Supporting Simulink models in Systems and Software Architecture Recovery To Automotive Embedded System

Amirali Hosseinzadeh Dolkhani

This thesis is presented as part of Degree of Bachelor of Science in Electrical Engineering

Blekinge Institute of Technology
November 2013
This page intentionally left blank.
ABSTRACT

In current complex vehicles, to have correct architectural models is fundamental for automotive E/E systems development. This is important:
- to understand how subsystems are dependent on each other,
- for compliance with ISO 26262, architecture models need to be maintained
- since requirements need to linked to the architecture (also according to ISO 26262)

Architecture recovery is to recover the software or systems architecture using the source code and other related artifacts. In literature, mostly software architecture recovery has been studied, and the source of information has been source code. However, industrial automotive software consists nowadays typically of a mix between manually written source code and generated source code from Simulink. Therefore, in this project, we are aiming to extend software architecture recovery from source code by integrating also architecture recovery from Simulink models.

In addition to deriving the methodology, one aim is also to implement an architecture recovery module, dedicated to Simulink models, and integrate that module into a prototype system under development in the research project ESPRESSO. The output from the module, the architectural model, is to be stored in a graph database. Furthermore, the architectural model needs to be visualized in an architecture viewer, which is also part of the prototype System.
This page intentionally left blank.
ACKNOWLEDGMENTS

I would really like to thank:

SUPERVISOR AT SCANIA:  
MATTIAS NYBERG

ADVISER AT SCANIA:  
ANTON EINARSON

SUPERVISOR AT BTH:  
ANDRES HULTGREN

EXMINER AT BTH:  
SVEN JOHANSSON

KTH:  
JAD EL KHOURY  
MAGNUS PERSSON

ERFAN YOUSEFI
# Table of Contents

Blank page ....................................................................................................................................................................... i  
Abstract......................................................................................................................................................................... ii  
Blank page .................................................................................................................................................................... .iii  
Acknowledgement......................................................................................................................................................... iv  
Table of contents ........................................................................................................................................................... v  
List of figures............................................................................................................................................................... vi  
Short list of abbreviations ............................................................................................................................................ vii  
Introduction .................................................................................................................................................................... 1  
Background and related work ........................................................................................................................................ 3  
Design and implementation............................................................................................................................................ 4  
  First Proposed Tool chain ....................................................................................................................................... 4  
  Second Proposed Tool chain.................................................................................................................................... 9  
  Study for Solution ..................................................................................................................................................... 10  
  Third Proposed Tool chain.................................................................................................................................... 10  
Results and conclusions ................................................................................................................................................. 12  
Future works................................................................................................................................................................. 15  
Appendix A .................................................................................................................................................................. 16  
Appendix B................................................................................................................................................................... 21  
Reference ...................................................................................................................................................................... 26
# List of Figures

- Figure 1: Tool chain for .c and .h files ................................................................. 3
- Figure 2: First proposed tool chain for .mdl files according the previous work for .c & .h files .......... 4
- Figure 3: Example of .mdl file structure ................................................................. 5
- Figure 4: Constant block in .mdl format ............................................................... 6
- Figure 5: Example of valid JSON structure .......................................................... 6
- Figure 6: Sample branched connection ............................................................... 7
- Figure 7: Branched connection in .mdl format from Integrator block ................... 7
- Figure 8: Second tool chain ................................................................................ 9
- Figure 9: Third tool chain (implemented one) ....................................................... 10
- Figure 10: A block on GMS before visualization ................................................. 14
- Figure 11: The same block after visualization ....................................................... 14
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>E/E</td>
<td>Electrical/ Electronic System</td>
</tr>
<tr>
<td>ECU</td>
<td>Electronic Control Unit</td>
</tr>
<tr>
<td>EMS</td>
<td>Engine Management System</td>
</tr>
<tr>
<td>CAN</td>
<td>Controller Area Network</td>
</tr>
<tr>
<td>GMS</td>
<td>Gearbox Management System</td>
</tr>
<tr>
<td>COO</td>
<td>Coordinator</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>CV</td>
<td>CAN Verifier</td>
</tr>
<tr>
<td>AB</td>
<td>Architecture Browser</td>
</tr>
<tr>
<td>JSON</td>
<td>Java Script Object Notation</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>RTDB</td>
<td>Real Time Data Base</td>
</tr>
</tbody>
</table>
INTRODUCTION

This work has been done by two students, bachelor student (Amir Dolkhani) and Master Student (Ashkan Fathi), and results has been split up in two separated bachelor and master reports which written based on each other.

Problem statement:
For beginning it is needed to explain in short about ESPRESSO since it was mentioned above in ABSTRACT and also because this work was a small part of this big industrial, research project.

