UNIVERSITY OF GOTHENBURG

MEASURING THE ABSTRACTION GAIN OF CODE-NEAR MODELS

A CASE STUDY OF VOLVO CAR CORPORATION

Bachelor of Science Thesis in the Programme Software Engineering & Management

ANASTASIA KONI
VICTOR EFFIOK

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, June 2016

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

{MEASURING THE ABSTRACTION GAIN OF CODE-NEAR MODELS}
[A CASE STUDY OF VOLVO CAR CORPORATION]

{ANASTASIA. KONI}
[VICTOR. EFFIOK]

{©ANASTASIA. KONI, June 16}
[© VICTOR. EFFIOK, June 2016]

Examiner: {MOHAMMAD. MOUSAVI}

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg,Sweden June 2016

Contents:

ABSTRACT

1. INTRODUCTION

1.1 Organisation Background

1.2 Thesis Goal

1.3 Research Questions

1.4 Structure

2. FOUNDATION

2.1 Simulink

2.2 Model-Driven Engineering (MDE)

2.3 Code Abstraction Gain Setting

2.4 Code Generation & Documentation

2.5 Model Code Traceability

3. METHODOLOGY

3.1 Pre- Study

3.2 Empirical Research Method

3.3 Validity Discussion

4. PROTOTYPE

4.1 Building Prototype

4.2 Applying Prototype

5. STUDY'S RESULTS

5.1 Traceability

5.2 Results from tool

6. DISCUSSION

6.1 Future Work

7. CONCLUSION

8. ACKNOWLEDGMENTS

9. REFERENCES

ABSTRACT

Abstraction in computer science is defined as the simplification of a system. Nevertheless, abstraction is also relevant in software engineering when we deal with Model Driven Development (MDD). What interest software engineers is whether a model represents a simplification of the code, generated by the model. Thus, it is of significant interest and importance to investigate the existence of abstraction gain from generated code to the model.

1. INTRODUCTION

Software development has been changing and evolving continuously in recent times, leading to invention of the method for defining software architecture solutions called Model Driven Development (MDD). MDD gives architects the capability to define and communicate a solution while generating artefacts that become part of a segment of the general solution. Models are used for designing and developing platforms that support the generation of code and facilitate 360 degrees synchronisation between the code and the diagram. Recent findings have shown that MDD is popular in both academia and industry as a way to handle the increasing complexity of modern software, and it is seen by many as the next step to handle the level of abstraction at which we build, maintain and reason about software [10]. Hebig (2014) argues that the core motivation for the use of models is to provide a more abstract view of the required software system which supports communication as well as documentation or even automated analysis of systems in context of quality assurance [6]. MDD includes various model-driven approaches to software development including model driven architecture, domain-specific modelling and model-integrated computing [10]. Code abstraction is considered beneficial to developers, sequel to the fact that it reduces code size. Sutter et al. (2003) argue that code abstraction is a technique in which program fragment that occurs multiple times in a program is abstracted into a separate procedure [8]. Sutter et al. (2003) reason that code abstraction technique is efficient in reducing the code size of a program, but it also has significant risk in terms of runtime overhead [8].

Our aim in this thesis work is to evaluate abstraction gain between model and generated code within a modelling language (Simulink) using a data set from Volvo Cars Corporation.

One of the main arguments that model-based code generation is beneficial, is that the level of abstraction is increased. This hope stems from two effects: on the one hand the models are said to be more intuitive than code and on the other hand, due to the code generation, developers have to specify less details in the modelling language to create the same code compared to manual coding.

In this thesis, we focus on this latter part of the hopes related to abstraction. Therefore, we define the term “**Abstraction gain**” as the difference in the amount of words or sentences in a language (e.g. blocks in a model and keywords or lines of code in source code) that need to be specified to create the same source code, when comparing model-based code generation and manual programming. However, this is our definition since there is limited literature on the subject.

For measuring the abstraction gain, it is necessary to decide for a unit of measurement. Within this thesis, we decided to compare blocks in the modelling language (i.e. single model elements which are comparable to “words” in a textual language) with lines of code in the generated source code

(i.e. sentences in a textual language). However, this does not exclude the potential use of other units of measurement.

In the same vein, whether the abstraction gain is significant enough to be considered as abstraction gain is not defined by any scholar. Thus, in order to define if we really have abstraction gain from code to model, we waited for the response from Volvo developers, since they are the ones who use this system and are more fit than us to determine whether the abstraction gain is significant. A different interpretation of abstraction gain, is the reduced work effort needed to develop the model in comparison to the manually produced source code.

1.1 Organisation Background

As a way of creating a better understanding to the reader, we will reflect on Volvo Cars Corporation (VCC) – the organisation which this thesis work was carried out. Volvo Cars Corporation is car manufacturing company, with headquarters in Goteborg, Sweden. The company was established in 1927. Volvo Car Corporation's vision is to be the world's most progressive and desired premium car brand. Currently, manufactures wide range of premium cars that includes sedan, SUVs, sports wagon, cross country cars and wagons. This thesis work was conducted in conjunction with the Volvo Cars Corporation IT section.

1.2 Thesis Goal

Implementation of MDD in automotive industry is subject to further research. However, MDD contributions have improved productivity in the automotive industry. Sequel to this reasoning, this thesis addresses the following challenges. It examines if a model, written within a modelling language is more abstract than the source code generated by the same modelling language and whether there are gains in abstraction between model and source code.

1.3 Research Question

The following research questions will be addressed in the thesis:

1. How can we statically measure an abstraction gain based on the model and code artefacts?

1.1. To what extent can elements from the models be traced to corresponding parts of the generated code?

We are interested in code traceability and abstraction gain because they are at an infant stage in software engineering.

1.4 Structure

In this report, we present an empirical study of code abstraction gain practice, traceability of generated code and executable modelling and MDD in automotive industry. The "Introductory" chapter gives a brief introduction of MDD. This is followed by the background view of this report. The next section reflects on organisation background, thesis goal, research questions and the structure of this report. Chapter two gives the theoretical framework; issues like validity discussion and empirical research methods will also be mentioned in chapter 3. Chapter 4 gives an overview of our prototype in theory and practice. Chapter 5 reflects on the result of the study. Chapter 6 reflects on general discussion while Chapter 7 gives our insight to the conclusion.

