Change Impact Analysis in Simulink Designs of Embedded Systems

By

Bennett Mackenzie, B.Eng.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfilment of the Requirements for the Degree

Master of Applied Science

McMaster University

Abstract

This thesis presents the *Boundary Diagram Tool*, a tool for change impact analysis of large Simulink designs of embedded systems. The Boundary Diagram Tool extends the Reach/Coreach Tool, an existing tool for model slicing within a single Simulink model, to trace the impact of model changes through multiple Simulink models and to network interfaces of an automotive controller. While the change impact analysis results can be viewed directly within the Simulink models, the tool also uses various block diagrams to represent the impact analysis results with different levels of abstraction, motivated by industrial needs. In order to effectively present the complex impact analysis results, various techniques for visual representation of large graphs are employed. Furthermore, the Reach/Coreach Tool as an underlying model slicing engine was significantly improved. The Boundary Diagram Tool is currently being integrated into the software development process of a large automotive OEM (Original Equipment Manufacturer). It provides support during several phases of the change management process: change request analysis and evaluation, as well as the implementation, verification and integration of software changes. The tool also aids impact analyses required for compliance with functional safety standards such as ISO 26262.

Acknowledgments

Thank you very much to my supervisors Dr. Mark Lawford and Dr. Alan Wassyng. They provided me with wonderful opportunities for undergraduate and graduate research as well as guided me throughout both my studies and the completion of this thesis.

Thank you very much to Vera Pantelic, whose guidance has been invaluable not only throughout my research endeavours, but also throughout the process of writing this thesis.

Thank you to my colleagues for making the years I spent doing research very enjoyable ones.

Thanks to all of my family and friends for their support throughout my life and academic career.

This wouldn't have been possible without all of you.

Contents

List of Figures

Chapter 1

Introduction

This chapter introduces the motivation and objective of the work presented in this thesis, in Section 1.1. The approach taken to accomplish the objective follows in Section 1.2. The main contributions of the thesis are discussed in Section 1.3. Finally, the overall structure of this thesis is defined in Section 1.4.

1.1 Motivation

Complexity of embedded software in industry is rapidly increasing. In the automotive industry, software implemented in modern cars consists of over 100M lines of code (Charette 2009). Given that software maintenance is the most resource-consuming phase of the software development life cycle (Bennett 1990), maintaining such large code bases of modern automotive controllers requires significant resources.

Change impact analysis can be defined as "identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change" (Bohner 1996). Change impact analysis can be used in analysis, evalu-

ation, implementation, and verification of changes in software (Bohner 1996). For example, in the analysis and evaluation of a proposed software change, impact analysis could be used to identify other components of the software that are likely undergo changes to support a given proposed software change. This information can be used to plan the release schedule according to the estimated time needed to implement a proposed change. In the implementation phase, the results of an impact analysis on the changed source code can be used to identify other software components that need to be changed in order to accommodate the implemented change. When verifying an implemented change, an impact analysis could be used to focus regression testing efforts on changed parts of software. Further, safety-critical systems standards such as ISO 26262 (ISO 2011b) and IEC 61508 (IEC 2010) mandate that an impact analysis is performed for any requested change in the software maintenance process of safety-critical systems. Inadequate change impact analysis of software systems was found to be a cause of accidents (de la Vara et al. 2016).

Manually performed impact analyses tend to be unreliable. Case studies show that developers tend to heavily underestimate the impact of software changes (Lindvall 1997). Therefore, it is very important to have proper tool support for change impact analyses in modern embedded systems.

In recent years, the automotive industry has trended towards utilizing the *Model-Driven Development* paradigm for developing embedded software, with Matlab/Simulink being commonly employed throughout the industry as the main platform supporting such development. Rather than directly writing C code, automotive controls developers now create Simulink models that are used to generate C code. As the level of abstraction has risen from source code to Simulink models, the software maintenance efforts has shifted from

source code maintenance to Simulink models maintenance. While tools for performing change impact analysis on code written in traditional programming languages exist (Galbo 2017), the current lack of corresponding tool-supported approaches to impact analysis on Simulink models presents an opportunity for the development of a new tool that would fill the gap.

Therefore in order to properly maintain Simulink designs of complex, production-scale embedded systems, proper tool support is required for performing change impact analysis on Simulink models. Proper tool support will (i) reduce the required resources and, consequently costs of maintenance, and (ii) decrease the incidence of software-related accidents in the automotive industry. In particular, the need for the tool developed in this thesis arose from our collaboration with a large automotive Original Equipment Manufacturer (OEM): a tool was needed for impact analysis within the Simulink design of the OEM's electrified powertrain supervisory controller. The application layer of the controller consists of dozens of Simulink models, with the largest models containing hundreds of thousands of blocks and a few tens of thousands of subsystems.

1.2 Approach

The tool described in this thesis, the *Boundary Diagram Tool (BDT)*, leverages the previously built *Reach/Coreach Tool* (Pantelic et al. 2018). The Reach/Coreach Tool is a tool for *model slicing* Simulink models, a process that involves extracting a functional subset of the blocks in a Simulink model based on an impact analysis of an initial selection. The tool traces through control and data dependencies within a *single* Simulink model while preserving

its hierarchical structure. More precisely, by tracing through the data and control dependencies from a selected set of blocks and/or signal lines in a Simulink model, the Reach/Coreach Tool identifies the affected blocks and lines of the model. This functionality is called either *Reach analysis* or *forward tracing*. Similarly, the Reach/Coreach Tool can identify the blocks and lines of the model which affect the specified blocks and lines. This functionality is referred to as either *Coreach analysis* or *backward tracing*. The Reach/Coreach Tool can be used in the design and static analysis of Simulink models (e.g., finding unreachable parts of a model), as well as in impact analysis within a single Simulink model. Other engines have been created for slicing of Simulink models (Reicherdt and Glesner 2012; MathWorks 2018a; Rapos and Cordy 2017; Kowalewski 2016), but the Reach/Coreach Tool was chosen due to its maturity, familiarity and availability (the tool is open source and non-commercial). A detailed comparison of related tools is provide later in Section 2.4.

The Boundary Diagram Tool builds upon the engine provided by the Reach/Coreach Tool, and extends it to an industrial strength change impact analysis tool that can track impact across numerous large scale models as well as to/from network interfaces of the application portion of the OEM's supervisory controller system for the powertrain of a hybrid electric vehicle. The tool has the ability to track both intra- and inter-model dependencies within the Simulink design of the controller. To the best of the author's knowledge, this is the first tool for impact analysis within Simulink designs of embedded systems. MathWorks' *Impact Analysis* tool (MathWorks 2018b) is very different than the BDT as it tracks only build dependencies between files in Matlab/Simulink implementations. It does not track connections between Simulink models that are specified outside of the modelling environment (e.g.,

via dependency matrices), as is often the case in industry. Also, it does not perform the fine-granularity intra-model analysis provided by the BDT via the Reach/Coreach Tool (Pantelic et al. 2018).

The BDT was specifically built to perform change impact analysis for the OEM's electrified powertrain supervisory controller. The connections between the models within the controller are specified using a dependency matrix. The BDT was developed with the information hiding principle in mind to encapsulate the encoding of the interconnections, so that the tool can then be easily adapted to other embedded systems implemented in Simulink that use a different approach/format to specify the inter-model connections. Recently there has been a push to standardize interfaces of software components and their interconnections within electronic control units and even between processors over network connections via the emerging automotive industry middleware standard called AUTOSAR (Fürst et al. 2009). The BDT has been designed so that it can be easily adapted to support the specification of inter-model connections between the software components that are described in AUTOSAR or any other standardized format.

The BDT is built to help our industry partner adhere to ISO 26262, the functional safety standard for road vehicles developed by the International Standard Organization (ISO 2011b). The standard covers product development at the system, hardware, and software level, as well as required supporting processes. ISO 26262 section 8 requires an impact analysis to be performed for change requests to the system's software in order to comply with the standard.

The BDT presents change impact analysis results using various diagrams. Aside from marking the impacted blocks and lines within Simulink models, the tool presents change impact using non-Simulink block diagrams. The

depiction of change impact analysis results using block diagrams of various abstraction levels was inspired by *boundary diagrams*. Boundary diagrams are often used in systems and safety engineering to present the components of a system, relationships between the components, as well as system's interface to external entities (Lindland 2007). They are typically used as a design tool that supports a hazard analysis technique called FMEA (Failure Mode and Effect Analysis). The generation of this kind of diagram is standard in mechanical and electrical designs of systems for our industry partner. As such one of the initial motivations of this tool was to be able to provide a similar view for software systems, however the classical boundary diagram didn't tell the entire story for software systems. Boundary diagrams focus on representing the interactions of systems and/or their components. While the diagrams generated by the tool include boundary diagrams, the tool also produces several other diagrams, most notably context diagrams, as well as slices of context and boundary diagrams showing only impacted components and dependencies identified by impact analysis. By a slight abuse of terminology, this thesis will henceforth refer to diagrams generated by the tool as boundary diagrams.

Due to the large number of Simulink models in the controller's implementation as well as high degree of coupling between them, the BDT-generated diagrams were found to be too complex to effectively represent the results of impact analysis. Therefore, techniques for visualization of large graphs have been employed, in particular, interactive exploration and unwinding of feedback loops to allow developers to understand and navigate impact analysis results (Herman, Melançon, and Marshall 2000) have been effective.

1.3 Contributions

The main contributions of this thesis are as follows:

1. To the best of the author's knowledge, this is the first tool for comprehensive change impact analysis within Simulink designs of embedded systems. The BDT allows a developer to perform change impact analysis on a production-scale embedded controller. The tool aids in analysis and evaluation, implementation, verification and integration of software change requests, and helps with compliance with modern software safety standards such as ISO 26262 (ISO 2011b).

2. The underlying intra-model slicing engine, the Reach/Coreach Tool, has itself undergone major modifications resulting in significant performance improvements, when applied to a large model. Such performance improvements were necessary in order to leverage the intra-model slicing tool to build a practical inter-model slicing tool for embedded controllers with a large number of models and complex interactions between them.

3. The tool presents change impact analysis results using various diagrams offering different abstractions of the results, motivated by the tool's application in an OEM's change management process. In order to depict dependencies within the complex system of the industrial controller, techniques such as interactive exploration and loop unwinding for visualizing large graphs are employed. While the techniques are not new, their application to diagrams generated by the BDT to navigate and document complex impact analysis results is novel.

4. The BDT's applications in the model-based change management process

were identified, including change request analysis in compliance with functional safety standards, as well as design, implementation, verification and integration of software changes.

1.4 Thesis Structure

This thesis will be broken down into five sections. First, preliminary background information relevant to the thesis will be discussed. Second, a section that will detail the tool and how the tool performs the impact analysis. Third, the boundary diagrams generated by the tool and the views they represent will be explained. Fourth, a section about how the tool fits into the change management process. And finally, the last section will contain conclusions and future work.

Chapter 2

Preliminaries

The material in this chapter describes the background for the research presented in this thesis. First, Section 2.1 introduces the Matlab/Simulink environment. Then, Section 2.2 describes the Reach/Coreach Tool used as a model slicing engine for the BDT. Finally, several existing impact analysis tools for Simulink are discussed and compared to the BDT in Section 2.4.

2.1 Matlab and Simulink

Matlab is a programming and numerical computing platform developed by MathWorks that is designed for use by scientists and engineers. The Matlab language programming language is a weakly typed programming language built with an emphasis on fast and precise matrix computations for scientific applications. In addition to its features involving matrix computations, Matlab has a suite of features for various scientific and engineering applications. This includes control systems, machine learning, parallel computation, math and statistics, image processing, and more. The MathWorks has additionally

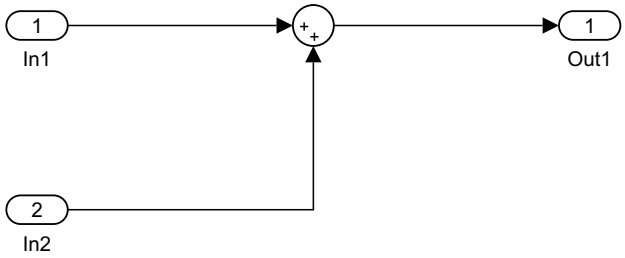


Figure 2.1: A very simple Simulink model

released a model-based design platform built upon Matlab called Simulink.

Simulink is an environment used to create, simulate, and analyze *models* of dynamic systems. These models are built through a GUI (Graphical User Interface) using the Simulink graphical programming language. The models are composed of two types of objects: blocks and signal lines. A block in Simulink performs an operation, and signals represent data flow between blocks. Blocks and signal lines are connected via the *ports* attached to Simulink blocks, which indicate where signal lines are to be inputted to the block and where signal lines are to be outputted from the block.

The example shown in Figure 2.1 represents a simple Simulink block diagram. The blocks labelled In1 and In2 on the left provide inputs to the model, and are called Inport blocks. The block in the middle is called a Sum block, that adds two signals. Finally, the block labelled Out1 on the right is a called an Outport block.

2.2 Reach/Coreach Tool

The *Reach/Coreach Tool* is a model-slicing tool (Pantelic et al. 2018) developed by McSCert for performing an impact analysis and slicing a Simulink model. After an initial prototype implementation of the tool, I took over the majority of

maintenance and improvements to the Reach/Coreach tool. Given my intimate knowledge of the tool and how it functions, using the Reach/Coreach tool as the engine for performing impact analysis for the BDT was a simple decision.

For a set of specified blocks and lines in a Simulink model, the Reach/Coreach Tool can perform two types of analyses. The *Reach analysis (forward tracing)* identifies all blocks and lines within the model that are affected by the selected blocks and lines. The *Coreach analysis (backward tracing)* finds all blocks and lines within a model that affect the selected blocks and lines. These analyses are performed by tracking *control flow* and *data flow* within a model—therefore, the tool tracks both control and data dependencies.

Data dependencies in a model reflect the potential transfer of data between different objects (such as blocks). Several examples of data dependencies exist in Figure 2.1. For example, the Sum block has a data dependencies on both of the Inport blocks. These are explicit data dependencies, where in each case the dependency is denoted by an object in the model (a signal line). There exist implicit data dependencies between blocks of specific block types in Simulink models: examples of implicit data dependencies in a model are connections between Goto and From blocks as well as connections between Data Store Memory, Data Store Write blocks and Data Store Read blocks. These dependencies will be discussed further in Section 2.2.1.

When performing either a Reach or Coreach analysis, the tool takes a conservative approach when determining the impact of a block or signal in the model. The tool assumes that a signal inputted to a block affects all outgoing signals from a block. Then, for a large number of discrete Simulink blocks typically found in controllers (for example, Subsystem, If, buses), the tool precisely tracks the impact through a block based on its function. In particular,

the tool traces a signal from signal lines to a port of a block, and checks if the block type is in a list of block types that the tool supports with analysis based on specific block functionality to determine the affected or affecting signals associated with that block. These specific block types are described in Section 2.2.1. Therefore, dependencies calculated by the Reach/Coreach Tool represent an over-approximation of the actual dependencies, with fine-grained tracing through a number of common blocks to increase the precision of the analysis and avoid missing dependencies.

It should be noted that the Reach/Coreach tool performs a static analysis on the model, so any modelling errors that would stop compilation are not caught by the tool.