“ESPRESSO is collaboration between KTH and Scania CV AB. The project aims at providing an efficient development methodology to achieve functional safety according to ISO 26262.” [Appendix A]

For taking another step forward, there is necessity to express that:
“In software engineering, legacy code and successful software components, systems or platforms, evolve over time. During this evolution, the architecture of the legacy code normally evolves too, implying that relevant documentation and architectural models often become incomplete or inconsistent as the code changes. This problem is also apparent in the automotive industry, which has become increasingly software-intensive during the past 30 years. The accelerated system evolution accelerates the architectural drift. At the same time, the introduction of ISO 26262 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 without explicit and up to date architectural descriptions. Even without safety requirements, it becomes hard to achieve an overall understanding of the system without architectural descriptions in place.” [Appendix A]

So it is absolutely necessary to have up-to-date and synced documentation and architectural model with the implemented software components, systems or platforms. And also it is known that:
“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 verification of the synchronization.” [Appendix A]

While working on solving this issue, should keep this in mind that:
“Principles on solving the drift should not merely be addressed through allocating more effort on documenting changes, but also towards the tools bridging the gap between design and implementation” [Appendix A]
Therefor “One feasible approach to bridge the gap is software architecture recovery.” [Appendix A]

Below definition of Software architecture recovery according to this project glossary is visible:
“Software architecture recovery is a process to extract high level architectural models with a specific level of abstraction from available artifacts of the system implementation; related work also uses terms such as architecture reconstruction or reverses architecting.” [Appendix A Background]

With all this knowledge, definitions and descriptions, the suggested intelligent and agile solution is having:
“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 for CAN [1] communication” [Appendix A]

About our task:
The above text mentioned necessity of project ESPRESSO in short which our work is part of it as below:

This thesis work can be describes as process and preparation of desired data for being adapted and usable in this software architecture recovery system by Architecture Browser and CAN Verifier and make them visualize.

In brief, in this report, we will describe the procedure of study, design and propose, implementation and test of a tool(tool chain)which will make us able to use files generated by MATLAB, Simulink [2][3] in format of .mdl or new format of SLX in order to gain the goals described before and bridge the above mentioned gaps.(which also one of its side effect is leading us to have more safety as result of having synced and up to date
architectural model with implemented one and make us able to analysis the whole system better and more accurate.)
BACKGROUND AND RELATED WORK

Background:
As background to our work according to Appendix A this kind of tool chain has been designed for other kind of files which are used in industry generally like .c, .h and YML files.

Existed tool chain for .c and .h files:
As you can see in Figure 1, .c and .h files getting converted to XML in first step by using ‘src2srcml’\(^1\) [4], then parsers parse these XML files and after that by using of parser coordinator sent to Neo4j database (graph database) which is used by Architecture Browser and CAN verifier.

![Figure 1: Tool chain for .c and .h files](image)

More complete explanation with explicit description of this project’s background is accessible in [Appendix A].

\(^{1}\) src2srcml - translates source code into the XML source-code representation srcML[4]
DESIGN AND IMPLEMENTATION

First proposed Tool chain:
The first solution has been decided to study, was using the same strategy that had been used for .c and .h files. It means if there is a possibility that we can adopt the same tool chain (as was used for .c and .h) or we need to design another one.

For beginning, we tried to keep the main body of existed tool chain without any change but replace the feeding file by .mdl files instead of .c and .h as you can see in Figure 2.

![Figure 2: First proposed tool chain for .mdl files according the existed tool chain for .c & .h file](image)

To investigate this solution, it should be known what type of code .mdl is and know its structure (1) then it should be understood what does mean parsing (2), also find out how convert .mdl to XML can be done (3), since these three subjects are essential for checking the possibility of this strategy.
1) Structure of .mdl file:
“Schemes created in Simulink are stored in plain text files with .mdl extension. Files with extension .mdl are easy-to-read for human. The structure of file consists of key-value pairs, while the key is a name of an attribute and the value may be single string or numeric value, vector, matrix or a composite value with one or more nested key-value pairs.

The attribute name is always a single word without spaces followed by one or more spaces and/or tabs to achieve aligned columns – see Figure 3.