2. FOUNDATION

This thesis work is based on a data set from Volvo Car Corporation which uses the AUTOSAR software architecture standard. It relates to software development, testing, usage and maintenance of embedded software system in the automotive industry. The automotive domain, such as automotive software development, has seen a significant development on model driven methods [11]. Due to a set of challenges, automotive software engineering requires substantial research work. There are many ways to improve automotive software, with Model Driven Development (MDD) being one of the most popular candidates, according to Manfred Broy [3]. Models are of great importance to the particular field we examine and they can contribute to many different necessities that occur during the development of an automotive software product. Information from documentation, to analysis and transformation. The semantics of modelling languages have not been formalised yet, as a result model driven development is used in fractions of the development process [3]. There are efforts by many different parties and many proposals for formalisation exist, but still there is not a unified formalisation for each modelling language. Another issue is that the techniques for the measurement of abstraction gain between model and generated code are missing. In this thesis work, we will research deeply on abstraction gain on generated code and the traceability of models to code blocks. However, modelling languages are not used yet in their full capacity and there is still space for lot of research.

This thesis work, we used Simulink graphical programming environment (an add-on product to Matlab) that supports simulating, automatic code generation, analysing multi-domain models and testing & verification of embedded systems. Simulink is a multi-discipline tool which enhances effective modelling with the use of graphical user interface (GUI) and customizable block libraries for building models as block diagram and simulating dynamic systems. For effective modelling of a system, it requires selecting the appropriate blocks and connecting them to each other, in a pattern that represent the mathematical models.

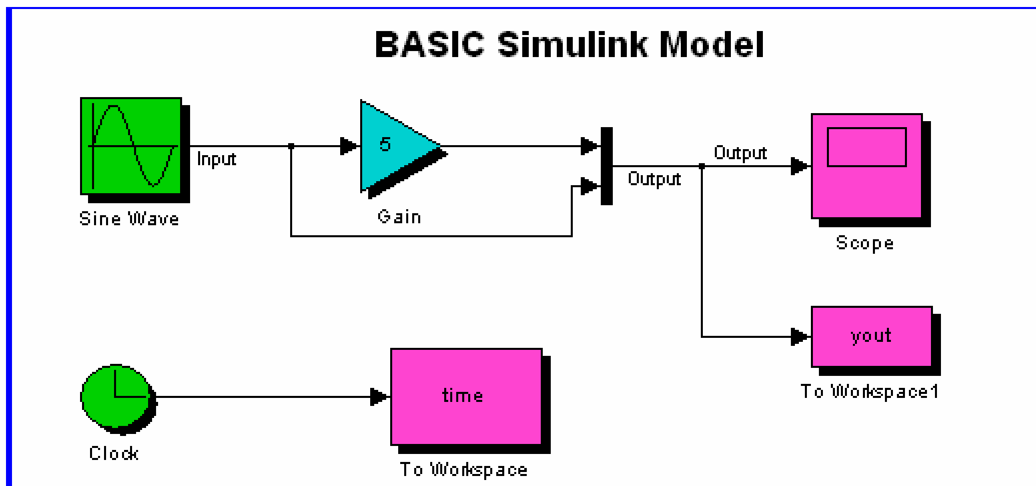

Figure 1, Simulink basic model

2.1 Simulink

Simulink settings consist of basic blocks with expected functionality. In the picture in Figure 1, Sine Wave generate an input signal for the model; Scope compares the input signal with the output; Workspace visualised the output on the workspace interface in Matlab and Gain process the input signal. Every signal has a name and is represented as a variable in the generated code. In case where the signal is unnamed or hidden, it is named after the block.

2.2 Model-Driven Engineering (MDE)

Sequel to the complexity in software development process, "Model Driven Engineering" (MDE) provides an alternative solution. MDE addresses this complexity by using models to represent abstractions levels and express domain concept appropriately. MDE is a development methodology and a technique that maximises productivity in embedded system development by increasing compatibility between systems. ''Modern-Driven Development'' (MDD), "Model-Driven Software Engineering" (MDSE) are synonyms to "Model Driven Engineering".

Figure 2, Model verification and checking [5]

However, MDE usage has a wider spectrum. In the software development process, models are used as a platform for communication among developers of the system and project stakeholders. Additionally, models provide an overview of large and complex system design through the usage of abstraction which aids in creation and verification of system design. Moreover, MDE tools are used in performing model checking that can detect and prevent errors in the development cycle [5].

2.3 Code Abstraction Gain Setting

Previously, the aim of code abstraction was to only reduce program size, but recent development has shown that code abstraction includes size reduction, as well as performance improvement. Sutter et al (2003) argued that code abstraction is an efficient technique in code size reduction especially C++ leading to code size reduction of approximately 35%. [8] Our main goal in this thesis work is to measure abstraction gain between generated code and manually written code of the same modelling language. We will not discuss in detail or technical specification of any abstraction technique. The tabular representations of abstraction gain are giving below;

Abstraction List	Original Source Code Form	Abstracted Source Code Form
1. Annotation	`/* annotation */, //annotation`	— *eliminate*
2. Variable declaration	`type var_name1 [, var_name2]*`	— *eliminate*
3. Format string	`"%(field-width) type-identifier"`	`""`
4. String argument	`"string"`	`""`
5. Expression statement	`variable = variable op (expr)`	— *eliminate* (some exceptions in function arguments)
6. *return* statement	`return variable`	— *eliminate*
7. Function call	`fn_name(argument_list)` `syscall_fn_name(argument_list)` `library_fn_name(argument_list)`	— *eliminate* `syscall_fn_name(argument_list)` `library_fn_name(argument_list)`
8. Conditional branch	`if (expr) ...` `if (expr) ... else ...` `switch (expr) ...`	`if ( ) ...` `if ( ) ... else ...` `switch ( ) ...`
9. Loop expression	`for (expr1; expr2; expr3) ...` `while (expr) ...` `do ... while(expr)`	`for ( ; ; ) ...` `while ( ) ...` `do ... while( )`