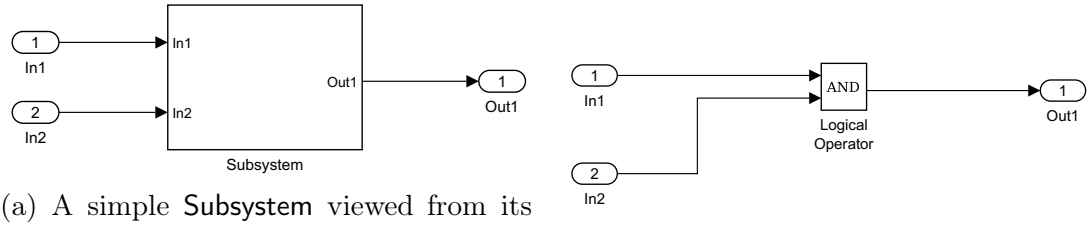
2.2.1 Support for Different Simulink Blocks

As mentioned previously, the Reach/Coreach Tool checks a list of blocks with specific behaviours in order to better track which input signals to a block affect which output signals of that block. The cases for special behaviour handling are listed as follows:

Subsystem Blocks

Simulink Subsystems allow developers to group related blocks and hierarchically organize the model. An example of a Subsystem in Simulink can be seen in Figure 2.2.

When a subsystem input port is encountered during a reachability analysis, the Reach/Coreach Tool will continue the reachability analysis inside the subsystem from the corresponding Inport block in the subsystem. Then, when

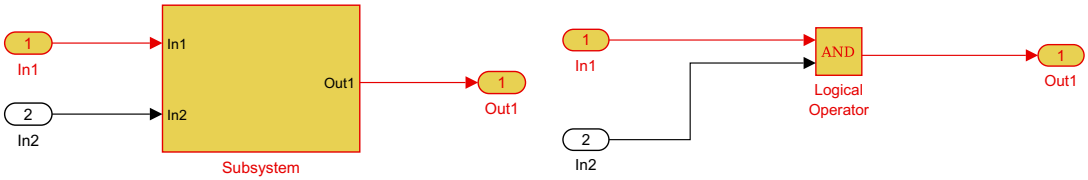


(a) A simple Subsystem viewed from its parent system

(b) Inside of a simple Subsystem

Figure 2.2: A simple example of using a Subsystem block

the analysis traces to an Outport block within a subsystem, it continues from the port of the subsystem the Outport corresponds to. An example of this is shown in Figure 2.3. In this manner, performing a reachability analysis (as well as a coreachability analysis) through subsystem blocks preserves the subsystem hierarchy.



(a) Reach through the simple Subsystem viewed from the parent system.

(b) Reach through the simple Subsystem viewed from inside the Subsystem.

Figure 2.3: An example demonstrating a reach from In1 through the simple Subsystem

Data Flow Blocks

Simulink offers two constructs to route data without the use of signal lines: the Goto/From construct and data stores.

The Goto block passes its input to its corresponding From blocks. There are three types of Goto blocks: *global* Gotos, *local* Gotos, and *scoped* Gotos. A local Goto and its corresponding Froms have to be located within the same subsystem—a local Goto is not visible outside its parent subsystem. Global Goto

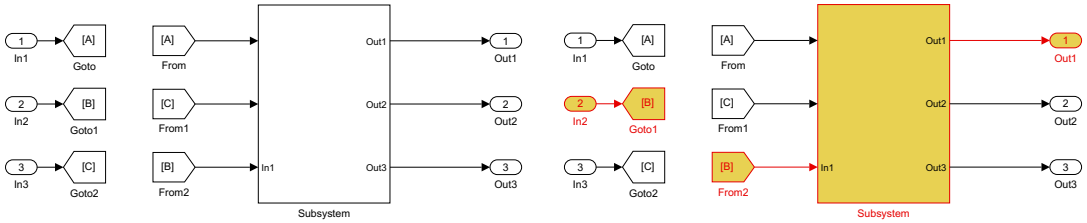
and its corresponding From blocks can be placed anywhere in a model as long as they do not cross non-virtual subsystem[1] boundaries. The visibility of a scoped Goto block is defined by the location of the corresponding Goto Tag Visibility block, which is of the same name as the Goto block and its corresponding Froms. The From and Goto blocks may be used within the subsystem where the Goto Tag Visibility block is placed and any subsystem lower in the model hierarchy.

In a Simulink diagram, a local Goto is denoted by square brackets around its tag. In Figure 2.4, every Goto block in the model is local. In the case of a scoped Goto or From, the tag would be surrounded by braces, for example "{A}". For a global Goto or From, the tag is not surrounded by braces or brackets.

For each of these Goto/From blocks, the tracing principle is the same. When performing a reachability analysis, if the tool encounters a Goto block, it will find the corresponding From block(s). When performing a coreachability analysis, if the tool encounters a From block, the tool will find the corresponding Goto block. Note that while one can find multiple From blocks corresponding to a single Goto, there can only be one Goto block corresponding to a From block. If a scoped Goto or From block is encountered in a reachability or coreachability analysis, the corresponding Goto Tag Visibility block is also considered to be reached or coreached. An example of tracing via Goto/From blocks is shown in Figure 2.4.

Data stores are analogous to variables in textual programming languages such as C. To utilize data stores, three blocks are needed. The first is the Data Store Memory block—analogous to a variable declaration. The second

[1]Non-virtual subsystem is a subsystem whose contents are evaluated as a single unit (atomic execution).

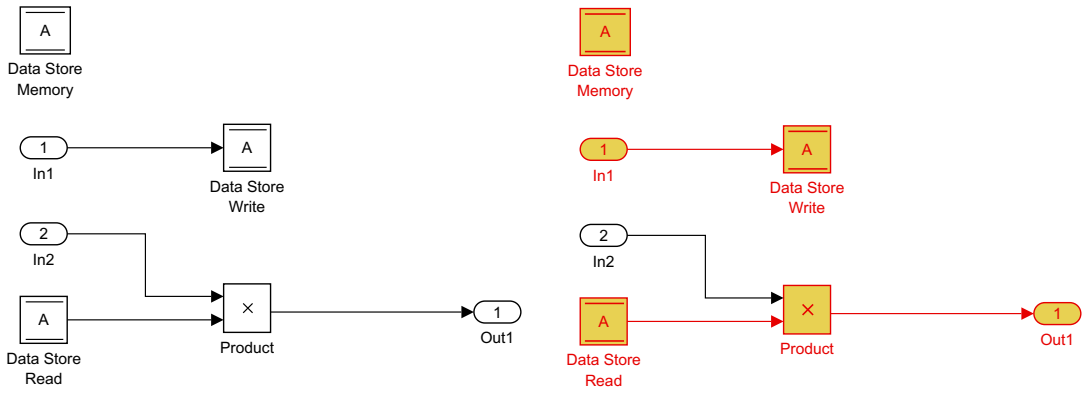


(a) A simple Goto/From pair in a Simulink model

(b) Reach from In2 (through a Goto/From pair) in a Simulink model

Figure 2.4: An example demonstrating how the Reach/Coreach Tool handles Goto/Froms

is the Data Store Write block, which writes data into the data store. The third is the Data Store Read block, which retrieves data from the data store. When performing a reachability analysis, if the tool encounters a Data Store Write block the tool will find the corresponding Data Store Read blocks that retrieve data from its corresponding data store as well as the associated Data Store Memory block. Note that there can be many Data Store Read blocks and Data Store Write blocks corresponding to the same data store denoted by a Data Store Memory block. When performing a coreachability analysis, if the tool encounters a Data Store Read block, the tool will find the corresponding Data Store Write blocks writing to its corresponding data store as well as the associated Data Store Memory block. When performing either the reachability or coreachability analysis, if the tool encounters a Data Store Memory block, the tool will find all corresponding Data Store Read blocks as well as Data Store Write blocks. An example of tracing through data store blocks is shown in Figure 2.5.



(a) A Simulink model with a data store (b) Reach from In1

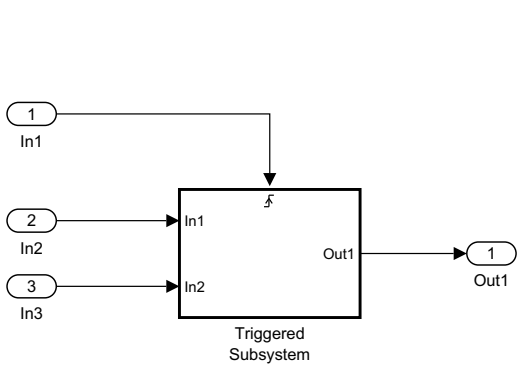
Figure 2.5: An example demonstrating how the Reach/Coreach Tool handles data stores

Subsystem Control Execution Blocks

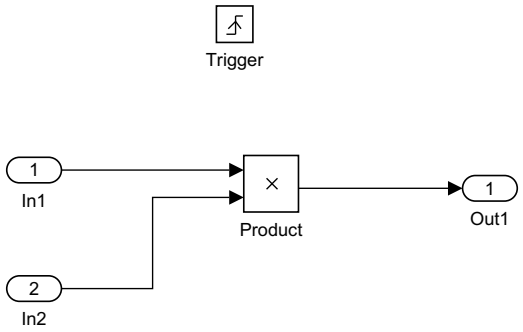
Simulink also has control dependencies. In Simulink, these are derived from subsystems whose contents are conditionally executed when a certain condition is met. For example, a Triggered Subsystem block only executes when a certain event is received at its control input, the TriggerPort block (see Section 2.2.1 for more details). As an illustration, the execution of a Triggered Subsystem as shown in Figure 2.6 is controlled by the signal at the subsystem's Trigger Port. Other examples of blocks that implement control flow include: For Iterator Subsystems, For Each Subsystems, While Iterator Subsystems, If Action Subsystems and the corresponding If blocks, and Enabled Subsystems.

If a control input of a conditional subsystem is encountered when performing a reachability analysis, all blocks and signals within the subsystem are traced. This is because the execution of all blocks and signals in the subsystem is dependent on the control input. This is shown in Figure 2.7.

Control flow logic in Simulink can also be implemented using While Iterator Subsystems and For Iterator Subsystems (for loops). These subsystems also have

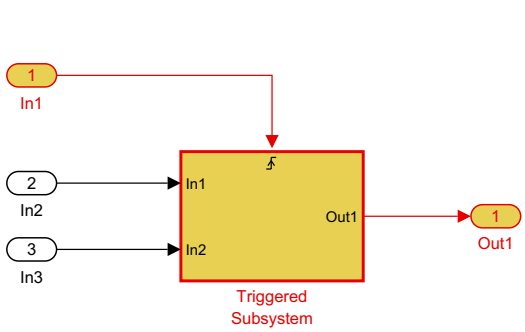


(a) A Triggered Subsystem in Simulink

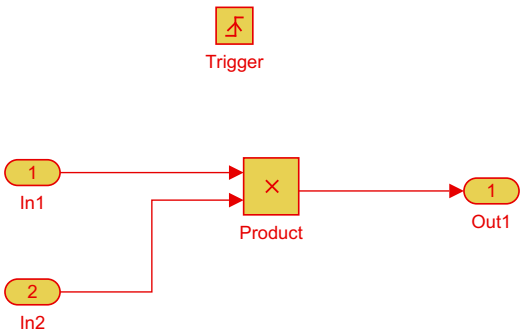


(b) The inside of a Triggered Subsystem in Simulink, with Trigger Port at the top

Figure 2.6: An example of a conditionally executed Subsystem in Simulink



(a) A view of a reach through a Triggered Subsystem from its parent.

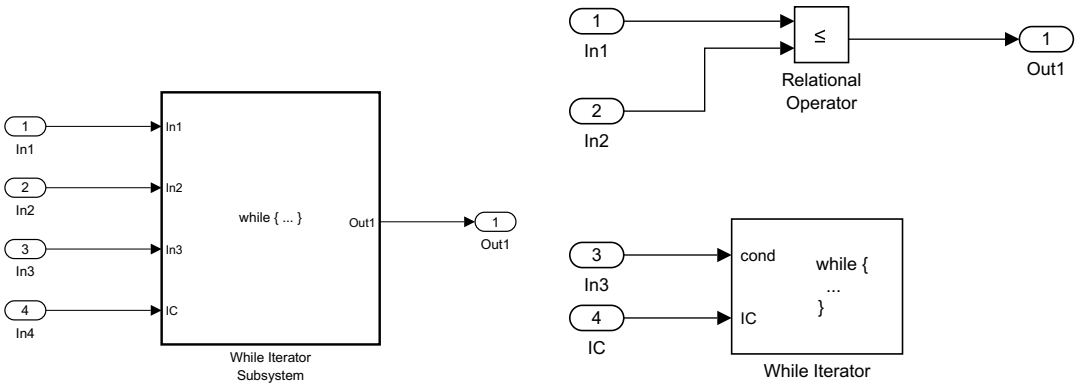


(b) A view of a reach through the inside of a Triggered Subsystem.

Figure 2.7: An example of a reach through a conditionally executed subsystem in Simulink.

control blocks analogous to Trigger Port: the While Iterator block and the For Iterator block. The iterator blocks have a condition signal (labelled as In3) and an initial condition signal (labelled as IC). These iterator blocks control execution of the subsystem as follows: when the initial condition is true, the subsystem executes until the condition signal is false. If the initial condition is false the subsystem doesn't execute. An example of this type of subsystem is shown in Figure 2.8. If the control signal condition is met, execution of the subsystem will occur. Thus, when performing a reachability analysis and one of

these iterator blocks is encountered, all blocks and signals within the subsystem are traced.



(a) A While Iterator Subsystem (b) Inside the While Iterator Subsystem

Figure 2.8: An example of While Iterator Subsystem

When performing a coreachability analysis, it is more difficult to identify any blocks that control the execution of the contents of a subsystem. This is because the control block itself does not have any outgoing connections to any blocks in the model to trace backwards through, as shown in both Figure 2.6 and Figure 2.8. To find any blocks that control execution of blocks within a subsystem, the tool must therefore check if any blocks traced belong to any subsystem whose execution is dependent on control blocks (such as an iterator block or a trigger block.)

If Blocks

The outputs of an If block are used to trigger If Action Subsystems, depending on the evaluation of a specified condition on inputs of the If block. If Action Subsystems have an Action Port block: the subsystem only executes when the input is true.

An example of an If block is shown in Figure 2.9. The If block evaluates

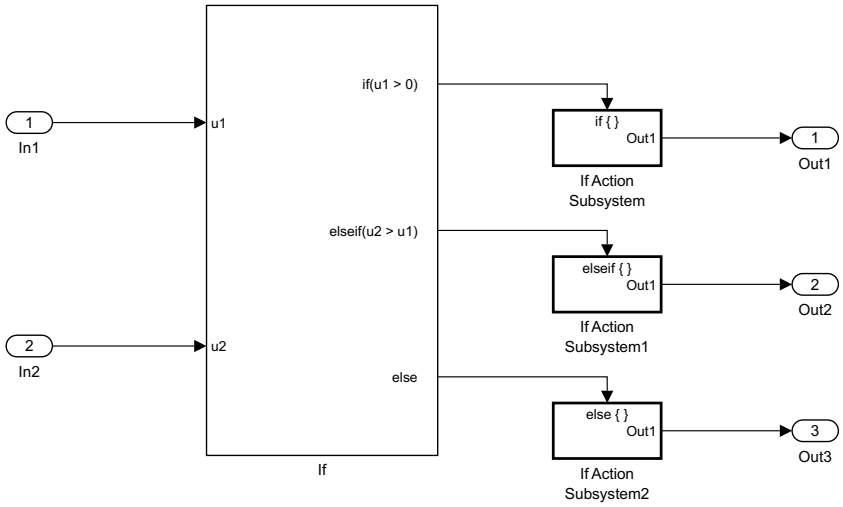


Figure 2.9: An example of an If block used in Simulink.

the expressions top-down. If a logical expression is evaluated to true, the corresponding output is set to true and the connected If Action Subsystem is triggered, and no other conditional expressions are evaluated. If the expression evaluates to false, the next conditional expression is evaluated.

