High-level HVDC Control and Protection Functions using Sequence Based Model Driven Architecture

A Comparative Analysis

Master's thesis in Sustainable Electric Power Engineering and Electromobility

ARJUNA ARAVINDA SENERATH

DEPARTMENT OF ELECTRICAL ENGINEERING

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2024
www.chalmers.se

Master's thesis 2024

High-level HVDC Control and Protection Functions using Sequence Based Model Driven Architecture: A Comparative Analysis

A comparison between implementation of HVDC Control and Protection Function using HiDraw (Hitachi's proprietary modeling software) and Stateflow of MATLAB Simulink

ARJUNA ARAVINDA SENERATH

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering
Division of Electric Power Engineering
Chalmers University of Technology
Gothenburg, Sweden 2024

High-level HVDC Control and Protection Functions using Sequence Based Model Driven Architecture: A Comparative Analysis
ARJUNA ARAVINDA SENERATH

Supervisors: Arsalan Hadaeghi & Jose Ruiz Yuncosa, Hitachi Energy
Examiner: Massimo Bongiorno, Electric Power Engineering

Master's Thesis 2024
Department of Electrical Engineering
Division of Electric Power Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46 31 772 1000

Cover: Modern HVDC Converter Station: Advanced Control and Protection Systems in Action. Image courtesy Hitachi Energy.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2024

High-level HVDC Control and Protection Functions using Sequence Based Model Driven Architecture: A Comparative Analysis
ARJUNA ARAVINDA SENERATH
Department of Electrical Engineering
Chalmers University of Technology

Abstract

The effectiveness of High Voltage Direct Current (HVDC) transmission systems heavily relies on advanced control and protection mechanisms. This thesis investigates the comparative implementation of HVDC control and protection functions using sequence-based Model Driven Architecture (MDA) in two different modeling environments: Hitachi's proprietary HiDraw and MATLAB Stateflow. The primary objective is to assess the feasibility and efficiency of sequence-based MDA in enhancing HVDC system performance.

The research focuses on developing several models, initially simple, progressing to complex with nested states, in both MATLAB Stateflow and HiDraw. Each model is designed to represent typical HVDC operational states and transitions, incorporating a heavy computational function to simulate realistic control and protection tasks. The generated code from both environments is extracted and executed on identical hardware to measure execution efficiency and performance.

Preliminary results indicate that MATLAB Stateflow maintains consistent execution times despite increasing model complexity, whereas HiDraw shows a proportional increase in execution time. The findings suggest that while MATLAB Stateflow offers advantages in terms of efficiency and scalability, HiDraw's sequential execution approach ensures predictability, which is crucial for real-time operating systems. This study provides valuable insights into optimizing HVDC control and protection systems through sequence-based MDA, highlighting potential improvements and future work areas.

Keywords: HVDC, Model Driven Architecture, HiDraw, MATLAB Stateflow, control systems, protection systems, real-time systems.

Acknowledgements

I would like to take a moment to express my sincere gratitude to all those who have supported me throughout the journey of completing my master's thesis. This significant milestone would not have been possible without their unwavering guidance and encouragement.

First and foremost, I would like to thank my two advisors, Arsalan Hadaeghi and Jose Ruiz Yuncosa, for their invaluable expertise, insightful suggestions, and constant support. Their guidance was instrumental in shaping the direction and outcome of this research. I am deeply grateful for the time and effort they invested in my work.

I extend my deep appreciation to my manager, Aditya Deb, for granting me the opportunity to conduct my master's thesis at Hitachi Energy and providing invaluable resources and unwavering support throughout the process.

I also extend my heartfelt thanks to my examiner, Professor Massimo Bongiorno, for agreeing to be my master's thesis examiner and for the immeasurable support provided during this journey.

Furthermore, I must thank my ever-loving wife for her undying support throughout my master's studies in Sweden. Without her, I would not have been able to complete my degree and thesis.

Additionally, I am grateful to all my family and friends for their encouragement during my master's studies and thesis journey. Their support provided comfort during stressful times.

Arjuna Aravinda Senerath, Gothenburg, June 2024

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

CB	Circuit Breaker
CSV	Comma-Separated Values
GPS	Global Positioning System
HVAC	High-voltage alternating current
HVDC	High-voltage direct current
I/O	Input/Output
IEEE	Institute of Electrical and Electronics Engineers
LCC	Line Commutated Converters
MACH™	Modular Advanced Control for HVDC
MDA	Model Driven Architecture
PC	Personal Computer
RAM	Random Access Memory
RTOS	Real-time Operating System
VSC	Voltage Sourced Converters

Contents

List of Figures

List of Tables

1
Introduction

In High Voltage Direct Current (HVDC) applications, the effectiveness of the transmission system heavily relies on the control and protection system. Hitachi Energy employs state-of-the-art technology derived from the realms of electronics and microprocessors for its control and protection system, ensuring advanced performance. Recent years have witnessed swift advancements in control and protection equipment for power system applications. This progress is primarily attributed to overall electronic development and the integration of novel application programming methods.
At Hitachi Energy, HiDraw is a proprietary tool, which is a fully graphical block programming language designed for operation on networked industrial PCs for control and protection system of HVDC stations. This intuitive tool operates on a simple point-and-click interface, allowing users to effortlessly combine and connect symbols (blocks) sourced from a symbol library to create a schematic. Subsequently, HiDraw generates code from the schematic, presenting it in either a high-level language (such as ANSI standard C, PL/M, or FORTRAN) or assembly language.[1]
HiDraw is a programming tool that, unlike Simulink, executes all generated code sequentially, even for states that are not currently active. In Simulink, while the required code for all states is generated, only the relevant code for the active state is executed during simulation or execution. These states can be considered as energized, de-energized etc. in HVDC transmission systems. Therefore, state sequential modeling approach might prove to be more efficient in executing certain control and protection system functions. This thesis seeks to investigate the feasibility of adopting sequence based Model-Driven Architecture (MDA) and assess the extent to which it can be applied to enhance the performance of Hitachi Energy's HVDC control and protection systems.

1.1 Project Objectives

The thesis compares the implementation of sequence-based Model-Driven Architecture (MDA), as used in MATLAB Stateflow, to traditional sequential execution, as seen in MATLAB Simulink and HiDraw, in model building. The study focuses on MATLAB Simulink, MATLAB Stateflow, and HiDraw within Hitachi Energy's HVDC control and protection systems to examine the benefits and downsides of MDA versus traditional sequential execution. Currently, the control and protection schematics generated in HiDraw are sequential in nature. The same can be more simply comprehended if the system was organized using models with hierarchical

symbols or charts that indicated the system's status. The goal of this thesis is to investigate the feasibility of adapting MATLAB Stateflow's Hierarchical chart object structure and code generation methodologies to the HiDraw programming environment. Therefore, a comparative examination of the created code's efficiency and performance, taking into account the complexities and whether it is appropriate for the intended application, which in this case is a real-time operating system.
Initial plan was to explore the processing times, memory utilization, and code generation efficiency in the C programming language. However, with progress it was found that the code generated from the HiDraw program was in C++ programming language which led to changing the code comparison to C++ coding in MATLAB Stateflow as well. The goal is to provide concise insights and recommendations for enhancing the design and implementation of HVDC systems through sequence based MDA.
The limitations of this project is constrained to the comparative analysis of models developed exclusively using MATLAB Simulink Stateflow and HiDraw (Hitachi's proprietary modeling software). Additionally, the focus of code generation will be confined to the C++ programming language for the models created within the aforementioned platforms.

1.2 Specifications

The thesis involve a comprehensive examination of the modeling tools MATLAB Simulink, MATLAB Stateflow, and HiDraw, with a specific emphasis on their application in the context of Hitachi Energy's HVDC control and protection systems. The investigation will delve into the intricacies of model development within each tool, analyzing their unique features, capabilities, and methodologies.
Furthermore, the thesis will scrutinize the generated code resulting from models constructed using these tools, specifically targeting the C++ programming language. The aim is to evaluate the efficiency, and performance of the generated code, considering the intricacies of the HVDC control and protection systems. Through this investigation, the thesis endeavors to provide valuable insights into the feasibility and potential benefits of adopting sequence based Model-Driven Architecture (MDA) for optimizing the design and implementation of such systems.

1.3 Report Outline

The current chapter, chapter 1, includes the introduction to the project and its objectives. Chapter 2 delves into the background of the project, which includes and overview of HVDC Control and Protection, Hitachi's MACH™ Control and Protection System and the programming environments (HiDraw and Simulink). Chapter 3 describes the methods implemented to achieve the project requirements in detail. Chapter 4 provides insight into the discussion of results achieved for the developed methods. The final chapter, chapter 5 gives the conclusive outcome of the thesis work.

2
Background

This chapter discusses the related background and information as part of the literature review pertaining to the development and need of the master thesis. The chapter discussion will be initiated with a focus on HVDC system, the converter station and its configurations. Afterwards, the control and protection systems in general as well as control platform implemented by Hitachi Energy will also be discussed. Additionally, a brief introduction of real time and non-real time systems is also discussed. Finally, the implementation in MATLAB Simulink which can be used to model similar systems as the programming environment used at Hitachi Energy will also be described.

2.1 High Voltage Direct Current (HVDC)

Electricity has now become a basic necessity in our modern lives. The power transmission system plays a crucial role in providing electricity to the end customer by connecting power generation companies to communities. As society grows and embraces renewable energy, the electric power transmission landscape needs to evolve and adapt new technologies to cater the demand of the growing society.
The primary medium for power transmission still remains to be High Voltage Alternate Current (HVAC), but HVDC has become economically viable for various applications such as Long distance overland overhead line transmission (above 800 km), Long submarine cable crossings (up to 80 km) and Interconnections between asynchronous networks. The development of the high-voltage mercury arc valve facilitated HVDC integration into AC networks[4]. As interconnected systems expand, the technical and economic benefits can diminish due to challenges such as load flow issues, power oscillations, and declining voltage quality. To enhance grid flexibility and performance, two complementary technologies have emerged: HVDC and FACTS (Flexible AC Transmission Systems). HVDC has been utilized for over 50 years and is now experiencing renewed interest due to reduced power electronic costs and improvements in power electronic control.
There are two main technologies used in HVDC technologies at present, using either Voltage Sourced Converters (VSC) or Line Commutated Converters (LCC). LCC being the more mature technology is more commonly seen in HVDC projects. However, given the continuous development in VSC technology, VSC is becoming the preferred technology for the interconnection of isolated grids to the power system, such as offshore wind plants and remotely located concentrated solar power plants. Its main characteristic is the continuous and independent control of active

and reactive power.[4]

2.1.1 Line Commutated Converters (LCC)

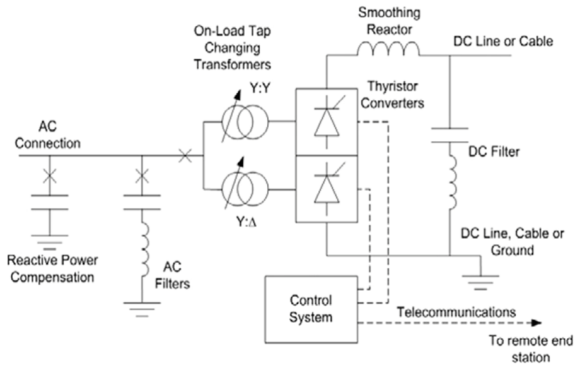


Figure 2.1: Components of a LCC-HVDC system[4]

Figure 2.1 depicts the setup of a LCC-HVDC system. The LCC-HVDC system comprises several components each serving specific functions. The core is represented by the LCC converters, driven by thyristor valves acting as the system's main component. Transformers link the AC network to valve bridges, adjusting voltage levels. AC filters are used to minimize the impact of current harmonics on the AC network. DC filters are used to mitigate voltage ripple and interference near the DC line which is essential for overhead transmission systems. The control system consists of two converters: one managing the DC voltage and the other controlling the DC current, together yielding the desired combination of voltage and current. Reactive power compensation enhances the system stability and optimize power availability.

2.1.2 Voltage Sourced Converters (VCC)

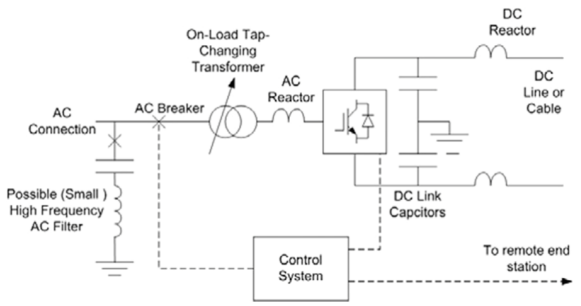


Figure 2.2: Components of a VSC-HVDC system[4]

Figure 2.2 depicts the setup of a VSC-HVDC system. Similar to the LCC-HVDC system, VSC-HVDC system also comprises of several components with the core being represented by the VSC converters driven by IGBT valves. Transformers adjust AC voltage to match DC levels, while phase reactors regulate currents between active and reactive power, reducing high-frequency harmonics. AC filters serve the

same function as in LCC-HVDC systems but with a smaller footprint. DC capacitors reduce voltage ripple on the DC side and provide energy storage which is crucial for power flow control. The control system, with a faster vector controller, independently manages the active and reactive power.

2.2 HVDC Converter Station

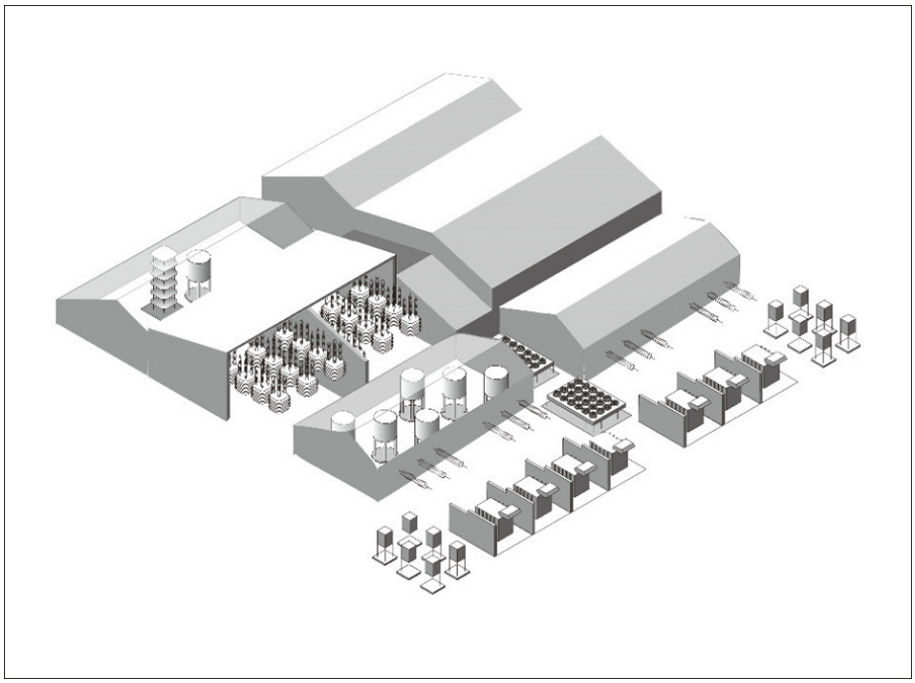


Figure 2.3: Illustration of a typical HVDC bipole Installation[3]

An HVDC station comprises numerous intricate components that require precise control to ensure optimal system performance. A commonly employed setup is the bi-pole configuration (see Figure 2.3), consisting of two poles, each with positive and negative polarity. These poles feature converter valves, phase reactors, indoor power transformers, and various high-voltage electrical devices like breakers, disconnectors, and current transformers. The converter valves necessitate a cooling system using deionized water, alongside an extensive auxiliary power system for station operation. Control of HVDC converters involves providing signals to operate semiconductor converter valves, which react within microseconds, making HVDC converters uniquely controllable devices in power systems. A well-designed control system can leverage these fast power changes to stabilize connected AC power systems by damping power oscillations, supporting sudden power generation loss, and balancing voltage changes using reactive power control.
Given the criticality of control system reliability, Hitachi Energy introduced a 100 percent digitized control system with rapid switchover to a hot standby system, ensuring uninterrupted power transfer since 1982. This configuration is standard across Hitachi Energy HVDC installations. Consequently, protection systems must also operate swiftly, with reaction times below 1 millisecond to prevent damage to

converter valves. The Modular Advanced Control for HVDC (MACH™) control system serves as the foundation for a fully duplicated centralized protection system, connected to measurement devices via fast optical process buses.
The MACH™ system's high performance enables integration of additional functions, such as fully duplicated and integrated transient fault recorders, generating easily readable records in standard IEEE common format for transient data exchange. Furthermore, high-speed GPS clock-synchronized analog inputs facilitate integration of line and cable fault locators utilizing traveling waves technology, accurately determining fault locations in the event of DC line or cable faults.[3]

2.2.1 Operation States of a Converter Station

The fundamental operating sequences of HVDC grids describe the transitions between various operating states of an HVDC grid, subsystem, or installation. These sequences pertain to individual devices (e.g., switches), units (e.g., AC/DC converter units), or installations. Typical sequences include: preparing to energize (from "NOT ready" to "ready to energize"), energizing (from "ready to energize" to "energized"), adapting DC voltages (from "energized" to "ready to connect"), connecting (from "ready to connect" to "no-load operation"), loading (from "no-load operation" to "on-load operation"), de-loading (from "on-load operation" to "no-load operation"), disconnecting (from "no-load operation" to "ready to connect"), stopping adaptation of DC voltages (from "ready to connect" to "energized"), de-energizing (from "energized" to "ready to energize"), shutting down (from "ready to energize" to "NOT ready"), and emergency shutting down (from any state to "NOT ready"). An illustration of these operating states[13] and sequences is given in Figure 2.4.

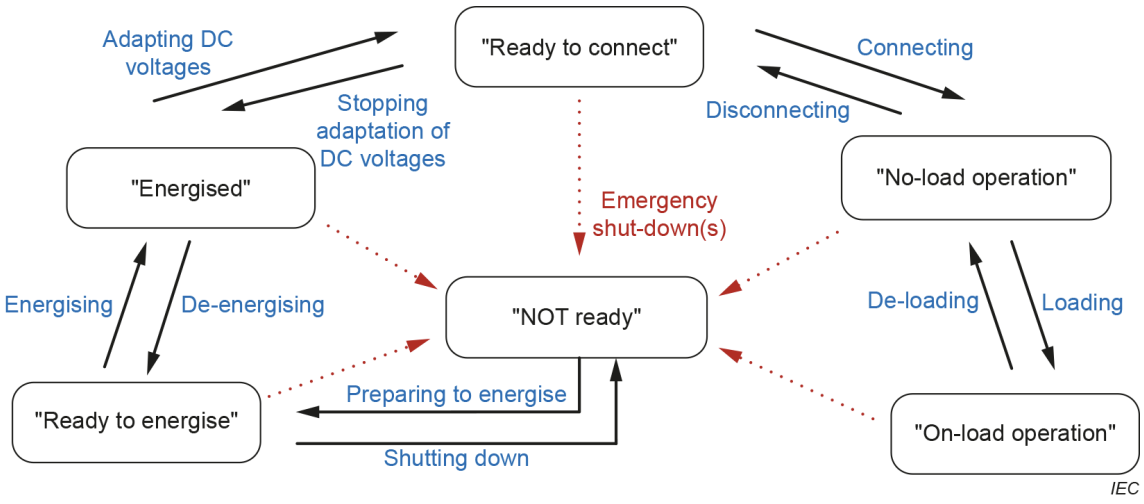


Figure 2.4: Typical operating sequences for transitions in operating states of a converter station[13]

These definitions differ from those provided by the IEC (2023) by introducing "Ready to connect" as an interim status during the transition from LIVE to COUPLED. Furthermore, "No-load operation" and "On-load operation" are not recognized as separate unit connection states. The IEC (2023) definitions are viewed as too specific

for converter units with de-rated gates, which limits their applicability in a broader architectural context. The distinction between "NOT ready" and "Ready to energize" is retained due to its importance for health and safety in high voltage environments. A gate in this context is a point of interconnection between elements or units within the HVDC grid, crucial for managing the flow of electricity between different parts of the system.[12].

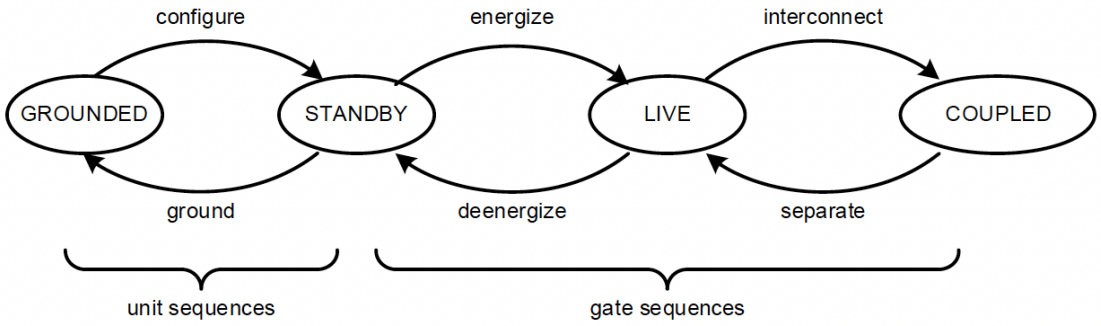


Figure 2.5: Unit States[12]

Each unit within a bipolar network can exist in one of four possible states as depicted in Figure 2.5. A transition between two states is referred to as a "unit sequence" when the actions involved are confined to procedures within the unit. When the transition involves changing the status of a gate, it is termed a "gate sequence". The status of an interconnecting gate can be either "CLOSED" or "OPEN".