Figure 3, Abstraction list of original source code form and abstracted source code form

Origianl Source Code

```
1 /* Average of input digit */
2 #include <stdio.h>
3 int main()
4 {
5   int n, a[i], *ptr;
6   int i, sum;
7   printf("The number of input?");
8   scanf("%d", &n);
9   if (n>0) {
10    for (i=0; i<n; i++) {
11      printf("Input digit:");
12      scanf("%d", &a[i]);
13    }
14    ptr = &a[0];
15    for (i=0, sum=0; i<n; i++)
16      sum += *(ptr+i);
17    printf("Average=%d\n", sum/n);
18  }
19  else
20    printf("Try again.\n");
21  return 0;
22 }
```

Abstracted Source Code

```
1 #include <stdio.h>
2 int main()
3 {
4   printf("");
5   scanf("", (& n));
6   if (n>0)
7   {
8     for (;;)
9     {
10      printf("");
11      scanf("", (& a[i]));
12    }
13    for (;;)
14    {
15    }
16    printf("", (sum/n));
17  }
18  else
19  {
20    printf("");
21  }
22 }
```

Figure 4, Example of original source code and abstracted source code [2]

Code abstraction techniques consist of 3 levels [8] that are discussed below;

1. All multiple code fragments need to be detected.
2. Application of transformation such as register renaming, code rescheduling and parameterisation
3. When abstractable fragments are detected, they have to be abstracted.

2.4 Code Generating & Documentation

Generating code is a technique used in building high-quality machine generated code for multifaceted applications framework. It basically involves writing programs that write programs and errors that trail manual coding are completely eliminated. The output products build by these generators are reliable and maintainable. Code generation from model to code or code to model can be implemented with or without the linking, compilation, and processing that occurs as part of a full model build. Code generation is a process whereby some automated tools are used to turn a more abstract input into less abstract output [3]. Hebig (2014) argues that a benefit associated with code generating source code is reuse [6].

Code generation provides significant benefits to developers and engineers at all levels during embedded software development. The gains of code generation include the following: [4]

1. Consistency: Code are consistent and explicit variable naming which make the result easy to understand and clear.

2. Quality: Hand written code reduces code quality, with generated code the consistent level is high which increased quality of the output.

3. Reduced design time: Time scheduling for generated code significantly reduced compared to hand coding project. Also, it may require additional time working on adequate design and prototype testing to avoid downstream.

A comparison of generated code and hand-coding is illustrated in the table below: [4]

GENERATED CODE	**HAND-CODING**
Code quality is consistent across all the entities	Inconsistent code base that reduces quality across the entities
Mass changes require updating the templates with new code and the generator rerun	Any addition requires altering every entity
Bugs are fixed automatic by changes to the template and rerun	Bugs are fixed manually
Entity and schema classes are created simultaneously by the same mechanism, and synchronisation is automatic	Entity and schema are separate tasks and implemented in parallel

Once the code generation phase is complete, measures are taken to ensure proper documentation. The two methods used in documentation are:

1. Architectural documentation that will be used by the maintenance workers. This document should include the following: [4]

- The design goal of the generators.
- Advantage and disadvantage of the generator approach.
- The format of any input file.
- Installer building process.

- The purpose of all files linked with generators.
- The block architecture diagram for the generators.

2. End user documents which describes how the system is to be used:

- Installing the generators.
- Testing & running the generator.

In this project, Simulink Coder™ is used in generating code through the Target Language Compiler (TLC) - a library function. Unfortunately, we could not publish the source code from VCC in this report because of copyright laws. Below is an example of generated code using Simulink coder from Matlab:

```
40   /* Model step function */
41   void sf_in_for_each_step(void)
42   {
43     /* local block i/o variables */
44     real_T rtb_HiddenBuf_InsertedFor_Pulse;
45
46     /* local scratch DWork variables */
47     int32_T ForEach_itr;
48     real_T rtb_ForEachSubsystem_IterInp_0[6];
49     real_T rtb_ImpAsg_InsertedFor_Out1_at_[6];
50     int32_T i;
51
52     /* SignalConversion: '<Root>/For Each Subsystem_IterInp_0' incorporates:
53      *  Inport: '<Root>/In1'
54      */
55     for (i = 0; i < 6; i++) {
56       rtb_ForEachSubsystem_IterInp_0[i] = sf_in_for_each_U.In1[i];
57     }
58
59     /* Outputs for iterator SubSystem: '<Root>/For Each Subsystem' incorporates:
60      *  ForEach: '<S1>/For Each'
61      */
62     for (ForEach_itr = 0; ForEach_itr < 6; ForEach_itr++) {
63       /* SignalConversion: '<S1>/HiddenBuf_InsertedFor_PulseDividerChart_at_inport_0'
64        *  ForEachSliceSelector: '<S1>/ImpSel_InsertedFor_In1_at_outport_0'
65        */
66       rtb_HiddenBuf_InsertedFor_Pulse = rtb_ForEachSubsystem_IterInp_0[ForEach_itr];
67
68       /* Stateflow: '<S1>/PulseDividerChart' incorporates:
69        *  TriggerPort: '<S2>/clockIn'
70        */
71       PulseDivider(rtb_HiddenBuf_InsertedFor_Pulse,
72                    &sf_in_for_each_B.CoreSubsys[ForEach_itr].sf_PulseDividerChart,
73                    &sf_in_for_each_DWork.CoreSubsys[ForEach_itr].
74                    sf_PulseDividerChart,
75                    &sf_in_for_each_PrevZCSigState.CoreSubsys[ForEach_itr].
76                    sf_PulseDividerChart);
77
78       /* ForEachSliceAssignment: '<S1>/ImpAsg_InsertedFor_Out1_at_inport_0' */
79       rtb_ImpAsg_InsertedFor_Out1_at_[ForEach_itr] =
80         sf_in_for_each_B.CoreSubsys[ForEach_itr].sf_PulseDividerChart.ClockOut;
81     }
82
83     /* end of Outputs for SubSystem: '<Root>/For Each Subsystem' */
84
85     /* Outport: '<Root>/Out1' */
86     for (i = 0; i < 6; i++) {
87       sf_in_for_each_Y.Out1[i] = rtb_ImpAsg_InsertedFor_Out1_at_[i];
88     }
89   }
```

Figure 5, Sample copy of a generated code from Matlab

2.5 Model - Code Traceability

As a result of high cost of manual maintenance of traceability information during software development process, traceability is considered as a quality attribute [1]. The following definitions have been given to traceability:

Gotel & Finkelstein (1994) define traceability as the ability to monitor the life of requirement in both directions i.e., from origin, through its development and specification, to its subsequent deployment and usage [1].

