Making Simulink Models Robust with Respect to Change

Making Simulink Models Robust with Respect to Change

By

Monika Jaskolka, B.Co.Sc., M.A.Sc.

a thesis

submitted to the Department of Computing and Software

and the School of Graduate Studies of McMaster University

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Doctor of Philosophy (December 2020) McMaster University

(Software Engineering) Hamilton, ON, Canada

Title:	Making Simulink Models Robust with Respect to Change
Author:	Monika Jaskolka B.Co.Sc. (Laurentian University) M.A.Sc. (McMaster University)
Supervisors:	Dr. Mark Lawford, Dr. Alan Wassyng
Number of Pages:	xi, 206

For my husband, Jason

Abstract

Model-Based Development (MBD) is an approach that uses software models to describe the behaviour of embedded software and cyber-physical systems. MBD has become an increasingly prevalent paradigm, with Simulink by MathWorks being the most widely used MBD platform for control software. Unlike textual programming languages, visual languages for MBD such as Simulink use block diagrams as their syntax. Thus, some software engineering principles created for textual languages are not easily adapted to this graphical notation or have not yet been supported. A software engineering principle that is not readily supported in Simulink is the modularization of systems using information hiding. As with all software artifacts, Simulink models must be constantly maintained and are subject to evolution over their lifetime. This principle hides likely changes, thus enabling the design to be robust with respect to future changes.

In this thesis, we perform repository mining on an industry change management system of Simulink models to understand how they are likely to change. Then, we explore the various modelling mechanisms available in the Simulink language to see how they could support modular design with information hiding. Next, we propose a module structure, syntactic interface, and modelling conventions for Simulink designs, which are supported by our

open-source Simulink Module Tool. Finally, we apply the proposed techniques on case studies from the aerospace and nuclear domains, in order to demonstrate their practicality and validate their effectiveness. Overall, the approach helped support information hiding by encapsulating secrets and facilitating likely changes. It also had a positive effect on interface complexity, cohesion, and coupling. The larger system also exhibited reductions to cyclomatic complexity, testing effort, and execution time, but the smaller case study benefited less in these areas.

Acknowledgments

My sincerest appreciation to Dr. Mark Lawford and Dr. Alan Wassyng for supervising me throughout my graduate studies. Thank you for encouraging me and providing me with incredible opportunities that enriched my studies.

Many thanks to Dr. Jacques Carette and Dr. Ryszard Janicki for serving on my supervisory committee, as well as for providing constructive input and thought-provoking questions that shaped the course of my doctoral research.

I would also like to thank my external examiner Dr. Pieter J. Mosterman for reviewing my thesis and for the interesting discussions at my defence.

Thank you to Dr. Vera Pantelic for her indispensable input and guidance. It was a pleasure to work with you over the years.

Many thanks to my colleagues and friends from the McMaster Centre for Software Certification (McSCert). I have immensely enjoyed publishing, developing tools, and working with such a dynamic group. Thank you to Stephen Scott for his collaboration on content presented in this thesis.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ACET Average Case Execution Time.

AHRS Attitude and Heading Reference System.

AL Actuator Loop.

AOP Aspect-Oriented Programming.

AUTOSAR AUTomotive Open System ARchitecture.

CLI Command Line Interface.

CMS Change Management System.

CR Change Request.

FCC Flight Control Computer.

HILC Helicopter Inner Loop Control.

HOLC Helicopter Outer Loop Control.

IMA Integrated Modular Avionics.

ISO International Organization for Standardization.

ITS Issue Tracking System.

LabVIEW Laboratory Virtual Instruments Engineering Workbench.

LVDT Linear Variable Differential Transformer.

MAB MathWorks Advisory Board.

MBD Model-Based Development.

MCDC Modified Condition/Decision Coverage.

MES Model Engineering Solutions GmbH.

MiL Model-in-the-Loop.

MISRA Motor Industry Software Reliability Association.

ms millisecond.

OOP Object-Oriented Programming.

PE Power Estimation.

PI Proportional/Integral.

PID Proportional/Integral/Derivative.

SCADE Safety Critical Application Development Environment.

SDS Shut Down System.

SDV Simulink Design Verifier.

SiL Software-in-the-Loop.

SysML Systems Modeling Language.

UML Unified Modeling Language.

WCET Worst Case Execution Time.

Chapter 1

Introduction

We begin by presenting the motivation for this research in Section 1.1. Then, Section 1.2 concisely sets forth the research questions. Section 1.3 lists the contributions that address the research questions and Section 1.4 lists the associated publications that resulted from the research. An outline of the remainder of this thesis is given in Section 1.5.

1.1 Motivation

At the beginning of the computing era, software primarily consisted of small programs, running on massive computers. Today, software has become ubiquitous. Massive programs run on multiple small devices, many of which are embedded in everyday objects. These programs can consist of over 100 million lines of code [Information Is Beautiful, 2015]. While the C language remains the most widely used programming language for embedded software [AspenCore, 2019; BARR Group, 2018], writing code by hand is prohibitively time-consuming and error-prone because of the increasing

complexity of software [Broy et al., 2014]. To address these concerns, the emerging software development paradigm of *Model-Based Development (MBD)* has gained widespread adoption for the development of embedded software via visual programming languages. Modelling languages such as Simulink [The MathWorks, 2020e] address the aforementioned problems by supporting automatic generation of C code, simulation capabilities, and verification and validation earlier in software development [Kakade et al., 2010]. Simulink has become one of the most widely used languages for modelling embedded software systems in industry [Liebel et al., 2014, 2018]. However, models also suffer from growing complexity, making them difficult to maintain and change [Dajsuren et al., 2013]. To combat the increasing complexity in textual programming languages, ideas on structured programming originated from individuals now considered to be pioneers of software engineering [Broy and Denert, 2002]. In particular, Parnas' design principle of *information hiding* and ideas on stable interfaces are fundamental to modern software engineering. Information hiding enables the building of systems that are robust with respect to anticipated changes. In developing systems according to this principle, qualities such as changeability, maintainability, understandability, and development independence are improved [Parnas, 1972a]. Therefore, information hiding is a valuable software engineering principle that should be leveraged in the MBD world.

1.1.1 Why Simulink?

There exists a wide spectrum of modelling languages. In software engineering in general, the Unified Modeling Language (UML) [Object Management

Group, 2017] is the de-facto modelling language standard. However, in the embedded software domain, where C is the most used programming language, the use of structural diagrams such as class diagrams is less relevant [Akdur et al., 2018]. In certain product domains, domain-specific architectures such as AUTomotive Open System ARchitecture (AUTOSAR) for automotive and Integrated Modular Avionics (IMA) for aerospace, are favoured over the structural diagrams of UML or the Systems Modeling Language (SysML) [Object Management Group, 2019].

The MATLAB [The MathWorks, 2020d], Simulink [The MathWorks, 2020j], and Stateflow [The MathWorks, 2020k] product family is the most widely used environment for specifying the behaviour of embedded software systems [Liebel et al., 2014, 2018]. Many major companies and agencies across automotive, aerospace, telecommunications, medical, and industrial automation industries actively use Simulink to develop their software and products (e.g., Toyota, General Motors, Tesla, NASA, U.S. Air Force, Boeing, Cochlear, Johnson & Johnson, etc.) [The MathWorks, 2020a]. The popularity of Simulink is due to its comprehensive toolset and ability to create executable designs that support simulation, early testing, and code generation. MATLAB itself is also a popular, widely used programming language in general [Spectrum, 2020], so the popularity of Simulink is also tied to its close integration with MATLAB.

Many other modelling languages exist that can be viewed as alternatives to Simulink, however they typically provide a subset of the capabilities of Simulink or are not as widely adopted. Safety Critical Application Development Environment (SCADE) [ANSYS, 2020] is a modelling language specializing in the development of safety-critical systems. When mapped to

Simulink, SCADE provides a subset of the capabilities of Simulink because it is only capable of modelling the discrete part of a system [Caspi et al., 2003; Colaco et al., 2017]. In this thesis, we too are primarily concerned with the modelling of discrete systems, however, the point is that Simulink is more a comprehensive modelling environment which likely contributes to its widespread use. Likewise, Modelica [The Modelica Association, 2020] is capable of modelling physical components, and is more comparable to the Simscape extension to Simulink. In this thesis, we focus on the modelling of controllers, rather than plant models. Laboratory Virtual Instruments Engineering Workbench (LabVIEW) [National Instruments, 2020] is a modelling language for implementing virtual measurement systems, and can be compared to the Dashboard blocks available in Simulink. Ptolemy II [University of California at Berkeley, 2020] is an alternative to Simulink, however, it is an academic tool and as such, has not seen wide adoption in industry. As a result, this thesis focuses on MBD using MATLAB Simulink.

Another reason we choose to focus on Simulink is because of an existing collaboration with an industrial partner that primarily uses this language for developing production software. We used our partner's software repository as the basis for Chapter 3.

1.2 Research Questions

This thesis endeavours to address the following research question areas:

RQ1: Model Changes. How do Simulink models change over time? Which parts of a model are particularly likely to change?

RQ2: Supporting Model Changes. How can the identified changes be better supported in Simulink? What well-known software engineering principles can be leveraged? How can the principles be supported in the Simulink environment? Are currently available language mechanisms sufficient? If the language is sufficient, how can the available mechanisms be leveraged to implement information hiding? If the language is deficient, how must it be augmented? Can tool support be provided to help support changes?

RQ3: Validation. Do the contributions resulting from **RQ2** benefit designs, developers, and others? Are they practical for real-world systems? What are the impacts on MBD designs and development activities (e.g., performance, complexity, and testing)? Are there any limitations?

The chapters in the main body of this thesis are focussed on each of these three research question areas: **RQ1** is addressed in Chapter 3, **RQ2** in Chapters 4 and 5, and **RQ3** in Chapter 6.

1.3 Thesis Contributions

The novel contributions of this thesis are outlined below.

1. **Data-supported insights into model changes over time, from an industrial change repository.** In order to propose solutions for

ensuring that Simulink models are robust with respect to change, we must first understand how they change and identify areas on which to focus. To date, there are currently no published industrial case studies that describe how Simulink models change over the span of years. As a result, we performed such an analysis on the software Change Management System (CMS) provided by an industrial partner. This provided novel insights into the changes that Simulink models undergo. The results are presented in Section 3.3, and show that models experienced an unexpected amount of changes to interfaces and model structure. These findings shaped the approach that is proposed in this thesis to make models robust with respect to change.

2. **A comparison of existing Simulink decomposition constructs.** There are several constructs with which to structure a Simulink model. Although MathWorks provides some guidance, no thorough comparison of the decomposition constructs has been done, in particular examining support for encapsulation and ultimately information hiding. As a result, we perform an in-depth analysis and present the results in Section 4. Although used more infrequently than the others, Simulink Functions proved to possess qualities that are better suited for structuring models. We chose to leverage this construct in our subsequent contributions.

3. **The creation of a Simulink module concept.** The current available literature does not clearly describe how the concept of a module can be used to structure Simulink models. We create the concept of a Simulink module in Section 5.2 in order to be able to construct models that are robust with respect to change.

4. **The definition of a Simulink module syntactic interface.** Well-defined interfaces are crucial for achieving modularity in designs. The current understanding of a Simulink model interface is deficient because it fails to reflect all of the interactions that can occur. As a result, in Section 5.3 we define the syntactic interface of a Simulink module, in order to give developers a complete understanding of the model interface.

5. **Four guidelines to address gaps in Simulink modelling standards.** Given the proposed Simulink module and interface concepts, we provide novel modelling guidelines to assist developers in utilizing the proposed approach effectively in practice. The guidelines can also be used in general, without structuring a Simulink design as we recommend in Contribution 3. The guidelines are described in Section 5.4.

6. **Two alternative Simulink modelling conventions.** We recognize that it may not always be appropriate or possible to restructure a design so that it relies on Simulink Functions, such as in earlier versions of Simulink that did not yet have this construct. For this reason, we describe two novel modelling conventions that can be used to structure a Simulink model to still support information hiding, albeit in an alternative structure. These conventions are given in Section 4.4.

7. **Two new open-source tools: the Model Comparison Utility,[1] and the Simulink Module Tool.[2]** The Model Comparison Utility was created to support the large-scale model comparisons required for

[1] https://doi.org/10.5281/zenodo.4321649
[2] https://doi.org/10.5281/zenodo.4321692

Contribution 1. Currently, MATLAB lacks functionality to support command line or programmatic querying of Simulink model comparisons. This tool supplies several necessary functions for searching and retrieving data about a model comparison. This enables us to perform model comparison analysis over thousands of models. This tool is described in Section 3.2.2. The Simulink Module Tool automates several operations that are tied to Contributions 2-4, including helping to structure existing models as Simulink modules, generating syntactic interfaces, checking adherence to the guidelines, and several others. This tool is designed to facilitate the application of our contributions in a real-world setting. Its full capabilities are presented in Section 5.5.

8. **Two case studies in the aerospace[3] and nuclear domains.** Both of these case studies were undertaken to apply and evaluate the contributions on real systems of varying sizes and from two different sources. They both found that our proposed module structure for Simulink designs resulted in models that were more robust with respect to change. Several other related criteria were evaluated such as design equivalence, coupling and cohesion, cyclomatic complexity, testability, and performance comparison, to better understand the impact of our proposed Simulink module structure. The aerospace case study is presented in Section 6.2. The nuclear case study is presented in Section 6.3.

[3] https://mathworks.com/matlabcentral/fileexchange/56056-do178_case_study

I was the primary creator/contributor of each of these contributions, aside from what follows. Towards Contribution 8, I performed the first proof-of-concept application on a portion of the nuclear case study, the results of which I described in [Jaskolka et al., 2020b]. With my direction, an undergraduate student assisted with the full treatment of the nuclear system. This was followed by the aerospace case study, which I also guided and described in [Jaskolka et al., 2020c].

I was the primary developer of the tools listed as Contribution 7. Two students later contributed to these tools by implementing bug fixes and enhancements. All contributors and their commit histories towards these tools are viewable on GitHub.

1.4 Related Publications and Submissions

Below is a list of the publications related to the research presented in this thesis. I am the first author on these publications.

- Bialy, M., Pantelic, V., Jaskolka, J., Schaap, A., Patcas, L., Lawford, M., and Wassyng, A. (2016). Software engineering for model-based development by domain experts. In Griffor, E., editor, *Handbook of System Safety and Security*, chapter 3, pages 39–64. Elsevier, Cambridge, MA, USA, 1 edition

- Jaskolka, M., Pantelic, V., Wassyng, A., and Lawford, M. (2020a). A comparison of componentization constructs in Simulink. In *SAE Technical Paper*, number 2020-01-1290, pages 1–16. SAE International

- Jaskolka, M., Scott, S., Pantelic, V., Wassyng, A., and Lawford, M. (2020c). Applying modular decomposition in Simulink. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 31–36

- Jaskolka, M., Pantelic, V., Lawford, M., and Wassyng, A. (2021). Repository mining for changes in Simulink models (In Preparation)

- Jaskolka, M., Pantelic, V., Wassyng, A., and Lawford, M. (2020b). Supporting modularity in Simulink models. arXiv:2007.10120 (In Preparation)

1.5 Thesis Outline

The thesis is organized into six chapters, with Chapter 1 being this introduction. Chapter 2 introduces the background concepts used throughout this work and establishes our understanding of relevant design principles. Chapter 3 presents the methodology and results for mining a CMS for Simulink model changes. Chapter 4 presents the results of a comparison of available Simulink model decomposition constructs. Chapter 5 presents an approach for supporting modular design in the Simulink language while leveraging information hiding and syntactic interfaces. Chapter 6 describes the methods, results, and challenges in performing two case studies to validate the contributions. Chapter 7 concludes with a summary and directions for future work.

Chapter 2

Preliminaries

In this chapter we introduce basic concepts used throughout this work. Section 2.1 summarizes some software engineering principles and techniques for supporting likely changes (hardware changes, behaviour changes, or software design decision changes) that are touched on in this work. Section 2.2 presents various Simulink language concepts, while Section 2.3 maps them to the C language.

2.1 Principles for Supporting Likely Changes

Information hiding is a seminal concept in software engineering, influencing ideas of structured design and object-oriented design. This section presents key concepts related to information hiding, which, although used extensively throughout literature on software design principles, have no universally accepted definitions. In particular, information hiding, modularity, separation of concerns, and encapsulation are commonly used interchangeably when discussing software design modularity. Although

related, they delineate different concepts. To avoid misinterpretation, we establish their definitions as we understand them and as we will use them throughout this work.

2.1.1 Information Hiding

Information hiding is a software design principle that provides criteria for performing system decomposition [Parnas, 1972b; Parnas et al., 1985]. More specifically, when dividing a system into a number of separate modules, information hiding seeks to do so such that modules each localize, or "hide," a *likely change* from the rest of the system. Likely changes of a system arise because of hardware changes, behaviour changes stemming from a change in requirements, or changes in software design decisions (e.g., change in data structure) [Parnas et al., 1985]. As a result, the main categories of modules are behaviour-hiding, hardware-hiding, and software design decision modules [Middleton and Sutton, 2005]. These likely changes are called "secrets" because they should remain a secret from outside the module. This is achieved by ensuring that the interface of the module does not reveal the module's inner workings through the passing of information that is too detailed, arbitrary, or potentially changeable [Parnas, 2002]. The interface must be designed such that it does not change when the module secret is modified [Parnas, 2003]. Information hiding enables the division of labour, as well as improves a system's reusability, maintainability, complexity, and testability [Korson and Vaishnavi, 1986; Parnas, 1972b; Parnas et al., 1989].

2.1.2 Modularity

A module is a component of a software system. It is a set of closely related programs and data objects [Parnas, 2018, 1972b]. A module can also be thought of as a work assignment for an individual or a group [Parnas, 2018; Parnas et al., 1989; Parnas, 1972a]. Designing a system that is modular means that the system is decomposed into simpler pieces, or modules [Ghezzi et al., 2002]. Using information hiding as the principle to guide the decomposition process will add the stipulation that each module of the system should implement a single design decision or likely requirements change (i.e., secret), and keep it hidden from the rest of the system.

2.1.3 Encapsulation

There are various definitions of encapsulation [Berard, 1993]. The simplest of these definitions states that encapsulation groups data and functions together into a single module [Schach, 2010], but does not describe the nature of the module, such as the degree of visibility of its internals (i.e., whether the module is a black, grey, or white box). In practice, encapsulation as supported by different programming languages, comes with varying degrees of visibility. The IEEE standard on vocabulary [International Organization for Standardization, 2010] also supports definitions of encapsulation that include notions of external interfaces and access restriction.

The terms encapsulation and information hiding are commonly used interchangeably, however, there is certainly a distinction between the two. Firstly, information hiding is a principle for how to go about decomposing a system into modules and encapsulation does not address this. Encapsulation

is a mechanism towards achieving information hiding, however, information hiding describes *what* it is that should be encapsulated. Secondly, information hiding states specifically that the internal module data is to be hidden from other modules, whereas the definitions of encapsulation are ambiguous in the most general case. Of course, how much data is able to be hidden is dependent on the language, and even so, can potentially be by-passed by the developer with bad design. In this work, we will adopt the definition of encapsulation that describes it as a means of grouping data and functions together as well as restricting access in order to hide implementation internals.

2.1.4 Separation of Concerns

Separation of concerns is a design principle first introduced by Edsger Dijkstra [Dijkstra, 1972, 1982]. Software *concerns*, such as features and properties, are isolated and addressed separately [Ghezzi et al., 2002; Sommerville, 2015]. Information hiding leads to separation of concerns [Parnas et al., 1989]. Information hiding treats likely changes as the concerns and, as with encapsulation, goes further by ensuring that concerns are in separate black-box modules.

2.1.5 Object-Orientation

The object-oriented approach models the logical units of a system via classes and objects, with inheritance relationships between the classes [Booch, 2004]. Specifically, classes are used to express modules in the design. The major difference between the classes and modules is that a class can be instantiated

as one or more objects. Information hiding states that the classes should contain a secret. Not only should it be grouped in a class, but the interface of the class should prevent access to the secret. In object-oriented design, the containment or scope of classes impacts access from outside the class, as well as the class interface.

2.1.6 Aspect-Orientation

Aspect-Oriented Programming (AOP) was first introduced by Gregor Kiczales [Kiczales et al., 2001]. Its aim is use *aspects* to encapsulate *crosscutting concerns* that cannot be modularized using procedural or object-oriented approaches. Examples of crosscutting concerns include logging, security, exception management, and others that usually impact several modules of an application. Although AOP is an approach for decomposing systems, we do not focus on it in this work.

2.2 Simulink

Simulink [The MathWorks, 2020e] is a visual programming language for creating time-based block diagrams. The Simulink language and its accompanying Simulink development environment are an add-on to the MATLAB environment. Several Simulink language metamodels exist (e.g., [Dajsuren et al., 2013; Legros et al., 2010; Amelunxen et al., 2008]). Figure 2.1 is a simple version that highlights the elements that are used throughout this work. The Simulink language is fundamentally comprised of the elements shown in Figure 2.1. A block diagram is a representation of a model. It contains blocks, that represent various mathematical concepts

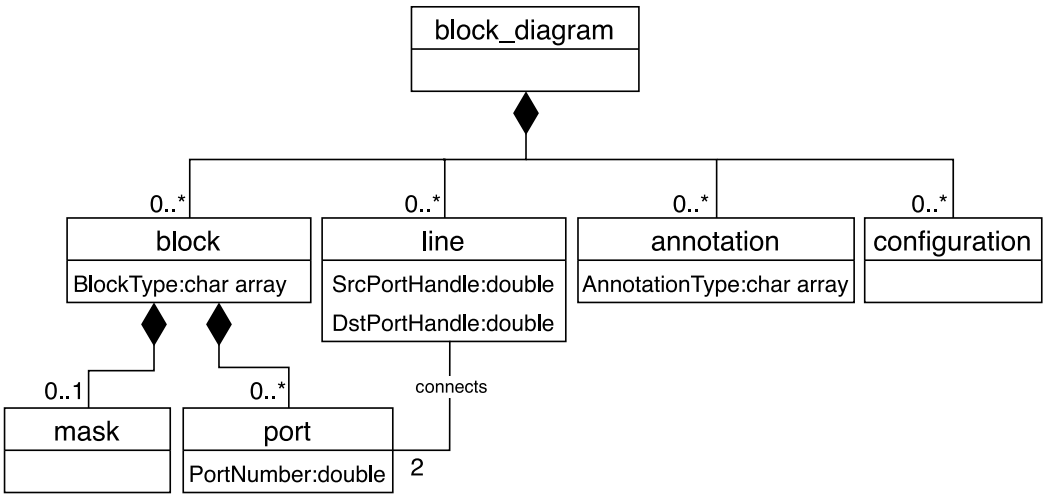


Figure 2.1: Simplified Simulink metamodel.

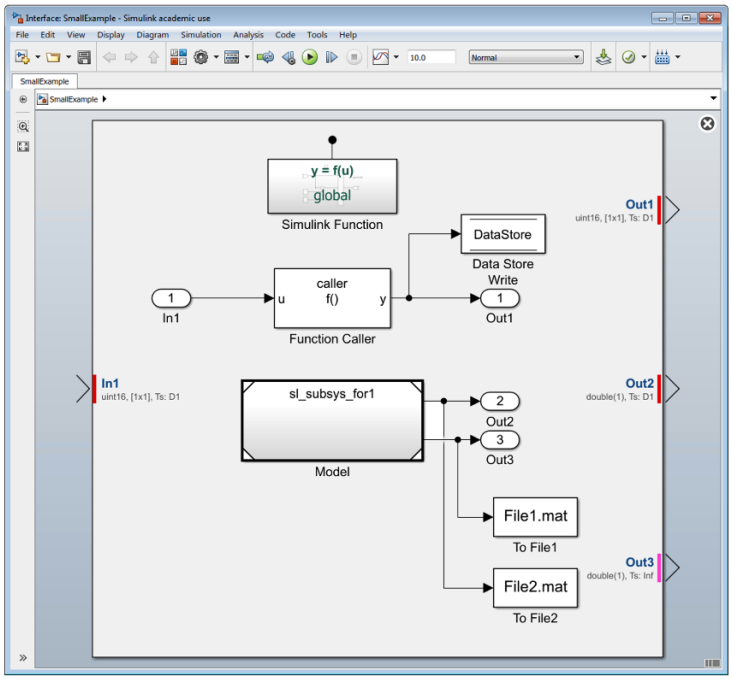
(e.g., unit delay, gain, subsystem), provided by a built-in block library. The Simulink block library provides 401 unique blocks. Blocks are further specified via parameters. For example, the *Value* parameter of a constant block is set to a number that specifies the block's outputted value. Other parameters that describe the block are read-only parameters. For example, each block has a *BlockType* parameter that describes the kind of block it is. The Simulink block library contains 150 unique block types. Blocks can also be composed of a mask, which provides a customized appearance for that block. Bocks are connected together via lines representing the passing of data or control. Blocks usually have ports, which is where a line connects blocks together. A block diagram can also be commented on through the use of annotations. Block diagrams can be further customized through the use of configurations, which specify parameters for simulation, diagnostics, and other properties. The frequency with which blocks, lines, annotations, etc. occur in Simulink models is discussed in Section 3.3.1.

A block diagram is stored as a Simulink model in either an `.mdl` or `.slx` file. A Simulink model is the primary design artifact of a Simulink system. Figure 2.2a shows a simple example model that computes the sum of positive integers using three methods: using Product and Sum blocks (Figure 2.2b), using a For Iterator Subsystem (Figure 2.2c, top), and using the Fcn block with textual operations (Figure 2.2c, bottom).

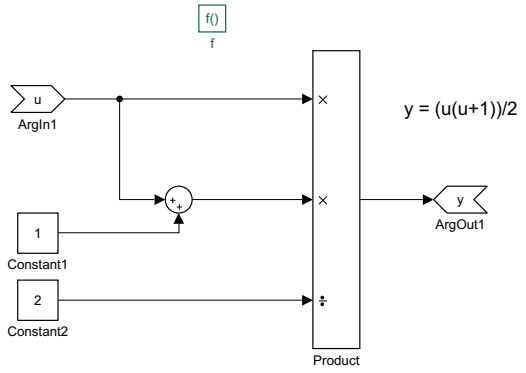
In the following sections we describe in further detail the Simulink blocks and other concepts that are integral to this work. We also provide analogies for some Simulink constructs to the standard C language (C18) [IEEE, 2018] to better understand Simulink and to eventually draw comparisons between their design principles. This analogy is discussed further in Section 2.3.

2.2.1 Subsystems

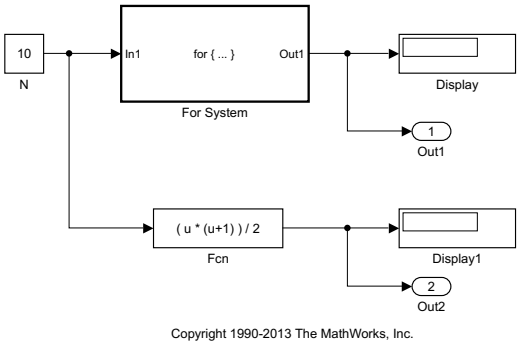
In general, subsystems are types of blocks, with their *BlockType* parameter set to *SubSystem*. There are many different kinds of subsystems, but they all visually organize a model by grouping together a part of a model's design. A subsystem also introduces hierarchical layering (or levels), where its contained elements are considered to be in a lower level of the hierarchy. All the kinds of subsystem blocks available in Simulink are shown in Figure 2.3, with a classification of their intended use on the left-hand side of the diagram. There are many built-in varieties of subsystems that not only add hierarchy, but also provide additional semantics, especially with respect to the execution of the model. For example, the For Iterator Subsystem is an Atomic Subsystem that also acts like a *for* loop, repeatedly executing its contained elements for a number of iterations. Another example is the Code



(a) A Simulink model (in Interface Display view).



(b) `Simulink Function` subsystem from Figure 2.2a.



(c) Model `sl_subsys_for1` referenced in Figure 2.2a.

Figure 2.2: A simple Simulink example computing integer sums.

Reuse Subsystem, which is simply an Atomic Subsystem with its *Function packaging* set to *Reusable function*. We do not elaborate on these subsystems, as they are too specialized to be pertinent for our purposes. They can be thought of as Atomic Subsystems, with additional semantics for iteration, branching, etc. We are primarily interested in subsystems that can be used for modularization purposes.

Subsystems can be viewed as either *virtual* (Section 2.2.1.1) or *nonvirtual* (Section 2.2.1.2) [The MathWorks, 2020j]. A virtual subsystem is any subsystem block with the *IsSubsystemVirtual* parameter set to *on*, while for a nonvirtual subsystem it is set to *off*. A virtual subsystem's boundaries are ignored when the block execution order is determined, while a nonvirtual subsystem is executed as a unit. These two groups of subsystems are delineated in Figure 2.3.

2.2.1.1 Virtual Subsystem

A virtual subsystem visually groups together a portion of the design, however, it does not have any behavioural impact on the model nor the generated code. It is simply a visual convenience for developers. The default subsystem (see Figure 2.3) is a virtual subsystem. It is the most commonly used kind of subsystem, so we refer to it simply as a Subsystem henceforth. The Simulink engine expands virtual subsystems in place before the execution of the model [The MathWorks, 2020j], akin to a C preprocessor expanding a macro—but one time only.

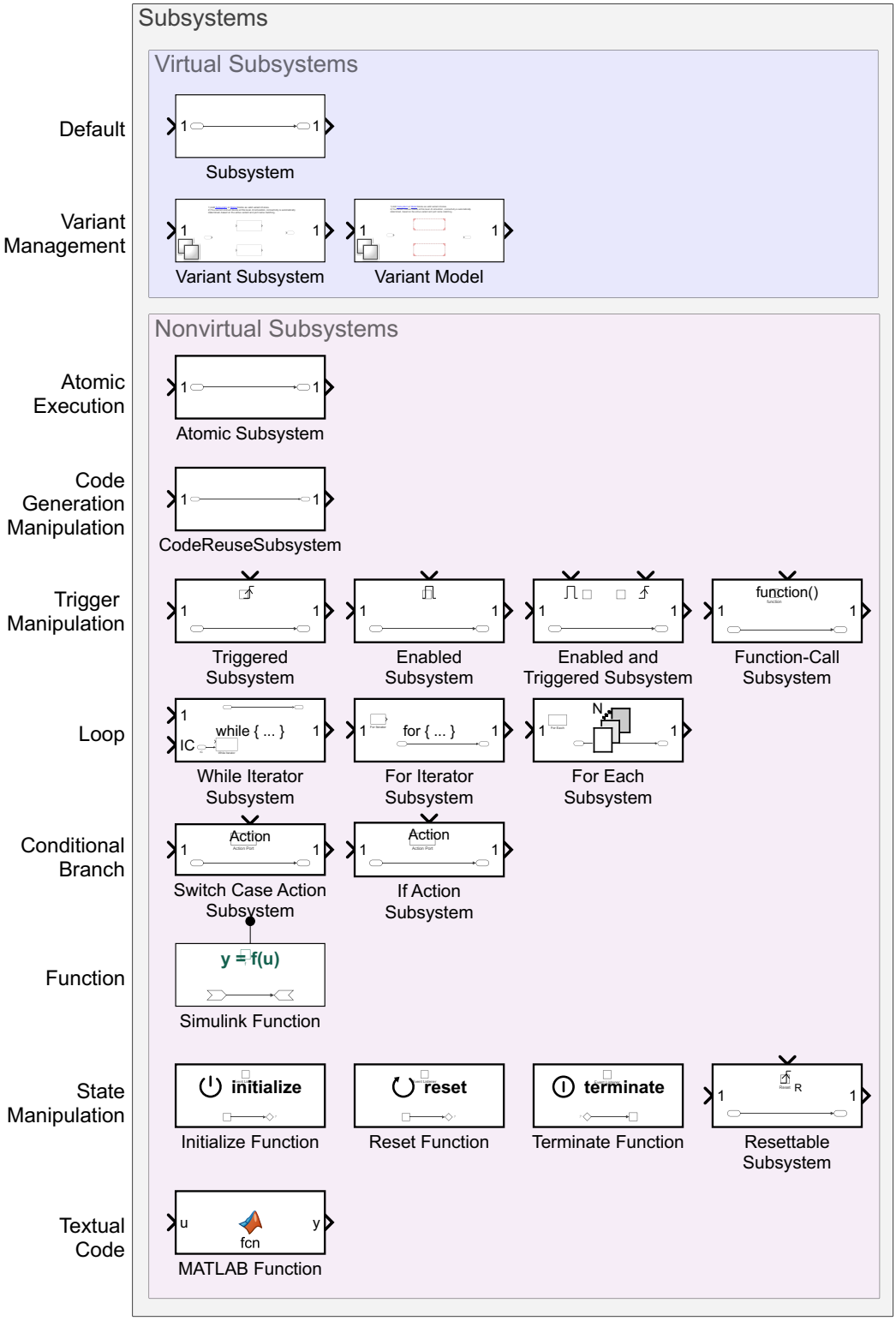


Figure 2.3: Summary of subsystems, classified as virtual and non-virtual.

2.2.1.2 Nonvirtual Subsystem

Although there are many nonvirtual subsystems in Simulink, we firstly focus on the Atomic Subsystem. An Atomic Subsystem is the most basic nonvirtual subsystem as it only has its *Is Subsystem Virtual* parameter set to *off*. All other nonvirtual subsystem blocks are variations of an Atomic Subsystem (e.g., with additional parameters enabled or containing special blocks that change behaviour), and/or are not pertinent to the goal of this evaluation (e.g., textual implementation, loops). A special type of nonvirtual subsystem is the Simulink Function, which is described below.

Atomic Subsystem An Atomic Subsystem is a Subsystem block with its *Treat as atomic unit* parameter enabled. This ensures that the blocks in the Atomic Subsystem are executed as one unit. The majority of other nonvirtual subsystem blocks have additional abilities to control their execution in various ways (e.g., Simulink Function, Triggered Subsystem, For Iterator Subsystem, etc.).

Simulink Function A Simulink Function is a kind of reusable subsystem. It was first introduced into Simulink in version R2014b. It can receive input via the conventional Inport and Outport blocks, as well as through input/output arguments via the ArgIn/ArgOut blocks. It can be invoked several ways: graphically via Function Caller blocks or using its prototype in textual contexts, such as MATLAB Function blocks or Stateflow chart transitions. As a result, a Simulink Function's advantage is that it need not be connected via signal lines to be invoked, and can be called many times from multiple locations. A simple example of a Simulink Function is shown in Figure 2.2a, where the Simulink Function `y = f(u)` is executed when its

Function Caller block is executed. Figure 2.2b shows the contents of the function.

A Simulink Function has a *Function Visibility* parameter (introduced in R2017b) which, in conjunction with its placement in the model, limits the accessibility, or scope, of the Simulink Function. The parameter can be set to either *scoped* or *global*, but by default it is scoped [The MathWorks, 2020j]. The rules for determining the scope are summarized in Table 2.1. An illustrative example is also provided in Figures 2.5–2.8, where there are three models: $model_1$ which contains a Simulink Function, $model_2$ which has a model reference to $model_1$, and $model_3$ which $model_1$ references. The Simulink Function symbol displays the prototype of the function (i.e., `fcn()`) and this symbol's placement in the model tree denotes the hierarchical placement of a Simulink Function in a model. The way in which the function can be called from the various subsystems/models where it is accessible is printed in the subsystem nodes. Subsystem nodes are empty if the function is not accessible from that location in the model. The legend is provided in Figure 2.4.

Case 1 (Figure 2.5) A Simulink Function with *global* function visibility can be placed anywhere in a model and will be accessible for external use in the model hierarchy. We see that although the Simulink Function is in ss_2, it is available in $model_1$ and any model in which it is referenced (in this example, $model_2$).

Case 2 (Figure 2.6) The Simulink Function has *scoped* visibility. As a result, its placement in the model affects its accessibility. In this case, it is placed in the root system. Thus, it is externally accessible in the model hierarchy,

Table 2.1: Summary of Simulink Function scope.

Case	Placement	Function Visibility	Scope
1	*Don't care*	Global	External to Model
2	Root	Scoped	External to Model
3	Virtual Subsystem	Scoped	Internal to Model
4	Nonvirtual Subsystem	Scoped	Internal to Subsystem

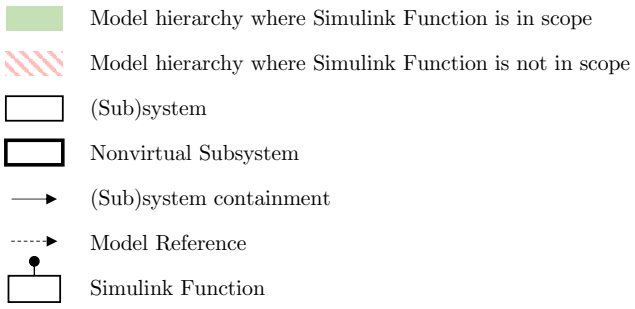


Figure 2.4: Legend for Figures 2.5–2.8.

specifically $model_2$. The difference between a global Simulink Function and a scoped Simulink Function placed at the root is in the way it is called. In the latter, the function name must be qualified with the model reference block name, as shown in the subsystems in $model_2$.

Case 3 (Figure 2.7) If the scoped Simulink Function is placed in a subsystem it is accessible in the parent subsystem and any descendants. In this figure, the Simulink Function is not available above its parent subsystem. This is denoted by the lined background.

Case 4 (Figure 2.8) A scoped Simulink Function placed in another nonvirtual subsystem can be accessed only within that subsystem. Here we see that ss_2 is a nonvirtual subsystem, and as a result, the function can only be called in ss_2 and below. Placing a global function in a nonvirtual

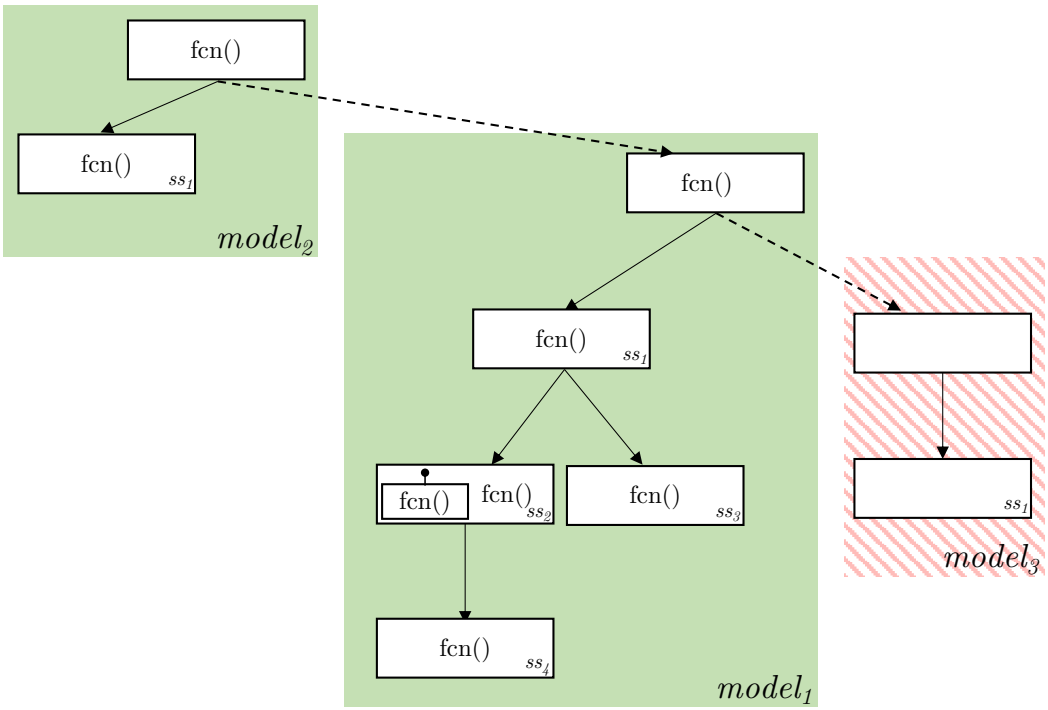


Figure 2.5: Case 1—*Function Visibility* of the Simulink Function is global. Therefore, it is available in the model hierarchy.

subsystem is not permitted, per the Simulink language rules [The MathWorks, 2020j].

The concept of a Simulink Function is analogous to a function in C, with some semantic differences. While C functions are external by default, Simulink Functions are scoped by default. In C, one can use functions from a different source by including the header file. To use a Simulink Function from a different model, that model must include a Model Reference and the function must have external scope. C `static` functions support modularity by restricting the scope of design details. In this case, the function's name is invisible outside of the file in which it is declared, and is analogous to a local Simulink Function.

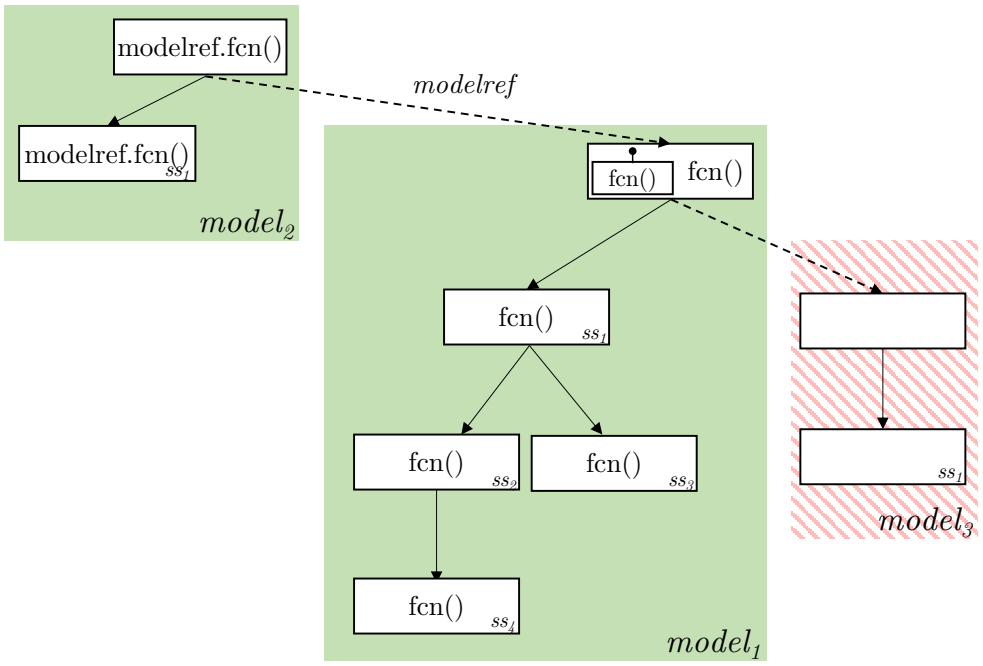


Figure 2.6: Case 2—*Function Visibility* of the Simulink Function is scoped and it is placed at the root. Therefore, it is available in the model hierarchy. Outside of the model its name is qualified.

2.2.2 Library

A Library is a special kind of block diagram (or model) which contains a collection of blocks. The contained Library blocks serve as prototype blocks that can be repeatedly instantiated in many models. When used in a model, the blocks act as a reference or link to the source Library block, and are updated pre-compile time. Any updates to the Library will propagate into the models that use the linked blocks. In general, one can place any subsystem block into a Library in order to make it reusable in multiple instances. A library block is akin to a normal (multi-use) macro in C.

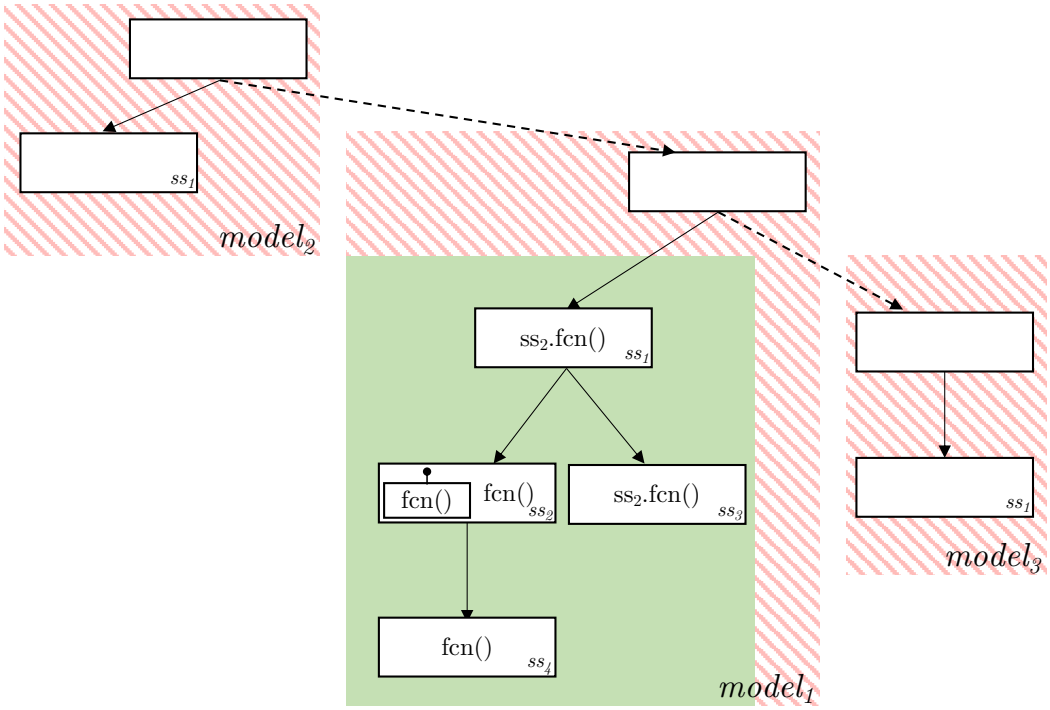


Figure 2.7: Case 3—*Function Visibility* of the Simulink Function is scoped and it is placed in a subsystem. Therefore, it is only available in the parent subsystem and its descendants.

2.2.3 Model Reference

A model can be directly referenced from another model using a Model Reference block. This effectively adds the contents of one model into another. Using a Model Reference is a way that entire models can be made reusable. Like a Library, any subsystem can be made reusable if placed in a model that is referenced in multiple locations. The difference is that the entire model is referenced, instead of only a particular block as in the case of a Library. In a similar way that subsystems introduce layers in a model, each Model Reference also introduces hierarchy [The MathWorks, 2020j]. Figure 2.2a shows a Model

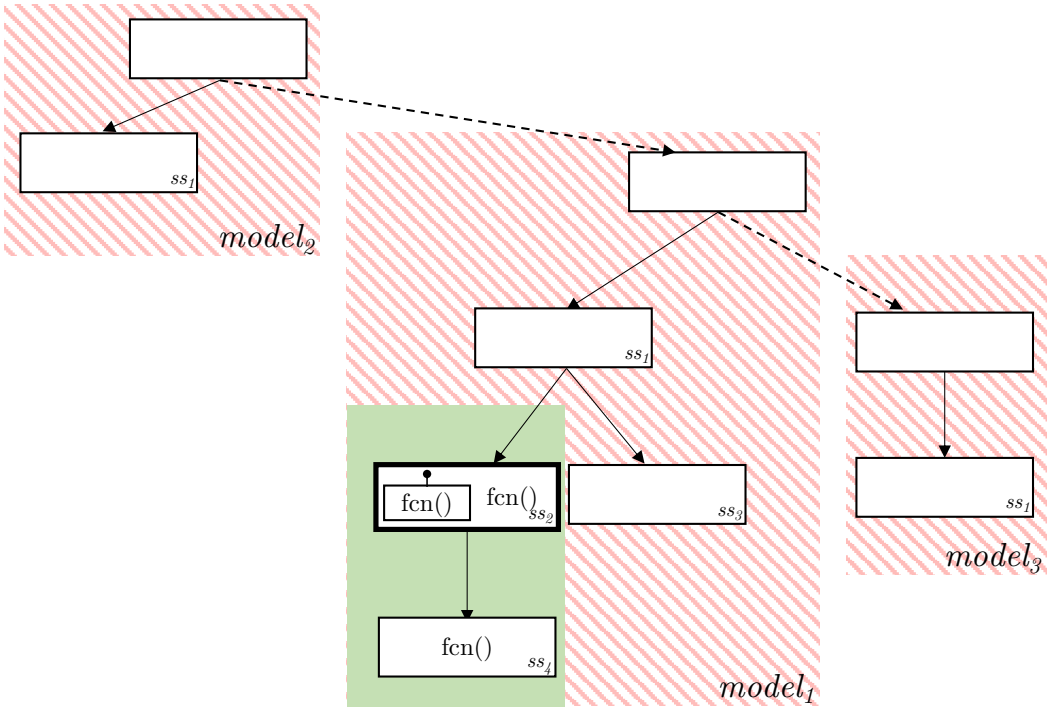


Figure 2.8: Case 4—*Function Visibility* of the Simulink Function is scoped and it is placed in a nonvirtual subsystem. Therefore, it is only available in the subsystem and its descendants.