For a condition at an output port of an If block, the BDT currently finds its dependencies on the If block's inputs based on the existence of the block's input names within the condition and within the If block's conditions at the ports above (since the conditions are evaluated top-down): if there exists a dependency between the input and the output, then the name of an input would be found in the condition for an output or in any of the If block's conditions above; otherwise, there is no dependency. An example is shown in Figure 2.10.

Note, however, that this is still an overapproximation of actual dependencies. For instance, if the condition at the second output of the If block in Fig. 2.10 is $u2 \wedge \neg u2$, the output signal is obviously not affected by the value of $u2$ although our tool would still highlight the dependency of the output on $u2$. This is due to the current implementation being a naive approach to evaluating

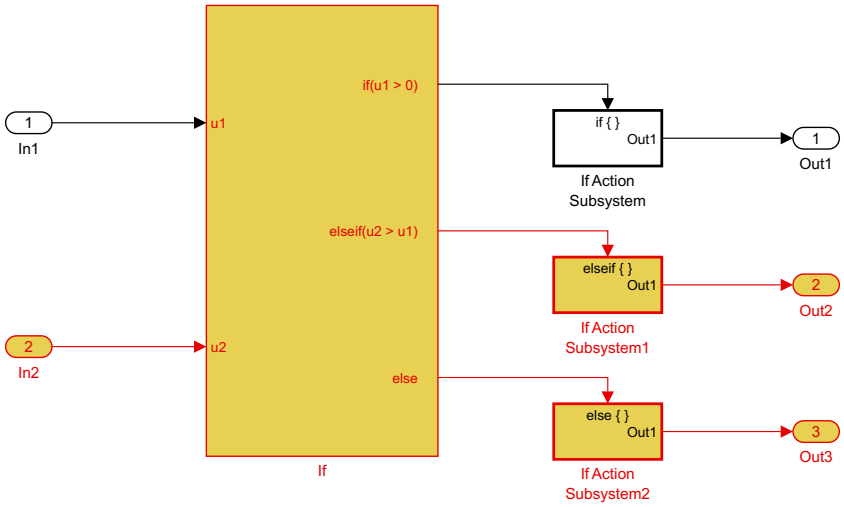


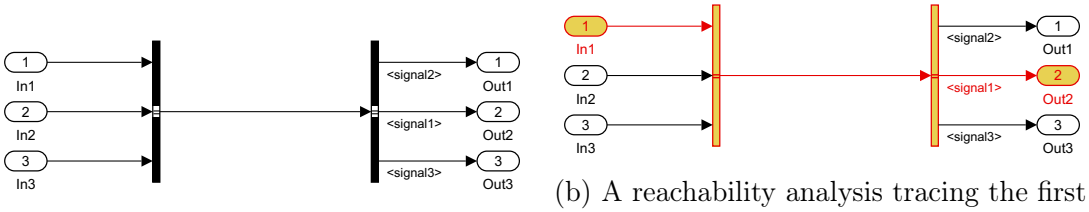
Figure 2.10: Reachability analysis from In2

logical expressions in order to reduce complexity of the tool. In future work, this aspect of the Reach/Coreach Tool could be refined.

Buses

A *bus* is a virtual construct in Simulink used to group signals together into a single *bus signal*. This signal is created at a Bus Creator block, which takes as input signals to group, and outputs the bus signal itself. This bus signal can be routed through inports/outports of subsystems, and through Goto/From pairs. The bus signal is then split into its constituent signals at a Bus Selector block. When performing a reachability analysis, the tool can trace an input signal to the Bus Creator through the bus signal and to the corresponding signal outputted from the Bus Selector. Similarly, when performing a coreachability analysis, the signal can trace an output signal from the Bus Selector through the bus signal and to the corresponding signal inputted to the Bus Creator. An example of a reachability analysis through a bus is shown in Figure 2.11. Figure 2.11 demonstrates that signal order into and out of the data bus is not

necessarily preserved, and the Reach/Coreach Tool traces the correct signal regardless.



(a) An example of a bus

(b) A reachability analysis tracing the first signal into a bus

Figure 2.11: A reachability analysis through a bus

Limitations

There are several limitations to the Reach/Coreach Tool in the context of this thesis. First, the analyses performed with the Reach/Coreach Tool are limited to a single Simulink model. This is a limitation that the BDT aims to address. Second, the analyses performed by the Reach/Coreach Tool do not take execution order in the model into account. Finally, there are several blocks with special behaviours that are not yet accounted for in the Reach/Coreach Tool. For these blocks, the conservative approach is taken (all inputs affect all outputs), but more precise tracing is possible. Most notably, the Reach/Coreach Tool currently does not provide fine tracing through Stateflow blocks.

2.3 The Auto Layout Tool

The Auto Layout Tool is a tool that provides a layout for a Simulink block diagram (McSCert 2018a). To do so, it represents the blocks and lines in the diagram as a directed graph, and interfaces with external graph visualization software in order to provide orderly positioning for the blocks. This is used as

an engine for generating the boundary diagrams.

2.4 Related Work

There are several tools available for model slicing within a single Simulink model. The only commercially available tool is the *Model Slicer* provided with MathWorks' *Simulink Design Verifier* (Reicherdt and Glesner 2012; MathWorks 2018a). The Model Slicer can trace the signal path through a model from a selected block. The tool can be used to find the signal paths affecting a block, or signal paths affected by a block. The tool is very similar in features to the Reach/Coreach Tool in terms of functionality, although there are some key differences. Most notably, any analysis beginning at *virtual* blocks is not supported, where virtual blocks are any blocks which do not affect the execution of the model. Typically, virtual blocks are used to facilitate the structuring of the model. Examples of blocks always considered to be virtual include Gotos, Froms, and Terminator blocks. Inport, Outport, and Subsystem blocks are usually considered virtual as well, except in some specific circumstances. Therefore, the Model Slicer cannnot trace impact from blocks such as Inports and Outports (Pantelic et al. 2018).

Furthermore, the Model Slicer does not trace control flow as precisely as the Reach/Coreach Tool. For example, the Reach/Coreach Tool can trace through the If block more accurately. The Model Slicer assumes that all inports affect all outports of an If block, a much rougher approximation than the tracing performed by the Reach/Coreach Tool (as shown in Section 2.2.1). Finally, the Model Slicer tool is commercial, sold as part of MathWorks' Simulink Design Verifier (SDV), and, as such, incurs a hefty cost. Since the Reach/Coreach Tool

is available via Matlab Central and was developed by the author of this thesis, the Reach/Coreach Tool was a logical choice as the impact analysis engine of the BDT.

Also, there exist some academic tools for performing similar analysis of Simulink models. The slicing tool developed by Glesner et al. (Reicherdt and Glesner 2012) takes into account both data dependencies and control dependencies. The approach uses signal lines to trace data dependencies and derives control dependencies from *Controlled Execution Contexts*, which are essentially schedules for sections of Simulink models that are conditionally dependent on certain Simulink blocks. An example of a Controlled Execution Context is a Triggered Subsystem. This is very similar to the approach used in the Reach/Coreach Tool. However, the Reach/Coreach Tool can trace data dependencies through data routing blocks such as Goto/From pairs in addition to data stores, whereas the tool in (Reicherdt and Glesner 2012) does not. Additionally, that tool is not available.

Another Simulink model slicing tool, *artshop*, is a model management repository used for performing architectural analysis of Simulink models (Kowalewski 2016). This is a multipurpose framework used to store models, as well as requirements and test cases mapped to the models. This framework comes with several built-in tools used for model analysis, including model-slicing tools and static analysis tools. However, these tools seem to be limited to performing analysis on individual models imported into the artshop repository, and the BDT is proposed to perform an impact analysis across a whole system of Simulink models. Additionally, the Reach/Coreach engine used in the BDT provides a far more precise trace than the approach used by the model slicer in artshop—the approach used in the artshop model slicer considers only the

data flow in a model (Kowalewski 2016), whereas the Reach/Coreach Tool additionally traces through control flow. Also, it is important to note that the artshop tool is not available, commercially or otherwise.

Another academic tool called *SimPact* performs change impact analysis within a Simulink model with respect to a previous version of that model to identify the parts of the model affected by a change, as well as to potentially inform testing based on the results of that impact analysis (Rapos and Cordy 2017). While similar to the goal of the BDT, the SimPact tool is limited to impact analysis within a single Simulink model. Since SimPact was not available for download and the Reach/Coreach Tool was more familiar, it was decided that the Reach/Coreach Tool would be better suited to be used as an engine for the BDT.

Finally, MathWorks' *Impact Analysis* tool can be used to track build dependencies between files within a Simulink project (MathWorks 2018b). The tool does not track any links between Simulink models that exist outside of the Matlab environment, such as inter-model connections listed in a dependency matrix. In industrial controllers, the use of this kind of dependency matrix is common practice. Additionally, the MathWorks Impact Analysis tool does not perform any tracing within a Simulink model, such as that performed by the Reach/Coreach Tool.

Chapter 3

The Boundary Diagram Tool

This chapter discusses the BDT and its approach to performing an impact analysis for the application layer of the supervisory controller for our industrial partner's hybrid powertrain system. First, Section 3.1 discusses a general overview of the tool. Then, the approach for performing an impact analysis between models and to network interfaces is discussed in Section 3.2. Performance considerations for the BDT to function in a robust and timely manner are discussed in Section 3.4. Filters for the impact analysis used by the BDT are detailed in Section 3.5. Finally, a quantitative metric used to describe the results of an impact analysis is presented in Section 3.6.

3.1 General Overview

This section presents a general overview of the tool, discussing the necessary operational environment and dependencies as well as requirements and the software architecture of the tool.

3.1.1 Operational Environment

The BDT is a set of Matlab/Simulink scripts and functions. The tool is built for Matlab/Simulink 2016b and later versions. Matlab/Simulink and the BDT are available on Windows, OS X, and Linux.

3.1.2 Dependencies

The tool is dependent on the following tools/files being present on the Matlab path:

1. The Reach/Coreach Tool—A tool for impact analysis within a model. The tool was discussed in Section 2.2.

2. Model Dependency Metadata File—A file containing dependencies between models, as discussed in Section 3.2.

3. CAN Dependency Mapping File—A file containing a mapping of signals in a model that communicate with the CAN network as discussed in Section 3.2.

4. The Auto Layout Tool—The BDT uses the Auto Layout Tool to generate boundary diagrams with a well formatted layout.

The two Matlab tools on which the BDT is dependent are available for Matlab versions 2016b and onwards. They are open-source and available via Matlab Central (McSCert 2018a; McSCert 2018b). The two dependency files used by the tool are Excel spreadsheets with the format specified by the industry partner.

3.1.3 Requirements

The requirements for the BDT were elicited via discussions with our industry partner. The industry partner pitched an initial set of requirements which we refined, elaborated upon, and appended. These requirements are as follows:

1. The tool must be able to track the control flow and data flow from a selection of any combination of blocks and lines in a Simulink model to identify all blocks and lines in a Simulink model affected by the selection.

2. The tool must be able to track the control flow and data flow from a selection of blocks and lines in a Simulink model to identify all blocks and lines in a Simulink model that affect the selection.

3. The tool must be able to identify all blocks in a Simulink model that connect to blocks in other Simulink models in a system via a dependency matrix in order to track control flow between models.

4. The tool must be able to identify what blocks of a given model send data to or receive data from the CAN network interface, and identify the CAN signal name or CAN message name.

5. The tool must calculate the impact metric $im = w_1 \cdot n_1 + w_2 \cdot n_2 + w_3 \cdot n_3$ where n_1 is the number of impacted level 2 safety models, n_2 is the total number of impacted models, and n_3 is the number of impacted CAN signals. Values $w_1, w_2, w_3 \in \mathbb{R}^+$ are weights that can be defined by the project manager.

6. When performing a change impact analysis, there will be no false negatives. All blocks and lines in the system that are impacted by the change must

be identified by the tool.

7. The tool must be able to generate legible and understandable boundary diagrams to illustrate the results of a change impact analysis. A user must be able to look at these diagrams and be able to trace the impact of their change from component to component with little difficulty.

8. The boundary diagrams generated by the tool must be able to be exported for the purposes of documentation.

9. The tool must not exceed an operational time of 15-20 minutes for a single analysis operation for any of the models used by our industry partner. This is in order to not significantly interfere with the development process of the user.

3.1.4 Limitations

As already mentioned, the BDT uses metadata that captures connections betweens Simulink models to properly traverse the dependencies between individual models. The format of the metadata is specific to our industry partner's system for which the tool was originally built. However, the tool is designed such that the format of the metadata is well encapsulated so that the tool can be easily modified for an application on another embedded controller.

Also, the current implementation of the BDT only traces to CAN network interfaces. Support for additional network interfaces can be added in the future and is subject to the same caveats with respect to metadata, since tool functionality built to parse metadata for one company may differ from another. In the future when support for AUTOSAR as a standard is widespread, this

issue can be avoided as companies will take a unified approach to how metadata is represented.

3.1.5 Software Architecture

The BDT is built upon a layered hierarchical software architecture consisting of three layers, with the USES hierarchy shown in Figure 3.1. Each box represents a software module which contains one or more .m files. The A USES B relation, where A is a module and B is a module or an external file, represents a relationship where a software module A calls a function or functions from module B or interacts with external data not contained in a software module (such as a data file).

The first layer can be denoted as the *Interface Layer*, wherein modules provide an interface between the user and the BDT itself. This layer consists of the following modules:

1. Boundary GUI — A module which contains all of the interfaces for the analysis component of the BDT operation. This module hides the function interface, arguments, and configurations for the impact anaysis operation.

2. Diagram GUI — A module which contains all of the interfaces for the diagram generation component of the BDT operation. This module hides the function interface, arguments, and configurations for the impact analysis operation.

The second layer can be denoted as the *Application Logic Layer*, consisting of the followingmodules:

1. Boundary Analysis - A module which contains all of the logic related to the impact analysis. This hides the algorithm of how the impact analysis is performed.

2. Diagram Generation - A module which is used to generate the boundary diagrams themselves. This hides the algorithm for creating the different views implemented as the boundary diagrams, and generating the diagrams themselves.

The final layer can be denoted as the *Data Interaction* layer, consisting of modules that interface with external data from the BDT for use in the main Application Logic layer. Modules in this layer include:

1. ReachCoreach — The module responsible for finding impacted or impacting blocks and signals at the model level. It hides the algorithm for performing impact analysis on a model.

2. GetAffectedCAN — The module responsible for finding impacted CAN signals from a set of blocks. This hides the method and metadata used to determine which blocks affect the CAN network.

3. GetAffectingCAN — The module responsible for finding CAN signals that impact a set of blocks. This hides the method and metadata used to determine which blocks are affected by the CAN network.

4. FindInputs — The module responsible for finding the inputs to a model that are affected by a set of blocks. This hides the method and metadata used to determine how output blocks of a model are connected to input blocks of another model.