Aizenbud-Reshef et al (2006) argue that, traceability is achieved by creating and maintaining correlation among artefacts involved in the software-engineering lifecycle during system development [12].
In addition, the IEEE Standard Glossary of Software Engineering Terminologies defines traceability as;

1. The degree to which each element in a software development product establishes its reason for existing.

2. The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate and design of a given software component match [1].

Code traceability in embedded software systems was previously driven by obligatory rules and directives, but recent changes in the software development industry have seen traceability as being recognised as a quality attribute and has witness a significant development in recent times. Example: the quality standard like IEEE Standard # 1219, ISO 9000ff and ISO 15504 now recommends code traceability in embedded software systems. Stakeholders also recommend the traceability relationship sequel to the fact that traceability aids in understanding the dependencies that subsist among software artefacts. Nevertheless, there is a wide-ranging deficiency of tools to support for automatic traceability analysis for embedded software systems. Wiederseiner et al (2011) argue that embedded systems companies need specific, yet flexible development and traceability tools. They also see traceability as having wider usage in the embedded industry, and enabling factor for better development, testing and maintenance of embedded systems [13].

In other to close this short fall in automatic traceability tools, MDD has created a platform for discovery of traceability relationships. Organisations have also channelled enormous resources in finding a solution to the traceability problem. For example, there is Automated Embedded Traceability Framework (AutoETF) tool set that can automatically derive and visualise traceability links between source code and test code artefacts in embedded software [13].

Traceability can be from model to code or code to model depending on the objective. In this project, we are focusing on automatic traceability from model to code and vice versa. The two traceability concepts are illustrated in the diagrams below. These traceability diagrams are possible to be generated by the feature in Simulink called Code Inspector. However, Code Inspector was implemented for the first time in the Matlab version 2014a and is available also in versions 2015 and 2016. Also, our models provided by the Volvo Car Corporation (VCC) are executable only in the version 2013b, due to the fact of compliance with the "AUTOSAR" standard. Hence, it is impossible to implement code to model traceability in this report.

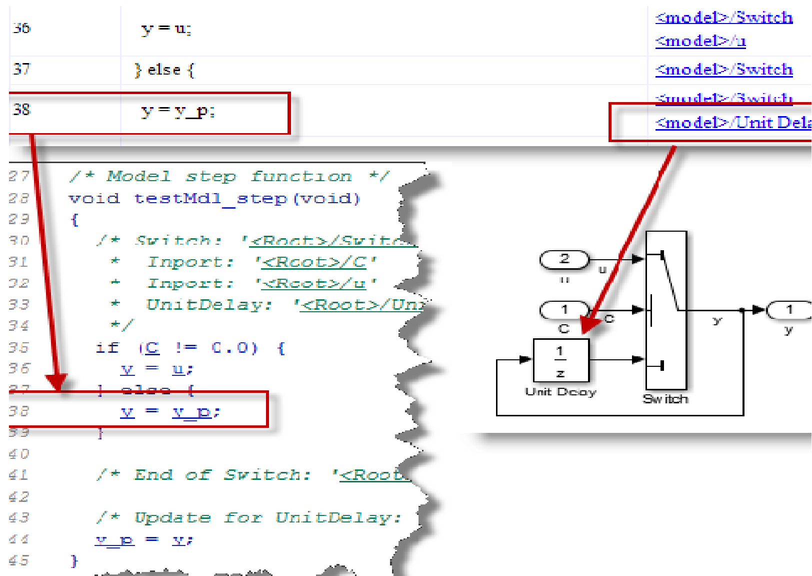

Figure 6, Code to Model Traceability [2]

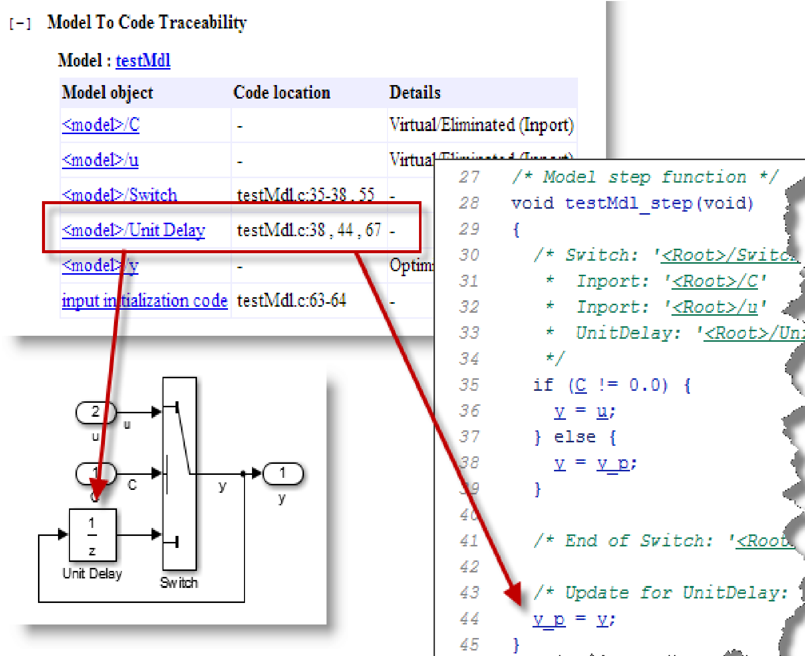

Figure 7, Model to Code Traceability [2]

3. METHODOLOGY

3.1 Pre-Study

Our initial task was to affiliate the type of a particular block with how many lines of code this particular type generates. There are two ways to achieve that. Starting from the model, and then proceed to the generated code or vice versa.