![Figure 3: Example of .mdl file structure](image)

As mentioned earlier in this section, attribute value may be present in these forms:

- numeric – any integer or float number
- boolean – possible values: on, off
- string – zero or more characters enclosed in quotes
- vector or matrix – standard MATLAB notation (e.g. [1, 2, 3] or [1, 2, 3; 4, 5, 6] )
- composite – key-value pairs enclosed in curly brackets

The whole structure of .mdl files resembles the JSON format (compare Figure 4 and Figure 5) introduced by Douglas Crockford in 2002 and standardized by RFC 4627 in July 2006. The JSON is widely used by web applications as a lightweight standard for data exchange. While in valid JSON, each attribute name must be wrapped in double quotes followed by a colon; there are no quotes, nor colon in the .mdl format. In addition, attribute/name pairs in .mdl are not comma separated as they are in JSON.

.mdl files used in Simulink have one feature that is missing in JSON, which is ability to use matrices written in the same format as in MATLAB. JSON contains only arrays that are similar to vectors used in MATLAB. More information about JSON is available on official website.

An .mdl file contains a plenty of data but we are not interested in all of them. For our needs we take into consideration only the name of a scheme, time and date of creation and last modification of the scheme and authors (these parts are marked with bold font in Figure 4).

Except of mentioned metadata, an .mdl file contains 2 important parts that describe the whole scheme. The first one, labeled as BlockParameterDefaults, contains a list of block types that appears in the scheme with a list of
parameters used as default values for each block type. These values are taken into account if the block in a scheme lacks specification of certain parameter value.

```json
System {
    Name          "Template",
    Location      \[480, 93, 1016, 386\],
    ...
    Block {
        BlockType Constant
        Name          "Constant",
        Position      \[140, 174, 210, 196\],
        ShowName      off
        Value         "1",
        OutDataType   "sfix(16)",
        OutScaling    "2^0"
    }
    Block {
    ...

Figure 4: Constant block in .mdl format

"System": {
    "Name": "Template",
    "Location": \[480, 93, 1016, 386\],
    ...
    "Block": {
        "BlockType": "Constant",
        "Name": "Constant",
        "Position": \[140, 174, 210, 196\],
        "ShowName": false,
        "Value": "1",
        "OutDataType": "sfix(16)",
        "OutScaling": "2^0"
    }
    "Block": {
    ...

Figure 5: Example of valid JSON structure

The second and most important part of an .mdl file is labeled as System. This part contains complete list of blocks and connections that appear in the scheme. Names of blocks are used as a unique identifier. List of parameters (bold in Figure 4) is specified directly inside Block attribute after specification of block type, name, location and other similar properties. It contains only parameters with values other than default. A connection is specified with the name of source and destination block and number of source and destination port. The shape depends on coordinates of points that the connection passes through. Coordinates are relative to the exact location of source port and they are noted as a matrix near Points keyword.

In case of more complex connections with nodes, the Branch keywords define multiple destination blocks. For an example of such connection, see Figure 6 and Figure 7. ” [8]
What does mean parsing:

The parsing definition cited form *Parsing Techniques a practical guide*:

“Parsing is the process of structuring a linear representation in accordance with a given grammar. This definition has been kept abstract on purpose to allow as wide an interpretation as possible. The “linear representation” may be a sentence, a computer program, a knitting pattern, a sequence of geological strata, a piece of music, actions in ritual behavior, in short any linear sequence in which the preceding elements in some way restrict the next element. For some of the examples the grammar is well known, for some it is an object of research, and for some our notion of a grammar is only just beginning to take shape.

For each grammar, there are generally an infinite number of linear representations (“sentences”) that can be structured with it. That is, a finite-size grammar can supply structure to an infinite number of sentences. This is the main strength of the grammar paradigm and indeed the main source of the importance of grammars: they summarize succinctly the structure of an infinite number of objects of a certain class.

There are several reasons to perform this structuring process called parsing. One reason derives from the fact that the obtained structure helps us to process the object further. When we know that a certain segment of a sentence is the subject that information helps in understanding or translating the sentence. Once the structure of a document has been brought to the surface, it can be converted more easily.

A second reason is related to the fact that the grammar in a sense represents our understanding of the observed sentences: the better a grammar we can give for the movements of bees, the deeper our understanding is of them. A third lies in the completion of missing information that parsers, and especially error-repairing parsers, can provide. Given a reasonable grammar of the language, an error-repairing parser can suggest possible word classes for missing or unknown words on clay tablets.” [9]
Now understood what parsing means and how does it work also find out how the structure of .mdl is and how its structure look likes as our two essential concepts for checking the possibility of using the existed tool chain (.c and .h tool chain but with .mdl files as input), The next step is to looking up for the third one which it is methods to convert this structure (.mdl) into XML in a way that is usable in existing or new suitable parsers to parse it.

3) Converting .mdl files to XML
A: it was learned that converting .mdl files to simple XML by already existed solution built in MATLAB is possible. This solution is using of `save-system` on already existed .mdl files.

