Experience on Applying Software Architecture Recovery on Automotive Embedded Systems

Xinhai Zhang[1], Magnus Persson[1], Mattias Nyberg[2,1], Behrooz Mokhtari[2,1], Anton Einarson[2], Henrik Linder[2,3], Jonas Westman[1,2], DeJiu Chen[1], Martin Törngren[1]

[1] Kungliga Tekniska Högskolan
Stockholm, Sweden
{xinhai, magnper}@kth.se

[2] Scania CV AB
Södertälje, Sweden
mattias.nyberg@scania.com

[3] HiQ AB
Stockholm, Sweden

Abstract— The importance and potential advantages with a comprehensive product architecture description are well described in the literature. However, developing such a description takes additional resources, and it is difficult to maintain consistency with evolving implementations. This paper presents an approach and industrial experience which is based on architecture recovery from source code at truck manufacturer Scania CV AB. The extracted representation of the architecture is presented in several views and verified on CAN signal level. Lessons learned are discussed.

Keywords—architecture recovery; distributed embedded systems; automotive industry; software engineering

I. INTRODUCTION

In software engineering, legacy code, which is previously developed and successful software components, systems or platforms, evolves over time. During this evolution, the architecture of the legacy code tends to "drift", which means that relevant documentation gradually becomes incomplete or inconsistent as the code changes[1].

The same problem is also suffered in the automotive industry, which has become increasingly software-intensive during the past 30 years [2]. The accelerated system evolution accelerates the architectural drift. At the same time, the introduction of ISO 26262 [3] implies an increasing emphasis on safety requirements in the automotive industry. In order to guarantee safety in terms of software, a comprehensive safety and impact analysis of new functions is needed. This is a challenge with drifted legacy code, which makes it hard to achieve an overall understanding of the whole system.

Architectural drift happens not only because keeping documentations synchronized with implemented changes is not of the highest priority of developers, or that it is error-prone and time consuming, but also because there is no proper tool to support or verify the synchronization [1]. Therefore, principles on solving the drift should not merely be addressed through allocating more effort on documenting changes, but also the tools bridging the gap between design and implementation. One feasible approach to bridge the gap is software architecture recovery [4]. Therefore, we have implemented a domain-specific toolset in an automotive company to recover the software architecture of a truck from its implementation and to check the consistency between implementation and design in CAN communication level.

In the rest of the paper, we describe as follows: A brief state of practice survey is made on how software architecture recovery is used on distributed embedded systems. (Section II). We present a case study describing how we recovered software architecture from Scania Electronic Control Unit (ECU) source files (Section III). The extracted representation of the architecture is presented in several views and verified on CAN signal level (Section IV). We also discuss the lessons learnt from this case study as well as unsolved challenges (Section V). We finally conclude and suggest future work. (Section VI).

II. BACKGROUND

Software architecture recovery (similar works also refer to the method as architecture reconstruction [5] or reverse architecting [6]) is a process to extract high level architectural models with a specific level of abstraction from available artifacts of the system implementation [4]. Results from architecture recovery can be used for system understanding, consistency checking [7], impact analysis [8], and other processes related to verification, maintenance and design. Most published works regarding architecture recovery cover general purpose software. After the proposal of the reflexion model [1], combining architecture recovery and consistency checking has become a common approach to detect architecture drift [7].

The concept of recovering software architecture for embedded systems was proposed nearly 15 years ago by the project ARES [9]. However, at that time, the complexity of embedded systems was much lower than it is now, and knowledge related to software architecture was also deficient. Few later works of architecture recovery on embedded systems can be found, especially for the automotive industry. Mendonca and Kramer [10] developed an approach for the recovery of distributed software architectures, however, real-time features were not taken into account. Baloh, Raghav, and Sivashankar [11] provided a method to extract a model-based executable specification from legacy embedded control software in Simulink. However, their work focused on the execution model inside a single module rather than the interactions between modules or system distribution.