2.2.1.1 Unit Connection Status GROUNDED ("NOT ready" [13])

In this state, the unit is isolated from all neighboring units, and all relevant grounding switches within the unit are closed. The unit is available for maintenance.

2.2.1.2 Unit Connection Status STANDBY ("Ready to energize" [13])

The unit stays isolated from neighboring units, but all grounding measures have been removed. For an aggregated unit—a group of units that must change status together—the necessary interconnecting gates have been closed. This status does not include temporary devices like peak current suppression devices. In this state, the unit is fully discharged with no voltage present. Special care may be needed for units with multi-level converters to ensure they are properly discharged.

2.2.1.3 Unit Connection Status LIVE

The defining characteristic of this status is the presence of operational voltage in the unit, with only one gate interconnecting to a LIVE (or COUPLED) neighboring unit closed. Temporary devices have been removed. In this status, active units such as converters can be operational.

2.2.1.4 Unit Connection Status COUPLED

In this status, at least two gates of the unit are closed (i.e., the unit can transfer power from one gate to another). Consequently, this status may not apply to a single gate unit.

2.2.2 MACH™ Control Platform

One of the most desired features in modern HVDC control systems is controllability. Hitachi Energy has designed the MACH™ control system to be capable of controlling various types of generations for HVDC systems, whether they were supplied by Hitachi Energy or other HVDC vendors. This advancement allows Hitachi Energy's customers to upgrade at least one control and protection system of an HVDC installation during its expected lifetime, providing access to the latest advancements in control and protection technology.

2.2.2.1 Building Blocks of MACH™

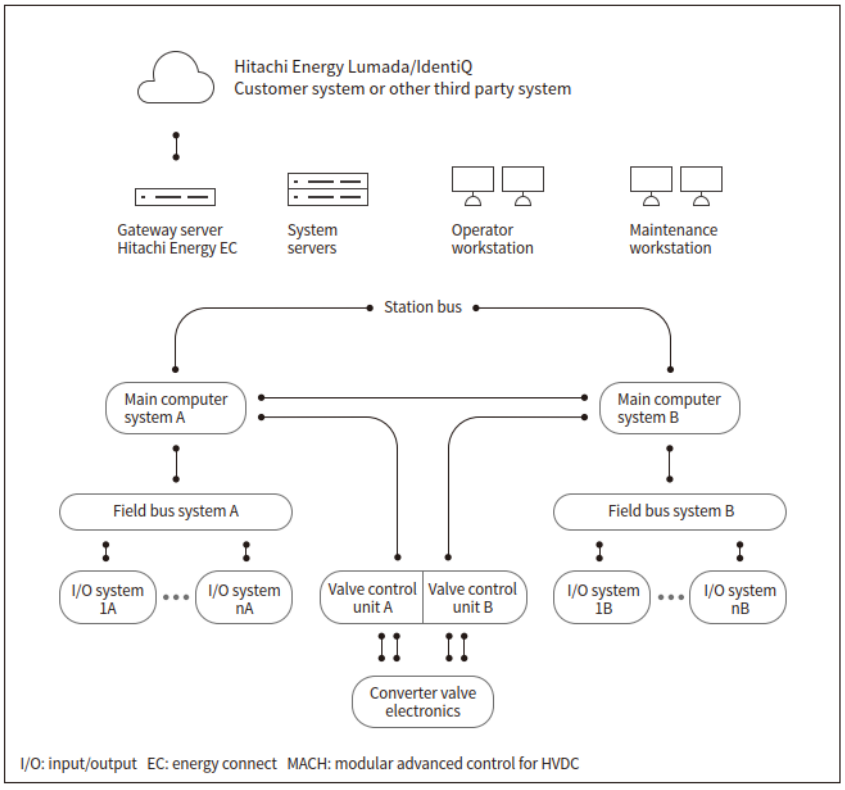


Figure 2.6: Hierarchical overview MACH™ system[3]

The MACH™ control and protection system can be divided into several different building blocks as shown in Figure 2.6. The control and protection loops consist of four distinct building blocks that operate within a range of tens of microseconds to milliseconds. These redundant systems include the main computer system, the main controller that receives input from the input/output (I/O) system via various process

busses, the converter valve electronics that directly connect to the semiconductors in each position of the controller valve, and the valve control unit, which serves as an interface between the hundreds or thousands of positions in the converter valves and the main control system.
The control and protection loops to operate at such levels of high response, measurements are needed to be taken periodically with high precision. The use of Real Time Operating System (RTOS) guarantees the the precision required for a specific task and its execution within the desired time frame. Hitachi Energy uses INtime for Windows as the real time operating system[2] in the main computer.

2.2.3 Real time operating System

Real-time operating systems are designed with time as a critical parameter, often featuring hard deadlines that must be met for proper functioning. There are two main types of RTOS:

1. Hard real-time systems
    - These systems must meet strict deadlines.
    - Common in industrial process control, avionics, and military applications.
    - Must provide absolute guarantees that a certain action will occur by a specific time.
2. Soft real-time systems
    - In soft real-time systems, occasional missed deadlines are acceptable.
    - They are commonly used in digital audio, multimedia, and digital telephones.
    - For instance, streaming media playback is a soft real-time task.

In real-time systems, meeting strict deadlines is paramount, sometimes leading to tightly coupled architectures where the operating system is directly linked with application programs. Therefore, MACH™ system uses RTOS to ensure high precision and response.[5]

2.3 HiDraw Programming Environment

Hitachi Energy has developed a graphical programming language called HiDraw, used for programming all application codes. This language enables design, real-time debugging, and simulations of complete converter systems before they are connected to power systems models. HiDraw has been the programming language for MACH™ for many decades, facilitating quick upgrades of older installed systems. Control loops and protection schemes can be directly re-used and integrated with the latest base designs. This graphical tools allows the entire range of hardware used by Hitachi Energy to be programmed using a single platform. HiDraw can not only generate source code, but also compile the generated source code to an executable file that can run on the target system.
A HiDraw user has a certain number of procedures to follow to create the desired program from HiDraw. Each application created in HiDraw consists of multiple schematics called *drawings*. The *drawings* are created using symbol blocks and interconnecting the symbols with lines. Lines can be interpreted as variables in a

programming language and will always have a name assigned to it within HiDraw. This name can be user defined or automatically defined as the user creates the drawing graphically. The relevant data type also has to be defined within the program (such as Boolean, integer, float, etc.) by the user. Figure 2.7 shows an example of a symbol block representing a SR Latch with signal lines connecting external signals. **CLOSE_CMD** & **OPEN_CMD** are external input signals to the latch while **SWITCH_STATUS** is an output signal which is used in the *drawings* with page number 12 and 14.

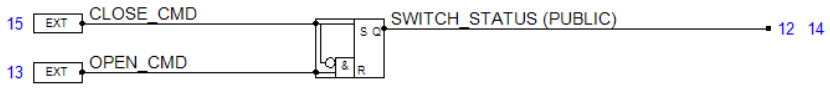


Figure 2.7: Symbol blocks and lines used in HiDraw

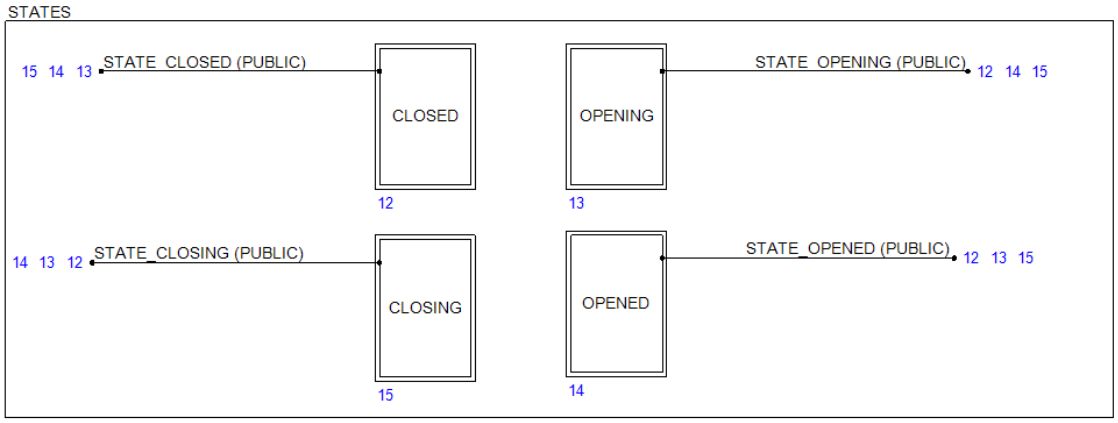


Figure 2.8: Hierarchical symbol in HiDraw

There are several types of *drawings* specified within HiDraw. The main types of *drawings* we aim to look at in this thesis would be *Task* and *Hierarchical* drawings. These would be explained in the following chapter in more detail. Figure 2.8 shows four Hierarchical symbols used within a drawing of type *task* and the symbol will be linked to a drawing of type *Hierarchical* which is a separate *drawing*.

2.4 MATLAB Stateflow®

Flow charts, truth tables, state transition tables, and state transition diagrams are all included in the visual language that Stateflow provides. It lets you specify how Simulink models and MATLAB algorithms react to events, time-based conditions, and input signals.
One may design supervisory control, job scheduling, fault management, user interfaces, communication protocols, and hybrid systems using Stateflow. Stateflow allows you to model both sequential and combinational decision logic, and it can be run in MATLAB or simulated in a Simulink model. Combinatorial logic's output

is only dependent on its current inputs. Sequential logic is not only dependent on its current inputs but also, it's previous inputs (thus internal "state"). A sequential component is not called a "gate", rather a "latch", "flip flop", or "register". One may instantly examine and troubleshoot your reasoning with the aid of the graphical animation tool. Furthermore, verification are performed at the editing and runtime stages to guarantee consistency and completeness of the design.

2.4.1 Finite State Machine

A finite state machine is a mathematical model of computation describing an abstract machine. The finite state machine will have the following characteristics.

- Finite number of states
- Only one active state at a given time
- Has a special state called start state (default transition state in Stateflow)
- Transition to another state is activated in response to some external input or when a transitional condition is satisfied
- The transition itself may involve executing a series of actions during the process

Finite state machines can be further described into two categories. One is called deterministic finite state machines, where for every state and input there is at most one transition to another state. The other is called nondeterministic finite state machines, where for some states and a given input/condition there are possible transition to multiple states.[6]

2.4.2 Stateflow® Charts

A graphical depiction of a finite state machine made up of states, transitions, and data is called a Stateflow® Charts chart. To specify how a system responds to events, time-based conditions, and external input signals, you may make a stateflow chart.

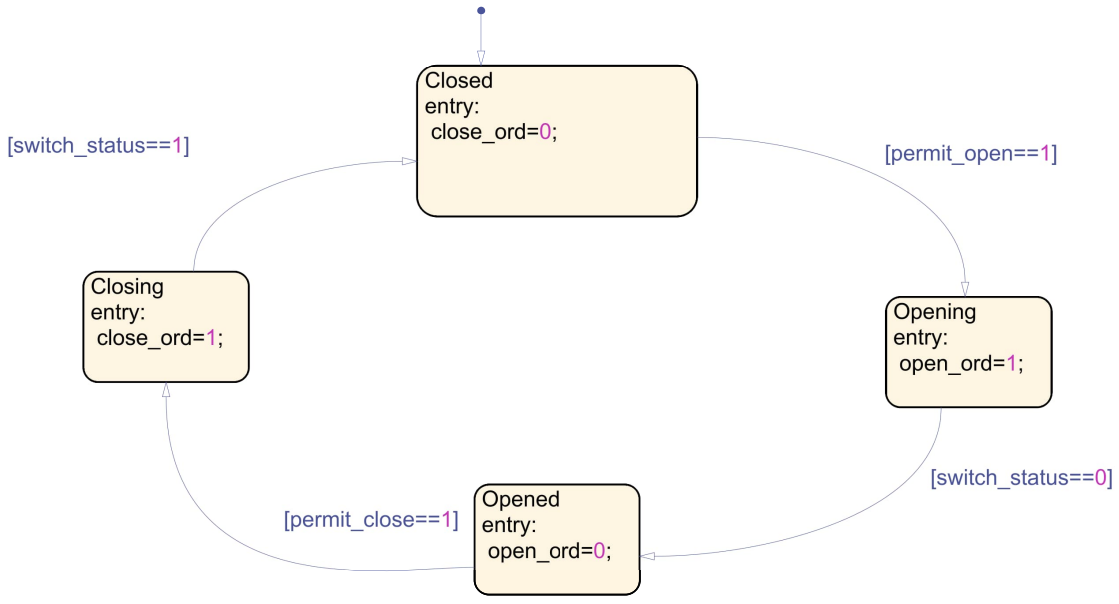


Figure 2.9: Example of a Stateflow chart

For example, the stateflow chart depicted in Figure 2.7 presents the underlying logic of a simplified operation of a circuit breaker. The chart contains four states labeled as *Closed*, *Opening*, *Opened*, and *Closing*. The arrow with a blue dot on the *Closed* state represents the default or starting state during initialization. In the *Closed* state, the output signal `close_ord` is set to zero. When the input signal `permit_open` equals 1, a transition occurs, and the active state transitions to the *Opening* state. Then, the commands inside the *Opening* state are executed. The process continues transitioning depending on the received inputs.

2.4.2.1 Data Scope in Stateflow Charts

In Stateflow, the terms 'inputs', 'outputs', and 'local variables' that are utilized in a chart are collectively known as 'chart data'. It's essential that all chart data have a defined scope. Table 2.1 lists the several common types of data scopes which are used in Stateflow charts.

Symbol	Name	Description
	Local Data	Refers to data that is exclusively used within the chart
	Input	A scope where a signal is received from Simulink through an input port
	Output	A scope in which a signal is sent to Simulink via an output port
	Parameter	A scope where a constant value is fetched from the MATLAB workspace or a Simulink mask parameter

Table 2.1: Common types of data scopes

2.4.2.2 Chart Actions

Chart actions are code executions that occur during chart simulation. There are two types of chart actions: **State Actions** and **Condition Actions**.

2.4.2.3 State Actions

State actions define what Stateflow does when a particular state is active. These actions occur at specific points during the state's life-cycle: when the state is entered, while the state is active, and when the state is exited. The three most common state actions are:

- **entry:** Executes code when entering a state.
- **during:** Executes code while remaining in a state.
- **exit:**Executes code when leaving a state.

For example, if you have code under the entry keyword, it will run every time the state is entered. Similarly, code under the exit keyword will execute when the state is exited. This structure ensures that specific operations are performed during these transitions.

2.4.2.4 Condition Actions

Condition actions define what Stateflow does when a transition condition is true. These actions are executed as soon as the transition condition evaluates to true and are specified within curly braces (`{code written here}`) to distinguish them from the condition, which is enclosed in square brackets (`[code written here]`).

2.4.2.5 Action Language

By default, Stateflow actions use the MATLAB language, where each line of code ends with a semicolon to suppress output in the Diagnostic Viewer. If you remove the semicolon, the output of each command will be printed. Stateflow also supports C as an action language, allowing you to choose the most suitable language for your needs.
This structured approach to specifying actions ensures that the Stateflow chart behaves predictably and that specific actions are executed at the correct times during simulation.

2.4.2.6 Chart Hierarchy

You may reduce redundancy by grouping states with comparable features or functions in your Stateflow chart using hierarchy. Complex Stateflow diagrams may be created by utilizing the Stateflow hierarchy of objects and defining a parent and child object containment structure. Generally speaking, a hierarchical design produces orderly, manageable diagrams with few transitions. Stateflow allows for a hierarchical configuration of states and charts. It is possible to have charts inside charts. A chart that is displayed inside another chart is called a sub-chart. In the same way, states can exist inside other states. Stateflow represents a hierarchy of states with superstates and substates.

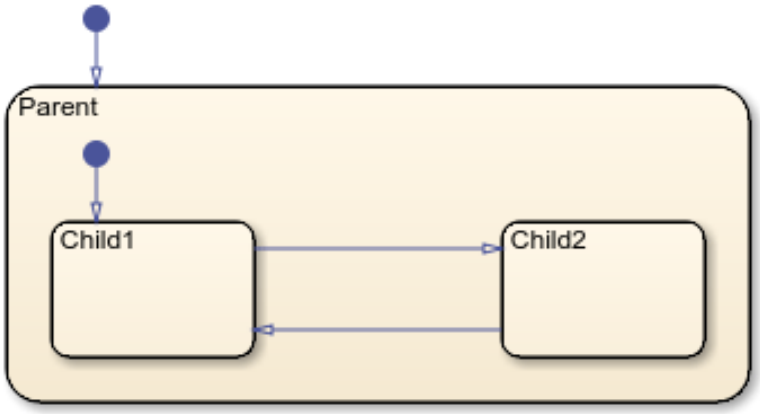


Figure 2.10: Stateflow chart hierarchy[7]

Figure 2.10 represents a simple hierarchical chart in Stateflow. There are three states shown in the figure. Namely *Parent*, *Child1* and *Child2*. The outer state acts as the parent or superstate of the inner states. These internal states are known as child states or substates of the outer state. Think of the contents within the

superstate as resembling a smaller chart. When the superstate is activated, one of the substates also becomes active. When the superstate becomes inactive, all of the substates also become inactive. There can also exists transitional conditions within the substates which are also shown by arrows depicted in the same figure.
Furthermore, state actions can be performed for the superstate as well as substates. These actions will execute as stated in section 2.4.2.3 during chart execution. For example, if there is a *during:* action stated in the superstate, this would execute regardless of which substate is active.

3
Methods

This chapter consists of a brief description of the thesis requirements by the company and the method development to achieve the requirements. The methods will be further described intricately using the various models developed. The generated code in C++ from these models and supporting code written to measure the efficiency will also be described in this chapter.

3.1 Requirements and Method Development

3.1.1 Company Requirement

The requirements of this thesis are to evaluate the features of both MATLAB Stateflow and HiDraw to determine the extent to which Stateflow's features can be implemented in HiDraw. The procedure will involve creating several cases in Stateflow, each with a fixed number of states, and then recreating the same models in HiDraw using existing symbols. Thereafter, code will be generated (in the same language) from each model within the two environments separately. The generated code will then be extracted from the local environments and executed on a single piece of hardware to compare code execution efficiency, i.e., the time required to run the model (time to run one iteration or step in the model). This approach ensures that the code execution time can be compared equally since both codes will be running on the same hardware.
The code generated from HiDraw will have a specific target environment, but a method will need to be developed to run it on the common hardware. Once the timings are compared, the results will be analyzed to assess the advantages and drawbacks of the two systems, depending on the requirements at Hitachi Energy.

3.1.2 Method Development

The thesis involves developing several models initially in MATLAB and subsequently recreating them in HiDraw using existing symbols. The process was carried out step-by-step, starting with familiarization with the MATLAB Stateflow and HiDraw programming environments. The first project involved modeling the operating logic of a circuit breaker with just four states in MATLAB Stateflow. This model was then recreated in HiDraw using existing SR latches, logic gates, and other supporting symbols for signal references.
Next, code was generated from MATLAB using the "Embedded Coder" app, a built-in tool for MATLAB Simulink. For HiDraw, its compiler was used to produce error-

free C++ code. The generated code from both programs was extracted and run on "Visual Studio Code" on the same computer to compare their execution times. After analyzing the timing results, a more complex project was undertaken: modeling the basic operation logic of an HVDC Converter station. This model was also simple, with just four states, but did not rely on any external inputs from Simulink. The same steps were followed in HiDraw, and the execution time was measured again. Following this, the implementation of nested states was explored. The Converter station model was expanded to include substates within one of the states. Four substates were incorporated into one of the existing states, and the same process was followed for the execution of the model. Subsequently, this expanded model was recreated in HiDraw, the code was extracted, and the timing was measured. This version of the model had four substates and three main states, totaling seven states during execution. For the final model, four substates were included in each of the four main states, resulting in a total of 16 states. Detailed descriptions of these models will follow in the report.

3.2 Circuit Breaker Model

The first step of the method development required a simple model to understand the basics of MATLAB Stateflow and implement the same in HiDraw. Therefore, simplified circuit breaker states were modeled in MATLAB and then replicated in HiDraw.

3.2.1 Implementation in MATLAB

Initially, as a learning objective, the operation of a simplified circuit breaker was modelled in MATLAB Simulink as shown in Figure 3.1. This consisted of a Stateflow chart with four states, 3 inputs and 2 outputs.