A. Model-> Generated code

We found three different ways to count the amount of blocks within a model as well as their type.

1. **find_system('parameter',' value'):** Is an API function. It accepts sets of parameters and values. If we set 'Blocktype' in the parameter and 'Goto' in the value, it returns all the 'Goto' blocks within our model independently from the layer that they are found, including their paths within the model. Similarly, we can search for all the 'Blocktypes' within our model. This API returns an object in the workspace interface of Matlab which consists of an nx1 table where n is the amount of blocks of that particular type found.

2. **sldiagnostics('sys', 'options'):** Is an API function. It accepts the name of a particular model and a set of different options to assess the properties of the particular model. If we set 'CountBlocks' in the options the API will count all the blocks within the particular model as well as their types and it will return a report in the workspace interface of Matlab which is an object and consist of an 1x1 table, containing the text with all the information about the blocks of our model.

3. **Class (slmetric.Engine, Package):** Is a class that generates model metric data for the specified metric engine object. To generate metric data for specific metrics, we use execute (slmetric_obj, MetricIDs) where; slmetric_obj is our metric engine object and MetricIDs the metric identifier. The main advantage of this method is that we can extract the data in an xml file. The disadvantage is, it is the only method that does not return the 'Blocktypes' of the blocks. It only counts the total blocks within the model.

Nevertheless, all those three different methods stop there. There is no way to affiliate the blocks or 'Blocktypes' counted with generated code.

B. Generated code-> Model

We tried connecting the generated code with the Blocktype. The generated code is a C file which comprises of generated code and comments. The comments "wrap" the generated code and they make reference to tags of the "<SXX>/Blockname" format. XX is a number assigned by the embedded coder tool which makes the code generation. 'Blocktype' is the name the developers named the specific block. Those tags serve as the glue between generated code and model. Those tags can be used within the *hilite_system*(block_path) API and it highlights the block with the particular tag. The concept here is, since those tags are included within the comments wrapping the generated code, that means the generated code is designated for those particular tags. The problem though is that *hilite_system* only highlights a block optically but cannot be used to retrieve a link/id to the block. It does not return an object. We cannot use it to get the parameters of the block, in particular, the 'Blocktype'. Hence, we could not affiliate lines of generated code with 'Blocktype'.

The next step was identifying a pattern in the pathing of the tool. In this way, we could possibly predict the path of the block by its “<SXX>/Blockname” tag. However, we need to mention that tag and block path are different concepts. Block path is the path within our model of the type: model/submodel1/submodel11/block. The only pattern we found was that blocks with same <SXX> tag belong in the same layer. Why layers are given a particular <SXX> number is hidden in the Matlab/Simulink internal algorithms. The reason we wanted to identify a pattern is that we wanted to make use of the get_param(block_path, Value) API, which given a block path returns its ‘Blocktype’ if we set as ‘Blocktype’ in the value argument.

Another approach was to create our own model. We implement this by importing a block of a particular type at a time and simulate the model in order to generate a C file. With this concept, we find out how many lines of code each ‘Blocktype’ generates. After proceeding, it became obvious after the first time generating, that the model for an individual ‘Blocktype’, that the C generated file had not the same format as the C generated files obtained by Volvo Cars Corporation. We ensured that all the model configuration parameters matched the configuration parameters of the Volvo Cars Corporation’s models. Nevertheless, in our C generated file it was impossible to connect our Block with generated lines of code. As a result, we concluded that in order to affiliate a Block with generated code, it needs to be part of a complete system, including signals (the connecting lines between blocks) and variable inputs in the blocks.

We also contacted the Matlab support to ask if they are aware of any way generated code can be traced to Blocktype with the use of 2013b Version, and they were unaware of such an API. The representative told us, we can only do it manually and not programmatically and he advised us to go to the generated code settings and change the format of the tags hoping it will make the tracing easier. That means we need to regenerate all models and changing the format of the tags does not provide any functionality. The only thing that would influence would be only our parser.

3.2 Empirical Research Method

Firstly, we tried manually to trace every tag in the generated code back to the model through the *hilite_system* API. Eventually, we abandoned this method but in the process, we discovered interesting findings that are mentioned in the results section and are relevant to the analysis of our system.

We concluded that to affiliate ‘Blocktypes’ with lines of code is doable through parsing the C generated file. By parsing the C generated file for a particular model, due to its structure it is possible to connect some ‘Blocktypes’ from the block tags which are included in the comments, with the lines of code that are wrapped by the particular comments. For this purpose, we developed a tool that parses the C generated file which extracts the data we are interested at.

3.3 Validity Discussion

Internal validity

Due to the fact that our tool is a text parser, it is designed to parse a particular text structure. We are examining 22 models, each generating a C text file of 2000 lines of code. 90% of the text is structured in the format "comment-generated code-closing comment". We eliminated the closing comment, so as to allow our parser in looking for the structure "comment-generated code" to make connections between 'Blocktypes' and lines of code generated for the particular 'Blocktype'. Sequel to the diversity of models and each model generates 2000 lines of code. These also include comments, declarations and importation of libraries, it is possible that the C file at some point may not follow the "comment-generated code" structure. Thus, our parser is unable to recognise a pattern and connect the 'Blocktype' with its dedicated generated code. An example is the "Gain Blocktype". After examining the C text file, we notice that the tag "Gain :<Snumber>/ BlockName" was met under the following text structure: "Comment (includes the Gain tag)- generated code-comment within the generated code (Including other tags)- continuation of the generated code". In this case, the parser recognises the text structure as two groups of "Comment-generated code". Counting correctly the generated code for the tags in the comment which is "wrapped" within generated code, but for the "Gain Blocktype", it counts only the generated code above the "wrapped" comment.

