual views that are best suited for their current concerns.

In summary, the main results in program comprehension with views are the analysis framework for C code and ASCET analysis along with a number of views, and the initiation of the development of a tool that provides interactive navigation and viewing of software documentation (ASCET models along with their textual description). The presented views are used as part of this.

IV. Experiences

In this section, we will discuss our experiences with this project, with reengineering applied in practice, and with our concrete maintainability index and views.

Practicability. The project was located in Bosch’s central research department, while function development is done in the business units. Therefore, an essential requirement for success was to develop methods and tools that are really usable by the business units. These requirements should also hold for academic research, to some extent.

From a researcher’s point of view, an important observation is that for practitioners, the ease of use and applicability in practice is much more important than perfectionism and pure methodology. We will discuss two examples for that.

Reengineering obstacles. As a first example, for the business units, it is not possible to do reengineering as a separate activity, as advocated by theory [10]. About 50% of all maintenance effort accounts for testing [7]. The business units cannot spend this effort without functional improvements, because only those can be sold to a customer. Another reason is that after implementation, the code is still an artifact that is worked on. The calibration engineer uses the code to see the impacts of their work, i.e., how the adjustment of a calibration value influences the functionality of a component or (sub-)system. As described in characteristic C9, ten times more calibration engineers have to work with the code than function developers, who write or reengineer the code. And for calibration engineers, it is much easier to work with familiar code than with restructured code – even when the latter has much better maintainability.

Practitioners’ needs. A second example against the necessity of perfectionism is the maintainability index. We presented the first results of the Bosch Maintainability index to the business units with some additional ideas to improve precision and recall. One idea was not only to use the precompiled software – as needed by QA-C –, but the source code with additional information on variance, or to use change or failure data from repositories. But for the business units, indication with the quality of the existing index was “good enough”, and they refocused our work to provide an easy-to-use environment for computing the index. This also implies that the applicability of tools and methods needs a good insight into the working environment of the users. Therefore, it was a prerequisite for the project to work on series code, do reengineering on that, and to see how function developers work. This insight showed the challenges and needs of function development and directs future work.

Maintainability index. The experience with our maintainability index is that it can easily be used to measure the complexity evolution of software components and prioritize reengineering efforts for a whole software system. It is accepted both by management (as an objective criterion) and developers (as a means to better communicate reengineering needs). Another experience is that the index heavily depends on the engineering style, i.e., it is different for other domains and has to be calibrated individually.

Views. In the program comprehension topic, we learned that there are only few views that are useful in general, as the two mentioned in Section III-C. We also learned that it is very important to provide the references from the views to the original code or model. Apart from that, there is a multitude of questions that can be answered more efficiently if the right individual view is at hand. These individual views can be provided based on a software analysis framework as described above.

Transfer of results. Because the project was not located in the business units, the results of the project had to be
delivered to them. One experience in transfer is that tools can be transferred more easily than methods, because tools can be used, but methods have to be applied. Nevertheless, only providing a tool is not sufficient for its usage. Also, a prototype tool is necessary to develop the approach and to show its benefit. However, it cannot be used for developing series product code. The tool has to be maintained, and this can hardly be done by research organizations. Therefore, a maintenance strategy has to be provided which adapts the tool to changing requirements in the business units.

To establish reengineering know-how and methodology, a training course was developed. This training teaches the required knowledge about reengineering. But again, trainings are not sufficient for the application of the methodology. Trainings can sensitize the trainees for the corresponding topic, but an additional coaching for concrete cases is usually necessary.

To get the ability for doing reengineering on its own, we provided training on the job for the function developers. Reengineering experts did dozens of reengineerings together with different function developers to transfer reengineering knowledge. The feedback for that was very positive. In the end, the function developer has three results:

1) Confidence that reengineering is beneficial in his context,
2) the ability to do reengineering on his own, and
3) a reengineered function with improved maintainability, which in turn saves maintenance effort in the future.

Especially the first point must not be underrated. It is not really possible to prove the benefit of reengineering, e.g., through a controlled experiment. The effort for that is much too high: A development organization would never spend the money to develop two systems – which are large enough for proving the benefit of reengineering – in parallel. Therefore, you have to convince the organization, and the easiest possibility is to provide successfully reengineered functions.

V. Future Work

In future work, our focus will be on two topics: First, the integration of analysis capabilities for different languages, and second, combining benefits of textual and graphical modeling languages.

For the views presented in Section III-C, we developed a common basis for C code and ASCET models. Matlab/Simulink models will be integrated in this analysis framework as well. This enables a seamless analysis of the whole application software of embedded automotive control units. Great benefit is expected in supporting the calibration task, because the calibration engineer has to understand dependencies in a software created by someone else.

Currently, there is a great difference in the level of IDE support for developers using modern textual languages like Java and those using model based languages like ASCET or Matlab/Simulink. A typical IDE for Java has features like text completion, advanced searching, metrics computation, refactoring, and with additional tools also clone detection, architecture reconstruction, static bug finding, etc. These features are hardly supported by model based languages, but become more and more important as the complexity of models increases.

Therefore, another research direction is to link textual and model based languages. The advantage of textual languages is that they are well supported by IDEs (see above). This means that code can be adopted and refactored more easily. Nevertheless, models can be read and understood more easily, especially by people who are not software engineers. For them, a graphical representation may be more intuitive. Therefore, textual languages should be used for creating the code, and a corresponding graphical representation should be generated. We made a very promising study in visualizing (restricted) C code in an ASCET-like manner. This visualization can be used for documentation and calibration of the code.

Furthermore, another study develops a prototype for clone detection in ASCET models. This work is based on an existing approach for clone detection on Matlab/Simulink models [20]. Again, the goal is to make the modeling language and environment as powerful as textual languages with respect to creating, maintaining, and analyzing models. This is inevitable, because models become arbitrarily complex as well.

VI. Summary

The main contribution of this paper is to point out the special characteristics and technologies used in the development and maintenance of embedded automotive software and how they can be handled with respect to reengineering and maintenance activities in an effective and efficient way. The result are probably as well applicable to other domains with real-time or embedded software development.

We first presented the reengineering phase model for Bosch automotive software development. We then focused on two important activities in handling and maintaining long living code: Identification of hot spots, and program understanding based on views.

As far as possible, we based our work on results from the research community and adopted them to fulfill the requirements of the embedded automotive domain. The work showed that the basic principles of the reengineering
approaches can be transferred to the embedded automotive domain.

Unfortunately, neither the domain nor the used technologies are regarded much in the research community. This is probably due to the fact that publicly available examples are rare. Also, the languages C and ASCET are not attractive for the community, because C is hard to analyze, and ASCET is hardly known.

We hope to have shown that the embedded/real-time/automotive domain is a wide and interesting area for research: Reengineering methods and tools are still very rare there.

References