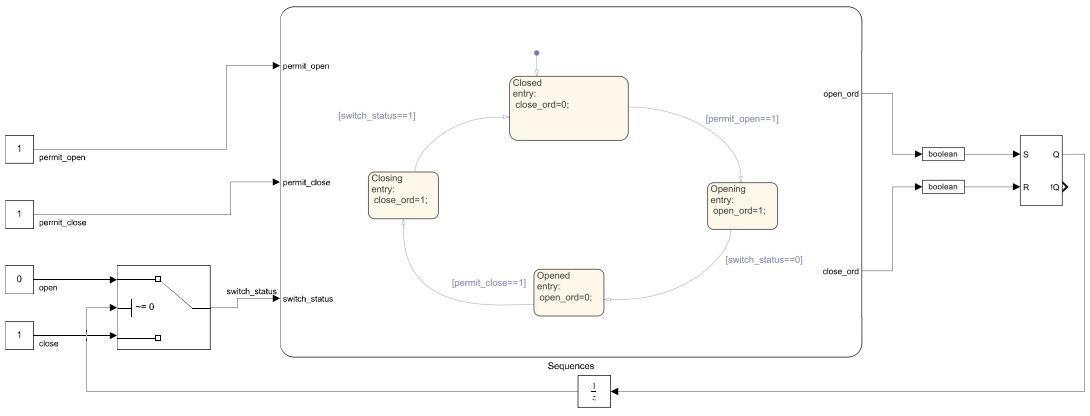


Figure 3.1: Circuit breaker model implemented in MATLAB Simulink

The model consisted of a Stateflow chart representing four states of the circuit breaker during operation: `Closed`, `Opening`, `Opened`, and `Closing`. The default

transition is set to the `Closed` state, and the transition conditions are indicated by the arrows connecting the states. The states transition from `Closed` to `Opening`, from `Opening` to `Opened`, from `Opened` to `Closing`, and from `Closing` to `Closed`, depending on the active state and whether the relevant transition conditions are met. Additionally, there are three inputs and two outputs to the Stateflow chart. The inputs are `permit_open`, `permit_close`, and `switch_status`. The two outputs are `close_ord` and `open_ord`.
The initial plan was to change the state for each step of the simulation. To achieve this, the inputs `permit_open` and `permit_close` are set as constants with a value of 1. Since these permit signals are always true, the states will keep transitioning depending on whether the circuit breaker is closed or opened. The other input to the Stateflow chart, `switch_status`, is fed through a built-in Simulink block called "switch," where the input value is determined by the circuit breaker condition. The input `switch_status` receives 1 when the circuit breaker is closed and 0 when the circuit breaker is open.
The outputs of the Stateflow chart are `close_ord` and `open_ord`, which simulate the commands given to the circuit breaker to open and close from the controller (in this case, the Stateflow chart). These commands are fed through an S-R flip flop, which holds the command to the switch block (representing the status of the breaker), causing the corresponding condition of the circuit breaker to change (open or close).
Thus, the model will keep changing states at each step and continue to run until the end of the simulation time.

3.2.2 Implementation in HiDraw

Implementing the circuit breaker model in HiDraw posed a somewhat difficult challenge due to the environment and symbols used in HiDraw being very different from those in other software. This required extensive learning time and consumed most of the time getting acquainted with the programming nature of the software. However, after overcoming the initial setup process of the target environments, a model similar to the circuit breaker was implemented.
The hierarchical symbols shown in Figure 2.8 were used to model each of the states. This symbol would create a separate drawing for each state and execute it. This approach is different from the environment in Simulink, where all the state information is contained within a single Stateflow chart. This posed a challenge in referencing external signals and debugging, which will be discussed in more detail in the results chapter.
The state logic implementation was also graphical in nature and required the use of gates and latches to achieve the desired functionality. The aim was to use existing symbols within HiDraw to recreate the model implemented in MATLAB Stateflow. Therefore, four different drawings, one for each state, were created. These used SR (Set-Reset) latches and logic gates to achieve the task. Figure 3.2 shows the logic implemented in HiDraw using the latches and logic gates. The figure does not show the actual symbols used within HiDraw due to proprietary restrictions within Hitachi Energy. However, the outline of the logic implemented in HiDraw is an

accurate representation of the `Closed` state as depicted in Figure 3.2.

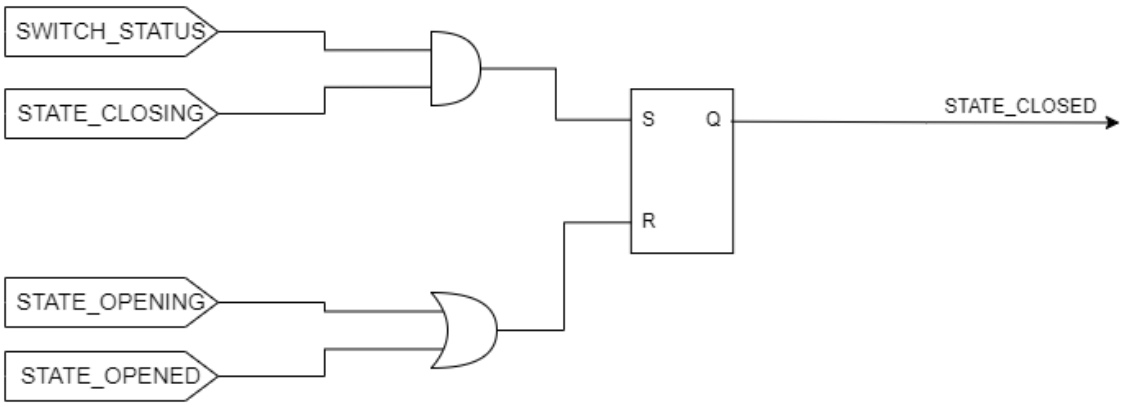


Figure 3.2: `Closed` state implementation in HiDraw

The `Closed` state implemented in HiDraw has three major components. The AND gate connected to the SET input of the SR latch represents the decision logic to make the state active. The OR gate connected to the RESET input of the SR latch represents the state being reset to 0 or being inactive. This is the second component, representing the reset logic. The third component is the output logic, which determines whether the state is currently active. Furthermore, the `Opened` state has a similar implementation to the `Closed` state, with only the signal names changed to reflect the state requirements.

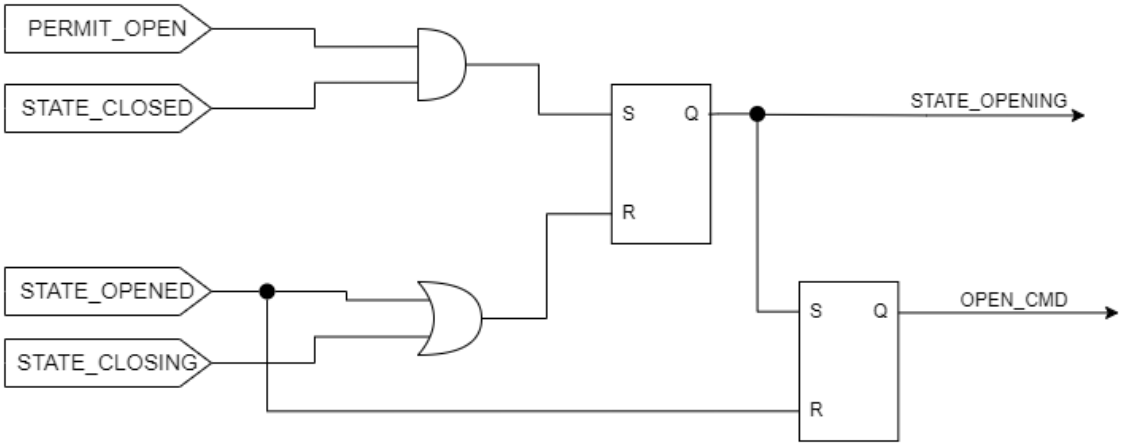


Figure 3.3: `Opening` state implementation in HiDraw

Figure 3.3 represents the implementation in HiDraw for the `Opening` state. This is also similar to the `Closed` state but includes an additional SR latch that determines the output command `open_cmd` for the circuit breaker to open. Similarly, the `Closing` state is also implemented in a similar manner to the `Opening` state, with only the signal names changed to reflect the state requirements.

3.2.3 Code Generation

Once the models were finalized, the next step was to generate the code and measure timing. To ensure that the timing was compared without any bias, the code was extracted from each of the generated programming environments and run on Visual Studio Code [8]. The computer hardware specifications are listed in Table 3.1, where the extracted code was run and compared for timing.

Component	Details
Processor	AMD Ryzen 3 5425U with Radeon Graphics (2.70 GHz)
Installed RAM	16.0 GB (15.3 GB usable)
Graphics Card	AMD Radeon (TM) Graphics
Total GPU Memory	8.2 GB (Dedicated: 421 MB, Shared: 0.3 GB)
Storage	239 GB WD PC SN810 SDCQNRY-256G-1006 (SSD)

Table 3.1: PC hardware specifications

The generated code was in C++ programming language and used Visual Studio Code, which was installed and configured on the computer with the specifications shown in Table 3.1. This ensured that both codes would run on similar hardware, providing equality during the timing comparison.

3.2.3.1 Time Measurement

The time measurement was done using the `chrono` [9] library, which provides clocks, time points, and durations for handling time-related operations. Clocks represent different sources of time, time points are instances in time, and durations measure time intervals. It is a powerful tool for precise time calculations. For measurement of time, `std::chrono::high_resolution_clock` was used, representing the clock with the smallest tick period provided by the implementation. Detailed implementation of the code can be found in Appendix A.1.
Additionally, the same could be implemented using a class included in the header file. The implementation of such a header file is shown in Appendix A.2, which has a class called `Timer` measuring the start time and, upon destruction of the class, the `Stop` function is called, and relevant timing information is printed out to the console. However, the method used in Appendix A.1 proved to be more useful since the assigned variables can be used to print out to a CSV file format for easier result generation.

3.2.3.2 Stateflow Generated Code

The model in Stateflow/Simulink was translated to C++ code using Simulink's built-in Embedded Coder application. The Embedded Coder [10] is a sophisticated tool that creates understandable, concise, and efficient C and C++ code for embedded processors typically used in mass manufacturing. It extends the capabilities of MATLAB Coder and Simulink Coder by providing advanced optimizations. These optimizations offer fine control over the resulting functions, files, and data. Addi-

tionally, Embedded Coder allows for simple integration with existing legacy code, various data formats, and calibration settings.
After analyzing the generated code, it was found that the execution of the states or determination of the active state was done using case logic. This can be seen in the C++ file generated for the model, shown in Listing 3.1.

Listing 3.1: Case logic generated for the model

```
// Chart: '<Root>/Sequences'
if (rtDW.is_active_c3_CB == 0U) {
  rtDW.is_active_c3_CB = 1U;
  rtDW.is_c3_CB = IN_Closed;
  rtDW.close_ord = 0.0;
} else {
  switch (rtDW.is_c3_CB) {
   case IN_Closed:
    rtDW.close_ord = 0.0;
    rtDW.is_c3_CB = IN_Opening;
    rtDW.open_ord = 1.0;
    break;

   case IN_Closing:
    rtDW.close_ord = 1.0;
    if (rtb_switch_status == 1) {
      rtDW.is_c3_CB = IN_Closed;
      rtDW.close_ord = 0.0;
    }
    break;

   case IN_Opened:
    rtDW.open_ord = 0.0;
    rtDW.is_c3_CB = IN_Closing;
    rtDW.close_ord = 1.0;
    break;

   default:
    // case IN_Opening:
    rtDW.open_ord = 1.0;
    if (rtb_switch_status == 0) {
      rtDW.is_c3_CB = IN_Opened;
      rtDW.open_ord = 0.0;
    }
    break;
  }
}

// End of Chart: '<Root>/Sequences'
```

This implies that only one case would be executed for each iteration or when the model is being stepped. This ensures mutually exclusive states, meaning that only one state would be active at a given time. The detailed implementation of the generated code can be found in the Appendices A.3 and A.4. The code used for result generation can also be found in these code files.

3.2.3.3 HiDraw Generated Code

The code generated from HiDraw is also in C++ language. The generation of code in HiDraw is different when compared with Embedded Coder. HiDraw generates code in a sequential manner, meaning that for each drawing, a separate *.cpp* file is generated. These files are interlinked with each other using a common header file.

Actual coding files cannot be included in this thesis due to proprietary restrictions within Hitachi Energy. However, a generic coding result will be explained with code snippets.
The generation of code in HiDraw was of a sequential nature. This ensured that the code would be predictable since the requirement was to run it in an RTOS, which benefits from having a constant loading on the computer rather than having variable, unpredictable code. The main C++ file generated by HiDraw is very specific to the target environment and unique. Therefore, it posed a challenge to extract and run the HiDraw generated C++ code without modification to the generated running structure. However, since the HiDraw drawings generate separate C++ files, all the generated C++ files relevant from the point of implementing the model and below were extracted. This necessitated writing a manual main C++ file to execute the files generated for the model in a manner similar to how HiDraw generated code runs on the target environment. A sample of code similar to what was executed to simulate the model within the computer specifications listed in Table 3.1 is included in Appendix A.5.

Listing 3.2: Similar code of HiDraw model execution code

```
// Header Files defined here

void CLOSINGState(void);
void OPENEDState(void);
void OPENINGState(void);
void CLOSEDState(void);


void CBModelMainDrawing (void) {
    // Execution of the states in sequence manner
    CLOSINGState();
    OPENEDState();
    OPENINGState();
    CLOSEDState();

    // The order of the above will be determined by how the states are defined in
        the GUI
}
```

As previously mentioned, the code generated for the model implemented in HiDraw is executed in a sequential manner. A code snippet similar to the sequential execution of the generated code is shown in Listing 3.2. The corresponding execution of each state can be seen, regardless of whether the state is active or not. This means that the code generated inside each state would be executed even when the respective state is not active. This method of code generation ensures predictability, making it suitable for implementation in an RTOS.
Once both models were completed and timing was measured using the method described in section 3.2.3.1, the results showed that the execution time for both the HiDraw model and the MATLAB Stateflow model was close to 0 milliseconds. This posed an issue, as it made it impossible to compare the two models since the time difference was not measurable. To address this issue, a separate function was included in each state of both the HiDraw and MATLAB Stateflow environments, designed to take a noticeable amount of time to execute.
Moreover, this function serves an additional purpose. Typically, the code implementation for control and protection functions within HVDC systems includes tasks

that execute in each state and consume a considerable amount of computing power. These tasks are generally executed even if the relevant state is not active, with their outputs calculated but never actually used since the state is inactive. This concept forms the basis of the study. By using a heavy function in each state to mimic this behavior, we can more clearly compare the execution differences between the two environments. Thus, it was decided to implement a heavy function, as described in chapter 3.3.1, within a new model that would represent an HVDC converter station and its operating states.

3.3 Converter Station Model

It was decided that the implementation of a converter station model would be more appropriate since the thesis requirement is to explore the possibilities of implementing Stateflow capabilities within the HVDC Control and Protection system, which is mainly applied to systems within the operating states of the converter station, as described in section 2.2.1. However, to overcome the timing measurement problem, a heavy task, described in section 3.3.1, was implemented within the model. This heavy function would represent similar actions performed during the actual implementation of control and protection code during operations within the given operating state.

3.3.1 Implementation of a Heavy Function

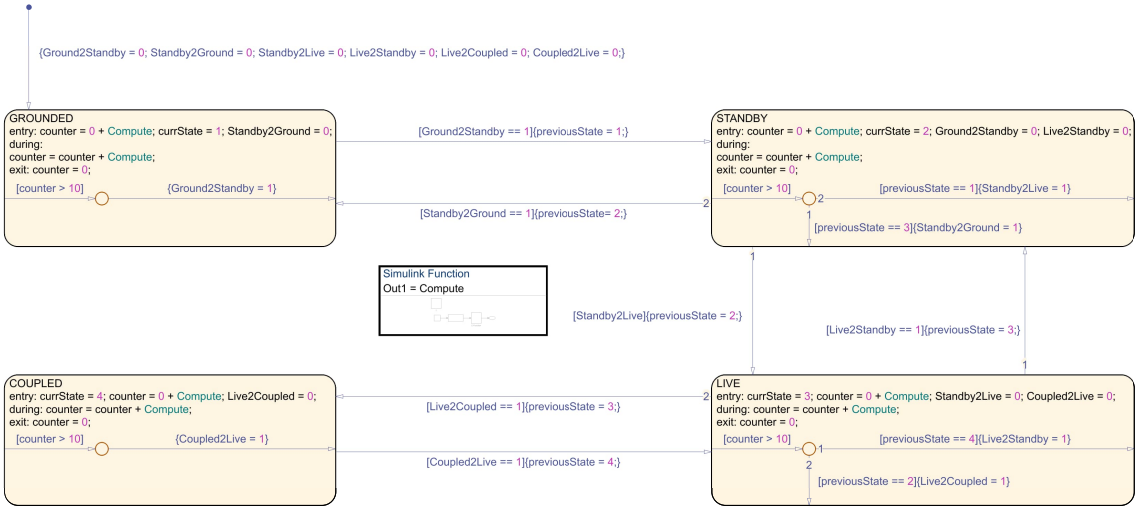
The function to increase the computing time was implemented to receive a boolean input, calculate a complex mathematical problem, and then return the same input. To tackle this, I decided to use the solution developed using C++ for **Problem 23** [11] of the Project Euler website, which involves summing all positive integers that cannot be written as the sum of two abundant numbers. The complete implemented function can be found in Appendix A.6, which returns the inputted boolean value after calculating the sum of all positive integers that cannot be written as the sum of two abundant numbers, with the answer being 4,179,871. This roughly took 400 milliseconds to execute on a machine with the hardware specifications mentioned in Table 3.1.
The upcoming chapters will describe how the custom heavy function was implemented within the states of both environments of the model, along with a description of the model in the respective environment.

3.3.2 Implementation in MATLAB

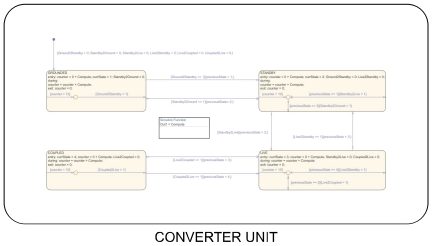
The converter station operation states were implemented within a Stateflow chart. This consisted of four states representing the operation states described in section 2.2.1, named `Grounded`, `Standby`, `Live`, and `Coupled`.
Figure 3.4 represents the model of the converter station implemented in MATLAB Simulink. From Figure 3.4b, it can be seen that the implemented Stateflow chart is free from inputs or outputs. Figure 3.4a represents the Stateflow chart, which is easy to understand by visual examination. The default transition is set to the `Grounded`

state, and the transition conditions are clearly visible on the arrows. Figure 3.4c shows the implementation of the heavy function inside the Simulink Function block within the Stateflow chart.

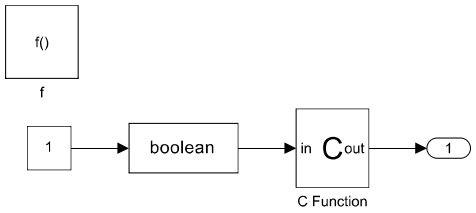
The model was designed to run independently from any external input signals to the Stateflow chart. This was achieved by setting up a counter, where the counter value would be initially set to 0. Once it enters a state, it executes the heavy function specified in section 3.3.1 and increments the counter value. While the state is active, the counter value continues to increase, and once it reaches the threshold value, it transitions to the next state. Additionally, the local data variable `previousState` is used to ensure that the transition does not revert to the previous state, thus ensuring a continuous iteration between the states from `Grounded` to `Standby` to `Live` to `Coupled`, and then back along the same path.



(a) Stateflow chart of the model



(b) Model view from Simulink



(c) Simulink function within the Stateflow Chart

Figure 3.4: Converter station model in MATLAB Simulink

3.3.3 Implementation in HiDraw

The state representation within HiDraw remained the same. However, since the logic within the states had changed, different types of symbol blocks had to be used. The actual symbols cannot be disclosed due to proprietary restrictions within Hitachi Energy, but similar block diagrams will be used to explain the model in this section.

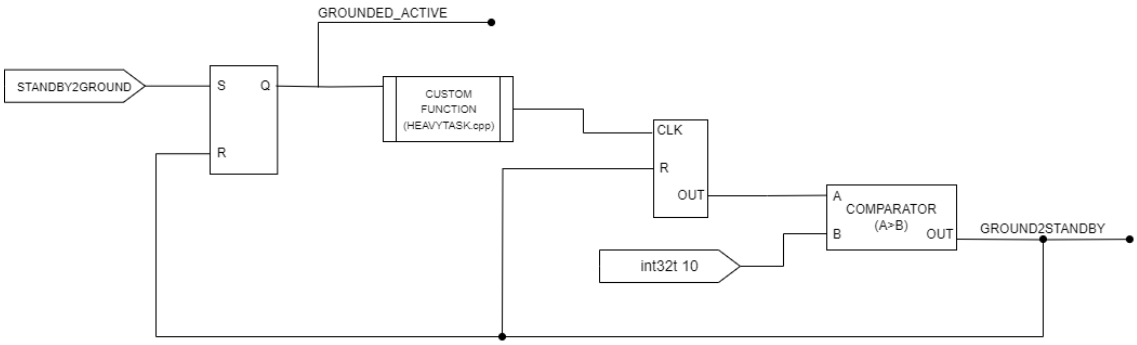


Figure 3.5: Grounded state drawing logic implementation

Figure 3.5 shows the `Grounded` state implementation. As mentioned previously, each state is considered as a separate *drawing* in HiDraw, so signal references between states are treated as external references. The signal `STANDBY2GROUND` is initially set to `true`, ensuring the default transition similar to MATLAB. The counter value shown will keep increasing by 1 per execution of the drawing as long as the `CLK` input is satisfied. The input is fed through the custom function block, which essentially runs the same heavy task mentioned in section 3.3.1. Once the counter value conditions are satisfied, the counter and the state will be reset, and the transition condition will be set. The transition to the `STANDBY` state will then occur. The implementation for the `COUPLED` state will remain the same, with the only difference being the signal naming.