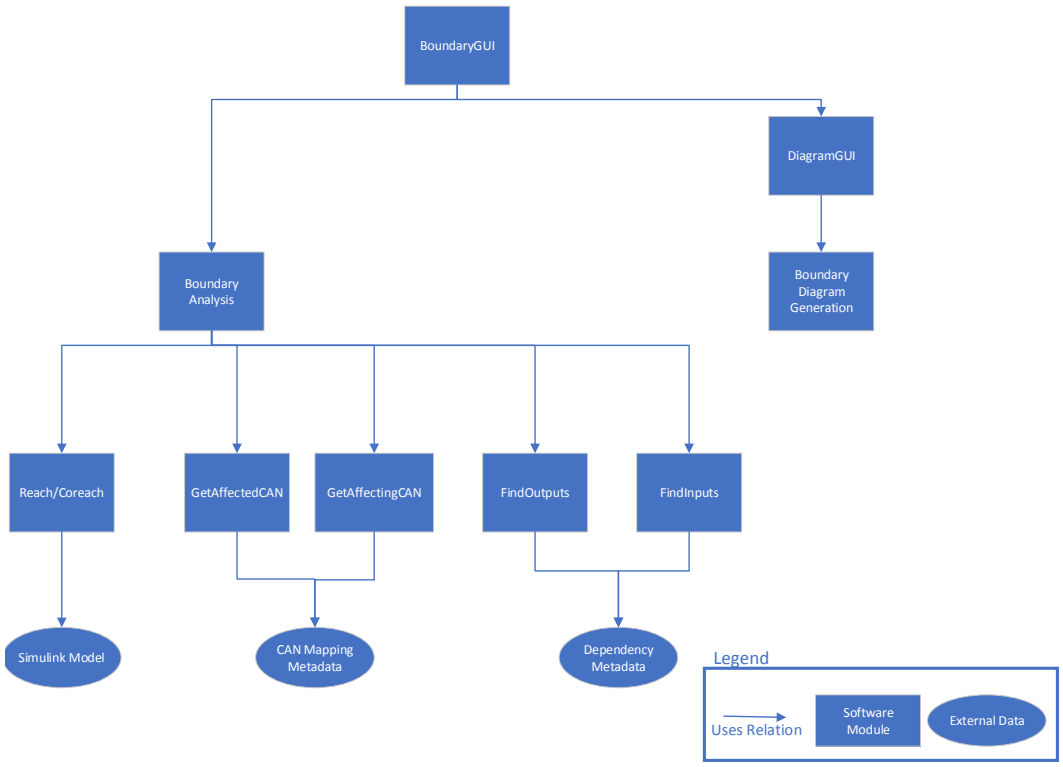


Figure 3.1: The USES hierarchy of the BDT's structure

5. FindOutputs — The module responsible for finding the outputs to a model that are affect a set of blocks. This hides the method and metadata used to determine how input blocks of a model are connected to output blocks of another model.

3.2 Tracing Between Models and to/from Network Interfaces

In many industrial applications, individual models in a large Simulink system are often not connected explicitly via signal lines in a master Simulink model. Instead, the connections between Simulink models are specified via external files. In the case of our industry partner, the connections between models are stored in a *dependency matrix*, which is then instantiated at run-time by base software. The automotive industry is moving towards utilizing the AUTOSAR standard for defining the connections in a standard way, but it is not uncommon for companies to still be utilizing existing in-house solutions or to be in the process of migration. The current implementation of the BDT assumes that connections between models are represented using our industrial partner's dependency matrix, implemented as an Excel spreadsheet, but as AUTOSAR becomes the standard, the tool could easily be adapted for AUTOSAR, and therefore become more general.

The Reach/Coreach Tool cannot trace the impact of a change beyond the boundaries of a model. In order to conduct inter-model forward tracing, as well as trace to network interfaces, once an initial reachability analysis of a change in a model (as performed by the Reach/Coreach Tool) reaches the model's affected outputs, the BDT parses the data representing the connections between Simulink models and to network interfaces, as stored in the dependency matrix and CAN mapping sheet, respectively. This information is then used to continue the reachability analysis from the model's affected outputs in order to find affected parts of other models. The flowchart of the BDT reachability analysis is as shown in Figure 3.2.

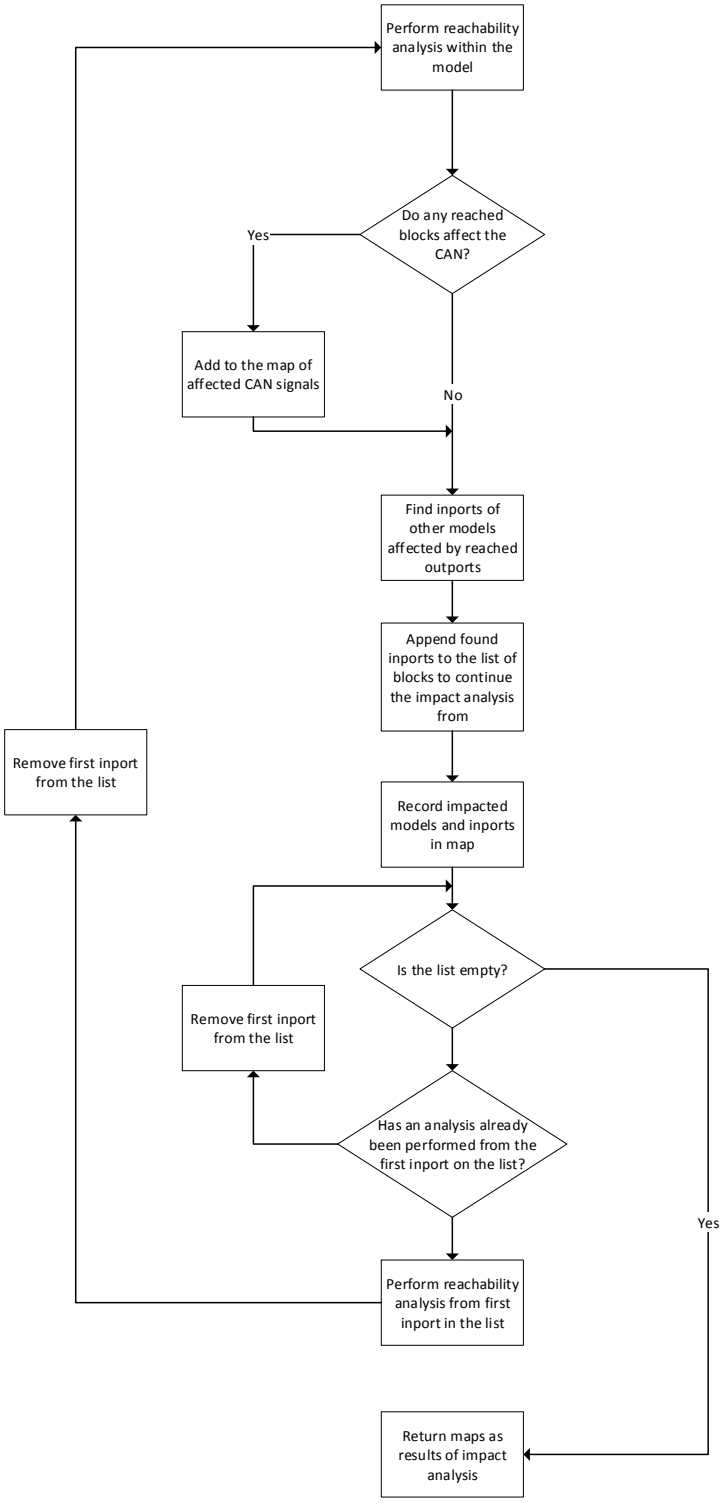


Figure 3.2: A flowchart of the BDT reachability analysis

An initial reachability analysis within a model is first performed from a selection of changes using the Reach/Coreach Tool. The analysis returns a list of reached blocks within the model. The Boundary Diagram Tool then checks this list for reached outports of the model, and identifies any inports in other models that depend on the reached outports. These inports in other models are added to a list of blocks from which new reachability analyses are performed, hereafter referred to as the *recurse cell*. Next, the tool checks the list of reached blocks and identifies reached blocks that affect the CAN interface. The tool then performs a reachability analysis for the first block in the recurse cell, removing it from the list. The process then repeats. The order in which reachability analyses occur when tracing the impact to affected models is breadth-first. If the initial reachability analysis performed within a changed model is considered to be at depth zero, all analyses within immediately affected models would be considered to be at depth one, etc. All analyses to be performed at a certain depth would have to be performed before beginning any analyses at the next depth. As noted, when each individual reachability analysis concludes, some data is recorded, which includes the depth of the analysis at which the reachability analysis from this input occurred. For a coreachability analysis, the algorithm is analogous, as shown in Figure 3.3. [

The dependency matrix used by our industry partner identifies, for each model, the other models that are dependent directly on the model's explicit outputs. The BDT implements a function that parses the matrix and identifies models with the top level inputs connected to a reached output (or top level outputs corresponding to coreached input). The function's inputs include the file containing the dependency matrix and the traced blocks from a reachability/coreachabilty analysis. The function returns blocks in other models

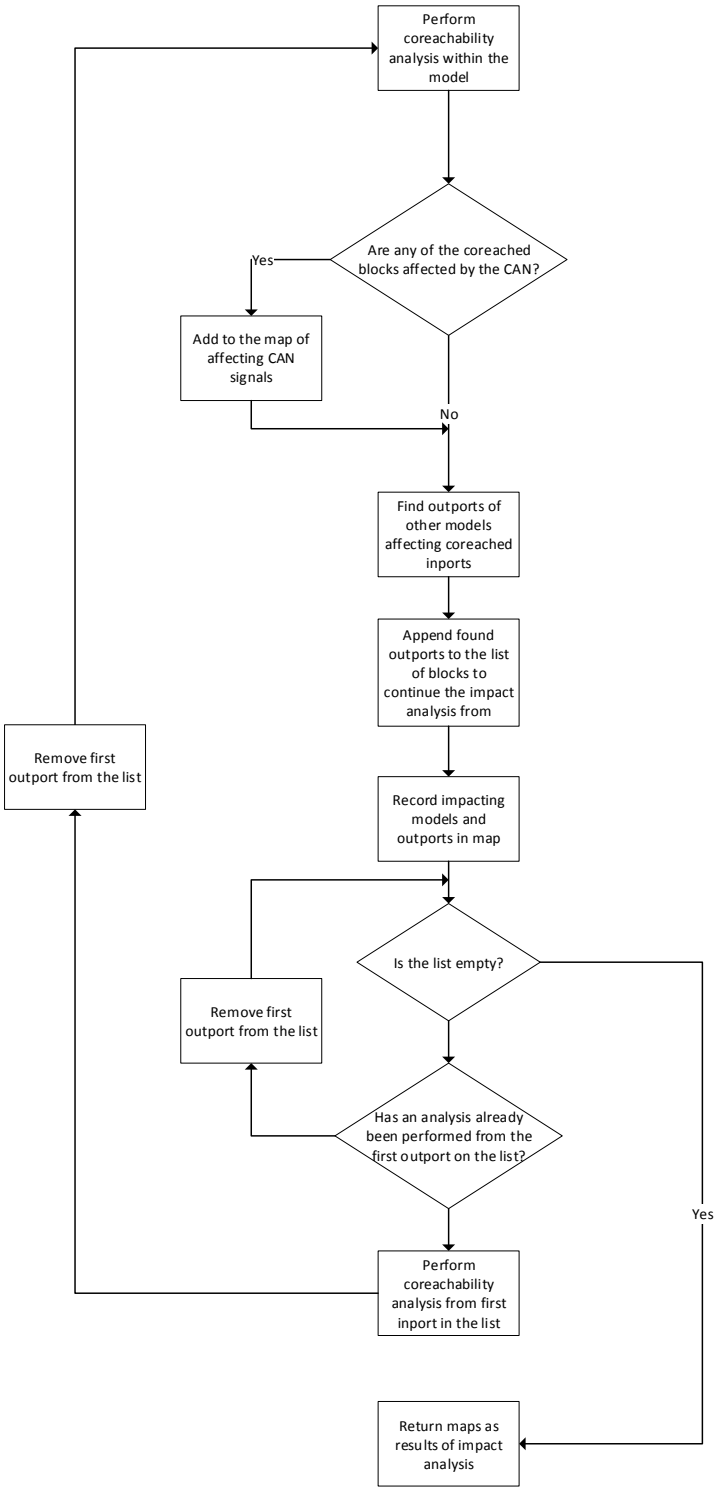


Figure 3.3: A flowchart of the BDT coreachability analysis

connected to these traced blocks. Given that the encoding of the connections between models is going to differ from company to company (or even system to system), the function was carefully designed as to encapsulate this encoding. This makes designing for any changes in the encoding simple.

With respect to tracing to network interfaces, the current implementation of the BDT only traces to CAN interfaces. The data used for identifying connections to the CAN is referred to as the *CAN Mapping Sheet*. The CAN Mapping Sheet denotes which blocks in the model send data to or receive data from CAN network interfaces at run-time; in the case of our industry partner through external function calls. The BDT implements a function that determines if any reached blocks correspond to blocks sending data to the CAN, or if any coreached blocks correspond to blocks receiving data from the CAN. The inputs to the function are the CAN Mapping Sheet and the blocks traced from a reachability/coreachabilty analysis. The function returns information regarding the affected CAN elements as output. This information includes CAN signal name, the name of the message that the CAN signal corresponds to, and the name of the CAN controller it belongs to. All functionality regarding the CAN interface is confined to this function to facilitate ease of adapting it to a differently formatted CAN mapping sheet, or in the future to AUTOSAR representations of CAN mappings.

3.3 Correctness of the BDT Analysis

The correctness of the BDT is almost entirely dependent on the correctness of the Reach/Coreach Tool. Currently, proving the effectiveness and correctness of the Reach/Coreach Tool via a formal proof is incredibly difficult, due to a

lack of formal semantics for Simulink blocks. This problem is compounded by the fact that as newer versions of MATLAB/Simulink are released, new blocks are constantly being released for use in Simulink. However, given the design of the Reach/Coreach tool and how it is used in the BDT, we can still show that the Reach/Coreach Tool (and therefore the BDT) can remain correct with minimal maintenance. This can be demonstrated as follows:

For the base case, assume that the Reach/Coreach Tool is correct, in the sense that no false negatives are returned (there aren't any impacted blocks/signals that are missed by the tool). This means that for the set of all blocks currently in use in Simulink, the Reach/Coreach Tool works as intended. When Simulink introduces a new block for use in Simulink, the tool still produces correct results when performing the impact analysis for all blocks the Reach/Coreach Tool previously worked with. Therefore, in order to ensure that the tool works correctly with the set of all Simulink blocks including the added block, only minor maintenance needs to be performed to add a new case for the Reach/Coreach Tool for the block behaviour of the new block. After this the behaviour of the Reach/Coreach Tool can be said to be correct for the set of all Simulink blocks. This approach can be applied for any number of Simulink blocks being added, and the tool would remain correct.

An alternative approach to proving correctness and effectiveness of the tool is to perform an empirical analysis to measure the effectiveness of the BDT. This will be discussed further in the Future Work section of this thesis.

3.4 Performance Considerations for Industrial Applications

The initial implementation of the analysis component of BDT for tracing between models and to network interfaces had unsatisfactory performance in industrial applications. More precisely, during testing on large scale industry models, it was found that the reachability analyses performed by the Reach/Coreach engine could take hours to complete. When considering that a typical BDT analysis could include on the order of hundreds of these reachability analyses, this result was obviously unacceptable. The Reach/Coreach engine that the BDT was built upon was quickly identified to be the bottleneck.

3.4.1 Performance Improvements for Reach/Coreach on Large Models

In order to identify functions that were candidates for performance improvements, the Reach/Coreach Tool was run and profiled on some of the larger industrial partner's models. Matlab has a built-in profiler for Matlab functions and scripts which can be used to find the number of times each line in the file is executed, as well as the total time spent executing that line of code. In this manner, several bottlenecks were identified. The first identified bottlenecks were the function that finds Data Store Reads corresponding to a given Data Store Write, and the function that finds Froms corresponding to a given Goto. These functions include numerous checks to avoid finding the wrong block if there exist shadowing Froms/Data Store Reads blocks in models where no such shadowing exists. Modeling guidelines for Simulink (The MathWorks 2012)

state that designs with this kind of shadowing are against best practices. In order to improve performance, the Reach/Coreach Tool checks, upon initialization, if shadowing exists in the model, and sets a flag indicating the presence of shadowing in the model. This flag can be referenced by the functions for which shadowing is relevant, and the excess computations can be skipped if the flag indicates that there is no shadowing in the model.