Conclusion validity

A threat for the validity of this research is the fact that we applied our tool to a particular amount of models (22 models). The sample is satisfying to make conclusions, but with a larger data set our conclusions would be even more valid. Another threat to validity, concerns counting the lines of generated code as a measure of abstraction gain. We are convinced that it is one of the right ways, sequel to the fact that we were restricted in using Matlab version 2013b because of the AUTOSAR architecture standard (it is not compatible with other versions, and it was essential in order to be able to view the models).

External validity

The research was based upon models provided by Volvo Cars Corporation. The results hold for the models of VCC. Nevertheless, there is threat that the results may alter from models from different companies, even by using the same modelling language (Simulink). Therefore, we cannot generalise the result outside the scope of this study.

Construct validity

Another significant threat to validity is that there are not many ways to trace blocks to lines of code. Thus, this may influence our results, since we cannot validate whether our results hold with the use of some other method. There are blocks that cannot be traced at all. So our focus is on the blocks we can easily trace. This will also affect the validity of this result since we have not found any relation between traceable blocks and non-traceable blocks.

4. PROTOTYPE

4.1 Building Prototype

The tool for measuring our data is developed externally, independently of the Matlab framework. The tool is developed in C# and serves as a parser for the generated code file produced by Matlab, when models are simulated. The parser is designed specifically to fit the format of the generated code file given by Volvo Cars Corporation generated in Simulink, thus any other file with a different format will not bring any results. Thus, it is important to always use the same “Symbols” (format of tags and generated code), as the “Symbols” the parser is designed on, in the model configuration parameters, before simulating a model.

The tool initiates by reading all the lines of our C file. With the use of Regular Expressions, we pattern match and group the comments with their dedicated generated code. The comments include the tags of the blocks for which the following code has been generated for. The tag for a given block, follows the format: Blocktype: Tag/BlockName (ex. Constant: <S22>/ Constant1). Blocktype is the type of the block and it is assigned automatically by Simulink. Tag is the letter S followed by a number. This combination refers in which layer(subsystem) our block is placed within the model. All blocks in the same layer share the same <Snumber>. ‘BlockName’ is the name of the block given by the developers who designed the model. It is important not to confuse the tag of a block with the variable Tag used within our tool which refers to the <Snumber> group. The tag of a block is the combination of Tag/BlockName. In order to trace a block with the *hilite_system* API provided by Matlab, it is important to provide the Tag/BlockName parameters. The ‘BlockType’ is just a convenient way to identify the ‘Blocktype’ of our block, provided by the Simulink simulator.

With the help of Regular Expressions, we pattern match and extract the ‘BlockType’, ‘Tag’, and ‘BlockName’ from the comments and insert them in a List structure. Due to the fact that we have grouped comments with their dedicated generated code, we insert in a new List structure, the tag (‘BlockType’, ‘Tag’, ‘BlockName’) and the “generated code” string. We split the “generated code” string and count the lines of code.

By the use of LINQ, which is a feature unique in C# we query our List and extract the data that we are interested in: ‘BlockType’, how many times encountered in our model the particular ‘BlockType’, the minimum lines of code, the maximum lines of code and the average lines of code.

Finally, we export our data in a csv file. The data are also visible in the Console Panel.

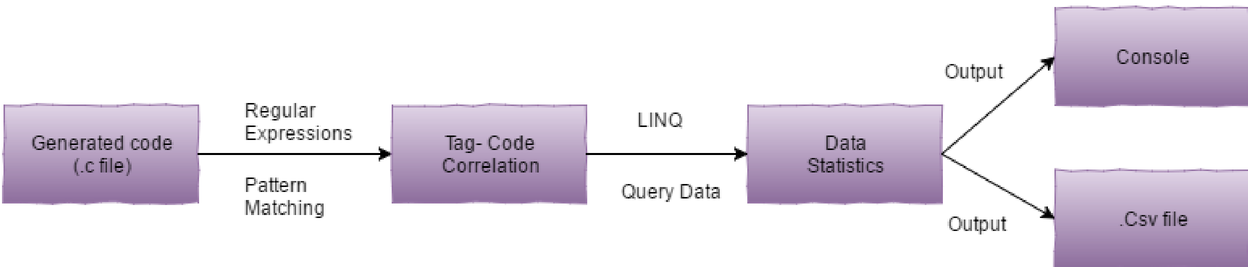

Figure 8, Prototype in diagram

4.2 Applying Prototype

When we apply our tool, the parameters which are subject to change, are the path of the text file we are parsing, as well as the path on where our csv file is going to be created. In addition, we can change our queries to acquire different data. For example, we can acquire data per 'BlockName' instead of 'BlockType'. This differentiation is of great importance, since 'BlockName' gives us different insight as explained below in the results section.

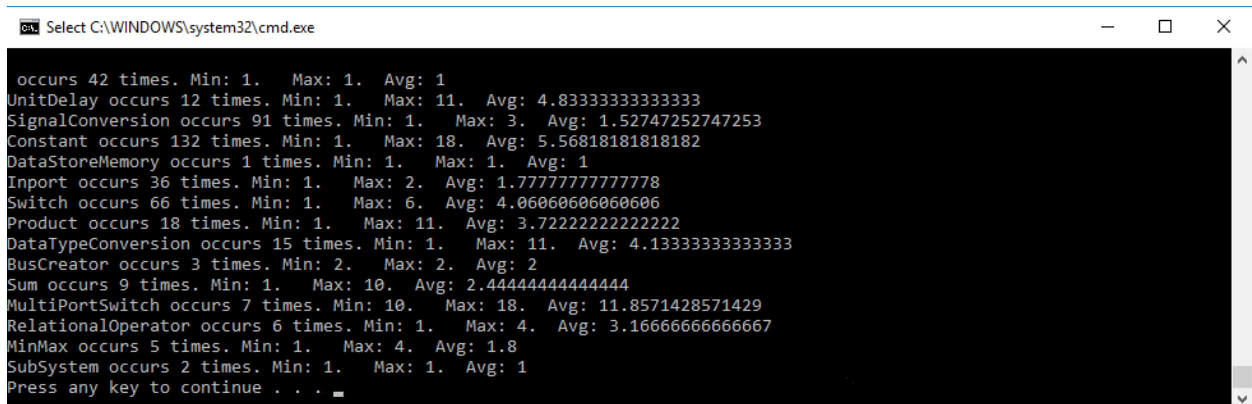

Figure 9, Prototyping application showing result from the model

Figure 9, represents a screenshot with the results from a model. We clearly see all the statistics of our data. In the first line, there is no 'BlockType'. This line refers to all the blocks that belong to the format <Snumber>/ BlockName, which have not been given a distinctive 'BlockType' by the simulator.