III. EMPIRICAL STUDY

Our work is a part of ESPRESSO project, which is collaboration between KTH and Scania CV AB. The project aims at providing an efficient development methodology to achieve functional safety regarding ISO 26262. The work presented here is the final outcome of a longer process and was preceded by initial separate proof-of-concept prototypes [12], [13]. We reorganized, improved and consolidated these

The work presented in this paper was funded by the Swedish national Vinnova project ESPRESSO.

proofed concepts and implemented them into a consistent toolset.

A. *Context of the Study*

On a Scania truck, most of the ECUs are distributed on three main CAN [14] buses: red, yellow and green bus. Fig. 1 provides an example of this network topology. Some of the ECUs are allocated on sub buses of their parent ECUs. The communication on the CAN buses is based on the standard J1939 protocol [15]. Of the ECUs, three were the focus of the work presented in this paper: the coordinator (COO), the engine management system (EMS), and the gearbox management system (GMS). They were chosen both since they are key ECUs and have been developed separately in different parts of Scania, hence having potential mismatch.

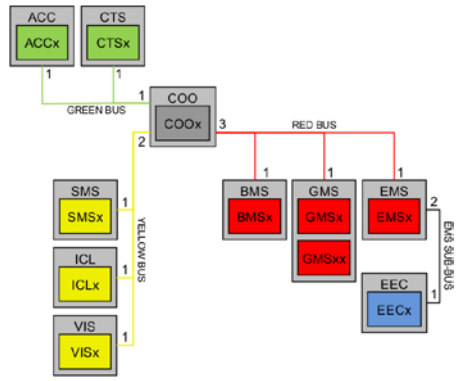

Fig. 1. Example of network topology in vehicle from Scania

AUTOSAR [16] is not used in Scania. Instead, they use their own software platform. Parts of the architectural principles related to our work are abstracted from the software architecture used in Scania and illustrated in Fig. 2.

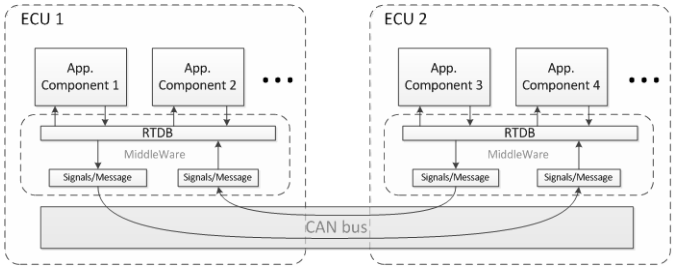

Fig. 2. Graphical illustration of software architecture in Scania. This architecture is abstracted in a way that is particular enough for the description in this paper. The one in actual use is slightly different.

An **Application Component** encapsulates part of one function. Each application component contains and only contains one real-time task which can be ether periodic or event triggered. Normally, an application component is managed as a .c file associated with related header file and calibration file. All these files are named as the application component acronym followed by the extension (.c, .h, etc). In some ECUs, application components are organized in different **Layers** and **Managers** which are represented only by the folder structure.

The **Real-Time DataBase (RTDB)** is a repository, on the running ECU, of all the information that is shared between Application Components. The RTDB at Scania can be seen as the equivalent to the RTE in AUTOSAR. It is used to handle the communication between both application components and ECUs. Variables stored in RTDB are called **RTDB variables**, which in implementation are micro-defined pointers pointing to a pre-allocated memory block. Application components can read from and write to RTDB variables via interfaces provided by RTDB. As restricted by the architecture, the RTDB is the only allowed way for application components to interact.

A **Signal** refers to a pointer pointing to one RTDB variable which is to be sent to or received from another ECU over CAN bus. RTDB variables are updated over time by application components, data read from hardware and signals received from CAN buses.

A **Message**, which is a collection of signals together with a header, is the unit for CAN communication according to the J1939 standard.