The next bottleneck was traced to the numerous `find_system` calls being made by the tool when searching for Data Store Reads that correspond to Data Store Writes, or Froms that correspond to Gotos. The function `find_system` is a built-in Simulink function that locates objects in a given Simulink model that match a set of parameters specified by the parameters of the function. This function is fast on small models, but for larger Simulink models (containing in the order of 10 000 subsystems and 100 000 blocks), it can take a significant amount of time, in the order of hours. To avoid this, we first identified the functions where `find_system` was being called the most. It turns out that the function calls were concentrated in the same functions where the first described optimization was made, where they were being used to find From blocks corresponding to a given Goto, or Data Store Read blocks corresponding to a given Data Store Write. To avoid this, at the beginning of the Reach/Coreach operation the From blocks for Goto blocks and Data Store Read Blocks for Data Store Write blocks are identified once at the beginning and put in a map. This map can then be referenced instead of performing these expensive function calls.

To illustrate the scale of the improvements to the tool performance, we compare post-optimization run-time of the tool with the initial run-time of the tool on an example model. The initial run-time of the tool on the example

model was 53058.47 seconds. After the first optimization was made, run-time decreased to 21676.428 seconds, cutting the run-time in half. The second optimization further decreased the run-time on the example model to 6017.394 seconds. While the improvements were large, the resulting run-times were still unsatisfactory—a reachability analysis on a single input of a model would still take over an hour and a half.

Even after the first optimization, if was found that there were still excessive calls being made to find corresponding From blocks to Goto blocks. As noted in Section 2.2.1, whenever a block that controls execution of a subsystem is reached, all blocks and signals within that subsystem are reached too. When the tool then reaches all objects contained in the subsystem, it checks for any From blocks outside of the subsystem that would correspond to reached Goto blocks in the subsystem. The initial implementation was naive as it made unwarranted checks for From blocks corresponding to local Gotos within the subsystem. However, the local Gotos by definition cannot have a corresponding From outside the subsystem. This was a particularly large issue in models used by our industry partner, where many local Goto/From pairs are used to increase model readability by reducing the number of signal lines. Therefore, the tool was updated to avoid searching outside the subsystem for From blocks that correspond to local Goto blocks.

The final bottleneck was due to a relatively significant amount of computation time being spent in checking whether a given port to be reached had already been reached. The tool was modified to make use of Matlab's set difference function to check the list of ports to reach against the list of already reached ports. This way, all ports to reach that have already been reached would be removed in a more efficient way by avoiding several function calls and

conditional statements. This optimization further reduced the run-time of the reachability analysis on the example model to 506.231 seconds from an initial time of 6017.394 seconds.

No other improvements to the Reach/Coreach engine were identified that would significantly reduce analysis run-time.

3.4.2 Cache Building

Even with the large performance improvements achieved with the optimization of the Reach/Coreach Tool as described in Section 3.4.1, the performance of an individual reachability analysis of the BDT was still not at a satisfactory level. If each intra-model reachability analysis from one input of a model (using the Reach/Coreach Tool) had a worst-case run-time of ten minutes, a BDT analysis (which is composed of potentially hundreds of these reachability analyses) could take over twelve hours. Even if the worst-case run-time for a single intra-model reachability analysis from a single input of a model was reduced to one minute, a full BDT analysis would still take hundreds of minutes—well above the target for the tool.

To address this issue, the following approach was proposed. A *cache* is constructed that stores the results of reachability analyses from each inport of each model to all the model's outputs. The results are used to avoid analysis within unchanged models: with the cache, the only reachability analysis that needs to be performed is the first one from blocks/signals of interest in a changed model, as all other reachability analyses have already been performed and the results cached.

The major benefit of this approach is a significant amount of time saved

in run-time for the average use case of the BDT. The cache building process is essentially a one-time process. The cache can then be distributed to all developers that are using the tool on the system for which the cache is built. As previously stated, once the cache is built, only a single reachability analysis would now need to be performed in the BDT analysis. This would be the initial reachability analysis from any changed blocks within a model, from which the user would wish to perform the change impact analysis for. From the reached outputs of the model, the tool would find corresponding inputs in other models. However, all subsequent reachability analyses would be replaced by referencing the cache, dramatically decreasing run-time.

These run-time improvements do come at a cost. Most notably, cache maintenance becomes a concern. Keeping the cache up to date becomes critical for ensuring accuracy of the change impact analyses, and this would become increasingly difficult in an agile development environment.

Implementation of Cache

The cache-builder function performs a series of reachability analyses to populate the cache. For each model in the system, from each of its inputs, a reachability analysis within the model is performed using the Reach/Coreach Tool. After each individual reachability analysis is completed, the function finds any top level outputs of the model in the list of reached blocks. Then, these outputs are stored in a list that is mapped to the corresponding input—the input from which the reachability analysis started. The cache is saved to a .MAT file, which contains Matlab formatted data. This file can be used to load the cache into the Matlab workspace, where it can be accessed and used by the BDT for the BDT analysis operations. The cache file can be distributed with the tool.

After the initial implementation of the cache, it was found that the run-time of the cache-builder was too long. While building the cache is a one-time cost with each iteration of the system, it is still desirable to ensure that upon integration it will not impede the software development process by creating unnecessary delay between integration. Therefore it was established that the cache-builder should be able to complete its operations overnight. Initial testing showed that this target was not achieved, so further optimizations would need to be made to the Reach/Coreach Tool. These optimizations were made much in the same way as before. Individual reachability analyses test cases that ran the longest were identified, and then profiled to find the bottlenecks. When these optimizations didn't prove to provide enough speedup, parallelization was considered to be the next best alternative to improve the cache building tool's performance.

The cache building operation presents a perfectly parallel problem. Each individual reachability analysis is entirely independent from the others, and therefore requires no communication between threads. This makes the cache-builder a particularly simple problem to parallelize, and parallelization should produce speedup directly proportional to how many threads are being used. To ensure optimal results for whatever machine it is run on, the cache-builder was parallelized to run on a configurable number of cores. We built the cache using two cores. Results showed a speedup with a factor slightly under 2. This is due to an uneven distribution of work across threads. The work is divided naively between the threads, but some reachability analyses have longer run-time than others.

It is notable that building the cache for the BDT coreachability analysis does not require the same time investment as the BDT reachability operations.

Instead of having to perform a series of coreachability analyses from the outports of the system, the backward cache can be produced simply by taking an inverse mapping of the forward cache. Additionally, in order to obtain a correct analysis one needs to ensure that the cache has been built for the most up to date versions of all models in the system. Each time there is an update to models in the system, the cache for the updated models would need to be rebuilt.

3.5 Filters for Impact Analysis

Examining potential use cases for the BDT by the industry partner, it was found that a full impact analysis throughout the entire system was often not necessary. Specifically, two use cases were identified that do not require full analysis across models: 1) when the developer only needs to see the impact by a change on adjacent models in the system, and 2) when the developer only cares about impact through certain models of interest.

When the developer needs to find only the immediately impacted models, there is no reason to continue the analysis after the propagation of impact to models immediately adjacent to the starting model in the system being analyzed. Therefore, a feature was added to restrict the depth of the impact analysis, as defined in Section 3.2. When a maximum depth is defined by the user, the BDT would not perform any reachability analyses past the indicated depth. In extremes, the maximum depth can be set to 1 to find only immediately impacted models; or it can be left unrestricted for a full depth analysis throughout the entire system.

A *whitelist* has been implemented for the use case when the developer is interested in the propagation of the impact through only certain model: when

using the whitelist the propagation of the impact will only be analyzed for the included the models. All other models would not be included in the results, and the propagation of the impact would not be traced through them.

3.6 Measuring Impact

The BDT quantifies the impact of a proposed change using the *impact metric*. This metric is useful in estimating the risk of implementing a proposed change. Further, it provides an indication of the testing effort and implementation effort required for introducing a change to the system. The metric is defined by our industry partner as follows. Let n_1 be the number of impacted Level-2 safety models, n_2 be the number of impacted models, n_3 and the number of impacted CAN signals. Then, the impact metric is defined as:

$$im = w_1 \cdot n_1 + w_2 \cdot n_2 + w_3 \cdot n_3,$$

where $w_1, w_2, w_3 \in \mathbb{R}^+$. The values for w_1, w_2, and w_3 are weights that can be defined by a user based on the risk associated with each type of impacted artifacts. This is because the inherent risk associated with impacting any of these artifacts is dependent on the context of the user's system. They can be affected by any number of factors such as the number of Level-2 safety models compared to other models, the number of CAN signals available in the system, and how many systems the CAN signals propagate to. Therefore, implementing static weights without considering the context of the system could lead to incorrect risk assessment. However this means that it is very important for the user to produce good values for these weights in order to get full value from

using the tool.

In general, an organization can define an impact metric to suit its needs: the metric can be customized according to the specifics of the embedded software at hand, as well as the specifics of the change management process and the metric's intended application in the process. Other metrics can be defined that e.g. account for the number of impacted blocks within models, the number of affected CAN controllers, etc.

3.7 Summary

This chapter discussed the requirements for the BDT, and described the implementation of the impact analysis component of the tool. These requirements are relevant to the boundary diagram generation discussed in the next chapter, and the implementation of the impact analysis component of the tool itself affects both how the boundary diagrams are generated as well as how the tool is used in the change management process.

Chapter 4

Boundary Diagram Generation

This chapter describes the approach used in the BDT to present results from the impact analysis by using various forms of diagrams. Besides marking dependencies within Simulink models, the BDT also presents impact analysis results using various other diagrams. The use of diagrams was initially inspired by boundary diagrams. Although the diagrams generated by the BDT include boundary diagrams, several of the diagrams produced by the tool are not strictly boundary diagrams, but rather block diagrams in general (e.g. context diagrams, and slices of context and boundary diagrams showing only relevant components and dependencies identified by impact analysis). However, for the purposes of this thesis, by a slight abuse of the terminology, the diagrams produced by the tool will be referred to as boundary diagrams.

First, the implementation of boundary diagram generation is discussed in Section 4.1. Legibility concerns with with generated boundary diagrams are described in Section 4.2. Next, a solution to the issues presented in Section 4.2 is presented in Section 4.3. Finally, several diagram views are presented in Section 4.4 to represent specific information from the full impact analysis.

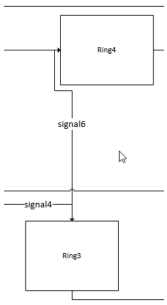


Figure 4.1: Illustration of issues with the boundary diagram generation in Microsoft Visio

4.1 Boundary Diagram Generation: Implementation

Several possible implementations for boundary diagram generation were considered. The initial idea was to generate diagrams in Graphviz (GraphViz 2018), but it was later discovered that our industry partner would not allow the use of open-source software as part of their toolchain. Microsoft Visio was then evaluated for diagram generation as it is a commercially licensed third-party tool that was already available to our industry partner. Initial results were promising, but several issues were discovered when attempting to create even moderately complex boundary diagrams. As shown in Figure 4.1, there were several issues with line routing in the diagram. Lines would consistently overlap at their start and endpoints, making the diagram difficult to read. Being able to differentiate between individual lines is very important for understanding how impact propagates through this diagram.

To address these issues, Matlab/Simulink was chosen for the implementation of boundary diagram generation. Although the directed graph generation capabilities of Matlab are fairly limited and would not be suitable for generating

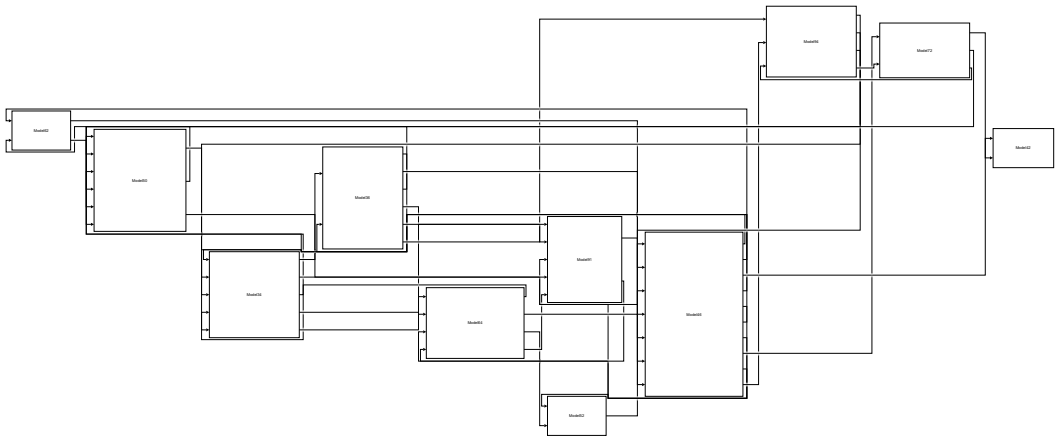


Figure 4.2: An example of a full feedback boundary diagram

boundary diagrams, the open-source *Auto Layout Tool* (McSCert 2018a)—properly adapted to handle some specific requirements of layout of boundary diagrams—has automatic layout capabilities required to generate comprehensible diagrams. Using Simulink for diagram visualization also offers developers a familiar look-and-feel, and keeps the tool implementation within Matlab.

4.2 Legibility Concerns

When testing the Simulink implementation of boundary diagram generation using the results of the impact analysis on our industrial partner's controller, a new issue presented itself. For example, observe Figure 4.2, which shows the impact from Model36 on Model42. Given the high coupling of the Simulink models within the electrified powertrain controller, a change made in a model has large impact on other models. Thus, the resulting diagrams for large industrial systems are excessively large and complicated. Even finding only the models that had been directly impacted by the change on the diagram from Figure 4.2 proved to be a difficult task. Due to low readability, these

full impact diagrams are often not useful in documenting/understanding the impact of a change.

For smaller systems of Simulink models, the full impact diagrams could still be readable. In general, this diagram, hereafter referred to as a *full feedback diagram*, was typically found to be impractical for analyses of large systems of Simulink models. Legibility concerns regarding the diagrams need to be addressed to make them suitable for industrial use.

4.3 Interactive Exploration

The first method of addressing legibility concerns with the full feedback boundary diagrams was to implement *interactive exploration* of the diagrams, as a well-known technique for visualization of large graphs (Herman, Melançon, and Marshall 2000). First, the tool produces an initial diagram showing just the initial model where a change was implemented, and its immediately affected models and CAN interfaces. Each model and CAN interface in the diagram is represented as a node, with edges between nodes representing the data flow between them. The user can then *expand* the diagram from a selected node, showing how the impact of the change further propagates from the node to its immediately affected models or CAN interfaces. This is shown in Figure 4.3, which generates an interactively explorable diagram for a different impact analysis: namely, the impact from Model45 to Model9 to a depth of 3. After the diagram demonstrating the immediate impact of Model45 is generated in Figure 4.3a, the developer then chooses a trace to explore by selecting a model from a set of immediately impacted models, e.g. Model24 or Model80. The BDT then shows immediately affected models for these models as shown in

Figure 4.3b.