5. STUDY'S RESULTS

5.1 Traceability

In the generated C file, there are blocks named by the developers as db (debug) and sw (switch). Those blocks cannot be traced back to the model. By tracing their layer through the <Snumber> tag, we were able to ascend one layer up and identify the block to which those belong within the model. The block they belong to is classified as a "Masked" block. Masked blocks in Simulink are called the blocks that are custom made by the developer of the model. In the description of this block, it was clear that this block is constructed in order to call external software called INCA. This software serves testing and validation purposes, in our case for testing and debugging the switches within our system. Due to the fact we lack this software (it is a commercial, non-open-source software), it is safe to assume that every time the model is executed, this software creates those extra blocks we could not trace in our system.

Another aspect we noticed through the manual attempt of tracing lines of code to Blocktype, is that, except the tag <Snumber> for the layers, we met the tag <Root>. This tag refers to the BlockType "SubSystem" which is a block encountered usually at the first layer of the model and is connected through signals with all the inputs and outputs of our model. Nevertheless, it can be encountered further in the model but it is like a "wrapping" block.

5.2 Results from tool

Below follows a summary of all 'Blocktypes' in our 22 models. We enlist how many times the 'Blocktypes' were encountered in all the models, as well as the Minimum, Maximum and finally Average lines of code per 'Blocktype' in all the models.

BlockType	Occurrence	Minimum LOC	Maximum LOC	LOC Avg
No_Type	499	1	34	1.28
BusCreator	12	2	8	2.71
Sum	180	1	21	3.88
MultiPortSwitch	23	0	41	9.51
RelationalOperator	394	1	23	3.71
MinMax	233	0	9	3.08
SubSystem	46	1	14	1.18
DataStoreMemory	24	1	2	1.21
SignalConversion	237	0	9	1.72
Constant	1618	1	41	4.37
Product	174	1	21	4.61
Inport	128	1	4	2.65
Switch	1115	0	34	3.53
UnitDelay	339	0	23	2.37
DataStoreRead	70	1	10	3.67
EnablePort	2	1	1	1
ArithShift	38	2	34	9.8
Chart	2	1	11	6
Logic	382	1	16	3.83
BusSelector	9	1	23	8.9
DataTypeConversion	47	1	11	4.04
D	32	1	11	4.36

DiscreteIntegrator	26	1	9	2.26
ActionPort	3	5	6	5.6
MultiPortSwitch	23	0	41	10.97
Saturated	4	0	1	0.5
SwitchCase	1	2	2	2

In the table above, we notice that some ‘Blocktypes’ have generated 0 lines of code (LOC). As we explained in the threats to validity above, this may be a result of the parser not recognising the pattern in the text file. The “Gain Blocktype” is not included in the table above because of the reason explained in the validity threat section.

With our parser, we can change the queries in our lists and obtain data per ‘Blockname’ instead of ‘Blocktype’. In this way, we can obtain information and gain insight on blocks such as the db(debug) and sw(switch). This is of great importance, since we can count how many blocks created by the developers that are encountered within the C generated file.

Below there are two tables with all the permitted used ‘Blocktypes’. They are divided according to encountered in the C generated code file or not. Sequel to the fact that those are not encountered in the generated code file, they generate no code.

Block_type with no gen_code

Block_Type (18)	
S-Function	Math
Demux	Assignment
From	DataTypeDuplicate
Goto	TriggerPort
Gain	SimpleTimeMath
Ground	
If	
Inport	
Mux	
Quantizer	
Terminator	
InitialCondition	

Block_Type with gen_code

BlockType (27)	
No_Type	Switch
BusCreator	UnitDelay
Sum	DataStoreRead
MultiPortSwitch	EnablePort
RelationalOpeartor	ArithShift
MinMax	Chart
SubSystem	Logic
DataStoreMemory	BusSelector
SignalConversion	DataTypeCoversion
Constant	D
Product	DiscreteIntegrator
Inport	ActionPort
Switch	MultiPortSwitch
UnitDelay	Saturated
DataStoreRead	SwitchCase

6. DISCUSSION

The aim of this study is to identify static measure of abstraction gain based on code and model artefacts. While the research was ongoing, it became obvious that there is no possibility in Simulink framework to trace Blocks to lines of code generated because of lack of an API developed for that purpose. Our effort to trace tags from the generated code, back to the model with the existing API, resulted in realising how restricted is the functionality provided by that API.

Since the tracing between generated code and model was impossible, the nearest way to have results was by parsing the generated code file and by affiliating tags in the comments with generated code. This technique is trustworthy, since the generated code file is consistent and "accidents" such as a missing break line do not occur. Thus the estimated abstraction gain will always be valid.

Based on our results we notice that 27 'Blocktypes' participate in generating code whereas 18 'Blocktypes' do not generate code at all. This is because the 'Blocktypes' that generate code provide the functionality of the system whereas the 'Blocktypes' that do not generate code are auxiliary and do not provide functionality in the system.

It is also noticeable that the average lines of generated code from the various 'Blocktypes' range from 1 to 10.97. In the cases, that 'Blocktypes' generate approximately 1 LOC, we have no abstraction gain. That means the developer will put the same work effort to drag and drop 1 block as to write 1 LOC. In the cases that 'Blocktypes' generate 2 or more LOC, then we definitely have abstraction gain and the modelling procedure saves work effort to the developers. Nevertheless, we are unable to reason whether the abstraction gain is worthwhile, since there is no literature defining when abstraction gain is significant.