```
save_system(sys, 'exported_file_name.xml', 'ExportToXML', true)
```

B: In addition we can convert .mdl files into SLX simply by saving .mdl files in MATLAB as SLX files which itself is XML type code. [5]

Both A and B we called Upgrade Models to SLX, so we did try A and B methods and generate two different files according to them. [6]

By knowing the structure of .mdl files before conversion and study converted files into XML by using save-system according to A, also checking the files stored in SLX according to B (which we found that they are C-type XML). We find that there are differences between these two generated XML files and XML files which already exists, parsers for .c and .h can parse them although both of them are XML type which reminds us that we need new parser for these two different types of XML files to gain the desired results.
Second proposed Tool chain:
Second proposed tool chain shown in Figure 8 designed based on our study to improving tool chain for .c and .h with this small revision that we add one more ring to this tool chain which it is “Simulink Coder”.

“Simulink Coder” generates and executes C and C++ code from Simulink diagrams, State flow charts, and MATLAB functions. So in this case we only have to use “Simulink Coder” to generate C codes from Simulink model and then deliver .c file to src2srcML and the rest of our tool chain would be the same. Still there are two issues remained, the XML codes generated by “src2srcml” needs new parsers (1), and the extra fee that we need to pay for “Simulink Coder”, make it costly in compare with other solutions (2).

![Second Tool chain](image)

After taking these initial steps and study designed tool chains by us, we found that we are really in need to find a solution with high reliability and stability and less complexity that make us able to use it by almost all .mdl files with consideration of all below requirements.

- Knowing that there are differences between XML files, generated by MATLAB in different methods and XML files which existed parsers can parse. (it means an urge to write new parsers)
- This fact that tremendous amount of already existed .mdl files need to convert to XML which it means spending huge amount of time, money and process.
- This knowledge that industry is constantly under development and change and in result the formats of files which Simulink uses to save schemas are under influence. [7]
- Plus this essential necessity that the solution (proposed tool chain) should be able to work for all kind of files which are exist and will be generate in future by Simulink MATLAB.

As another step to forward we begin to search for the alternative methods according to these attributes (more reliability and stability and less complexity) which brings us to this conclusion that may be we need to design and implement a whole new tool chain from A to Z to satisfy this goal.
Study for solution:
We need a native tool inside of MATLAB (for having maximize stability) which allows us to communicate with Simulink and let us to extract data from .mdl files which bring us to this point that an API can solve this solution for us.

What is an API?
“API Stands for Application Programming Interface. An API can provide a hook for colleagues, Partners, or third-party developers to access data and services to build in applications such as iPhone app quickly.” [10]

In our case we need this hook help us as a third-party to access to Simulink’s data. As result, we come to this conclusion to find a native API within MATLAB since there are many advantages in using native API like maximum stability and prevent unnecessary changes in API in case of further change in MATLAB or Simulink.

According our study the best set of these API’s in line with our goals already build in MATLAB as documented API command lines [7], such as “find_system ”, “get_param” and “set_param” which can let us to extract a large variety of data from .mdl files or set them.

In this point, it’s better to remind an important subject .we assume two elements fixed in our tool chain, first one is .mdl file which it is feed for our tool chain and the second one is our Neo4j database and its database handler in other hand the first and the last ring in our tool chain.
In result, it’s logical to consider this two fixed components as a key factor in choosing another parts of our tool chain to have maximum synchronization in order to improve the integrity, efficiency and speed of our tool chain.
So as we have our data base handler and our Neo4j codes in java, also considering above discussion we prefer to have our codes in java plus it was suggested by our supervisor at Scania to use it because of other technical reasons.
Next step we need to take is choosing an IDE for coding and another API which let this IDE to communicate with MATLAB. Between Eclipse and NetBeans, we chose Eclipse.
For API which lets Eclipse and MATLAB communicate, according our study (and suggestion Kth-simulink-exchange) two options could help us to make our job done, “Kth-simulink-exchange” and “matlabcontrol”.
Finally we made our decision to use “matlabcontrol” in our project because of having two main reasons
First: in compare with “Kth-simulink-exchange” and with respect to our goals like simplicity “matlabcontrol” is first choice because of using “eval” and “feval”.
Second: because of some legally issues it was suggested by our supervisor to use “matlabcontrol”

Third proposed Tool chain:
According to mentioned above our tool chain which have required specification and functionality will be:

![Third tool chain (implemented one)](image_url)

As you can see in our final tool chain Simulink models (.mdl or SLX files) are feed of our tool chain, then you can see MATLAB environment which we can communicate by using API command lines as below:
By using of “find_system” we can find systems, blocks, lines, ports and annotations.
By using of “get_param” we can get parameters name and extract data in Simulink models.
And by using of “set_param” we can set system and block parameters (which we have no use)

The next environment in our tool chain is Java (Eclipse) which can communicate with MATLAB by create a
proxy via “matlabcontrol” so we can control MATLAB via Java.
We use “eval” and “Returning_eval” as below in this part of tool chain:

“eval” will let us to control MATLAB from Java and remotely run our command.
“Returning_eval” will let to get back the result of “eval” to Java.

Then we need to have our RTDB variables by help of two java class which will be written (“Group.java”) and
finally by help of “DBhandler” we will have our data in DB(Neo4j).

**Study to organize a system for coding:**

Each model in Simulink despite of its type is consist of system, subsystem, blocks, signal, etc.so we tried to find
how the program can make a distinguish between them?
And we found that in Simulink models, a unique number is assigned to each single element which is exist (all
kind of blocks, lines, branch and trunk lines)
This unique number called “Handle”.
By calling that handle within the suitable command you can have all desired data. Below you can find in details
how we use handles to obtain our goal.
RESULTS AND CONCLUSIONS

Implementing code in MATLAB:
As almost all of complex SIMULINK models contain several subsystems the first challenge in MATLAB code is, how to distinguish subsystems from simple blocks in the model? Since subsystems need to be considered as separate systems rather than normal blocks because they may consist of several blocks and lines themselves.

First of all, we thought that if we could find a full path for each block in the system, then we are able to draw a proper hierarchy for whole model. Obviously, subsystems will have some children however normal blocks will not. Using “find_system” command enables us to have a complete path (full path) for each existing block in the considered model.

\[
\text{BlockPath} = \text{find_system(name, 'type', 'block')};
\]

“find_system” command in this case accepts system name as input and according to the specified attribute pair (‘type’ and ‘block’) returns full block path for each block.

Also, if we could specify type of each block we could easily say if the block is a subsystem or not and it is possible by using “get_param” command.

\[
\text{blockType} = \text{get_param(BlockPath, 'BlockType')};
\]

“get_param” command in this case accepts block path and according to specified attribute (‘BlockType’), returns block type for each existing block in the model.

Now we have extracted full path and block type for each block and these information could be enough to propose a proper solution for the first challenge and sketch desired hierarchy for the model.

Line (signal) recognition is the second challenge, since we need to specify the connection between different blocks or simply which block connects to which block(s).

A line or a signal would be defined by a source block and a destination block. Therefore, we just need to find a source block and a destination block for a specific line. Hence, after some investigation on this issue we find out, again using “find_system” command with more attribute pairs enables us to extract line handle which is a unique ID for each line and then after that using “get_param” command to retrieve source port handle and destination port handle which are unique IDs respectively for source port and destination port for each specific line.

\[
\text{lineHandle} = \text{find_system(name, 'FindAll', 'on', 'type', 'line')};
\]

“find_system” command in this case accepts system name as input and according to the specified attribute pairs (‘FindAll’, ‘on’) and (‘type’, ‘line’), returns all existing line handles in the model.

\[
\text{sourcePortHandle} = \text{get_param(lineHandle, 'SrcPortHandle')};
\]

\[
\text{destinationPortHandle} = \text{get_param(lineHandle, 'DstPortHandle')};
\]

“get_param” in these cases accepts line handle as input and according to the specified attributes (‘SrcPortHandle’) and (‘DstPortHandle’) respectively returns source port handle and destination block handle for each line.

As we expected, these information seems enough in order to propose a solution for the second challenge however, imagine there might be some lines which have the same source block (source port) but different destination blocks (destination ports).

So, we would have two different types of source block, first type with a unique destination block and second type with two or more than two destination blocks.

Hence, after some more study and inquiry, we realized that line recognition needs to be done in two different separate procedures since we have two different source block types. Afterward, we decide to divide lines to tow different existing types in Simulink called “trunk” and “branch” for the rest of studies. Again, using “find_system” command gives us trunk line handles and branch line handles as below.
trunkHandle = find_system (name, 'FindAll', 'on', 'type', 'line', 'segmentType', 'trunk');

branchHandle = find_system (name, 'FindAll', 'on', 'type', 'line', 'segmentType', 'branch');