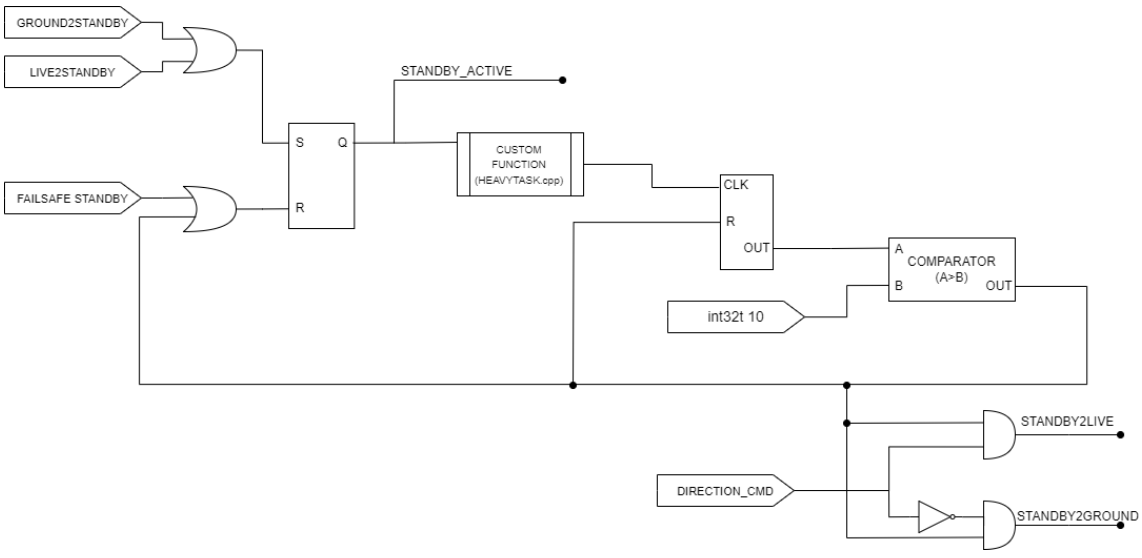


Figure 3.6: Standby state drawing logic implementation

Figure 3.6 shows the `STANDBY` state implementation. The difference compared to the `GROUNDED` state is the addition of decision logic to transfer to the next state. Here, a separate input determines the direction of transfer, and in combination with logic gates, it determines the transition to the next state. Similarly, the `LIVE` state remains the same as `STANDBY`, with the only difference being the signal names. Furthermore, compared to the MATLAB state implementation, we needed to implement mutual exclusivity to ensure that no other state would be active while one state is active. Therefore, a priority system was developed, as shown in Table 3.2.

Priority	State Name
1	`GROUNDED State`
2	`STANDBY State`
3	`LIVE State`
4	`COUPLED State`

Table 3.2: State-priority to ensure mutual exclusivity

3.3.4 Code Generation

The code generation process remained largely similar to the Circuit Breaker model. However, the complexity of the generated code increased. The MATLAB model generated case logic, while HiDraw had sequential execution, similar to the code generation explained in section 3.2.3. The code for the MATLAB models can be found in Appendix A.7 and A.8. HiDraw coding cannot be included due to proprietary restrictions within Hitachi Energy, but the generated code is similar in nature to Listing 3.2.
The implementation of the heavy task showed promising results, making it easier to compare since the execution time for the function was roughly 400 milliseconds. A detailed comparison will be discussed in Chapter 4. This enabled us to understand how the execution would be affected depending on state activity, code generation, and efficiency.
It's important to note that the reason for needing a measurable execution time is to properly evaluate and compare the performance of different models. Without this measurable time, it would be challenging to assess the impact of various factors on the models' performance.

3.4 Nested States Model

The next step after evaluating the converter station model was to implement nested states, i.e., states within states or substates. This model was necessary because the states mentioned in section 2.2.1 are the top hierarchical states, while during implementation, there would be several substates. Therefore, it was essential to see the next level of implementation of the hierarchy. This model includes four additional states implemented within the `GROUNDED` state: `State1`, `State2`, `State3`, and `State4`.

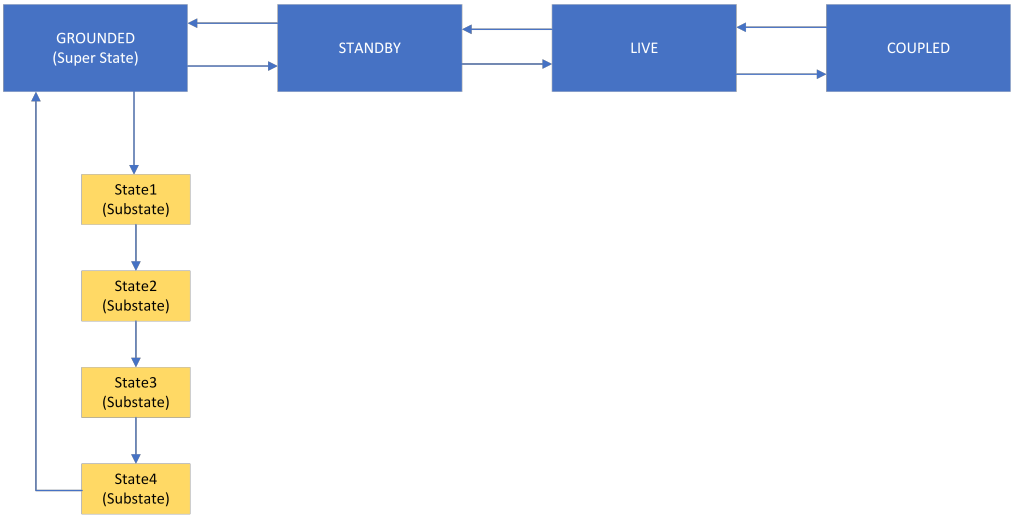


Figure 3.7: Nested states model hierarchy

Figure 3.7 shows the hierarchy of the model implementation, which contains four substates and four main states.

3.4.1 Implementation in MATLAB

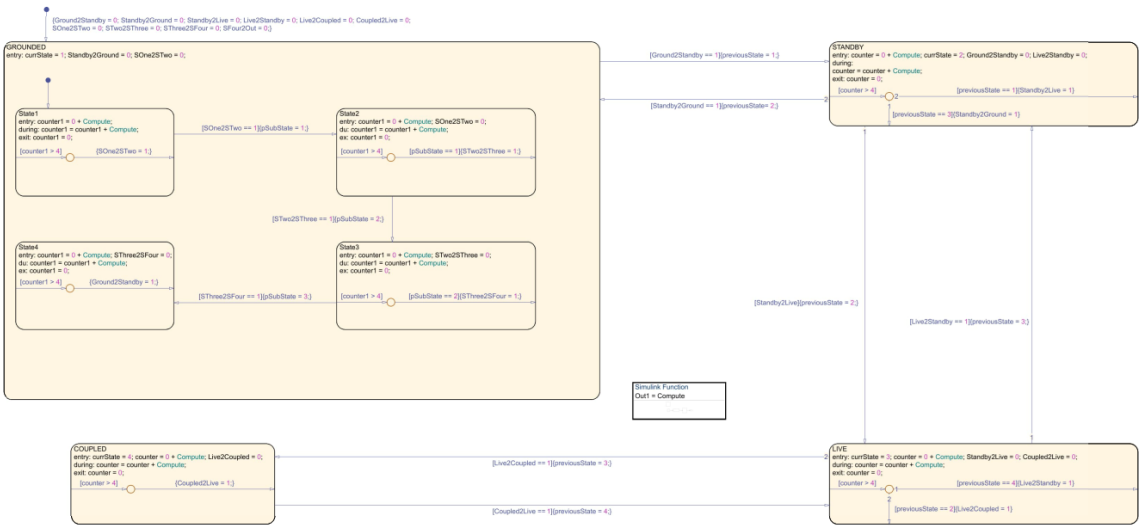


Figure 3.8: Stateflow chart of the model

Figure 3.8 represents the Stateflow chart, which is easy to understand by visual examination. The addition of four more states within the `GROUNDED` state is shown in Figure 3.9. The default transition is set to the `Grounded` state. The additional use of a default transition is required when implementing nested states. This is clearly seen in the figure, as there are two arrows with blue circles. The implementation of the heavy function using the Simulink Function block remained the same as in the previous model.

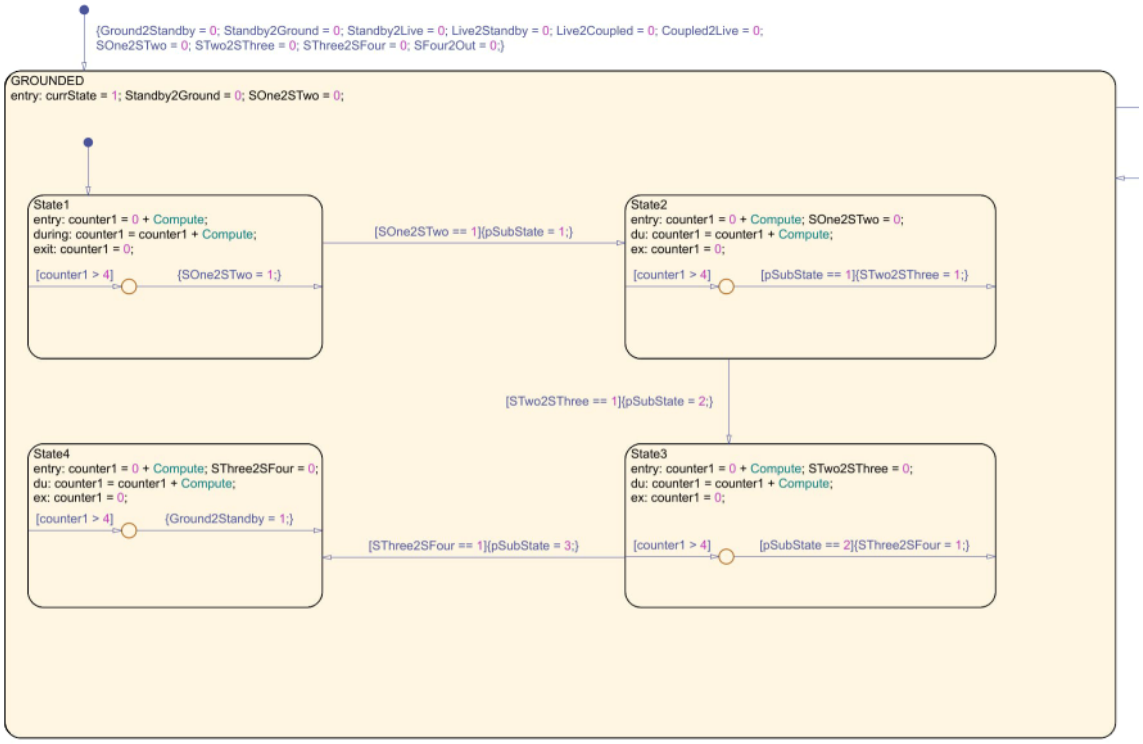


Figure 3.9: Nested states within `GROUNDED` state

The state transitions for the nested states were designed to be executed in sequence, rather than being similar to the earlier model. When the `GROUNDED` state is active, it enters the `State1` substate and transitions to `State2` once the counter conditions are satisfied. It then transitions to `State3`, to `State4`, and finally out of `GROUNDED` to `STANDBY`. It follows the same transition conditions as shown in Figure 3.8. It is worth noting that the Heavy Task function is executed at each end-state level, meaning it is executed in each of the substates and not within superstates. However, if no substates are present, it would be executed in the state. Once the model was finalized, the code was extracted and timing was measured.

3.4.2 Implementation in HiDraw

The model was modified from the previous version, and hierarchical symbols within a hierarchical symbol were used to implement the substates, similar to the MATLAB implementation. This can be seen in Figure 3.10, where the states within the `GROUNDED` state are shown. The transitions were also made with similar symbol blocks, which is straightforward.

The similar logic implemented can be seen in Figure 3.11 which shows the implementation within `State1`, all the four substates would remain the same with just signal names being the difference.

Afterwards, the code was generated and debugged. Once the code was determined to be working correctly, the timing was measured and results were generated to a CSV file.

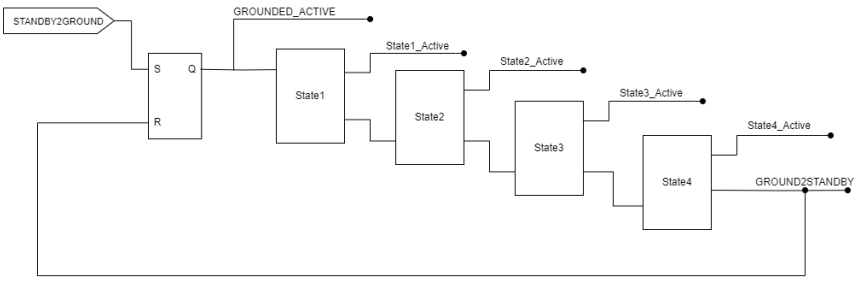


Figure 3.10: `GROUNDED` state implementation

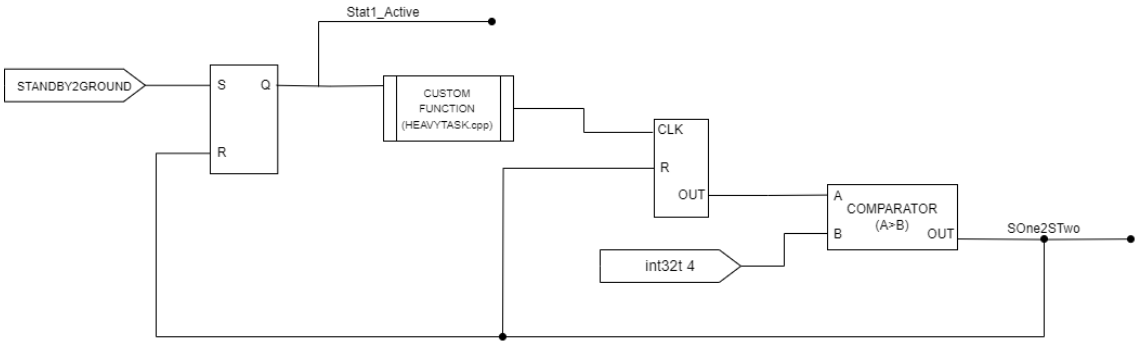


Figure 3.11: `State1` substate implementation

3.5 Completely-nested States Model

The final step of the method is to develop a model and complete the hierarchy. This model includes four substates within each of the four main states, totaling 16 substates. Figure 3.12 shows the hierarchy implemented in this model. This model implements four substates within each of the main states in a converter station. However, these states would not have identical workloads since each state would execute different functions depending on the requirements. In our model, each state executes the heavy function only and can be assumed to be equally loaded. Therefore, this will not be an accurate representation of a converter station's operational states, but for the purpose of this study, it fulfills the requirement to demonstrate functionality.

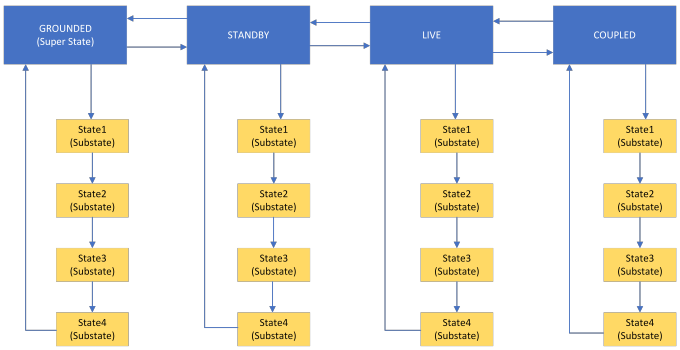


Figure 3.12: Completely nested model hierarchy

3.5.1 Implementation in MATLAB

The model is an expansion of the nested states model implemented in Section 3.4. The same four substates implemented in the *Nested States Model* are implemented in the remaining three main states, totaling 16 substates for the complete model. Figure 3.13 shows the Stateflow chart implemented in MATLAB Simulink. It can be seen that a similar implementation within the `Grounded` state has been applied to the other three states. However, the counter limit (iterations where the state would be active) has been reduced to 2, as it would take longer to execute in HiDraw even though the MATLAB execution time would remain the same.

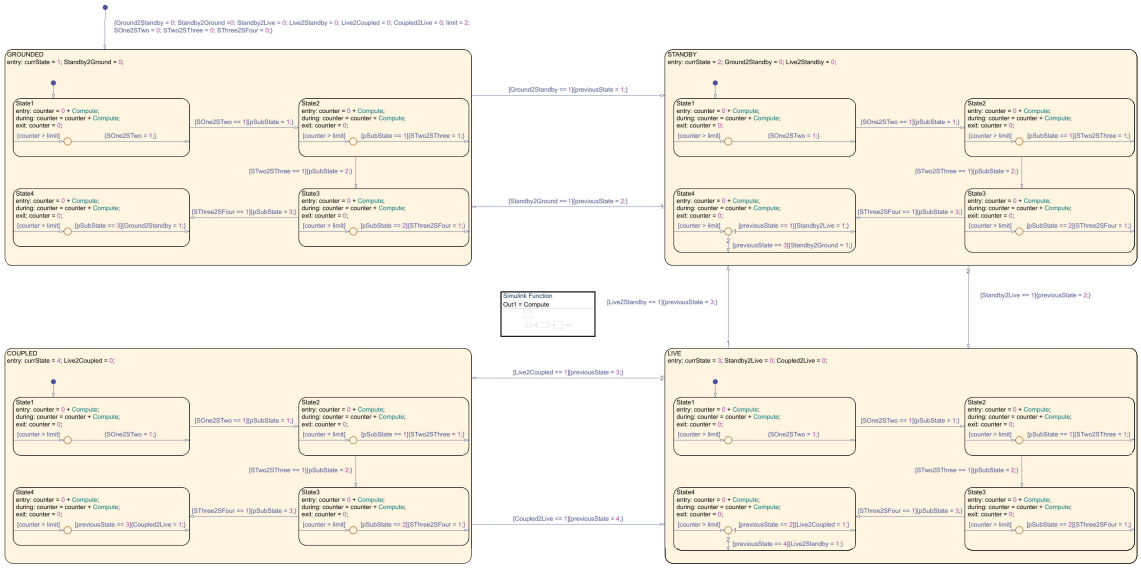


Figure 3.13: Stateflow chart for completely nested model

3.5.2 Implementation in HiDraw

The implementation of HiDraw for this model is an expansion of the model implemented in Section 3.4.2. The same procedure was followed, with a minor change to the limit for the state activity.

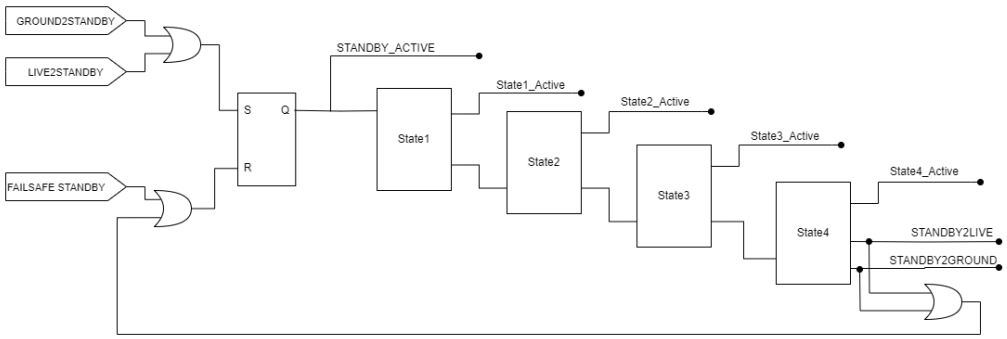


Figure 3.14: *STANDBY* state implementation in HiDraw

Figure 3.14 shows the implementation for the `Standby` state. This is similar to what was implemented in Section 3.4.2, with only changes to the decision logic and the inclusion of mutual exclusivity logic. Figure 3.15 depicts the substate `State 4` implemented within the superstate `Standby`. This follows the same procedure as described earlier, with a change to the state activity limit from 4 to 2. Additionally, following the hierarchy, the decision logic to transfer to the next state is also included within the substate itself.