Figure 4.3c shows how the diagram would look if every node is recursively expanded, essentially showing all possible paths that the impact propagates from Model45 to Model9. Note that no feedback loops exist in the diagram even though the diagram shows the same impact analysis results from Figure 4.2. This is because in the interactive exploration diagrams, all loops in the diagram are unwound in order to increase legibility and comprehensability of diagrams. This is achieved by introducing a fresh node to the boundary diagram each time a model is encountered in an impact analysis, whether or not a node for that model was previously already created in the diagram. The naming convention for the nodes seen in Figure 4.3 is as follows: $ModelName\_d\_i$, where $ModelName$ is the name of the model, d is the depth at which the model was impacted, and i is the unique identifier of the model's copy at depth d. Also, the tool allows the user to interactively click on a line in the interactive exploration diagram and view the names of all associated signals to that line. Generating the initial immediate impact and expanding a node takes a few seconds for each node.

The approach of interactive exploration coupled with the unwinding of loops in the boundary diagram has substantially improved the legibility and comprehensibility of the BDT generated diagrams. It becomes much more obvious for any user to identify the relative order in which a change impacts several models, which can be quite difficult to identify in a diagram with numerous feedbacks. Additionally, unrolling the feedback allows the diagram to be drawn as a strict left-to-right hierarchical graph. This makes parsing the diagram far easier for any user to follow and makes interactive exploration for the diagram far more intuitive.

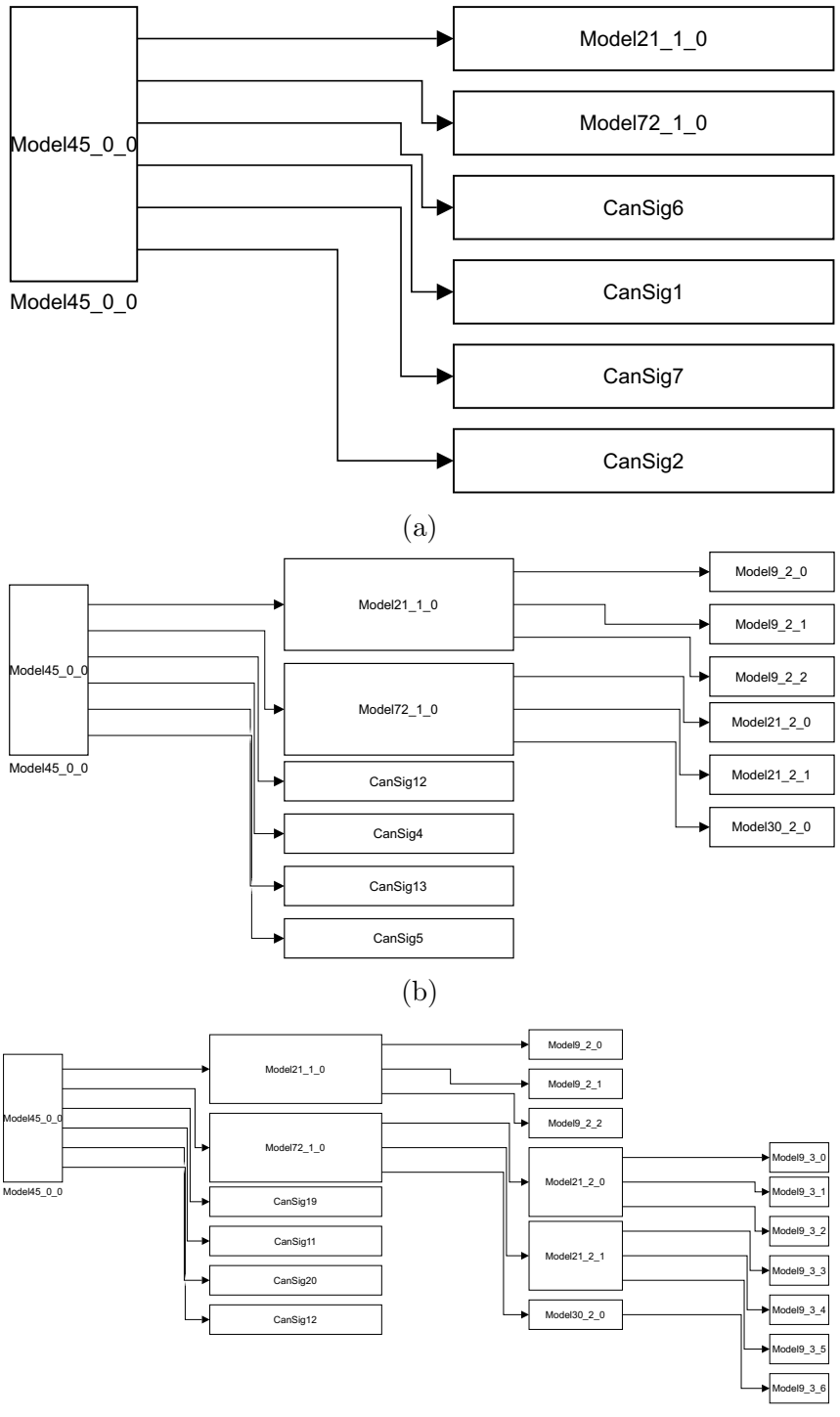


Figure 4.3: An example of an interactively explored boundary diagram for the impact of Model45 to Model9.

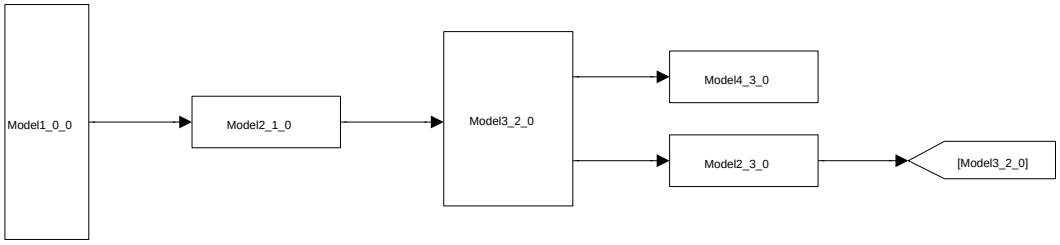


Figure 4.4: Gotos are used for feedback

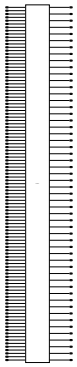
Of course, while the diagram from Figure 4.3c can be generated via interactive exploration, the BDT can also generate it without intermediate steps. For the case of the example from Figure 4.3c, while traces may encounter the same model multiple times, each time it is with different signals. If a trace were to encounter the same signals multiple times, then its exploration would continue looping indefinitely. In this case, Simulink Goto constructs are used to illustrate which block represents the model slice with the re-impacted signals. This is illustrated in Figure 4.4: the output signals of Model2_3_0 are fed back to Model3_2_0.

4.4 Specific Diagram Views

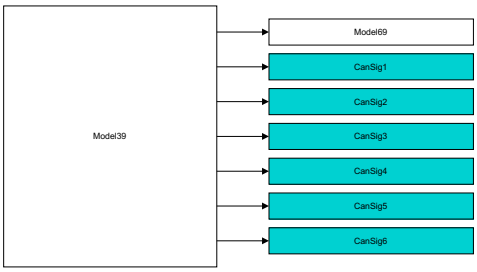
Several views were identified as useful to present specific information from the full impact analysis. These views complement the full feedback diagrams and interactively explorable diagrams described above. For possible applications of these different types of diagrams, please refer to Section 5.3.

4.4.1 Immediate Impact

This view shows the initial model from which a change was made, and the models and CAN signals which are directly affected by this change.



(a) The top level of a Model39.



(b) The immediate impact of a change Model39

This view is shown in Figure 4.5b. Effectively, the diagram represents a slice of Model39's context diagram that shows only the impacted parts of Model39's interface, which is shown in Figure 4.5a. It deliberately excludes the unaffected inputs and outputs. Included in this slice are both the models and network interface components immediately impacted by the change. In the case of Figure 4.5b, 6 CAN signals and 1 other model within the system are impacted.

4.4.2 Model A to Model B

This view shows how the impact of a change in a model A propagates to a model B. Sometimes, a developer might be interested in the impact of change to a specific model of interest. This view can be represented as a full feedback diagram, an interactively explorable diagram, or an immediate impact diagram. The Model A to B view shows a slice of the full impact of a change impact analysis originating in Model A, where only part of the impact that propagates to Model B is included. An example of the Model A to B filter being used is in Figure 4.3, where the diagram shows the impact of a change in Model36 on Model42.

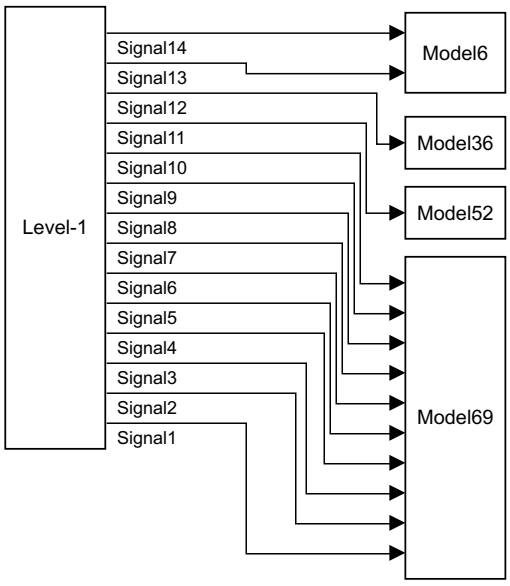


Figure 4.6: An example of a boundary diagram view showing the change impact on safety

4.4.3 Impact on Functional Safety

ISO 26262 requires that each change request must undergo a comprehensive impact analysis to identify impacts to functional safety (ISO 2011a). Our industrial partner uses the E-Gas 3-Level Monitoring Concept (Workgroup 2013). This monitoring concept is composed of three levels. Level-1 is the functional level, which contains the control software. Level-2 monitors for any defective processes in Level-1. Level-3 is the controller monitoring level, which independently tests the control software at run-time. The (sliced) context diagram in Figure 4.6 shows the impact of changes within Level-1 models (non-safety, control models) on Level-2 (safety) models—in this particular case, only one safety model is affected. For systems built to comply with ISO 26262, a view can be provided demonstrating the impact of a change originating from Level-1 (non-safety, control models) to Level-2 (functional safety models). This view abstracts all of the Level-1 models into a single black box, and demonstrates

impacted signals connecting to individual Level-2 safety models and how the impact of that change propagates from those initial Level 2 models through the rest of Level-2. This can be shown in Figure 4.6, where all the control models are abstracted to the black box denoted as Level-1.

This view aids in satisfying the ISO 26262 requirement which states that changes to a system require a change impact analysis to determine the impact on functional safety. It is important to note, however, that the tool only traces through Simulink models and to the network interfaces to other non-Simulink components of the system. To fully comply with ISO 26262, there should be analyses performed beyond these interfaces to other connected software components in addition to the analysis from the BDT as described in Figure 5.3.

Chapter 5

Boundary Diagram Tool in Software Change Management

This chapter describes a generic process for managing changes within an industrial-scale software system and details the BDT's integration into the process. Section 5.1.1 introduces a generic software change request procedure. Section 5.2 discusses practical considerations for updating and maintaining the BDT's cache without significantly interfering with developer workflow. Finally, Section 5.3 discusses opportunities for the BDT's application in several different phases of the aforementioned software change management procedure.

5.1 Software Change Management

5.1.1 The Software Change Procedure

The change procedure referenced in this thesis is a generic process for managing changes within industrial-scale software systems, as applied to the model-based development. This generic procedure, presented in Figure 5.1, parallels the

change request procedure used by our industrial partner. It involves changing the system iteratively: between two iterations, system-level change requests are analyzed, implemented, the modifications are integrated, and a working version of the system (called a baseline) is used by the developers as a base for the next set of changes. The procedure outlined here is for one system change between two baselines.

The procedure begins with a request for a system-level change that needs to be made to the system, called the *system change request*. Each system level change is first analyzed and evaluated to determine whether the change will indeed be implemented, and, if so, when to schedule it. If the system change request is determined to be worthwhile, it is then decomposed from the system-level to several model-level change requests, shown in Figure 5.1 as a *software change request*. These software change requests are then assigned to developers, and the changes are evaluated again with respect to their impact on the functionality of individual models. If a software change is determined unsuitable, the system-level change decomposition might be revisited, and the implementation of the system change (and all of its descendant software changes) may be rejected or postponed.

If all software change requests have been approved, the individual developers derive a set of specific requirements for the change that they are to implement. Then, the developer designs and implements the necessary changes to the model to satisfy the requirements. The resulting model is then tested against the newly derived requirements, as well as all existing requirements to assure that the implemented changes do not interfere with existing functionality. When testing has been completed, the changes and the accompanying design documentation are reviewed by a senior engineer to approve the changes for integration, where

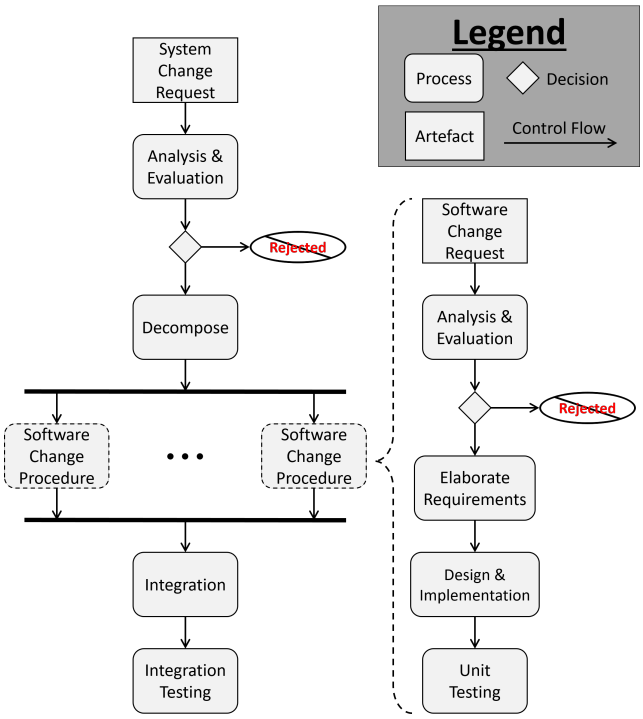


Figure 5.1: Change Request Procedure

system-level conflicts between individual software change requests are resolved. The newly integrated system is then tested using model-based design techniques such as Model-in-the-Loop, Hardware-in-the-Loop testing, etc.

5.2 Maintaining Cache

As mentioned previously in Section 3.4.2, one of the challenges with the implementation of the tool is ensuring that the cache is up-to-date. Developers typically work implementing changes to models concurrently. Thus, any evaluation of an impact analysis will not take into account all of the changes being made simultaneously by other developers, but rather reflect the impact of the change against the baseline. For example, let us assume that two developers are working in parallel on software change requests for Model A and Model B. If

Model B is dependent upon Model A, the impact analysis from a change within Model A may show that the change in A impacts Model B. However, any further propagation from Model B will not take into account any modifications currently being made to Model B. Therefore, an impact analysis on a model will only be accurate if the cache has been updated to reflect all of the model-level changes that are decomposed from a system change request.

Building the cache for our industrial partner is a task that takes approximately 10–12 hours. Performing a 10–12 hour operation to rebuild the cache every time a developer makes a change would be unacceptably intrusive in developer workflow. The current implementation of the tool can perform a less intensive "update" to the cache, where the cache is only rebuilt for selected models. The cache can then be rebuilt for any model for which there were changes since the previous version of the cache was built. This reduces the full 10–12 hour cache building operation to a more manageable run-time. However, this reduced run-time will vary from model to model. For the largest models in use by our industry partner, the update takes several hours. For smaller models, updates can take in the order of minutes. Additionally, distributing the cache to each developer every time an individual developer makes a modification to a model is infeasible. Therefore, ensuring that all impact analyses are performed with an up-to-date cache is infeasible in practice. And so the frequency and timing of cache updates must therefore be carefully determined based on the tool's intended application.