“find_system” command in this case accepts system name as input and according to the specified attribute pairs ('FindAll', 'on'), ('type', 'line') and ('segmentType', 'trunk' or 'segmentType', 'branch'), returns all existing trunk line handles or branch line handles in the model.

Therefore, if we have a trunk line handle with a unique destination port handle, we can say it is the first scenario which means that there is a unique source and a unique destination.

On the other hand, if we have a branch line handle with a unique source port handle and a unique destination handle, we can say it’s the second scenario which means that the branch line is connected to a unique source which is a common source (common between at least two lines) and at the same time has its own unique destination.

Then, after this step, again it is time to use “get_param” command in order to retrieve source port handles and destination port handles for different line types.

trunkSourcePortHandle = get_param(trunkSourceHandle, 'SrcPortHandle');

branchSourcePortHandle = get_param(branchSourceHandle, 'SrcPortHandle');

trunkDestinationPortHandle = get_param(trunkSourceHandle, 'DstPortHandle');

branchDestinationPortHandle = get_param(branchSourceHandle, 'DstPortHandle');

“get_param” in this case accepts trunk source handle or branch source handle as input and according to specified attributes ('SrcPortHandle' or 'SrcPortHandle') returns source port handles and destination port handles for both trunk and branch line types.

Then, considering the extracted information from the model, now we need to characterize source and destination block paths in order to define each line. Using “getfullname” command gives us a full path containing the name of source and destination blocks.

trunkSourceName = getfullname(trunkSourcePortHandle);

branchSourceName = getfullname(branchSourcePortHandle);

“getfullname” in this case accepts source port handles as input in order to retrieve source full path for both trunk and branch line types.

trunkDestinationName = getfullname(trunkDestinationPortHandle);

branchDestinationName = getfullname(branchDestinationPortHandle);

“getfullname” in this case accepts destination port handles as input in order to retrieve destination full path for both trunk and branch line types.

Thus, we have a complete list of source blocks and the correspondent destination blocks which provides enough data in order to implement the solution for the second challenge.

Implementing code in Java:
As we mentioned before, after proper studies and investigations we decided to use “matlabControl” application in order to establish a connection between MATLAB and Java through the provided proxy.

Also, this proxy enables us to obtain an interaction between MATLAB and Java through “proxy.eval()” and “proxy.returningEval()”. 

13
"proxy.eval()" evaluates MATLAB commands from Java workspace and gives the opportunity to execute our developed MATLAB function from Java. On the other hand, "proxy.returningEval()" retrieves the extracted data by MATLAB function which is available in MATLAB workspace.

Thus, using these two syntaxes ("proxy.eval()" and "proxy.returningEval()") we will have exactly the same data in Java workspace this time which we used to extract from the Simulink model in MATLAB workspace.

Extracted data are:
I) Block types
II) Block paths
III) Source block paths
IV) Destination block paths

We would need (I) and (II) in order to create the same hierarchy, (III) and (VI) in order to define existing lines in the model so, we retrieve these data from MATLAB to Java and now the data is ready to parse a bit through Java code in order to be compatible with the database handler input format.

In Figure10 you can see a block diagram (GMS) before visualization and implementation our job, and in Figure11 you can see same block diagram after our job and visualization was done.
Future Work

During writing this report we found that maybe we can improve exist work by working on special blocks like “bus selector” or other more complicated blocks to see if more data we can extract from them or not in order to use for other possible use.
APPENDIX A
Experience on Applying Software Architecture Recovery to Automotive Embedded Systems

Xinhai Zhang¹, Magnus Persson¹, Mattias Nyberg²,²¹, Behrooz Mokhtari²,²¹, Anton Einarson², Henrik Linder³, Jonas Westman²,²¹, DeJiu Chen¹, Martin Törngren¹

¹Kungliga Tekniska Högskolan, Stockholm, Sweden {xinhai, magnper}@kth.se
²Scania CV AB Södertälje, Sweden mattias.nyberg@scania.com
³HiQ AB, Stockholm, Sweden m.n.yberg@hiq.se

Abstract—The importance and potential advantages of 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 learnt are discussed.

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

I. INTRODUCTION

In software engineering, legacy code and successful software components, systems or platforms, evolve over time. During this evolution, the architecture of the legacy code normally evolves too, implying that relevant documentation and architectural models often become incomplete or inconsistent as the code changes [1].

This problem is also apparent 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 without explicit and up-to-date architectural descriptions. Even without safety requirements, it becomes hard to achieve an overall understanding of the system without architectural descriptions in place.

Architectural drift happens not only because keeping documentation 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 verification of the synchronization [1]. Therefore, principles on solving the drift should not merely be addressed through allocating more effort on documenting changes, but also towards 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 for CAN communication.