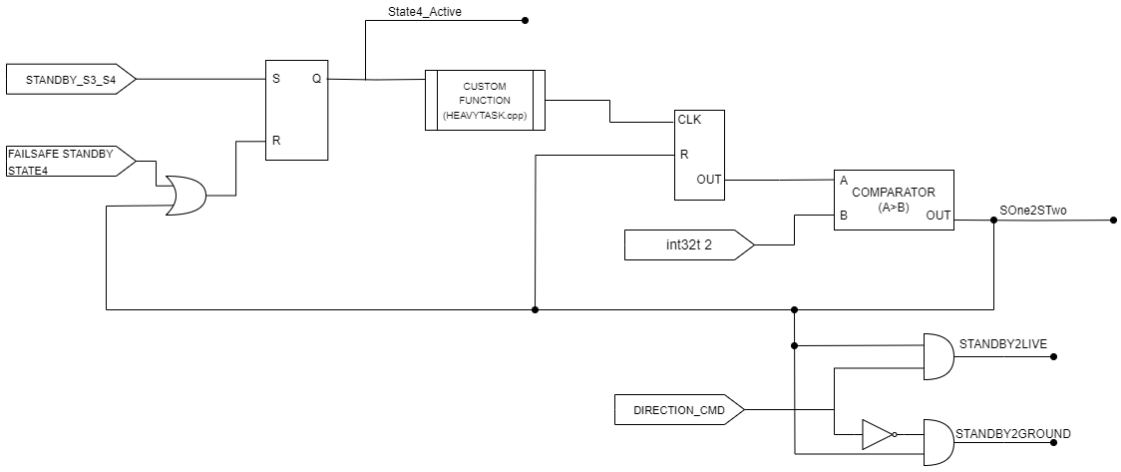


Figure 3.15: HiDraw implementation of `State 4` within `STANDBY` state

4
Results

This chapter will discuss the results generated from the four models implemented in Chapter 3. The analysis will focus on various performance metrics, including execution time, state transition efficiency, and system responsiveness. Each model's performance will be evaluated to understand how the number of states and the complexity of functions within those states affect the overall execution.
The first model, which serves as a baseline, will be analyzed to establish a reference point for performance metrics. Subsequent models, which incorporate increasing levels of complexity and additional states, will be compared against this baseline to highlight the impact of these changes. This comparative analysis will provide insights into the scalability of the state models and the efficiency of the implementations in both MATLAB Stateflow and HiDraw environments.
Particular attention will be given to the execution time per iteration for each model. This metric is critical as it directly impacts the system's real-time performance and responsiveness. The chapter will delve into how MATLAB Stateflow maintains execution time consistency despite increasing model complexity, and contrast this with the proportional increase in execution time observed in HiDraw. Understanding these differences will be crucial for optimizing future models and selecting the appropriate tools and techniques for state modeling.
Through detailed analysis and comparison of the results from these four models, this chapter aims to provide a comprehensive understanding of the performance characteristics and challenges associated with state modeling in dynamic systems. The findings will offer valuable insights for future developments and improvements in state-based system design and implementation.

4.1 Model Summary

This thesis involved developing four models in MATLAB Stateflow and HiDraw programming environments. Once the models were complete, the code was generated from the respective environments in C++ and then measured for execution timing on the same computer. The timing results will be discussed for each model.
To recap on the models, the main focus was to examine the implementation of states and how it would affect the timing. Table 4.1 provides a summarized overview of the models implemented in this thesis, highlighting important parameters such as the number of states and the number of states with the heavy function implemented.
Table 4.1 provides a summary of the models which are common to both programming environments (Matlab Stateflow and HiDraw).

Model Name	Number of Main States	Number of substates	Number of States with Heavy Function
Circuit Breaker	4	0	0
Converter Station	4	0	4
Nested States	4	4	7
Completely Nested	4	16	16

Table 4.1: Summary of models

4.2 Results for Circuit Breaker Model

This section will discuss the results of the Circuit Breaker Model implemented in section 3.2 with the measurement of execution timing using the method described in section 3.2.3.1.

4.2.1 Results obtained from MATLAB Stateflow Model

The Table 4.2 shows the results (timing measurements) in both milliseconds and micro seconds for each iteration of the code. It also mentions the active state for the respective iteration. The complete results are listed in Appendix B.1.

Iteration	Active State	Time (µs)	Time (ms)
1	CLOSED	0	0
2	OPENING	0	0
3	OPENED	0	0
4	CLOSING	0	0
5	CLOSED	0	0
6	OPENING	0	0
7	OPENED	0	0
8	CLOSING	0	0
9	CLOSED	0	0
10	OPENING	0	0
11	OPENED	0	0
12	CLOSING	0	0
13	CLOSED	0	0
14	OPENING	0	0
15	OPENED	0	0
16	CLOSING	0	0
17	CLOSED	0	0
18	OPENING	0	0
19	OPENED	0	0
20	CLOSING	0	0
21	CLOSED	1	0.001
22	OPENING	0	0

Iteration	Active State	Time (µs)	Time (ms)
23	OPENED	0	0
24	CLOSING	0	0
25	CLOSED	0	0
26	OPENING	0	0
27	OPENED	0	0
28	CLOSING	0	0
29	CLOSED	0	0
30	OPENING	0	0

Table 4.2: Timing measurements for circuit breaker model implemented in MATLAB

4.2.2 Results obtained from HiDraw Model

The Table 4.3 shows the results (timing measurements) in both milliseconds and micro seconds for each iteration of the code. It also mentions the active state for the respective iteration. Additionally, if there are any other active states in the same iterations, it is also shown in a separate column. The complete results are listed in Appendix B.2.

Iteration	Active State	Time (µs)	Time (ms)	Other Active State
0	CLOSED	0	0	
1	OPENING	0	0	
2	OPENED	0	0	
3	CLOSING	0	0	
4	CLOSED	0	0	CLOSING
5	OPENING	0	0	
6	OPENED	0	0	
7	CLOSING	0	0	
8	CLOSED	0	0	CLOSING
9	OPENING	0	0	
10	OPENED	0	0	
11	CLOSING	1	0.001	
12	CLOSED	0	0	CLOSING
13	OPENING	1	0.001	
14	OPENED	0	0	
15	CLOSING	0	0	
16	CLOSED	0	0	CLOSING
17	OPENING	0	0	
18	OPENED	1	0.001	
19	CLOSING	0	0	
20	CLOSED	0	0	CLOSING
21	OPENING	0	0	
22	OPENED	0	0	

Iteration	Active State	Time (µs)	Time (ms)	Other Active State
23	CLOSING	1	0.001	
24	CLOSED	0	0	CLOSING
25	OPENING	0	0	
26	OPENED	0	0	
27	CLOSING	0	0	
28	CLOSED	1	0.001	CLOSING
29	OPENING	0	0	
30	OPENED	0	0	

Table 4.3: Timing measurements for circuit breaker model implemented in HiDraw

4.2.3 Discussion of Results

The model implemented involved only the assignment of variables and stepping the model. Consequently, the measured execution time was almost zero, making it difficult to assess the differences between the two models, as each iteration's result was nearly zero.
Additionally, it was noted that for the HiDraw generated code, in some iterations there would be two active states. This issue arose because the code is generated in a sequential manner. For example, referring to Listing 3.2, it is clear that the `Closing` state is executed before the `Closed` state. If the `Closing` state is intended to be active for the current iteration and transition to the `Closed` state in the next iteration, the `Closing` state executes before the `Closed` state and will not be reset since the `Closed` state does not become active until it has executed, which occurs at the end of the iteration.
Therefore, in the following iteration, both the `Closed` and `Closing` states would be inactive, and the `Opening` state would become active. This issue was not problematic because the target environment for HiDraw is a Real-Time Operating System (RTOS). Thus, this would not be a noticeable problem during operation since execution at the lowest level would yield negligible results. No additional effort was required to resolve this issue.

4.3 Results for Converter Station Model

This section will discuss the results of the Converter Station Model implemented in section 3.3 with the measurement of execution timing using the method described in section 3.2.3.1.

4.3.1 Results obtained for MATLAB Stateflow Model

The Table 4.4 shows the results (timing measurements) in both milliseconds and micro seconds for each iteration of the code. It also mentions the active state for the respective iteration.

Iteration	Active State	Time (µs)	Time (ms)
1	Current state: GROUNDED	382044	382.04
2	Current state: GROUNDED	380593	380.59
3	Current state: GROUNDED	406856	406.86
4	Current state: GROUNDED	420186	420.19
5	Current state: GROUNDED	402397	402.40
6	Current state: GROUNDED	399607	399.61
7	Current state: GROUNDED	384986	384.99
8	Current state: GROUNDED	384109	384.11
9	Current state: GROUNDED	393436	393.44
10	Current state: GROUNDED	384328	384.33
11	Current state: GROUNDED	403727	403.73
12	Current state: STANDBY	390502	390.50
13	Current state: STANDBY	390969	390.97
14	Current state: STANDBY	390482	390.48
15	Current state: STANDBY	393406	393.41
16	Current state: STANDBY	397679	397.68
17	Current state: STANDBY	389462	389.46
18	Current state: STANDBY	395768	395.77
19	Current state: STANDBY	396405	396.41
20	Current state: STANDBY	391286	391.29
21	Current state: STANDBY	393575	393.58
22	Current state: STANDBY	388518	388.52
23	Current state: LIVE	417124	417.12
24	Current state: LIVE	392217	392.22
25	Current state: LIVE	394390	394.39
26	Current state: LIVE	388907	388.91

Table 4.4: Timing measurements for converter station model implemented in MATLAB Stateflow

4.3.2 Results obtained for HiDraw Model

The Table 4.5 shows the results (timing measurements) in both milliseconds and micro seconds for each iteration of the code. It also mentions the active state for the respective iteration.

Iteration	Active State	Time (µs)	Time (ms)
1	Current state: GROUNDED	1537756	1537.76
2	Current state: GROUNDED	1536020	1536.02
3	Current state: GROUNDED	1523504	1523.50
4	Current state: GROUNDED	1522874	1522.87
5	Current state: GROUNDED	1557896	1557.90
6	Current state: GROUNDED	1541526	1541.53
7	Current state: GROUNDED	1535374	1535.37

Iteration	Active State	Time (µs)	Time (ms)
8	Current state: GROUNDED	1517831	1517.83
9	Current state: GROUNDED	1521257	1521.26
10	Current state: GROUNDED	1544097	1544.10
11	Current state: GROUNDED	1566267	1566.27
12	Current state: STANDBY	1545150	1545.15
13	Current state: STANDBY	1556852	1556.85
14	Current state: STANDBY	1565552	1565.55
15	Current state: STANDBY	1532654	1532.65
16	Current state: STANDBY	1549792	1549.79
17	Current state: STANDBY	1565802	1565.80
18	Current state: STANDBY	1551190	1551.19
19	Current state: STANDBY	1546969	1546.97
20	Current state: STANDBY	1509320	1509.32
21	Current state: STANDBY	1508247	1508.25
22	Current state: STANDBY	1573503	1573.50
23	No Active State	1533565	1533.57
24	Current state: LIVE	1530979	1530.98
25	Current state: LIVE	1534517	1534.52
26	Current state: LIVE	1527465	1527.47

Table 4.5: Timing measurement for converter station model implemented in HiDraw

4.3.3 Discussion of Results

The implementation of the Heavy Function, as stated in Section 3.3.1, demonstrated significant differences in execution time between the two programming environments. In MATLAB, the code generation was based on case logic, whereas in HiDraw, it was sequential, as described in Section 3.2.3. The case logic implemented in MATLAB ensured that only the current state was executed, resulting in approximately 400 milliseconds per iteration. In contrast, the sequential logic implemented in HiDraw ensured that all states were executed in one iteration, causing the Heavy Function to run for each state, regardless of whether the state was active. This resulted in an execution time of approximately 1600 milliseconds per iteration, four times that of the case logic.

Additionally, in the HiDraw model, there are iterations where no active state is present. This occurs due to the sequential execution of the code. At certain points, an inactive state will be present because the iteration runs the intended active state before the previous state has been set as inactive. As a result, there will be no active state in the current period. This can pose a significant problem, and is a notable disadvantage of creating state machines in a fully sequential execution environment like HiDraw. In a Real-Time Operating System (RTOS), this could lead to potential errors during real-world execution, making it a critical consideration in the design of such systems.

4.4 Results for Nested States Model

This section will discuss the results of the Nested States Model implemented in section 3.4 with the measurement of execution timing using the method described in section 3.2.3.1.

4.4.1 Results obtained for MATLAB Stateflow Model

The Table 4.6 shows the results (timing measurements) in both milliseconds and micro seconds for each iteration of the code. It also mentions the active state and any active substate for the respective iteration. The complete results are listed in Appendix B.3.

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
1	GROUNDED	GROUNDED STATE1	407640	407.64
2	GROUNDED	GROUNDED STATE1	392063	392.063
3	GROUNDED	GROUNDED STATE1	403712	403.712
4	GROUNDED	GROUNDED STATE1	394265	394.265
5	GROUNDED	GROUNDED STATE1	396014	396.014
6	GROUNDED	GROUNDED STATE2	399016	399.016
7	GROUNDED	GROUNDED STATE2	395592	395.592
8	GROUNDED	GROUNDED STATE2	394068	394.068
9	GROUNDED	GROUNDED STATE2	389835	389.835
10	GROUNDED	GROUNDED STATE2	405389	405.389
11	GROUNDED	GROUNDED STATE3	404010	404.01
12	GROUNDED	GROUNDED STATE3	388576	388.576
13	GROUNDED	GROUNDED STATE3	412333	412.333
14	GROUNDED	GROUNDED STATE3	402015	402.015
15	GROUNDED	GROUNDED STATE3	408162	408.162
16	GROUNDED	GROUNDED STATE4	403640	403.64
17	GROUNDED	GROUNDED STATE4	415288	415.288
18	GROUNDED	GROUNDED STATE4	400660	400.66
19	GROUNDED	GROUNDED STATE4	394651	394.651
20	GROUNDED	GROUNDED STATE4	405852	405.852
21	STANDBY	No Active Substate	406792	406.792
22	STANDBY	No Active Substate	409546	409.546
23	STANDBY	No Active Substate	503747	503.747
24	STANDBY	No Active Substate	482338	482.338
25	STANDBY	No Active Substate	455019	455.019

Table 4.6: Timing measurements for nested states model implemented in MATLAB Stateflow

4.4.2 Results obtained for HiDraw Model

The Table 4.7 shows the results (timing measurements) in both milliseconds and micro seconds for each iteration of the code. It also mentions the active state and

any active substates for the respective iteration. The complete results are listed in Appendix B.4.

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
1	GROUNDED	GROUNDED STATE1	2975643	2975.64
2	GROUNDED	GROUNDED STATE1	2889311	2889.31
3	GROUNDED	GROUNDED STATE1	2818644	2818.64
4	GROUNDED	GROUNDED STATE1	2858019	2858.02
5	GROUNDED	GROUNDED STATE1	2875274	2875.27
6	GROUNDED	GROUNDED STATE2	2842261	2842.26
7	GROUNDED	GROUNDED STATE2	3295381	3295.38
8	GROUNDED	GROUNDED STATE2	3074096	3074.1
9	GROUNDED	GROUNDED STATE2	2924302	2924.3
10	GROUNDED	GROUNDED STATE2	3058324	3058.32
11	GROUNDED	GROUNDED STATE3	3209107	3209.11
12	GROUNDED	GROUNDED STATE3	3074969	3074.97
13	GROUNDED	GROUNDED STATE3	3318881	3318.88
14	GROUNDED	GROUNDED STATE3	3190056	3190.06
15	GROUNDED	GROUNDED STATE3	2967725	2967.72
16	GROUNDED	GROUNDED STATE4	2971601	2971.6
17	GROUNDED	GROUNDED STATE4	3030577	3030.58
18	GROUNDED	GROUNDED STATE4	2885931	2885.93
19	GROUNDED	GROUNDED STATE4	2875328	2875.33
20	GROUNDED	GROUNDED STATE4	2941991	2941.99
21	STANDBY	No Substate Active	2952864	2952.86
22	STANDBY	No Substate Active	2933195	2933.2
23	STANDBY	No Substate Active	2838262	2838.26
24	STANDBY	No Substate Active	3285443	3285.44
25	STANDBY	No Substate Active	3268066	3268.07

Table 4.7: Timing measurements for nested states model implemented in HiDraw

4.4.3 Discussion of Results

This model implementation consists of 7 states, each with a heavy function embedded within them. Therefore, we can deduce that there are 7 end states, comprising a superstate with 4 substates and 3 additional states. Despite this increase in the number of states, MATLAB Stateflow maintained an approximate execution time of 400 milliseconds per iteration. This consistency in execution time suggests that MATLAB Stateflow efficiently manages the additional computational load introduced by the increased number of states.
However, in HiDraw, the execution time increased proportionately with the number of states. This linear increase in execution time is expected due to the added complexity and the sequential processing of each state. As more states are added in future models, we can anticipate a further proportional increase in execution time. This behavior underscores the importance of optimizing state transitions and function executions within each state to manage the overall execution time effectively.

Additionally, the observation of "No Active State" in some iterations of the HiDraw-generated code remains. This phenomenon occurs due to the sequential nature of state execution, where at certain points, no state is active because the iteration is processing the transition from one state to another. This situation is not problematic, provided the system is designed to run on a Real-Time Operating System (RTOS). An RTOS can handle these brief periods without an active state without causing noticeable errors or delays in execution, ensuring the system's reliability and stability in real-world applications.
Overall, while both MATLAB Stateflow and HiDraw demonstrate robustness in handling complex state models, the scalability and performance aspects need careful consideration, particularly when increasing the number of states. Efficient management of state transitions and optimizing the functions within each state are crucial to maintaining acceptable execution times and system performance.

4.5 Results for Completely Nested Model

This section will discuss the results of the Nested States Model implemented in section 3.5 with the measurement of execution timing using the method described in section 3.2.3.1.

4.5.1 Results obtained for MATLAB Stateflow Model

The Table 4.8 shows the results (timing measurements) in both milliseconds and micro seconds for each iteration of the code. It also mentions the active state and any active substate for the respective iteration. The model was run for 100 iterations in total and the complete results can be found in Appendix B.5.

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
1	GROUNDED	Grounded STATE1	396948	396.948
2	GROUNDED	Grounded STATE1	394128	394.128
3	GROUNDED	Grounded STATE1	390087	390.087
4	GROUNDED	Grounded STATE2	383930	383.93
5	GROUNDED	Grounded STATE2	381894	381.894
6	GROUNDED	Grounded STATE2	389548	389.548
7	GROUNDED	Grounded STATE3	401154	401.154
8	GROUNDED	Grounded STATE3	384611	384.611
9	GROUNDED	Grounded STATE3	386238	386.238
10	GROUNDED	Grounded STATE4	399178	399.178
11	GROUNDED	Grounded STATE4	394219	394.219
12	GROUNDED	Grounded STATE4	382126	382.126
13	STANDBY	Standby STATE1	393051	393.051
14	STANDBY	Standby STATE1	390772	390.772
15	STANDBY	Standby STATE1	382019	382.019
16	STANDBY	Standby STATE2	383343	383.343
17	STANDBY	Standby STATE2	386446	386.446

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
18	STANDBY	Standby STATE2	409522	409.522
19	STANDBY	Standby STATE3	384495	384.495
20	STANDBY	Standby STATE3	387292	387.292
21	STANDBY	Standby STATE3	385637	385.637
22	STANDBY	Standby STATE4	382714	382.714
23	STANDBY	Standby STATE4	390497	390.497
24	STANDBY	Standby STATE4	384114	384.114
25	LIVE	Live STATE1	434920	434.92

Table 4.8: Timing measurements for completely nested model implemented in MATLAB Stateflow

4.5.2 Results obtained for HiDraw Model

The Table 4.9 shows the results (timing measurements) in both milliseconds and micro seconds for each iteration of the code. It also mentions the active state and any active substates for the respective iteration. The model was run for 100 iterations in total and the complete results can be found in Appendix B.6.

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
1	GROUNDED	GROUNDED STATE1	6650633	6650.63
2	GROUNDED	GROUNDED STATE1	6378560	6378.56
3	GROUNDED	GROUNDED STATE1	6537154	6537.15
4	GROUNDED	GROUNDED STATE2	6284979	6284.98
5	GROUNDED	GROUNDED STATE2	6291534	6291.53
6	GROUNDED	GROUNDED STATE2	6326340	6326.34
7	GROUNDED	GROUNDED STATE3	6420411	6420.41
8	GROUNDED	GROUNDED STATE3	6506535	6506.53
9	GROUNDED	GROUNDED STATE3	6335674	6335.67
10	GROUNDED	GROUNDED STATE4	6256502	6256.50
11	GROUNDED	GROUNDED STATE4	6317533	6317.53
12	GROUNDED	GROUNDED STATE4	6376635	6376.64
13	STANDBY	STANDBY STATE1	6573735	6573.74
14	STANDBY	STANDBY STATE1	6258275	6258.28
15	STANDBY	STANDBY STATE1	6276901	6276.90
16	STANDBY	STANDBY STATE2	6292056	6292.06
17	STANDBY	STANDBY STATE2	6271430	6271.43
18	STANDBY	STANDBY STATE2	6346406	6346.41
19	STANDBY	STANDBY STATE3	6300817	6300.82
20	STANDBY	STANDBY STATE3	6287753	6287.75
21	STANDBY	STANDBY STATE3	6303522	6303.52
22	STANDBY	STANDBY STATE4	6258455	6258.45
23	STANDBY	STANDBY STATE4	6268247	6268.25
24	STANDBY	STANDBY STATE4	6393694	6393.69

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
25	LIVE	LIVE STATE1	6255922	6255.92
26	LIVE	LIVE STATE1	6320106	6320.11
27	LIVE	LIVE STATE1	6245688	6245.69
28	LIVE	LIVE STATE2	6247782	6247.78
29	LIVE	LIVE STATE2	6298780	6298.78
30	LIVE	LIVE STATE2	6249336	6249.34

Table 4.9: Timing measurements for completely nested model implemented in HiDraw

4.5.3 Discussion of Results

This model implementation consists of 4 super states and comprises of 4 substates in each superstate. Each substate comprises the heavy function within them. Therefore, we can deduce that there are 16 executable states. Despite the increase in the number of states compared to the Nested States Model, MATLAB Stateflow maintained the same execution time. This consistency in execution time suggests that MATLAB Stateflow efficiently manages the additional computational load introduced by the increased number of states.
HiDraw results are as expected and similar to what is discussed in section 4.4.3 where the execution time increased proportionately to the number of executable states. Therefore, the total execution time is approximately around 6400 milliseconds which is 16 times the execution time in MATLAB per iteration.