B. *Approach and Tool Chain*

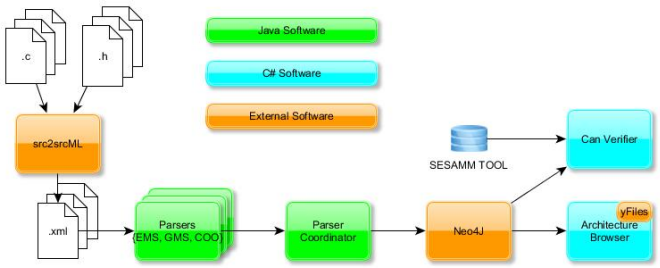

Fig. 3. Overview of the developed tool chain

The developed tool-chain, as illustrated in Fig. 3, largely consists of four main parts: A back-end with parsers and a parser coordinator, a standard Neo4J [17] database, and two front-end applications: *Architecture Browser* and *CAN Verifier*.

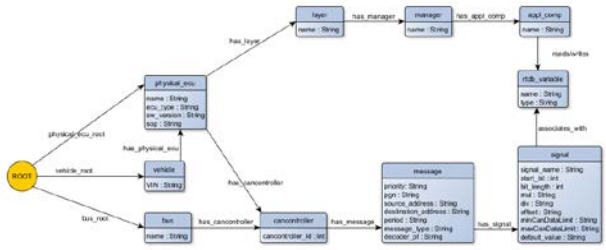

Fig. 4. Meta-model of information that need to be extracted from source code

1) *Parser and Coodinator*

The main work of the parser and coordinator is to retrieve architectural information from the source files (source code and calibration files) and store it into the database. Fig. 4 illustrates the meta-model used for all the retrieved information. Both the parser and the coordinator are built in Java.

To directly parse the c files is difficult and error-prone. Therefore, we use srcML toolkit [18] to transform all the source files into XML files in which c code is wrapped with information from the Abstract Syntax Tree (AST). Those XML files together with their positions in the folder structure form the inputs of the parser. The parser only focuses on source files

in the application layer and CAN communication layer. Information related to managers and application components can be retrieved from the folder structure of the source files in each ECU. Other information illustrated in the meta-model is extracted from the generated XML files using xPath [19].

The parsers locate architectural information in the source files by looking for specific patterns. For example RTDB interfaces are used to locate interactions between application components and RTDB variables. Specified data structures according to J1939 standard are used to parse associations of RTDB variables to signals, as well as the affiliations of signals to messages. The retrieved information is stored in temporary data structures before the coordinator uploads them to the graph database.

2) Database

The recovered information is stored in a more convenient way using a standard graph database, Neo4J. The choice of a graph database was motivated by the fact that its structure is explicitly built to support the kind of model data that is expected to be generated from architecture recovery, making queries and development easier to perform.

3) Architecture Browser

The first front-end tool in the toolset is the Architecture Browser. It is a purpose-built tool implemented to interactively visualize the implemented software architecture as-is. The development environment is C#/.NET, with yFiles WPF [20] being one of the main components. It currently can present two different views:

- *Network diagram*, presenting the ECUs and the main networks only.
- *SW/HW view*, additionally showing the internal structure of each ECU including RTDB variables and communication at a CAN signal level between ECUs.

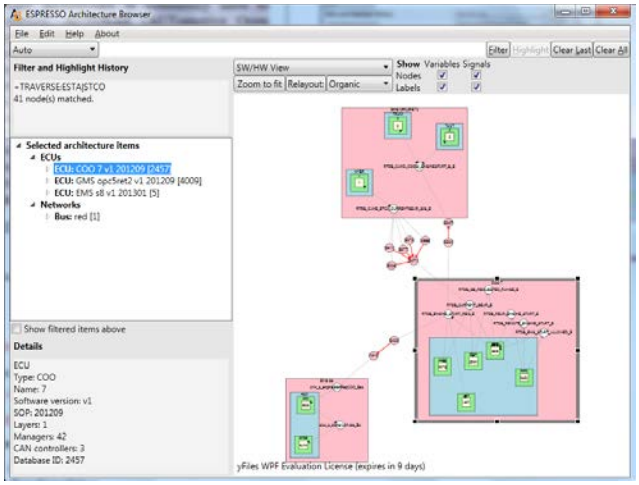

Fig. 5. A screenshot of the Architecture Browser, showing the three ECUs and signal flow from the ESTA (engine start) application component both internally within the COO and over CAN to the other ECUs.

The Architecture Browser uses an advanced filtering system in order such that only the relevant part of the architecture is made visible. Filtering can be done on all relevant model entities, and filters can be successively added onto each other to give the wanted result. Filters can be used to both add and remove content, and are also available in negated variants. Finally, filtering based on dataflow is possible, i.e. traversing the dataflow chain from a specific application module. Together these make for precise targeting of the architecture elements that are most relevant for the user.

The information is presented both in a tree view and in a hierarchical graph built with yFiles WPF. The latter can be layouted automatically and several options for display detail are given, depending on how large the depicted portion of the architecture is. For example, the visibility of labels giving the RTDB variable names can be toggled on and off.

The tool gives superior overall understanding of the architecture, enabling efficient real-time browsing of the architecture. For example, signal flow can very easily be followed by interactively and successively querying for signal chains. Hence, it is easy to use for improving understanding of the system architecture and even to trace the actual signal flow.

4) CAN Verifier

The second front-end tool of the tool set is the CAN Verifier, also built in C#. The purpose of the tool is to verify the content of the CAN communication layer, ECU by ECU, against the external design databases (SESAMMtool) at Scania. Inconsistencies such as absent buses, messages, signals and signals that have divergent definitions are detected and presented to the user, which may then use the information to improve either the code or the design database and make them more consistent.

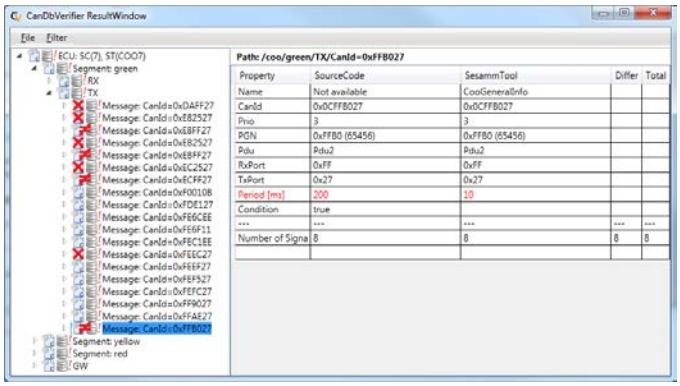

Fig. 6. A screenshot of the results window of CAN Verifier, showing a message with a mismatching period.

IV. RESULTS

The approach has been demonstrated to be feasible and practical. The toolset was validated through demonstration for the relevant system developers at Scania. A recurring comment was that the tool will help with system overview, understanding and impact analysis.

In the Architecture Browser, the architecture description is visualized in a proper way that developers can easily understand. The filter function also helps developers to focus on an expected part of the architecture or trace a certain signal. In the CAN Verifier, a number of inconsistent signals and messages were found between implementation and specification that were not previously known.

The execution time of the toolset is reasonable. The parsers and coordinator only need to be run after the source files are changed. The parser execution time is approximately three minutes per ECU. All the operations in Architecture Browser and CAN Verifier are within seconds, enabling interactive browsing. The cost of the development is also deemed acceptable. By the demonstration we made at Scania, the total workload was approximately 16 man months excluding the proof-of-concept prototypes, which were not directly reused for this phase of the project.

V. DISCUSSION

Unlike most automotive companies, which outsource much development to subcontractors, Scania performs a large part of their development in-house. In this setting, the toolset may benefit many stages within software development lifecycle such as design, verification and maintenance. The approach inherently relies on having access to the source code of the entire distributed system, which is not always the case for all automotive Original equipment manufacturers (OEMs). Still, the ideas behind this work may still be applicable for tier-1 software suppliers who provide complete subsystems and other kinds of software-intensive distributed embedded systems manufacturers.

The architectural information stored in the database can also be queried for other kinds of analysis, such as conformance checking between design model and implementation model or fault tree generation.

There were three main challenges during the work. Firstly, different parsers have to be implemented for ECUs from different departments, since they use different coding conventions and even different code structure. Secondly, no solution has yet been created for parsing the source files generated by third-party code generator, e.g. Simulink-generated code. Finally, variability related to end-of-line parameters that are configuring the ECUs at the truck assembly line is hard to resolve – it is both difficult to parse and difficult to visualize in an easily understandable way and has not yet been covered.

VI. CONCLUSION AND FUTURE WORK

In this paper, we have reported our findings in a four-month long empirical study on recovering software architecture from three representative ECUs out of an automotive embedded system. Architecture Browser presents the recovered architecture in several views and CAN Verifier checks the consistency between specification and implementation on CAN communication level. This work was demonstrated inside Scania. Feedbacks from the demonstration together with the efficiency and development cost of the toolset have also been summarized. Our work proved that this bottom-up approach can be treated as an alternative way to get one step closer to functional safety regarding ISO 26262, as compared to a more top-down one such as e.g. the AUTOSAR methodology. Future extension of this work can be addressed on the following aspects.

- Enlarging the coverage of Architecture Browser by taking variability, additional software versions, intra-component model structure and generated code into account.
- There are also plans to connect the design requirements to the recovered architecture, to enable better testing, traceability, and also combating the problem of architectural erosion of the requirements.
- Reflexion modeling can be introduced to check consistency between design and implementation under expected coverage.
- Architecture recovery can also be integrated with forward engineering to support development. One of the examples is continuous architectural supervision during software development.

REFERENCES

[1] G. C. Murphy, D. Notkin, and K. J. Sullivan, “Software reflexion models: bridging the gap between design and implementation,” *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, Apr. 2001.

[2] M. Broy, “Challenges in automotive software engineering,” *Proceeding of the 28th international conference on Software engineering - ICSE ’06*, p. 33, 2006.

[3] ISO, CD. ‘26262, Road vehicles–Functional safety.’ International Standard ISO/FDIS 26262 (2011).

[4] G. Rasool and N. Asif, “Software Architecture Recovery,” *International Journal of Computer, Information, and …*, vol. 1.1, pp. 421–426, 2007.

[5] [5] S. Ducasse and D. Pollet, “Software Architecture Reconstruction: A Process-Oriented Taxonomy,” *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 573–591, Jul. 2009.

[6] R. L. Krikhaar, “Reverse architecting approach for complex systems,” *Proceedings International Conference on Software Maintenance*, pp. 4–11, 1997.

[7] N. Ali, J. Rosik, and J. Buckley, “Characterizing real-time reflexion-based architecture recovery: An In-vivo Multi-Case Study,” in *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures - QoSA ’12*, 2012, p. 23.

[8] G. Antoniol, G. Canfora, G. Casazza, a. De Lucia, and E. Merlo, “Recovering traceability links between code and documentation,” *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 970–983, Oct. 2002.

[9] E. Wolfgang, L. Warholm, R. Klösch, and H. Gall, “Software architecture recovery of embedded software,” in *19th International Conference on Software Engineering (ICSE’97)*, 1997, no. 20477

[10] N. C. Mendonca and J. Kramer, “Developing an approach for the recovery of distributed software architectures,” in *Proceedings. 6th International Workshop on Program Comprehension. IWPC’98 (Cat. No.98TB100242)*, pp. 28–36.

[11] M. Baloh, G. Raghav, and S. Sivashankar, “Key considerations in the translation of legacy embedded control software to Model Based Executable Specifications,” in *2006 IEEE Conference on Computer Aided Control System Design*, pp. 539–544.

[12] O. Chammam and A. Zamouche, “Towares Automated Recovery of Embedded System Functional Architecture,” Master Thesis, KTH, 2013.

[13] J. Greco and B. Mokhtari, “Network Architecture Recovery in the context of Automotive CAN Communication Juan Greco,” Master Thesis, KTH, 2013.

[14] Bosch, “CAN Specification Version 2.0,” 1991.

[15] Standard, S. A. E. ‘SAE J1939 Standards Collection.’.

[16] AUTOSAR, ‘AUTOSAR–Technical Overview V2. 2.1.’ (2008).

[17] “Neo4j.” [Online]. Available: http://www.neo4j.org/.

[18] “srcML.” [Online]. Available: http://www.srcml.org/.

[19] “xPath.” [Online]. Available: http://www.w3.org/TR/xpath/.

[20] “yWorks.” [Online]. Available: http://www.yworks.com/en/index.html.