In the rest of the paper, we describe as follows: A brief state of practice survey on software architecture recovery for distributed embedded systems is provided. (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 is a process to extract high-level architectural models with a specified level of abstraction from available artifacts of the system implementation [4]; related work also uses terms such as architecture reconstruction [5] or reverse architecting [6]. 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 work regarding architecture recovery treats 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. Some later work on architecture recovery for 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.

A different approach is to start out from an architecture model, and then proceed towards detailed design and implementation. Normally, however, such approaches suffer from architectural drift unless synthesis from the architecture model is possible, and changes to code are not allowed to violate the architecture. One such approach is outlined by [12]...
III. EMPIRICAL STUDY

Our work is a part of ESPRESSO project, a collaboration between KTH and Scania CV AB. The project aims at providing an efficient development methodology to achieve functional safety according to ISO 26262. The work presented here is part of long term work, and was preceded by initial separate proof-of-concept prototypes [13], [14]. We reorganized, improved and consolidated these concepts and implemented them into a consistent toolset.

A. Context of the Study

On a Scania truck, most of the ECUs (embedded computers) are distributed over three main CAN [15] 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 [16]. 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.

![Fig. 1. Example of network topology in vehicle from Scania](image)

AUTOSAR [17] is not used in Scania. Instead, Scania uses a proprietary 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 detailed enough for the description in this paper. The one in actual use is slightly different.](image)

An Application Component encapsulates part of one function. Each application component just contains one real-time task which can be either periodic or event triggered. Normally, an application component is managed as a .c file associated with related header and calibration files. 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), which can be seen as the equivalent to the RTE in AUTOSAR, stores information that is shared between Application Components and also supports inter-ECU communication. Variables stored in RTDB are called RTDB variables. 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 one RTDB variable which is to be sent to or received from another ECU over the CAN bus. RTDB variables are updated 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

The developed tool-chain, as illustrated in Fig. 3, consists of four main parts: A back-end with parsers and a parser coordinator, a Neo4J [18] database, and two front-end applications: Architecture Browser and CAN Verifier.

![Fig. 3. Overview of the developed tool chain](image)

1) Parser and Coordinator

The main work of the parsers 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 parsers and the coordinator are built in Java.

![Fig. 4. Meta-model of information that need to be extracted from source code](image)
To directly parse the C files is difficult and error-prone. Therefore, we use srcML toolkit [19] to transform all the source files into XML files in which C code is wrapped with information from the Abstract Syntax Tree (AST). These XML files together with their positions in the folder structure form the inputs of the parsers. 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. Other information illustrated in the meta-model is extracted from the generated XML files using XPath [20].

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 [21] 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.

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 laid out 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 also 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.

![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.](image)

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 person 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 software 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). The ideas behind this work may also 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 had 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 generators, e.g. Simulink-generated code. Finally, variability related to end-of-line parameters that configure 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.

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. The Architecture Browser presents the recovered architecture in several views and the CAN Verifier checks the consistency between specification and implementation on CAN communication level. The results were demonstrated at Scania. Feedback from the demonstration, together with the efficiency and development cost of the toolset, has also been summarized. Our work has shown that such a 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 extensions of this work include the following aspects:

- Enlarging the coverage of Architecture Browser by taking variability, additional software versions, intra-component model and generated code into account.