Reference to the (built-in) MathWorks model `sl_subsys_for1`, displayed in Figure 2.2c.

The use of a model reference is similar to the C preprocessing directive `#include`. A referenced model is modelled independently, and is code generated separately from the models in which it is referenced. A model reference in some parent model will make available all exported Simulink functions (see Section 2.2.1.2) to that parent model in a similar way that including a C header will give access to externally defined functions and variables.

2.2.4 Data Passing

The passing of data in a Simulink model is represented using signal lines. However, constructs such as Goto/From pairs and Data Store Memory/Data Store Read/Data Store Write blocks enable the passing of data implicitly throughout a model, without a direct line connection. This is known as hidden data passing because data can cross certain block boundaries (e.g., Subsystems) without being immediately evident [Bender et al., 2015].

In C, variable names represent stored data, while in Simulink data is represented with (named or unnamed) signal lines. In C, there are ways we can store and move data that are difficult to trace, using pointers for instance. As mentioned previously, Simulink has a variety of ways of storing and passing data that, if not used carefully, make it difficult to understand a model's data flow. For that reason, it is important to define interfaces of modules to improve their understandability.

2.2.4.1 Data Store Memory

A data store is named memory that can be written to with signal data, and as well as read. A data store can either be local to the model, or externally defined and global to the model. There are three blocks for modelling data store memory that is local to the model:

- Data Store Memory – Declaration of a named memory region.
- Data Store Read – Obtains data from the Data Store Memory.
- Data Store Write – Writes data to the Data Store Memory.

The placement of a Data Store Memory block in the model affects the ability for its Data Store Read/Data Store Write blocks to access it. If the Data Store Memory is placed in the top-level system, it is accessible within the entire model. If placed in a Subsystem, it is accessible in the Subsystem and all layers below it. In all cases where a Data Store Memory block is used, the scope of the data is confined to the model in which it resides. Data store memory can also be defined globally in the base workspace or a data dictionary as a *Simulink.Signal* object, instead of a Data Store Memory block. This is the usual use for production embedded code development. Figure 2.2a contains a Data Store Write that passes data outside of the model.

2.2.4.2 Goto, From, and Goto Tag Visibility

Instead of using signal lines to pass data, Goto and From blocks can be used. Data fed into a Goto is passed implicitly to all From(s) with the same *Tag* parameter. A single Goto block can have multiple From blocks, but a From may only receive data from a single Goto. The *Tag Visibility* parameter of a Goto allows the scope of where the signal can be accessed in the model by From blocks to be altered. This includes:

- local – Accessible in the same level of the hierarchy (default).
- scoped – Accessible to all levels at or below the Goto Tag Visibility block.
- global – Accessible to all levels of the model.

2.2.5 Workspaces and Data Dictionaries

Information that contributes to the specification and simulation of a Simulink model can exist outside of the block diagram that forms a model, that is,

in the MATLAB (base) workspace, model workspace, and/or one or more data dictionaries [The MathWorks, 2020j]. The MATLAB workspace is for temporary data storage. For permanent data storage, a model workspace is used to associate data with a model, or a data dictionary can store persistent data in a separate `.sldd` file associated with one or more models. Workspaces and data dictionaries are similar to a C module dedicated to storing/defining external variables.

2.2.6 Exporting Data

The To Workspace block writes signal data to the base workspace as a variable.[1] The To File block writes the signal data into a `.mat` file. Two To File blocks can be seen in Figure 2.2a. Data is written incrementally to the file throughout simulation, and if the file already exists, data is overwritten. These blocks are analogous to the standard file output functions in C.

2.2.7 Stateflow

Stateflow [The MathWorks, 2020k] is another graphical language that is an add-on to MATLAB or Simulink. It has separate syntax and semantics from Simulink that includes state transition diagrams, state transition tables, and truth tables. Full treatment of the Stateflow language is left to future work.

[1] The exception is a MATLAB function using a *sim* command to simulate a model. In this case, the signal data is written to the calling function workspace.

2.3 C to Simulink Concept Mapping

In this section we draw an analogy between the Simulink and C languages in order to better understand Simulink and to eventually draw comparisons between their design principles. Surveys routinely list C as one of the most widely used programming languages for embedded software [Spectrum, 2020; AspenCore, 2019; BARR Group, 2018]. It is a language in which software design principles have been thoroughly studied and applied over the years. Although it does not support Object-Oriented Programming (OOP), modular programming in C is a well-known technique [Kochan, 2014; Reddy and Ziegler, 2009; Qian et al., 2009; Oualline, 1997]. Similarly, Simulink is a very popular modelling language in the embedded domain, and it too does not support OOP. However, modular design for Simulink has not yet been introduced. For this reason we believe there is a strong basis for comparison between the two languages. Moreover, in our experience in industry, most developers of Simulink models also work closely with C, because models are generated into C code, and it is common practice to inspect model changes at the C level. Therefore, explaining concepts with respect to C is an effective way of introducing ideas that are easy for engineers to understand. Note, we do not discuss how a model is generated into C code, but rather position a model as the primary design artifact in Simulink, in the same way source files are in C. We are interested in the design-time view of the software, rather than the compile-time view.

The comparison between C and Simulink constructs, as explained throughout Section 2.2, is summarized in Table 2.2. A Simulink model (`.mdl`/`.slx`) is comparable to a C source file (`.c`). However, there is no

Table 2.2: Comparison of C and Simulink constructs.

C	Simulink
Source file	Model
Header file	*Not Available*
Include	Model Reference
Function	"Global" Simulink Function (Case 1)
Member function	"Scoped" Simulink Function (Case 2)
Static function	"Local" Simulink Function (Case 3 & 4)
Macro (single-use)	Virtual Subsystem
Macro	Library
Variable	Data Store Memory
Goto/Label	Goto/From
External data definitions	Workspace/Data Dictionary data

notion of a header file (`.h`) and the interface that it provides. This is because of the top-level block diagram not providing sufficient information about the interface. This issue is explored in more detail in Section 5.3. Given this mapping, Simulink Functions can be used with other Simulink constructs to support modularity in a way that facilitates *information hiding*. This is further described in Chapter 4.

2.4 Chapter Summary

This chapter summarized software engineering principles and techniques for supporting likely changes. We will focus on information hiding and modularity in subsequent chapters. We also described Simulink and C concepts that will be used throughout this thesis.

Chapter 3

Model Changes

In Simulink, models are the primary design artifact and, as with all software, they must be constantly maintained and evolved over their lifetime. It is necessary to develop models that support likely changes in order to assist with development and maintenance processes. Thus, the types of frequently performed changes must be understood, and appropriate language mechanisms must be available to support these likely changes. However, Simulink model changes are currently not well understood. In this chapter, we analyze an industrial software repository and its version control system to provide insights into the likely changes for Simulink.

Section 3.1 presents the current state of the literature. Section 3.2 presents the methodology and tools used. Section 3.3 explains changes in Simulink models in general, and then presents several more focussed analyses of changes. Section 3.4 provides a summary of the results.

3.1 Related Work

Much work has been done to classify the changes of software in general. Taxonomies of software evolution have been predominantly based on evaluating the purpose of a change. Swanson [Swanson, 1976] created a typology of three reasons for change: corrective (because of failures), adaptive (because of changes in data or environment), and perfective (to enhance performance, efficiency, etc.). The International Organization for Standardization (ISO) added a fourth type: preventive maintenance, for dealing with latent faults. The typology was expanded to a taxonomy of 12 types of software changes, grouped into four clusters: *support interface* (evaluative, consultive, training), *documentation* (updative, reformative), *business rules* (enhancive, corrective, reductive), and *software properties* (adaptive, performance, preventive, groomative) [Chapin et al., 2001]. In this work, we study software changes at a more basic level. We are less concerned with the purpose of the change and more with what elements are changing.

There is much work on identifying changes in software systems developed with textual programming languages at the source file level [Robbes et al., 2008; Canfora and Cerulo, 2005; Ying et al., 2004], program element level [Zimmermann et al., 2005], and code statement level [Giger et al., 2012; Rysselberghe and Demeyer, 2004]. Other work is targeted at identifying clones [Abd-El-Hafiz, 2012], predicting faults/defects [Kim et al., 2008; Hata et al., 2010; Yatish et al., 2019], predicting vulnerabilities [Scandariato et al., 2014; Neuhaus et al., 2007], etc. While these focus on textual code, we are concerned with graphical models, prior to code generation. This is because in

MBD, the code is volatile and not the primary design artifact with which developers interact.

Work is limited when it comes to categorizing changes for Simulink models, particularly on how models evolve. Much of the work that has been published is from a variant management perspective. There has been work on detecting clones and variation points, and different authors classify Simulink model variabilities in different ways. Ryssel et al. identify three variations across models: adding/removing components, adding/removing connections, and changing parameter values [Ryssel et al., 2010]. Schlie et al. focus on two changes: adding blocks and adding hierarchical levels (i.e., inserting Subsystem blocks) [Schlie et al., 2017]. Haber et al. specify delta operations as the addition, removal, modification, or replacement of elements, where the elements in question are subsystems, connections, ports, and model references [Haber et al., 2013]—a small subset of the Simulink language. Alalfi et al. propose a taxonomy of Simulink mutations in order to support the injection of clones into models, and later define a set of variability operators for clone detection in a single model [Alalfi et al., 2014]. Mutations include layout modification, renaming blocks/lines, adding/deleting blocks, changing block parameters, and Subsystem grouping [Stephan et al., 2014]. The derived variability operators categorize variations into five categories: block variability, input/output variability, function variability, layout variability, and subsystem name variability [Alalfi et al., 2019]. The results in that work were empirically derived from six demonstration models from MathWorks. This is useful because it illustrates how a group of experts modified relatively small examples. We also need to

analyze changes that occur in industry projects, where models can be comprised of hundreds of thousands of blocks, and thousands of subsystems.

Current classifications of variations appear to be relatively ad hoc. Nothing is said about the completeness of these classifications with respect to the entire Simulink language, which is comprised of 150 unique blocks. Current work focuses on specific blocks or operations, while neglecting others. A more complete treatment of Simulink is required to better understand where to focus evolution, maintenance, and variant management efforts in the first place. In general, analyzing changes across different model versions is needed. Our work considers changes to models made during the development and maintenance phases of the software development lifecycle. While the analyzed models belong to different software variants within a software product line, we do not analyze the differences between the models belonging to different variants.

3.2 Methodology

The overall methodology of this thesis is shown in Figure 3.1. The dark grey steps used to analyze changes in industrial Simulink models are the focus of this chapter. The tools used to support these steps are detailed in Section 3.2.1. The light grey steps are the focus of subsequent chapters.

1. *Extract Data:* Relevant data from the repository is extracted by querying the Issue Tracking System (ITS) for Change Requests (CRs). Labelling of CRs is built into the ITS, as managers assign the Change Category and Change Stimulus to each request. We use these designations to construct queries to the CMS which return CRs that satisfy the query constraints.

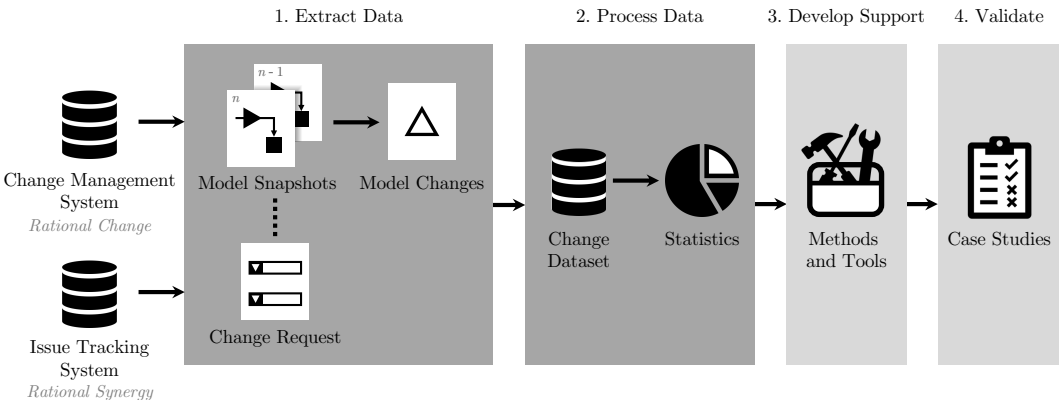


Figure 3.1: Methodology for examining and supporting model changes.

For each CR, we further focus on those that have a change impact on one or more control model files, and retrieve the before and after model snapshots related to the change. To understand what elements were changed and how they were changed, a model difference is performed and saved for later processing.

2. *Process Data:* Statistics on the changes are gathered, computed, and visualized. These are presented in Section 3.3.

3. *Develop Support:* Given an understanding of the most common model changes, methods and tools are developed to support robustness with respect to the changes. This activity is presented in Chapters 4 and 5.

4. *Validate:* The proposed methods and tools are then validated on case studies to demonstrate and evaluate their effectiveness in supporting changes. This activity is presented in Chapter 6.

We analyzed a proprietary software repository of an industrial partner, which we are unable to name. A total of 1,354 repository commits were retrieved, with 21,369 model snapshots collected. The dataset spans a time

frame of 6 years. These models contain 3,206,448 changed elements in total. Data on 429,074 elements were discarded, pertaining to the following:

- *Stateflow element changes.* Stateflow is a graphical language with separate syntax and semantics from Simulink. Full treatment of the Stateflow language is left to future work.

- *Parameter nodes.* Usually, model elements that have had their parameters changed are represented by a single node, where the node's *Parameters* attribute describes the modified parameters. When parameters are more complex (e.g., structure array), they will instead be represented with a node of their own, as a child to the model element. These do not represent distinct elements in the model, so we do not treat them as a standalone change.

- *Nonfunctional and defaults changes.* The MathWorks Simulink Comparison Tool automatically applies filters to the comparison to discard nonfunctional changes (e.g., layout) and block defaults. At the time of writing, there is no way to remove these filters when performing the comparison from the command line.

The remaining 2.7 million model change elements are analyzed in this thesis.

3.2.1 Tool Support

The toolchain for the methodology described in Section 3.2 is shown in Figure 3.2. The scripts surrounded with a solid line were developed as part of this thesis, while a dashed line denotes third-party software.

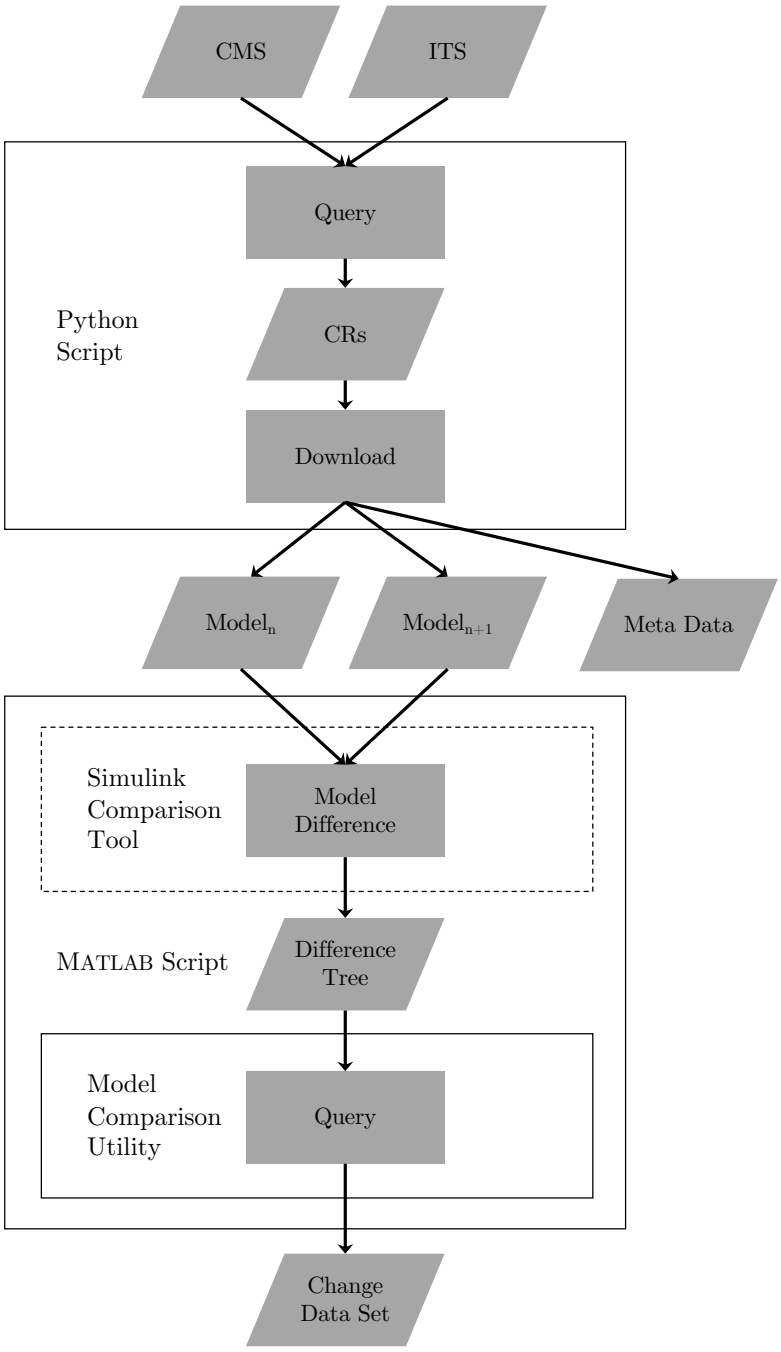


Figure 3.2: Toolchain for extracting the change data set.

This first step of the methodology is automated with a Python script that uses Command Line Interface (CLI) commands of the CMS to query the CMS, find those with models, and download their data. Next, the Model Comparison Utility was developed to facilitate interactions with the Simulink Comparison Tool output. A MATLAB script bundles these steps together and performs the model differencing using the Simulink Comparison Tool, saves the comparison tree, and then the Model Comparison Utility is used to create a database of changes. The next section provides more detail on the Model Comparison Utility.

3.2.2 Model Comparison Utility

Differencing between two models is natively supported in Simulink. Using a method based on hierarchical XML comparison [Chawathe et al., 1996], the Simulink Comparison[1] Tool can generate a Word or HTML report of the differences between the two models. As an alternative, the Simulink Comparison Tool can also output the results in the MATLAB workspace as an two n-ary trees of differences between two models, stored in an *xmlcomp.Edits* object. If the two models are a single design before and after changes have been made, the trees effectively represent model elements before and after changes. We will simply call these models the *before* model and *after* model. The nodes represent elements from the model, and links between nodes are representative of the model hierarchy. Unfortunately, MathWorks provides no built-in commands to be able to easily and programmatically query or parse the trees from the command line or a script. Moreover, extracting information from the tree requires thorough knowledge

[1] https://www.mathworks.com/help/matlab/matlab_env/compare-xml-files.html

of the tree structure and the object parameters, and thus is nontrivial. The Model Comparison Utility[2] is an open-source tool that was created to facilitate such operations via a collection of commands.

The scripts provided by the Simulink Module Tool support searching the comparison tree, getting node handles or paths, plotting the tree, and printing a custom summary of changes. The following is a list of some of the functions that this tool provides:

- *find_node* – Searches the comparison tree for nodes with specific block types, changes, names, etc.
- *getHandle* – Gets the handle of the model element associated with the node from the comparison tree.
- *getPath* – Gets the pathname of the model element associated with the node from the comparison tree.
- *getPathTree* – Gets the node's full path in the comparison tree.
- *plotTree* – Plots the graph of the comparison tree.
- *highlightNodes* – Colours model elements corresponding to nodes from the comparison tree.
- *summaryOfChanges* – Prints a summary report of the changes in the comparison tree to the Command Window or a `.txt` file.

Many other commands are included but are not listed here. Please see the User Guide provided with the Model Comparison Utility for more details. After a model comparison was performed, the Simulink Module Tool was used to query

[2] https://doi.org/10.5281/zenodo.4321650

the results of the comparison in order to extract the information required to produce the results described in the following sections.

3.3 Changes in Simulink Models

At a fundamental level, we categorize changes in model elements into four types: added elements, deleted elements, modified elements, and renamed elements, where elements are any blocks, lines, ports, or annotations in a Simulink model. We created these categories based on the information provided by the comparison tree generated by the MathWorks Simulink Comparison tool. The four types of changes are represented in Figure 3.3. Added elements are those that are not present in the before model but exist in the after model (e.g., new blocks added). Deleted elements are those that exist in the before model, but are not present in the after model (e.g., removed blocks). Modified elements exist in both the before and after models, but have had their parameters changed (e.g., a gain block's *Gain* parameter changed from 1 to 2). Note that we consider a model element to be modified if the element itself has changed, whether or not its children are changed. Renamed elements also exist in both the before and after models, but have their *Name* changed to a different value. These could be considered to be modified elements, however, the Simulink Comparison Tool treats renamed and modified elements as two separate types of changes.[3] As a result, our scripts identify renamed elements in a different manner from those that are modified. Elements can be both renamed and modified.

[3]This may be confusing because in a Simulink model, an element's name is considered a parameter. In the Simulink comparison tree however, it is not treated as a parameter.

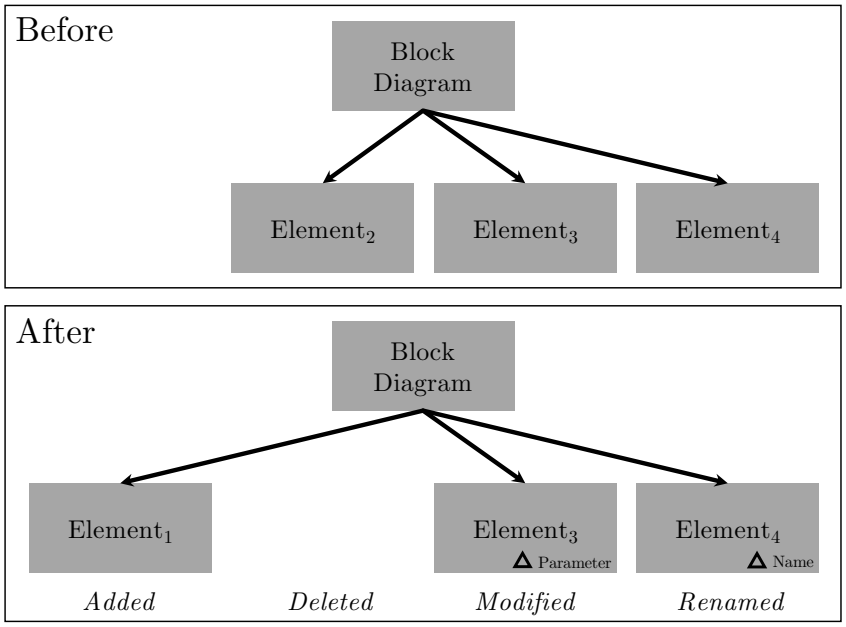


Figure 3.3: Examples of basic changes in Simulink models and comparison trees.

Similarly to the presence/absence of these elements in the before/after models, these change types are represented the same way in the trees generated via the Simulink comparison operation. For example, elements that are added to a model are found in the after tree, but not the before tree. Changes to models can then be classified in terms of changes to model elements and also as combinations of changes to elements. In this work, we focus on the above four fundamental types of changes to model elements. Identifying and analyzing typical compound changes comprised of several fundamental changes (e.g., block replacements, which are a combination of deleting and adding a block) is left to future work. Currently, these changes are treated individually.

The results of the mining analysis are broken down into four different categories:

1. *Changes to basic elements.* The basic elements of Simulink were shown in Figure 2.1. Section 3.3.1 presents the frequency of changes to these elements over all the tasks included in the repository.

2. *Changes to Simulink block types.* Section 3.3.2 describes the frequency with which different types of Simulink blocks are changed.

3. *Model changes in a single commit to the repository.* Section 3.3.3 presents the most frequent block changes that were included in a single commit. This is one measure of what a likely change may look like.

4. *Categories of Simulink blocks that are changed.* Section 3.3.4 categorizes blocks as pertaining to "Interfaces," "Logic," etc. and then analyze what percentage of these categories are changed.

3.3.1 What basic elements change the most?

Given the fundamental building blocks of a Simulink model as given by the metamodel in Figure 2.1, the frequency with which they change is shown in Figure 3.4. The bars are further broken down to describe the types of change that they undergo. Blocks in general provide the main functionality of block diagrams, so it is natural that they are also the most frequently manipulated between model versions.

Lines appear to follow closely in the frequency in which they change. Upon a deeper inspection of how MathWorks performs the model comparison operation, it was discovered that it is impossible to distinguish between the addition of a new line versus a block replacement. For example, if a block is replaced with another, any lines into and out of the original block will be

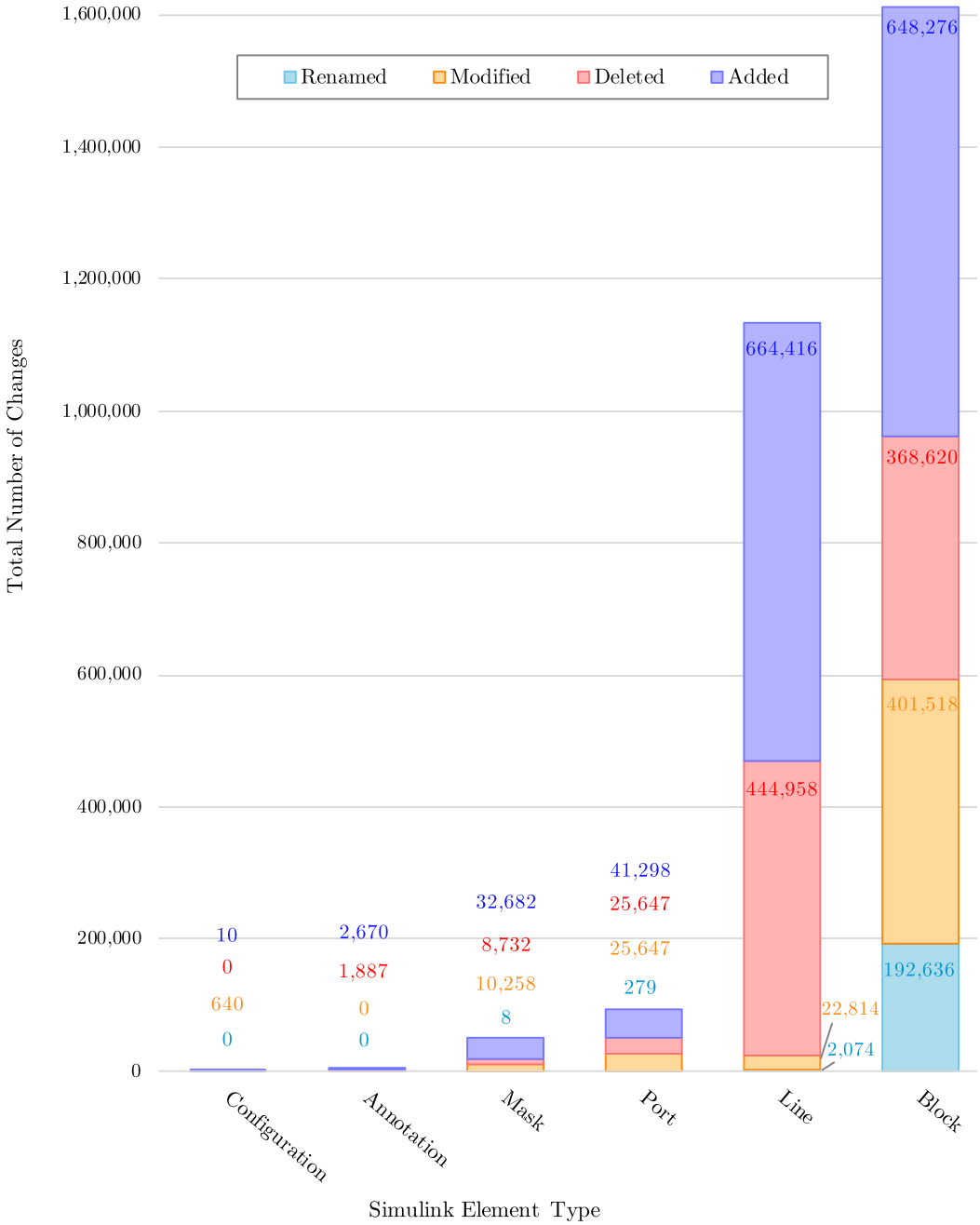


Figure 3.4: Types of Simulink model elements that change.

considered as deleted when they are disconnected. When the new block is connected to the original lines, these lines are now considered to be new, that is, added. This makes it seem like lines are added/deleted when they actually are not. As a result, we largely omit lines from direct consideration in subsequent analyses, because of the way MathWorks performs the comparison operation. Signal routing changes are still monitored, as described in Section 3.3.4.2.

In a similar way, changes to ports are not particularly useful to analyze. Ports are automatically added or deleted when their corresponding Inport/Outport block is added or deleted, and this is done automatically by the Simulink environment. Therefore, changes to ports are primarily related to the occurrence of blocks that represent or enable ports, that is, Inport, Outport, Trigger, Enable, etc. block changes. For example, for every port-related block that is added, a corresponding port is added. Also, ports are automatically renumbered when ports preceding them are added/deleted (e.g., if port 3 is deleted, port 4 becomes port 3, port 5 becomes port 4, etc.). This too can make it appear like many elements are affected when merely a single port-related block is added/deleted. As a result, we also omit ports from consideration in subsequent analyses.

Finally, masks, annotations, and configurations are the least involved in model changes, and will not be examined in the remainder of this work.

3.3.2 What blocks are most often involved in changes?

The Simulink block library contains 401 uniquely named blocks with which a developer can design a model. However, some have a common *BlockType*

parameter (e.g., AND and OR blocks have the *BlockType* of "Logic"). There are 150 unique block types. The blocks that change the most in total are shown in Figure 3.5. The top three changed blocks are Inport, Outport, and From block. It is notable that the most frequently changed blocks deal with interfaces at the model and subsystem levels. Logic blocks are only in the seventh spot.

Although there are 150 unique block types, the version of Simulink used and the configuration of the designed system will impact what blocks will actually be used by any one project. For the example used in this study, newer blocks introduced after R2016b are not used at all, such as the 18 string manipulation blocks introduced in R2018a. Furthermore, we are working with control systems that use a discrete solver, and so continuous blocks are never used.

3.3.3 What does a commit usually entail?

In order to understand which elements are usually modified in any given commit to a Simulink project, we examine the median of all the commits that changed models. The median change is shown in Figure 3.6 and describes how a model usually changes during a single commit. We include information about both the block type and the kind of change (i.e., whether it was added/deleted/modified/renamed). Most of the changes deal with elements being added to the model, because it is most common for designs to be augmented with new functionality to meet new requirements. Changes to Subsystems make up the most changes. It is concerning why many Subsystems are modified in a single change, and points to poor

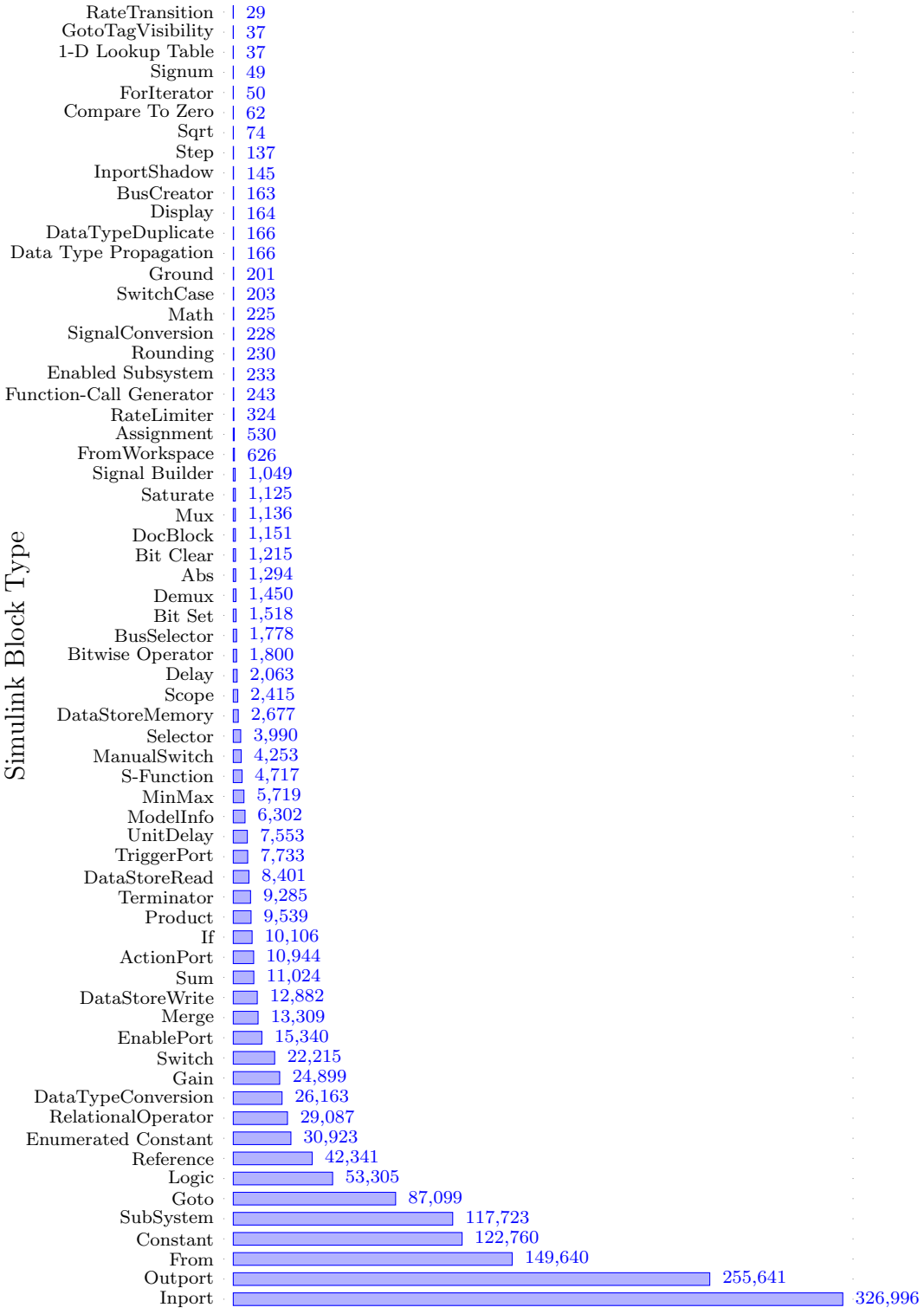


Figure 3.5: Simulink block types that change > 20 times.

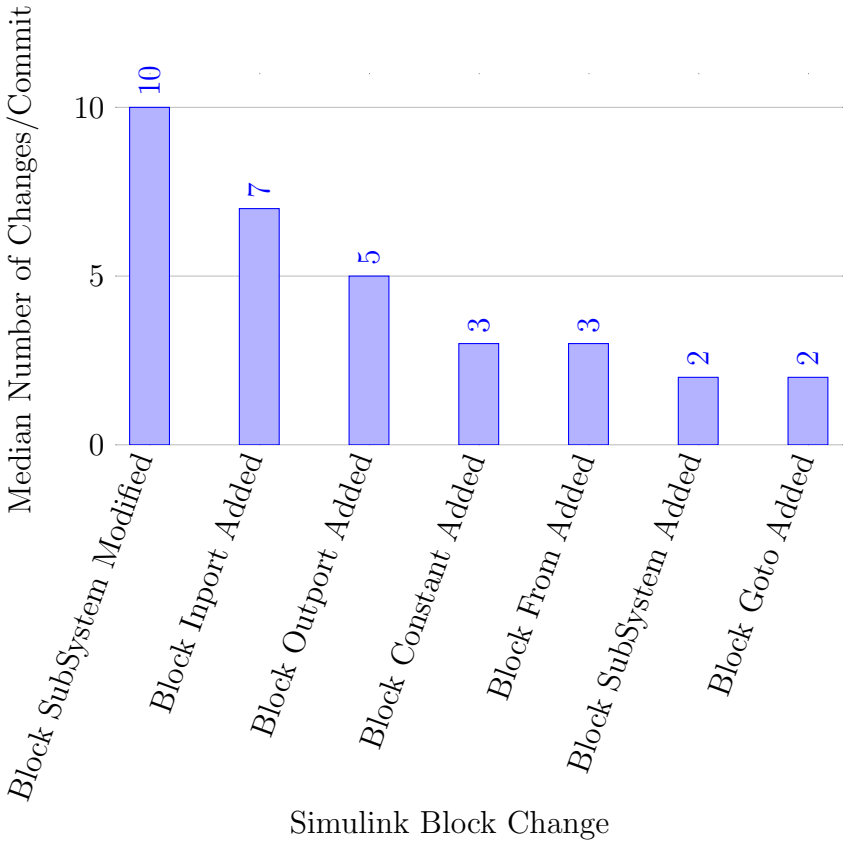


Figure 3.6: Median changes per commit.

modularization of the model. If likely changes were adequately hidden, a single change should ideally impact a single Subsystem. This is clearly not the case, and motivates our work for providing a better structure for Simulink models such that they support information hiding via modularity.

We also examined the average change, however, the data had a large variance. This was because of several reasons. First, migration to a new version of Simulink resulted in changes to *all* the models that comprise the system, where thousands of elements in each model were changed during migration activities. Migration of custom, company-specific, block libraries is a significant pain-point during the migration of the Simulink ecosystem to a

newer version. Custom blocks are usually used in two cases: to provide functionality that is lacking in the current modelling language and to specially configure blocks so that they function in a desired manner across all designs (e.g., to generate specific code, to use a specific rounding approach, etc.). Companies are hesitant to migrate to newer versions of Simulink because of the time and effort required to prepare models for migration and ensure that the generated code behaviour is not inadvertently altered. In the change dataset, over 300,000 block changes were associated with migration efforts to Simulink R2016b. This represents 11% of the 2.7 million changes to models we analyzed.

Second, some drastic changes (e.g., new algorithms, model restructuring) result in change sets that comprise hundreds to thousands of modifications. This occurs when a new product is being developed, for example, which involves novel feature implementations. Last, models that make up an industrial project vary in size greatly. Some are smaller models of a few hundred blocks or less and are primarily used in a model referencing capacity. Other models we have encountered are comprised of over a million blocks.

3.3.4 What are identified categories of change?

The Simulink language block libraries already categorize blocks into categories pertaining to their purpose (e.g., signal routing) or a common quality (e.g., discrete), as shown in Figure 3.7. We took direction from these groups and applied some modifications to create the categories of changes identified in Table 3.1. Examples of blocks within each category are also listed.

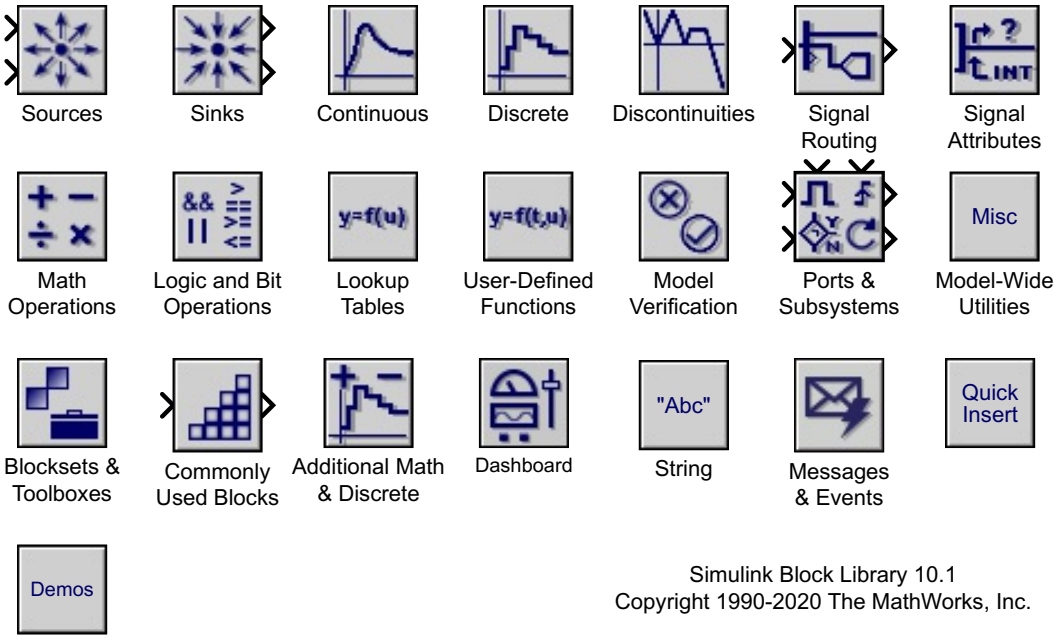


Figure 3.7: Categories in the Simulink block library.

We did not use all the MathWorks groups directly, for several reasons. First, some are just a regrouping of blocks present in other groups, such as *Commonly Used Blocks* and *Quick Insert*. Second, we broke up some groups into smaller, more homogenous groups. For example, the *Ports and Subsystems* group contained many different types of blocks together. We separated this group into four: Conditional, Trigger, Structural, and Interface. Moreover, based on the definition of a Simulink module interface [Jaskolka et al., 2020b] we further added to the Interface group by including blocks that make up a module interface. This also helped to delineate top-level interfaces from internal subsystem interfaces that we grouped in the *Signal Routing* category. For these two categories, we distinguish between global and local data stores. Global data stores are on the model interface, since they pass data outside of the model. Local data stores are those that pass data inside the model only, thus they are included in the *Signal Routing* category. The same treatment is

Table 3.1: Categories of Simulink blocks used in Figure 3.8.

Category	Example Blocks
(Model) Interface	root-level Inport/Outport, global DataStoreRead/DataStoreWrite, FromFile/ToFile, FromSpreadsheet, ToWorkspace/FromWorkspace
Signal Routing	non-root Inport/Outport, Goto/From, local DataStoreRead/Write, BusCreator, Merge, Assignment
Signal Attributes	RateTransition, DataTypeDuplicate, SignalConversion, DataTypeConversion
Structural	SubSystem, Reference
Conditional	If, Switch, SwitchCase, ManualSwitch
Discrete	Delay, UnitDelay, Filter, Integrator
Math	Sum, MinMax, Rounding, Abs, Gain
Logic	RelationalOperator, LogicalOperator
Trigger	TriggerPort, EnablePort, ActionPort
Sources	Ground, Step, Clock, Constant
Sinks	Terminator, Scope, Display
Documentation	ModelInfo, DocBlock
Custom	S-Function

given to inports and outports. When they are located at the root of the model, they are on the model interface, while Inport/Outport blocks inside subsystems are used to pass signals internally.

The Sources and Sinks groups remained largely unchanged, except that Inport and Outport blocks were removed, as mentioned previously. The Math, Discrete, and Signal Attributes categories are unchanged from the MathWorks grouping.

We made a separate Documentation group for blocks related to documentation within *Model-Wide Utilities*. Some other notables changes were: the Switch block in the Simulink block library is part of the *Signal*

Routing group, however, we put it into the Conditional group, as it is traditionally known as a conditional control structure; Simulink Function blocks are contained in the *User-Defined Functions* group, but are in fact Subsystem blocks, so they were added to the Structural group.

Lastly, we renamed the *User-Defined Functions* group to simply *Custom*, as the blocks in this category were entirely project-specific custom blocks, and not blocks that were defined by the engineers (users) themselves.

Not all of the groups from the Simulink block library are present in our categories. As mentioned in Section 3.3.2, this particular project does not make use of concepts such as *Dashboard*, *Strings*, etc., so those are omitted.

It is worth noting that our classification scheme has another distinct advantage—element changes are counted in only one of the categories; there are no overlaps. For instance, global data stores contribute to the *Interface* category, while data stores internal to a block contribute to *Signal Routing*.

Figure 3.8 shows the occurrence of block changes in each category. First, we can see that the largest amount of change occurs to model interfaces at the root level. Second, signal routing within the model is changed extensively. Third, structural changes to subsystems, model references, and library links occur frequently. It is interesting to note that these three aforementioned changes happen more than actual math and logic changes to the control algorithms. This leads one to conclude that developers are spending less time actually implementing the algorithms, and are more focused on passing data across the system. The large amount of interface changes at the top level points to unstable interfaces and change propagation between models. Even within the models, signals are being modified and moved around the model hierarchy.

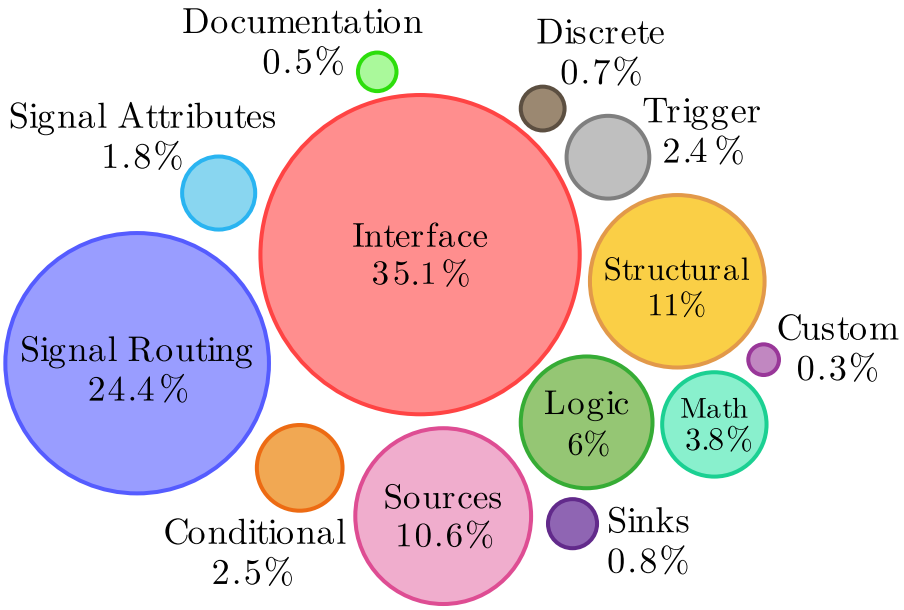


Figure 3.8: Categories of Simulink blocks that change.

3.3.4.1 Changes to Interface Elements

This section describes the changes to Simulink interfaces and their frequency over the lifespan of the models. A model or subsystem interface is comprised of both the explicit and implicit incoming/outgoing data flow [Jaskolka et al., 2020b; Bender et al., 2015]. For the Simulink language, the explicit data flow is represented with Inport and Outport blocks, and the MathWorks documentation considers this to be the entire interface. However, implicit data passes via Data Store Memory, Goto Tag Visibility, To Workspace, To File, and many other blocks, across model or subsystem boundaries, as described in the interface definition in [Jaskolka et al., 2020b].

Using the definition of a Simulink module interface [Jaskolka et al., 2020b], we identify the changes that occur to Simulink model interfaces. This includes changes to top-level Inport/Outport blocks, global Memory Read/Write, To File/From File blocks, To Workspace/From Workspace blocks,

From Spreadsheet blocks, and exported Simulink Functions. 35.1% of all block changes are interface changes. This is larger than expected, and points to unstable interfaces that are sensitive to implementation changes. In fact, Inports at the root-level of the model change 6 times as often as internal Inports (281,236 changes vs. 45,760 changes). Root-level Outports are also changed 6 times as often as internal outports (219,291 vs. 36,350). However, the internal routing of signals can be achieved via several other Simulink blocks, and indeed we find that they are heavily used, as discussed in the next section.