5
Discussion & Conclusion

This chapter discusses the advantages and disadvantages of both MATLAB and HiDraw models. It also evaluates the outcomes of the thesis, provides insights, and suggests future work. Finally, conclusions are presented.

5.1 Discussion

MATLAB's implementation results in faster execution times due to its efficient handling of state transitions, leading to a lower computational load which is beneficial for applications that require quick responses. The use of case logic in MATLAB ensures that only the current state is executed, minimizing unnecessary computations and enhancing performance and efficiency. However, this case-wise execution can result in non-predictable code performance, as the execution time can vary depending on which state is active. While MATLAB's execution model is not inherently designed for real-time systems, additional optimizations and modifications could potentially adapt it to meet real-time requirements, warranting further exploration. MATLAB also provides a highly intuitive graphical interface for modeling state machines, making it easier for users to visualize and manage state transitions. Additionally, MATLAB's graphical interface and case-wise execution make it straightforward to identify and debug the active state, facilitating quicker troubleshooting and development cycles.

On the other hand, HiDraw's sequential execution model results in slower execution times as it processes all states in sequence, leading to a higher computational load, which can be a drawback for time-sensitive applications. In HiDraw, all states are executed sequentially regardless of their activity, ensuring that no state is missed but leading to inefficiencies as inactive states are also processed. However, this consistent execution pattern makes HiDraw highly predictable, with stable performance regardless of the active state, which is crucial for systems where timing consistency is essential. HiDraw's predictable and consistent execution pattern makes it well-suited for real-time systems where deterministic behavior is a key requirement. While HiDraw offers a graphical interface, it is not as intuitive as MATLAB's, making it less straightforward to navigate and manage state transitions. The sequential execution and less intuitive graphical interface also make it harder to identify and debug the active state, potentially leading to longer troubleshooting times.

Furthermore, during method development observations were made on the main components required for a state to function. Following main components are required generally for a state.

- Decision Logic to set the State Active
- Operation during State Activity
- Decision Logic to Transfer to Next Possible State
- Mutual Exclusivity Logic (Check if other states are active)
- Logic to prevent "No Active State" scenario

These four main components are not ensured in HiDraw while in MATLAB these functions are built into the programming environment and ensures that all requirements are satisfied for the model to function. Therefore, it is essential that these requirements are manually checked and ensured in the HiDraw programming environment.

5.2 Advantages and Disadvantages

MATLAB Stateflow has several advantages. It offers faster execution time, leading to a lower computational load, and its efficient case-wise code implementation only executes the current state. MATLAB also provides a user-friendly graphical representation for easier state management and is easier to debug and find the active state due to its intuitive graphical interface. However, MATLAB has its disadvantages. Its code performance is non-predictable, with variability depending on the active state, and it is not inherently suitable for real-time systems without further optimization.
HiDraw also has its advantages. It provides predictable code with consistent performance independent of the active state and is well-suited for real-time systems due to its deterministic execution. HiDraw ensures that no state is missed due to its sequential execution. However, HiDraw's disadvantages include slower execution time, leading to a higher computational load, and inefficient processing of non-active states. Additionally, it has a less user-friendly graphical representation compared to MATLAB and is harder to debug and find the active state due to its sequential execution model and less intuitive interface.
Table 5.1 shows the comparison of features between MATLAB and HiDraw, in terms of execution, performance, graphical representation, and suitability for real-time systems.

Feature	MATLAB Stateflow	HiDraw
Execution Time	Faster execution time, leading to lower computational load	Slower execution time, leading to higher computational load
Code Implementation	Efficient case-wise code implementation only executes the current state	Inefficient processing of non-active states
Graphical Representation	User-friendly graphical representation for easier state management	Less user-friendly graphical representation

Feature	MATLAB Stateflow	HiDraw
Debugging	Easier to debug and find the active state due to intuitive graphical interface	Harder to debug and find the active state due to sequential execution model and less intuitive interface
Code Performance	Non-predictable code performance with variability depending on the active state	Predictable code with consistent performance independent of the active state
Suitability for Real-time Systems	Not inherently suitable for real-time systems without further optimization	Well-suited for real-time systems due to deterministic execution
State Management	Efficient state management	Ensures no state is missed due to sequential execution

Table 5.1: Comparison of MATLAB Stateflow and HiDraw

5.3 Conclusion

In conclusion, both MATLAB and HiDraw have distinct advantages and disadvantages. MATLAB excels in execution speed due to its efficient handling of state transitions and user-friendly interfaces, making it an excellent tool for applications that require quick responses and easy debugging. However, MATLAB struggles with predictability and is not inherently suitable for real-time applications without significant optimization. Its performance can vary depending on the active state, which introduces non-deterministic behavior that is unsuitable for real-time systems where timing consistency is critical.
HiDraw, on the other hand, offers predictable and deterministic execution, making it well-suited for real-time systems. Its sequential execution model ensures stable performance independent of the active state, which is crucial for maintaining consistency in real-time applications. However, this comes at the cost of slower execution times and higher computational loads, as all states are processed sequentially, including inactive ones. Additionally, HiDraw's interface is less intuitive compared to MATLAB's, making it harder to navigate and debug, leading to a steeper learning curve for users.
The aim of this thesis was to study both systems and identify features from MATLAB Stateflow that could be adapted into the HiDraw programming environment. As stated above, the graphical interface features of MATLAB, such as clear visualization of state transitions, active states, and ensuring mutually exclusive states, are key takeaways for improving HiDraw. These graphical enhancements could significantly improve the usability of HiDraw, making it easier for engineers to design and manage state machines.
The case-wise logic implementation used in MATLAB, while efficient for certain applications, is not suitable for HiDraw. This is because HiDraw targets Real-Time Operating Systems (RTOS), where uneven loading on the computer is undesirable. Therefore, adapting HiDraw to include a more sophisticated graphical interface,

similar to MATLAB's, would be more beneficial. This could include symbols and visual cues that indicate state conditions, flow between states, and conditions for state transitions. Additionally, preset conditions to ensure state activity, decision logic for activating and deactivating states, mutual exclusivity logic, and output logic for state transitions could further enhance HiDraw's usability.
Future work could focus on optimizing MATLAB for real-time performance to address its current limitations in this area. For HiDraw, improving the interface and debugging capabilities could enhance user experience and efficiency. Furthermore, exploring the impact of memory utilization in both environments could provide insights for further optimizations. Implementing these improvements would make HiDraw a more powerful and user-friendly tool, ultimately benefiting design engineers, particularly those working at Hitachi, by streamlining their workflow and improving the efficiency of their design processes.

5.4 Suggestions and Future Work

The following points outline potential areas for future research and development to enhance both MATLAB and HiDraw implementations:

- Explore the impact of memory utilization in both MATLAB and HiDraw implementations.
- Implement a new symbol in HiDraw that mimics the visual representation of Stateflow to enhance usability.
  - This symbol should indicate the state conditions (active or inactive), the flow between states, and the conditions required to transition to another state.
  - Include preset conditions to ensure state activity:
    - Decision logic to set the state active.
    - Decision logic to set the state inactive.
    - Mutual exclusivity logic to ensure that no other state is active simultaneously.
    - Output logic to show state transitions and set outputs as required.
- Ensure that the main component requirements are generally met for a state to function, as described in section 5.1.

The study provides sufficient insights into the need for implementing states within the HiDraw programming environment with the evolution of HVDC systems. The next step would be to implement the aforementioned criteria while ensuring the requirements of high-level control and protection functions as per industry standards.

Bibliography

[1] Karlsson, T., Hyttinen, M., Carlsson, L., & Björklund, H. (2020). Modern control and protection system for HVDC.

[2] Ruiz Yuncosa, J. L. (2020). Test Automation for HVDC System Protection A Proof of Concept for ABB.

[3] Björklund, H., Hallmans, D., Gahane, B., & Buell, V. (2023). Control and Protection for HVDC: An introduction to MACH™. Hitachi Review

[4] Pierri, E., Binder, O., Hemdan, N. G., & Kurrat, M. (2017). Challenges and opportunities for a European HVDC grid. Renewable and Sustainable Energy Reviews, 70, 427-456.

[5] Tanenbaum, A. (2009). Modern operating systems. Pearson Education, Inc.,.

[6] Wang, J. (2019). Formal methods in computer science. Chapman and Hall/CRC.

[7] Create a hierarchy to manage system complexity. Create a Hierarchy to Manage System Complexity - MATLAB & Simulink - MathWorks Nordic. (n.d.). `https://se.mathworks.com/help/stateflow/gs/hierarchy.html`

[8] Visual Studio Code - Code Editing. Redefined. (2021, November 3). `https://code.visualstudio.com/`

[9] Date and time utilities - cppreference.com. (n.d.). `https://en.cppreference.com/w/cpp/chrono`

[10] Embedded Coder. (n.d.). `https://se.mathworks.com/products/embedded-coder.html`

[11] Hughes, C. (n.d.). #23 Non-Abundant Sums - Project Euler. `https://projecteuler.net/problem=23`

[12] Siemens Energy Global GmbH & Co. KG. (2023). *HVDC Grid Textbook*. Siemens Energy.

[13] International Electrotechnical Commission. (2023). IEC TS 63291-1:2023 High voltage direct current (HVDC) grid systems and connected converter stations - Guideline and parameter lists for functional specifications - Part 1: Guideline (1st ed.). IEC.

A

Appendix - Programming Code

Listing A.1: Measuring of timing in C++

```
// This Code Snippet is written to explain how the timing was measure for the executed code
// First the relevant header files are included
#include <chrono> // This is the library required to access the relevant time measurements
/*
Other header files to be included here to make the code work
Followed by other supporting code generated
*/
// The main program
int main () {
    /*
    Variables and other supporting code generated to fulfill the code execution
    */
    // Initiating the counter to determine the number of iterations (steps of the model) to run
    int counter = 0;
    // while loop setup to step the model
    while (counter <= 100) {
        counter++; // Increase the iteration number
        //start measuring of time and execute code
        auto start_time = std::chrono::high_resolution_clock::now();
        /*
        Model executed here. Usually run in a separate C++ file which contains the model. A single line execution.
        For example in the code generated in MATLAB Stateflow:
        The call syntax for rt_OneStep is
        rt_OneStep(); // This line is executed here
        */
        //End measuring time and assign duration variables
        auto endTimepoint = std::chrono::high_resolution_clock::now();
        auto start = std::chrono::time_point_cast<std::chrono::microseconds>(start_time).time_since_epoch().count();
        auto end = std::chrono::time_point_cast<std::chrono::microseconds>(endTimepoint).time_since_epoch().count();
        auto duration = end - start;
        double ms = duration * 0.001; // Conversion of measured time to milliseconds
        /*
        Using duration and ms variables, the relevant time can be measured for the execution of the model step.
        The variables can be written to a CSV file or printed to the console as required.
        */
    }
    return 0;
}
```

Listing A.2: Time Measurement using a header file in C++

```cpp
// Timer.h

#ifndef TIMER_H
#define TIMER_H

#include <chrono>
#include <iostream>

class Timer {
public:
    Timer() {
        m_StartTimepoint = std::chrono::high_resolution_clock::now();
    }

    ~Timer() {
        Stop();
    }

    void Stop() {
        auto endTimepoint = std::chrono::high_resolution_clock::now();

        auto start = std::chrono::time_point_cast<std::chrono::microseconds>(
            m_StartTimepoint).time_since_epoch().count();
        auto end = std::chrono::time_point_cast<std::chrono::microseconds>(
            endTimepoint).time_since_epoch().count();

        auto duration = end - start;
        double ms = duration * 0.001;

        std::cout << duration << " microseconds (" << ms << " milliseconds)\n";
    }

private:
    std::chrono::time_point<std::chrono::high_resolution_clock> m_StartTimepoint;
};

#endif // TIMER_H
```

Listing A.3: Main File Generated for CB Model using Embedded Coder

```cpp
//
// File: ert_main.cpp
//
// Code generated for Simulink model 'CB'.
//
// Model version                  : 1.9
// Simulink Coder version         : 9.9 (R2023a) 19-Nov-2022
// C/C++ source code generated on : Mon May 13 08:53:27 2024
//
// 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 <stdio.h>              // This example main program uses printf/fflush
#include "CB.h"                        // Model header file

static Switch_Model rtObj;             // Instance of model class

// define variables and structures required for result generation in CSV format -
    Added by Arjuna

uint8_T mainState;
```

```
25
26 struct SignalInfo {
27     std::string name;
28      int ptr;
29 };
30
31 std::vector<SignalInfo> mainStateList; // creation of vector to store the States
       and Substates
32
33 //
34 // Associating rt_OneStep with a real-time clock or interrupt service routine
35 // is what makes the generated code "real-time".  The function rt_OneStep is
36 // always associated with the base rate of the model.  Subrates are managed
37 // by the base rate from inside the generated code.  Enabling/disabling
38 // interrupts and floating point context switches are target specific.  This
39 // example code indicates where these should take place relative to executing
40 // the generated code step function.  Overrun behavior should be tailored to
41 // your application needs.  This example simply sets an error status in the
42 // real-time model and returns from rt_OneStep.
43 //
44 void rt_OneStep(void);
45 void rt_OneStep(void)
46 {
47   static boolean_T OverrunFlag{ false };
48
49   // Disable interrupts here
50
51   // Check for overrun
52   if (OverrunFlag) {
53     rtmSetErrorStatus(rtObj.getRTM(), "Overrun");
54     return;
55   }
56
57   OverrunFlag = true;
58
59   // Save FPU context here (if necessary)
60   // Re-enable timer or interrupt here
61   // Set model inputs here
62
63   // Step the model
64   rtObj.step();
65
66   // Get model outputs here
67
68   // Indicate task complete
69   OverrunFlag = false;
70
71   // Disable interrupts here
72   // Restore FPU context here (if necessary)
73   // Enable interrupts here
74 }
75
76 //
77 // The example main function illustrates what is required by your
78 // application code to initialize, execute, and terminate the generated code.
79 // Attaching rt_OneStep to a real-time clock is target specific. This example
80 // illustrates how you do this relative to initializing the model.
81 //
82 int_T main(int_T argc, const char *argv[])
83 {
84   // Unused arguments
85   (void)(argc);
86   (void)(argv);
87
88   // Initialize model
89   rtObj.initialize();
90
```

```
91  // Attach rt_OneStep to a timer or interrupt service routine with
92  //  period 0.2 seconds (base rate of the model) here.
93  //  The call syntax for rt_OneStep is
94  //
95  //   rt_OneStep();
96
97  printf("Warning: The simulation will run forever. "
98         "Generated ERT main won't simulate model step behavior. "
99         "To change this behavior select the 'MAT-file logging' option.\n");
100 fflush((nullptr));
101
102 // Added code part to initialize the variables required for result generation -
        Added by Arjuna
103
104 int counter = 0; // Initialize counter
105
106 mainStateList = {
107       {"CLOSED",1},
108       {"CLOSING",2},
109       {"OPENED",3},
110       {"OPENING",4},
111       {"No Active State",0}
112 };
113
114 // The output variables are to be printed to an excel file showing the Active
        State and Substate for each iteration
115
116 std::ofstream csvFile("ResultsSimulinkCB.csv"); // Create the CSV File
117
118 csvFile << "Iteration,Active State,Time (µs), Time (ms)" << std::endl; // Headers
        of the CSV File
119
120 while (counter < 50) {
121   //  Perform application tasks here
122
123   counter++; // increase the counter value
124
125   //start measuring of time and execute code
126   auto start_time = std::chrono::high_resolution_clock::now();
127   rt_OneStep();
128
129   //End measuring time and assign duration variables
130   auto endTimepoint = std::chrono::high_resolution_clock::now();
131   auto start = std::chrono::time_point_cast<std::chrono::microseconds>(start_time
        ).time_since_epoch().count();
132   auto end = std::chrono::time_point_cast<std::chrono::microseconds>(endTimepoint
        ).time_since_epoch().count();
133
134   auto duration = end - start;
135   double ms = duration * 0.001;
136
137   // Result generation to CSV File
138
139   mainState = rtObj.state;
140
141
142   csvFile << counter << ",";
143
144   for (SignalInfo& info : mainStateList) {
145
146     if (info.ptr == int(mainState)) {
147       csvFile << info.name;
148       break;
149     }
150   }
151   csvFile << "," << duration << "," << ms << std::endl;
152 }
```

```
153
154   return 0;
155 }
156
157 //
158 // File trailer for generated code.
159 //
160 // [EOF]
161 //
```

Listing A.4: C++ file generated for the CB Model using Embedded Coder

```
1  //
2  // File: CB.cpp
3  //
4  // Code generated for Simulink model 'CB'.
5  //
6  // Model version                  : 1.9
7  // Simulink Coder version         : 9.9 (R2023a) 19-Nov-2022
8  // C/C++ source code generated on : Mon May 13 08:53:27 2024
9  //
10 // Target selection: ert.tlc
11 // Embedded hardware selection: Intel->x86-64 (Windows64)
12 // Code generation objectives:
13 //    1. Execution efficiency
14 //    2. RAM efficiency
15 // Validation result: Not run
16 //
17 #include "CB.h"
18 #include "rtwtypes.h"
19
20 // Named constants for Chart: '<Root>/Sequences'
21 const uint8_T IN_Closed{ 1U };
22
23 const uint8_T IN_Closing{ 2U };
24
25 const uint8_T IN_Opened{ 3U };
26
27 const uint8_T IN_Opening{ 4U };
28
29 // Model step function
30 void Switch_Model::step()
31 {
32   int32_T rtb_switch_status;
33   boolean_T rtb_Logic_idx_0;
34
35   // Switch: '<Root>/Switch' incorporates:
36   //   UnitDelay: '<Root>/Unit Delay'
37
38   rtb_switch_status = !rtDW.UnitDelay_DSTATE;
39
40   // Chart: '<Root>/Sequences'
41   if (rtDW.is_active_c3_CB == 0U) {
42     rtDW.is_active_c3_CB = 1U;
43     rtDW.is_c3_CB = IN_Closed;
44     rtDW.close_ord = 0.0;
45   } else {
46     switch (rtDW.is_c3_CB) {
47      case IN_Closed:
48       rtDW.close_ord = 0.0;
49       rtDW.is_c3_CB = IN_Opening;
50       rtDW.open_ord = 1.0;
51       break;
52
53      case IN_Closing:
54       rtDW.close_ord = 1.0;
55       if (rtb_switch_status == 1) {
```

```
56          rtDW.is_c3_CB = IN_Closed;
57          rtDW.close_ord = 0.0;
58        }
59        break;
60
61      case IN_Opened:
62        rtDW.open_ord = 0.0;
63        rtDW.is_c3_CB = IN_Closing;
64        rtDW.close_ord = 1.0;
65        break;
66
67      default:
68        // case IN_Opening:
69        rtDW.open_ord = 1.0;
70        if (rtb_switch_status == 0) {
71          rtDW.is_c3_CB = IN_Opened;
72          rtDW.open_ord = 0.0;
73        }
74        break;
75      }
76    }
77
78    // End of Chart: '<Root>/Sequences'
79
80    // CombinatorialLogic: '<S1>/Logic' incorporates:
81    //   DataTypeConversion: '<Root>/Cast To Boolean'
82    //   DataTypeConversion: '<Root>/Cast To Boolean1'
83    //   Memory: '<S1>/Memory'
84
85    rtb_Logic_idx_0 = rtConstP.Logic_table[(((static_cast<uint32_T>(rtDW.open_ord
86      != 0.0) << 1) + (rtDW.close_ord != 0.0)) << 1) + rtDW.Memory_PreviousInput];
87
88    // Update for UnitDelay: '<Root>/Unit Delay'
89    rtDW.UnitDelay_DSTATE = rtb_Logic_idx_0;
90
91    // Update for Memory: '<S1>/Memory'
92    rtDW.Memory_PreviousInput = rtb_Logic_idx_0;
93
94    state = rtDW.is_c3_CB;
95  }
96
97  // Model initialize function
98  void Switch_Model::initialize()
99  {
100   // (no initialization code required)
101 }
102
103 // Constructor
104 Switch_Model::Switch_Model() :
105   rtDW(),
106   rtM()
107 {
108   // Currently there is no constructor body generated.
109 }
110
111 // Destructor
112 // Currently there is no destructor body generated.
113 Switch_Model::~Switch_Model() = default;
114
115 // Real-Time Model get method
116 Switch_Model::RT_MODEL * Switch_Model::getRTM()
117 {
118   return (&rtM);
119 }
120
121 //
```