- 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

Appendix B
public class Source {
    // Create a proxy, which we will use to control MATLAB.
    public static MatlabProxyFactory factory = new MatlabProxyFactory();
    public static Object[] BPaths;
    public static Object[] BTypes;
    public static Object[] SrcPaths;
    public static Object[] DstPaths;

    public static void main(String[] args) throws MatlabConnectionException,
    MatlabInvocationException {
        String fileName = "toyModel";
        MatlabProxy proxy = factory.getProxy();
        proxy.eval("[BPaths, BTypes, SrcPaths, DstPaths] = Scania(" + fileName + ")");
        // Returning the extracted data to Java.
        BPaths = proxy.returningEval("BPaths", 1);
        BTypes = proxy.returningEval("BTypes", 1);
        SrcPaths = proxy.returningEval("SrcPaths", 1);
        DstPaths = proxy.returningEval("DstPaths", 1);
        String BP = Arrays.deepToString((Object[]) BPaths);
        String BT = Arrays.deepToString((Object[]) BTypes);
        String SP = Arrays.deepToString((Object[]) SrcPaths);
        String DP = Arrays.deepToString((Object[]) DstPaths);
        // Split the returned data
        String splitBP[] = BP.split(",");
        String splitBT[] = BT.split(",");
        String splitSP[] = SP.split(",");
        String splitDP[] = DP.split(",");
        for (int i = 0; i < splitBP.length; i++) {
            splitBT[i] = splitBT[i].replace("[", "").replace("]", ").trim();
            splitBP[i] = splitBP[i].replace("[", ").replace("]", ").trim();
        }
        HashMap<String, Group> mapG = new HashMap<String, Group>();
        Group main = new Group();
        main.groupName = fileName;
        mapG.put(fileName, main);
```java
for (int i=0; i<splitBP.length; i++) {
    String rPath = splitBP[i];
    String rType = splitBT[i];
    String moop[] = rPath.split("/");
    String rName = moop[moop.length - 1];
    String parrent = rPath.replaceAll("/' + rName + ";");
    Group parrentGroup = mapG.get(parrent);
    Group Sub = new Group();
    if (parrentGroup != null) {
        Sub.groupName = rName;
        parrentGroup.subSystem.add(Sub);
    }
    mapG.put(rPath, Sub);
    System.out.println("the block name is " + rName + " and its parent is "+ parrent);
}
System.out.println("\n");
String[] Source = new String[splitSP.length];
String[] Destination = new String[splitSP.length];
for (int i = 0; i < splitSP.length; i++) {
    String SrcPath = splitSP[i].replace("[", " ").replace(", ", " ").trim();
    int srcIndex = SrcPath.lastIndexOf("/");
    String newSrc = SrcPath;
    if (srcIndex >= 0)
        newSrc = (String) SrcPath.subSequence(0, srcIndex);
    String DstPath = splitDP[i].replace("[", " ").replace(", ", " ").trim();
    int dstIndex = DstPath.lastIndexOf("/");
    String newDst = DstPath;
    if (dstIndex >= 0)
        newDst = (String) DstPath.subSequence(0, dstIndex);
    Source[i] = newSrc;
    Destination[i] = newDst;
}
String[][] lines = new String[splitSP.length][2];
for (int i = 0; i < splitSP.length; i++) {
    for (int j = 0; j < 2; j++) {
        if (j == 0) {
            lines[i][j] = Source[i];
            System.out.println("line number: " + i + ") Source:
" + lines[i][j]);
        } else {
            lines[i][j] = Destination[i];
            System.out.println("Destination: " + lines[i][j]);
        }
    }
}
```
24
85. }
86. proxy.disconnect();
87. }
88. }

Groups:

1. import java.util.*;
2.
3. public class Group {
4.
5.     public String groupName = "";
6.
7.     public List <Vars> read_rtdbvariables = new ArrayList<Vars>();
8.     public List <Vars> write_rtdbvariables = new ArrayList<Vars>();
9.     public List <Vars> create_rtdbvariables = new ArrayList<Vars>();
10.    public List <Vars> subBlock = new ArrayList<Vars>();
11.    public List <Group> subSystem = new ArrayList<Group>();
12. }

MATLAB Code:

%This function gets Simulink model name and returns block paths, block types,
%source block paths for a line and destination block path for the same line.
function [BPaths, BTypes, SrcPaths, DstPaths] = Scania(filename)
%open the system.
open(filename);
[pathstr, name, ext] = fileparts(filename)
%BPaths = find_system(name, 'type', 'block');
%BTypes = get_param(BPaths,'BlockType');
%SrcPaths = find_system(name, 'FindAll', 'on', 'type', 'line', 'segmentType', 'branch');
%DstPaths = find_system(name, 'FindAll', 'on', 'type', 'line', 'segmentType', 'trunk');

%filter the line handles to get rid of useless information and find the source port handles.
bSrcH = [];
    for i = 1:length(branchH),
        BH = branchH(i);
        if (isempty(get_param(BH, 'LineChildren')) && (any(get_param(BH, 'SrcPortHandle') ~= -1)))
            bSrcH(length(bSrcH)+1,:) = BH;
        end
    end

tSrcH = [];
    for i = 1:length(trunkH),
        TH = trunkH(i);
        if (isempty(get_param(TH, 'LineChildren')) && (any(get_param(TH, 'DstPortHandle') ~= -1)))
            tSrcH(length(tSrcH)+1,:) = TH;
        end
    end

24
%find the source and destination port handles.
tSrcPortH = get_param(tSrcH, 'SrcPortHandle');
bSrcPortH = get_param(bSrcH, 'SrcPortHandle');
tDstPortH = get_param(tSrcH, 'DstPortHandle');
bDstPortH = get_param(bSrcH, 'DstPortHandle');

%find path for trunk and branch source handles.
tSrcName = getfullname(tSrcPortH);
bSrcName = getfullname(bSrcPortH);
%find path for trunk and branch destination handles.
tDstName = getfullname(tDstPortH);
bDstName = getfullname(bDstPortH);
%join trunk and branch paths for source and destination.
SrcPaths = [tSrcName; bSrcName];
DstPaths = [tDstName; bDstName];
end
REFERENCES AND WORKS CITED