3.3.4.2 Changes to Signal Routing and Attributes

Changes to signal routing entails modifying the way signal data is created, combined, passed, and selected. As mentioned in Section 3.3.1, directly analyzing changes to the lines themselves will not yield meaningful results. For this category of changes, we consider the list of blocks in the *Signal Routing* group, which includes Goto/From, Mux, Bus, non-global Data Store Read/Write, etc. We can see in Figure 3.8 that routing signals throughout models is a very large portion of the changes. In a similar way that using goto statements in textual programming languages leads to "spaghetti code," over-use of these similar Simulink blocks can lead to "spaghetti models." Although abstraction through hierarchy is a well-known approach to managing complexity, the side-effect is that large hierarchies cause substantial signal routing of blocks between layers. Both Subsystems and Model References introduce a hierarchical layer in a Simulink model. Clearly, models need to be better structured in order to reduce the pervasive passing of signals.

3.4 Chapter Summary

This chapter provided data-supported insights about how large Simulink models change over time. We analyzed over 2.7 million changes over a span of six years in an industrial partner's repository. This chapter shed light on how Simulink models change with respect to their basic language elements and block types. A categorization of the changes that occur in Simulink models was also presented to succinctly show where the development and maintenance effort resides in the design of a model. It highlighted that Simulink model interfaces and structure experience large amounts of change. These are areas that require further attention in order to facilitate modularity and information hiding with Simulink.

Chapter 4

Decomposition of Simulink Models

It is well-known that modular designs better facilitate evolution [Baldwin and Clark, 2000; Parnas et al., 1985; MacCormack et al., 2007]. Thus, as a model grows larger and more complex, it is desirable to decompose it into smaller units of related functionality. In traditional imperative programming languages, a system is decomposed into modules, with facilities to create private and public functionality to support encapsulation and information hiding [Parnas, 1972a]. International standards such as AUTOSAR [autosar.org, 2018] seek to apply traditional software engineering principles in the automotive domain. For example, AUTOSAR emphasizes separation and well-defined interfaces between application software, base software, and hardware. However, limited guidance is given pertaining to decomposition at the application layer where components typically correspond to Simulink models. It is not currently clear how to further decompose the functionality of a Simulink model.

As a first step in achieving a modular decomposition, we seek to determine which Simulink constructs can be used to group together functionality and data as a unit that can be separate, reusable, encapsulated, and ultimately facilitate information hiding. To support this goal, this chapter examines and compares the available constructs in the Simulink language in a more focused and comprehensive manner than is available in the MathWorks documentation [The MathWorks, 2019, 2020b,h] and that we have found in the literature. The contributions of this chapter will assist practitioners in choosing appropriate componentization constructs for structuring designs in Simulink models. It also lays the groundwork for creating a Simulink module, as described in Chapter 5.

Section 4.1 begins with related literature. Section 4.2 performs an in-depth comparison of the available Simulink model decomposition constructs.[1] Section 4.3 discusses how to convert between constructs. Section 4.4 presents modelling conventions for supporting modular design with information hiding. Lastly, Section 4.5 provides a summary of this chapter.

4.1 Related Work

The literature sets forth recommendations on the constructs to use for decomposing models for various purposes. MathWorks provides a general guide in the Simulink User's Guide [The MathWorks, 2019] for choosing appropriate decomposition constructs. It focuses on only three constructs—Subsystem, Library, and Model Reference—and compares them according to how each supports the development process, model

[1] Also called "componentization techniques" in the MathWorks documentation.

Table 4.1: Componentization summary fragment from the Simulink User's Guide [The MathWorks, 2019].

Requirement or Feature	Subsystem	Library	Model Reference
Component reuse	Not supported	Well suited	Well suited
Intellectual property protection	Not supported	Not Supported	Well suited
Simulation speed for large models	Supported, with limitations	Supported, with limitations	Well suited
Code generation	Supported, with limitations	Supported, with limitations	Well suited

performance, component reuse, and other factors. Modularity is not examined explicitly, nor are encapsulation or handling of program state. Reusability of each construct is examined, with a Model Reference or Library being best suited for reuse. A fragment of this guide is given in Table 4.1, highlighting the more relevant differences between constructs. We refer the reader to the Simulink User's Guide for an in-depth comparison [The MathWorks, 2019]. Subsystems are most commonly used in practice (e.g., [Xiao and Agbossou, 2009; Yang et al., 2012; Astrov and Pedai, 2012], etc.) as they are recommended for designs with less than 500 blocks [The MathWorks, 2019]. Otherwise, a Library or Model Reference are recommended. General descriptions of code generation outcomes are given throughout other MathWorks documentation [The MathWorks, 2020b,h], however, no comparison is concisely provided.

Other work on structural patterns for Simulink models usually proposes decomposing Simulink models using virtual Subsystems or Model References (e.g., [The MathWorks, 2020c],[Whalen et al., 2014]), while other work assumes Subsystems as the decomposition construct used [Dajsuren et al.,

2013]. Drawbacks to using Subsystems have been noted as their lack of reuse and formal interface [Rau, 2001; Bender et al., 2015; Bialy et al., 2016]. As far as we are aware, no analysis has been published that determines which Simulink constructs can, and should, be used for supporting the traditional software engineering principles of modularity, encapsulation, and information hiding.

4.2 Comparison of Constructs

The decomposition of a model into smaller units can be achieved with three different constructs that are built into the Simulink language: Subsystem (Section 2.2.1), Library (Section 2.2.2), and Model Reference (Section 2.2.3). However, there are many different kinds of subsystems, as shown in Figure 2.3. We evaluate three specific kinds of subsystems that are most suitable for decomposition purposes: virtual Subsystem (Section 2.2.1.1), Atomic Subsystem (Section 2.2.1.2), and Simulink Function (Section 2.2.1.2), along with the Library and Model Reference constructs. These five constructs are shown in Figure 4.1.

A comparison of the five different constructs is now presented, focusing on how they are used in industry. The different constructs are also compared with respect to four characteristics: reusability, sharing of program state, encapsulation, and code generation. This section consolidates and summarizes the Simulink User's Guide where relevant comparison information is available. Production-scale models are used to exemplify how the constructs have been used in industrial designs. Additionally, simple experimentation models have been designed to evaluate and illustrate core

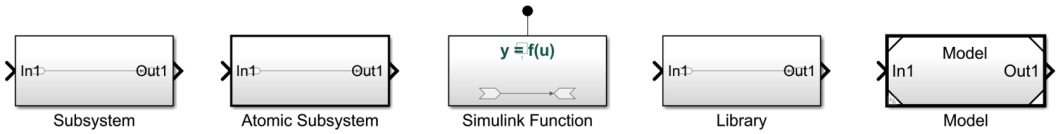


Figure 4.1: Simulink componentization constructs examined.

capabilities of each construct. Other factors such as model loading, simulation speed, or memory storage, are not considered in the comparison, as they are beyond the scope of this work.

4.2.1 Use in Industry

Let us begin by examining how each of these constructs are used in industry, by analyzing projects from an industrial partner. The frequency with which these constructs are used is shown in Figure 4.2 and Figure 4.3. The Simulink Function, Library, and Model Reference constructs are defined and used with separate blocks, thus we present two charts. Figure 4.2 shows how many constructs are defined, while Figure 4.3 shows how many times they are used (i.e., called, linked, referenced). Note that the numbers in the two charts are the same for virtual Subsystems and Atomic Subsystems because their definition and use are one and the same block. The projects are displayed in decreasing order according to project size, as reflected in the number of Simulink models used in their implementation. Specifically, Project_1 is comprised of 653 Simulink models, Project_2 of 134 Simulink models, and Project_3 of 92 Simulink models. We provide more insight into the structure of each project in the following paragraphs.

Project_1 is a relatively new project that began development in 2015 and is currently implemented in Simulink R2016a. It contains approximately 419,000

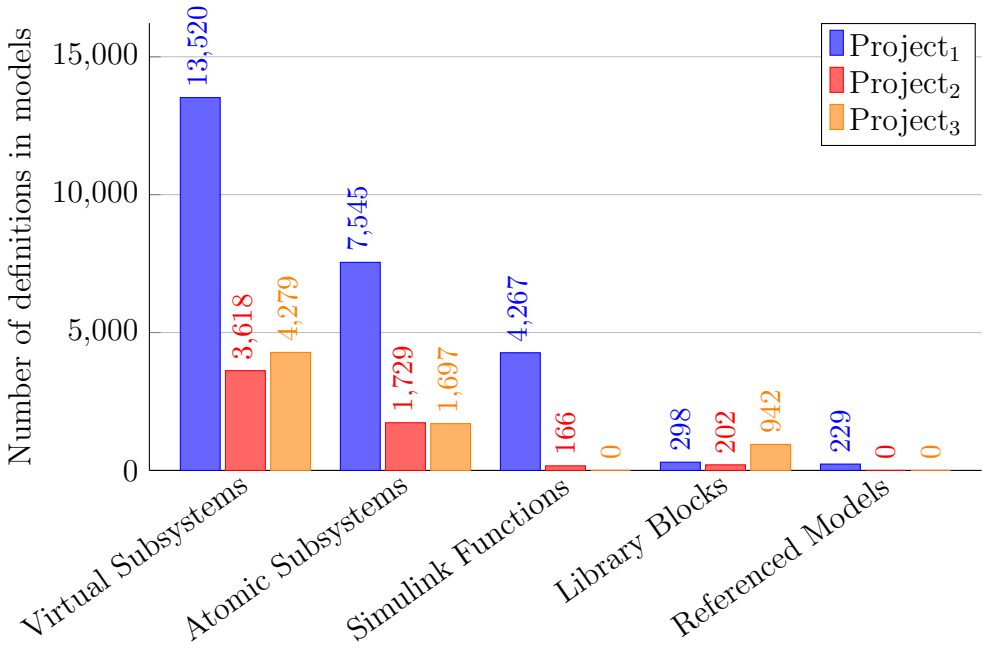


Figure 4.2: Definitions of componentization constructs in industry projects.

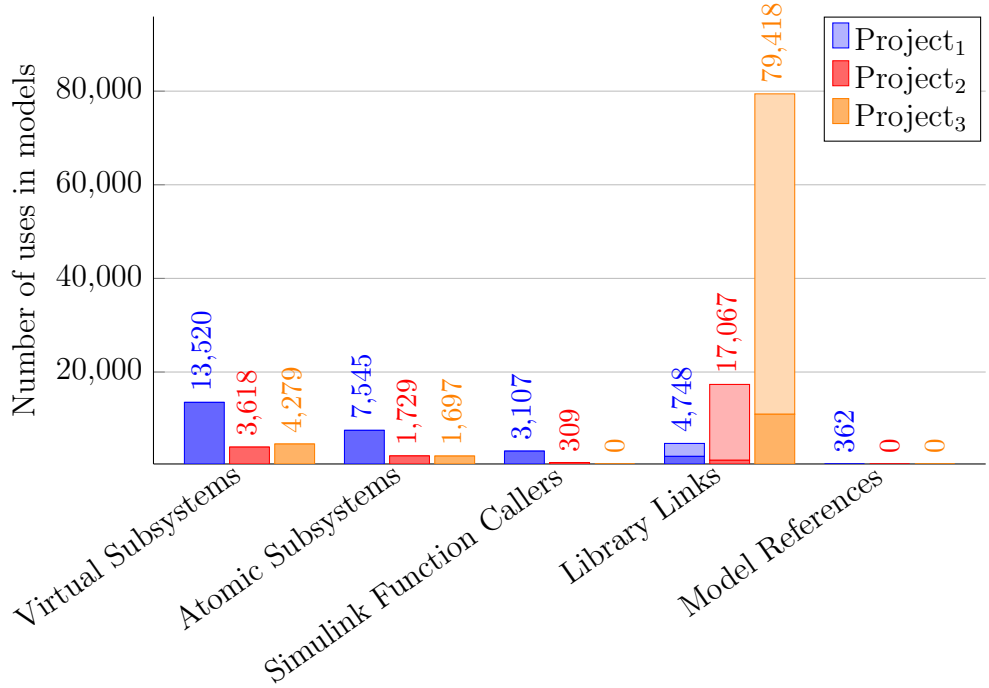


Figure 4.3: Uses of componentization constructs in industry projects.

blocks, and of these blocks 6% are among the componentization constructs examined in this work. Each model on average contains 642 blocks. Virtual Subsystems are used 53% of the time, followed by Atomic Subsystems at 29%. As this is a newer project, the use of Simulink Functions is also sizeable at 17%.

Project_2 was also developed in 2015 and is also implemented in Simulink R2016a. It is comprised of approximately 250,000 blocks and on average a single model contains 1,894 blocks, with only 2% being componentization constructs. As with Project_1, Project_2 also relies mostly on virtual Subsystems, followed by Atomic Subsystems, and Simulink Functions.

Project_3 is a large legacy project that was originally created in Simulink R14, and has been migrated over the years to newer Simulink versions. It contains approximately 1.3 million blocks, which is on average 14,338 blocks per model. This project is made up of very large models, where componentization constructs only amount for 0.5% of all the blocks used. Project_3 relies primarily on virtual Subsystems, followed by Atomic Subsystems. This project only recently migrated from MATLAB Simulink R2011b to R2016b, thus Simulink Function blocks are not used in this project. This project uses 942 unique Library blocks in its implementation. These Library blocks are used very frequently with over 79,000 instances (or links) throughout the models. However, this large number of Library links is attributed to the use of a custom Library of blocks that are developed in-house, rather than using the built-in Simulink blocks. The custom Library for this project contains customized blocks that fulfill the same function of a core Simulink block, but are pre-configured or modified in order to better support the project (e.g., with more efficient code generation). Figure 4.3 shows that custom block usage makes up 85% of the cases of Library links, as

denoted by the lighter area of the bar. The remaining Library links are to Library blocks whose purpose is to structurally decompose the model. Project_1 and Project_2 also use the custom Libraries to redefine some of Simulink's built-in blocks, but to a much lesser extent.

From this data, it is clear that virtual Subsystems are most commonly used in practice. Two of the three projects (Project_2 and Project_3) also rely on Atomic Subsystem blocks, but barely use Simulink Function blocks. Project_1 on the other hand uses Simulink Function blocks to a greater extent, but they are still third to virtual Subsystems and Atomic Subsystems. Out of all the constructs, Model References are used the least, across all the projects.

4.2.2 Reusability

One of the purposes of componentization constructs is to reuse functionality throughout one or more models. Reusability is directly tied to the maintainability of a system, as it reduces the number of models or subsystems that must be modified when a change is made. When reusability is not adequately supported, developers often turn to the use of clones, which negatively affect the model's understandability and ability to be refactored. Here, we compare the reusability of the different constructs. The reusability of a construct is determined by the Simulink language semantics, and is described in the MathWorks User's Guide [The MathWorks, 2019]. The reusability of the five constructs is outlined in the following paragraphs.

Subsystem A Subsystem block cannot be reused in multiple places in a model without copying and pasting the block. Using clones of Subsystem blocks through a model is not recommended, as this hinders maintainability

and limits readability [Stephan et al., 2013; Alalfi et al., 2012]. Otherwise, the same Subsystem can be invoked multiple times in a single model via loops. On its own, a Subsystem is not a construct that supports reusability. However, combined with a Library or Model Reference, they can be made reusable.

Atomic Subsystem Atomic Subsystems behave like Subsystems when it comes to reusability. See the paragraph above for more details.

Simulink Function A Simulink Function can be invoked multiple times in its own model and any model that references the model where the Simulink Function is defined (see Figure 2.5 and Figure 2.6). Unlike the other constructs, it can be invoked both via signal lines or textually, making it a flexible construct that alleviates complex signal routing. Unlike the Library or Model Reference constructs, a Simulink Function has the benefit of not needing to be stored in a separate file to be reused.

Library A Library can store Subsystems in order to make them reusable. Subsystem blocks placed in a Library can be linked and used in other models. Even so, an alternative is desired so that a Subsystem is reusable without the need to keep it in a Library file, separate from its associated model [Rau, 2001]. For small functions especially, separate Libraries are cumbersome to use, and in fact, the Simulink User's Guide recommends the use of Libraries for large-scale modelling with more than 500 blocks [The MathWorks, 2019]. Moreover, it is generally intended for storing utility functions that are infrequently changed [The MathWorks, 2019].

Model Reference Model referencing lends itself well to reusability, however, on the scale of a model. A model can be referenced from other models via Model blocks. However, like Libraries, this construct is more appropriate for reuse of very large designs. Moreover, the user must include the entire contents of the model.

In summary, the Simulink Function, Library, and Model Reference constructs can be used to implement reusable functionality, while the Subsystem and Atomic Subsystem are not reusable (without the aid of a Library or Model Reference).

4.2.3 Sharing of Program State

A program's *state* is comprised of all the contents of its stored data in memory, at any point during program execution. In textual programming languages, these storage locations are commonly represented by variables. In Simulink, Data Store Memory blocks are comparable to variables, so they are included in the state of a model. Additionally, when the outputs of a model are functions of its previous values, these values are also states [The MathWorks, 2019]. The two types of states that can occur in a Simulink model are discrete (e.g., unit delay) or continuous (e.g., integrator). These kinds of blocks require persistent memory to store values representing the state of the block between consecutive time intervals.

For constructs that are reusable, their state can be persistent and shared between calls, or each instance can be treated as separate with no sharing of data. In general, it is well-known that avoiding the sharing of internal state data is useful for decreasing complexity and bugs, as well as reducing

dependencies [Schwinghammer et al., 2010; O'Hearn et al., 2009; Edwards, 1997; Hoare, 1971]. This approach prevents users from unintentionally causing an invalid or inconsistent state. Moreover, it prevents users from becoming reliant on internal data outside of its intended scope. Exposed state data can lead to design decisions creeping into the interfaces of subsystems, making them context dependent, and ultimately less modifiable in the future. Encapsulating shared state reduces implicit interdependencies and facilitates reuse, maintainability, and evolution. The decomposition constructs we are evaluating handle program state differently, and we describe this in the following paragraphs.

Subsystem As discussed previously, Subsystems are not reusable on their own. As a result, each Subsystem in a model is its own separate entity. Thus, Subsystem blocks that are clones of each other do not share any state.

Atomic Subsystem Atomic Subsystems behave like Subsystems when it comes to program state. See the previous subsection for more details.

Simulink Function Simulink Function blocks have persistent and shared state between function calls by default. However, it is possible for the developer to use additional blocks in the Simulink Function to selectively reset state values, for example, by using a Resettable Subsystem[2] or Reset Function.[3]

Library Each Library block in a model is a separate instance and state between blocks is not shared.

[2] https://www.mathworks.com/help/simulink/slref/resettablesubsystem.html
[3] https://www.mathworks.com/help/simulink/slref/resetfunction.html

Model Reference Model Reference blocks do not share state data between each reference, however, one can get around this via a Data Store Memory block with the *Share across model instances* parameter enabled.

In summary, Subsystem, Atomic Subsystem, and Library blocks do not share state. Simulink Function blocks do share state by default, but it is possible to implement a reset of state. Generally, Model Reference blocks do not share state, but it is possible with a Data Store Memory block only.

4.2.4 Information Hiding and Encapsulation

Information hiding is a fundamental principle in supporting modularity and robustness with respect to change [Parnas, 1972a] during the evolution of a system. Information hiding aims to decompose a system such that each likely change is localized (hidden) in a single module (e.g., hardware changes, behaviour changes, and software design decision changes [Parnas et al., 1985]). Encapsulation is integral to supporting information hiding as it is a mechanism for restricting access to a portion of the module's data [International Organization for Standardization, 2017; Snyder, 1986]. Two properties are needed in Simulink to support encapsulation:

- *Limitation of Use* – The ability to selectively hide or expose a portion of the functionality.
- *Restriction of Data Passing* – The ability to effectively restrict information passing that is outside of the explicit interface.

This section describes a set of experiments that reveal how each of the constructs support, or do not support, these properties. Note that the

Table 4.2: Simulink construct support for encapsulation.

Construct	Limitation of Use	Restriction of Data Passing	
		Goto/From	Data Store
Subsystem	No	No	No
Atomic Subsystem	No	No	No
Simulink Function	Yes	Yes	No
Library	No	No	No
Model Reference	No	Yes	Local Only

experiments evaluate how each construct *enforces* these properties. It is possible to use conventions to *recommend* best practices, as described later in Section 4.4. Our stance is that language enforcement is a stronger approach, as opposed to trusting developers to adhere to guidelines.

4.2.4.1 Limitation of Use

In this section, we examine if it is possible to selectively restrict or allow the use of functionality encapsulated by a componentization construct. The results are shown in Table 4.2 under "Limitation of Use."

Subsystem As Subsystems are not reusable, this property is not applicable. If a Subsystem is "reused" via copy-and-paste, a Subsystem copy can be placed and used anywhere in a model or other models without restriction.

Atomic Subsystem Atomic Subsystems behave like Subsystems when it comes to limitation of use. See the previous subsection for more details.

Simulink Function Simulink Functions have the ability to be scoped, allowing one to limit its accessibility. This can be leveraged to make private,

or hide, the information within the model, or choose to expose it to other models, as described in Section 2.2.1.2.

Library Library blocks can be instantiated and used anywhere in other models without restriction.

Model Reference A model can be referenced from any other model, without restriction. Models have the option to enable "intellectual property protection," however, it is intended for third-party suppliers to be able to password protect certain operations on a model (e.g., simulation and code generation). Thus, it is not particularly useful for information hiding purposes.

In summary, a Simulink Function is the only construct that has the ability to specify its scope. All others examined can be used anywhere in a model.

4.2.4.2 Restriction of Data Passing

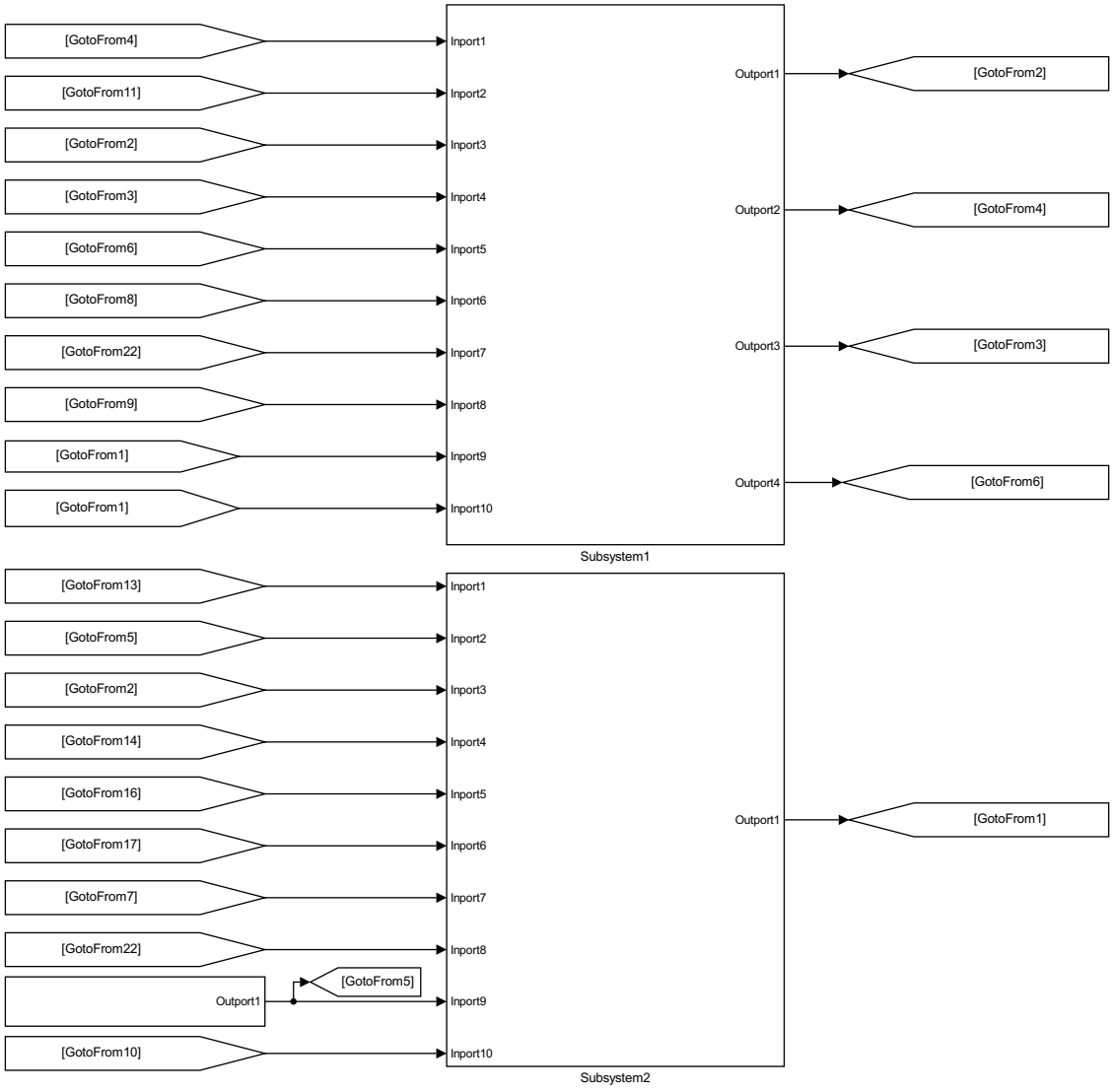
Next, we look into how each of the constructs restricts implicit data passing across its boundaries. We want to ensure that implicitly exposing internal design, or implicitly reading in data, outside of the explicit interface is not possible. In particular, the hidden data passing mechanisms of Goto/From and data stores can be used to implicitly pass data across the construct boundaries [Bender et al., 2015]. Moreover, they can reduce the readability and traceability of a model [Tran et al., 2013]. We examine these two ways of passing data in/out in the following scenarios. We construct Simulink models to test these scenarios by trying to implicitly pass a signal with value 1

across the construct boundaries via Goto/From blocks and Data Store Read/Write blocks.

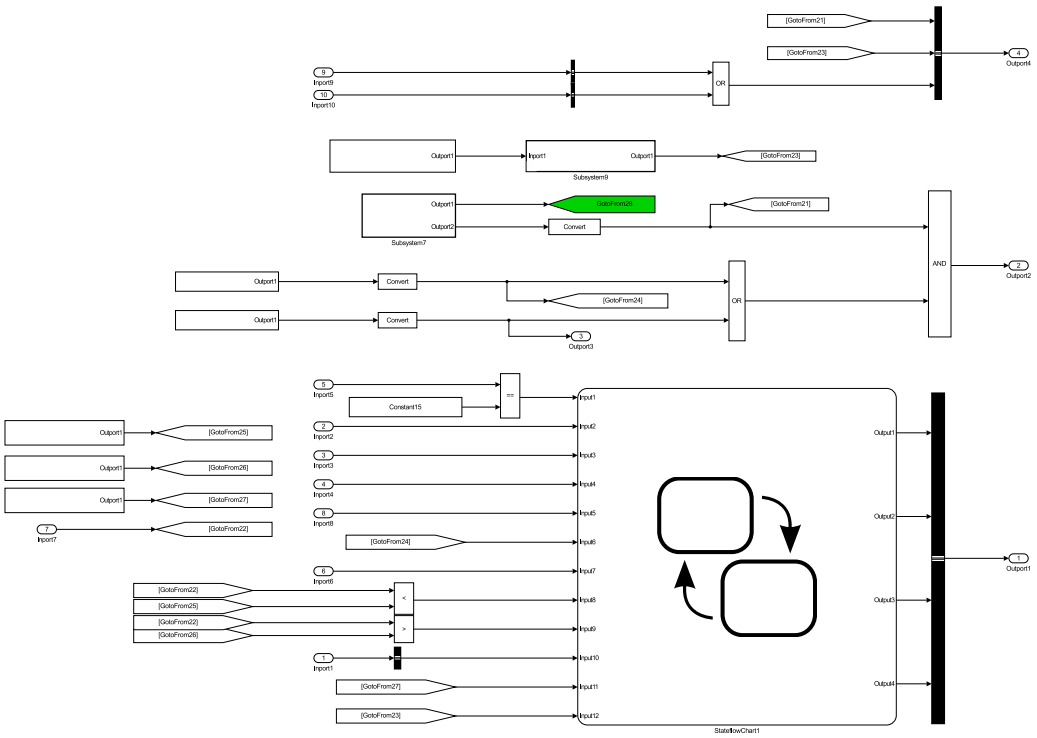
Case 1 (Goto/From as Implicit Input and Output) Implicit data passing via global or scoped Goto/From connections occurs in practice, and Figure 4.4a shows a portion of a production-scale model that implements on-board diagnostics functionality. In addition to the data that passes through the ports of the two subsystems, data is implicitly passed via the highlighted global Goto/From connection shown originating in the subsystem shown in Figure 4.4b and being passed into the subsystem of Figure 4.4c. This type of behaviour should be avoided to support encapsulation. A construct should effectively hide its internal data by not allowing for it to be passed out implicitly, and likewise, a construct should not implicitly read in external data.

The example demonstrates that implicit data passing is possible for the Subsystem construct, however, we need to investigate how the other constructs handle such a scenario. Thus, we create an experiment model (Figure 4.5a) where a global Goto is placed outside of each construct to see if it is possible to pass data into the construct by writing to an internal From block inside of the construct. Each of the constructs is coloured in grey, while the Goto/From blocks are green. Display blocks inside each construct are connected to the From blocks in order to report their values (not pictured).

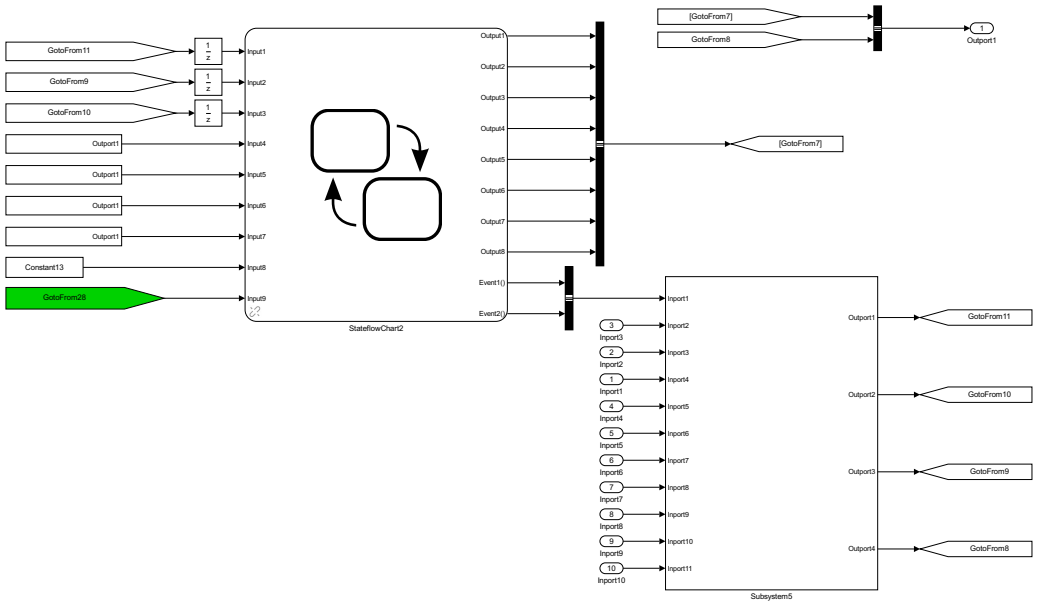
Likewise, to determine if the construct effectively hides its internal data, a global Goto is placed inside the construct. This second experiment model is shown in Figure 4.6a. We attempt to access its data outside of the construct by reading it via an external From block. Each of the constructs is coloured in



(a) Model with constructs using Goto/From as input.

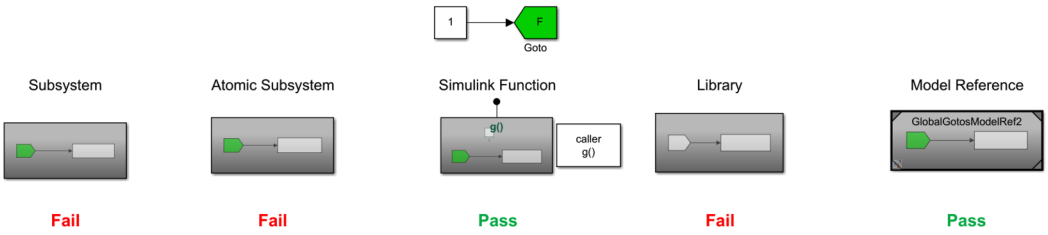


(b) Subsystem1 from Figure 4.4a implicitly outputting data via a Goto.

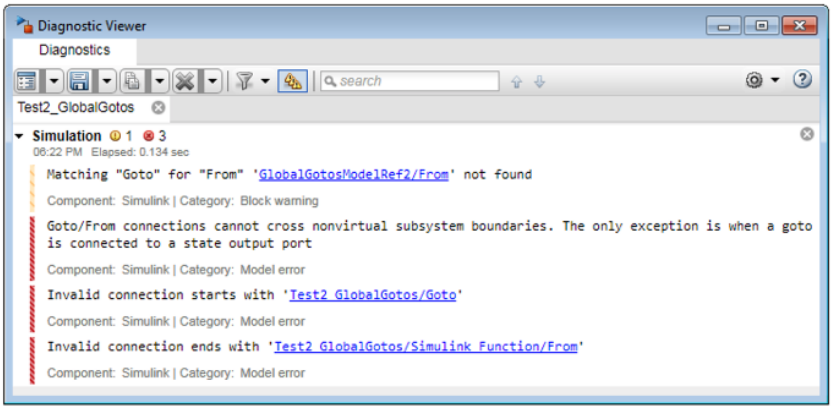


(c) Subsystem2 from Figure 4.4a implicitly reading data via a From.

Figure 4.4: An on-board diagnostics example from industry, showing implicit Goto/From data passing.



(a) Model with Subsystem constructs using Goto/From as input.



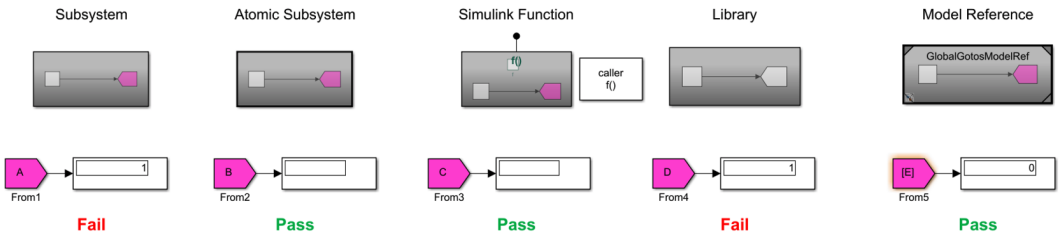
(b) Simulation warnings and errors for Figure 4.5a.

Figure 4.5: Experiment with Goto/From input.

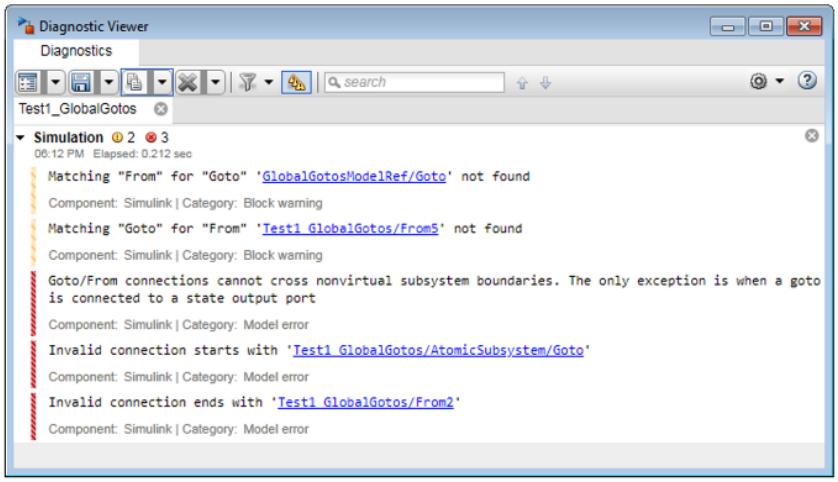
grey, while the Goto/From blocks are magenta. Display blocks are connected to each From block in order to display their values.

The result of the first test shows that a Goto/From can be implicitly read inside a Subsystem, Atomic Subsystem, and Library, but not read in a Simulink Function or Model Reference. The Simulink Function causes the error shown in Figure 4.5b, while the Model Reference causes warnings of missing Goto/From connections. The result of the second test is that Atomic Subsystems and Simulink Functions restrict access to their internal Goto, as no result is shown in the From block and the error in Figure 4.6b is thrown.[4] The Model Reference also successfully restricts access because the default 0 is passed to

[4]Note that the Atomic Subsystem causes the error shown in Figure 4.6b. However, if the Atomic Subsystem is commented-out, the Simulink Function will cause the error.



(a) Model with constructs using Goto/From as output.



(b) Simulation warning and errors for Figure 4.6a.

Figure 4.6: Experiment with Goto/From output.

the From block and warnings of missing Goto/From connections are displayed (Figure 4.6b). On the other hand, a Subsystem and Library do not restrict access, as we can see that the value 1 is indeed passed implicitly across their boundaries.

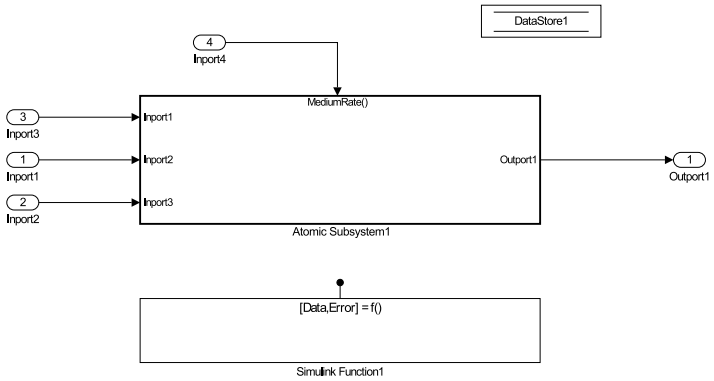
Case 2 (Data Store as Implicit Input and Output) Data Store Memory blocks are another mechanism by which data can be passed implicitly into and out of constructs. A production-scale example from industry that deals with the shifter position checking is shown in Figure 4.7a. Datastore1 is declared at this level, and we see that there is an Atomic Subsystem (Subsystem1) and a Simulink Function (Simulink Function1). The internals of the Atomic Subsystem

are shown in Figure 4.7b, where we see that it is implicitly outputting data via a Data Store Write. This data is passed across the Atomic Subsystem boundary and consumed by the Simulink Function via a Data Store Read. Neither of these constructs effectively restrict the passing of implicit data across their boundaries.

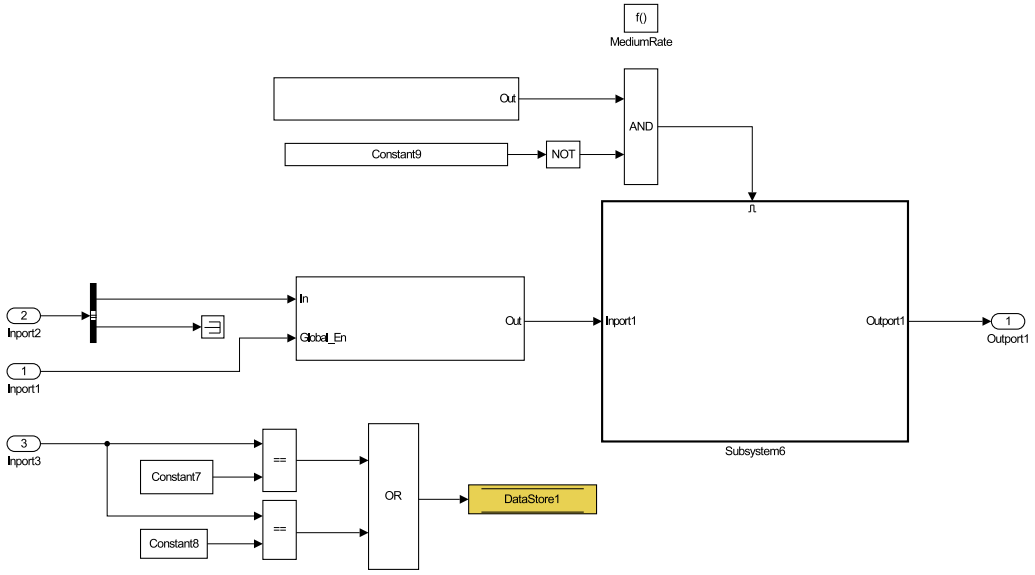
Again, we replicate such a scenario for the other constructs. We do this by placing a Data Store outside of the construct to determine if it is possible to pass data into the construct. It is possible to define a data store as a Data Store Memory block in the model, or outside the model in the base workspace as a `Simulink.Signal` object. In both cases, we attempt to read from it inside the construct via a Data Store Read block. The models to test these cases are shown in Figure 4.8a and Figure 4.9. Each of the constructs is coloured in grey, while the Data Store Read/Write blocks are yellow or cyan. Display blocks inside each construct are connected to the Data Store Read blocks in order to display their values (not pictured).

The result of these tests show that a Data Store Memory in the model, but outside of a Subsystem, Atomic Subsystem, Simulink Function, or Library can be read implicitly by these constructs. On the other hand, this is not allowed with a Model Reference, and the error shown in Figure 4.8b is raised. All of these constructs do not restrict the passing of data store memory when it is global (Figure 4.9).

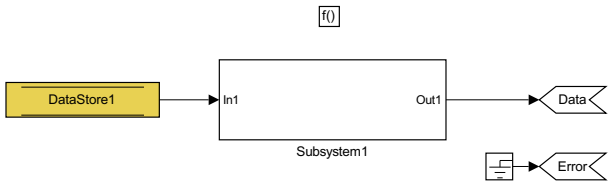
A summary of results is shown in Table 4.2. In general, no construct effectively prevents hidden data passing from circumventing the construct's explicit interface, and this is a deficiency of these Simulink constructs. Nevertheless, out of the constructs examined, a Simulink Function most



(a) Model with constructs using Data Store Read/Write as input/ouput.

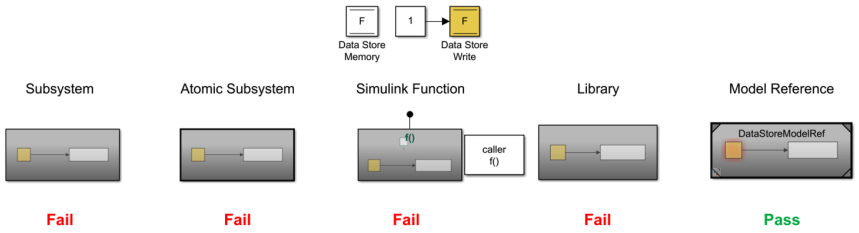


(b) Atomic Subsystem1 from Figure 4.7a implicitly outputting data via a Data Store Write.

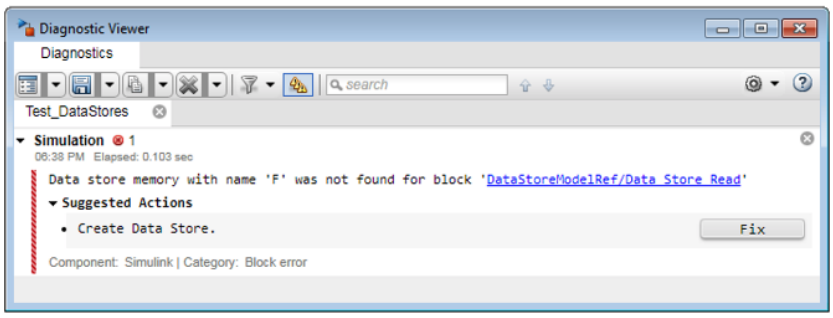


(c) Simulink Function1 from Figure 4.7a implicitly reading data via a Data Store Read.

Figure 4.7: A shifter position checking example from industry, showing implicit Data Store Memory data passing.



(a) Model with constructs using a local Data Store Memory block as input.



(b) Simulation error for Figure 4.8a.

Figure 4.8: Experiment with local Data Store Memory input.

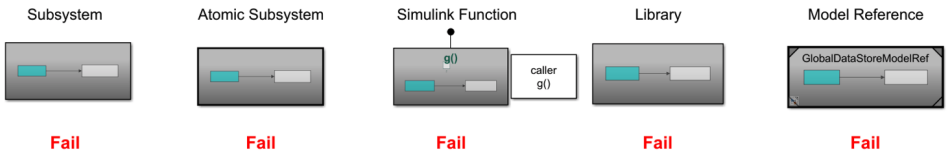


Figure 4.9: Experiment with global Data Store Memory input.

effectively supports information hiding by allowing developers to restrict its access via its scoping ability.

4.2.5 Code Generation

This section explores the code generation outcomes of the constructs. In most embedded development, particularly when adhering to AUTOSAR, Simulink models are used to implement individual software components, and the integration of components happens at the C code level. In this scenario, the generated code is of primary importance since the generated software

files are the integration units. Efforts to make the generated code more modular have been explored [Lublinerman and Tripakis, 2008; Tripakis and Lublinerman, 2018].

We compare the different constructs in terms of the C code they generate. Differences can arise depending on the complexity of the contained blocks, as well as how many times the exact same construct is used/called. Because of this variability, we examine how the code generated for the constructs in several scenarios, and for each construct we create a model such as that shown in Figure 4.10. The Subsystem blocks containing simple logic perform a signal Gain (multiplication in the C code) while the more complex Subsystem blocks contain a Switch based on the input (if-else branch in the C code). Then, we generate the code and use the built-in code mappings to trace blocks to C code. The following code generation outcomes were observed, and are also documented in the Embedded Coder [The MathWorks, 2020b] and Simulink Coder [The MathWorks, 2019] documentation. The generated code for each construct is provided in Appendix A.

Note that the observed code generation rules apply in general, however, Simulink Coder and Simulink Embedded Coder may perform further optimizations to the code, potentially resulting in inlined code. Moreover, we do not go into the finer Simulink Coder configurations or block parameters that impact code generation (e.g., storage class and function packaging). We use the blocks in their default configuration.

Subsystem The model used to observe code generation for Subsystem blocks is shown in Figure 4.10. In all cases, Subsystem blocks are flattened, thus no code is generated for the Subsystem block itself (i.e., the Subsystem does not

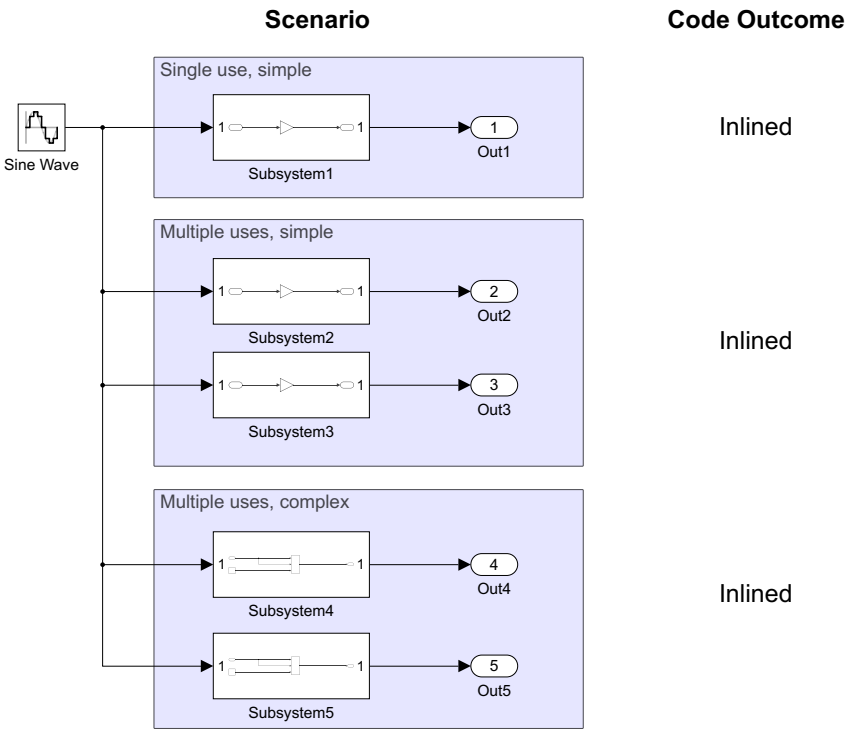


Figure 4.10: Code generation model for the Subsystem construct.

translate into a function). Instead, the contents of a Subsystem are repeatedly generated as inlined code throughout the model, usually resulting in code duplication.

Atomic Subsystem An Atomic Subsystem code can be generated in different ways, depending on how complex its internals are, and whether the Atomic Subsystem is used more than once in a model. If the Atomic Subsystem is trivial (e.g., Gain block only), it will be inlined, otherwise it will be generated into a function. Simulink attempts to recognize identical instances of the same Subsystem so it can generate function code for only one, increasing code reuse [The MathWorks, 2019].

Simulink Function In the case of an exported Simulink Function, it is generated as an external function with its own separate C module. Function Caller blocks are then translated into calls to the external functions of the module. If a Simulink Function is not exported, then it remains a local function in the module where it is defined. Fundamentally, a Simulink Function is akin to an Atomic Subsystem with added behaviour, so the C code of a scoped or local Simulink Function results in code similar to code generated for a Library with an Atomic Subsystem. Otherwise, if the Simulink Function is global, the generated code is comparable to a Model Reference.

Library When a Subsystem is placed in a Library, it will not result in a function in the code, however, nonvirtual subsystems will, as long as they are nontrivial and used more than once in the model. That is, a single use of a nonvirtual subsystem from a Library will result in inlined code, however, blocks used more than once will result in a single function with multiple calls.

Model Reference Model Reference blocks refer to a separate model and so are generated as separate code modules (with their own `.c`/`.h` files). The Model Reference blocks are then code generated into calls to the model via their external functions.

4.2.6 Comparison Summary

The comparison is summarized in Table 4.3. Although used the most frequently in industry, the default virtual Subsystem and Atomic Subsystem constructs have the disadvantage of not being reusable, not offering users the ability to scope the internal implementations, and generating code that

usually is inlined. Using Simulink Function blocks as a componentization construct leads to more reusable functionality in Simulink, as well as in the generated code. It is also possible to scope the implementation appropriately, which to our knowledge, is not supported by any other construct in Simulink. Therefore, the Simulink Function construct is a powerful means by which to decompose a model while also supporting encapsulation and facilitating information hiding. Library blocks and referenced Models, while reusable, do not have built-in capabilities to selectively scope contents and hide information. Ideally, information hiding (via scoping or otherwise) would be supported by the Simulink language. However, because it is not, modelling conventions or rules can be introduced, and we do this in Section 4.4. Although it was demonstrated that decomposition via the use of Simulink Functions most effectively supports encapsulation and thus information hiding, this approach may not suit all applications, for example, due to code generation needs. Depending on the design requirements of the system, other constructs may be more suitable. Section 4.3 further elaborates on some of the considerations when converting between componentization constructs.

4.3 Conversion and Limitations

Conversion between the different componentization constructs is possible. MathWorks provides a guide on converting Subsystems into Model References [The MathWorks, 2019], however it does not address other combinations. Converting a Subsystem, Atomic Subsystem, Model Reference, or Library block to a Simulink Function is possible, however, one must be

[5] e.g., Resettable Subsystem, Reset Function

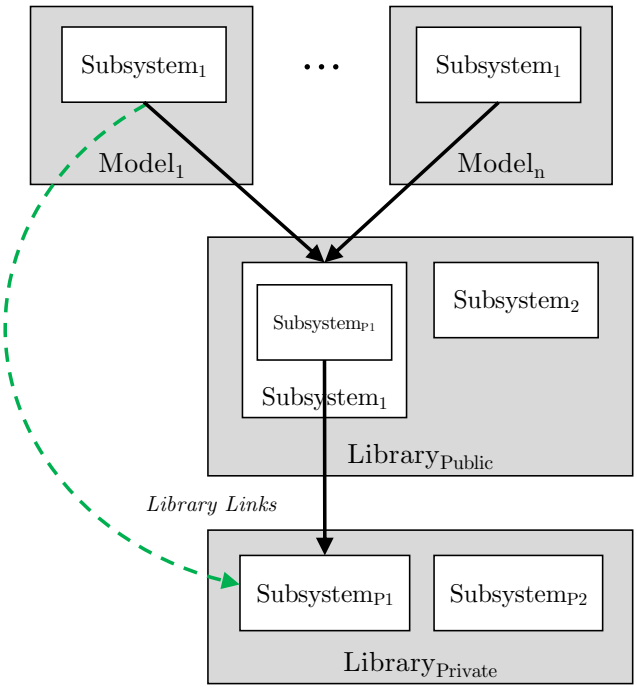
Table 4.3: Summary of the comparison of componentization constructs.

Construct	Reusable	Shared State	Limitation of Use	Code Generation
Subsystem	No	No	No	Inlined code
Atomic Subsystem	No	No	No	Separate function if used multiple times; otherwise inlined code
Simulink Function	Yes	Yes, by default. Can reset using other blocks[5]	Yes	Separate function (If exported, in separate module also)
Library	Yes	No	No	Separate function if nonvirtual and used multiple times; otherwise inlined code
Model Reference	Yes	Data Store only	No	Separate function, in separate module

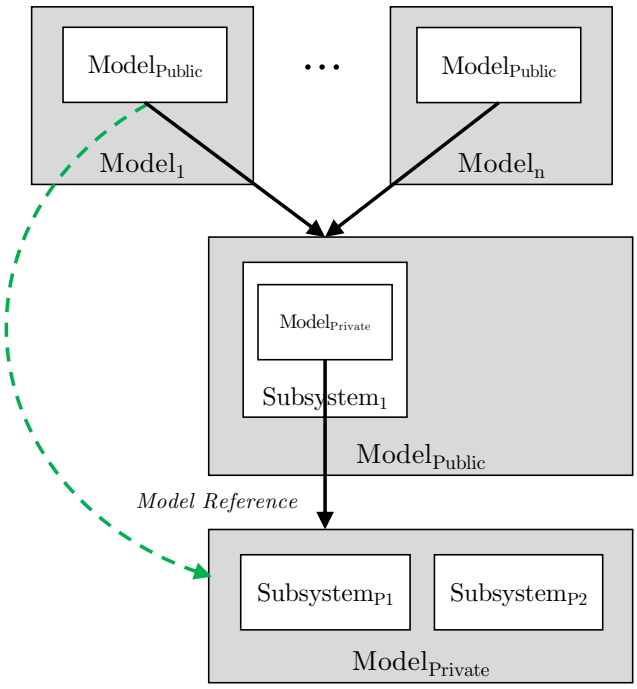
aware that there can be differences in the code that is generated. This may be important for some applications and should be taken into consideration. Moreover, shared state is treated differently. Thus, the constructs are not interchangeable without slight modification to the design. When converting to a Simulink Function, special consideration for blocks with state needs to be given, with a possible remedy being the exclusion of such blocks from the Simulink Function, or implementing resets. Also, MathWorks does not recommend using Simulink Functions for continuous systems, as they do not inherit continuous sample time [The MathWorks, 2019].

4.4 Conventions for Modularity

Conventions can be used to encourage best practices when it comes to information hiding, particularly when a language does not have built-in support, such as in the absence of a construct like the Simulink Function.



(a) Library hierarchy convention.



(b) Model reference hierarchy convention.

Figure 4.11: Conventions to support public/private functionality.

Next, we present a modelling convention alternative to modelling with Simulink Functions. The approach is shown in Figure 4.11a. Two libraries exist for separately storing public functions and private functions ($\text{Library}_{\text{Public}}$ and $\text{Library}_{\text{Private}}$, respectively). Developers then must adhere to a convention that allows them to use the functions from the public Library, but dictates that the private Library cannot be used directly. Only public functions are able to directly use private functions in their implementation. The disadvantage of this approach is that it relies on users to adhere to a convention, but does not actively enforce access restrictions. It is easily possible to bypass this convention, as shown by the dashed line in Figure 4.11a. Automated checks for these convention violations could provide a solution to this. A hierarchy of Model References can also be achieved in a similar fashion, as shown in Figure 4.11b. The difference between using a Model Reference versus a Library is that model referencing means including the entire model, instead of a specific block. Another well-known approach for supporting modularity is through the decomposition of a software system into *modules* via information hiding [Parnas, 1972a]. We will introduce this approach for Simulink in Chapter 5.

4.5 Chapter Summary

The comparison performed in this section lays the groundwork for achieving modularity in the Simulink language. In particular, it demonstrated that Simulink Function blocks are capable of encapsulating reusable implementations that can be made public or private by altering the Simulink Function block's scope. Two conventions for supporting modular design were also introduced,

which did not rely on the use of Simulink Function blocks. However, because Simulink Function blocks can be used to actively hide implementation details and enforce encapsulation, we recommend using them in a stronger approach. To this end, Chapter 5 presents the creation of a module structure for Simulink models, along with a formal interface definition.

Chapter 5

A Simulink Module Structure

Information hiding is a fundamental principle for modularizing software so that it is *robust with respect to change* [Parnas, 1972a; Parnas et al., 1985]. It aims to decompose a system such that each likely change (e.g., hardware changes, behaviour changes, software design decision changes [Parnas et al., 1985]) is treated as a "secret" and localized (hidden) in a single module. Surprisingly, information hiding and encapsulation have not been readily supported in Simulink [Bialy et al., 2016; Molotnikov et al., 2016; Ziegenbein et al., 2020]. Parnas criticized the widely used approach of decomposing a system in a "flowchart" manner, in which modules are simply major processing steps in the program [Parnas, 1972a]. As Simulink is a graphical modelling language, this is the de facto method of decomposition currently used. However, with the introduction of constructs in the language, in particular Simulink Functions (available since Simulink R2014b), it is possible to design models that break free from the data flow approach. This chapter presents a novel approach for decomposing Simulink models that supports information hiding via the use of Simulink Function blocks. In particular, a

Simulink module and a module's syntactic interface are defined. Well-defined interfaces are crucial for achieving modularity in designs. A syntactic interface should make clear *all* the communication and dependencies of a module and ensure that private information is not exposed on the interface. A visual representation to provide a model-level view of this interface is created. New modelling guidelines to support best practices using the new decomposition and interface concepts are established. A tool to support decomposition, interface views, and guideline checking is also developed.

Section 5.1 provides an overview of the literature related to model structuring, modularity, and interfaces in Simulink. Section 5.2 introduces the idea of a Simulink module and presents design principles to support modularity. Section 5.3 defines the notion of a module interface. A visual representation is also created in order to represent the interface in a model. Tying this all together, Section 5.4 presents new guidelines for structuring designs as modules based on Simulink Functions and interfaces. Section 5.5 introduces an open-source tool for supporting module creation, interface representation, and guideline checking. Finally, Section 5.6 concludes with a summary.

5.1 Related Work

In this section, related work on model structure and interfaces is summarized.

5.1.1 Model Structure

MathWorks is the authority when it comes to Simulink model structuring. The Simulink User's Guide [The MathWorks, 2019] provides a guide for

choosing the decomposition construct at the model level and compares three constructs—subsystems, libraries, and model referencing—according to how each supports the development process, model performance, component reuse, etc. Modularity is not examined explicitly, however, there is a discussion on the related concept of component reuse, which positions the Model Reference and Library constructs as well suited for reuse, but not Subsystems. Nevertheless, the ability to hide implementation details is not discussed in the guide. The Simulink User's Guide also provides recommendations for interface design when it comes to Bus usage, naming conventions, parameter partitioning, and explicit interface configuration [The MathWorks, 2019].

The MathWorks Advisory Board (MAB) proposes decomposition using Subsystem blocks also, but more specifically recommends structuring a model into several layers, as shown in Figure 5.1. This decomposition is broadly separated into top and bottom layers, with further decomposition into other layers optional and possible in different combinations. The top (or root) layer gives an overview of the feature being modelled as well as triggering information. The bottom layers can be comprised of a subfunction layer for organizing individual functional units, a control flow layer that deals with input processing and intermediate processing for the functional units, and a data flow layer that implements the actual control algorithm. A selection layer can also be included in order to select between different Subsystems of control algorithms.

Whalen et al. propose structuring Simulink models for the purpose of verification [Whalen et al., 2014], decomposing a system into models based on their role: functional, property (requirement), environment, or test input

Table 5.1: Model structure recommended by MAB [The MathWorks, 2020c].

	Layer Concept	Layer Purpose
Top Layer	Function layer	Broad functional division
	Schedule Layer	Expression of execution timing (sampling, order)
Bottom Layer	Subfunction layer	Detailed function division
	Control flow layer	Division according to processing order (input → judgement → output, etc.)
	Selection layer	Division (select output with Merge) into a format that switches and activates the active Subsystem
	Data flow layer	Layer that performs one calculation that cannot be divided

models. Furthermore, vertical decomposition separates each subsystem into its own file. This structure supports independent development and traceability.

Dajsuren et al. define metrics for modularity in Simulink models in terms of coupling (number of exchanged input/output signals) and cohesion (related functionality) for subsystems, ports, and signals [Dajsuren et al., 2013]. Coupling metrics include: *Coupling Between Subsystems*, *Degree of Subsystem Coupling*, *Number of input Ports*, *Number of output Ports*, *Number of input Signals*, and *Number of output Signals*. Cohesion metrics include: *Depth of a Subsystem*, *Number of Contained Subsystems*, and *Number of Basic Subsystems*. It is clear that the Subsystem is considered to be a module in this context.

Section 4.2 presented a thorough comparison of the five available Simulink componentization constructs for decomposition purposes [Jaskolka et al., 2020a]. A Simulink Function's ability to be scoped makes it unique because one can hide it from other parts of the model in which it resides, or

other models. As a result, we leverage Simulink Functions in our proposed module structure in Section 5.2.

5.1.2 Interfaces

Well-defined interfaces are an integral part of achieving modularity in designs. Commonly, a Simulink model's interface is considered to be comprised of the Inports and Outports of the top-level system [Dörr, 2017; Gerlitz et al., 2015], also called the *explicit* interface. Bender et al. concluded that *implicit* data flow is a crucial part of a Subsystem's interface, and go on to define a *signature* as a representation of the interface of a Simulink Subsystem that effectively captures both the explicit and implicit data flow between Subsystems [Bender et al., 2015]. We use a similar approach to define a module interface. Rau's work on Simulink model interfaces recommends simplifying the signal flow into and out of a model by using a Bus to group them together [Rau, 2002]. Masked Subsystem blocks are then used to encapsulate signal operations such as selection, conversion, and renaming. The drawback is a loss of direct visibility of data flow, so we will not incorporate it in our work. The use of pre-/post-condition contracts as verifiable interface specifications for Subsystems has also been proposed [Boström, 2011; Boström et al., 2007; Iwu et al., 2004]. While this design-by-contract approach provides a way of ensuring desired behaviour at the Subsystem level, our approach aims to document the interface syntax in a complete fashion at the Model level. Our interface definition can then be used to establish the sets of inputs and outputs between which constraints can be expressed. We discuss interface-related guidelines in Section 5.4.

5.2 A Simulink Module

A *module* is a component of a software system. It is a separate unit of a program that encapsulates closely related algorithms (e.g., functions and procedures) and data (e.g., data structures and variables) [Parnas, 1972a]. Encapsulation means restricting access to a portion of the module, such that certain elements are not accessible outside of the module, but can be manipulated via public elements revealed on the module's interface. In this section, the notion of a module in Simulink is introduced, drawing from the C analogy in Section 2.3.

Modular programming in C entails decomposing a system into separate modules [Srivastava et al., 2008; Oualline, 1997]. Each module consists of a source file that groups together definitions of related functionality and data, while a module's interface is described by its header file. A module's implementation should be considered private, or internal, to the module, and only those elements listed on the interface should be accessible to other modules which import the interface. The ability to selectively hide or expose functions is achieved via the use of the `static` keyword, as shown in Figure 5.1, where the functions `set` and `get` are public, while the static function `foo` and static variable `var1` are private. If another module wishes to use this module's public elements, it can do so by including the module's interface (i.e., its header file), and then making calls to public functions in the module, with parameter values from the calling program.

It is possible to use the same modular approach in Simulink. Although there are several componentization constructs in Simulink, we wish to use the construct that best supports encapsulation, and thus information hiding. In

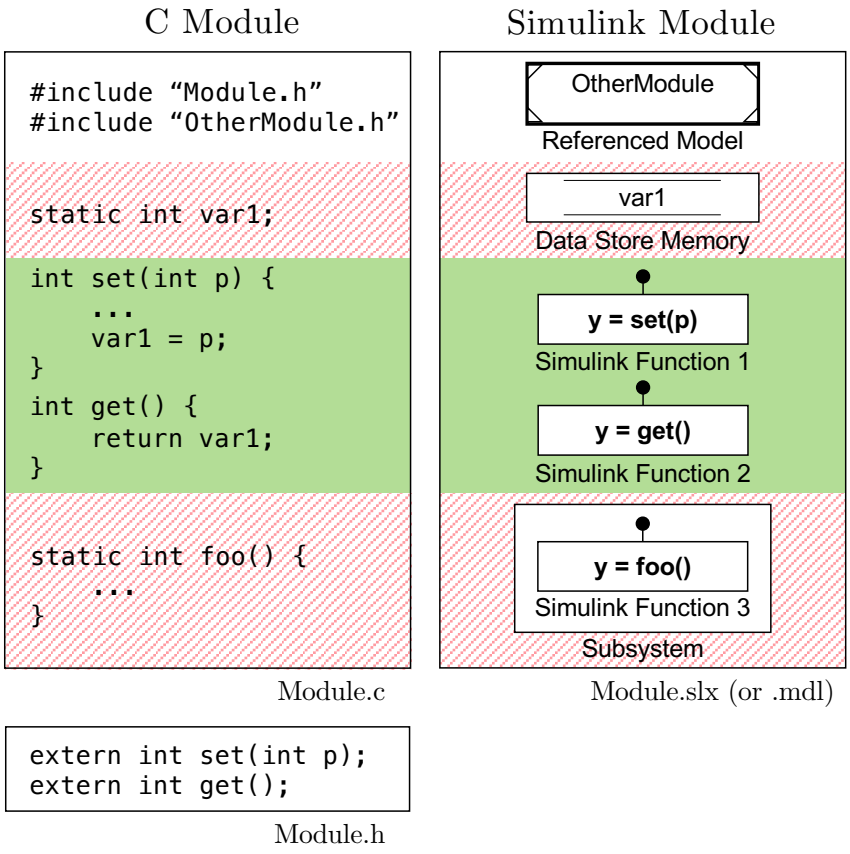


Figure 5.1: Module structure in Simulink based on C.

Chapter 4 we found that a Simulink Function's unique ability to be scoped enables one to effectively hide it from other parts of the model in which it resides, or other models (Table 4.2). Moreover, a Simulink Function helps to prevent some hidden data flow implicitly crossing its boundary. Simulink Functions prove to be best suited to help us build modular Simulink designs that can actively support encapsulation and thus facilitate information hiding. The result of the comparison of Simulink componentization constructs (i.e., [Jaskolka et al., 2020a]) leads to the proposed method for constructing modules in Simulink. Figure 5.1 illustrates how we can build Simulink modules based on the essential components of a module in C. The Simulink module uses:

- *Model References* to import other Simulink modules;
- *Data Stores* that are properly scoped as state data private to the module (alternatively, in some cases Unit Delay or Memory blocks may be used);
- *Simulink Functions* as functions exported by the module; and
- *Subsystems* to restrict a Simulink Function so that it is private to the module.

Like in C, the proposed Simulink modules are not object-oriented classes, and cannot be instantiated multiple times to create multiple objects. However, they make it possible to achieve *information hiding* in Simulink designs, by making modules that are separate, with the ability to selectively hide or expose their internals. These modules can be used to structure Simulink designs that are more robust with respect to anticipated changes.

5.3 A Simulink Module Interface

An interface is the set of services that each module provides to its clients [Ghezzi et al., 2002]. A syntactic interface is generally represented as a statement of elements and their properties that the module chooses to make known to a user or client modules. As shown in Figure 5.1, a Simulink module has no concept of an explicit interface like the header file provided in C. For this reason, we now define a Simulink module interface, so as to be able to extract it automatically from a Simulink module. A syntactic module interface contains:

- inputs — data the client needs to provide

- outputs — data the module promises to provide
- exports — functionality the module provides to users

It is important that only necessary information is disclosed to the client on the interface. The client needs to know what the module agrees to provide via the interface but does not need to understand the details of the implementation. As long as the interface remains the same, changes to the implementation can take place without affecting users in any way.

This interface is consistent with how many programming languages specify interfaces. Modern programming languages typically use keywords such as *Definition* or *Public* to delineate the "interface," and *Implementation* or *Private* to separate out the private implementation. The interface usually consists of only those elements that are necessary to make use of the exported functionality (e.g., constant, type, variable, and function prototype), but can also import elements that are needed in the interface itself. Note that it is usually possible to make module variables public, but in the spirit of information hiding, module variables (as opposed to "parameters") should never be exposed on the interface.

The prevailing view is that a Simulink *model's* interface is comprised of the Inports and Outports of the top-level system (e.g., [Dörr, 2017; Gerlitz et al., 2015]), as shown in Figure 5.2a. This is reflected in the *Interface Display* feature provided by Simulink, as shown in Figure 2.2a. The Interface Display aims to provide users with a better view of the system's interface [The MathWorks, 2019], and we can see that one Inport and three Outports are displayed around the edge of the model. However, a model's interaction with other systems and its environment can consist of additional

communication constructs that the Interface Display fails to show—in this case a Data Store Write, two To File blocks, and an exported Simulink Function. The Interface Display is thus insufficient in describing the actual interface of a model. Although this example is simple, the interface of a model can quickly become more difficult to understand due to the fact that these constructs can be placed *anywhere* in a model, potentially several layers deep. Thus, it can be difficult to "see" a Simulink module's interface and to understand what the module is exposing to other modules. The notion of a *signature* of a Simulink Subsystem can be used to represent the subsystem's interface [Bender et al., 2015]. It addresses the concerns with implicit data flow between subsystems by including in the subsystem's signature both the explicit data flow mechanisms (i.e., Inport/Outport) and the implicit data flow mechanisms (scoped or global Goto/From blocks and Data Store Read/Write/Memory blocks). We build on this idea to define the module's interface and represent it in the module in a similar way. This definition is then used to develop a visual representation in Simulink, as well as tool support to automatically extract and visualize it in a model.

5.3.1 Definition

A depiction of all the elements of a Simulink module's interface is shown in Figure 5.2b. Shared data is defined via Inport blocks, as well as others that are usually not considered, such as the From File and global Data Stores that are read. The interface also includes the module outputs via Outport blocks, as well as other blocks such as To Workspace and global Data Stores that are written to. Shared functions are Simulink Functions that are exported from a

module. The lines show all the possible data flow between a module and other workspaces

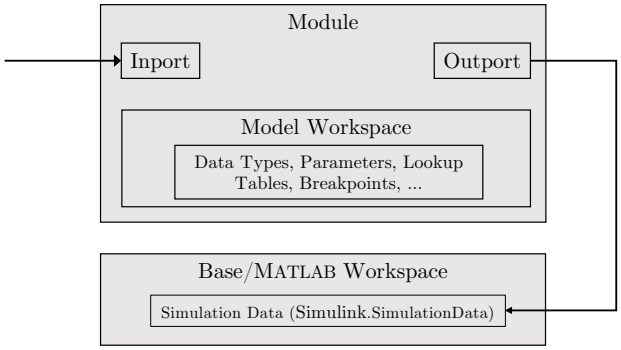
We can describe a Simulink module interface via standard set-theoretic definitions. These definitions not only help to make the interface precise, they also support the creation of tools. In particular, we use this definition to implement the interface extraction and representation in the Simulink Module Tool (Section 5.5).

A set containing n elements is written $\{a_1, \ldots, a_n\}$. We define sets using the usual set builder notation $A = \{a \mid P(a)\}$ where A is a set containing elements satisfying property P. The notation $a \in A$ denotes element a is contained in set A. The notation $A \subseteq B$ indicates that A is a subset of the set B. The union of A and B, denoted $A \cup B$, is the set containing elements in either A or B. The difference of A and B, denoted $A \setminus B$, is the set containing elements in A but not in B. The $\wedge$ symbol denotes logical conjunction. A function f with domain D and range R is denoted in the usual way: $f : D \to R$.

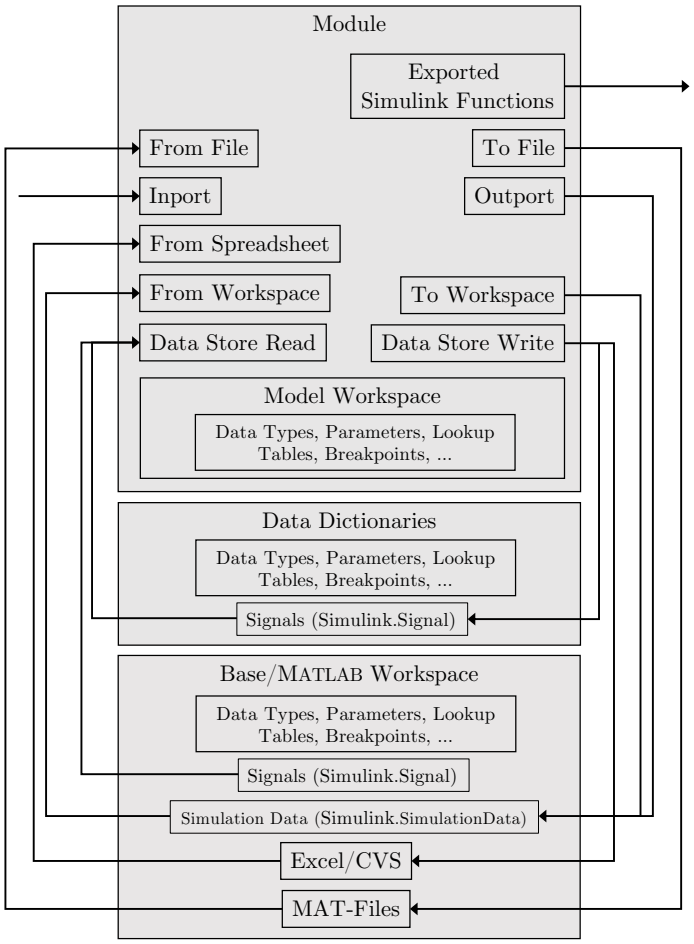
We define a Simulink module M simply as a set of blocks. We consider the block diagram (top-level) system itself to be a part of this set, so as to treat the system and subsystems the same. We abstract away the notion of signal flow between the blocks, as we are not concerned with intra-module communication (it was previously addressed [Bender et al., 2015]), but rather the inter-module communication not represented by conventional signals.

Definition 1 (Identifiers I).

- $\mathcal{B}(M)$ is the set of *all* blocks in the module M, that is, those at the top level as well as any that are contained within other blocks (regardless of hierarchy).



(a) Generally accepted view of the interface.



(b) All possible interface data flow.

Figure 5.2: Simulink interface data flow.

- $\mathcal{S}(M)$ is the set of all Subsystem blocks in the module M, as well as the root system, so $\mathcal{S}(M) \subseteq \mathcal{B}(M)$.

Definition 2 (Block Containment). For some blocks b and c, $b \in c$ denotes that b is wholly contained in c. It can also be said that b is a child of the container c.

Definition 3 (Parent Block). The function $parent : \mathcal{B}(M) \cup \{\mathsf{undefined}\} \to \mathcal{S}(M) \cup \{\mathsf{undefined}\}$ is defined,

$$parent(b) = \begin{cases} s & s \in \mathcal{S}(M) \wedge b \in s \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

Definition 4 (Root Block). The function $atRoot : \mathcal{B}(M) \to \mathbb{B}$, where $\mathbb{B} = \{\mathsf{false}, \mathsf{true}\}$, is defined,

$$atRoot(b) = \begin{cases} \mathsf{true} & parent(parent(b)) = \mathsf{undefined} \\ \mathsf{false} & \text{otherwise} \end{cases}$$

A block b is in the root system of module M when its parent in turn does not have a parent, or, block b has no defined grandparent.

Definition 5 (Identifiers II).

- *IP(M)* is the set of all Inport blocks
  - ⋆ *IR(M)* represents root level inports (*IR(M)* $\subseteq$ *IP(M)*) and is defined, $IR(M) = \{ir \mid ir \in IP(M) \wedge atRoot(ir)\}$
- *OP(M)* is the set of all Outport blocks

  - ⋆ *OR(M)* represents root-level outports ($OR(M) \subseteq OP(M)$) and is defined, $OR(M) = \{or \mid or \in OP(M) \wedge atRoot(or)\}$

- *FD(M)* is the set of all Simulink Function blocks
  - ⋆ *FG(M)* represents global functions ($FG(M) \subseteq FD(M)$)
  - ⋆ *FS(M)* represents scoped functions ($FS(M) \subseteq FD(M)$)
  - ⋆ *FL(M)* represents local functions ($FL(M) \subseteq FS(M)$) and is defined, $FL(M) = \{fl \mid fl \in FS(M) \wedge \neg\, atRoot(fl)\}$
- *TF(M)* is the set of all To File blocks
- *FF(M)* is the set of all From File blocks
- *FS(M)* is the set of all From Spreadsheet blocks
- *TW(M)* is the set of all To Workspace blocks
- *FW(M)* is the set of all From Workspace blocks
- *DS(M)* is the set of all global data stores
  - ⋆ *DSR(M)* represents global data stores that have a corresponding Data Store Read block
  - ⋆ *DSW(M)* represents global data stores that have a corresponding Data Store Write block

Definition 6 (Inputs). The inputs of module M, denoted $IN(M)$, is a tuple $\big(IR(M),\ FF(M),\ FS(M),\ FW(M),\ DSR(M)\big)$ of root-level Inport, From File, From Spreadsheet, From Workspace, and global Data Store Read blocks.

Definition 7 (Outputs). The outputs of module M, denoted $OUT(M)$, is a tuple $\big(OR(M),\ TF(M),\ TW(M),\ DSW(M)\big)$ of root-level Outport, To File, To Workspace, and global Data Store Write blocks.

Definition 8 (Exports). The exports of module M is the set of exported Simulink Function blocks, $EX(M) = FG(M) \cup \big(FS(M) \setminus FL(M)\big)$. Global Simulink Functions are included as they are always on the module interface. Scoped Simulink Functions are included if they are at root-level, i.e., they will be exported on the module interface.

Definition 9 (Interface). The interface $\mathcal{I}$ of a module M, denoted $\mathcal{I}(M)$, is a tuple $\big(IN(M), OUT(M), EX(M)\big)$ of inputs, outputs, and exports.

5.3.2 Limitations

Although unlikely, it is possible that a Simulink model can use custom code and data from outside sources via the use of model callbacks,[1] S-functions,[2] and C Functions.[3] Callbacks are used to automatically run commands when a model is opened or closed, for example, and could be used to dynamically define data for the model. S-functions are custom Simulink block implementations using MATLAB, C, C++, or Fortran, which could potentially read/write data externally via file I/O functions. C Function blocks allow models to call external C code. These three mechanisms are typically used by advanced Simulink users. Parsing the textual code used to apply these techniques are outside of the scope of this work, and thus are not reflected in the interface definition.

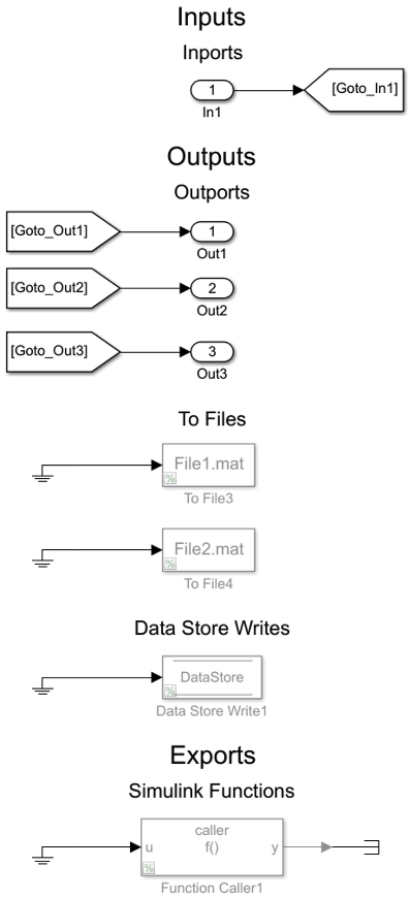
[1] https://www.mathworks.com/help/simulink/ug/model-callbacks.html
[2] https://www.mathworks.com/help/simulink/sfg/what-is-an-s-function.html
[3] https://www.mathworks.com/help/simulink/slref/cfunction.html

5.3.3 Representation

We created a visual representation of the interface within the model file to provide an easy to understand view of the module's interface based on Definition 9. The visual representation is placed in a Simulink module at the root, to the left of any other elements at that hierarchical level. It contains labelled sections corresponding to Definitions 6–8. Where possible, the interface is represented using commented out blocks, thus preventing it from having any behavioural impact on the module or adding new code during code generation. If modifying in the module is not possible, the interface can be represented in text form in the MATLAB Command Window. This textual representation can also be used for automatic documentation generation. In the textual description of the interface, each element's full path name, data type, dimensions, and sample time are listed. The tool we developed, the Simulink Module Tool (Section 5.5), supports the automatic creation of an interface in both visual and textual forms. The visual interface for Figure 2.2a, as generated by the tool, is shown in Figure 5.3a, and has four elements that the MathWorks Interface Display does not show. The textual interface is shown in Figure 5.3b. This is a concrete application of the definition of a module interface as presented in Section 5.3.1.

5.3.4 Benefits

There are several practical benefits and situations in a software engineering methodology where an interface in a Simulink module is beneficial. We describe these use cases in what follows.



(a) Visual interface representation.

```
Inputs
------
Inports:
  SmallExample/In1, uint16, 1, 1

Outputs
-------
Outports:
  SmallExample/Out1, Inherit: auto, 1, 1
  SmallExample/Out2, Inherit: auto, -1, 1
  SmallExample/Out3, Inherit: auto, -1, 1
To Files:
  SmallExample/To File1, Timeseries, N/A, -1
  SmallExample/To File2, Timeseries, N/A, -1
Data Store Writes:
  SmallExample/Data Store Write, uint16, 1, 1

Exports
-------
Simulink Functions:
  SmallExample/Simulink Function,
    In: uint16, 1, -1
    Out: uint16, 1, -1
```

(b) Textual interface description.

Figure 5.3: Interface representations for Figure 2.2a, as generated by the Simulink Module Tool.

Development Passing information that is too detailed, unnecessary, arbitrary, or potentially changeable, violates software design principles. Clear interfaces help developers review them critically and examine whether their constituents are, for example, likely to change, too low-level, or unnecessary.

Collaboration The presence of an interface is also invaluable in understanding a module for the first time, particularly when it originates from a different developer or source. If an interface is provided, the developer

can use the module in a black-box fashion, without taking the time to understand the internals of the design.

Testing With an easy to identify interface, all the module inputs and outputs are evident to a tester. When using a third-party Model-in-the-Loop (MiL) testing tool, such as Reactis by Reactive Systems, a developer can quickly identify module inputs/outputs that may not be included automatically by the tool. Some testing tools neglect to automatically create input vectors, or record input/output vectors for constructs outside of Inport/Outport blocks. A developer can quickly identify module inputs/outputs that need additional test points to be created prior to testing in order to record the results for omitted inputs/outputs. For example, Data Store Read and Data Store Write blocks may not be exercised adequately. With a complete and visual interface, the tester can easily find these on the interface and harness them [Bender et al., 2015]. Also, depending on the testing tool, blocks such as From Spreadsheet, To File, etc. may not be supported at all. In such cases, it is useful if they are made evident so that the user can deal with them appropriately, resulting in better coverage.

Production Several constructs that can be on a module interface are not recommended for a module that is to be used to generate production code (e.g., To File and From Workspace). However, these constructs are useful to developers during module development and simulation. An interface will capture such constructs, and empower developers to use them with the knowledge that they will be easy to identify and remove once a module is ready to be transitioned to production.

Documentation, Refactoring, and Maintenance Documentation of Simulink models is often deficient [Schaap et al., 2018; Pantelic et al., 2019]. It can be difficult for developers to understand the overall functionality of complex models, as well as how they interact with other models. An interface makes this clear at the root level of the module, saving the developer from navigating to other levels. Structuring a module such that it always contains an up-to-date interface eases documentation efforts and supports the concept of "self-documenting" software. Textual representation of the interface is particularly useful for automatic documentation generation.

5.4 Modelling Guidelines

The use of Simulink Functions and a syntactic interface help create Simulink modules. Guidelines are useful for further supporting good practices when using these approaches. In this section, we discuss existing modelling guidelines for Simulink, and present new ones to address gaps where current guidelines fall short. The Simulink Module Tool provides automated compliance checking for these guidelines and is discussed in Section 5.5. A user is able to select one or more of these guidelines and any violating blocks will be reported. To the best of our knowledge, no other tools support these guidelines.

5.4.1 Simulink Functions

MathWorks is the de facto authority on best practices for designing with Simulink. Their advisory boards [The MathWorks, 2020c] provide the most influential guidelines, but currently none address Simulink Function scoping.

The Motor Industry Software Reliability Association (MISRA) Simulink guidelines also pre-date Simulink Functions [The Motor Industry Software Reliability Association, 2009]. Recommendations on using Simulink Functions to promote best practices for supporting modularity and information hiding are introduced below. These guidelines are widely accepted in other languages to increase understandability, promote maintainability, and reduce errors, thus, we adapt them for Simulink R2014b and newer releases.

Guideline 1 (Simulink Function Placement). Place the Simulink Function block in the lowest common parent of its corresponding Function Caller blocks. Do not position the Simulink Function in the top layer without a reason. Avoid placing Simulink Function blocks below their corresponding Function Caller blocks.

Guideline 2 (Simulink Function Visibility). Limit the *Function Visibility* parameter of the Simulink Function block's trigger port to *scoped* if possible.

In textual programming languages, it is good practice to ensure variables and functions are declared at the minimum scope from which their identifiers can still reference them [Martin, 2008]. This promotes readability, reliability, and reusability of the code [Carnegie Mellon University, 2020]. In Simulink, the same treatment is recommended for Data Store Memory blocks and Goto blocks, in order to support code comprehension, maintenance, as well as to avoid unintended access [Pantelic et al., 2018; The MathWorks, 2020c; The Motor Industry Software Reliability Association, 2009]. For these reasons, we introduce the two aforementioned guidelines for Simulink Function blocks.

Guideline 1 describes how to position Simulink Function blocks in order to minimize their accessibility both inside and outside the module. This is

achieved by placing Simulink Function blocks as low as possible in the hierarchy, while still allowing any function calls to reference their corresponding Simulink Function block without added name qualifiers. The exception to this occurs when the intent is to associate a Simulink Function with its parent subsystem. This may be to increase the reusability of the subsystem itself, so the Simulink Function is encapsulated by that subsystem, even though Function Callers may be present above it in the hierarchy.

The hierarchical placement of a Simulink Function can also affect its presence on the module's interface. If it is *scoped* and placed at the root, it will be externally accessible by other modules. A similar treatment for a Simulink Function's *Function Visibility* parameter is recommended in Guideline 2. It should be set to its most restrictive setting possible, unless otherwise required.

Guideline 3 (Simulink Function Shadowing). Do not place Simulink Functions with the same name and input/output arguments within each other's scope.

Function overloading occurs when multiple definitions of a function exist with the same name but different input or output arguments. Simulink does not allow a Simulink Function to be placed in a Subsystem that already contains a Simulink Function with the same name. However, if the placement of a Simulink Function is at a different hierarchical level than another of the same name, one can define functions that shadow/mask each other. Since scoping rules for Simulink Functions are complex, and users may be unaware of a naming collision, it is best to avoid situations where more than one function with the same name and arguments is accessible. The MAB

guideline jc_0791 recommends a similar treatment for data stored in multiple workspaces [The MathWorks, 2020g].

5.4.2 Interfaces

The Simulink User's Guide discusses good practices for interface design, including Simulink subsystem interfaces [The MathWorks, 2019]. The guidelines provide information about where model objects/data *can* be stored, but provide no real guidance on where they *should* be stored. Moreover, the use of constructs that contribute to hidden data flow into or out of the model is not addressed. The MathWorks Simulink Check provides guidelines for "high integrity systems modelling" for models that must comply with DO-178C/DO-331, ISO 26262, and other standards [The MathWorks, 2020i]. One guideline recommends that top-level Inport blocks must have *data type*, *port dimensions*, and *sample time* parameters populated. This is good practice in general and will assist in making the details of the interface data flow clear.

MAB provides a single guideline regarding interfaces, recommending the enabling of strong-typing in Stateflow charts. This is not directly useful for examining the model's top-level interface. MAB also provides a guideline that lists prohibited Simulink blocks, including To File and To Workspace blocks in control models [The MathWorks, 2020c]. Similarly, the Embedded Coder User's Guide describes which Simulink blocks support C code generation, and provides details on how certain blocks are treated during code generation [The MathWorks, 2020b]. In particular, the blocks described in our proposed definition of an interface are treated as follows.

- *Supported*: Inport/Outport, Data Store Read/Write, Model Reference, Library, Simulink Function, Function Caller
- *Ignored*: To Workspace/From Workspace
- *Not recommended for production*: To File/From File, From Spreadsheet

Although To File, From File, and From Spreadsheet blocks are not recommended for *production*, developers may use them during *development* because they are valuable for prototyping and logging purposes. Thus, To File, From File, and From Spreadsheet may be represented on the interface. When using these blocks for prototyping, an interface that highlights these constructs will help in identifying them so they can be removed once the design is finalized. This approach will help support the Embedded Coder guideline.

Guideline 4 (Use of the Base Workspace). Do not use the base workspace for storing, reading, or writing data on which a module depends. Instead, place data in either the model workspace, if it is used in a single module, or a data dictionary if it is shared across modules.

A likely change for a module that is used for code generation is that it will change workspaces, from being situated in the base workspace of the Simulink development environment to being flashed onto the target embedded device. One can anticipate and prepare for this future change by creating a stable interface from the first stages of development. This not only minimizes the need for changes later on, but can also reduce dependencies. This is achieved by restricting the use of interface elements that are used for prototyping or that do not support code generation. In particular, developers should avoid

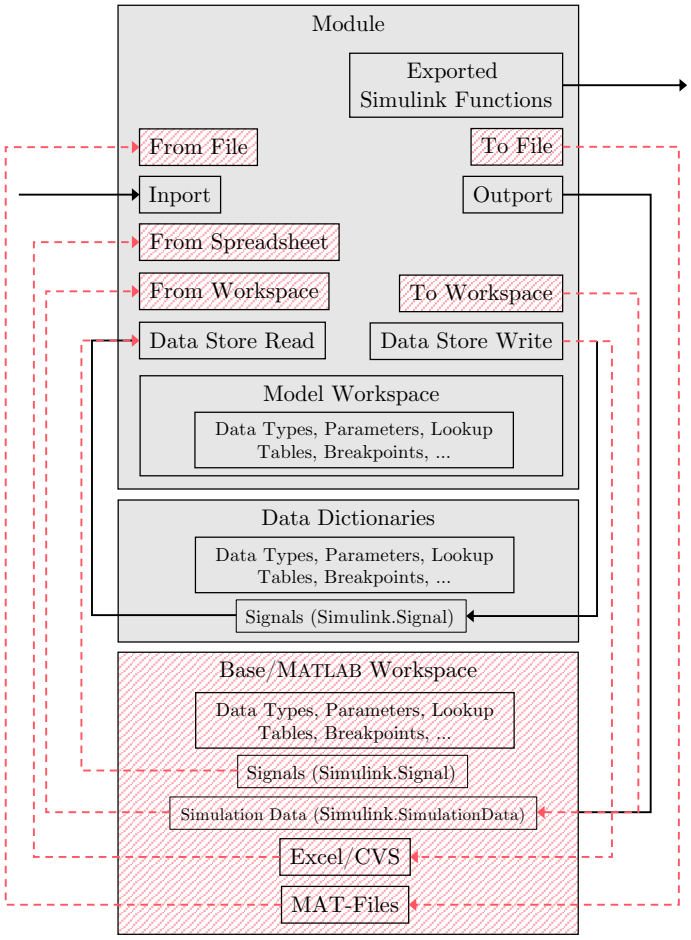


Figure 5.4: Restricted interface elements dashed/crossed out, per Guideline 4 for production-ready models.

using the base workspace for storing, reading, or writing data (including types, signals, etc.). Naturally, it follows that the use of blocks that read/write to the base workspace (e.g., To File/From File) should be avoided, unless they are placeholders for root-level Inport/Outport blocks that will be added in the future. Also, it may be the case that these restricted blocks are used for testing purposes (e.g., reading test vectors to exercise the model). In this case, a test harness model should be created to separate the testing-related blocks and allow the module to remain production-ready.

To see Guideline 4 applied, Figure 5.4 has restricted items dashed/crossed out, such as the base workspace and associated constructs. As a result, the data flow has been simplified significantly, with the dashed lines showing data flow that is eliminated. Interestingly, MAB explicitly prohibits the use of To File and To Workspace blocks, but recommendations for their counterparts, the From File and From Workspace blocks as well as the From Spreadsheet block, are not provided.

5.5 The Simulink Module Tool

The Simulink Module Tool was developed to assist with applying our approach for constructing Simulink modules as described in Section 5.2, generating the interface defined in Section 5.3, and checking compliance to the guidelines proposed in Section 5.4. It is open-source and available on the MATLAB Central File Exchange, GitHub, and directly within MATLAB using the *Add-On Explorer*. This tool is an extension to the Simulink environment, and adds new options directly into the Simulink Context Menu (demonstrated in Figures 5.5–5.7 and 5.9). The capabilities of the tool are described in the following sections.

5.5.1 Subsystem to Simulink Function Conversion

The tool automatically converts Subsystems into Simulink Functions. In a Simulink model, one can right-click on a Subsystem, choose *Convert Subsystem*, specify the scope, and the tool will automatically replace it with a Simulink Function. This is shown in Figure 5.5. This action encompasses changes to the Subsystem, Trigger, and Inports/Outports. The pseudocode of

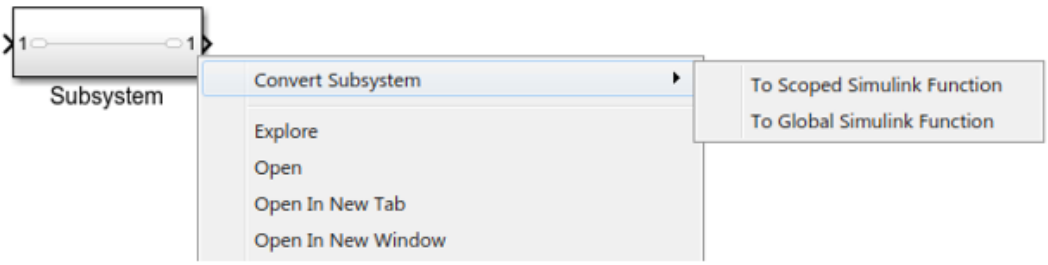


Figure 5.5: Simulink Module Tool: Convert a Subsystem into a Simulink Function.

the conversion algorithm is presented in Algorithm 1. We used this functionality throughout the case studies in Chapter 6 to greatly speed-up the migration of existing designs to the Simulink module structure.

In the case where ones chooses to convert an existing design to a module structure, the MathWorks automated theorem proving toolbox Simulink Design Verifier (SDV) can be used to prove that the design before and after changes is behaviourally equivalent. Note that a limitation to our module approach may arise when converting between componentization constructs that contain blocks with discrete or continuous states, because while Simulink Function blocks have persistent state between function calls, other constructs may not share state. This limitation is elaborated on in 6.4. To avoid this, one can leave out stateful blocks from Simulink Functions and ensure behaviour is the same via SDV proof.

5.5.2 Scope Changes

The tool converts between the different kinds of scoping for Simulink Functions, so the user does not have to be concerned with remembering the complex scoping rules regarding *Function Visibility* and placement. The user simply

Algorithm 1 Converting a subsystem into a Simulink function.

1: **procedure** SUBTOSIMFCN(ss, f_name, f_scope)
Ensure: *ss* is a subsystem
Ensure: *f_name* is a valid function name
Ensure: *f_scope* is a valid function scope
▷ Configure the Subsystem as a Simulink Function
2: *ss.LinkStatus* ← *none*
3: *ss.TreatAsAtomicUnit* ← *on*
4: Add trigger block *t* into *ss*
5: *t.TriggerType* ← *function-call*
6: *t.IsSimulinkFunction* ← *on*
7: *t.FunctionName* ← *f_name*
8: *t.FunctionVisibility* ← *f_scope*
▷ Convert inports to input arguments
9: *allInports* ← inports of *ss*
10: *allInportsParams* ← parameters of *allInports*
11: Replace *allInports* with ArgIn blocks
12: Fix *allInportsParams* that are incompatible with Simulink Functions
13: ArgIn parameters ← *allInportsParams*
▷ Convert outports to output arguments
14: *allOutports* ← outports of *ss*
15: *allOutportsParams* ← parameters of *allOutports*
16: Fix *allOutportsParams* that are incompatible with Simulink Functions
17: Replace *allOutports* with ArgOut blocks
18: ArgOut parameters ← *allOutportsParams*

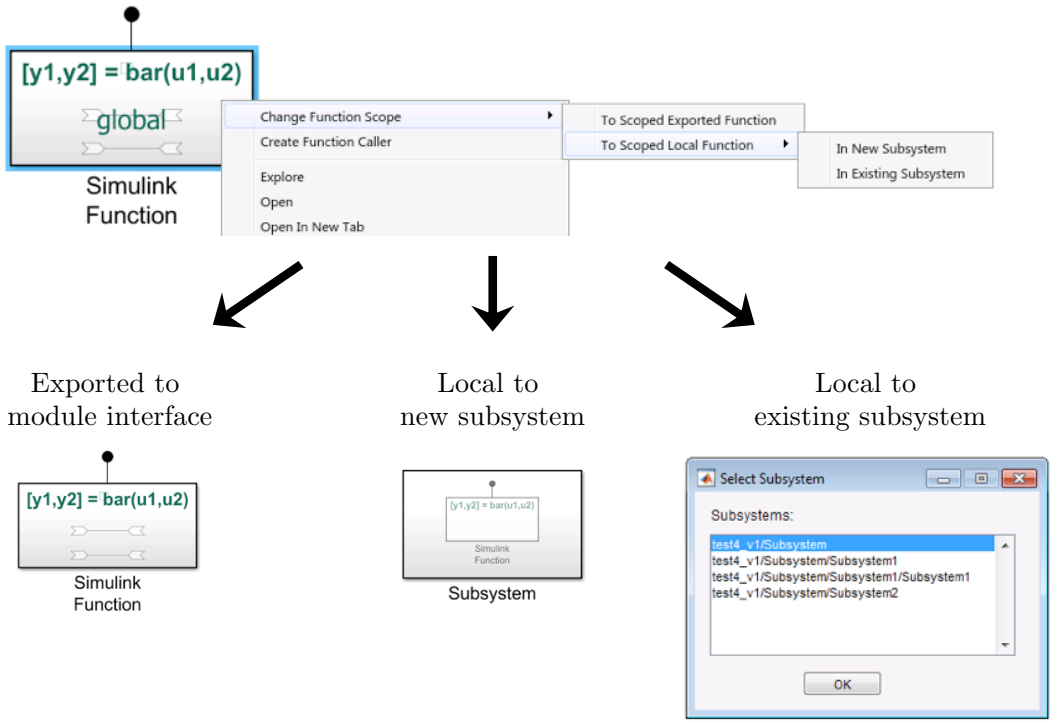


Figure 5.6: Simulink Module Tool: Change the scope of a Simulink Function.

needs to right-click on a Simulink Function, select *Change Function Scope*, and choose how to scope the function. This is shown in Figure 5.6.

5.5.3 Function Calling

The tool assists users in calling Simulink Functions that are in scope, with their appropriate qualifiers. Right-clicking in the model and then selecting *Call Function...* from the Context Menu displays a listbox showing Simulink Functions that can be called from that location. These steps are shown in Figure 5.7. In order to view a similar list from MathWorks, one must first manually add a Function Caller to the model, and then click on the *Function prototype* field, which is shown in the centre of Figure 5.8. Moreover, the Simulink Module Tool list is more accurate than the information that

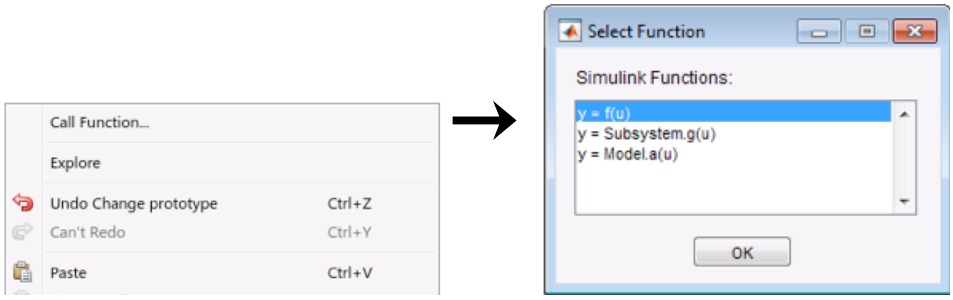


Figure 5.7: Simulink Module Tool: Call Simulink Functions that are in scope.

Simulink provides when configuring a Function Caller, as shown in the example in Figure 5.8. On the left, a Simulink Function *foo* is stored in an Atomic Subsystem, making it inaccessible at the current level of the hierarchy. When trying to configure the Function Caller shown, Simulink lists the Simulink Function in the list of callable functions (centre window). However, selecting this Simulink Function will result in a simulation error that states the Simulink Function is inaccessible. On the right, the Simulink Module Tool does not list this Simulink Function. The Simulink Module Tool provides a more accurate list, and does not require that a Function Caller block is first added.

5.5.4 Automatic Function Configuration

Making a selection from the callable Simulink Function list, as described in Section 5.5.3, creates a Function Caller and automatically populates its *Function prototype*, *Input argument specifications*, and *Output argument specifications* parameters. Simulink does not populate these fields automatically, and it can be tedious to specify for a function with many inputs and outputs.

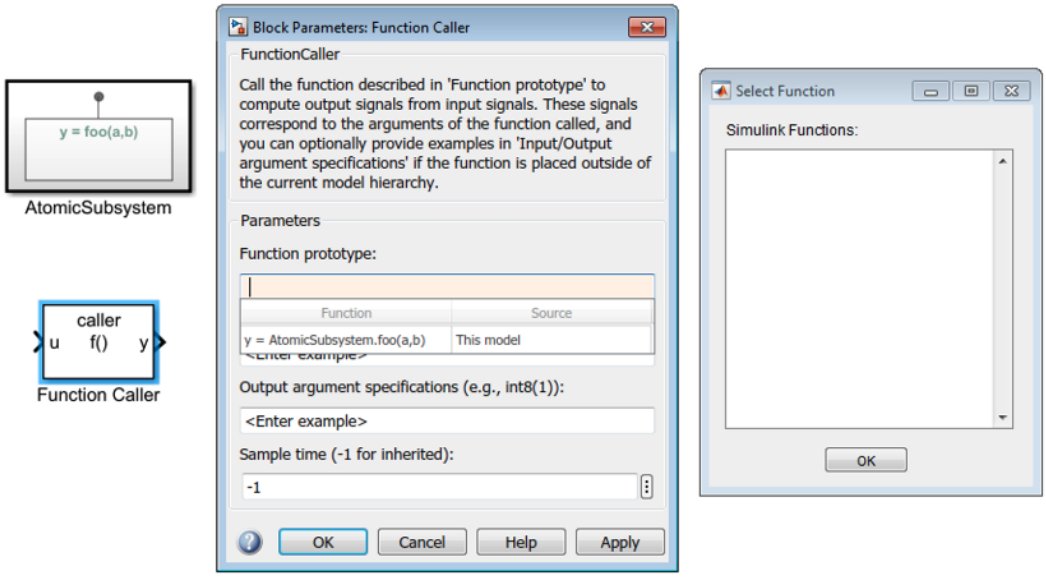


Figure 5.8: Simulink lists an inaccessible Simulink Function in the list of callable functions (centre), whereas the Simulink Module Tool does not (right).

5.5.5 Interface Generation

The syntactic interface (Section 5.3) for a Simulink module can be automatically generated. The user simply needs to right-click in the model and select the *Interface* > *Show Interface* option to represent it visually in the model, or *Print Interface* to textually print it to the Command Window (Figure 5.9). Both of these two views were shown in Figure 5.3.

When the interface is to be represented in the model, the tool will add its representation into the model. The tool is also capable of updating the interface representation when requested by the user. It is also possible to delete the interface representation to revert the model to its original state, by selecting *Delete Interface*, as shown in Figure 5.9.

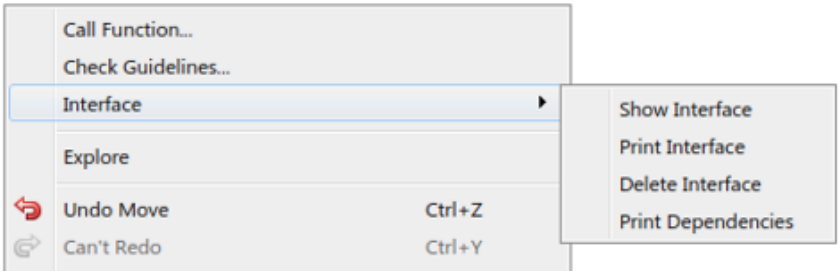


Figure 5.9: Simulink Module Tool: Generate interface and dependency views.

5.5.6 Dependency Viewing

Module dependencies, such as Model Reference, Library, and data dictionaries, can be detected by the tool, and summarized for the developer. This is useful for ensuring that the necessary definitions/files are available in order to compile and simulate. This is option available by right-clicking in the model and selecting *Interface* > *Print Dependencies*, as shown in Figure 5.9.

5.5.7 Guideline Checking

The four guidelines presented in Section 5.4 can be selected (Figure 5.10), automatically checked, and lists of violations are returned to the user as shown in Figure 5.11.

5.6 Chapter Summary

Previously, it was determined that a Simulink model's structure and interface were changing at a high frequency compared to other model elements. We proposed the well-known approach of modularization via information hiding to minimize change propagation. In comparing the available Simulink decomposition constructs, we found that Simulink Functions were best suited

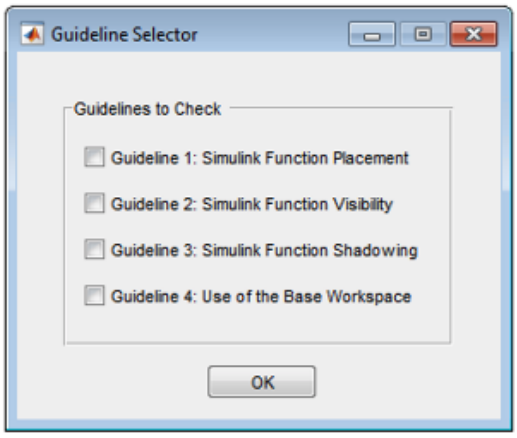


Figure 5.10: Simulink Module Tool: Check module guideline compliance.

```
Guideline 1 'Simulink Function Placement' violations:
       Example/Simulink Function can be moved to Example/Subsystem
       Example/Simulink Function1 can be moved to Example/Subsystem2/Subsystem1

Guideline 2 'Simulink Function Visibility' violations:
       Example/Simulink Function
       Example/Subsystem/Simulink Function1

Guideline 3 'Simulink Function Shadowing' violations:
       Example/Subsystem2/Subsystem1/Simulink Function1 is shadowed by:
              Example/Subsystem/Simulink Function1

Guideline 4 'Use of Base Workspace' violations:
       Example/From File
```

Figure 5.11: Simulink Module Tool: Example output of guideline checks.

to support encapsulation while also creating reusable designs. Simulink Functions also have the unique ability to be scoped as public/private functionality. As a result, a Simulink module concept was created that leverages Simulink Functions. Additionally, a Simulink module syntactic interface was defined. Four modelling guidelines were also proposed to further encourage good practices when using these approaches. Finally, the Simulink Module Tool was presented in order to automate and further support developers when leveraging the aforementioned approaches. Altogether, these contributions present a novel way of enabling modularization using information hiding in the Simulink language.

Chapter 6

Case Studies

This chapter describes how the concepts introduced in Chapter 5 were applied to production industrial models. First, evaluation methods for both studies are introduced in Section 6.1. Then, case studies in the aerospace and nuclear domains are provided in Sections 6.2 and 6.3, respectively. Limitations and workarounds are discussed in Section 6.4.

6.1 Evaluation Methods

Through the use of our proposed Simulink module structure, we aim to achieve designs that are robust with respect to change. In order to perform an evaluation of the proposed module structure, we sought to objectively quantify the improvement to modularity and information hiding. This was done by evaluating characteristics that are widely considered effective indicators of design structure and modularity of large systems, such as coupling and cohesion. In addition, we evaluated the approach by examining

potential impacts to the designs in terms of complexity, structural coverage, and performance.

6.1.1 Design Equivalence

We want to ensure that the restructured models are behaviourally equivalent to the original designs, and that no unintended behaviour was introduced as a result of the restructuring. To do this, we use verification to formally prove equivalence between specification outputs. MathWorks provides the Simulink Design Verifier (SDV)[1] toolbox that we use to perform a formal analysis on models to either prove or disprove specified properties.

The process is shown in Figure 6.1. First, we begin with the Simulink designs before and after restructuring. A verification model is created that references both designs. We then instrument the verification model with proof objectives which we want to remain invariant over the entire execution of the verification model. To ensure equivalent behaviour, we specify that each output of the original system must be equivalent to the corresponding output of the restructured system. SDV is then executed in "property proving" mode to formally prove the specified properties. This ensures that the outputs are always identical, for each time step. A report of results is produced. It states whether the objectives were satisfied or not satisfied. If they were not satisfied, a counter example is provided. If all properties are successfully proven for our verification model, this means that the before/after systems are behaviourally equivalent. For both the case studies presented, we formally proved that the designs before and after restructuring were indeed behaviourally equivalent.

[1] https://www.mathworks.com/products/simulink-design-verifier.html

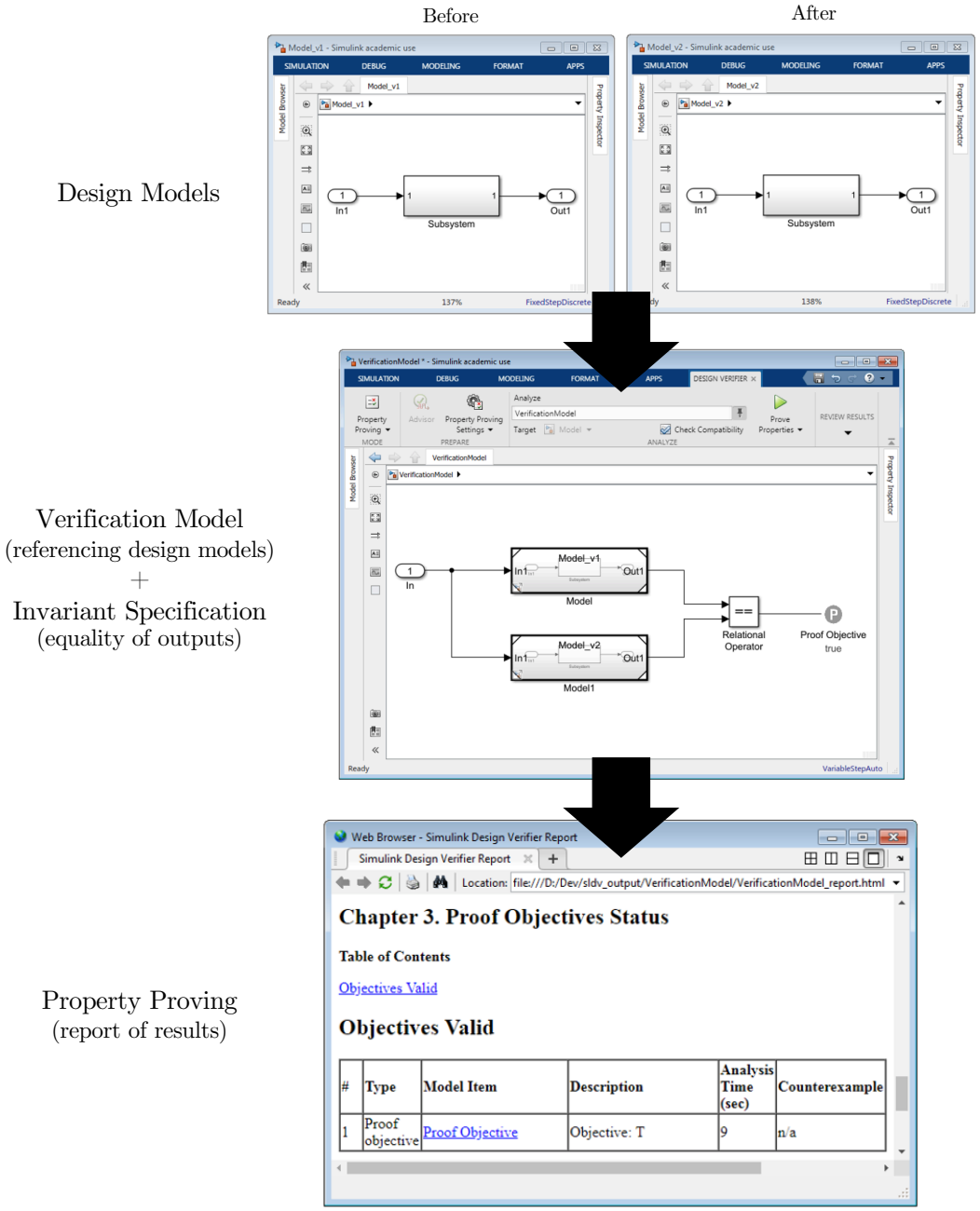


Figure 6.1: Verification of equivalence between model versions.

6.1.2 Information Hiding

Although directly measuring information hiding has been attempted [Rising and Calliss, 1994], no metric has been widely accepted in either academia or industry. As a result, we use a qualitative analysis to reason about the effectiveness of our approach in supporting information hiding in the system. Firstly, given a list of secrets that are hidden appropriately in a module, it should be the case that changes to the secret will not impact any other modules of the system [Parnas et al., 1989]. Parnas referred to this as the *changeability* of the system and used this as a way of demonstrating the effectiveness of information hiding [Parnas, 1972a]. Secondly, use of the "hidden" internal implementation of the module should be restricted, that is, private to the module in which it is defined [Parnas et al., 1989]. We evaluate the first of these two scenarios by performing the same change on the before and after systems, and documenting the parts of the system that needed to be modified in order to implement the change. The expectation is that the system that was designed without information hiding in mind will require changes to several parts of the system, while the modularization approach that we propose effectively restricts changes to a single module. The second scenario is tested by creating a probe model that attempts to access internal implementations that should be hidden from other modules.

6.1.3 Interface Complexity

The complexity of a system is often attributed to the interactions, or interfaces, between the system's components. This complexity directly impacts the reusability, testability, and maintainability of the components.

Interfaces are essentially the links between modules that allow for separation of concerns. Minimal and stable interfaces are integral to achieving information hiding. As discussed in Section 5.3, the interface of a Simulink model is generally considered to be comprised of Inport and Outport blocks, when in fact many other Simulink elements also contribute to the interface. This is also reflected in the tools currently available for interface complexity checking, such as the Simulink Check Metrics Dashboard[2] and Model Engineering Solutions GmbH (MES) M-XRAY.[3] As a result, we used the Simulink module interface definition in Section 5.3 to provide a complete syntactic description of a module's interactions. The Simulink Module Tool automatically generates interface information, as well as dependency information, for a Simulink module (Section 5.5). Together this information provides an overall view of the many interactions that exist in a Simulink system. We compared the interfaces of the before and after system to understand how the new module structure impacted interface complexity.

6.1.4 Coupling and Cohesion

Coupling and cohesion are well-known as indicators of the quality of a program decomposition, and are related to the concept of information hiding [Stevens et al., 1999]. Coupling is a measure of the interconnections between modules, and increases as the complexity of the interfaces between modules increases [Stevens et al., 1999]. While information hiding aims to hide the implementation details of a module, coupling measures how much another module is reliant on another module. Minimizing the coupling of a

[2] www.mathworks.com/help/slcheck/ug/collect-and-explore-metric-data-by-using-metrics-dashboard.html

[3] www.model-engineers.com/en/quality-tools/mxray/

module makes it more robust with respect to changes because it reduces the connections by which changes and errors can propagate [Stevens et al., 1999]. Thus, a good design that implements information hiding will also exhibit low coupling. Designs with low coupling and high cohesion lead to software that is more reliable and more maintainable [Fenton and Bieman, 2014]. Cohesion is a measure of the relationships of the elements *within* a module, with the aim of ensuring that module elements are highly related to each other [Stevens et al., 1999]. Cohesion also supports information hiding by ensuring that the contents of a module are strongly related to one secret. In the context of Simulink, coupling and cohesion are typically defined on a single Simulink model, based on the interactions of the contained blocks [Olszewska, 2011], or specifically Subsystem blocks [Gerlitz and Kowalewski, 2016; Dajsuren et al., 2013]. There is a lack of system-level metrics for coupling and cohesion in the Simulink environment, and in turn, an absence of tools that automatically measure these qualities. Consequently, we manually analyzed the impact to coupling and cohesion in our case studies.

6.1.5 Cyclomatic Complexity

The effort needed to maintain a software system is related to the complexity of the system. Cyclomatic complexity is the most widely-used metric in industry for gauging the structural complexity of software [Ebert et al., 2016]. Cyclomatic complexity measures the amount of decision logic in a program, or more specifically, the number of linearly independent execution paths through a program [McCabe, 1976]. McCabe created this metric to

provide a way of reasoning about the modularity and maintainability of a program [McCabe, 1976]. It is indicative of the maintainability of software [Banker et al., 1989; Gill and Kemerer, 1991; Watson and McCabe, 1996], which is our ultimate goal in supporting information hiding and modularity. The cyclomatic complexity metric has also been adapted for Simulink [Olszewska et al., 2016; Model Engineering Solutions, 2020; The MathWorks, 2020g] and is used widely in this context. MathWorks supports this metric for use directly on Simulink models, and is provided as an architecture metric via the Simulink Check toolbox. For more information on how MathWorks adapts this metric to Simulink models, please see the Simulink Check Reference [The MathWorks, 2020f]. We leveraged Simulink Check to automatically compute cyclomatic complexity values for the designs before and after restructuring, in order to compare them.

6.1.6 Testability

Information hiding encourages the decomposition of systems such that module changes are prevented from propagating throughout several modules, thus allowing the independent testing of modules [Parnas, 1972a]. Some studies in OOP suggest that information hiding has a negative impact on testing [Voas, 1996]. Therefore, we seek to understand and evaluate our approach with respect to testing impact—both in the testing effort required and test coverage.

The SDV toolbox automatically generates test cases for Simulink models in order to maximize the structural coverage metrics of decision, condition, Modified Condition/Decision Coverage (MCDC), and execution coverage. It

also records the test effort in terms of total objectives needed for testing. A harness model was created and the vectors were used to perform MiL testing in order to record these metrics.

6.1.7 Performance Comparison

As our approach relies heavily on the use of Simulink Function blocks, it is important to be cognizant of the potential for added overhead due to the increase in function calls and switching between modules in the new decomposition [Parnas, 1972a]. To determine whether there was a change in efficiency between the original system and our modified system, they were both simulated as Software-in-the-Loop (SiL), and the Average Case Execution Time (ACET) and Worst Case Execution Time (WCET) were measured and compared. SiL simulation means that the model is generated into code, and then the code is executed.

6.2 Aerospace Case Study

This section describes how the concepts of the previous chapters were applied to an example from the aerospace domain. We use a helicopter control system that is available on the MATLAB Central File Exchange[4] as a small open-source system that was originally developed by a MathWorks employee. This system is shown in Figure 6.2. We focus on the Flight Control Computer (FCC) (the controller) as the other components of the system are plant models.

First, the original design structure is described in Section 6.2.1, followed by a description of how the module structure was applied in Section 6.2.2.

[4] https://mathworks.com/matlabcentral/fileexchange/56056-do178_case_study

Section 6.2.3 highlights how the Simulink Module Tool was used throughout the system restructuring process and Section 6.2.4 evaluates the designs before and after the changes according to the criteria set out in Section 6.1. Section 6.2.5 summarizes the findings of the FCC case study.

6.2.1 Flight Control Computer (FCC) Components

The FCC system is shown in Figure 6.2, and is a typical closed-loop control system comprised of a pilot model providing set points, plant models that represent the actuators and the helicopter, sensor models providing feedback, and an FCC model that provides attitude and heading control. The FCC is a small Simulink model comprised of six top-level Model References, as shown in Figure 6.3a. The referenced models, from left to right, implement the Attitude and Heading Reference System (AHRS) Voter, Helicopter Outer Loop Control (HOLC), Helicopter Inner Loop Control (HILC), and the Actuator Loop (AL), which is instantiated thrice. The structure of the FCC is also represented in Figure 6.4a. Note, the AL is a single model, so it is shown only once in the graph.

6.2.1.1 Attitude and Heading Reference System (AHRS) Voter

The AHRS Voter model (Figure 6.3b) takes digital bus input from three sensors and outputs either the middle, average, or individual sensor values depending on the voting criteria. The input bus includes signals relating to: AHRS sensor validity, pitch attitude, roll attitude, pitch rate, roll rate, and yaw rate. The AHRS Voter contains three virtual Subsystems, namely: Mid Value, Avg Value,

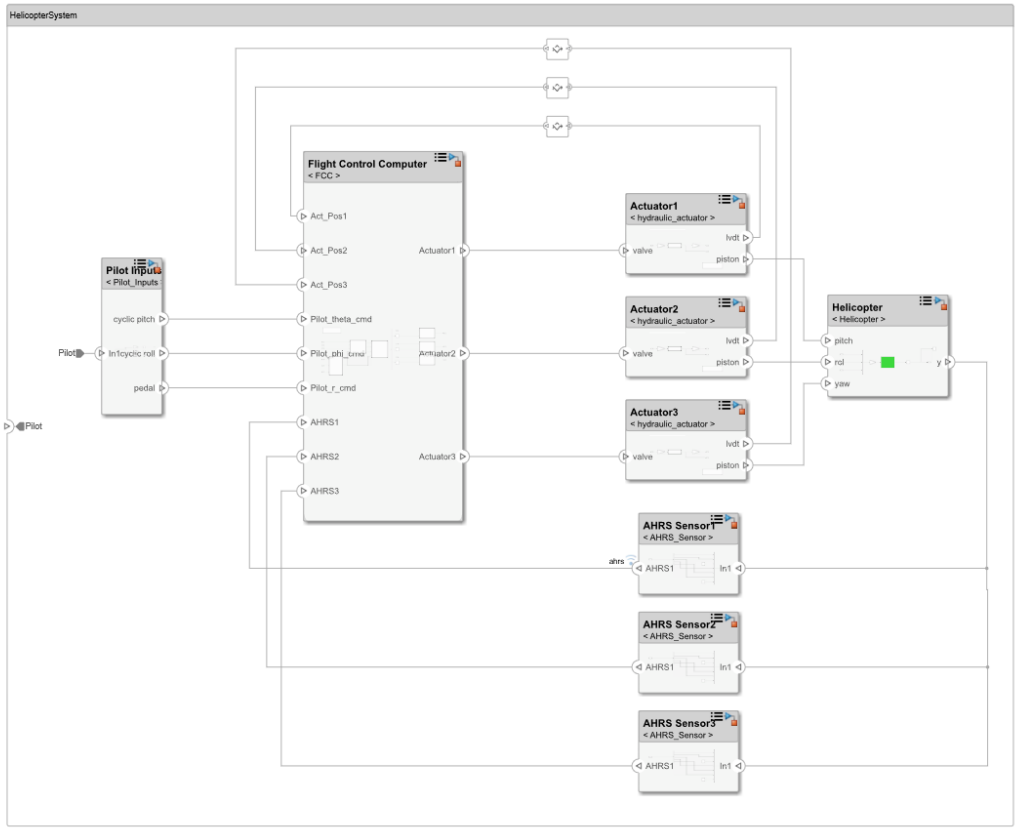
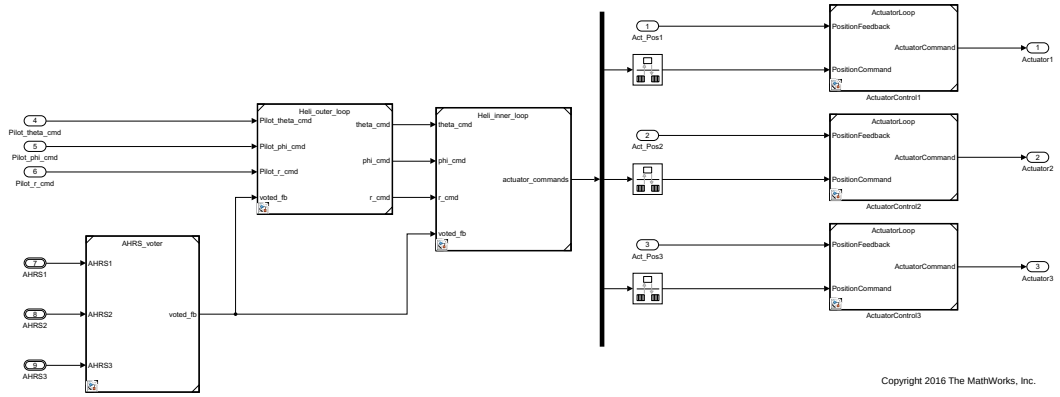
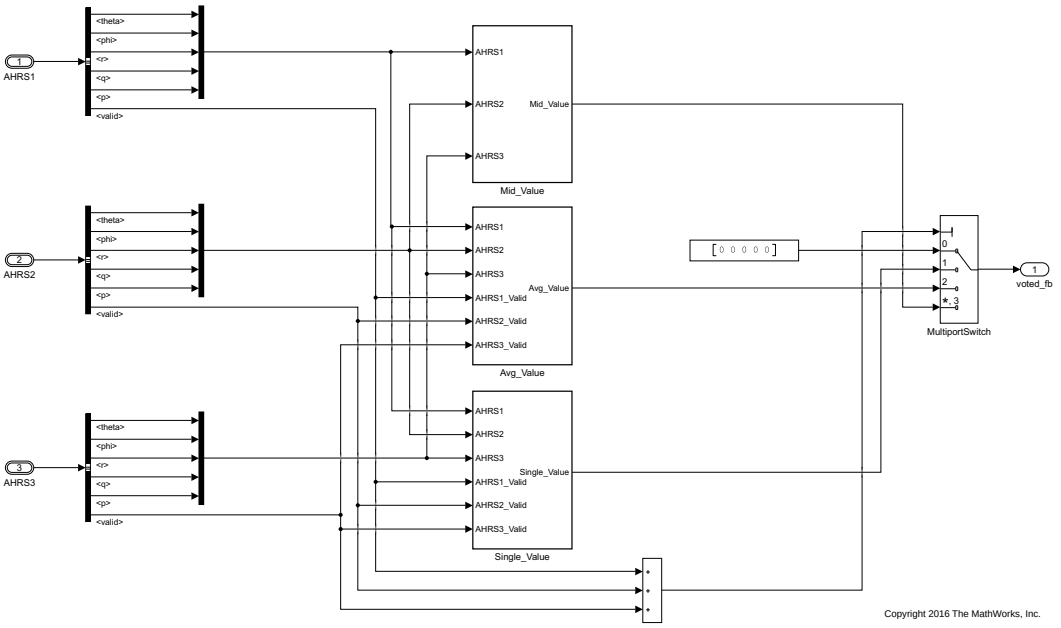


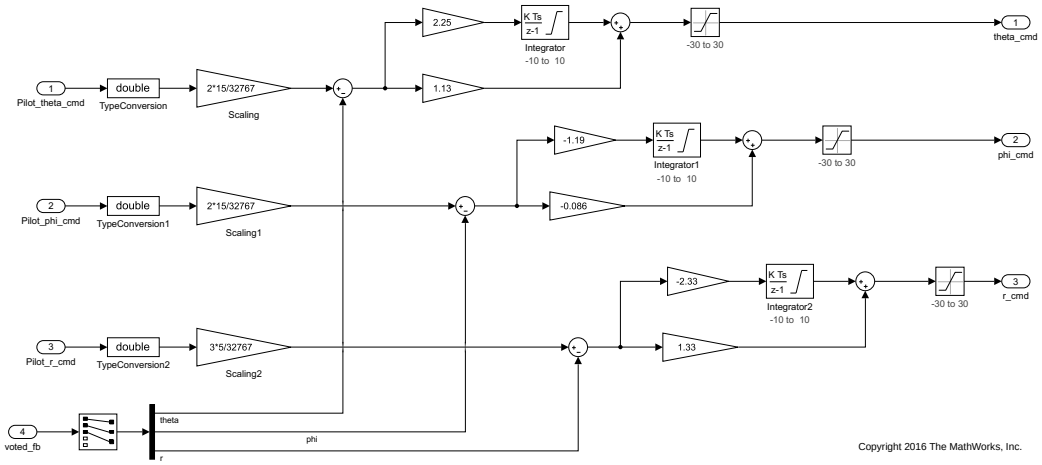
Figure 6.2: Top-level view of the helicopter system example.



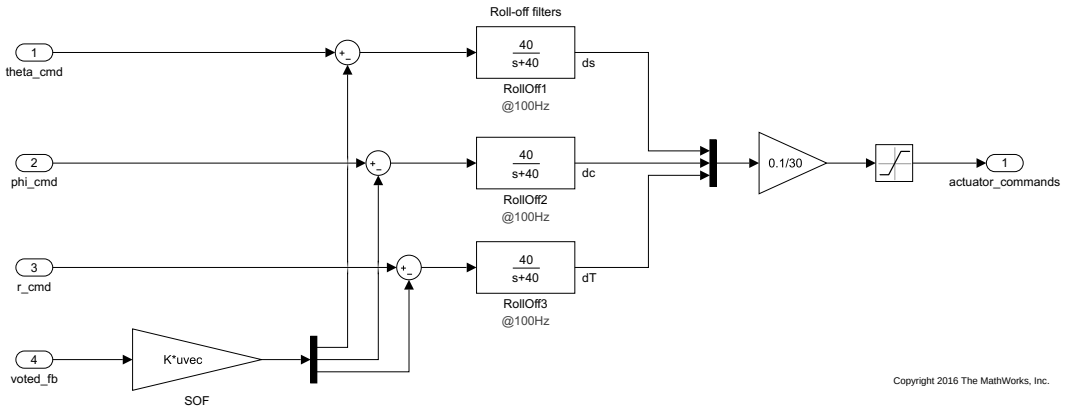
(a) Top-level view of the original FCC decomposition.



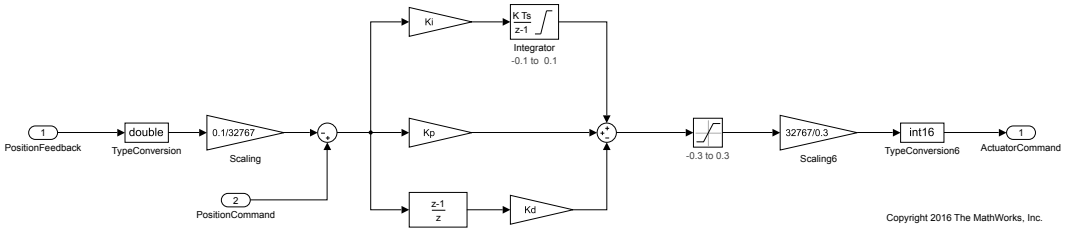
(b) AHRS Voter Subsystem from Figure 6.3a.



(c) Heli Outer Loop Model from Figure 6.3a.



(d) Heli Inner Loop Model from Figure 6.3a.



(e) Actuator Loop Model from Figure 6.3a.

Figure 6.3: Original FCC decomposition.

and Single Value, which contain the corresponding sensor voting algorithms. Per the system requirements, the voting criteria are as follows:

- If three sensors are valid, output the middle value for each bus parameter.
- If two sensors are valid, output the average of the valid bus parameters.
- If one sensor is valid, output the value of the valid bus parameter.

6.2.1.2 Helicopter Outer Loop Control (HOLC)

The HOLC model (Figure 6.3c) receives pilot input commands for pitch, roll, and yaw, as well as AHRS Voter sensor values. The HOLC outputs pitch, roll, and yaw commands for the HILC model. Within the HOLC model, there are three Proportional/Integral (PI) control loops, pertaining to the pitch, roll, and yaw commands. Each PI control loop contains a state-holding Discrete-Time Integrator block to perform accumulation of the input signals, as well as unique controller gains. Per the system requirements, the HOLC model operates on a 10 millisecond (ms) sample time.

6.2.1.3 Helicopter Inner Loop Control (HILC)

The HILC model (Figure 6.3d) receives pitch, roll, and yaw commands from the HOLC model, as well as AHRS Voter sensor values. The HILC outputs actuator position commands to three AL models. Within the HILC model, there are three roll-off blocks for each of the input commands. These blocks are linked from the external Library `Heli Library`. The roll-off blocks each contain a Unit Delay block and act as filters for the input command signals at a bandwidth of 40 rad/sec. The AHRS Voter signal is multiplied with a

gain matrix to compute the closed-loop control feedback signals. Similar to the HOLC model, the HILC model also operates on a 10 ms sample time.

6.2.1.4 Actuator Loop (AL)

The three AL blocks in Figure 6.3a are Model References to one AL model, shown in Figure 6.3e. The AL model receives actuator position command input from the HILC model as well as current actuator position from three Linear Variable Differential Transformer (LVDT) feedback signals. The AL model outputs actuator commands to the three helicopter actuators. Within the AL model, there are three Proportional/Integral/Derivative (PID) control loops, pertaining to the pitch, roll, and yaw actuators. Each PID control loop contains a state-holding Discrete-Time Integrator block to perform accumulation of the input signals, a Unit Delay block to assist in the discrete differentiation, and unique controller gains. In contrast to the HOLC and HILC models, it is required that the AL model operates on a 1 ms sample time, and thus three Rate Transition blocks are used to adjust the sample time of the input signals from the HILC model to the AL models.

6.2.2 Application of the Simulink Module Structure

The FCC was decomposed using the approach described in Section 5, by grouping models based on the secrets they contain (Table 6.1) and the model sample time. These secrets were established based on the accompanying software and system requirements documentation for the FCC project, as well as our understanding of the system. Existing Subsystems were converted to Simulink Functions using the automated process provided by the Simulink

Module Tool. The change in the FCC model structure is represented in Figure 6.4. The AHRS Voter and AL models remained from the original system, however, the new model AHRS Control was added to replace the original HOLC and HILC models, as it was established that these two models were collectively hiding the AHRS cascade-control secret of the system. In each of the three Model References, a Simulink Function was added to encapsulate the top-level block diagrams, and then called via corresponding Function Callers. The new decomposition is shown in Figure 6.5a. Simulink Functions and their corresponding Function Callers are highlighted in the same colour to ease readability. Lastly, SDV was used to formally verify equivalence between units of the original and new designs, as outlined in Section 6.1.1.

The following subsections elaborate on the decomposition of each referenced model. Although the entire system was restructured, we show figures of the changes that occurred in the HOLC and HILC because they are the most complex. The entire system before and after restructuring can be viewed on GitHub.[5]

6.2.2.1 Attitude and Heading Reference System (AHRS) Voter

The AHRS Voter was straightforwardly converted to a Simulink module. We wanted to encapsulate the global sensor voting approach, as well as the three sensor-dependent voting algorithms. As a result, the AHRS Voter itself was encapsulated in a Simulink Function that is exported and callable outside of the module. Each of the three voting algorithms were encapsulated in their

[5] https://groke.cas.mcmaster.ca/gitlab/scotts24/do178

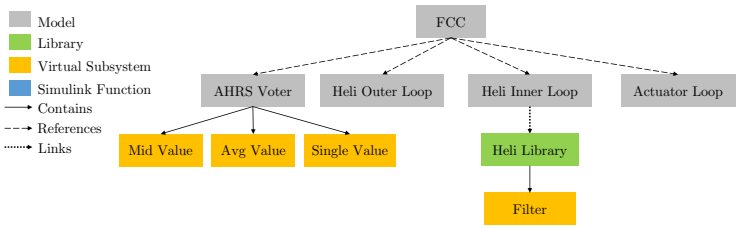
Table 6.1: FCC system module secrets.

Module	Secret	Module Type
AHRS Voter	Sensor voting algorithm	Behaviour-hiding
Mid Value	Algorithm condition	Behaviour-hiding
Avg Value	Algorithm condition	Behaviour-hiding
Single Value	Algorithm condition	Behaviour-hiding
AHRS Control	Cascade controller	Behaviour-hiding
Heli Outer Loop	Pilot control algorithm, scaling, saturation limits	Behaviour-hiding, Hardware-hiding
Pitch Loop	Pilot theta controller	Software Design Decision
Roll Loop	Pilot phi controller	Software Design Decision
Yaw Loop	Pilot r controller	Software Design Decision
Heli Inner Loop	Command control algorithm	Behaviour-hiding
Filter	Noise filter implementation	Software Design Decision
Pitch Feedback	Fore/aft cyclic position command controller	Behaviour-hiding
Roll Feedback	Left/right cyclic position command controller	Behaviour-hiding
Yaw Feedback	Pedal left/right command controller	Behaviour-hiding
Actuator Loop	Actuator control algorithm, scaling, saturation limits	Behaviour-hiding, Hardware-hiding
Actuator1 Loop	Actuator 1 controller	Behaviour-hiding
Actuator2 Loop	Actuator 2 controller	Behaviour-hiding
Actuator3 Loop	Actuator 3 controller	Behaviour-hiding

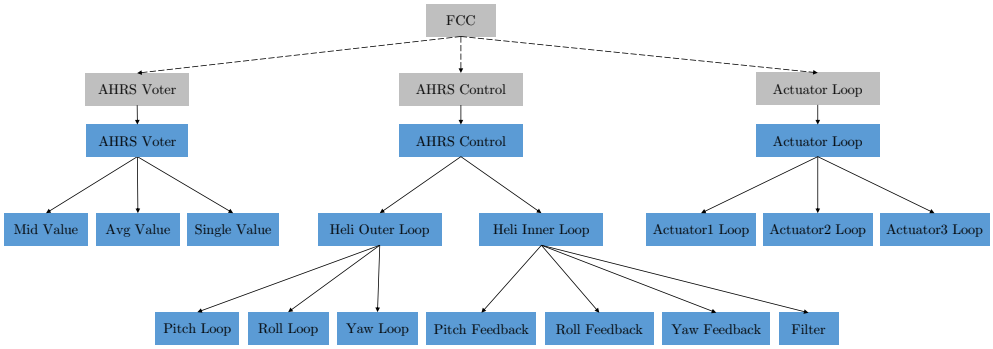
own scoped Simulink Function, where they each previously resided in a virtual Subsystem.

6.2.2.2 Attitude and Heading Reference System (AHRS) Control

Before applying the approach to the AHRS Control model (Figure 6.5b), improvements were made to better encapsulate design secrets deemed likely to change. Within the HOLC model (Figure 6.5f), each of the PI controllers is likely to change, and thus three Simulink Functions were added to encapsulate each of these controllers. Within the HILC model (Figure 6.5d),



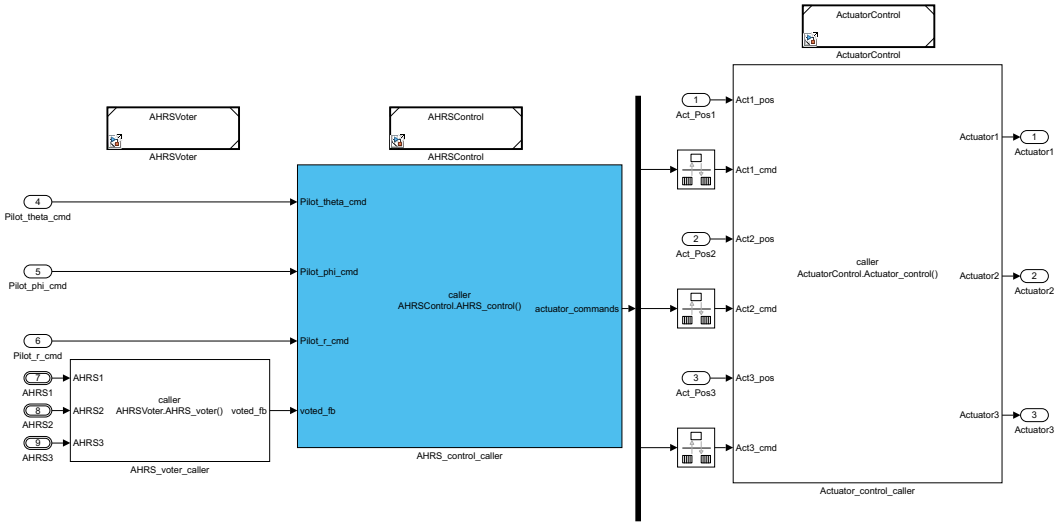
(a) Structure of the original FCC decomposition.



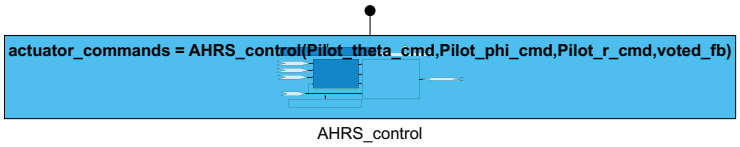
(b) Structure of the new FCC decomposition.

Figure 6.4: Structure of the FCC before and after restructuring.

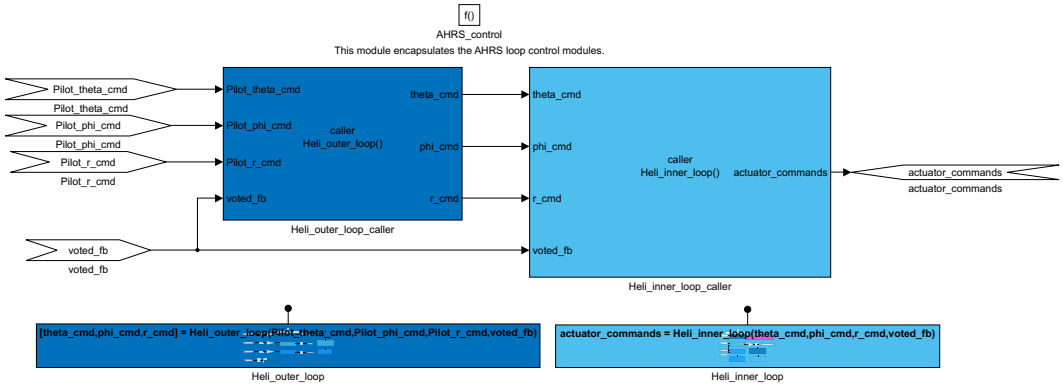
the matrix gain was likely to change, and thus it was split into three separate gains corresponding to the pitch, roll, and yaw signals. Then, three new Simulink Functions were added to encapsulate each of the new gains with the feedback summation block and the derivative roll-off filter as shown in Figure 6.5e. The roll-off filter Library Links were removed and the blocks were prepared for conversion to Simulink Functions by placing the Unit Delay outside of the Subsystem, then routing the output of the Unit Delay as an input to the Subsystem. This change was necessary as the filter block was to be used in three separate instances, and thus it was undesirable for the Subsystem to retain state. After this change, the Subsystem was converted into a Simulink Function.



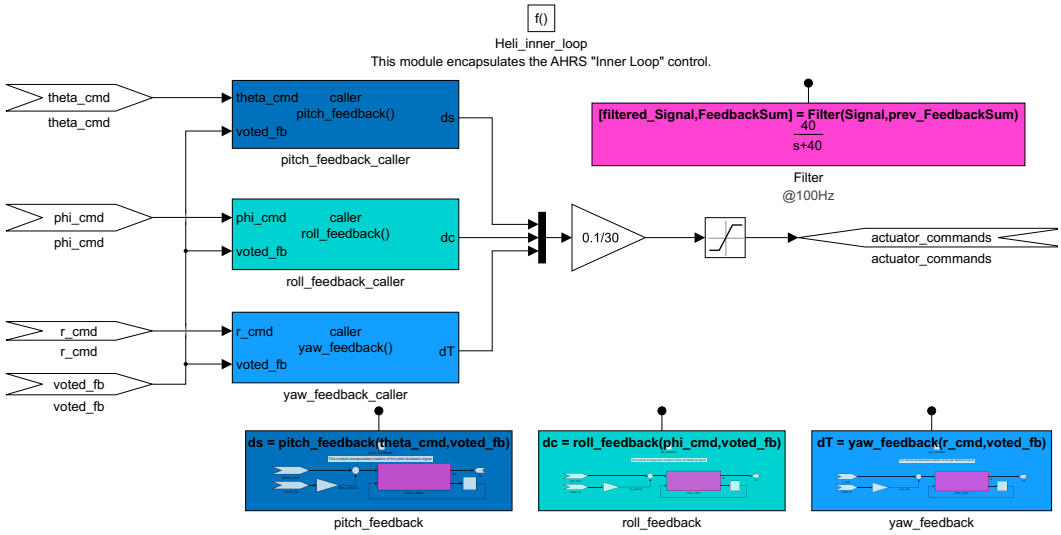
(a) Top-level view of the new FCC decomposition.



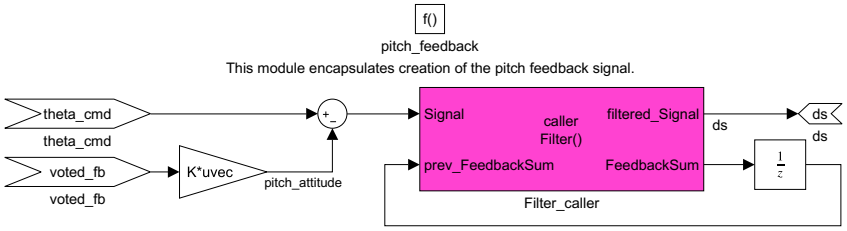
(b) AHRS Control Model Reference from Figure 6.5a.



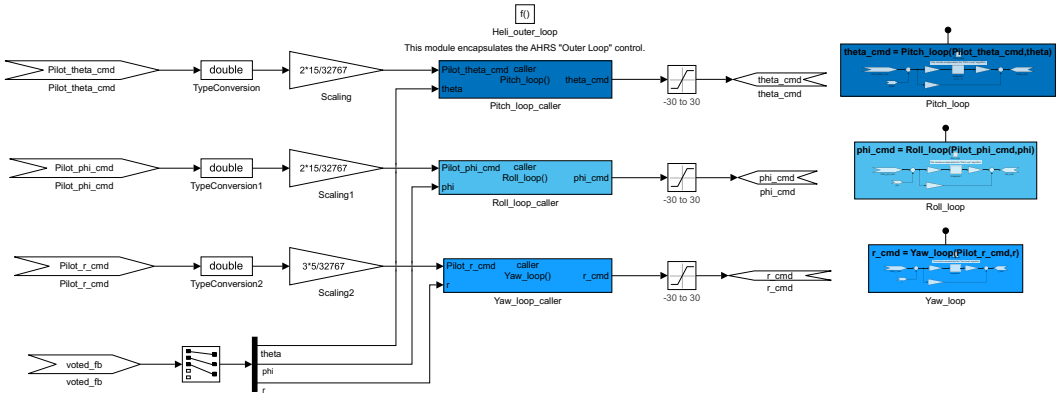
(c) AHRS_control Simulink Function from Figure 6.5b.



(d) HILC Simulink Function from Figure 6.5c.



(e) pitch_feedback Simulink Function from Figure 6.5d.



(f) HOLC Simulink Function from Figure 6.5c.

Figure 6.5: New FCC decomposition.

6.2.2.3 Actuator Loop (AL)

In the AL Simulink model, the fact that the model was instantiated three times for each actuator was internalized in the new AL module. The AL is contained in a Simulink Function in order to encapsulate the control algorithm and several hardware dependent values dealing with saturation limits, scaling, and type conversions. This Simulink Function is exported from the module. Moreover, the three loops were separated into their own Simulink Functions. This allows for the control algorithms and their parameters to vary independently between the three actuators. Changing the parameters of one actuator will no longer impact the other two.

6.2.3 Using the Simulink Module Tool

We leveraged the tool to support the restructuring activities outlined in the previous section. While the Simulink Module Tool is described in detail in Section 5.5, we now describe how it was used in the context of the FCC case study.

- The tool was used to convert 18 Subsystems into Simulink Function blocks, as described earlier. This process encompasses several steps for each Subsystem, such as adding a Trigger to the Subsystem, replacing Inport/Outport blocks with ArgIn/ArgOut blocks, and configuring multiple parameters for each of these blocks. The process was outlined in detail in Algorithm 1. Automating this operation saved a significant amount of time.

- After a Simulink Function was added to the design, it was necessary to ensure that it was scoped at the appropriate visibility. To do this, the tool converted between the different kinds of scope for Simulink Functions when we were restructuring the FCC. Simulink Functions were easily made internal/external to the module, as needed.

- For each Simulink Function block, it was necessary to also add Function Caller blocks to call the Simulink Function. In the entire FCC, we added 20 Function Caller blocks, which the Simulink Module Tool was able to create and configure automatically, also saving time.

- The syntactic interface and dependencies for the FCC were automatically generated as shown in Figure 6.8. This information was used in the interface complexity evaluation in Section 6.2.4.2, and coupling evaluation in Section 6.2.4.3.

- The four guidelines presented in Section 5.4 were automatically checked to ensure that the newly added Simulink Functions were properly scoped, that they were placed at the appropriate level in the module, and that we did not introduce Simulink Function shadowing.

In summary, the Simulink Module Tool saved time when restructuring the new FCC, and made many of the activities less tedious to perform.

6.2.4 Evaluation

Through the use of our proposed Simulink module structure, we aim to achieve designs that are robust with respect to change. In this section, we quantify the improvement to the design of the FCC in terms of information hiding and

modularity by evaluating characteristics that are considered indicative of good design structure, as described in Section 6.1.

6.2.4.1 Information Hiding

We evaluated the designs from the information hiding perspective by observing how likely changes would propagate through the systems. One of the likely changes that could impact the design is the use of a cascade controller implementation. This was identified as a secret in Table 6.1. In order to determine whether the new design more effectively hides this design decision than the original design, we apply a change to the cascade controller and observe how it impacts the system. Figure 6.3a shows the original implementation of the FCC, where the HOLC and HILC models implement the cascade controller. If the design is changed to a single loop controller, the HILC would be deleted, and the model reference would be removed from the FCC model, thus impacting two models. This is shown in Figure 6.6a. Furthermore, the HOLC would also require changes in parameters in order to maintain the required performance, meaning a third model would require modification. Moreover, its interface would need to be modified in order to output the appropriate commands. Since the custom roll-off filter block and the Library are only used in the HILC model, it would be possible to delete them entirely, as they are no longer used by the system. Deleting the Library model means that a total of four models were deleted to implement the new controller strategy.

To implement this same change in the newly restructured FCC system, no changes would be required in the FCC model, and the model would remain as shown in Figure 6.5a. The change to the controller strategy is contained within

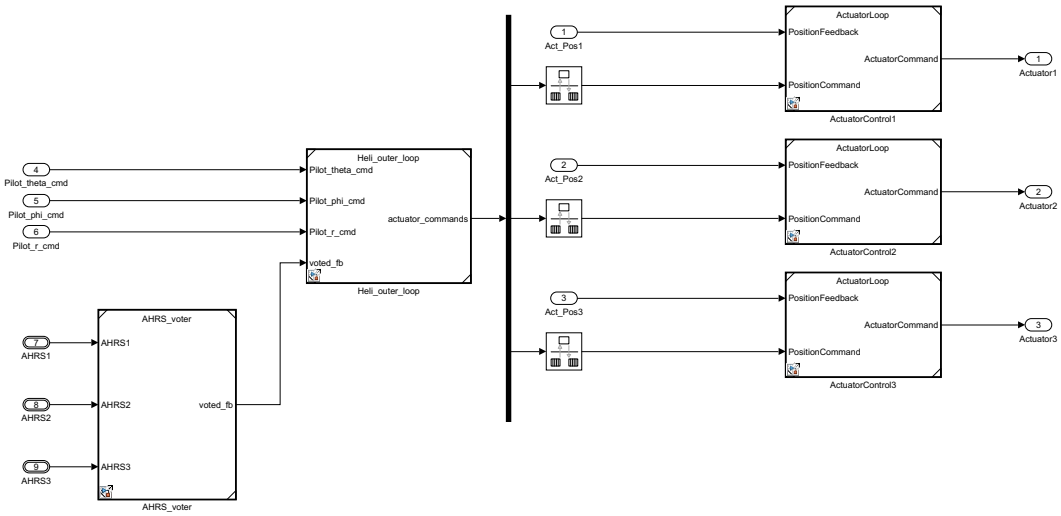
the AHRS Control model, and no interface changes are necessary. The changed model is shown in Figure 6.6b, as compared to the original in Figure 6.5c.

We also evaluated how the new structure actively restricts access to the "hidden" design details. The original design used Model References, which were further decomposed into Subsystems, and neither of these constructs limits the use of its internals [Jaskolka et al., 2020a]. That is, a model can be referenced from any other model, without restriction, and all of its functionality will be available. In leveraging Simulink Functions that are locally scoped, we ensure that upon the model being referenced, they will not be callable outside of the model. This allows one to hide functionality that should remain hidden inside the module.

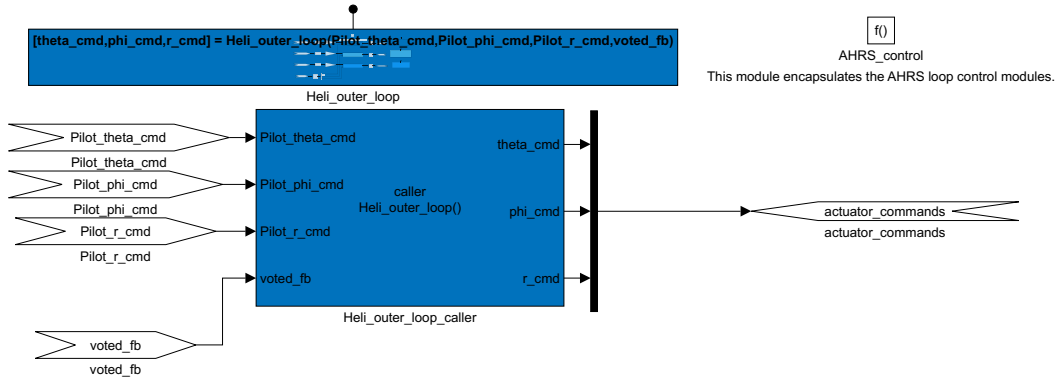
To summarize, changing the cascade controller to a single loop controller would require several model changes in the original decomposition, while the new design requires the modification of only one model. Therefore, it is clear that the new decomposition effectively hides the secret of the cascade controller implementation and facilitates future changes.

6.2.4.2 Interface Complexity

In the original design, the HILC had three links to a roll-off filter block in an external Library named `Heli Library`. A Library was clearly being used in order to make one block reusable, however, this can alternatively be achieved via Simulink Functions. Instead of using a separate Library, the roll-off filter library block was converted to a Simulink Function and moved directly into the HILC, as shown in Figure 6.5d. Then, the three linked blocks were replaced with three Function Callers to the same Simulink Function. As a result, the Library is no longer required. The interface of the FCC was improved by



(a) Changes required in the FCC model to implement single loop control, compared to Figure 6.3a.



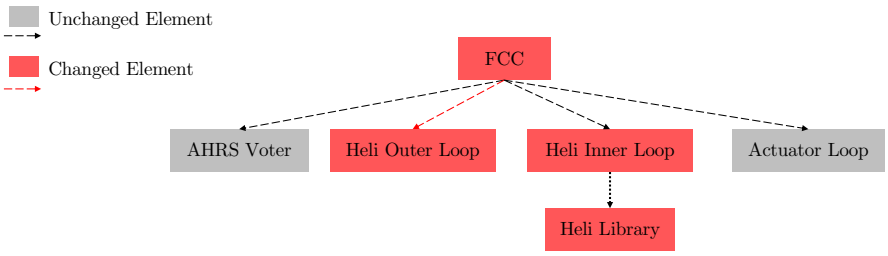
(b) Changes required in the AHRS Control model to implement single loop control, compared to Figure 6.5c.

Figure 6.6: Implementation of a change in the FCC.

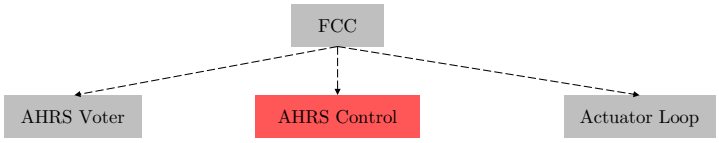
removing links to this library entirely, as shown in Figure 6.8. Additionally, because we hid knowledge of the number of actuator loops required, the number of model references was reduced from 6 to 4.

6.2.4.3 Coupling and Cohesion

In the original model, the HILC and HOLC models were two independent models, however, they were tightly coupled because together they

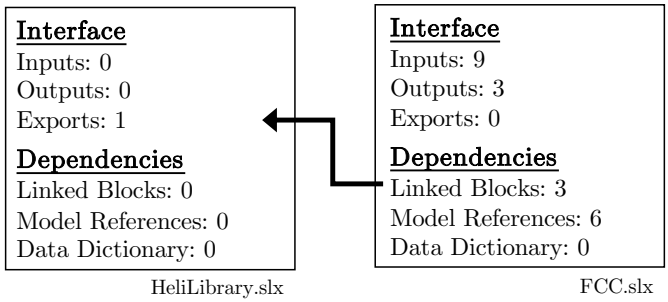


(a) Representation of change in the original FCC modules.

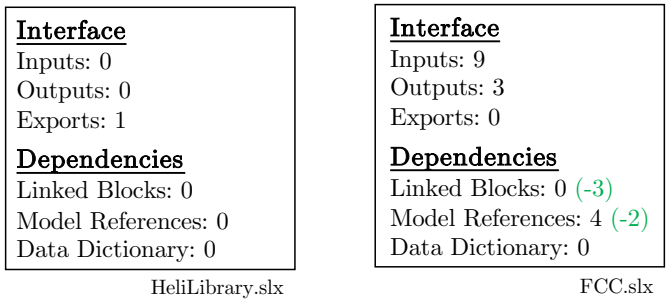


(b) Representation of change in the new FCC modules.

Figure 6.7: Structure of the FCC before and after applying a change to the controller strategy.



(a) Before restructuring, the FCC relied on an external Library.



(b) After restructuring, the FCC does not require the Library.

Figure 6.8: Interface changes between FCC and the Library.

implemented a cascade controller. Combining their functionality into the new AHRS Control model resulted in a more cohesive and less coupled design.

This also eliminated one model reference. Moreover, in the original FCC, the AL model was referenced three times, but in the new FCC, the three actuator control loops were entirely contained by the AL module, which further decreased coupling and increased cohesion. Furthermore, the coupling between the FCC and the external Library was eliminated which also reduced coupling.

6.2.4.4 Cyclomatic Complexity

Table 6.2 shows the change to cylomatic complexity. The original FCC had a cyclomatic complexity of 41, whereas the restructured FCC has a cyclomatic complexity of 61, resulting in an increase in complexity by 20. The change in cyclomatic complexity is a result of our module structure leveraging Simulink Function blocks to thoroughly decompose the system. MathWorks has adapted the cyclomatic complexity metric to the Simulink language such that each atomic Subsystem adds a value of 1 to the complexity [The MathWorks, 2020f], because of the added decision point for calling/not calling the function. As a result, for 15 of the 18 Simulink Functions, a value of 1 was added to the cyclomatic complexity. The remaining three Simulink Functions did not add complexity as they were directly converted from blocks that were already atomic units, which already contributed a complexity of 1. Other changes in complexity resulted from loss of reuse of the AL model, as the new decomposition assumes that the actuator controllers are likely to vary independently from one another (+8), and the removal of three model references (–3).

Table 6.2: FCC complexity, testing, and SiL performance comparison.

Metric	Before	After	Difference	Percent Difference
Cyclomatic Complexity	41	61	+20	+48.8%
Test Objectives	121	139	+18	+14.9%
Satisfied	113 (93.4%)	131 (94.2%)	+18	+0.8%
Unsatisfiable	8 (6.6%)	8 (5.8%)	0	0%
Decision	101/105 (96.2%)	131/139 (94.2%)	+30/+34	–2%
Execution	89/89 (100%)	185/185 (100%)	+96/+96	0%
ACET (ns)	697	878	+181	+25.9%
WCET (ns)	5,360	4,257	-1,103	-20.6%

6.2.4.5 Testability

The SDV toolbox was used to automatically generate test vectors that maximize structural coverage metrics of decision, condition, MCDC,[6] and execution coverage. A harness model was created and the vectors were used to perform MiL testing, with the results given in Table 6.2. Naturally, with increased complexity resulting from new Simulink Function blocks, there are corresponding coverage objectives and test cases that are also added. We can see that 18 additional test objectives were added for each of the new Simulink Function blocks, and each of these were satisfied.

The number of decision objectives satisfied and total decision objectives increased by 30 and 34, respectively. This was due to each of the 18 new Simulink Functions adding 1 satisfied decision objective. Similar to the cyclomatic complexity, the remaining 12 satisfied decision objectives and 16 total decision objectives resulted from the loss of reuse of the AL model. The AL model originally had a decision coverage of 6/8, and was a single model that was referenced three times, so the contained logic only contributed once to the coverage objectives. However, in the restructured system, the logic

[6]Note, the FCC design did not contain blocks that require condition nor MCDC coverage.

contained in the AL is now defined three times as separate Simulink Functions, so that they can be modified independently. This results in decisions being added two more times, and thus the objectives associated with the logic also increases twice.

Overall, the restructuring did not have a substantial impact on the testing results, with the most significant change being the added objectives as a result of Simulink Functions.

6.2.4.6 Performance Comparison

C code was generated for the FCC using the Embedded Coder toolbox. Then, SiL testing was performed with the same test vectors from MiL testing, and code execution profiling was performed to understand the performance of the system before and after the decomposition changes. The results are reported in Table 6.2. Overall, we expected the use of Simulink Functions to increase the execution time because of the added function call overhead [Parnas, 1972a; Jaskolka et al., 2020b] and to reflect the increase in cyclomatic complexity. This was indeed the case as an increase of approximately 26% was observed in the ACET of the system's generated code. However, the WCET reduced by approximately 20%.

6.2.5 Case Study Summary

In summary, the application of the approach on the FCC was beneficial in reducing interface complexity, decreasing coupling, and increasing cohesion. Most importantly, the new decomposition better supported information hiding, and changes were easier to make on the restructured system.

Nevertheless, the design's additional Simulink Functions each added a cyclomatic complexity value and decision test objective, and because the FCC is a relatively small system, the amount of Simulink Function reuse did not offset the added objectives (contrary to the second case study in Section 6.3). As a result, the total cyclomatic complexity, and number of decision objectives both increased in the new FCC, and the system exhibited a decrease in ACET performance by approximately 26%.

6.3 Nuclear Case Study

This section describes how our concepts were applied to restructure a Simulink implementation of a nuclear Shut Down System (SDS). The SDS senses whether conditions in a nuclear reactor are no longer safe, and controls the lowering of control rods to stop ("shut down") the reaction. With 605 subsystems, 74 top-level inputs, 7 top-level outputs, and 6036 total blocks, the model is too large to be presented here. However, it represents the size of small to medium size designs found in practice. Both the documentation and implementation details of the system are proprietary. It was developed by researchers at McMaster University from proprietary requirements provided by an industry partner. Although both the size and the proprietary nature of the system prevent us from including a full comparison of design details, we can report the details of restructuring one module, as well as the results of the entire system, using the approach outlined in Chapter 5.

Section 6.3.1 begins by describing how the proposed module structure was applied in the SDS on one module: the Power Estimation (PE) module. The approach is first described in detail on this module, and an overview of the

restructuring of the entire SDS follows. An explanation of how the Simulink Module Tool supported this process is provided in Section 6.3.2. Section 6.3.3 goes on to evaluate the approach's effectiveness in terms of achieving a more modular system. Finally, Section 6.3.4 summarizes the nuclear case study.

6.3.1 Application of the Simulink Module Structure

This section describes how the module structure was applied on the PE module and the entire SDS.

6.3.1.1 Power Estimation (PE) Module

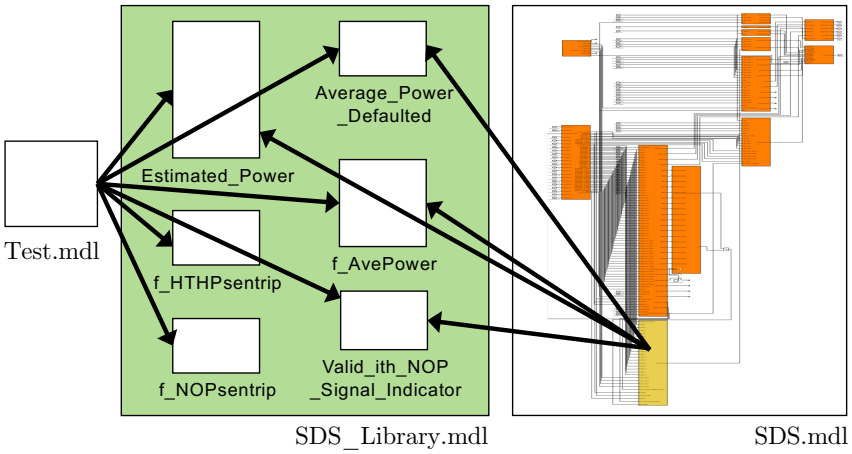
The first SDS implementation was implemented in Simulink R2012a by other developers some years before Simulink Functions were introduced in the Simulink language. The design makes heavy use of linked blocks, which link to various blocks in the SDS Library. The structure of the system is shown in Figure 6.9a. The PE subsystem estimates the power of the reactor based on the average neutron over-power sensor values. It is one of the more complex components in the SDS. The PE implementation consists of several subsystems, which are defined in the SDS Library. PE contains secrets related to both hardware (e.g., which sensors are used) and software (e.g., how to accommodate for insufficient sensor readings). Unfortunately, a Library does not enforce information hiding, as shown in Chapter 4. This is confirmed by the creation of a test model (`Test.mdl` in Figure 6.9) to probe the Library. Any of the blocks in the SDS Library can be used without restriction, and the internals of any subsystem are free to clients to use as well, even if this is not

the developer's intent. It is not possible to selectively expose or hide functionality, as it is with our approach.

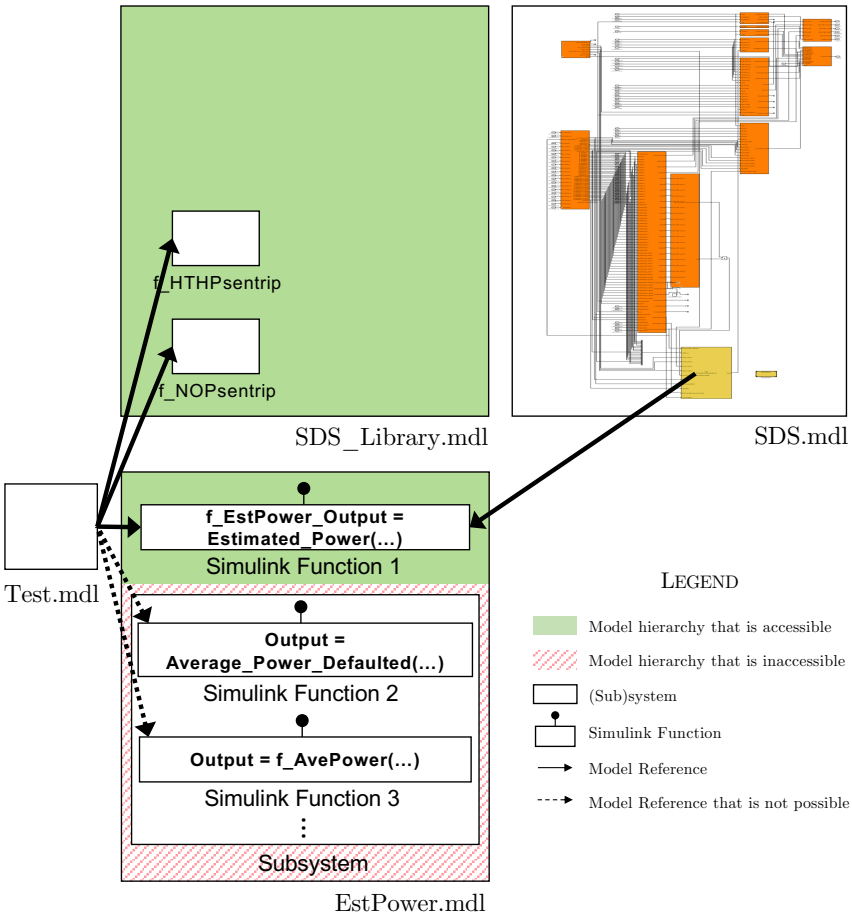
By restructuring the PE subsystems into a Simulink module, we aim to hide implementation details from users of the PE module—*the users should only be able to access the estimated power output value.* This is an essential difference between defining modules as we recommend compared to using "coding" guidelines that are not enforced by the language (such as those proposed in Section 4.4). By using Simulink Function blocks, the Simulink module structure actively enforces the hiding of design details. Figure 6.9b shows the resulting module structure. A new model file (`EstPower.mdl`) was created and all related functionality was structured as a module as described in Section 5. This entailed organizing the operations as Simulink Function blocks, choosing which are to be external and which are hidden in the module, and scoping them based on our guidelines (Section 5.4). While there are many possible decompositions, the only exported function that is available for other modules to use is *Estimated_Power*. By placing it at the root level and setting the *Function Visibility* parameter to *scoped*, it can be called like a member function (i.e., *EstPower.Estimated_Power (...)*). The SDS model imports this function definition using a Model Reference to the module and calls the function using a Function Caller block wherever the function is to be executed. Functionality in the SDS Library unrelated to PE, such as *f_HTHPsentrip* and *f_NOPsentrip*, remained as is.

6.3.1.2 Entire Shut Down System (SDS)

Before applying the approach to the rest of the SDS, improvements were made to the original design to better encapsulate hardware secrets. These



(a) Before: PE implemented in a Library.



(b) After: PE implemented with a module structure.

Figure 6.9: Structure of the SDS system, focusing on power estimation.

improvements included adding an additional Subsystem at the top-level to encapsulate behaviour related to push-button hardware, as well as relocating Subsystems related to communications behaviour into one dedicated communications Subsystem.

Then, the SDS was restructured by converting all the top-level Subsystems to Model References. Within each respective Model Reference, a Simulink Function was added to encapsulate the root block diagram of the Model. Then, in each model, all nested Subsystems were converted to Simulink Functions with the exception of two, which remained as linked Library Subsystems because they provided low-level utility functionality throughout the SDS. Finally, the top-level Simulink Function of each Model Reference was called from the root of the SDS Model.

Throughout the original SDS, many of the linked Library Subsystems were used multiple times, which resulted in duplicate blocks of code in the generated code. These Subsystems presented an excellent opportunity to achieve Simulink Function reusability; however, many of the Subsystems contained state-retaining Unit Delay blocks. Before replacing the Subsystems with multiple Function Callers to the same Simulink Function, the state-retaining Unit Delay blocks had to be moved outside of the Subsystems. Section 6.4.3 further explores the reasoning behind this design change. Subsequently, where multiple Function Callers to the same Simulink Function were used, an additional Simulink Function was added to encapsulate the multiple callers. Lastly, SDV was used to formally prove that the designs before and after restructuring were behaviourally equivalent, as outlined in Section 6.1.1.

6.3.2 Using the Simulink Module Tool

The Simulink Module Tool facilitates the creation of Simulink modules, as we have done in the SDS example. The tool was used to make the changes described in Section 6.3, generate syntactic interfaces, and check guidelines, as described in this section.

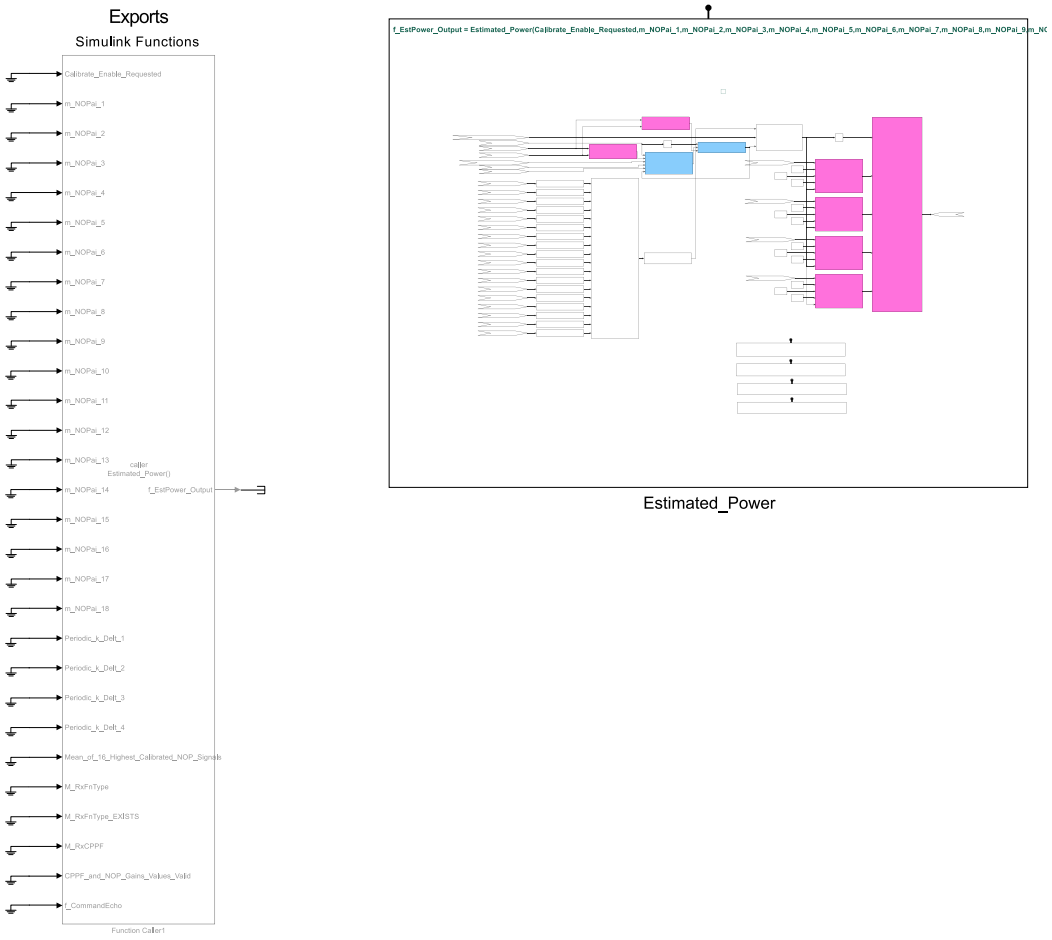
Subsystem to Simulink Function Conversion The tool was a significant help because of its automated Subsystem to Simulink Function conversion functionality. This was particularly useful for the SDS because we introduced 142 Simulink Function blocks. Manual conversion would be time consuming without the automated assistance of the Simulink Module Tool. Moreover, many of the Subsystems had a large number of Inport blocks, with some containing over 50 Inports. Without the tool, we would need to manually replace each Inport with an ArgIn block, and then configure its parameters.

Scope Changes The tool converted between the different kinds of scoping for Simulink Functions when we were decomposing the SDS functionality into Simulink modules. The majority of Simulink Functions were easily made internal to their module, while some externally visible Simulink Functions were appropriately scoped so that they were exported Simulink Functions and available to the SDS model. The tool allows one to quickly change the scope without the need to remember the scoping rules described in Section 2.2.1.2.

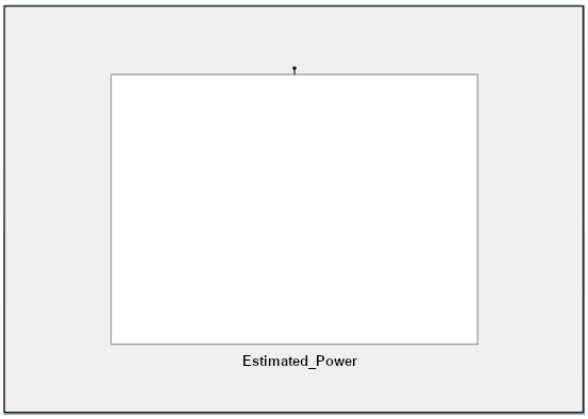
Function Calling With the introduction of many Simulink Function blocks comes the need to also add Function Caller blocks. In the entire SDS, we added

304 Function Caller blocks, which the Simulink Module Tool was able to create and configure automatically. For example, the tool assisted in calling the *f_EstPower_Output* Simulink Function, with its appropriate qualifier, from the SDS model. A Function Caller was automatically created with its *Prototype*, *Input argument specifications*, and *Output argument specifications* parameters pre-populated, saving time.

Interface Generation The syntactic interface for the SDS and PE module was automatically generated. The interface information was used in the evaluation in Section 6.3.3.2 when examining changes to interface complexity. Figure 6.10 shows the generated interface representations from both the Simulink Module Tool and the MathWorks Interface Display for the PE module. The representation of the interface should show that a single Simulink Function is exported from PE, as also depicted in Figure 6.9b and Figure 6.11b. In the interface generated by the Simulink Module Tool shown in Figure 6.10a, we can see the *Estimated_Power* function under the "Exports" heading, to the left of the implementation. Unfortunately, the exported function is not shown in the MathWorks Interface Display view shown in Figure 6.10b, because it only shows the Inport and Outports, of which there are none in this module. Our definition of an interface promotes a better, more complete, view of elements present on the interface, as demonstrated by the visible exported Simulink Function present in Figure 6.10a, but missing from Figure 6.10b. Although this is a simple example, it indicates that our definition of a Simulink module interface ultimately leads to a better understanding of the data flow into and out of the model.



(a) The Simulink Module Tool interface showing one exported Simulink Function.



(b) The MathWorks Interface Display does not show any exported Simulink Functions.

Figure 6.10: PE module interface representations.

Dependency Viewing Module dependencies were detected and listed by the tool. This information was also used in the evaluation when analyzing interface complexity and coupling, as shown in Figure 6.11.

Guideline Checking The four guidelines presented in Section 5.4 were automatically checked. The entire new design adheres to the guidelines, but we discuss the PE module as an example. First, all functions were placed as low as possible in their module such that any corresponding Function Caller blocks in the SDS module can still access them (Guideline 1). This minimizes the scope of the Simulink Functions, and restricts their accessibility in the module to only where it is required. In the PE module, only the *Estimated_Power* function is exported in order to make this functionality available on the PE interface, so that the SDS model can import it. All Simulink Function *Function Visibility* parameters are set to *scoped*, as global visibility is not required (Guideline 2). These guidelines helped to enforce information hiding in the system, and ensured the interfaces were as minimal as possible. Furthermore, each Simulink Function has a unique name, thus avoiding shadowing (Guideline 3). This made the new decomposition of the PE module easy to understand. Lastly, the SDS was already prepared as a production model. As a result, it did not contain any constructs that used the base workspace (Guideline 4). No further action was needed to support this guideline.

In summary, the Simulink Module Tool saved much time in restructuring the new SDS, calling external module functionality, and analyzing interfaces, dependencies, and guideline compliance.

6.3.3 Evaluation

In order to perform an evaluation of the proposed Simulink module structure, we sought to objectively quantify the improvement to modularity and information hiding. This was done by evaluating characteristics that are widely considered effective indicators of design structure and modularity of large systems, as described in Section 6.1. Throughout this section we discuss the PE module as an example, because the details of the entire SDS are proprietary. We provide results on the entire SDS where possible.

6.3.3.1 Information Hiding

Although the secrets of the SDS are proprietary and cannot be published, we examine a single likely change of the system, in the PE module. The PE implementation contains secrets related to both hardware (e.g., which sensors are used), and software (e.g., how to accommodate for insufficient sensor readings). In the original design, knowledge of these secrets was easily leaked to the rest of the system, because the PE implementation internals were accessible from the SDS library without restriction. This was demonstrated through the use of a test model to probe the SDS library (Figure 6.9). Ultimately, the test model is able to access any of the elements in the library. For example, the *f_AvePower* function can be used by the test model without restriction, even though this should be a hidden function as it will expose the secret of how the average is computed. Moreover, users can also directly create links to any blocks that are *internal* to these top-level library blocks, further leaking internal design decisions. Restructuring the PE-related subsystems into a Simulink module allowed us to hide implementation details

from users of the PE module. In the newly modified design, information hiding is enforced so that the user can access *only* the estimated power output value. With reference to Figure 6.9a and Figure 6.9b, it is clear that the new module effectively hides the secrets that were easily previously accessible via the *Average_Power_Defaulted*, *f_AvePower*, and *Valid_ith_NOP_Signal_Indicator* Library blocks. In the restructured system, attempting to access any of these functions from the test model yields an error. This demonstrates that information hiding has improved.

We also examined how robust the new design was with respect to change. For example, we observed the impact of applying a hardware change to the system by switching from mechanical push buttons to a touch-screen interface. To implement this change in the original design, 9 library-linked subsystems were removed and the internals of 14 library-linked subsystems were modified, 4 of which were top-level subsystems in the system. In the new design, the changes were confined to the push-button hardware-hiding module. Thus, the new system effectively hides the secret pertaining to the use of push-button hardware, demonstrating its robustness with respect to change.

6.3.3.2 Interface Complexity

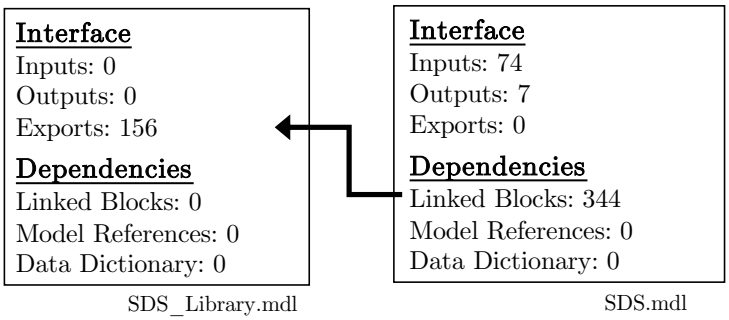
The interactions for the nuclear example are shown in Figure 6.11. Simulink models are represented with their interfaces and dependencies listed, while arrows represent the interactions. In Figure 6.11a, the SDS model was heavily coupled with the SDS Library, with 344 links in its implementation to the 156 blocks exported from the Library. We restructured the PE functionality into its own module, and five of the SDS Library blocks were moved into this new module. The same treatment was applied to the

remaining Library blocks, however, the structure of the rest of the created modules is not shown for confidentiality reasons. Only two Subsystems remained in the Library, because they were low-level utility functions, and storing them in a Library was appropriate in their case. We can see the SDS Library in Figure 6.11a is reduced from 156 blocks to 2 blocks in Figure 6.11b.

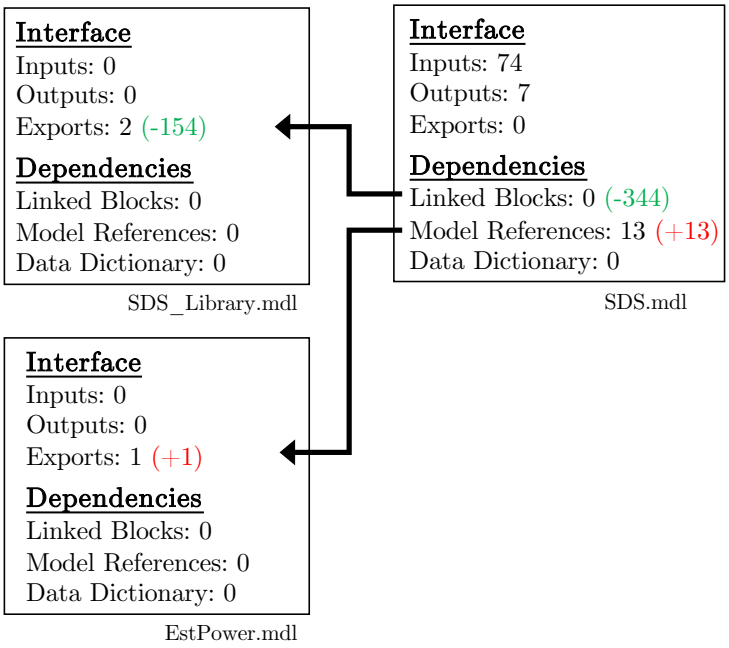
Because of the PE module's support for information hiding, only one of its blocks is actually exported by the module, and the remaining four are hidden and not available on the interface, causing a reduction of 4 fewer blocks on the interface. In total, these hidden blocks reduced the dependency of the SDS model on the Library by 31 links. However, a new Model Reference to include the interface of `EstPower.mdl` was introduced, as reflected in the Model Reference count in the SDS in Figure 6.11b. In fact, since 13 total Simulink modules were created when we refactored the entire system (including PE), the top-level SDS model has 13 new model references. This is a vast improvement compared to the original system that had 344 Library links. By restructuring the system into cohesive and minimally coupled Simulink modules, we have eliminated nearly all of the interconnections between the SDS and its monolithic Library.

6.3.3.3 Coupling and Cohesion

The interconnections between the SDS Library and the SDS model as they were originally structured is shown in Figure 6.9a via the arrows that exist between the two. Although only a few of the connections are drawn in the figure, it is evident that the PE implementation in `SDS.mdl` was highly coupled with `SDS_Library.mdl` when compared with the restructured design in Figure 6.9b. The second design reduced the interconnections to only one interconnection between `SDS.mdl` and `EstPower.mdl`. This reduction in



(a) Before restructuring, the SDS Library exposed all of its functionality, and the SDS heavily depended on it.



(b) After PE restructuring, the SDS Library functionality related to PE was hidden in EstPower, and only one function was exposed to the SDS.

Figure 6.11: Interactions of the SDS system.

Table 6.3: SDS complexity, testing, and SiL performance comparison.

Metric	Before	After	Difference	Percent Difference
Cyclomatic Complexity	2,413	1,363	-1,050	-43.5%
Test Objectives	7,099	3,874	-3,225	-45.4%
Satisfied	2,996 (42.2%)	1,589 (41%)	-1,407	-1.2%
Unsatisfiable	2,044 (28.8%)	1,179 (30.5%)	-865	+1.7%
Undecided	2,059 (29%)	1,104 (28.5%)	-955	-0.5%
Decision	1,249/2,051 (61%)	751/1,168 (64%)	-498/-883	+3%
Condition	946/2,606 (36%)	530/1,414 (37%)	-416/-1,192	+1%
MCDC	188/1,242 (15%)	89/640 (14%)	-99/-602	-1%
Execution	1,351/1,351 (100%)	1,424/1,428 (99.7%)	+73/+77	-0.3%
ACET (ns)	15,005	8,357	-6,649	-44.3%
WCET (ns)	90,397	88,224	-2,173	-2.4%

coupling was additionally observed when comparing the syntactic interfaces of the models, as described in the previous section and shown in Figure 6.11.

In the original system, the SDS library contained all functionality related to the SDS. In the restructured system, related functionality was restructured into cohesive Simulink modules that aimed to hide secrets. In our example, functionality related to PE was separated into a Simulink module of its own. When examining the entire SDS, coupling to the SDS Library was reduced because of only two utility blocks. We created 13 new modules to hide the various secrets listed in the design documentation. Related functionality was grouped inside the new modules, which created a more cohesive design, however, at the cost of necessary coupling links to the new Simulink modules. Overall, our approach positively impacted the design by reducing coupling and making the system more cohesive.

6.3.3.4 Cyclomatic Complexity

Table 6.3 shows the change of cyclomatic complexity for the entire system before and after the restructuring. Typically, the addition of one Simulink

Function increases the cyclomatic complexity by a value 1. This is because MathWorks has adapted the cyclomatic complexity metric such that each atomic Subsystem adds a value of 1 to the complexity [The MathWorks, 2020f]. However, due to the reusability of the Simulink Functions, if a developer wishes to reuse (i.e., call) a Simulink Function several times in a model, it will not add to the complexity. This is different for a block that is stored in a Library because every reuse (i.e., link) creates a separate instance of the Library block. Therefore, each instance of a Library contributes to the complexity of the system. As a result, converting a virtual Subsystem stored in a Library to a Simulink Function block will add 1 complexity value, but if is it reused multiple times, the complexity savings will be greater. The substantial decrease in cyclomatic complexity in the SDS by approximately 43% can be attributed to the removal of 154 Library Subsystems, which were replaced by 20 Simulink Functions that were reused multiple times. There is a high degree of reusability with an average of 7.7 Function Callers to each Simulink Function. The change from a virtual Subsystem stored in a Library to a Simulink Function will result in a decrease in cyclomatic complexity if that Simulink Function is called more than once, to offset the initial increase of 1 to the cyclomatic complexity.

In summary, although the use of Simulink Function blocks may come at a small initial cost to cyclomatic complexity, in this larger case study, the amount of reuse in the system means that the cyclomatic complexity reduces substantially due to the more reusable Simulink Function construct.

6.3.3.5 Testability

The SDV toolbox was used to compute the decision, condition, MCDC, and execution structural coverage metrics. SDV was run on each system for approximately 20 hours each, and the results are reported in Table 6.3. There were approximately 43% fewer total test objectives in the restructured system, which is a significant reduction in the test effort required by the testing tool. Overall, the number of satisfied, unsatisfied, and undecided objectives remained relatively stable.

Table 6.3 reports relatively small changes to structural coverage in the totals for the entire system, with condition and decision coverage slightly increasing, and MCDC and execution coverage slightly decreasing. We found that the addition of the Simulink Function blocks increases the number of decision objectives, and adds satisfied decision coverage. This was to be expected as MathWorks' definition of decision coverage encompasses function call execution. The reduction observed in the other structural metics occurred for the same reasons outlined in Section 6.3.3.4. The coverage decreases when a Library block is converted to a Simulink Function, making it one reusable function definition instead of several separate definitions. This eliminates structural objectives, which were previously satisfied. Although the metrics appear to slightly decrease, this is simply due to the elimination of clones from the model, that were each adding to the satisfied objectives.

6.3.3.6 Performance Comparison

To determine whether there was a change in efficiency between the original system and our restructured system, the SDS was simulated via SiL. The

results are reported in Table 6.3. The new structure of the system reduced the ACET time by approximately 44%. Although the new modularization increased the C program by 9,253 lines, the global memory decreased from 2,015 bytes to 461 bytes due to the reusability of the modules.

6.3.4 Case Study Summary

In this section, the proposed Simulink module approach was demonstrated and evaluated on a nuclear example. We decomposed the system with the intent of grouping together related functionality, and hiding the secrets of the system in Simulink modules. This was largely facilitated by the use of the Simulink Module Tool throughout the decomposition and evaluation process. In our evaluation, we examined some well-known indicators of modularity and information hiding. We observed that the restructured system had better support for information hiding due to the appropriate scoping of design secrets in our Simulink module structure. Applying a change to the system also demonstrated that changes were restricted to the Simulink module, and did not propagate to other parts of the system—unlike the original design. There was also a decrease in interface complexity, as the new decomposition hid previously exposed internal functions, thus removing them from the interface. This in turn resulted in a decrease in coupling, as the system interactions were reduced. An increase in cohesion was observed due to the grouping of related functionality into Simulink modules. This was a stark difference from the original design, which stored all functionality together in one large Library. The cyclomatic complexity of the SDS decreased due to the decomposition of the system utilizing Simulink Functions, which create designs that are more

reusable, as compared to Library blocks. Although there were no significant changes to structural test metrics, there was a large reduction in the number of coverage objectives, also due to the reusable nature of Simulink Functions. The reusability of the Simulink Functions in the generated C code, as compared to blocks stored in a Library, resulted in significant improvements to average and worst case execution time on SiL. In summary, the use of the proposed Simulink module concept for decomposing a Simulink system exhibits improvements in key qualities that are indicative of modularity. The application of our proposed approach lead to more a modular design overall.

6.4 Challenges and Limitations

This section discusses the challenges encountered in the application of the approach. Some have been acknowledged by MathWorks as limitations of the Simulink Function block itself, or more generally atomic Subsystems [The MathWorks, 2020j]. These are factors that should be considered prior to applying the approach. Alternatively, conventions using other decomposition constructs are available [Jaskolka et al., 2020a] when switching to Simulink Functions is not possible.

6.4.1 Variable-Step Solvers and Continuous States

Our approach is suitable for discrete, fixed-step controllers. It is not applicable for systems using variable-step solvers or containing blocks with continuous time (e.g., Simscape blocks) [The MathWorks, 2020j]. Using it in this situation will result in a simulation error when executing the model. As a result, this approach is not suited to physical modelling components, such

as plants. Furthermore, SDV is not compatible with variable-step solvers, so equivalence-checking and testing would not be possible.

6.4.2 Inheriting Sample Time

All blocks nested in a Simulink Function must inherit sample time. As a result, Subsystems that incorporate multiple sample times cannot be modularized solely by means of Simulink Functions. For these systems, it is recommended that other conventions for modularity, such as those proposed in [Jaskolka et al., 2020a], be used to achieve information hiding. As a workaround to this issue, blocks can be grouped into Models based on sample time, then decomposed using Simulink Functions in each Model Reference.

6.4.3 Block States

Simulink Functions hold state between multiple Function Callers [Jaskolka et al., 2020a]. As a result, using a Simulink Function to encapsulate state-holding blocks, such as Unit Delays and Discrete-Time Integrators, results in undesired behaviour between multiple Function Callers. Thus, Subsystems containing state-holding blocks should not be modularized using Simulink Functions if function reusability is desired. A workaround is to move all state-containing blocks outside of the Subsystem, prior to converting to a Simulink Function. This becomes more difficult as the amount of state-holding blocks increases.

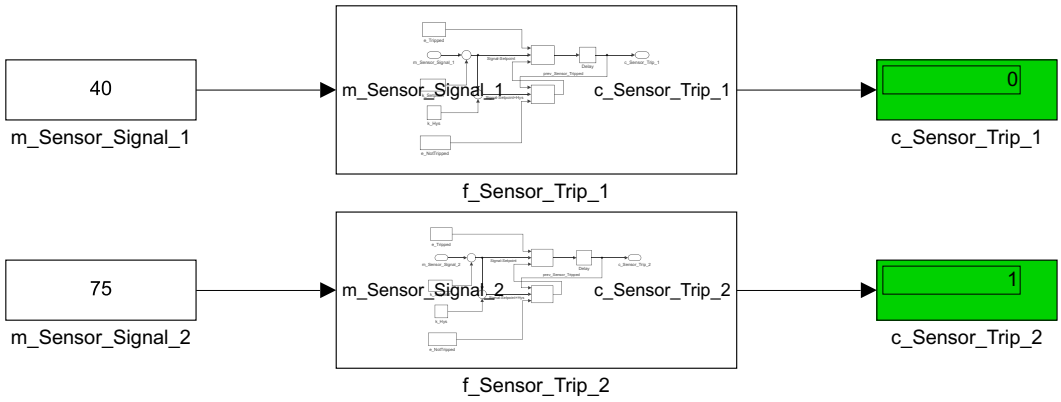
An example that demonstrates incorrect behaviour due to state retention between multiple Function Callers is shown in Figure 6.12. It is a simple sensor trip algorithm. The sensor is tripped (1) when then input is above the setpoint (50), otherwise the sensor is not tripped (0). There is a region of no-change

between 45-50 in which the sensor cannot change from a tripped to not tripped state. We provide inputs of 40 and 75 to sensors 1 and 2, respectively. This results in the first sensor not being tripped, but the second is tripped (see green Display blocks).

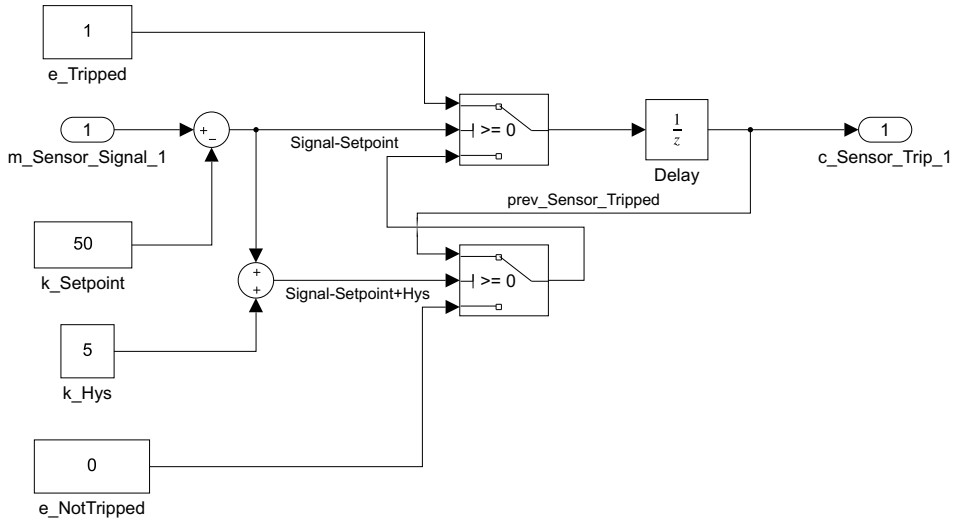
In the correct example (Figure 6.13), the same behaviour is exhibited. In the incorrect example (Figure 6.14), the same inputs were provided to the sensors, but the output is the opposite of what is expected. This is due to the Unit Delay block in the sensor trip Subsystem, which is used to feedback the previous value of the sensor into the sensor trip algorithm. When the Simulink Function is called multiple times, the state of the Unit Delay is held between the two callers, resulting in incorrect behaviour in the separate sensors. Because of the nature of this design, the Unit Delay was moved outside of the Simulink Function, and the resulting output of the Unit Delay was routed back as an input to the Simulink Function. This type of change was implemented in various locations in the SDS, as well as for the roll-off filter block in the FCC.

6.4.4 Algebraic Loops

When a virtual Subsystem is converted into a non-virtual Subsystem (e.g., Simulink Function), the inadvertent introduction of algebraic loops is a common issue [The MathWorks, 2020j]. This occurs as a result of a circular dependency between blocks. In the FCC system, the Simulink environment automatically diagnosed the issue during simulation and it was resolved by moving a Unit Delay to prevent direct feedback.

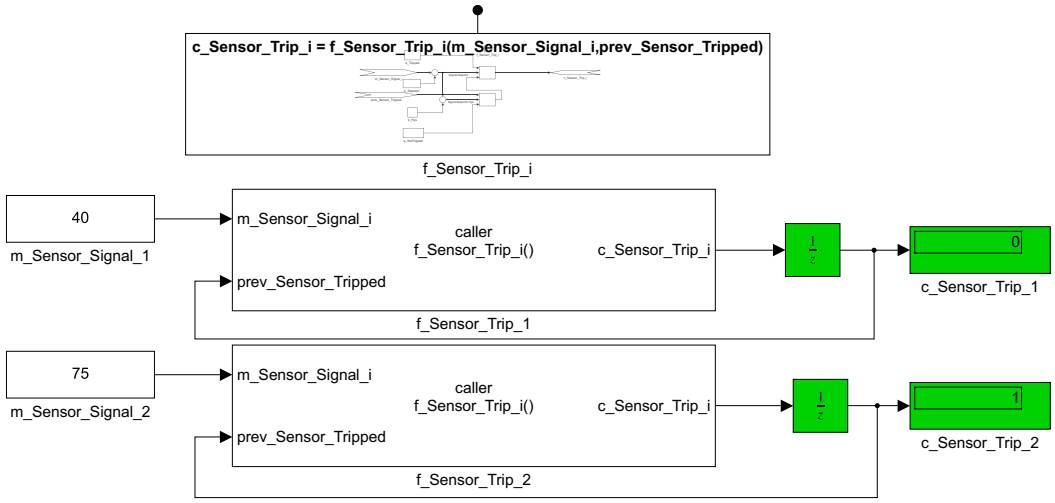


(a) Top-level of original sensor trip.

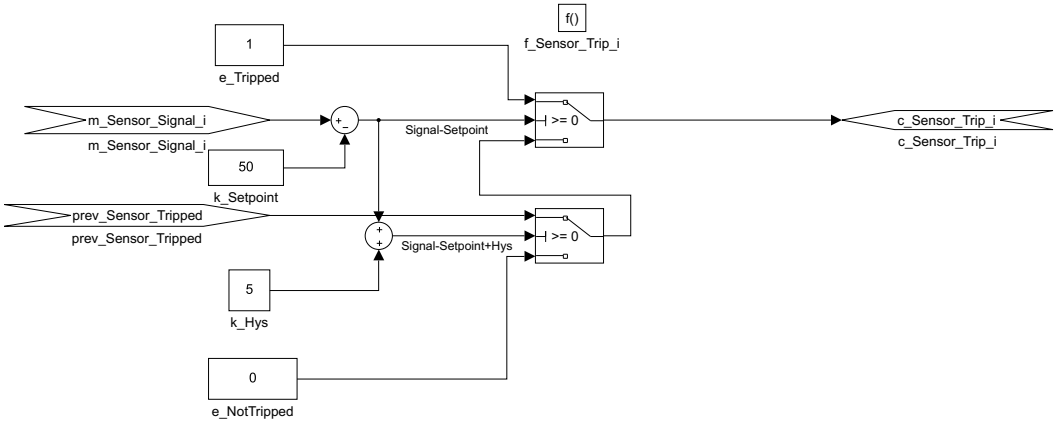


(b) Internals of f_Sensor_Trip_i from Figure 6.12a.

Figure 6.12: Original sensor trip example.

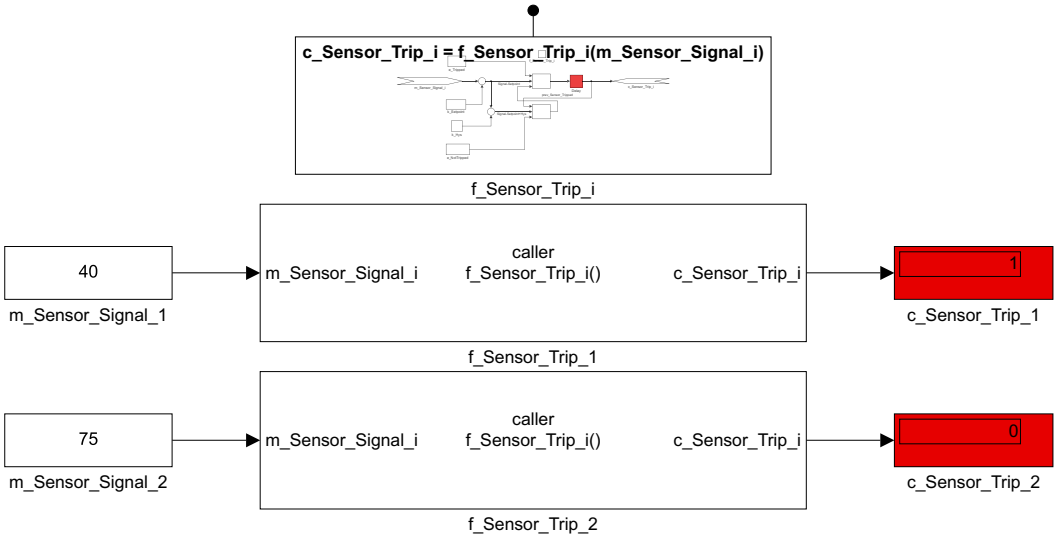


(a) Top-level of correct sensor trip.

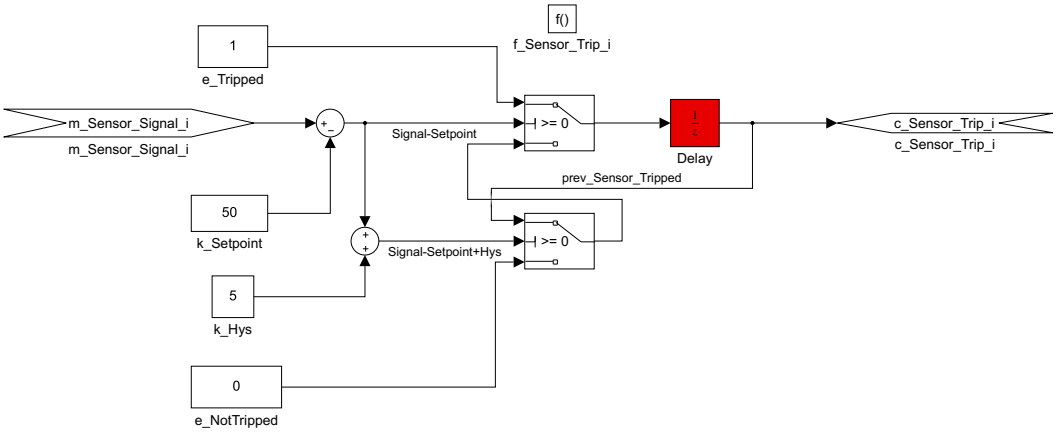


(b) Internals of f_Sensor_Trip_i Simulink Function from Figure 6.13a.

Figure 6.13: Correct new sensor trip example.



(a) Top-level of incorrect sensor trip.



(b) Internals of f_Sensor_Trip_i Simulink Function from Figure 6.14a.

Figure 6.14: Incorrect new sensor trip example.

In summary, there are use cases that do not lend themselves to our modularization approach. Limitations arise due to the nature of Simulink Functions, such as the need to use particular solvers, inherit sample time, or prevent the use of stateful blocks.

Chapter 7

Conclusion

As with all software, Simulink models are constantly maintained and are subject to evolution over their lifetime. The increasing complexity of Simulink models, and their position as primary design artifacts maintained over many years, demands approaches for ensuring that Simulink models are robust with respect to change.

7.1 Summary of Contributions

This thesis began with a survey of likely changes in industrial Simulink models. The Model Comparison Utility was developed to support the analysis of model differences. Millions of Simulink models from an industry version control system were analyzed and frequent changes were categorized. This activity revealed that the interfaces and model structure were particularly volatile. The large amount of changes to internal signal passing mechanisms also indicated that changes were propagating throughout models. Modularity and information hiding were identified as principles that

would enable Simulink designs to be more robust with respect to future changes. However, it was unclear how information hiding could be leveraged in a graphical language such as Simulink. To understand how to adapt these practices for Model-Based Development (MBD) with Simulink, this thesis presented a comparison of Simulink constructs to the constructs of the C language. Then, a novel approach for modular modelling was proposed and described. The approach entails structuring models as Simulink modules to support encapsulation and facilitate information hiding. The definition of a module interface was given, which effectively represents all data flow across the module boundary. For those cases in which Simulink Functions are not appropriate, two alternative ways of structuring Simulink modules were created. These two conventions rely on guidelines rather than enforcement by the Simulink language itself. Four new guidelines to encourage modelling best practices were presented. The Simulink Module Tool was developed to automate the aforementioned contributions and made available as an open-source contribution on GitHub and the MATLAB Central File Exchange.

The contributions were evaluated in two case studies: an open-source Flight Control Computer (FCC) from the aerospace domain and a proprietary Shut Down System (SDS) from the nuclear domain. We demonstrated that in both case studies, the restructured system that leveraged Simulink modules increased support for information hiding, simplified interfaces, decreased module coupling, and increased cohesion. Changes applied to the restructured systems were better encapsulated, and did not propagate to other parts of the system. Impact on interfaces as a result of performing a change was also eliminated. In the smaller, more

simple Flight Control Computer (FCC) system, an increase in cyclomatic complexity was observed. This was because our approach relies on Simulink Functions as a decomposition construct, and each added Simulink Function contributed a complexity value. However, on the larger Shut Down System (SDS), using Simulink Functions significantly reduced cyclomatic complexity because they are a reusable construct that does not result in clones throughout the model. The FCC did not have a high amount of reuse, so it did not benefit in the same way. Again, the larger SDS benefitted from the Simulink Functions by increased decision coverage, while the aerospace case study did not. Otherwise, there were no significant changes to the testability of the systems as a result of the Simulink module structure. Again, the reusability of Simulink Functions resulted in a substantial reduction in average execution time for the SDS, but incurred a performance penalty in the FCC. Nevertheless, the case studies show that production industrial systems that are medium to large in size would benefit from the use of Simulink modules as a new approach for achieving designs that are robust with respect to change.

7.2 Future Work

This section describes how the body of this work can be further improved and how it leads to directions for future research.

Expanding to More Languages As future work, we plan to extend this approach to the Stateflow environment, which is a separate language that models finite state machines. First, we wish to apply the repository mining

approach on Stateflow implementations, to understand how its basic elements (e.g., states, transitions, and junctions) are affected by model changes. Then, recommendations on Stateflow design structure should be explored, in a similar fashion as we did for Simulink.

There are several other graphical data flow languages which may benefit from a similar treatment. Future work will entail studying languages such as Safety Critical Application Development Environment (SCADE) [ANSYS, 2020], Modelica [The Modelica Association, 2020], Laboratory Virtual Instruments Engineering Workbench (LabVIEW) [National Instruments, 2020], Ptolemy II [University of California at Berkeley, 2020], and others. A broader view of several languages used for MBD may lead to observations about common obstacles present in these languages, or the MBD domain in general.

Open-source Repositories The findings of Chapter 3 are based on a single, large, long-term engineering project encompassing a product line from an industry partner. The rationale for the case study selection was simply that we were granted unprecedented access to an industry product development Change Management System (CMS). This gave us an unparalleled opportunity to obtain insights into years worth of data, which is typically exceedingly difficult for researchers to obtain. It would be beneficial to expand our analysis to additional repositories, particularly those which are open-source. Currently, there are 212 GitHub repositories on the topic of Simulink,[1] and 38 on Simulink models specifically.[2] These could form the basis of another study on model changes.

[1] https://github.com/topics/simulink
[2] https://github.com/topics/simulink-model

Additional Model Changes The MathWorks Simulink Comparison Tool does not provide information on changes to layout and other nonfunctional changes from the command line. MathWorks has been made aware of this limitation, and an enhancement request has been submitted to its developers. It would be beneficial to consider these types of changes since previous studies have shown that developers spend around 30% of their time on model layout [Klauske and Dziobek, 2010].

Furthermore, we wish to examine more complex, compound changes in Simulink models. Each change is currently treated as an individual change, without considering the relationships between changes. For example, we would also like to support identifying block replacements (deletions followed by additions), and other more complex changes that are not identified by the MathWorks Simulink Comparison Tool. For this, an automated identification/categorization based on patterns is needed.

We also plan to look into determining the top n (e.g., $n = 10$, 15, etc.) types of changes that are made to a model. This will involve analyzing combinations of elements that are changed in a model in a single commit, most likely with frequent pattern mining techniques [Han et al., 2007]. For example, a frequent change may be to change one or more inports as well as one or more subsystems; or perhaps to change only subsystems and logic within the model. The number of combinations is large, and this creates an even more complex mining and evaluation procedure.

Software Engineering Principles The focus of this research was on the information hiding and modularity principles. However, there are many other software engineering principles (e.g., object orientation) which could prove

beneficial if mapped to the MBD approach. Future directions for research should entail a comprehensive review of which software engineering principles need to be better supported in Simulink, followed by further work towards addressing any identified gaps.

7.3 Closing Remarks

Simulink is one of the most popular languages used for the model-based development of embedded software systems. Because of the size, complexity, and safety-critical nature of systems developed using Simulink, we need to focus our efforts on supporting software engineering principles for the MBD paradigm. Creating designs utilizing principles of modularity and information hiding is just the first step in adapting principles that will ease model evolution, maintenance, and ultimately make designs robust with respect to change.

Appendix A

Construct Comparison Generated Code

This appendix provides the code generation outcomes for various Simulink decomposition constructs. Each section shows the Simulink model and the corresponding generated C code.

A.1 Virtual Subsystem Generated Code

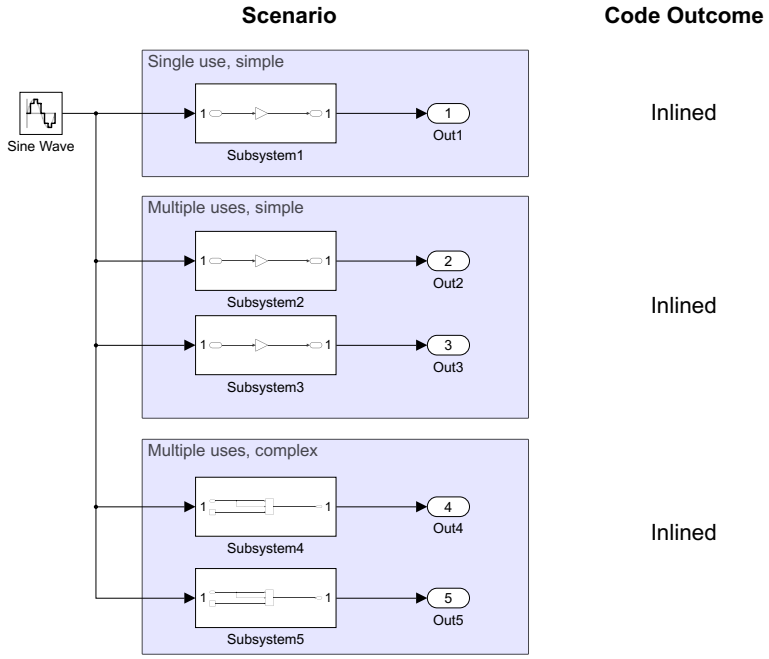


Figure A.1: Simulink model for generating virtual Subsystem code in order to determine C code outcomes.

Listing A.1: VirtualSubsystem.c

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: VirtualSubsystem.c
 *
 * Code generated for Simulink model 'VirtualSubsystem'.
 *
 * Model version                  : 1.7
 * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 13:37:39 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#include "VirtualSubsystem.h"

/* Block signals and states (default storage) */
DW rtDW;

/* External outputs (root outports fed by signals with default storage) */
ExtY rtY;

/* Real-time model */
RT_MODEL rtM_;
RT_MODEL *const rtM = &rtM_;

/* Model step function */
void VirtualSubsystem_step(void)
{
  real_T rtb_SineWave;

  /* Sin: '<Root>/Sine Wave' */
  if (rtDW.systemEnable != 0) {
    rtb_SineWave = ((rtM->Timing.clockTick0) * 0.1);
    rtDW.lastSin = sin(rtb_SineWave);
    rtDW.lastCos = cos(rtb_SineWave);
    rtDW.systemEnable = 0;
  }

  rtb_SineWave = (rtDW.lastSin * 0.99500416527802571 + rtDW.lastCos *
    -0.099833416646828155) * 0.99500416527802571 + (rtDW.lastCos *
    0.99500416527802571 - rtDW.lastSin * -0.099833416646828155) *
    0.099833416646828155;

  /* End of Sin: '<Root>/Sine Wave' */

  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<S1>/Gain'
   */
  rtY.Out1 = 2.0 * rtb_SineWave;

  /* Outport: '<Root>/Out2' incorporates:
   *  Gain: '<S2>/Gain'
   */
  rtY.Out2 = 3.0 * rtb_SineWave;

```

```
64    /* Outport: '<Root>/Out3' incorporates:
65     *  Gain: '<S3>/Gain'
66     */
67    rtY.Out3 = 3.0 * rtb_SineWave;
68
69    /* Switch: '<S4>/Switch' incorporates:
70     *  Switch: '<S5>/Switch'
71     */
72    if (rtb_SineWave > 0.0) {
73      /* Outport: '<Root>/Out4' */
74      rtY.Out4 = rtb_SineWave;
75
76      /* Outport: '<Root>/Out5' */
77      rtY.Out5 = rtb_SineWave;
78    } else {
79      /* Outport: '<Root>/Out4' incorporates:
80       *  Constant: '<S4>/Constant'
81       */
82      rtY.Out4 = 1.0;
83
84      /* Outport: '<Root>/Out5' incorporates:
85       *  Constant: '<S5>/Constant'
86       */
87      rtY.Out5 = 1.0;
88    }
89
90    /* End of Switch: '<S4>/Switch' */
91
92    /* Update for Sin: '<Root>/Sine Wave' */
93    rtb_SineWave = rtDW.lastSin;
94    rtDW.lastSin = rtDW.lastSin * 0.99500416527802571 + rtDW.lastCos * 0.099833416646828155;
95    rtDW.lastCos = rtDW.lastCos * 0.99500416527802571 - rtb_SineWave * 0.099833416646828155;
96
97    /* Update absolute time for base rate */
98    /* The "clockTick0" counts the number of times the code of this task has
99     * been executed. The resolution of this integer timer is 0.1, which is the step size
100    * of the task. Size of "clockTick0" ensures timer will not overflow during the
101    * application lifespan selected.
102    */
103   rtM->Timing.clockTick0++;
104 }
105
106 /* Model initialize function */
107 void VirtualSubsystem_initialize(void)
108 {
109   /* Enable for Sin: '<Root>/Sine Wave' */
110   rtDW.systemEnable = 1;
111 }
112
113 /*
114  * File trailer for generated code.
115  *
116  * [EOF]
117  */
```

Listing A.2: VirtualSubsystem.h

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: VirtualSubsystem.h
 *
 * Code generated for Simulink model 'VirtualSubsystem'.
 *
 * Model version              : 1.7
 * Simulink Coder version     : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 13:37:39 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#ifndef RTW_HEADER_VirtualSubsystem_h_
#define RTW_HEADER_VirtualSubsystem_h_
#include <math.h>
#ifndef VirtualSubsystem_COMMON_INCLUDES_
# define VirtualSubsystem_COMMON_INCLUDES_
#include "rtwtypes.h"
#endif                                 /* VirtualSubsystem_COMMON_INCLUDES_ */

/* Model Code Variants */

/* Macros for accessing real-time model data structure */

/* Forward declaration for rtModel */
typedef struct tag_RTM RT_MODEL;

/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  real_T lastSin;                      /* '<Root>/Sine Wave' */
  real_T lastCos;                      /* '<Root>/Sine Wave' */
  int32_T systemEnable;                /* '<Root>/Sine Wave' */
} DW;

/* External outputs (root outports fed by signals with default storage) */
typedef struct {
  real_T Out1;                         /* '<Root>/Out1' */
  real_T Out2;                         /* '<Root>/Out2' */
  real_T Out3;                         /* '<Root>/Out3' */
  real_T Out4;                         /* '<Root>/Out4' */
  real_T Out5;                         /* '<Root>/Out5' */
} ExtY;

/* Real-time Model Data Structure */
struct tag_RTM {
  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick0;
  } Timing;
};
```

```
64
65 /* Block signals and states (default storage) */
66 extern DW rtDW;
67
68 /* External outputs (root outports fed by signals with default storage) */
69 extern ExtY rtY;
70
71 /* Model entry point functions */
72 extern void VirtualSubsystem_initialize(void);
73 extern void VirtualSubsystem_step(void);
74
75 /* Real-time Model object */
76 extern RT_MODEL *const rtM;
77
78 /*-
79  * The generated code includes comments that allow you to trace directly
80  * back to the appropriate location in the model. The basic format
81  * is <system>/block_name, where system is the system number (uniquely
82  * assigned by Simulink) and block_name is the name of the block.
83  *
84  * Use the MATLAB hilite_system command to trace the generated code back
85  * to the model. For example,
86  *
87  * hilite_system('<S3>') - opens system 3
88  * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
89  *
90  * Here is the system hierarchy for this model
91  *
92  * '<Root>' : 'VirtualSubsystem'
93  * '<S1>'  : 'VirtualSubsystem/Subsystem1'
94  * '<S2>'  : 'VirtualSubsystem/Subsystem2'
95  * '<S3>'  : 'VirtualSubsystem/Subsystem3'
96  * '<S4>'  : 'VirtualSubsystem/Subsystem4'
97  * '<S5>'  : 'VirtualSubsystem/Subsystem5'
98  */
99 #endif                             /* RTW_HEADER_VirtualSubsystem_h_ */
100
101 /*
102  * File trailer for generated code.
103  *
104  * [EOF]
105  */
```

A.2 Atomic Subsystem Generated Code

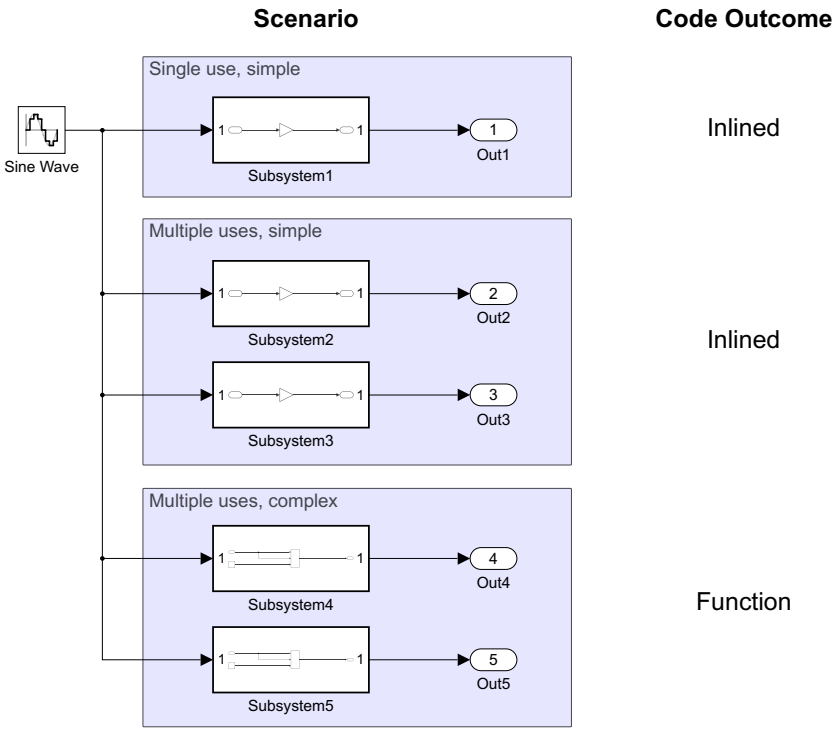


Figure A.2: Simulink model for generating Atomic Subsystem code in order to determine C code outcomes.

Listing A.3: AtomicSubsystem.c

```
1  /*
2   * Academic License - for use in teaching, academic research, and meeting
3   * course requirements at degree granting institutions only. Not for
4   * government, commercial, or other organizational use.
5   *
6   * File: AtomicSubsystem.c
7   *
8   * Code generated for Simulink model 'AtomicSubsystem'.
9   *
10  * Model version                  : 1.9
11  * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
12  * C/C++ source code generated on : Sat Oct 3 13:29:52 2020
13  *
14  * Target selection: ert.tlc
15  * Embedded hardware selection: Intel->x86-64 (Windows64)
16  * Code generation objectives:
17  *    1. Execution efficiency
18  *    2. RAM efficiency
19  * Validation result: Not run
20  */
21
22 #include "AtomicSubsystem.h"
```

```
23
24 /* Block signals and states (default storage) */
25 DW rtDW;
26
27 /* External outputs (root outports fed by signals with default storage) */
28 ExtY rtY;
29
30 /* Real-time model */
31 RT_MODEL rtM_;
32 RT_MODEL *const rtM = &rtM_;
33 static real_T Subsystem4(real_T rtu_In1);
34
35 /*
36  * Output and update for atomic system:
37  *    '<Root>/Subsystem4'
38  *    '<Root>/Subsystem5'
39  */
40 static real_T Subsystem4(real_T rtu_In1)
41 {
42   real_T rty_Out1_0;
43
44   /* Switch: '<S4>/Switch' incorporates:
45    *  Constant: '<S4>/Constant'
46    */
47   if (rtu_In1 > 0.0) {
48     rty_Out1_0 = rtu_In1;
49   } else {
50     rty_Out1_0 = 1.0;
51   }
52
53   /* End of Switch: '<S4>/Switch' */
54   return rty_Out1_0;
55 }
56
57 /* Model step function */
58 void AtomicSubsystem_step(void)
59 {
60   real_T rtb_SineWave;
61
62   /* Sin: '<Root>/Sine Wave' */
63   if (rtDW.systemEnable != 0) {
64     rtb_SineWave = ((rtM->Timing.clockTick0) * 0.1);
65     rtDW.lastSin = sin(rtb_SineWave);
66     rtDW.lastCos = cos(rtb_SineWave);
67     rtDW.systemEnable = 0;
68   }
69
70   rtb_SineWave = (rtDW.lastSin * 0.99500416527802571 + rtDW.lastCos *
71                   -0.099833416646828155) * 0.99500416527802571 + (rtDW.lastCos *
72     0.99500416527802571 - rtDW.lastSin * -0.099833416646828155) *
73     0.099833416646828155;
74
75   /* End of Sin: '<Root>/Sine Wave' */
76
77   /* Outputs for Atomic SubSystem: '<Root>/Subsystem1' */
78   /* Outport: '<Root>/Out1' incorporates:
79    *  Gain: '<S1>/Gain'
80    */
81   rtY.Out1 = 2.0 * rtb_SineWave;
82
83   /* End of Outputs for SubSystem: '<Root>/Subsystem1' */
84
85   /* Outputs for Atomic SubSystem: '<Root>/Subsystem2' */
86   /* Outport: '<Root>/Out2' incorporates:
```

```
87     *  Gain: '<S2>/Gain'
88     */
89    rtY.Out2 = 3.0 * rtb_SineWave;
90
91    /* End of Outputs for SubSystem: '<Root>/Subsystem2' */
92
93    /* Outputs for Atomic SubSystem: '<Root>/Subsystem3' */
94    /* Outport: '<Root>/Out3' incorporates:
95     *  Gain: '<S3>/Gain'
96     */
97    rtY.Out3 = 3.0 * rtb_SineWave;
98
99    /* End of Outputs for SubSystem: '<Root>/Subsystem3' */
100
101   /* Outputs for Atomic SubSystem: '<Root>/Subsystem4' */
102   /* Outport: '<Root>/Out4' */
103   rtY.Out4 = Subsystem4(rtb_SineWave);
104
105   /* End of Outputs for SubSystem: '<Root>/Subsystem4' */
106
107   /* Outputs for Atomic SubSystem: '<Root>/Subsystem5' */
108   /* Outport: '<Root>/Out5' */
109   rtY.Out5 = Subsystem4(rtb_SineWave);
110
111   /* End of Outputs for SubSystem: '<Root>/Subsystem5' */
112
113   /* Update for Sin: '<Root>/Sine Wave' */
114   rtb_SineWave = rtDW.lastSin;
115   rtDW.lastSin = rtDW.lastSin * 0.99500416527802571 + rtDW.lastCos *
116     0.099833416646828155;
117   rtDW.lastCos = rtDW.lastCos * 0.99500416527802571 - rtb_SineWave *
118     0.099833416646828155;
119
120   /* Update absolute time for base rate */
121   /* The "clockTick0" counts the number of times the code of this task has
122    * been executed. The resolution of this integer timer is 0.1, which is the step size
123    * of the task. Size of "clockTick0" ensures timer will not overflow during the
124    * application lifespan selected.
125    */
126   rtM->Timing.clockTick0++;
127 }
128
129 /* Model initialize function */
130 void AtomicSubsystem_initialize(void)
131 {
132   /* Enable for Sin: '<Root>/Sine Wave' */
133   rtDW.systemEnable = 1;
134 }
135
136 /*
137  * File trailer for generated code.
138  *
139  * [EOF]
140  */
```

Listing A.4: VirtualSubsystem.h

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: AtomicSubsystem.h
 *
 * Code generated for Simulink model 'AtomicSubsystem'.
 *
 * Model version              : 1.9
 * Simulink Coder version     : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 13:29:52 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#ifndef RTW_HEADER_AtomicSubsystem_h_
#define RTW_HEADER_AtomicSubsystem_h_
#include <math.h>
#ifndef AtomicSubsystem_COMMON_INCLUDES_
# define AtomicSubsystem_COMMON_INCLUDES_
#include "rtwtypes.h"
#endif                                 /* AtomicSubsystem_COMMON_INCLUDES_ */

/* Model Code Variants */

/* Macros for accessing real-time model data structure */

/* Forward declaration for rtModel */
typedef struct tag_RTM RT_MODEL;

/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  real_T lastSin;                      /* '<Root>/Sine Wave' */
  real_T lastCos;                      /* '<Root>/Sine Wave' */
  int32_T systemEnable;                /* '<Root>/Sine Wave' */
} DW;

/* External outputs (root outports fed by signals with default storage) */
typedef struct {
  real_T Out1;                         /* '<Root>/Out1' */
  real_T Out2;                         /* '<Root>/Out2' */
  real_T Out3;                         /* '<Root>/Out3' */
  real_T Out4;                         /* '<Root>/Out4' */
  real_T Out5;                         /* '<Root>/Out5' */
} ExtY;

/* Real-time Model Data Structure */
struct tag_RTM {
  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick0;
  } Timing;
};
```

```
64
65 /* Block signals and states (default storage) */
66 extern DW rtDW;
67
68 /* External outputs (root outports fed by signals with default storage) */
69 extern ExtY rtY;
70
71 /* Model entry point functions */
72 extern void AtomicSubsystem_initialize(void);
73 extern void AtomicSubsystem_step(void);
74
75 /* Real-time Model object */
76 extern RT_MODEL *const rtM;
77
78 /*-
79  * The generated code includes comments that allow you to trace directly
80  * back to the appropriate location in the model. The basic format
81  * is <system>/block_name, where system is the system number (uniquely
82  * assigned by Simulink) and block_name is the name of the block.
83  *
84  * Use the MATLAB hilite_system command to trace the generated code back
85  * to the model. For example,
86  *
87  * hilite_system('<S3>') - opens system 3
88  * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
89  *
90  * Here is the system hierarchy for this model
91  *
92  * '<Root>' : 'AtomicSubsystem'
93  * '<S1>'  : 'AtomicSubsystem/Subsystem1'
94  * '<S2>'  : 'AtomicSubsystem/Subsystem2'
95  * '<S3>'  : 'AtomicSubsystem/Subsystem3'
96  * '<S4>'  : 'AtomicSubsystem/Subsystem4'
97  * '<S5>'  : 'AtomicSubsystem/Subsystem5'
98  */
99 #endif                             /* RTW_HEADER_AtomicSubsystem_h_ */
100
101 /*
102  * File trailer for generated code.
103  *
104  * [EOF]
105  */
```

A.3 Simulink Function Generated Code

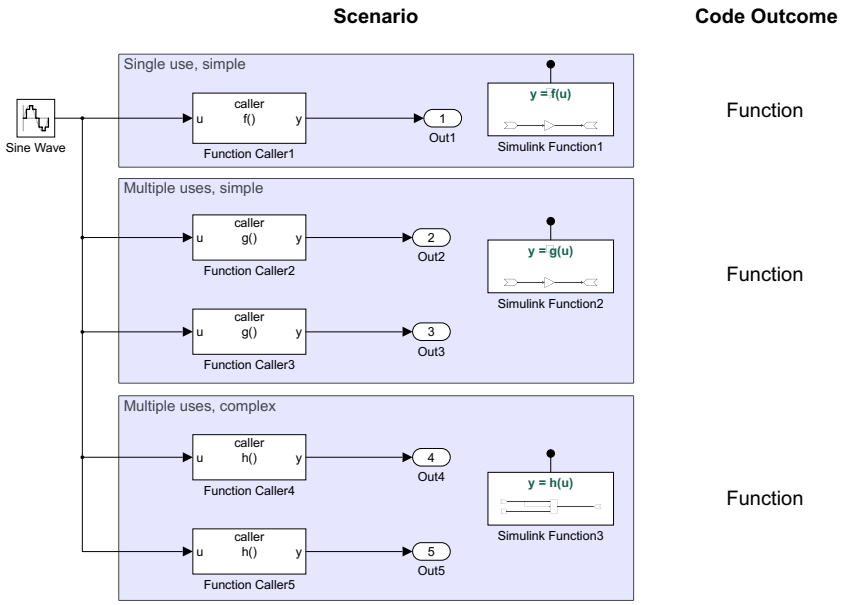


Figure A.3: Simulink model for generating Simulink Function code in order to determine C code outcomes.

Listing A.5: SimulinkFunction.c

```
1  /*
2   * Academic License - for use in teaching, academic research, and meeting
3   * course requirements at degree granting institutions only. Not for
4   * government, commercial, or other organizational use.
5   *
6   * File: SimulinkFunction.c
7   *
8   * Code generated for Simulink model 'SimulinkFunction'.
9   *
10  * Model version                  : 1.13
11  * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
12  * C/C++ source code generated on : Sat Oct 3 12:56:55 2020
13  *
14  * Target selection: ert.tlc
15  * Embedded hardware selection: Intel->x86-64 (Windows64)
16  * Code generation objectives:
17  *    1. Execution efficiency
18  *    2. RAM efficiency
19  * Validation result: Not run
20  */
21
22 #include "SimulinkFunction.h"
23
24 /* Block signals and states (default storage) */
25 DW rtDW;
26
27 /* External outputs (root outports fed by signals with default storage) */
```

```
28 ExtY rtY;
29
30 /* Real-time model */
31 RT_MODEL rtM_;
32 RT_MODEL *const rtM = &rtM_;
33
34 /* Output and update for Simulink Function: '<Root>/Simulink Function1' */
35 real_T SimulinkFunction_f(const real_T rtu_u)
36 {
37   /* SignalConversion generated from: '<S1>/y' incorporates:
38    *  Gain: '<S1>/Gain'
39    *  SignalConversion generated from: '<S1>/u'
40    */
41   return 2.0 * rtu_u;
42 }
43
44 /* Output and update for Simulink Function: '<Root>/Simulink Function2' */
45 real_T SimulinkFunction_g(const real_T rtu_u)
46 {
47   /* SignalConversion generated from: '<S2>/y' incorporates:
48    *  Gain: '<S2>/Gain'
49    *  SignalConversion generated from: '<S2>/u'
50    */
51   return 3.0 * rtu_u;
52 }
53
54 /* Output and update for Simulink Function: '<Root>/Simulink Function3' */
55 real_T SimulinkFunction_h(const real_T rtu_u)
56 {
57   real_T rty_y_0;
58
59   /* Switch: '<S3>/Switch' incorporates:
60    *  SignalConversion generated from: '<S3>/u'
61    */
62   if (rtu_u > 0.0) {
63     /* SignalConversion generated from: '<S3>/y' */
64     rty_y_0 = rtu_u;
65   } else {
66     /* SignalConversion generated from: '<S3>/y' incorporates:
67      *  Constant: '<S3>/Constant'
68      */
69     rty_y_0 = 1.0;
70   }
71
72   /* End of Switch: '<S3>/Switch' */
73   return rty_y_0;
74 }
75
76 /* Model step function */
77 void SimulinkFunction_step(void)
78 {
79   real_T rtb_FunctionCaller5;
80   real_T rtb_FunctionCaller4;
81   real_T rtb_FunctionCaller3;
82   real_T rtb_FunctionCaller2;
83   real_T rtb_FunctionCaller1;
84
85   /* Sin: '<Root>/Sine Wave' */
86   if (rtDW.systemEnable != 0) {
87     rtb_FunctionCaller5 = ((rtM->Timing.clockTick0) * 0.1);
88     rtDW.lastSin = sin(rtb_FunctionCaller5);
89     rtDW.lastCos = cos(rtb_FunctionCaller5);
90     rtDW.systemEnable = 0;
91   }
```

```
92
93    rtb_FunctionCaller5 = (rtDW.lastSin * 0.99500416527802571 + rtDW.lastCos *
94      -0.099833416646828155) * 0.99500416527802571 + (rtDW.lastCos *
95      0.99500416527802571 - rtDW.lastSin * -0.099833416646828155) *
96      0.099833416646828155;
97
98    /* End of Sin: '<Root>/Sine Wave' */
99
100   /* FunctionCaller: '<Root>/Function Caller1' */
101   rtb_FunctionCaller1 = SimulinkFunction_f(rtb_FunctionCaller5);
102
103   /* FunctionCaller: '<Root>/Function Caller2' */
104   rtb_FunctionCaller2 = SimulinkFunction_g(rtb_FunctionCaller5);
105
106   /* FunctionCaller: '<Root>/Function Caller3' */
107   rtb_FunctionCaller3 = SimulinkFunction_g(rtb_FunctionCaller5);
108
109   /* FunctionCaller: '<Root>/Function Caller4' */
110   rtb_FunctionCaller4 = SimulinkFunction_h(rtb_FunctionCaller5);
111
112   /* Outport: '<Root>/Out5' incorporates:
113    *  FunctionCaller: '<Root>/Function Caller5'
114    */
115   rtY.Out5 = SimulinkFunction_h(rtb_FunctionCaller5);
116
117   /* Outport: '<Root>/Out4' */
118   rtY.Out4 = rtb_FunctionCaller4;
119
120   /* Outport: '<Root>/Out3' */
121   rtY.Out3 = rtb_FunctionCaller3;
122
123   /* Outport: '<Root>/Out2' */
124   rtY.Out2 = rtb_FunctionCaller2;
125
126   /* Outport: '<Root>/Out1' */
127   rtY.Out1 = rtb_FunctionCaller1;
128
129   /* Update for Sin: '<Root>/Sine Wave' */
130   rtb_FunctionCaller5 = rtDW.lastSin;
131   rtDW.lastSin = rtDW.lastSin * 0.99500416527802571 + rtDW.lastCos * 0.099833416646828155;
132   rtDW.lastCos = rtDW.lastCos * 0.99500416527802571 - rtb_FunctionCaller5 *
133     0.099833416646828155;
134
135   /* Update absolute time for base rate */
136   /* The "clockTick0" counts the number of times the code of this task has
137    * been executed. The resolution of this integer timer is 0.1, which is the step size
138    * of the task. Size of "clockTick0" ensures timer will not overflow during the
139    * application lifespan selected.
140    */
141   rtM->Timing.clockTick0++;
142 }
143
144 /* Model initialize function */
145 void SimulinkFunction_initialize(void)
146 {
147   /* Enable for Sin: '<Root>/Sine Wave' */
148   rtDW.systemEnable = 1;
149 }
150
151 /*
152  * File trailer for generated code.
153  *
154  * [EOF]
155  */
```

Listing A.6: SimulinkFunction.h

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: SimulinkFunction.h
 *
 * Code generated for Simulink model 'SimulinkFunction'.
 *
 * Model version              : 1.13
 * Simulink Coder version     : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 12:56:55 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#ifndef RTW_HEADER_SimulinkFunction_h_
#define RTW_HEADER_SimulinkFunction_h_
#include <math.h>
#ifndef SimulinkFunction_COMMON_INCLUDES_
# define SimulinkFunction_COMMON_INCLUDES_
#include "rtwtypes.h"
#endif                                 /* SimulinkFunction_COMMON_INCLUDES_ */

/* Model Code Variants */

/* Macros for accessing real-time model data structure */
#ifndef rtmGetErrorStatus
# define rtmGetErrorStatus(rtm)    ((rtm)->errorStatus)
#endif

#ifndef rtmSetErrorStatus
# define rtmSetErrorStatus(rtm, val) ((rtm)->errorStatus = (val))
#endif

/* Forward declaration for rtModel */
typedef struct tag_RTM RT_MODEL;

/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  real_T lastSin;                      /* '<Root>/Sine Wave' */
  real_T lastCos;                      /* '<Root>/Sine Wave' */
  int32_T systemEnable;                /* '<Root>/Sine Wave' */
} DW;

/* External outputs (root outports fed by signals with default storage) */
typedef struct {
  real_T Out1;                         /* '<Root>/Out1' */
  real_T Out2;                         /* '<Root>/Out2' */
  real_T Out3;                         /* '<Root>/Out3' */
  real_T Out4;                         /* '<Root>/Out4' */
  real_T Out5;                         /* '<Root>/Out5' */
} ExtY;

/* Real-time Model Data Structure */
struct tag_RTM {
  const char_T * volatile errorStatus;

```

```
  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick0;
  } Timing;
};

/* Block signals and states (default storage) */
extern DW rtDW;

/* External outputs (root outports fed by signals with default storage) */
extern ExtY rtY;

/* Model entry point functions */
extern void SimulinkFunction_initialize(void);
extern void SimulinkFunction_step(void);
extern real_T SimulinkFunction_f(const real_T rtu_u);
extern real_T SimulinkFunction_g(const real_T rtu_u);
extern real_T SimulinkFunction_h(const real_T rtu_u);

/* Real-time Model object */
extern RT_MODEL *const rtM;

/*-
 * The generated code includes comments that allow you to trace directly
 * back to the appropriate location in the model. The basic format
 * is <system>/block_name, where system is the system number (uniquely
 * assigned by Simulink) and block_name is the name of the block.
 *
 * Use the MATLAB hilite_system command to trace the generated code back
 * to the model. For example,
 *
 * hilite_system('<S3>') - opens system 3
 * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
 *
 * Here is the system hierarchy for this model
 *
 * '<Root>' : 'SimulinkFunction'
 * '<S1>'  : 'SimulinkFunction/Simulink Function1'
 * '<S2>'  : 'SimulinkFunction/Simulink Function2'
 * '<S3>'  : 'SimulinkFunction/Simulink Function3'
 */
#endif                                 /* RTW_HEADER_SimulinkFunction_h_ */

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

A.4 Library Import Generated Code

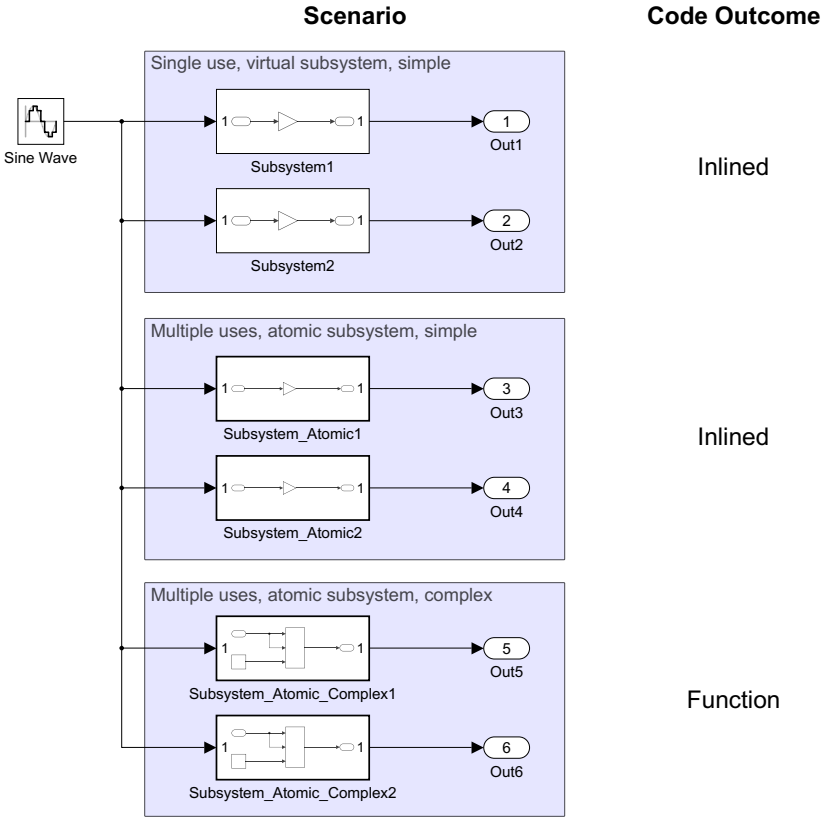


Figure A.4: Simulink model for generating Library code in order to determine C code outcomes.

Listing A.7: LibraryImport.c

```
1  /*
2   * Academic License - for use in teaching, academic research, and meeting
3   * course requirements at degree granting institutions only. Not for
4   * government, commercial, or other organizational use.
5   *
6   * File: LibraryImport.c
7   *
8   * Code generated for Simulink model 'LibraryImport'.
9   *
10  * Model version              : 1.14
11  * Simulink Coder version     : 9.3 (R2020a) 18-Nov-2019
12  * C/C++ source code generated on : Sat Oct 3 14:04:24 2020
13  *
14  * Target selection: ert.tlc
15  * Embedded hardware selection: Intel->x86-64 (Windows64)
16  * Code generation objectives:
17  *    1. Execution efficiency
18  *    2. RAM efficiency
19  * Validation result: Not run
```

```
 */

#include "LibraryImport.h"

/* Block signals and states (default storage) */
DW rtDW;

/* External outputs (root outports fed by signals with default storage) */
ExtY rtY;

/* Real-time model */
RT_MODEL rtM_;
RT_MODEL *const rtM = &rtM_;
static real_T Subsystem_Atomic_Complex1(real_T rtu_In1);

/*
 * Output and update for atomic system:
 *    '<Root>/Subsystem_Atomic_Complex1'
 *    '<Root>/Subsystem_Atomic_Complex2'
 */
static real_T Subsystem_Atomic_Complex1(real_T rtu_In1)
{
  real_T rty_Out1_0;

  /* Switch: '<S5>/Switch' incorporates:
   *  Constant: '<S5>/Constant'
   */
  if (rtu_In1 > 0.0) {
    rty_Out1_0 = rtu_In1;
  } else {
    rty_Out1_0 = 1.0;
  }

  /* End of Switch: '<S5>/Switch' */
  return rty_Out1_0;
}

/* Model step function */
void LibraryImport_step(void)
{
  real_T rtb_SineWave;

  /* Sin: '<Root>/Sine Wave' */
  if (rtDW.systemEnable != 0) {
    rtb_SineWave = ((rtM->Timing.clockTick0) * 0.1);
    rtDW.lastSin = sin(rtb_SineWave);
    rtDW.lastCos = cos(rtb_SineWave);
    rtDW.systemEnable = 0;
  }

  rtb_SineWave = (rtDW.lastSin * 0.99500416527802571 + rtDW.lastCos *
    -0.099833416646828155) * 0.99500416527802571 + (rtDW.lastCos *
    0.99500416527802571 - rtDW.lastSin * -0.099833416646828155) *
    0.099833416646828155;

  /* End of Sin: '<Root>/Sine Wave' */

  /* Outport: '<Root>/Out1' incorporates:
   *  Gain: '<S1>/Gain'
   */
  rtY.Out1 = 2.0 * rtb_SineWave;

  /* Outport: '<Root>/Out2' incorporates:
   *  Gain: '<S2>/Gain'
```

```
   */
  rtY.Out2 = 2.0 * rtb_SineWave;

  /* Outputs for Atomic SubSystem: '<Root>/Subsystem_Atomic1' */
  /* Outport: '<Root>/Out3' incorporates:
   *  Gain: '<S3>/Gain'
   */
  rtY.Out3 = 3.0 * rtb_SineWave;

  /* End of Outputs for SubSystem: '<Root>/Subsystem_Atomic1' */

  /* Outputs for Atomic SubSystem: '<Root>/Subsystem_Atomic2' */
  /* Outport: '<Root>/Out4' incorporates:
   *  Gain: '<S4>/Gain'
   */
  rtY.Out4 = 3.0 * rtb_SineWave;

  /* End of Outputs for SubSystem: '<Root>/Subsystem_Atomic2' */

  /* Outputs for Atomic SubSystem: '<Root>/Subsystem_Atomic_Complex1' */
  /* Outport: '<Root>/Out5' */
  rtY.Out5 = Subsystem_Atomic_Complex1(rtb_SineWave);

  /* End of Outputs for SubSystem: '<Root>/Subsystem_Atomic_Complex1' */

  /* Outputs for Atomic SubSystem: '<Root>/Subsystem_Atomic_Complex2' */
  /* Outport: '<Root>/Out6' */
  rtY.Out6 = Subsystem_Atomic_Complex1(rtb_SineWave);

  /* End of Outputs for SubSystem: '<Root>/Subsystem_Atomic_Complex2' */

  /* Update for Sin: '<Root>/Sine Wave' */
  rtb_SineWave = rtDW.lastSin;
  rtDW.lastSin = rtDW.lastSin * 0.99500416527802571 + rtDW.lastCos * 0.099833416646828155;
  rtDW.lastCos = rtDW.lastCos * 0.99500416527802571 - rtb_SineWave * 0.099833416646828155;

  /* Update absolute time for base rate */
  /* The "clockTick0" counts the number of times the code of this task has
   * been executed. The resolution of this integer timer is 0.1, which is the step size
   * of the task. Size of "clockTick0" ensures timer will not overflow during the
   * application lifespan selected.
   */
  rtM->Timing.clockTick0++;
}

/* Model initialize function */
void LibraryImport_initialize(void)
{
  /* Enable for Sin: '<Root>/Sine Wave' */
  rtDW.systemEnable = 1;
}

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

Listing A.8: LibraryImport.h

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: LibraryImport.h
 *
 * Code generated for Simulink model 'LibraryImport'.
 *
 * Model version                  : 1.14
 * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 14:04:24 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#ifndef RTW_HEADER_LibraryImport_h_
#define RTW_HEADER_LibraryImport_h_
#include <math.h>
#ifndef LibraryImport_COMMON_INCLUDES_
# define LibraryImport_COMMON_INCLUDES_
#include "rtwtypes.h"
#endif                                 /* LibraryImport_COMMON_INCLUDES_ */

/* Model Code Variants */

/* Macros for accessing real-time model data structure */

/* Forward declaration for rtModel */
typedef struct tag_RTM RT_MODEL;

/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  real_T lastSin;                      /* '<Root>/Sine Wave' */
  real_T lastCos;                      /* '<Root>/Sine Wave' */
  int32_T systemEnable;                /* '<Root>/Sine Wave' */
} DW;

/* External outputs (root outports fed by signals with default storage) */
typedef struct {
  real_T Out1;                         /* '<Root>/Out1' */
  real_T Out2;                         /* '<Root>/Out2' */
  real_T Out3;                         /* '<Root>/Out3' */
  real_T Out4;                         /* '<Root>/Out4' */
  real_T Out5;                         /* '<Root>/Out5' */
  real_T Out6;                         /* '<Root>/Out6' */
} ExtY;

/* Real-time Model Data Structure */
struct tag_RTM {
  /*
   * Timing:
   * The following substructure contains information regarding
   * the timing information for the model.
   */
  struct {
    uint32_T clockTick0;
  } Timing;
```

```
64 };
65
66 /* Block signals and states (default storage) */
67 extern DW rtDW;
68
69 /* External outputs (root outports fed by signals with default storage) */
70 extern ExtY rtY;
71
72 /* Model entry point functions */
73 extern void LibraryImport_initialize(void);
74 extern void LibraryImport_step(void);
75
76 /* Real-time Model object */
77 extern RT_MODEL *const rtM;
78
79 /*-
80  * The generated code includes comments that allow you to trace directly
81  * back to the appropriate location in the model. The basic format
82  * is <system>/block_name, where system is the system number (uniquely
83  * assigned by Simulink) and block_name is the name of the block.
84  *
85  * Use the MATLAB hilite_system command to trace the generated code back
86  * to the model. For example,
87  *
88  * hilite_system('<S3>') - opens system 3
89  * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
90  *
91  * Here is the system hierarchy for this model
92  *
93  * '<Root>' : 'LibraryImport'
94  * '<S1>'  : 'LibraryImport/Subsystem1'
95  * '<S2>'  : 'LibraryImport/Subsystem2'
96  * '<S3>'  : 'LibraryImport/Subsystem_Atomic1'
97  * '<S4>'  : 'LibraryImport/Subsystem_Atomic2'
98  * '<S5>'  : 'LibraryImport/Subsystem_Atomic_Complex1'
99  * '<S6>'  : 'LibraryImport/Subsystem_Atomic_Complex2'
100  */
101 #endif                                 /* RTW_HEADER_LibraryImport_h_ */
102
103 /*
104  * File trailer for generated code.
105  *
106  * [EOF]
107  */
```

A.5 Model Reference Generated Code

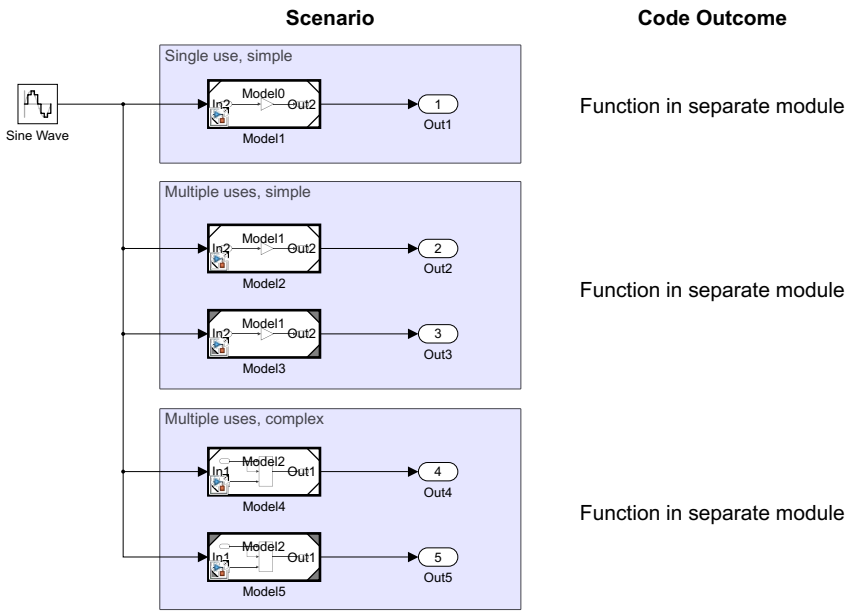


Figure A.5: Simulink model for generating Model Reference code in order to determine C code outcomes.

Listing A.9: ModelReference.c

```
1  /*
2   * Academic License - for use in teaching, academic research, and meeting
3   * course requirements at degree granting institutions only. Not for
4   * government, commercial, or other organizational use.
5   *
6   * File: ModelReference.h
7   *
8   * Code generated for Simulink model 'ModelReference'.
9   *
10  * Model version                  : 1.10
11  * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
12  * C/C++ source code generated on : Sat Oct 3 14:31:25 2020
13  *
14  * Target selection: ert.tlc
15  * Embedded hardware selection: Intel->x86-64 (Windows64)
16  * Code generation objectives:
17  *    1. Execution efficiency
18  *    2. RAM efficiency
19  * Validation result: Not run
20  */
21
22 #ifndef RTW_HEADER_ModelReference_h_
23 #define RTW_HEADER_ModelReference_h_
24 #include <math.h>
25 #ifndef ModelReference_COMMON_INCLUDES_
26 # define ModelReference_COMMON_INCLUDES_
27 #include "rtwtypes.h"
```

```
28 #endif                             /* ModelReference_COMMON_INCLUDES_ */
29
30 /* Child system includes */
31 #include "Model0.h"
32 #include "Model1.h"
33 #include "Model2.h"
34
35 /* Model Code Variants */
36
37 /* Macros for accessing real-time model data structure */
38
39 /* Forward declaration for rtModel */
40 typedef struct tag_RTM_ModelReference_T RT_MODEL_ModelReference_T;
41
42 /* Block signals and states (default storage) for system '<Root>' */
43 typedef struct {
44   real_T lastSin;                  /* '<Root>/Sine Wave' */
45   real_T lastCos;                  /* '<Root>/Sine Wave' */
46   int32_T systemEnable;            /* '<Root>/Sine Wave' */
47 } DW_ModelReference_T;
48
49 /* External outputs (root outports fed by signals with default storage) */
50 typedef struct {
51   real_T Out1;                     /* '<Root>/Out1' */
52   real_T Out2;                     /* '<Root>/Out2' */
53   real_T Out3;                     /* '<Root>/Out3' */
54   real_T Out4;                     /* '<Root>/Out4' */
55   real_T Out5;                     /* '<Root>/Out5' */
56 } ExtY_ModelReference_T;
57
58 /* Real-time Model Data Structure */
59 struct tag_RTM_ModelReference_T {
60   /*
61    * Timing:
62    * The following substructure contains information regarding
63    * the timing information for the model.
64    */
65   struct {
66     uint32_T clockTick0;
67   } Timing;
68 };
69
70 /* Block signals and states (default storage) */
71 extern DW_ModelReference_T ModelReference_DW;
72
73 /* External outputs (root outports fed by signals with default storage) */
74 extern ExtY_ModelReference_T ModelReference_Y;
75
76 /* Model entry point functions */
77 extern void ModelReference_initialize(void);
78 extern void ModelReference_step(void);
79
80 /* Real-time Model object */
81 extern RT_MODEL_ModelReference_T *const ModelReference_M;
82
83 /*-
84  * The generated code includes comments that allow you to trace directly
85  * back to the appropriate location in the model. The basic format
86  * is <system>/block_name, where system is the system number (uniquely
87  * assigned by Simulink) and block_name is the name of the block.
88  *
89  * Use the MATLAB hilite_system command to trace the generated code back
90  * to the model. For example,
91  *
```

```
 * hilite_system('<S3>') - opens system 3
 * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
 *
 * Here is the system hierarchy for this model
 *
 * '<Root>' : 'ModelReference'
 */
#endif                                 /* RTW_HEADER_ModelReference_h_ */

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

Listing A.10: ModelReference.h

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: ModelReference.h
 *
 * Code generated for Simulink model 'ModelReference'.
 *
 * Model version                  : 1.10
 * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 14:31:25 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#ifndef RTW_HEADER_ModelReference_h_
#define RTW_HEADER_ModelReference_h_
#include <math.h>
#ifndef ModelReference_COMMON_INCLUDES_
# define ModelReference_COMMON_INCLUDES_
#include "rtwtypes.h"
#endif                                 /* ModelReference_COMMON_INCLUDES_ */

/* Child system includes */
#include "Model0.h"
#include "Model1.h"
#include "Model2.h"

/* Model Code Variants */

/* Macros for accessing real-time model data structure */

/* Forward declaration for rtModel */
typedef struct tag_RTM_ModelReference_T RT_MODEL_ModelReference_T;

/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  real_T lastSin;                      /* '<Root>/Sine Wave' */
  real_T lastCos;                      /* '<Root>/Sine Wave' */
  int32_T systemEnable;                /* '<Root>/Sine Wave' */
} DW_ModelReference_T;
```

```
48
49  /* External outputs (root outports fed by signals with default storage) */
50  typedef struct {
51    real_T Out1;                       /* '<Root>/Out1' */
52    real_T Out2;                       /* '<Root>/Out2' */
53    real_T Out3;                       /* '<Root>/Out3' */
54    real_T Out4;                       /* '<Root>/Out4' */
55    real_T Out5;                       /* '<Root>/Out5' */
56  } ExtY_ModelReference_T;
57
58  /* Real-time Model Data Structure */
59  struct tag_RTM_ModelReference_T {
60    /*
61     * Timing:
62     * The following substructure contains information regarding
63     * the timing information for the model.
64     */
65    struct {
66      uint32_T clockTick0;
67    } Timing;
68  };
69
70  /* Block signals and states (default storage) */
71  extern DW_ModelReference_T ModelReference_DW;
72
73  /* External outputs (root outports fed by signals with default storage) */
74  extern ExtY_ModelReference_T ModelReference_Y;
75
76  /* Model entry point functions */
77  extern void ModelReference_initialize(void);
78  extern void ModelReference_step(void);
79
80  /* Real-time Model object */
81  extern RT_MODEL_ModelReference_T *const ModelReference_M;
82
83  /*-
84   * The generated code includes comments that allow you to trace directly
85   * back to the appropriate location in the model. The basic format
86   * is <system>/block_name, where system is the system number (uniquely
87   * assigned by Simulink) and block_name is the name of the block.
88   *
89   * Use the MATLAB hilite_system command to trace the generated code back
90   * to the model. For example,
91   *
92   * hilite_system('<S3>') - opens system 3
93   * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
94   *
95   * Here is the system hierarchy for this model
96   *
97   * '<Root>' : 'ModelReference'
98   */
99  #endif                             /* RTW_HEADER_ModelReference_h_ */
100
101 /*
102  * File trailer for generated code.
103  *
104  * [EOF]
105  */
```

Listing A.11: Model0.c

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: Model0.c
 *
 * Code generated for Simulink model 'Model0'.
 *
 * Model version                  : 1.9
 * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 14:30:57 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#include "Model0.h"

/* Output and update for referenced model: 'Model0' */
void Model0(const real_T *rtu_In2, real_T *rty_Out2)
{
  /* Gain: '<Root>/Gain' */
  *rty_Out2 = 2.0 * *rtu_In2;
}

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

Listing A.12: Model0.h

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: Model0.h
 *
 * Code generated for Simulink model 'Model0'.
 *
 * Model version                  : 1.9
 * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 14:30:57 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#ifndef RTW_HEADER_Model0_h_
#define RTW_HEADER_Model0_h_
#ifndef Model0_COMMON_INCLUDES_
```

```
# define Model0_COMMON_INCLUDES_
#include "rtwtypes.h"
#endif                                 /* Model0_COMMON_INCLUDES_ */

/* Model Code Variants */
extern void Model0(const real_T *rtu_In2, real_T *rty_Out2);

/*-
 * The generated code includes comments that allow you to trace directly
 * back to the appropriate location in the model. The basic format
 * is <system>/block_name, where system is the system number (uniquely
 * assigned by Simulink) and block_name is the name of the block.
 *
 * Use the MATLAB hilite_system command to trace the generated code back
 * to the model. For example,
 *
 * hilite_system('<S3>') - opens system 3
 * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
 *
 * Here is the system hierarchy for this model
 *
 * '<Root>' : 'Model0'
 */
#endif                                 /* RTW_HEADER_Model0_h_ */

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

Listing A.13: Model1.c

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: Model1.c
 *
 * Code generated for Simulink model 'Model1'.
 *
 * Model version                  : 1.10
 * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 14:31:10 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#include "Model1.h"

/* Output and update for referenced model: 'Model1' */
void Model1(const real_T *rtu_In2, real_T *rty_Out2)
{
  /* Gain: '<Root>/Gain' */
  *rty_Out2 = 3.0 * *rtu_In2;
}

/*
```

```
 * File trailer for generated code.
 *
 * [EOF]
 */
```

Listing A.14: Model1.h

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: Model1.h
 *
 * Code generated for Simulink model 'Model1'.
 *
 * Model version                  : 1.10
 * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 14:31:10 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#ifndef RTW_HEADER_Model1_h_
#define RTW_HEADER_Model1_h_
#ifndef Model1_COMMON_INCLUDES_
# define Model1_COMMON_INCLUDES_
#include "rtwtypes.h"
#endif                                 /* Model1_COMMON_INCLUDES_ */

/* Model Code Variants */
extern void Model1(const real_T *rtu_In2, real_T *rty_Out2);

/*-
 * The generated code includes comments that allow you to trace directly
 * back to the appropriate location in the model. The basic format
 * is <system>/block_name, where system is the system number (uniquely
 * assigned by Simulink) and block_name is the name of the block.
 *
 * Use the MATLAB hilite_system command to trace the generated code back
 * to the model. For example,
 *
 * hilite_system('<S3>') - opens system 3
 * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
 *
 * Here is the system hierarchy for this model
 *
 * '<Root>' : 'Model1'
 */
#endif                                 /* RTW_HEADER_Model1_h_ */

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

Listing A.15: Model2.c

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: Model2.c
 *
 * Code generated for Simulink model 'Model2'.
 *
 * Model version                  : 1.9
 * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 14:24:19 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
 *    1. Execution efficiency
 *    2. RAM efficiency
 * Validation result: Not run
 */

#include "Model2.h"

/* Output and update for referenced model: 'Model2' */
void Model2(const real_T *rtu_In1, real_T *rty_Out1)
{
  /* Switch: '<Root>/Switch' incorporates:
   *  Constant: '<Root>/Constant'
   */
  if (*rtu_In1 > 0.0) {
    *rty_Out1 = *rtu_In1;
  } else {
    *rty_Out1 = 1.0;
  }

  /* End of Switch: '<Root>/Switch' */
}

/*
 * File trailer for generated code.
 *
 * [EOF]
 */
```

Listing A.16: Model2.h

```
/*
 * Academic License - for use in teaching, academic research, and meeting
 * course requirements at degree granting institutions only. Not for
 * government, commercial, or other organizational use.
 *
 * File: Model2.h
 *
 * Code generated for Simulink model 'Model2'.
 *
 * Model version                  : 1.9
 * Simulink Coder version         : 9.3 (R2020a) 18-Nov-2019
 * C/C++ source code generated on : Sat Oct 3 14:24:19 2020
 *
 * Target selection: ert.tlc
 * Embedded hardware selection: Intel->x86-64 (Windows64)
 * Code generation objectives:
```

```
17  *    1. Execution efficiency
18  *    2. RAM efficiency
19  * Validation result: Not run
20  */
21
22  #ifndef RTW_HEADER_Model2_h_
23  #define RTW_HEADER_Model2_h_
24  #ifndef Model2_COMMON_INCLUDES_
25  # define Model2_COMMON_INCLUDES_
26  #include "rtwtypes.h"
27  #endif                             /* Model2_COMMON_INCLUDES_ */
28
29  /* Model Code Variants */
30  extern void Model2(const real_T *rtu_In1, real_T *rty_Out1);
31
32  /*-
33   * The generated code includes comments that allow you to trace directly
34   * back to the appropriate location in the model. The basic format
35   * is <system>/block_name, where system is the system number (uniquely
36   * assigned by Simulink) and block_name is the name of the block.
37   *
38   * Use the MATLAB hilite_system command to trace the generated code back
39   * to the model. For example,
40   *
41   * hilite_system('<S3>') - opens system 3
42   * hilite_system('<S3>/Kp') - opens and selects block Kp which resides in S3
43   *
44   * Here is the system hierarchy for this model
45   *
46   * '<Root>' : 'Model2'
47   */
48  #endif                             /* RTW_HEADER_Model2_h_ */
49
50  /*
51   * File trailer for generated code.
52   *
53   * [EOF]
54   */
```

Bibliography

Abd-El-Hafiz, S. K. (2012). A metrics-based data mining approach for software clone detection. In *Proceedings of the IEEE 36th Annual Computer Software and Applications Conference*, pages 35–41, Izmir, Turkey. IEEE.

Akdur, D., Garousi, V., and Demirörs, O. (2018). A survey on modeling and model-driven engineering practices in the embedded software industry. *Software & Systems Modeling*, 91:62–82.

Alalfi, M. H., Cordy, J. R., Dean, T. R., Stephan, M., and Stevenson, A. (2012). Models are code too: Near-miss clone detection for simulink models. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 295–304. IEEE.

Alalfi, M. H., Rapos, E. J., Stevenson, A., Stephan, M., Dean, T. R., and Cordy, J. R. (2014). Semi-automatic identification and representation of subsystem variability in simulink models. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE.

Alalfi, M. H., Rapos, E. J., Stevenson, A., Stephan, M., Dean, T. R., and Cordy, J. R. (2019). Variability identification and representation for automotive Simulink models. In *Automotive Systems and Software Engineering*, pages 109–139. Springer International Publishing.

Amelunxen, C., Legros, E., Schürr, A., and Stürmer, I. (2008). Checking and enforcement of modeling guidelines with graph transformations. In *Applications of Graph Transformations with Industrial Relevance*, pages 313–328. Springer, Springer Berlin Heidelberg.

ANSYS (2020). SCADE suite. https://www.ansys.com/products/embedded-software/ansys-scade-suite. [Online; accessed Oct 2020].

AspenCore (2019). 2019 embedded markets study. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf. [Online; accessed Oct 2020].

Astrov, I. and Pedai, A. (2012). Three-rate neural control of tuav with coaxial rotor and ducted fan configuration for enhanced situational awareness. In *Proceedings of the 1st International Conference on Control, Automation and Information Sciences (ICCAIS)*, pages 78–83, Ho Chi Minh, Vietnam.

autosar.org (2018). Autosar classic platform release 4.4.0. https://www.autosar.org/standards/classic-platform/classic-platform-440. [Online; accessed Oct 2020].

Baldwin, C. Y. and Clark, K. B. (2000). *Design rules: The power of modularity*, volume 1. MIT press.

Banker, R. D., Datar, S. M., and Zweig, D. (1989). Software complexity and maintainability. In *Proceedings of the tenth international conference on Information Systems - ICIS '89*. ACM Press.

BARR Group (2018). Embedded systems safety & security survey.

Bender, M., Laurin, K., Lawford, M., Pantelic, V., Korobkine, A., Ong, J., Mackenzie, B., Bialy, M., and Postma, S. (2015). Signature required: Making Simulink data flow and interfaces explicit. *Science of Computer Programming*, 113, Part 1:29–50. Model Driven Development (Selected & extended papers from MODELSWARD 2014).

Berard, E. V. (1993). Abstraction, encapsulation, and information hiding. http://www.tonymarston.net/php-mysql/abstraction.txt. [Online; accessed Oct 2020].

Bialy, M., Pantelic, V., Jaskolka, J., Schaap, A., Patcas, L., Lawford, M., and Wassyng, A. (2016). Software engineering for model-based development by domain experts. In Griffor, E., editor, *Handbook of System Safety and Security*, chapter 3, pages 39–64. Elsevier, Cambridge, MA, USA, 1 edition.

Booch, G. (2004). *Object-Oriented Design and Analysis: With applications*. Pearson Education India, 2 edition.

Boström, P. (2011). Contract-based verification of Simulink models. In *Formal Methods and Software Engineering*, pages 291–306. Springer Berlin Heidelberg.

Boström, P., Morel, L., and Waldén, M. (2007). Stepwise development of Simulink models using the refinement calculus framework. In Jones, C. B., Liu, Z., and Woodcock, J., editors, *Theoretical Aspects of Computing – ICTAC 2007*, pages 79–93, Berlin, Heidelberg. Springer Berlin Heidelberg.

Broy, M. and Denert, E., editors (2002). *Software Pioneers: Contributions to Software Engineering*. Springer.

Broy, M., Kirstan, S., Krcmar, H., Schätz, B., and Zimmermann, J. (2014). *What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry?*, chapter 17, pages 310–334. Software Design and Development: Concepts, Methodologies, Tools, and Applications. IGI Global.

Canfora, G. and Cerulo, L. (2005). Impact analysis by mining software and change request repositories. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS)*, pages 20–29, Como, Italy. IEEE.

Carnegie Mellon University (2020). SEI CERT C coding standard.

Caspi, P., Curic, A., Maignan, A., Sofronis, C., Tripakis, S., and Niebert, P. (2003). From Simulink to SCADE/Lustre to TTA. *ACM SIGPLAN Notices*, 38(7):153–162.

Chapin, N., Hale, J. E., Khan, K. M., Ramil, J. F., and Tan, W.-G. (2001). Types of software evolution and software maintenance. *Journal of Software Maintenance and Evolution: Research and Practice*, 13(1):3–30.

Chawathe, S. S., Rajaraman, A., Garcia-Molina, H., and Widom, J. (1996). Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 493–504, Montreal. ACM Press.

Colaco, J.-L., Pagano, B., and Pouzet, M. (2017). SCADE 6: A formal language for embedded critical software development (invited paper). In *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE.

Dajsuren, Y., van den Brand, M. G. J., Serebrenik, A., and Roubtsov, S. (2013). Simulink models are also software: Modularity assessment. In *Proceedings of the 9th international ACM Sigsoft Conference on Quality of Software Architectures (QoSA)*, pages 99–106, Vancouver, Canada. ACM.

Dijkstra, E. W. (1972). *Structured Programming*, chapter Notes on Structured Programming, pages 1–82. New York: Academic Press.

Dijkstra, E. W. (1982). On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer New York.

Dörr, H. (2017). Good interfaces in large models. https://model-engineers.com/files/upload/academy/mgigroup/large_models.pdf. Modeling Guidelines Interest Group (MGIGroup) [Online; accessed Oct 2020].

Ebert, C., Cain, J., Antoniol, G., Counsell, S., and Laplante, P. (2016). Cyclomatic complexity. *IEEE Software*, 33(6):27–29.

Edwards, S. H. (1997). Representation inheritance: A safe form of "white box" code inheritance. *IEEE Transactions on Software Engineering*, 23(2):83–92.

Fenton, N. E. and Bieman, J. (2014). *Software Metrics: A Rigorous and Practical Approach*. CRC Press, 3 edition.

Gerlitz, T. and Kowalewski, S. (2016). Architectural analysis of MATLAB/Simulink models with artshop. In *Proceedings of the 2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 307–310. IEEE.

Gerlitz, T., Tran, Q. M., and Dziobek, C. (2015). Detection and handling of model smells for MATLAB/Simulink models. In *International Workshop on Modelling in Automotive Software Engineering (MASE)*, volume 1487, pages 13–22, Ottawa, Canada. CEUR.

Ghezzi, C., Jazayeri, M., and Mandrioli, D. (2002). *Fundamentals of Software Engineering*. Prentice-Hall, Upper Saddle River, NJ, USA, 2 edition.

Giger, E., Pinzger, M., and Gall, H. C. (2012). Can we predict types of code changes? an empirical analysis. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 217–226, Zurich, Switzerland. IEEE.

Gill, G. and Kemerer, C. (1991). Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12):1284–1288.

Haber, A., Kolassa, C., Manhart, P., Nazari, P. M. S., Rumpe, B., and Schaefer, I. (2013). First-class variability modeling in Matlab/Simulink. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 11–18, Pisa, Italy. ACM Press.

Han, J., Cheng, H., Xin, D., and Yan, X. (2007). Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86.

Hata, H., Mizuno, O., and Kikuno, T. (2010). Fault-prone module detection using large-scale text features based on spam filtering. *Empirical Software Engineering*, 15(2):147–165.

Hoare, C. A. R. (1971). Towards a theory of parallel programming. In Hoare, C. A. R. and Perrot, R. H., editors, *Operating Systems Techniques*, pages 61–71. Academic Press.

IEEE (2018). Information technology – programming languages – C. ISO/IEC 9899: 2018.

Information Is Beautiful (2015). Codebases: Millions of lines of code. https://www.informationisbeautiful.net/visualizations/million-lines-of-code/. [Online; accessed Oct 2020].

International Organization for Standardization (2010). Systems and software – vocabulary. ISO/IEC/IEEE 24765:2010.

International Organization for Standardization (2017). Systems and software – vocabulary. ISO/IEC/IEEE 24765:2017(E).

Iwu, F., Galloway, A., Toyn, I., and McDermid, J. (2004). Practical formal specification for embedded control systems. *IFAC Proceedings Volumes*, 37(4):165–170.

Jaskolka, M., Pantelic, V., Lawford, M., and Wassyng, A. (2021). Repository mining for changes in Simulink models.

Jaskolka, M., Pantelic, V., Wassyng, A., and Lawford, M. (2020a). A comparison of componentization constructs in Simulink. In *SAE Technical Paper*, number 2020-01-1290, pages 1–16. SAE International.

Jaskolka, M., Pantelic, V., Wassyng, A., and Lawford, M. (2020b). Supporting modularity in Simulink models. arXiv:2007.10120.

Jaskolka, M., Scott, S., Pantelic, V., Wassyng, A., and Lawford, M. (2020c). Applying modular decomposition in Simulink. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 31–36.

Kakade, R., Murugesan, M., Perugu, B., and Nair, M. (2010). Model-based development of automotive electronic climate control software. In Kühne, T., Selic, B., Gervais, M.-P., and Terrier, F., editors, *Proceedings of the 6th European Conference on Modelling Foundations and Applications (ECMFA)*, volume 6138, pages 144–155, Berlin, Heidelberg. Springer Berlin Heidelberg.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming*, pages 327–354. Springer Berlin Heidelberg.

Kim, S., Whitehead, E. J., and Zhang, Y. (2008). Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196.

Klauske, L. K. and Dziobek, C. (2010). Improving modeling usability: Automated layout generation for Simulink. In *Proceedings of the MathWorks Automotive Conference (MAC)*.

Kochan, S. (2014). *Programming in C*. Pearson Education (US).

Korson, T. D. and Vaishnavi, V. K. (1986). An empirical study of the effects of modularity on program modifiability. In Soloway, E. and Iyengar, S., editors, *Papers Presented at the First Workshop on Empirical Studies of Programmers*, pages 168–186, Norwood, NJ, USA. Ablex Publishing Corp.

Legros, E., Schäfer, W., Schürr, A., and Stürmer, I. (2010). MATE - a model analysis and transformation environment for MATLAB Simulink. In Giese, H., Karsai, G., Lee, E., Rumpe, B., and Schätz, B., editors, *Model-Based Engineering of Embedded Real-Time Systems*, pages 323–328. Springer Berlin Heidelberg.

Liebel, G., Marko, N., Tichy, M., Leitner, A., and Hansson, J. (2014). Assessing the state-of-practice of model-based engineering in the embedded systems domain. In Dingel, J., Schulte, W., Ramos, I., Abrahão, S., and Insfran, E., editors, *Model-Driven Engineering Languages and Systems*, volume 8767, pages 166–182. Springer International Publishing.

Liebel, G., Marko, N., Tichy, M., Leitner, A., and Hansson, J. (2018). Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*, 17(1):91–113.

Lublinerman, R. and Tripakis, S. (2008). Modularity vs. reusability: Code generation from synchronous block diagrams. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 1504–1509. ACM.

MacCormack, A., Rusnak, J., and Baldwin, C. Y. (2007). The impact of component modularity on design evolution: Evidence from the software industry. *Harvard Business School Technology & Operations Mgt. Unit Research Paper*, pages 1–36.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 1 edition.

McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320.

Middleton, P. and Sutton, J. (2005). *Lean Software Strategies: Proven Techniques for Managers and Developers*. CRC Press.

Model Engineering Solutions (2020). M-XRAY user guide. v4.3.

Molotnikov, Z., Schorp, K., Aravantinos, V., and Schätz, B. (2016). Future programming paradigms in the automotive industry. *FAT-Schriftenreihe*, 287:108.

National Instruments (2020). LabVIEW. http://www.ni.com/labview. Version 2020. [Online; accessed Oct 2020].

Neuhaus, S., Zimmermann, T., Holler, C., and Zeller, A. (2007). Predicting vulnerable software components. In *ACM Conference on Computer and Communications Security*, pages 529–540, Alexandria, Virginia, USA. ACM.

Object Management Group (2017). Unified Modeling Language (UML). https://www.omg.org/spec/UML/2.5.1. Version 2.5.1 [Online; Accessed Oct 2020].

Object Management Group (2019). Omg system modeling language (SysML). http://www.omg.org/spec/SysML/1.6. Version 1.6. [Online; accessed Oct 2020].

O'Hearn, P. W., Yang, H., and Reynolds, J. C. (2009). Separation and information hiding. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(3):1–48.

Olszewska, M. (2011). Simulink-specific design quality metrics. TUCS Technical Report 1002, Åbo Akademi University, Turku, Finland.

Olszewska, M., Dajsuren, Y., Altinger, H., Serebrenik, A., Waldén, M., and van den Brand, M. G. J. (2016). Tailoring complexity metrics for Simulink models. In *Proccedings of the 10th European Conference on Software Architecture Workshops - ECSAW '16*. ACM Press.

Oualline, S. (1997). *Practical C Programming*. Nutshell Handbooks. O'Reilly Media, Sebastopol, CA, USA, 3 edition.

Pantelic, V., Postma, S., Lawford, M., Jaskolka, M., Mackenzie, B., Korobkine, A., Bender, M., Ong, J., Marks, G., and Wassyng, A. (2018). Software engineering practices and Simulink: bridging the gap. *International Journal on Software Tools for Technology Transfer*, 20(1):95–117.

Pantelic, V., Schaap, A., Wassyng, A., Bandur, V., and Lawford, M. (2019). Something is rotten in the state of documenting Simulink models. In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development*, volume 1 of *MODELSWARD 2019*, pages 503–510. INSTICC, SciTePress.

Parnas, D. L. (1972a). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Parnas, D. L. (1972b). A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336.

Parnas, D. L. (2002). The secret history of information hiding. In *Software Pioneers*, pages 398–409. Springer, Berlin, Heidelberg.

Parnas, D. L. (2003). Structured programming: A minor part of software engineering. *Information Processing Letters*, 88(1-2):53–58.

Parnas, D. L. (2018). Software structures: A careful look. *IEEE Software*, 35(6):68–71.

Parnas, D. L., Clements, P. C., and Weiss, D. M. (1985). The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266.

Parnas, D. L., Clements, P. C., and Weiss, D. M. (1989). Enhancing reusability with information hiding. In Biggerstaff, T. J. and Perlis, A. J., editors, *Software Reusability: Concepts and Models*, volume 1, chapter 6, pages 141–157. ACM, New York, NY, USA.

Qian, K., Haring, D. D., and Cao, L. (2009). *Embedded Software Development with C*. Springer-Verlag GmbH.

Rau, A. (2001). On model-based development: Decomposition and data abstraction in SIMULINK. *Softwaretechnik-Trends*, 21(3):1–6.

Rau, A. (2002). On model-based development: A pattern for strong interfaces in SIMULINK. *Gesellschaft für Informatik, FG*, 2(1):12.

Reddy, R. and Ziegler, C. (2009). *C Programming for Scientists and Engineers with Applications*. Jones & Bartlett Learning.

Rising, L. S. and Calliss, F. W. (1994). An information-hiding metric. *Journal of Systems and Software*, 26(3):211–220.

Robbes, R., Pollet, D., and Lanza, M. (2008). Logical coupling based on fine-grained change information. In *Proceedings of the 15th Working Conference on Reverse Engineering*, pages 42–46, Antwerp, Belgium. IEEE.

Ryssel, U., Ploennigs, J., and Kabitzsch, K. (2010). Automatic variation-point identification in function-block-based models. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–32, Eindhoven, The Netherlands. ACM.

Rysselberghe, F. V. and Demeyer, S. (2004). Mining version control systems for FACs (frequently applied changes). In Hassan, A. E., Holt, R. C., and Mockus, A., editors, *Proceedings of the 1st International Workshop on Mining Software Repositories (MSR)*, pages 48–52, Edinburgh, UK.

Scandariato, R., Walden, J., Hovsepyan, A., and Joosen, W. (2014). Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006.

Schaap, A., Marks, G., Pantelic, V., Lawford, M., Selim, G., Wassyng, A., and Patcas, L. (2018). Documenting Simulink designs of embedded systems. In *Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS): Companion Proceedings*, pages 47–51, Copenhagen, Denmark. ACM.

Schach, S. R. (2010). *Object-Oriented and Classical Software Engineering*. McGraw-Hill, 8 edition.

Schlie, A., Wille, D., Schulze, S., Cleophas, L., and Schaefer, I. (2017). Detecting variability in MATLAB/Simulink models. In *Proceedings of the*

21st International Systems and Software Product Line Conference (SPLC), volume A, pages 215–224, Sevilla, Spain. ACM Press.

Schwinghammer, J., Yang, H., Birkedal, L., Pottier, F., and Reus, B. (2010). A semantic foundation for hidden state. In Ong, L., editor, *Foundations of Software Science and Computational Structures*, volume 6014, pages 2–17, Berlin, Heidelberg. Springer Berlin Heidelberg.

Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*, volume 21, pages 38–45. ACM.

Sommerville, I. (2015). *Software Engineering*. Addison-Wesley, 10 edition.

Spectrum, I. (2020). Interactive: The top programming languages. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020. [Online; accessed Oct 2020].

Srivastava, S., Hicks, M., Foster, J. S., and Jenkins, P. (2008). Modular information hiding and type-safe linking for C. *IEEE Transactions on Software Engineering*, 34(3):357–376.

Stephan, M., Alalfi, M. H., and Cordy, J. R. (2014). Towards a taxonomy for Simulink model mutations. In *Proceedings of the 7th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 206–215, Cleveland, OH, USA. IEEE.

Stephan, M., Alalfi, M. H., Cordy, J. R., and Stevenson, A. (2013). Evolution of model clones in Simulink. In *ME 2013 – Models and Evolution Workshop Proceedings*, pages 40–49. Citeseer.

Stevens, W. P., Myers, G. J., and Constantine, L. L. (1999). Structured design. *IBM Systems Journal*, 38(2.3):231–256.

Swanson, E. B. (1976). The dimensions of maintenance. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE)*, pages 492–497, San Francisco, CA, USA. IEEE Computer Society Press.

The MathWorks (2019). Simulink user's guide. https://www.mathworks.com/help/releases/R2019b/pdf_doc/simulink/sl_using.pdf. Version R2019b [Online; accessed Oct 2020].

The MathWorks (2020a). Company overview. https://www.mathworks.com/content/dam/mathworks/handout/2020-company-factsheet-8-5x11-8282v20.pdf. [Online; accessed Oct 2020].

The MathWorks (2020b). Embedded Coder user's guide. https://www.mathworks.com/help/releases/R2020b/pdf_doc/ecoder/ecoder_ug.pdf. Version R2020b [Online; accessed Oct 2020].

The MathWorks (2020c). Mathworks advisory board control algorithm modeling guidelines. https://www.mathworks.com/help/pdf_doc/simulink/simulink_mab_guidelines.pdf. Version 5.0. [Online; accessed Oct 2020].

The MathWorks (2020d). MATLAB. https://www.mathworks.com/products/matlab.html. [Online; accessed Oct 2020].

The MathWorks (2020e). Simulink. https://www.mathworks.com/products/simulink.html. [Online; accessed Oct 2020].

The MathWorks (2020f). Simulink Check reference. https://www.mathworks.com/help/releases/R2020b/pdf_doc/slcheck/slcheck_ref.pdf. Version 2020b [Online; accessed Oct 2020].

The MathWorks (2020g). Simulink Check user's guide. https://www.mathworks.com/help/releases/R2020b/pdf_doc/slcheck/slcheck_ug.pdf. Version R2020b [Online; accessed Oct 2020].

The MathWorks (2020h). Simulink Coder user's guide. https://www.mathworks.com/help/releases/R2020b/pdf_doc/rtw/rtw_ug.pdf. Version R2020b [Online; accessed Oct 2020].

The MathWorks (2020i). Simulink modeling guidelines for high-integrity systems. https://www.mathworks.com/help/releases/R2020a/pdf_doc/simulink/simulink_hi_guidelines.pdf. Version 2020b [Online; accessed Oct 2020].

The MathWorks (2020j). Simulink user's guide. https://www.mathworks.com/help/releases/R2020b/pdf_doc/simulink/simulink_ug.pdf. Version R2020b [Online; accessed Oct 2020].

The MathWorks (2020k). Stateflow. https://www.mathworks.com/products/stateflow.html. [Online; accessed Oct 2020].

The Modelica Association (2020). Modelica. https://www.modelica.org/modelicalanguage. Version 4.0.0. [Online; accessed Oct 2020].

The Motor Industry Software Reliability Association (2009). MISRA AC SLSF modelling design and style guidelines for the application of Simulink and Stateflow. Version 1.0.

Tran, Q. M., Wilmes, B., and Dziobek, C. (2013). Refactoring of Simulink diagrams via composition of transformation steps. In *International Conference on Software Engineering Advances*, pages 140–145. Citeseer.

Tripakis, S. and Lublinerman, R. (2018). Modular code generation from synchronous block diagrams: Interfaces, abstraction, compositionality. In *Lecture Notes in Computer Science*, pages 449–477. Springer International Publishing.

University of California at Berkeley (2020). Ptolemy II. https://ptolemy.berkeley.edu/ptolemyII/index.htm. Version 11.0. [Online; accessed Oct 2020].

Voas, J. M. (1996). Object-oriented software testability. In Bologna, S. and Bucci, G., editors, *Achieving Quality in Software*, pages 279–290. Springer US.

Watson, A. H. and McCabe, T. J. (1996). Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST Special Publication*, 500(235):1–114.

Whalen, M. W., Murugesan, A., Rayadurgam, S., and Heimdahl, M. P. E. (2014). Structuring Simulink models for verification and reuse. In *Proceedings of the 6th International Workshop on Modeling in Software Engineering (MiSE)*, pages 19–24, Hyderabad, India. ACM.

Xiao, Y. and Agbossou, K. (2009). Interface design and software development for PEM fuel cell modeling based on Matlab/Simulink environment. In *2009 WRI World Congress on Software Engineering*, volume 4, pages 318–322, Xiamen, China.

Yang, Y., Shen, D., Xie, Y., and Li, X. (2012). Matlab Simulink of COST231-WI model. *International Journal of Wireless & Microwave Technologies*, 3:1–8.

Yatish, S., Jiarpakdee, J., Thongtanunam, P., and Tantithamthavorn, C. (2019). Mining software defects: Should we consider affected releases? In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pages 654–665, Montreal, Quebec, Canada. IEEE Press.

Ying, A. T. T., Murphy, G. C., Ng, R., and Chu-Carroll, M. C. (2004). Predicting source code changes by mining change history. *IEEE Transactions on Software Engineering*, 30(9):574–586.

Ziegenbein, D., Saidi, S., Hu, X. S., and Steinhorst, S. (2020). Future automotive HW/SW platform design (Dagstuhl seminar 19502). *Dagstuhl Reports*, 9(12):28–66.

Zimmermann, T., Zeller, A., Weissgerber, P., and Diehl, S. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445.