Listing A.5: Manually written Main C++ file to run the HiDraw model (Without Actual variables)

```cpp
// Header files to be used
#include <chrono>
#include <iostream>
#include <vector>

/*
Functions and variables used defined here
*/
void CBModelMainDrawing(void);

// Structure for result generation
struct SignalInfo {
    std::string name;
    volatile bool* ptr;
};

int main () {
    // Define the counter to step the model or determine number of iterations
    int counter = 0;

    while (counter < 50) {
        counter++; // Increase the iteration number

        //start measuring of time and execute code
        auto start_time = std::chrono::high_resolution_clock::now();
        CBModelMainDrawing();

        //End measuring time and assign duration variables
        auto endTimepoint = std::chrono::high_resolution_clock::now();
        auto start = std::chrono::time_point_cast<std::chrono::microseconds>(
            start_time).time_since_epoch().count();
        auto end = std::chrono::time_point_cast<std::chrono::microseconds>(
            endTimepoint).time_since_epoch().count();

        auto duration = end - start;
        double ms = duration * 0.001;

        // Other code to implement for result generation (writing to CSV File)
    }
    return 0;
}
```

Listing A.6: Heavy Task Mimicking Function

```cpp
// Function definition for HEAVYTASK
// Below code is used to calculate the solution for Project Euler Problem 23. The
    relevant problem can be found at https://projecteuler.net/problem=23
// This will be used to increase the run time for the thesis work since it would
    roughly take about 400 ms to run this in my computer.

#include <iostream>
#include <math.h>
#include "HEAVYTASK.h"
#include <vector>

// Sub Functions required

int divSum(int num) {
  int sum = 1; // Initialize with 1 (since every number is divisible by 1)
  for (int i = 2; i * i <= num; ++i) {
    if (num % i == 0) {
      sum += i;
      if (i != num / i) {
        sum += num / i;
```

```
      }
    }
  }
  return sum;
}

// Check if a number is abundant
bool is_abundant(int n) {
  return divSum(n) > n;
}

bool HEAVYTASK(bool a) {
  {
    const int limit = 28122; // Limit defined as per the problem - Limit is 28123
    std::vector<bool> is_abundant_number(limit + 1, false);

    // Mark abundant numbers
    for (int i = 12; i <= limit; ++i) {
      if (is_abundant(i)) {
        is_abundant_number[i] = true;
      }
    }

    // Calculate the sum of non-abundant sums
    long long sum = 0;
    for (int i = 1; i <= limit; ++i) {
      bool can_be_expressed_as_sum = false;
      for (int abundant = 12; abundant <= i / 2; ++abundant) {
        if (is_abundant_number[abundant] && is_abundant_number[i - abundant]) {
          can_be_expressed_as_sum = true;
          break;
        }
      }
      if (!can_be_expressed_as_sum) {
        sum += i;
      }
    }
  // Below commented out code part prints the solution to given limit in the
      console.
  // This will not be used in the custom function since nothing needs to be printed
       out.

  std::cout << "Sum of non-abundant sums: " << sum << std::endl;

  }



  return a;
}
```

Listing A.7: Main File Generated for Converter Station Model using Embedded Coder

```
//
// File: ert_main.cpp
//
// Code generated for Simulink model 'CB'.
//
// Model version                  : 1.13
// Simulink Coder version         : 9.9 (R2023a) 19-Nov-2022
// C/C++ source code generated on : Fri Apr  5 09:05:09 2024
//
// 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 <stdio.h>              // This example main program uses printf/fflush
#include "CB.h"                         // Model header file
#include <chrono>
#include <iostream>
#include <memory>

static Switch_Model rtObj;              // Instance of model class

// define variables and structures required for result generation in CSV format - Added by Arjuna

uint8_T mainState;

struct SignalInfo {
    std::string name;
     int ptr;
};

std::vector<SignalInfo> mainStateList; // creation of vector to store the States and Substates

//
// Associating rt_OneStep with a real-time clock or interrupt service routine
// is what makes the generated code "real-time".  The function rt_OneStep is
// always associated with the base rate of the model.  Subrates are managed
// by the base rate from inside the generated code.  Enabling/disabling
// interrupts and floating point context switches are target specific.  This
// example code indicates where these should take place relative to executing
// the generated code step function.  Overrun behavior should be tailored to
// your application needs.  This example simply sets an error status in the
// real-time model and returns from rt_OneStep.
//
void rt_OneStep(void);
void rt_OneStep(void)
{
  static boolean_T OverrunFlag{ false };

  // Disable interrupts here

  // Check for overrun
  if (OverrunFlag) {
    rtmSetErrorStatus(rtObj.getRTM(), "Overrun");
    return;
  }

  OverrunFlag = true;

  // Save FPU context here (if necessary)
  // Re-enable timer or interrupt here
  // Set model inputs here

  // Step the model
  rtObj.step();

  // Get model outputs here

  // Indicate task complete
  OverrunFlag = false;

  // Disable interrupts here
  // Restore FPU context here (if necessary)
  // Enable interrupts here
}
```

```
78
79  //
80  // The example main function illustrates what is required by your
81  // application code to initialize, execute, and terminate the generated code.
82  // Attaching rt_OneStep to a real-time clock is target specific. This example
83  // illustrates how you do this relative to initializing the model.
84  //
85
86
87  int_T main(int_T argc, const char *argv[])
88  {
89    // Unused arguments
90    (void)(argc);
91    (void)(argv);
92
93    // Initialize model
94    rtObj.initialize();
95
96    // Attach rt_OneStep to a timer or interrupt service routine with
97    //  period 0.2 seconds (base rate of the model) here.
98    //  The call syntax for rt_OneStep is
99    //
100   //   rt_OneStep();
101
102
103   // printf("Warning: The simulation will run forever. "
104   //        "Generated ERT main won't simulate model step behavior. "
105   //        "To change this behavior select the 'MAT-file logging' option.\n");
106   // fflush((nullptr));
107   // Added code part to initialize the variables required for result generation -
        Added by Arjuna
108
109   int counter = 0; // Initialize counter
110
111   mainStateList = {
112         {"CLOSED",1},
113         {"CLOSING",2},
114         {"OPENED",3},
115         {"OPENING",4},
116         {"No Active State",0}
117   };
118
119   // The output variables are to be printed to an excel file showing the Active
        State and Substate for each iteration
120
121   std::ofstream csvFile("ResultsSimulinkCBWithHeavyTask.csv"); // Create the CSV
        File
122
123   csvFile << "Iteration,Active State,Time (µs), Time (ms)" << std::endl; // Headers
        of the CSV File
124
125   while (counter < 50) {
126     //  Perform application tasks here
127
128     counter++; // increase the counter value
129
130     //start measuring of time and execute code
131     auto start_time = std::chrono::high_resolution_clock::now();
132     rt_OneStep();
133
134     //End measuring time and assign duration variables
135     auto endTimepoint = std::chrono::high_resolution_clock::now();
136     auto start = std::chrono::time_point_cast<std::chrono::microseconds>(start_time
          ).time_since_epoch().count();
137     auto end = std::chrono::time_point_cast<std::chrono::microseconds>(endTimepoint
          ).time_since_epoch().count();
138
```

```
    auto duration = end - start;
    double ms = duration * 0.001;

    // Result generation to CSV File

    mainState = rtObj.State;


    csvFile << counter << ",";

    for (SignalInfo& info : mainStateList) {

      if (info.ptr == int(mainState)) {
        csvFile << info.name;
        break;
      }
    }
    csvFile << "," << duration << "," << ms << std::endl;
  }

  return 0;
}

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

Listing A.8: C++ Model file generated for the Converter Station Model using Embedded Coder

```
//
// File: CB.cpp
//
// Code generated for Simulink model 'CB'.
//
// Model version                  : 1.13
// Simulink Coder version         : 9.9 (R2023a) 19-Nov-2022
// C/C++ source code generated on : Fri Apr  5 09:05:09 2024
//
// 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 "CB.h"
#include "rtwtypes.h"
#include "HEAVYTASK.h"


// Named constants for Chart: '<Root>/Sequences'
const uint8_T IN_Closed{ 1U };

const uint8_T IN_Closing{ 2U };

const uint8_T IN_Opened{ 3U };

const uint8_T IN_Opening{ 4U };

// System initialize for function-call system: '<S2>/HeavyTask1'
void Switch_Model::HeavyTask1_Init(boolean_T *rty_Out1)
{
  // Start for CFunction: '<S3>/C Function'
```

```
35    *rty_Out1 = false;
36  }
37
38  // Output and update for function-call system: '<S2>/HeavyTask1'
39  void Switch_Model::HeavyTask1(boolean_T *rty_Out1)
40  {
41    // CFunction: '<S3>/C Function'
42    *rty_Out1 = HEAVYTASK(true);
43  }
44
45  // Model step function
46  void Switch_Model::step()
47  {
48    int32_T rtb_switch_status;
49    boolean_T CFunction;
50
51    // Switch: '<Root>/Switch' incorporates:
52    //   UnitDelay: '<Root>/Unit Delay'
53
54    rtb_switch_status = !rtDW.UnitDelay_DSTATE;
55
56    // Chart: '<Root>/Sequences'
57    if (rtDW.is_active_c3_CB == 0U) {
58      rtDW.is_active_c3_CB = 1U;
59      rtDW.is_c3_CB = IN_Closed;
60      rtDW.close_ord = 0.0;
61
62      // Outputs for Function Call SubSystem: '<S2>/HeavyTask1'
63      HeavyTask1(&CFunction);
64
65      // End of Outputs for SubSystem: '<S2>/HeavyTask1'
66    } else {
67      switch (rtDW.is_c3_CB) {
68       case IN_Closed:
69        rtDW.close_ord = 0.0;
70        rtDW.is_c3_CB = IN_Opening;
71        rtDW.open_ord = 1.0;
72
73        // Outputs for Function Call SubSystem: '<S2>/HeavyTask1'
74        HeavyTask1(&CFunction);
75
76        // End of Outputs for SubSystem: '<S2>/HeavyTask1'
77        break;
78
79       case IN_Closing:
80        rtDW.close_ord = 1.0;
81        if (rtb_switch_status == 1) {
82          rtDW.is_c3_CB = IN_Closed;
83          rtDW.close_ord = 0.0;
84
85          // Outputs for Function Call SubSystem: '<S2>/HeavyTask1'
86          HeavyTask1(&CFunction);
87
88          // End of Outputs for SubSystem: '<S2>/HeavyTask1'
89        }
90        break;
91
92       case IN_Opened:
93        rtDW.open_ord = 0.0;
94        rtDW.is_c3_CB = IN_Closing;
95        rtDW.close_ord = 1.0;
96
97        // Outputs for Function Call SubSystem: '<S2>/HeavyTask1'
98        HeavyTask1(&CFunction);
99
100       // End of Outputs for SubSystem: '<S2>/HeavyTask1'
101       break;
```

```
102
103       default:
104         // case IN_Opening:
105         rtDW.open_ord = 1.0;
106         if (rtb_switch_status == 0) {
107           rtDW.is_c3_CB = IN_Opened;
108           rtDW.open_ord = 0.0;
109
110           // Outputs for Function Call SubSystem: '<S2>/HeavyTask1'
111           HeavyTask1(&CFunction);
112
113           // End of Outputs for SubSystem: '<S2>/HeavyTask1'
114         }
115         break;
116       }
117     }
118
119     // End of Chart: '<Root>/Sequences'
120
121     // CombinatorialLogic: '<S1>/Logic' incorporates:
122     //   DataTypeConversion: '<Root>/Cast To Boolean'
123     //   DataTypeConversion: '<Root>/Cast To Boolean1'
124     //   Memory: '<S1>/Memory'
125
126     CFunction = rtConstP.Logic_table[(((static_cast<uint32_T>(rtDW.open_ord != 0.0)
127       << 1) + (rtDW.close_ord != 0.0)) << 1) + rtDW.Memory_PreviousInput];
128
129     // Update for UnitDelay: '<Root>/Unit Delay'
130     rtDW.UnitDelay_DSTATE = CFunction;
131
132     // Update for Memory: '<S1>/Memory'
133     rtDW.Memory_PreviousInput = CFunction;
134
135     State = rtDW.is_c3_CB;
136   }
137
138   // Model initialize function
139   void Switch_Model::initialize()
140   {
141     {
142       boolean_T CFunction;
143
144       // SystemInitialize for Chart: '<Root>/Sequences' incorporates:
145       //   SubSystem: '<S2>/HeavyTask1'
146
147       HeavyTask1_Init(&CFunction);
148     }
149   }
150
151   // Constructor
152   Switch_Model::Switch_Model() :
153     rtDW(),
154     rtM()
155   {
156     // Currently there is no constructor body generated.
157   }
158
159   // Destructor
160   // Currently there is no destructor body generated.
161   Switch_Model::~Switch_Model() = default;
162
163   // Real-Time Model get method
164   Switch_Model::RT_MODEL * Switch_Model::getRTM()
165   {
166     return (&rtM);
167   }
168
```