In addition, our first research question was "how to statically measure abstraction gain based on the code and model artefacts?" We therefore consider its practical usage. Our reasoning is that after we measure the abstraction gain practically, we can compare the work effort the developer would put if he/she wrote the source code instead of drop and dragging a block.

Furthermore, reflecting further on research question 1 by "statically" measuring abstraction gain, our aim is to measure the relationship between model blocks and lines of code in the generated C file independently from any other interactions within the model. In addition, we consider how the experimental setting will look like if we had used "dynamically measuring abstraction gain" in research question 1. Considering the trade-off between "statically" – analysing the artefact and "dynamically"- information execution, static analysis of the system is cheap and gives early results, while the dynamic analysis might enable us to get information that we cannot get statically. If we use "dynamically" set up in the execution, this could help us to gain additional or more detailed information example; better traces, and information about the signals. However, dynamic analysis would be infeasible (too expensive) since it would require a manipulation of the code generator and a complete regeneration of the code with correct parameterisation information and setup is very expensive.

In answering our Research Questions:

1. How can we statically measure an abstraction gain based on the model and code artefacts?

We can statically measure an abstraction gain by connecting tags and their dedicated generated code by a parsing tool based on the C generated file. However, the model is left out of the equation.

1.1. To what extend can elements from the models be traced to corresponding parts of the generated code?

In the Matlab version 2013be are researching on, it is impossible to trace elements from models to corresponding parts of the generated code. We can though trace generated code back to model with the existing API, but mostly optically, without further information for the block's path.

6.1 FUTURE WORK

Sequel to the delayed response from VCC developers, it is currently impossible to answer our Research Question 2: Is the abstraction gain in the case study at hand considered worthwhile by the developers? So we leave open the second Research Question, to be answered by future researchers.

In addition, more APIs are needed to be developed from Matlab for version 2013b, to provide more options and possibilities to the users of Simulink, as well as the researchers who aim on better understanding Modelling languages and explore their benefits and capabilities.

7. CONCLUSION

In this research work, we have been able to show that lines of code (LOC) to block traceability within Simulink framework is impossible with 2013b version. Traceability (code to model or model to code) is possible in Simulink 'Code inspector' which is available only in version R2015b (Version 2.4), R2014b (Version 2.3), R2014b (Version 2.2) and R2014a (Version 2.1).

The level of abstraction gain has raised yet another debate. Unfortunately, we will not reflect on how high the abstraction level should be before we conclude that there is gain. Also, we could not ascertain if there is abstraction gain by developers, since we have not collected the response from Volvo representative as at when this report was published. From the results we discussed in chapter 5, twenty-two (22) models were used in this research work to evaluate the average LOC. Also, we suggest in the future research; effort should be made to unveil if code generation and high abstraction gain improve the quality of software produced.

8. ACKNOWLEDGEMENTS

We would like to thank our supervisor Regina Hebig for her constant feedback and for directing us during the course of our research. We would also like to thank Jonn Lantz from Volvo Cars Corporation (VCC), for providing to us the models to conduct our thesis on.

9. REFERENCES

1. O. Gotel and A. Finkelstein, "An Analysis of the Requirements Traceability Problem," Proceedings of the First International Conference on Requirements Engineering, Utrecht, The Netherlands (1994), Publisher: IEEE. pp. 94-101. ISBN: 0-8186-5480-5

2. http://blogs.mathworks.com/simulink/2014/07/30/what-can-simulink-code-inspector-do-for-you/

3. http://c2.com/cgi/wiki?CodeGeneration

4. J. Herrington. "Code Generation in Action". Publisher: Manning Publications Co. Greenwich, CT, USA. July 2003. ISBN 1930110979. pp 368.

5. D C. Schmidt. "Model-Driven Engineering". Vanderbilt University (2006). IEEE Computer, February 2006 (Vol.39, No.2) pp 25-31

6. Hebig, Regina, "Thesis: Evolution of Model-Driven Engineering Setting in Practice" Hasso Plattner Institute for Software Engineering System Analysis & Modelling Group. June 20, 2014.

7. http://www.slideshare.net/mbrambil/model-driven-development-and-code-generation-of-software-systems

8. B. Sutter, H. Vandierendonck, B. Bus and K. Bosschere. "On the Side-Effects of Code Abstraction". Electronics and Information System (ELIS) Department, Ghent University, Sint-Pietersnieuwstraat 41, Belgium. LCTES. Publisher: ACM NY, USA @ 2003. pp 244-253.

9. S. Park, S. Ko, J. Choi, H. Han, S. Cho and J. Choi. " Detecting Source Code Similarity Using Code Abstraction". Published in ICUIMC '13 Proceedings for the 7th International Conference on Ubiquitous Information Management and communication. Article No. 74, NY USA @2013, pp53-69. ISBN: 978-1-4503-1958-4. Vol. 38 Issue 7.

10. J. Hutchinson, M. Rouncefield, and J. Whittle. "Model_Driven Engineering Practices in Industry". 2011 33rd International Conference on Software Engineering (ICSE), May 2011., Honolulu, USA. Pp 633- 642. E-ISBN 978-1-4503-0445-0. Publisher IEEE.

11. M. Broy "Challenges in Automotive Engineering". Institute of Information Technology University of Munchen. Published in: Proceeding ICSE '06 Proceedings of the 28th international conference on Software engineering. ACM NY, USA @2006. pp 33-42. ISBN : 1-59593-375-1.

12. N. Aizenbud_Reshef, B. Nolan, J. Rubin, and Y. Shaham-Gafni. "Model Traceability". IBM SYSTEMS Journals. Vol 45, No. 3 , 2006. pp515-526. ISSN: 0018-8670

13. C. Wiederseiner, V. Garousi, and M. Smith., "Tool support for automated traceability of test/code artifacts in embedded software systems." *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*. IEEE, 2011. pp1109- 1117, ISSN: 2324-898X