5.3 Applications in the Software Change Procedure

Many of the impact analysis activities hinge on determining whether some model or network interface of interest is impacted by a change. This can be determined by selecting the modified model elements for use as the starting point of the BDT reachability analysis. Alternatively, Simulink's *Model Comparison* can be used to find differences between a model prior to, and after a change; the BDT reachability analysis can then be applied from the changed parts of the model. The resulting trace is useful in determining not just if a model is impacted, but also how the impact propagates through the system to reach it.

The BDT has several possible applications in the software change procedure of Figure 5.1. It can be used to perform impact analysis to aid in compliance with safety standards such as ISO 26262, inform decisions regarding whether to integrate a change to a given model, and aid in relieving testing and verification efforts for models both during implementation and integration of changes. The applications will be discussed next.

5.3.1 BDT in Software Change Request Analysis and Evaluation

ISO 26262 requires that each change request must undergo a comprehensive impact analysis (ISO 2011a). Such an impact analysis would be performed in the Analysis and Evaluation phase of the software change procedure depicted on the right of Figure 5.1. The impact analysis requirements are described in Section 8.4.3 of ISO 26262, Part 8 (ISO 2011a). They dictate five main points

be addressed by the impact analysis. These points are as follows:

1. the identification of the type of change request,
2. the identification of work products that are changed and/or affected,
3. the identification and involvement of the parties involved with and/or affected by the change,
4. the potential impact on functional safety, and
5. the scheduling for realization and verification of the requested change.

Of these items, the BDT can assist in addressing the latter four whereas the first item, the type of change request, should be determined by the developer.

Although the impact analysis as per ISO 26262 should identify the impact on all relevant work products—not Simulink models only—the use of the BDT at the Simulink level can identify affected models and CAN network interfaces. This information can then be leveraged to identify parties affected by the change as well: after finding affected models and network interfaces, the developer can alert other developers who are responsible for any affected models. Since the BDT can identify affected models that are specifically designated as functional safety models, as demonstrated in the Impact to Safety diagram discussed in Section 4.4.3 and the impact metric, the impact on functional safety can be addressed. If models that affect functional safety are identified, the impact analysis results are used to inform the appropriate safety management process. Finally, while the tool does not directly estimate the time taken to implement and verify a change, information gleaned from the generated diagrams as well as from the impact metric could prove a useful aid in performing such an estimation.

Finally, we note that, in order to perform the impact analysis using the BDT in the Analysis and Evaluation phase of the software change procedure, the developer needs to rely on their own skills, experience, and grasp of the model where changes would be implemented to estimate which model elements would be modified in the change implementation, and perform the analysis from those elements. The process typically involves an overestimation of the parts of the model to be modified—consequently, the impact analysis results might present a false positives with regards to the actual impact of the actual change that would be implemented. However, the results are still the safest approximation of the impact of the planned change given the information available at this phase of the software change procedure.

5.3.2 BDT in Change Implementation

The core functionality of the BDT—to identify models and network interfaces impacted by a change—can be leveraged to provide an immediate impact analysis of a change in a model, identifying the models and network interfaces directly connected to the model where the change is made. This allows the user to easily identify other developers with whom they need to collaborate to ensure that implementation of the change proceeds smoothly. For example, if the data type of a top-level outport of a model is changed, the user can notify developers of other models that use that outport to accommodate for the change in data type. Through use of the generated diagrams or the impact metric to view the full scope of the impact of a change, a developer can be alerted when the scope of change's impact is larger than the expected impact determined in the change request evaluation. These diagrams can also be included in the design

documentation to be reviewed in the change request procedure by management as part of the procedure described in Section 5.1.1.

Finally, the BDT's Reach/Coreach engine can be used at a local level, within a Simulink model, to improve model comprehension, aid in refactoring, and identify dead/unreachable parts of the model, as reported in (Pantelic et al. 2018). Further, these activities can be extended at the inter-model level using the full capabilities of the BDT to trace beyond the model boundaries and continue the impact analysis.

It is important to note, as mentioned in Section 5.2, individual software changes on different models are performed in parallel. This means that the impact analysis during this phase of the software change procedure does not take into account any implemented changes made on other models. Thus, possible unintended interactions between changes made to individual models will not necessarily be considered.

5.3.3 Impact Analysis in Regression Testing

Good software development practices indicate that, once a change is made to a Simulink model, the model must be verified with respect to its requirements. This includes existing requirements as well as new requirements generated in the software change procedure. In order to ensure any previous requirements are still satisfied, a suite of regression test cases corresponding to these existing requirements that can be autonomously ran after a proposed change is implemented to ensure that said requirements still hold. As complexity of software components increases, the number of regression test cases required to verify all existing requirements increases and a larger computation time is required to

verify this suite of regression tests.

In the related work on UML diagrams, regression test cases are classified into three types: *obsolete*, *retestable*, and *reusable* (Briand, Labiche, and He 2009). The obsolete test cases are the test cases that are no longer applicable to the modified model: e.g., an input has been deleted from the original model. Retestable test cases are test cases that need to be re-run due to a change potentially affecting the result. Reusable test cases are test cases for which their result would be completely unaffected by the change.

The BDT can be used to potentially identify test cases which are reuseable as opposed to retestable, and thus can reduce testing effort needed to perform regression testing upon making a change. Additionally, future testing efforts can be saved by identifying certain test cases as obsolete. Using the BDT in this manner requires an existing regression test suite with a mapping from these regression tests to corresponding components (blocks, signals) in the model. Automated test selection can be accomplished for regression tests for an individual model after making a change, or for regression tests of the system at large after making a change, since the BDT can perform its impact analysis beyond the initial model to the rest of the system.

Additionally, the BDT and an off-the-shelf tool for automatic test generation for Simulink models can be integrated in the following manner. After a change has been made to the Simulink model, the BDT can be used to make a slice of the model that only includes blocks and signals that either affect any changed blocks and signals or are affected by the changed blocks or signals. Then, the off-the-shelf tool can be used on the model slice to generate test cases that thoroughly exercise model's behaviour. This would reduce the off-the-shelf tool's total testing time, and as well as allowing the BDT to classify the test

cases, update the regression test suite accordingly, and run the retestable cases.

The functionality for regression test case selection has not been implemented yet.

5.3.4 Impact Analysis in Integration

By the time the integration phase of the software change procedure occurs, all model-level change requests have been completed. However, as noted at the end of Section 5.3.2, possible interactions between individual changes of different models would not be identified by any impact analysis performed in the implementation phase of the software change procedure. This can be rectified during the integration phase. After all the changes have been implemented, the BDT cache can be rebuilt using the updated models. The impact analyses performed by each developer can be run again in the context of the fully integrated system. By doing so, developers can identify and investigate any unexpected impact. The results from the impact metric or generated diagrams from these analyses can also be included in the development artefacts documenting integration.

Chapter 6

Conclusions and Future Work

This thesis proposes the BDT, a tool for impact analysis within Simulink designs of embedded systems. The tool improves safety by performing an impact analysis, which can identify additional changes that may need to be made that may otherwise have not been made. In doing so it additionally improves developer productivity by providing automation of impact analysis as part of the software change management process. In particular, it supports analysis, implementation, and verification of software changes. As ISO 26262 requires impact analysis to be performed for any software change request (ISO 2011a), the results from the impact analysis with the BDT aid compliance with the standards. For the purposes of this thesis, the tool was implemented for a specific embedded system: it tracks impact of changes in the supervisory controller of the electrified powertrain system of our industrial partner. This system consists of dozens of Simulink models, which contain hundreds of thousands of blocks and tens of thousands of subsystems.

The BDT was built upon the Reach/Coreach Tool, a tool that performs model slicing within a single model. The BDT leverages this functionality

to perform impact analysis within a Simulink model, and extending that functionality to a system of Simulink models. The inter-model analysis stops at the boundaries indicated by network interfaces, where the models interface with the communication hardware such as CAN. To implement inter-model tracing and tracing to network interfaces, the BDT leverages dependency matrices and CAN mapping sheets that specify connections between models, and communication interfaces, respectively. The Reach/Coreach Tool underwent significant optimizations in order to achieve better performance. Consequently, the Reach/Coreach Tool now represents a robust model slicing engine that can be used in other industrial applications as well.

The BDT can present impact analysis results directly within Simulink models, as well as using various boundary diagrams. Boundary diagrams provide different views of the results, at different levels of abstraction. Due to the complexity of our industrial partner's software system, presenting impact propagation through a complex web of dependencies in a legible manner proved difficult. To address the issue, techniques such as interactive exploration and loop unwinding for visualizing large graphs were applied. This approach was selected due to design constraints restricting the use of third party tools (such as GraphViz) for visualization of the boundary diagrams. If using third-party software was a possibility, graph visualization could demonstrably improve (due to constantly improving graph visualization techniques) while incurring less of a maintenance cost for the BDT itself.

The tool also employs a quantitative approach to evaluate the impact of a change: it measures the extent of the impact of a change by calculating a metric carefully defined by our industrial partner to incorporate relevant factors such as the number of affected models, the number of affected safety models,

and the number of CAN signals affected.

Finally, applications of the BDT in the software change request procedure of our industry partner were identified and the tool was integrated in the industrial partner's change management process. The BDT can be used in several phases of the software change request procedure: the analysis and evaluation phase, the design/implementation phase, and the integration phase all benefit from change impact analyses performed by the BDT.

The development of the BDT analysis and diagram generation, the improvements to the Reach/Coreach Tool for impact analysis on industrial models, and the identification of applications of the BDT in the change management process are all contributions of this thesis.

6.1 Future Work

This section discusses the BDT's future improvements and new features, extending the work done in this thesis. The first major work that could be undertaken is to measure the effectiveness of the tool. This would be accomplished through calculating *precision* and *recall* metrics, similarly to the work performed in (Rapos and Cordy 2017), on the models of our industrial partner. Additionally, an approach to regression test selection using the BDT was discussed in Section 5.3.3—this functionality is not implemented yet. Future work will implement this functionality. It would save developers valuable time when running and maintaining intensive regression test suites.

Another potential improvement of the BDT tool is tracking change propagation to network interfaces other than CAN. The tool currently tracks change propagation to CAN network interfaces only. However, other communication

protocols exist that are also used in the automotive industry, as well as by our industry partner, such as Shared Memory and LIN (Local Interconnect Network). Further, future versions of the BDT could include support for AUTOSAR-compliant embedded systems. Currently, the tool provides impact analysis for one of our industry partner's controllers: the tool utilizes a dependency matrix and a CAN mapping sheet, respectively, to trace the dependencies within the Simulink designs of the controller, with both of these items being specific to our industry partner. Support for identifying these dependencies based with representation standardized by the AUTOSAR architecture would make the tool more universally applicable to users outside of our industry partner. In order to support all these new dependencies, another possible avenue of future work is to abstract away from the metadata being used and create a traceability metamodel that could apply to multiple types of dependencies and implementations thereof. This would reduce the amount of effort in maintenance for keeping the BDT updated with newly supported dependencies.

Finally, there are two categories of major improvements to tracing performed by the BDT that could be implemented in the future. The first is extending the analysis beyond the network interfaces. The current implementation of the BDT only supports tracing of dependencies within a system of Simulink models and the network interfaces. Future versions of the BDT could support tracing beyond these network interfaces to other software systems or even to mechanical or electrical systems. Secondly, the Reach/Coreach Tool used as a model slicing engine could be improved in several ways. The Reach/Coreach Tool as used in the BDT could be extended to track the impact of a change on the timing of a model's execution. Future versions of the BDT could include

this functionality either by updating the Reach/Coreach tool to support timing impacts, or by leveraging a different engine that supports timing analysis in addition to the Reach/Coreach Tool. The Reach/Coreach Tool could also be updated to support keeping track of the impact of each individual block or signal being reached or coreached. With this information, at a significant memory cost, the cache building operation could potentially take significantly less time. Additionally, precision of the Reach/Coreach Tool could be improved by adding support for more precise tracing through Stateflow.

6.2 Closing Remarks

Model-driven development using Simulink is becoming increasingly prevalent across industries. Industrial software systems built using Simulink are becoming very large and difficult to maintain. When multiple engineers are working on changes for individual models that compose a complex embedded system, unintended interaction between changes within these models can cause significant problems and potentially even critical failures during integration. Impact analyses are needed in order to prevent these issues. A need for an automated tool that performs impact analysis on systems of Simulink models inspired the work presented in this thesis. A successful implementation of this tool was developed for integration into our industry partner's software change management process, demonstrating the relevance and usefulness of this work.

Appendices

Appendix A

User Guide

A.1 User Interface

The BDT is used via the Simulink context menu, which can be launched by right-clicking on a Simulink model. This is shown in Figure A.1.

The BDT can be used when selecting a block, a signal line, or some amount of either. The following options are available with the BDT:

1. BDT– Brings up the BDT GUI.
2. Configuration – Allows the user to set up several configuration parameters.
3. Get Signal Name – Gets the name of a signal in an unrolled feedback boundary diagram.
4. Expand Diagram Node – Expands a node in the interactive exploration boundary diagram.

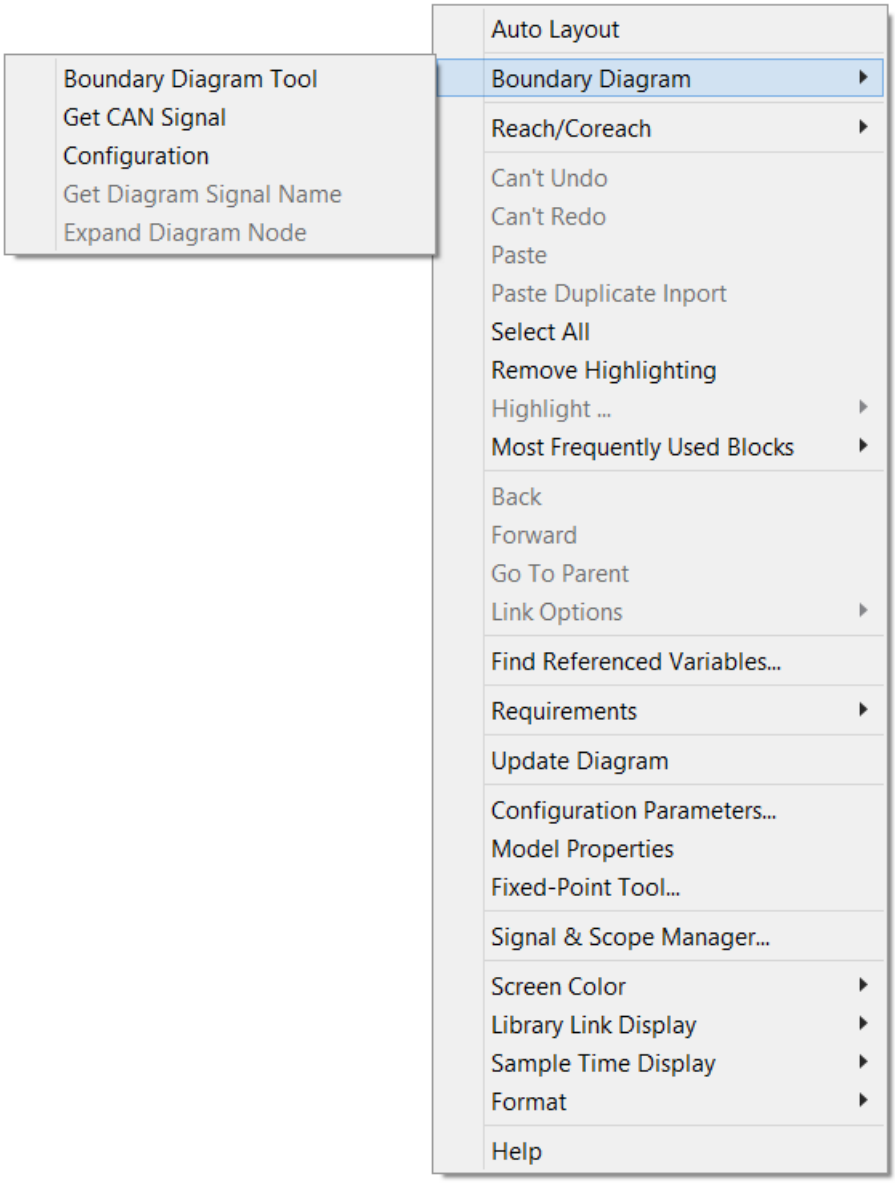


Figure A.1: *BDT* in the Simulink context menu.

A.1.1 BDT GUI

Right-clicking a selection of blocks and/or signal lines in a Simulink model and selecting the *BDT* option in the context menu will bring up the BDT GUI.

The BDT GUI has several options that modify how the tool tracks dependencies, as well as several ways to present the results of the analysis to the user:

1. Whitelist — A list of all models. Here the user chooses which model(s) to examine in order to find if the selection affects them or is affected by them, depending on the direction of the analysis.

2. Search Depth — Choose whether to find models which are immediately dependent on the selection, or are dependent on the selection via other intermediate models (full). A full analysis is still subject to the Maximum Recursive Depth, which limits the total depth of models visited in the analysis (Note that if the Maximum Recursive Depth is set to 1, and a full analysis is done, this will result in Figure 4.5b).

3. Direction — A toggle to determine whether to run the BDT's analysis in an upstream or downstream direction.

4. Analyze — Start the analysis to find the models and CAN signals that the selection affects. This populates the *Affected Models* list.

5. Affected Models — A list of models that are affected by z. Select which models are to be opened in order to have their dependent data/control flow highlighted. *Note:* It is recommended to open one model at a time.

6. Showing — Control whether the *Affected Models* list shows all affected

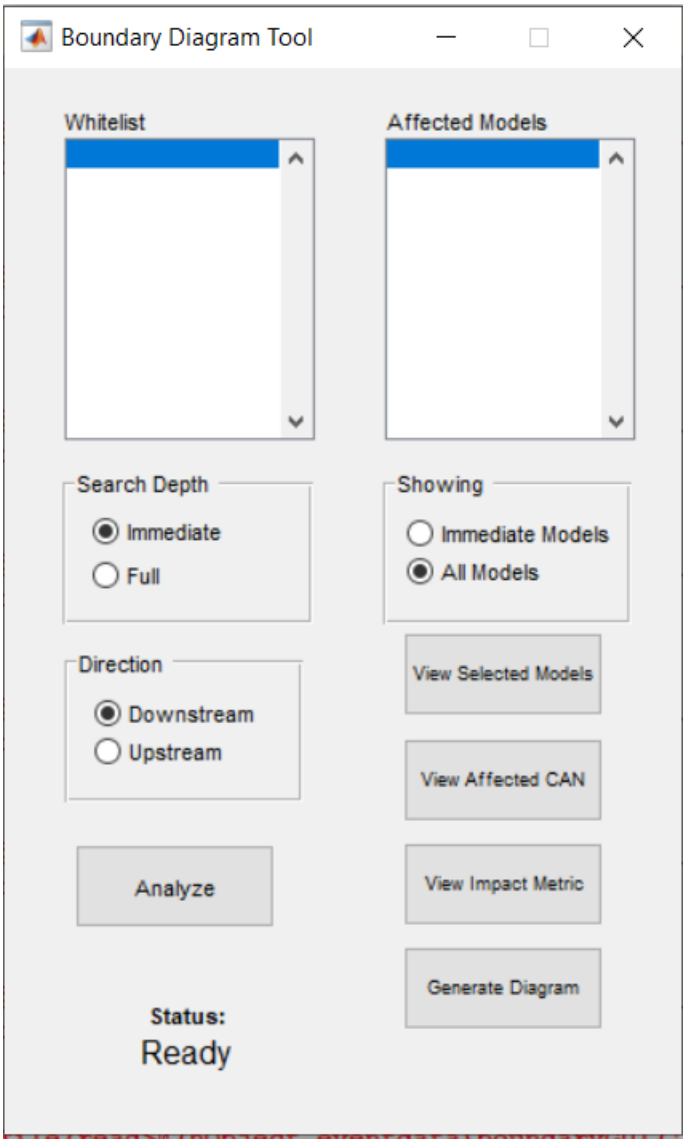


Figure A.2: The *Boundary Analysis* GUI for the BDT.

models, regardless of depth, or only immediately affected models. This option is only enabled when *Search Depth* is set to *full*.

7. View Selected Models — Open the model(s) selected in the *Affected Models* list and highlight data/control flow to show how they are affected by the selection. This button is only enabled after the analysis is complete.

8. View Affected CAN — Show the CAN signal(s) that the selection affects. This button is only enabled after the analysis is complete.

9. View Impact Metric — Show the score of the impact analysis as determined by the Impact Metric.

10. Generate Diagram — Opens a GUI for generation of boundary diagrams, demonstrating different views of the impact of the selection. For more information, see Chapter 4.

11. Status — A display of the current progress of the tool.

A.1.2 Diagram Generation GUI

After performing an analysis with the BDT the user may push the *Generate Boundary Diagram* on the GUI, which brings up this secondary GUI.

This GUI presents to the user several options pertaining to how the boundary diagram is to be generated:

1. Diagram Type — These three options represent the types of diagrams that can be generated: All Impacts, Impact to Safety, and Interactive Exploration. For more information on these types of diagrams, see Chapter 4.

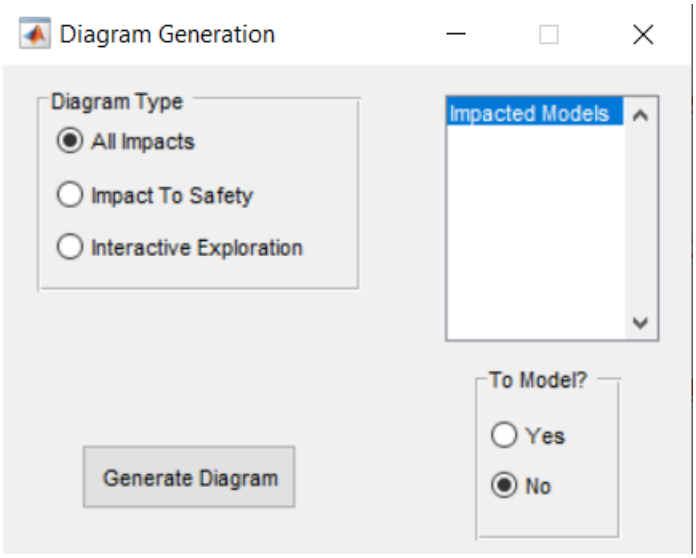


Figure A.3: The *Diagram Generation* GUI for the BDT.

2. Generate Diagram — A button that generates a boundary diagram of the type indicated in the Diagram Type button group.

3. Impacted Models — A list of models found in the impact analysis.

4. To Model? — A button group that lets the user toggle whether or not the user wishes to restrict the scope of the generated boundary diagram by only showing impacts to the selected models in the Impacted Models listbox.

A.1.3 Configuration GUI

Right-clicking on the Simulink model to bring up the context menu, and then selecting the *Configuration* option in the GUI brings up the *Configuration GUI*.

This GUI presents several options to the user that modify how the BDT operates:

- **Trace to CAN?** – Enable/disable the component of the analysis which

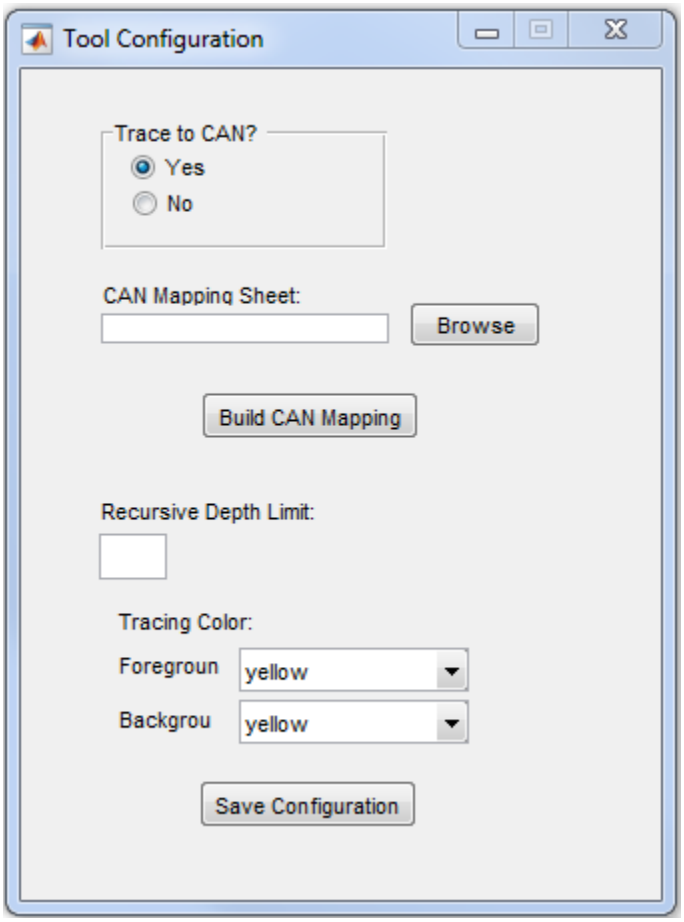


Figure A.4: The *Configuration* GUI for the BDT.

determines the impact to the CAN. Should be set to *No* if you do not have a mapping sheet available.

- **CAN Mapping Sheet** – The CAN mapping sheet being used to determine impact to the CAN, as described in Section 3.2
- **Build CAN Mapping** – The BDT uses several internal mappings to speed up its operations, as indicated in Section 3.4.2. For the CAN, the mapping should be generated once, every time a new CAN mapping sheet is used, and are saved internally by the tool to use for any subsequent CAN related operations.
- **Recursive Depth Limit** – The depth at which to stop the Analysis operation. If a full analysis is being done, this will limit the depth that the analysis will trace to other models. Setting this at 0 will indicate that there is no depth limit being used.
- **Tracing Color** – The colors to use when viewing the impact on models.

Bibliography

Bennett, K. H. (1990). "An introduction to software maintenance". In: *Information and Software Technology* 12.4, pp. 257–264 (cit. on p. 1).

Bohner, S. A. (1996). "Impact analysis in the software change process: a year 2000 perspective". In: *1996 Proceedings of International Conference on Software Maintenance*, pp. 42–51 (cit. on pp. 1, 2).

Briand, Lionel C, Yvan Labiche, and Siyuan He (2009). "Automating regression test selection based on UML designs". In: *Information and Software Technology* 51.1, pp. 16–30 (cit. on p. 65).

Charette, R. N. (2009). *This car runs on code.* `https://spectrum.ieee.org/transportation/systems/this-car-runs-on-code`. [Online; accessed June 4th, 2018] (cit. on p. 1).

Fürst, Simon et al. (2009). "AUTOSAR–A Worldwide Standard is on the Road". In: *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden.* Vol. 62, p. 5 (cit. on p. 5).

Galbo, Stephanie Perez Day (2017). "A Survey of Impact Analysis Tools for Effective Code Evolution". PhD thesis (cit. on p. 3).

GraphViz (2018). *GraphViz - Graph Visualization Software.* `https://www.graphviz.org/`. [Online; accessed June, 2018] (cit. on p. 48).

Herman, Ivan, Guy Melançon, and M Scott Marshall (2000). "Graph visualization and navigation in information visualization: A survey". In: *IEEE Transactions on visualization and computer graphics* 6.1, pp. 24–43 (cit. on pp. 6, 50).

IEC (2010). *IEC 61508 - Functional Safety of E/E/Programmable Electronic Safety-related Systems*. International Electrotechnical Commission (cit. on p. 2).

ISO (2011a). *ISO 26262-8: Road vehicles – Functional safety – Part 8: Supporting processes, International Organization for Standardization (ISO)* (cit. on pp. 55, 61, 67).

— (2011b). *ISO 26262: Road vehicles – Functional safety, International Organization for Standardization (ISO)* (cit. on pp. 2, 5, 7).

Kowalewski, Thomas Gerlitz1 Norman Hansen1Christian Dernehl1Stefan (2016). "artshop: A continuous integration and quality assessment framework for model-based software artifacts". In: *Tagungsband des Dagstuhl-Workshops*, p. 13 (cit. on pp. 4, 23, 24).

Lindland, J. L. (2007). *The Seven Failure Modes - Failure Modes and Effects Analysis*. The Bella Group, Inc. USA (cit. on p. 6).

Lindvall, Mikael (1997). "Evaluating Impact Analysis – A Case Study". In: *Empirical Softw. Engg.* 2.2, pp. 152–158. ISSN: 1382-3256 (cit. on p. 2).

MathWorks (2018a). *Isolating Problematic Behavior with Model Slicer*. `https://www.mathworks.com/products/sldesignverifier/features.html#isolating-problematic-behavior-with-model-slicer`. [Online; accessed June, 2018] (cit. on pp. 4, 22).

MathWorks (2018b). *Perform Impact Analysis*. `https://www.mathworks.com/help/simulink/ug/perform-impact-analysis.html`. [Online; accessed June, 2018] (cit. on pp. 4, 24).

McSCert (2018a). *Auto Layout Tool*. `https://www.mathworks.com/matlabcentral/fileexchange/51228-auto-layout-tool`. [Online; accessed June, 2018] (cit. on pp. 21, 26, 49).

— (2018b). *Reach/Coreach Tool*. `https://www.mathworks.com/matlabcentral/fileexchange/51180-reach-coreach-tool`. [Online; accessed June, 2018] (cit. on p. 26).

Pantelic, Vera et al. (2018). "Software engineering practices and Simulink: bridging the gap". In: *International Journal on Software Tools for Technology Transfer* 20.1, pp. 95–117. ISSN: 1433-2787 (cit. on pp. 3, 5, 10, 22, 64).

Rapos, Eric J and James R Cordy (2017). "SimPact: Impact analysis for simulink models". In: *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, pp. 489–493 (cit. on pp. 4, 24, 69).

Reicherdt, Robert and Sabine Glesner (2012). "Slicing MATLAB Simulink Models". In: *Proc. 2012 Intl Conf. on Software Engineering*. ICSE 2012. Zurich, Switzerland: IEEE Press, pp. 551–561. ISBN: 978-1-4673-1067-3 (cit. on pp. 4, 22, 23).

The MathWorks (2012). *MathWorks Automotive Advisory Board (MAAB): Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow, Version 3.0*. Version 3.0. URL: `www.mathworks.com/solutions/automotive/standards/maab.html` (cit. on p. 38).

de la Vara, J. L. et al. (2016). "An Industrial Survey of Safety Evidence Change Impact Analysis Practice". In: *IEEE Transactions on Software Engineering* 42.12, pp. 1095–1117. ISSN: 0098-5589 (cit. on p. 2).

Workgroup, E.G.A.S. (2013). *Standardized E-Gas Monitoring Concept for Gasoline and Diesel Engine Control Units* (cit. on p. 55).