```
169 //
170 // File trailer for generated code.
171 //
172 // [EOF]
173 //
```

B

Appendix - Detailed Results

Table B.1: Timing Measurements for Circuit Breaker Model implemented in MATLAB

Iteration	Active State	Time (µs)	Time (ms)
1	CLOSED	0	0
2	OPENING	0	0
3	OPENED	0	0
4	CLOSING	0	0
5	CLOSED	0	0
6	OPENING	0	0
7	OPENED	0	0
8	CLOSING	0	0
9	CLOSED	0	0
10	OPENING	0	0
11	OPENED	0	0
12	CLOSING	0	0
13	CLOSED	0	0
14	OPENING	0	0
15	OPENED	0	0
16	CLOSING	0	0
17	CLOSED	0	0
18	OPENING	0	0
19	OPENED	0	0
20	CLOSING	0	0
21	CLOSED	1	0.001
22	OPENING	0	0
23	OPENED	0	0
24	CLOSING	0	0
25	CLOSED	0	0
26	OPENING	0	0
27	OPENED	0	0
28	CLOSING	0	0
29	CLOSED	0	0
30	OPENING	0	0
31	OPENED	1	0.001

Table B.1 continued from previous page

Iteration	Active State	Time (µs)	Time (ms)
32	CLOSING	0	0
33	CLOSED	0	0
34	OPENING	0	0
35	OPENED	0	0
36	CLOSING	0	0
37	CLOSED	0	0
38	OPENING	0	0
39	OPENED	0	0
40	CLOSING	0	0
41	CLOSED	0	0
42	OPENING	0	0
43	OPENED	0	0
44	CLOSING	1	0.001
45	CLOSED	0	0
46	OPENING	0	0
47	OPENED	0	0
48	CLOSING	0	0
49	CLOSED	0	0
50	OPENING	0	0

Table B.2: Timing Measurements for Circuit Breaker Model implemented in HiDraw

Iteration	Active State	Time (µs)	Time (ms)	Other Active State	Remarks
0	CLOSED	0	0		
1	OPENING	0	0		
2	OPENED	0	0		
3	CLOSING	0	0		
4	CLOSED	0	0	CLOSING	Two Main State Active
5	OPENING	0	0		
6	OPENED	0	0		
7	CLOSING	0	0		
8	CLOSED	0	0	CLOSING	Two Main State Active
9	OPENING	0	0		
10	OPENED	0	0		
11	CLOSING	1	0.001		
12	CLOSED	0	0	CLOSING	Two Main State Active
13	OPENING	1	0.001		
14	OPENED	0	0		
15	CLOSING	0	0		
16	CLOSED	0	0	CLOSING	Two Main State Active
17	OPENING	0	0		
18	OPENED	1	0.001		
19	CLOSING	0	0		
20	CLOSED	0	0	CLOSING	Two Main State Active
21	OPENING	0	0		
22	OPENED	0	0		
23	CLOSING	1	0.001		
24	CLOSED	0	0	CLOSING	Two Main State Active
25	OPENING	0	0		
26	OPENED	0	0		
27	CLOSING	0	0		
28	CLOSED	1	0.001	CLOSING	Two Main State Active

Iteration	Active State	Time (µs)	Time (ms)	Other Active State	Remarks
29	OPENING	0	0		
30	OPENED	0	0		
31	CLOSING	0	0		
32	CLOSED	1	0.001	CLOSING	Two Main State Active
33	OPENING	0	0		
34	OPENED	0	0		
35	CLOSING	0	0		
36	CLOSED	0	0	CLOSING	Two Main State Active
37	OPENING	0	0		
38	OPENED	0	0		
39	CLOSING	0	0		
40	CLOSED	0	0	CLOSING	Two Main State Active
41	OPENING	0	0		
42	OPENED	0	0		
43	CLOSING	0	0		
44	CLOSED	0	0	CLOSING	Two Main State Active
45	OPENING	1	0.001		
46	OPENED	1	0.001		
47	CLOSING	0	0		
48	CLOSED	0	0	CLOSING	Two Main State Active
49	OPENING	0	0		
50	OPENED	0	0		

Table B.3: Timing Measurements for Nested States Model implemented in MATLAB Stateflow

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
1	GROUNDED	GROUNDED STATE1	407640	407.64
2	GROUNDED	GROUNDED STATE1	392063	392.06
3	GROUNDED	GROUNDED STATE1	403712	403.71
4	GROUNDED	GROUNDED STATE1	394265	394.27
5	GROUNDED	GROUNDED STATE1	396014	396.01
6	GROUNDED	GROUNDED STATE2	399016	399.02
7	GROUNDED	GROUNDED STATE2	395592	395.59
8	GROUNDED	GROUNDED STATE2	394068	394.07
9	GROUNDED	GROUNDED STATE2	389835	389.84
10	GROUNDED	GROUNDED STATE2	405389	405.39
11	GROUNDED	GROUNDED STATE3	404010	404.01
12	GROUNDED	GROUNDED STATE3	388576	388.58
13	GROUNDED	GROUNDED STATE3	412333	412.33
14	GROUNDED	GROUNDED STATE3	402015	402.02
15	GROUNDED	GROUNDED STATE3	408162	408.16
16	GROUNDED	GROUNDED STATE4	403640	403.64
17	GROUNDED	GROUNDED STATE4	415288	415.29
18	GROUNDED	GROUNDED STATE4	400660	400.66
19	GROUNDED	GROUNDED STATE4	394651	394.65
20	GROUNDED	GROUNDED STATE4	405852	405.85
21	STANDBY	No Active Substate	406792	406.79
22	STANDBY	No Active Substate	409546	409.55
23	STANDBY	No Active Substate	503747	503.75
24	STANDBY	No Active Substate	482338	482.34

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
25	STANDBY	No Active Substate	455019	455.02
26	LIVE	No Active Substate	408729	408.73
27	LIVE	No Active Substate	535253	535.25
28	LIVE	No Active Substate	445676	445.68
29	LIVE	No Active Substate	420850	420.85
30	LIVE	No Active Substate	404800	404.80
31	COUPLED	No Active Substate	478923	478.92
32	COUPLED	No Active Substate	436854	436.85
33	COUPLED	No Active Substate	403010	403.01
34	COUPLED	No Active Substate	492941	492.94
35	COUPLED	No Active Substate	493052	493.05
36	LIVE	No Active Substate	473267	473.27
37	LIVE	No Active Substate	472394	472.39
38	LIVE	No Active Substate	439861	439.86
39	LIVE	No Active Substate	441649	441.65
40	LIVE	No Active Substate	500296	500.30
41	STANDBY	No Active Substate	486044	486.04
42	STANDBY	No Active Substate	450253	450.25
43	STANDBY	No Active Substate	497017	497.02
44	STANDBY	No Active Substate	471909	471.91
45	STANDBY	No Active Substate	488281	488.28
46	GROUNDED	GROUNDED STATE1	448608	448.61
47	GROUNDED	GROUNDED STATE1	425307	425.31
48	GROUNDED	GROUNDED STATE1	480766	480.77
49	GROUNDED	GROUNDED STATE1	474462	474.46
50	GROUNDED	GROUNDED STATE1	497541	497.54
51	GROUNDED	GROUNDED STATE2	495970	495.97
52	GROUNDED	GROUNDED STATE2	474850	474.85
53	GROUNDED	GROUNDED STATE2	538537	538.54
54	GROUNDED	GROUNDED STATE2	412930	412.93
55	GROUNDED	GROUNDED STATE2	449156	449.16
56	GROUNDED	GROUNDED STATE3	475499	475.50
57	GROUNDED	GROUNDED STATE3	497375	497.38
58	GROUNDED	GROUNDED STATE3	486121	486.12
59	GROUNDED	GROUNDED STATE3	427151	427.15
60	GROUNDED	GROUNDED STATE3	430643	430.64
61	GROUNDED	GROUNDED STATE4	442303	442.30
62	GROUNDED	GROUNDED STATE4	513189	513.19
63	GROUNDED	GROUNDED STATE4	465831	465.83
64	GROUNDED	GROUNDED STATE4	441067	441.07
65	GROUNDED	GROUNDED STATE4	424271	424.27
66	STANDBY	No Active Substate	492840	492.84
67	STANDBY	No Active Substate	500640	500.64
68	STANDBY	No Active Substate	481858	481.86

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
69	STANDBY	No Active Substate	457671	457.67
70	STANDBY	No Active Substate	427801	427.80
71	LIVE	No Active Substate	465992	465.99
72	LIVE	No Active Substate	421578	421.58
73	LIVE	No Active Substate	425224	425.22
74	LIVE	No Active Substate	476092	476.09
75	LIVE	No Active Substate	514264	514.26
76	COUPLED	No Active Substate	474005	474.01
77	COUPLED	No Active Substate	486007	486.01
78	COUPLED	No Active Substate	475037	475.04
79	COUPLED	No Active Substate	480286	480.29
80	COUPLED	No Active Substate	474149	474.15
81	LIVE	No Active Substate	446695	446.70
82	LIVE	No Active Substate	409496	409.50
83	LIVE	No Active Substate	473365	473.37
84	LIVE	No Active Substate	473210	473.21
85	LIVE	No Active Substate	433577	433.58
86	STANDBY	No Active Substate	458707	458.71
87	STANDBY	No Active Substate	456120	456.12
88	STANDBY	No Active Substate	434454	434.45
89	STANDBY	No Active Substate	427557	427.56
90	STANDBY	No Active Substate	439147	439.15
91	GROUNDED	GROUNDED STATE1	535162	535.16
92	GROUNDED	GROUNDED STATE1	443917	443.92
93	GROUNDED	GROUNDED STATE1	416046	416.05
94	GROUNDED	GROUNDED STATE1	448689	448.69
95	GROUNDED	GROUNDED STATE1	459463	459.46
96	GROUNDED	GROUNDED STATE2	511079	511.08
97	GROUNDED	GROUNDED STATE2	424230	424.23
98	GROUNDED	GROUNDED STATE2	471933	471.93
99	GROUNDED	GROUNDED STATE2	405064	405.06
100	GROUNDED	GROUNDED STATE2	411012	411.01
101	GROUNDED	GROUNDED STATE3	430652	430.65

Table B.4: Timing Measurements for Nested States Model implemented in HiDraw

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
1	GROUNDED	GROUNDED STATE1	2975643	2975.64
2	GROUNDED	GROUNDED STATE1	2889311	2889.31
3	GROUNDED	GROUNDED STATE1	2818644	2818.64
4	GROUNDED	GROUNDED STATE1	2858019	2858.02
5	GROUNDED	GROUNDED STATE1	2875274	2875.27
6	GROUNDED	GROUNDED STATE2	2842261	2842.26

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
7	GROUNDED	GROUNDED STATE2	3295381	3295.38
8	GROUNDED	GROUNDED STATE2	3074096	3074.10
9	GROUNDED	GROUNDED STATE2	2924302	2924.30
10	GROUNDED	GROUNDED STATE2	3058324	3058.32
11	GROUNDED	GROUNDED STATE3	3209107	3209.11
12	GROUNDED	GROUNDED STATE3	3074969	3074.97
13	GROUNDED	GROUNDED STATE3	3318881	3318.88
14	GROUNDED	GROUNDED STATE3	3190056	3190.06
15	GROUNDED	GROUNDED STATE3	2967725	2967.72
16	GROUNDED	GROUNDED STATE4	2971601	2971.60
17	GROUNDED	GROUNDED STATE4	3030577	3030.58
18	GROUNDED	GROUNDED STATE4	2885931	2885.93
19	GROUNDED	GROUNDED STATE4	2875328	2875.33
20	GROUNDED	GROUNDED STATE4	2941991	2941.99
21	STANDBY	No Substate Active	2952864	2952.86
22	STANDBY	No Substate Active	2933195	2933.20
23	STANDBY	No Substate Active	2838262	2838.26
24	STANDBY	No Substate Active	3285443	3285.44
25	STANDBY	No Substate Active	3268066	3268.07
26	LIVE	No Substate Active	2898931	2898.93
27	LIVE	No Substate Active	2982683	2982.68
28	LIVE	No Substate Active	2975995	2975.99
29	LIVE	No Substate Active	2847698	2847.70
30	LIVE	No Substate Active	2926027	2926.03
31	COUPLED	No Substate Active	2996200	2996.20
32	COUPLED	No Substate Active	3062752	3062.75
33	COUPLED	No Substate Active	2935669	2935.67
34	COUPLED	No Substate Active	2966065	2966.07
35	COUPLED	No Substate Active	3026657	3026.66
36	No Main State Active	No Substate Active	2842429	2842.43
37	LIVE	No Substate Active	2845892	2845.89
38	LIVE	No Substate Active	2811245	2811.24
39	LIVE	No Substate Active	2865835	2865.84
40	LIVE	No Substate Active	3097542	3097.54
41	LIVE	No Substate Active	2841212	2841.21
42	No Main State Active	No Substate Active	2840342	2840.34
43	STANDBY	No Substate Active	2790826	2790.83
44	STANDBY	No Substate Active	2811030	2811.03
45	STANDBY	No Substate Active	2812829	2812.83
46	STANDBY	No Substate Active	2821371	2821.37
47	STANDBY	No Substate Active	2811678	2811.68
48	No Main State Active	No Substate Active	2815339	2815.34
49	GROUNDED	GROUNDED STATE1	2812710	2812.71
50	GROUNDED	GROUNDED STATE1	2809735	2809.74
51	GROUNDED	GROUNDED STATE1	2793730	2793.73
52	GROUNDED	GROUNDED STATE1	2791751	2791.75
53	GROUNDED	GROUNDED STATE1	2789954	2789.95

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
54	GROUNDED	GROUNDED STATE2	2799394	2799.39
55	GROUNDED	GROUNDED STATE2	2806450	2806.45
56	GROUNDED	GROUNDED STATE2	2815309	2815.31
57	GROUNDED	GROUNDED STATE2	2789772	2789.77
58	GROUNDED	GROUNDED STATE2	2813859	2813.86
59	GROUNDED	GROUNDED STATE3	2796581	2796.58
60	GROUNDED	GROUNDED STATE3	2804945	2804.95
61	GROUNDED	GROUNDED STATE3	2840468	2840.47
62	GROUNDED	GROUNDED STATE3	2876134	2876.13
63	GROUNDED	GROUNDED STATE3	2996118	2996.12
64	GROUNDED	GROUNDED STATE4	2919628	2919.63
65	GROUNDED	GROUNDED STATE4	2861912	2861.91
66	GROUNDED	GROUNDED STATE4	2946897	2946.90
67	GROUNDED	GROUNDED STATE4	2857911	2857.91
68	GROUNDED	GROUNDED STATE4	2953763	2953.76
69	STANDBY	No Substate Active	2816147	2816.15
70	STANDBY	No Substate Active	2866953	2866.95
71	STANDBY	No Substate Active	2777098	2777.10
72	STANDBY	No Substate Active	2806227	2806.23
73	STANDBY	No Substate Active	2816699	2816.70
74	LIVE	No Substate Active	2824072	2824.07
75	LIVE	No Substate Active	2914379	2914.38
76	LIVE	No Substate Active	2830079	2830.08
77	LIVE	No Substate Active	2815324	2815.32
78	LIVE	No Substate Active	2814103	2814.10
79	COUPLED	No Substate Active	2847841	2847.84
80	COUPLED	No Substate Active	3091715	3091.72
81	COUPLED	No Substate Active	3153355	3153.36
82	COUPLED	No Substate Active	2804738	2804.74
83	COUPLED	No Substate Active	2967300	2967.30
84	No Main State Active	No Substate Active	2953247	2953.25
85	LIVE	No Substate Active	2926348	2926.35
86	LIVE	No Substate Active	2852178	2852.18
87	LIVE	No Substate Active	2990192	2990.19
88	LIVE	No Substate Active	2920774	2920.77
89	LIVE	No Substate Active	3063035	3063.03
90	No Main State Active	No Substate Active	3131686	3131.69
91	STANDBY	No Substate Active	2902930	2902.93
92	STANDBY	No Substate Active	3226974	3226.97
93	STANDBY	No Substate Active	3185770	3185.77
94	STANDBY	No Substate Active	2937560	2937.56
95	STANDBY	No Substate Active	3003959	3003.96
96	No Main State Active	No Substate Active	3054360	3054.36
97	GROUNDED	GROUNDED STATE1	3008367	3008.37
98	GROUNDED	GROUNDED STATE1	3055675	3055.68
99	GROUNDED	GROUNDED STATE1	2893370	2893.37
100	GROUNDED	GROUNDED STATE1	3085093	3085.09

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
101	GROUNDED	GROUNDED STATE1	3144321	3144.32

Table B.5: Timing Measurements for Completely Nested Model implemented in MATLAB Stateflow

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
1	GROUNDED	Grounded STATE1	396948	396.948
2	GROUNDED	Grounded STATE1	394128	394.128
3	GROUNDED	Grounded STATE1	390087	390.087
4	GROUNDED	Grounded STATE2	383930	383.93
5	GROUNDED	Grounded STATE2	381894	381.894
6	GROUNDED	Grounded STATE2	389548	389.548
7	GROUNDED	Grounded STATE3	401154	401.154
8	GROUNDED	Grounded STATE3	384611	384.611
9	GROUNDED	Grounded STATE3	386238	386.238
10	GROUNDED	Grounded STATE4	399178	399.178
11	GROUNDED	Grounded STATE4	394219	394.219
12	GROUNDED	Grounded STATE4	382126	382.126
13	STANDBY	Standby STATE1	393051	393.051
14	STANDBY	Standby STATE1	390772	390.772
15	STANDBY	Standby STATE1	382019	382.019
16	STANDBY	Standby STATE2	383343	383.343
17	STANDBY	Standby STATE2	386446	386.446
18	STANDBY	Standby STATE2	409522	409.522
19	STANDBY	Standby STATE3	384495	384.495
20	STANDBY	Standby STATE3	387292	387.292
21	STANDBY	Standby STATE3	385637	385.637
22	STANDBY	Standby STATE4	382714	382.714
23	STANDBY	Standby STATE4	390497	390.497
24	STANDBY	Standby STATE4	384114	384.114
25	LIVE	Live STATE1	434920	434.92
26	LIVE	Live STATE1	387511	387.511
27	LIVE	Live STATE1	389059	389.059
28	LIVE	Live STATE2	406638	406.638
29	LIVE	Live STATE2	384185	384.185
30	LIVE	Live STATE2	383746	383.746
31	LIVE	Live STATE3	382663	382.663
32	LIVE	Live STATE3	400685	400.685
33	LIVE	Live STATE3	396824	396.824
34	LIVE	Live STATE4	397332	397.332
35	LIVE	Live STATE4	388943	388.943
36	LIVE	Live STATE4	392310	392.31
37	COUPLED	Coupled STATE1	392149	392.149
38	COUPLED	Coupled STATE1	384591	384.591

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
39	COUPLED	Coupled STATE1	397275	397.275
40	COUPLED	Coupled STATE2	398519	398.519
41	COUPLED	Coupled STATE2	406542	406.542
42	COUPLED	Coupled STATE2	382285	382.285
43	COUPLED	Coupled STATE3	382111	382.111
44	COUPLED	Coupled STATE3	382530	382.53
45	COUPLED	Coupled STATE3	382646	382.646
46	COUPLED	Coupled STATE4	396280	396.28
47	COUPLED	Coupled STATE4	409532	409.532
48	COUPLED	Coupled STATE4	390881	390.881
49	LIVE	Live STATE1	385010	385.01
50	LIVE	Live STATE1	396132	396.132
51	LIVE	Live STATE1	417678	417.678
52	LIVE	Live STATE2	400773	400.773
53	LIVE	Live STATE2	382808	382.808
54	LIVE	Live STATE2	389622	389.622
55	LIVE	Live STATE3	387404	387.404
56	LIVE	Live STATE3	383980	383.98
57	LIVE	Live STATE3	385886	385.886
58	LIVE	Live STATE4	383117	383.117
59	LIVE	Live STATE4	382549	382.549
60	LIVE	Live STATE4	384176	384.176
61	STANDBY	Standby STATE1	388093	388.093
62	STANDBY	Standby STATE1	399879	399.879
63	STANDBY	Standby STATE1	382049	382.049
64	STANDBY	Standby STATE2	381591	381.591
65	STANDBY	Standby STATE2	400851	400.851
66	STANDBY	Standby STATE2	381118	381.118
67	STANDBY	Standby STATE3	383676	383.676
68	STANDBY	Standby STATE3	382588	382.588
69	STANDBY	Standby STATE3	382780	382.78
70	STANDBY	Standby STATE4	384999	384.999
71	STANDBY	Standby STATE4	385837	385.837
72	STANDBY	Standby STATE4	390168	390.168
73	GROUNDED	Grounded STATE1	382595	382.595
74	GROUNDED	Grounded STATE1	402442	402.442
75	GROUNDED	Grounded STATE1	400767	400.767
76	GROUNDED	Grounded STATE2	389404	389.404
77	GROUNDED	Grounded STATE2	381618	381.618
78	GROUNDED	Grounded STATE2	385323	385.323
79	GROUNDED	Grounded STATE3	382172	382.172
80	GROUNDED	Grounded STATE3	383809	383.809
81	GROUNDED	Grounded STATE3	384569	384.569
82	GROUNDED	Grounded STATE4	384373	384.373

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
83	GROUNDED	Grounded STATE4	413128	413.128
84	GROUNDED	Grounded STATE4	393350	393.35
85	STANDBY	Standby STATE1	388210	388.21
86	STANDBY	Standby STATE1	394044	394.044
87	STANDBY	Standby STATE1	389166	389.166
88	STANDBY	Standby STATE2	384791	384.791
89	STANDBY	Standby STATE2	392348	392.348
90	STANDBY	Standby STATE2	388663	388.663
91	STANDBY	Standby STATE3	396097	396.097
92	STANDBY	Standby STATE3	432979	432.979
93	STANDBY	Standby STATE3	444916	444.916
94	STANDBY	Standby STATE4	409890	409.89
95	STANDBY	Standby STATE4	389421	389.421
96	STANDBY	Standby STATE4	397098	397.098
97	LIVE	Live STATE1	391372	391.372
98	LIVE	Live STATE1	408524	408.524
99	LIVE	Live STATE1	388341	388.341
100	LIVE	Live STATE2	392308	392.308

Table B.6: Timing Measurements for Completely Nested Model implemented in HiDraw

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
1	GROUNDED	GROUNDED STATE1	6650633	6650.63
2	GROUNDED	GROUNDED STATE1	6378560	6378.56
3	GROUNDED	GROUNDED STATE1	6537154	6537.15
4	GROUNDED	GROUNDED STATE2	6284979	6284.98
5	GROUNDED	GROUNDED STATE2	6291534	6291.53
6	GROUNDED	GROUNDED STATE2	6326340	6326.34
7	GROUNDED	GROUNDED STATE3	6420411	6420.41
8	GROUNDED	GROUNDED STATE3	6506535	6506.53
9	GROUNDED	GROUNDED STATE3	6335674	6335.67
10	GROUNDED	GROUNDED STATE4	6256502	6256.50
11	GROUNDED	GROUNDED STATE4	6317533	6317.53
12	GROUNDED	GROUNDED STATE4	6376635	6376.64
13	STANDBY	STANDBY STATE1	6573735	6573.74
14	STANDBY	STANDBY STATE1	6258275	6258.28
15	STANDBY	STANDBY STATE1	6276901	6276.90
16	STANDBY	STANDBY STATE2	6292056	6292.06
17	STANDBY	STANDBY STATE2	6271430	6271.43
18	STANDBY	STANDBY STATE2	6346406	6346.41
19	STANDBY	STANDBY STATE3	6300817	6300.82
20	STANDBY	STANDBY STATE3	6287753	6287.75
21	STANDBY	STANDBY STATE3	6303522	6303.52
22	STANDBY	STANDBY STATE4	6258455	6258.45

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
23	STANDBY	STANDBY STATE4	6268247	6268.25
24	STANDBY	STANDBY STATE4	6393694	6393.69
25	LIVE	LIVE STATE1	6255922	6255.92
26	LIVE	LIVE STATE1	6320106	6320.11
27	LIVE	LIVE STATE1	6245688	6245.69
28	LIVE	LIVE STATE2	6247782	6247.78
29	LIVE	LIVE STATE2	6298780	6298.78
30	LIVE	LIVE STATE2	6249336	6249.34
31	LIVE	LIVE STATE3	6274372	6274.37
32	LIVE	LIVE STATE3	6258705	6258.70
33	LIVE	LIVE STATE3	6254532	6254.53
34	LIVE	LIVE STATE4	6286112	6286.11
35	LIVE	LIVE STATE4	6290535	6290.53
36	LIVE	LIVE STATE4	6295242	6295.24
37	COUPLED	COUPLED STATE1	6274537	6274.54
38	COUPLED	COUPLED STATE1	6281766	6281.77
39	COUPLED	COUPLED STATE1	6280085	6280.09
40	COUPLED	COUPLED STATE2	6272423	6272.42
41	COUPLED	COUPLED STATE2	6276240	6276.24
42	COUPLED	COUPLED STATE2	6250633	6250.63
43	COUPLED	COUPLED STATE3	6281191	6281.19
44	COUPLED	COUPLED STATE3	6270380	6270.38
45	COUPLED	COUPLED STATE3	6395951	6395.95
46	COUPLED	COUPLED STATE4	6234197	6234.20
47	COUPLED	COUPLED STATE4	6238502	6238.50
48	COUPLED	COUPLED STATE4	6325062	6325.06
49	No Main State Active	No Substate Active	6284457	6284.46
50	LIVE	LIVE STATE1	6262975	6262.98
51	LIVE	LIVE STATE1	6316054	6316.05
52	LIVE	LIVE STATE1	6262654	6262.65
53	LIVE	LIVE STATE2	6253774	6253.77
54	LIVE	LIVE STATE2	6276457	6276.46
55	LIVE	LIVE STATE2	6260396	6260.40
56	LIVE	LIVE STATE3	6260837	6260.84
57	LIVE	LIVE STATE3	6255458	6255.46
58	LIVE	LIVE STATE3	6314469	6314.47
59	LIVE	LIVE STATE4	6298522	6298.52
60	LIVE	LIVE STATE4	6267080	6267.08
61	LIVE	LIVE STATE4	6579164	6579.16
62	No Main State Active	No Substate Active	6270590	6270.59
63	STANDBY	STANDBY STATE1	6266446	6266.45
64	STANDBY	STANDBY STATE1	6294207	6294.21
65	STANDBY	STANDBY STATE1	6259340	6259.34
66	STANDBY	STANDBY STATE2	6275041	6275.04
67	STANDBY	STANDBY STATE2	6606610	6606.61
68	STANDBY	STANDBY STATE2	6411130	6411.13
69	STANDBY	STANDBY STATE3	6427569	6427.57

Iteration	Active State	Active Substate	Time (µs)	Time (ms)
70	STANDBY	STANDBY STATE3	6523431	6523.43
71	STANDBY	STANDBY STATE3	6272044	6272.04
72	STANDBY	STANDBY STATE4	6282815	6282.82
73	STANDBY	STANDBY STATE4	6305276	6305.28
74	STANDBY	STANDBY STATE4	6254705	6254.70
75	No Main State Active	No Substate Active	6257150	6257.15
76	GROUNDED	GROUNDED STATE1	6287627	6287.63
77	GROUNDED	GROUNDED STATE1	6277672	6277.67
78	GROUNDED	GROUNDED STATE1	6256815	6256.82
79	GROUNDED	GROUNDED STATE2	6435638	6435.64
80	GROUNDED	GROUNDED STATE2	6390503	6390.50
81	GROUNDED	GROUNDED STATE2	6282370	6282.37
82	GROUNDED	GROUNDED STATE3	6322395	6322.40
83	GROUNDED	GROUNDED STATE3	6418477	6418.48
84	GROUNDED	GROUNDED STATE3	6438151	6438.15
85	GROUNDED	GROUNDED STATE4	6279826	6279.83
86	GROUNDED	GROUNDED STATE4	6311955	6311.95
87	GROUNDED	GROUNDED STATE4	6277347	6277.35
88	STANDBY	STANDBY STATE1	6267580	6267.58
89	STANDBY	STANDBY STATE1	6275775	6275.78
90	STANDBY	STANDBY STATE1	6260411	6260.41
91	STANDBY	STANDBY STATE2	6266713	6266.71
92	STANDBY	STANDBY STATE2	6336559	6336.56
93	STANDBY	STANDBY STATE2	6309384	6309.38
94	STANDBY	STANDBY STATE3	6299982	6299.98
95	STANDBY	STANDBY STATE3	6281149	6281.15
96	STANDBY	STANDBY STATE3	6301680	6301.68
97	STANDBY	STANDBY STATE4	6276585	6276.59
98	STANDBY	STANDBY STATE4	6280139	6280.14
99	STANDBY	STANDBY STATE4	6258290	6258.29
100	LIVE	LIVE STATE1	6243945	6243.94

DEPARTMENT OF ELECTRICAL ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY