Safety assurance of high voltage control module in a robotic paint system

Martin Sirevåg
Rabah Saleh Hagag

Supervisors
David Anisi
Yvonne Murray

This Master's Thesis is carried out as a part of the education at the University of Agder and is therefore approved as a part of this education. However, this does not imply that the University answers for the methods that are used or the conclusions that are drawn.

University of Agder, 2020
Faculty of Engineering and Science
Department of Engineering Sciences

Abstract

This thesis is a case study considering the use of formal verification methods as means of verifying the functionality of the High Voltage Controller (HVC) for a robotic paint system. The goal of this thesis is to use model checking, which is an output-driven approach, as a complement to standard testing of control systems by introducing a way to verify specific properties of the model. Formal verification originally comes from embedded hardware systems, and it can be used to verify properties of the model before testing the physical system. In this case study it is used to verify the properties of the software for the system. This approach will ideally help identify potential problems early in the development phase of a control system. In this case study, the development phase has already been completed, which makes this a somewhat unique situation. Two different tools have been used to model and verify the software for the HVC. The first is Simulink Design Verifier by MathWorks. The second is RoboTool, which is a tool developed by the RoboStar group at the University of York. The biggest difference between the tools is that Simulink Design Verifier uses the Simulink framework and therefore has support for input-driven simulation models. That means that the model can be modelled around input values and run a dynamic simulation. Simulink Design Verifier can also be used in higher abstraction with a more output-driven approach, that is analysing the behaviour regardless of the input values. Whereas RoboTool is designed with model checking of robotic systems in mind. It is intended for a higher abstraction and a more output-driven modelling approach. RoboTool is still under development, and the possibility of converting RoboChart models to simulation models, which they call RoboSim, is something RoboStar is currently working on. Not every property was verified in either tool, but combined, every property of interest was verified.

Contents

Abstract **I**

Contents **III**

List of Abbreviations **IV**

1 Introduction **1**

1.1 Application Description 1

1.2 Motivation 4

1.3 Background 4

2 Theory **7**

2.1 Formal Verification and Model Checking 7

2.2 Verification and Standard Verification Techniques 7

2.3 Temporal Logic 12

2.4 Linear Temporal Logic 13

2.5 Computational Tree Logic 14

2.6 Communicating Sequential Processes 15

3 Modelling and Verification Tools **16**

3.1 Verification and Validation in Simulink 16

3.2 RoboTool 22

3.3 FDR4 24

4 Modelling 28

4.1 General Modelling Procedure . . . 28

4.2 Simulink Modelling . . . 29

4.3 RoboTool Modelling . . . 36

5 Formal Verification 44

5.1 Verification in Simulink Design Verifier . . . 44

5.2 Verification in RoboTool and FDR4 . . . 48

6 Results 52

6.1 Simulink Design Verifier Results . . . 52

6.2 FDR4 results . . . 57

6.3 Tool comparison . . . 58

7 Discussion and Conclusion 59

7.1 Conclusion . . . 59

7.2 Future work . . . 60

7.2.1 Stateflow and Simulink Design Verifier . . . 60

7.2.2 RoboTool and FDR4 . . . 61

Bibliography 64

List of Figures 68

List of Tables 69

List of Abbreviations

AC Alternating Current.

AP Atomic Propositions.

CPU Central Processing Unit.

CSP Communicating Sequential Processes.

CTL Computational Tree Logic.

FDR Failures-Divergences Refinement.

FSM Finite State Machine.

HVC High Voltage Controller.

IPS Integrated Paint System.

kV Kilo Volt.

LTL Linear Temporal Logic.

PID Proportional Integral Derivative.

PWM Pulse Width Modulation.

SLDV Simulink Design Verifier.

Chapter 1

Introduction

1.1 Application Description

Electrostatic painting is mainly used in the bulk and mass production industry for coating small objects such as cars and bicycles. Electrostatic painting increases the quality of the coat and the efficiency of the painting by reducing the required amount of paint, decreasing the painting time and making the process a fully automated operation by the help of robots. Nevertheless, the electrostatic painting method has some alarming hazards if the procedure and safety precautions are underestimated. Those hazards can be for instance, sparks that increase the risk of fire in the facility or even deadly electrical shock during operation or maintenance. Both of these hazards must be seriously taken into account. Figure 1.1 shows the robotic paint system.

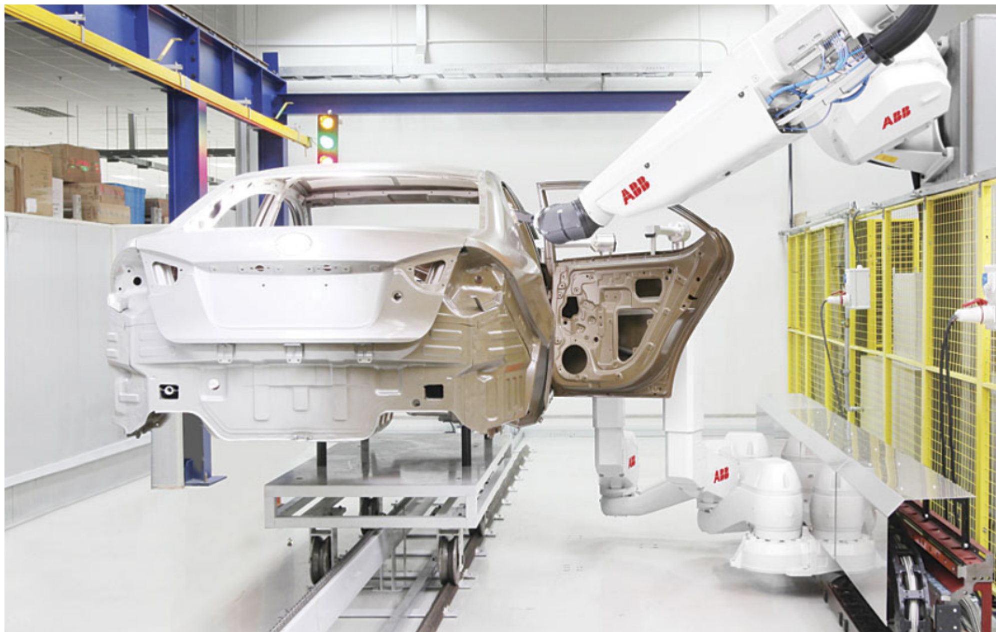


Figure 1.1: Robotic paint system from ABB[21]

1.1.1 Electrostatic Painting Principle

The main difference between the electrostatic method and the conventional coating method is that the conventional method uses pressurized air to apply the coat, while the electrostatic method uses electrostatic fields of 50 to 100 kV. The spray particles are charged by a negative high voltage at the applicator. The particles follow the lines of the electrostatic field from the applicator (cathode) to the earthed object (anode) as shown in Figure 1.2[1].

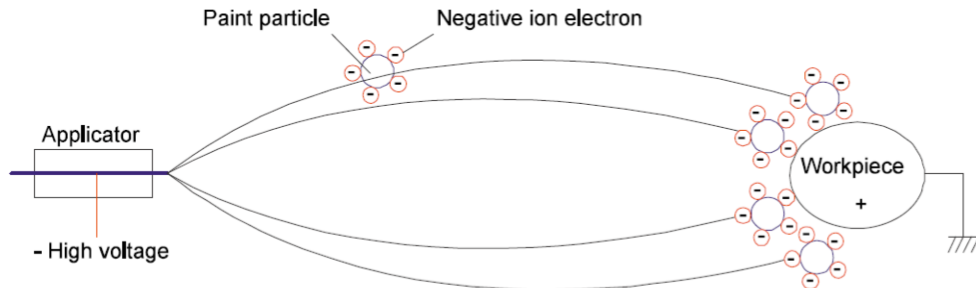


Figure 1.2: Electrostatic painting principle[1]

Several factors have to be monitored in order to guarantee the efficiency and success of the painting process. Some of these are the shape of the work piece, the distance between the applicator and the work piece, the high voltage current at the tip of the applicator and the presence of contamination.

The high voltage current is the main part of the process. By increasing the high voltage current, the effectiveness of the operation and transferable painting go up, and as a result the possibility of discharge sparks rises. If the applicator gets within a certain distance from the work piece, a safety function is activated to make the high voltage current drop to zero in order to ensure personal safety.

The company ABB has produced their own high voltage controller unit (HVC) for supervising, controlling and monitoring the painting process. The next subsection will explain some of its features and functionalities in detail.

1.1.2 High Voltage Control System

A general state machine was developed and provided to the group. Figure 1.3 shows the sequential behaviour of this state machine. There are three additional monitors running concurrently with the system to monitor it, and cause the system to transition to the *ErrorMode* state if the monitor has detected an error.

The high voltage control system is mainly composed of the high voltage controller (HVC-02), the high voltage cascade unit and the applicator. The CCPU is incorporated into the high voltage controller module (HVC-02). The CCPU is a particular CPU that ABB uses, it can be connected to application cards, such as the HVC. Figure 1.4 illustrates the components of the high voltage control system and their relations and work sequence[1].

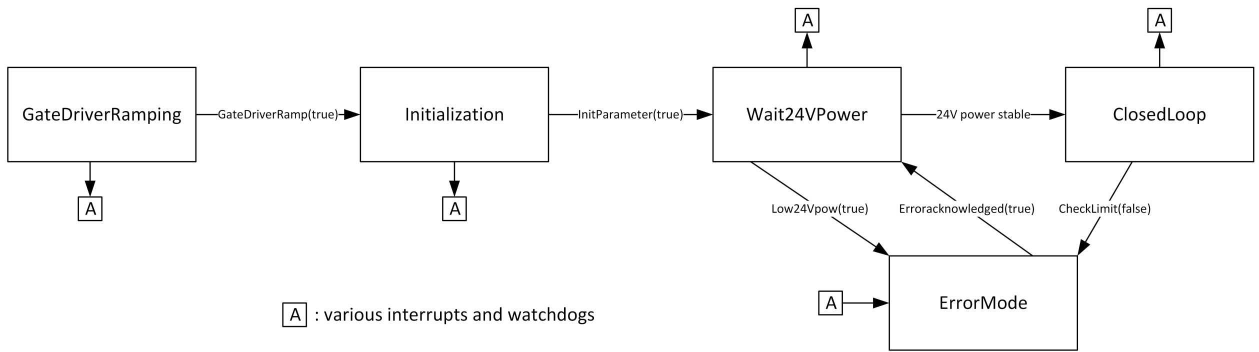


Figure 1.3: Finite state diagram of the High Voltage Controller (HVC)

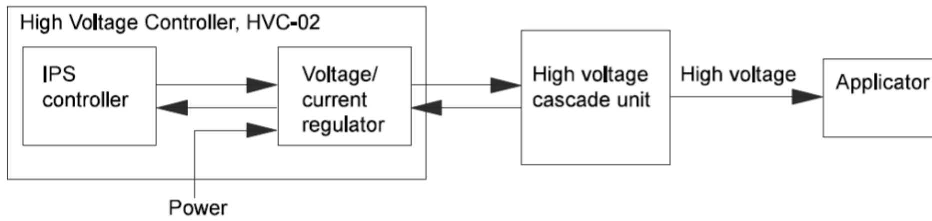


Figure 1.4: High voltage control system unit[1]

The high voltage control module provides the cascade unit with the necessary power for driving the applicator. It generates a 24V Alternating Current (AC) signal that goes into the transformer of the cascade, which in turn supplies a rectified high voltage to the applicator.

The required value of the high voltage and the other reference parameters are set up in the Integrated Paint System (IPS) configuration files. The HVC-02 module performs closed loop control by monitoring the output of the cascade unit, and feeds it back from the cascade unit to the IPS controller in the HVC-02 module. The HVC-cascade interface is shown in Figure 1.5.

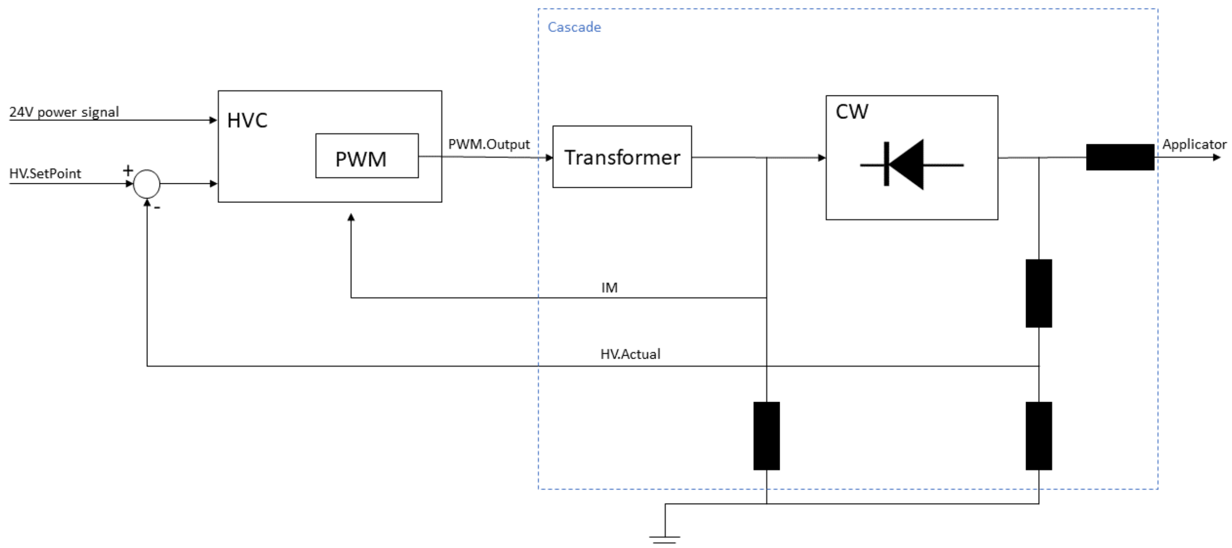


Figure 1.5: Block diagram of one part of the paint robot, containing the HVC[1]

1.2 Motivation

The main motivation in this thesis is to do a comparison between formal verification and standard software testing. Therefore, this thesis is a case study, where formal verification methods are applied to the HVC of a robotic paint robot, where traditional testing methods have already been applied. ABB provided the group with the use case, including documents and datasheets describing the system, as well as detailed explanations of the errors. When using model checking methods, it is important to know which errors could occur. Formal verification does not only require a model that accurately captures the behaviour of the system, but also well defined properties to look for within the model.

Many systems in the industrial market can be quite complex. Using traditional testing methods can be a challenge, since it can be hard to know every scenario to test for. Formal verification methods uses output-driven models[13], this concept is further discussed in Chapter 4.1. When verifying such a model, a property must first be formally described by the programmer. Running the analysis will verify whether the described property is satisfied during every possible execution of the system. Formal verification methods require an accurate representation of the system to be verified. If the model does not accurately capture the system, the results will be unreliable. The properties to verify also have to be well defined. Knowledge and experience about the system to be verified is required in order to know which situations might occur. During this case study there is an advantage by the fact that the errors revealed during operation of the system has been provided, and are well defined. It can still be a challenge to translate informally described errors into formal specifications required by the verification tools.

Multiple tools and languages are available for model checking and formal verification. Another motivating factor in this thesis is to test different tools, and to see the difference in the capabilities of the tools and how they are used. In this project, Stateflow chart from Simulink and RoboChart will be used for modelling purposes. RoboChart uses FDR for verification, while Stateflow uses Simulink Design Verifier (SLDV). There are also other languages and tools for formal verification, such as finite automata, but this thesis will mostly focus on the ones used for the case study.

1.3 Background

In this case study, the goal is to check the reliability of the controller software for the high voltage control unit. The investigation will be carried out by Simulink, validating and reporting its ability for such tasks is part of the study. In parallel, the same verification will be done in RoboTool. The report will also do comparisons between the flexibility and usability of both these model checking tools.

Figure 1.5 shows a simplified block diagram of the painting system, containing the HVC module. The 24V power signal powers the HVC module. *HV.Actual* is a high voltage signal measured from the system, and is used as a reference value. *HV.Actual* should follow the value of *HV.SetPoint*. *IM* is a measurement of the current and is used to compare to internally defined limits in order to detect errors regarding over or under-current limits.

A few errors went undetected by traditional testing methods, but was later discovered during operation. These errors were identified and documented both in text and visualized as graphs. Four issues were detected, the first three regarding *HV.Actual*. The fourth issue is regarding the 24V power signal missing.

The issues regarding the *HV.Actual* were as follows:

1. Sometimes both the *HV.Actual* and *HV.SetPoint* had non-zero values that differed from each other. In this case the HVC did not respond to changes of the *HV.SetPoint* value, and had a constant actual value.

2. In some cases *HV.SetPoint* had no value, but *HV.Actual* still had a non-zero value.

3. In some cases, there was a non-zero *HV.SetPoint*, but the *HV.Actual* continued to be zero.

Regarding the 24V power signal, there was one case discovered:

4. Sometimes the system would report that the 24V power signal was missing, even if it was present, resulting in a deadlock.

Figures 1.6 - 1.8 demonstrates these issues graphically. The issues regarding *HV.Actual* occurred if the 24V power signal was turned off while *HV.SetPoint* had a non-zero value. When this happened, *HV.SetPoint* would be turned off due to the missing power signal, and *HV.Actual* would drop to zero. However, the Pulse Width Modulation (PWM) output, which drives the cascade, was not turned off. Because of this, when the 24V power signal was turned back on, the PWM signal that was still on would feed the cascade and the voltage would increase, which makes the integrator term of the PID controller to increase causing a windup effect. This can be seen in Figure 1.6 when the 24V power is switched back on. The blue line indicates the *HV.Actual*, this rise was caught by a limit alarm which disabled the PWM.

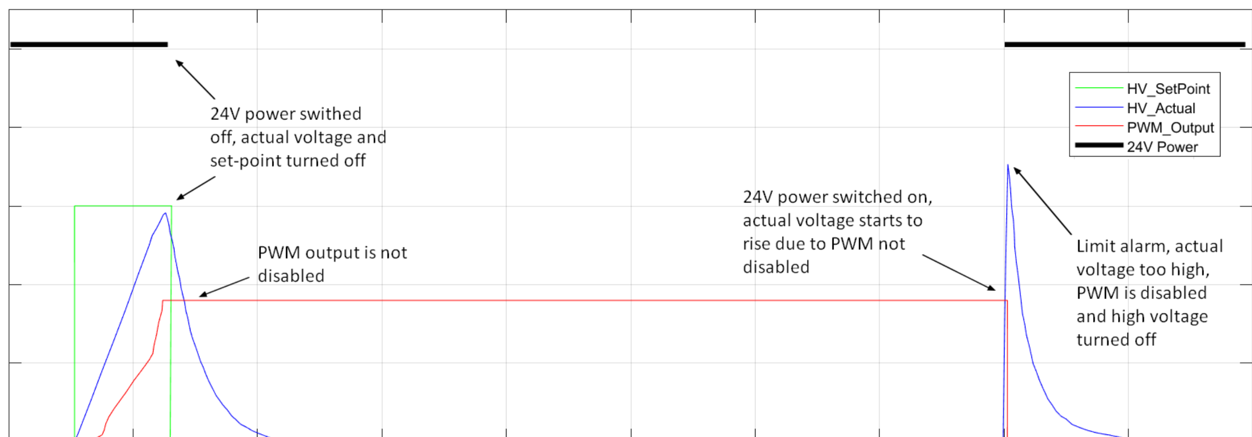


Figure 1.6: Error concerning *HV.Actual* value

An additional error would sometimes occur where the HVC froze when the 24V power was disabled, which would result in the alarm not catching the rising voltage. This would cause the voltage to continue to increase until the system was reset or the power was turned off. This is shown in Figure 1.7.

The issue where the 24V power signal was indicated as missing was due to a deadlock in the system

which caused the controller to report the signal as missing. This can be seen in Figure 1.8.

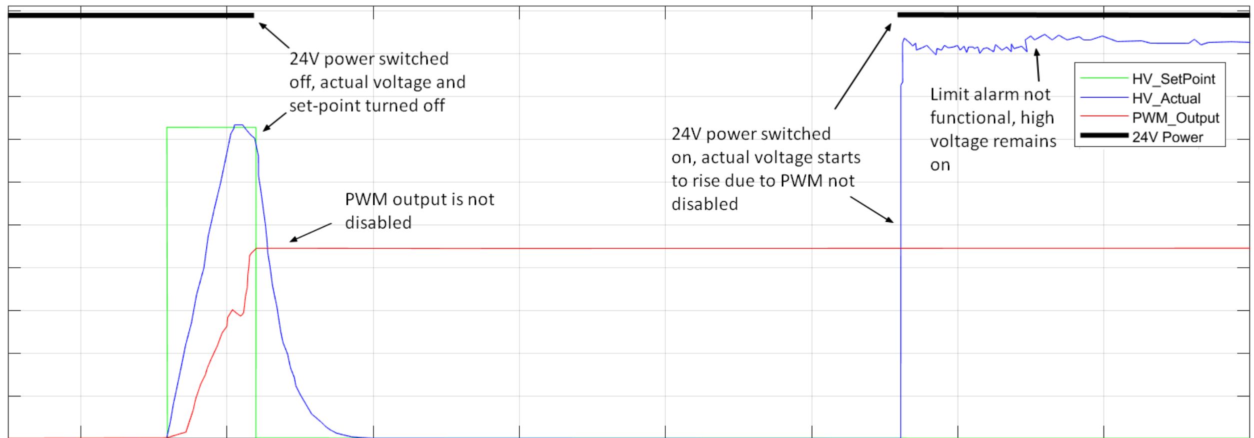


Figure 1.7: Error when the 24V power signal failed and the HVC froze

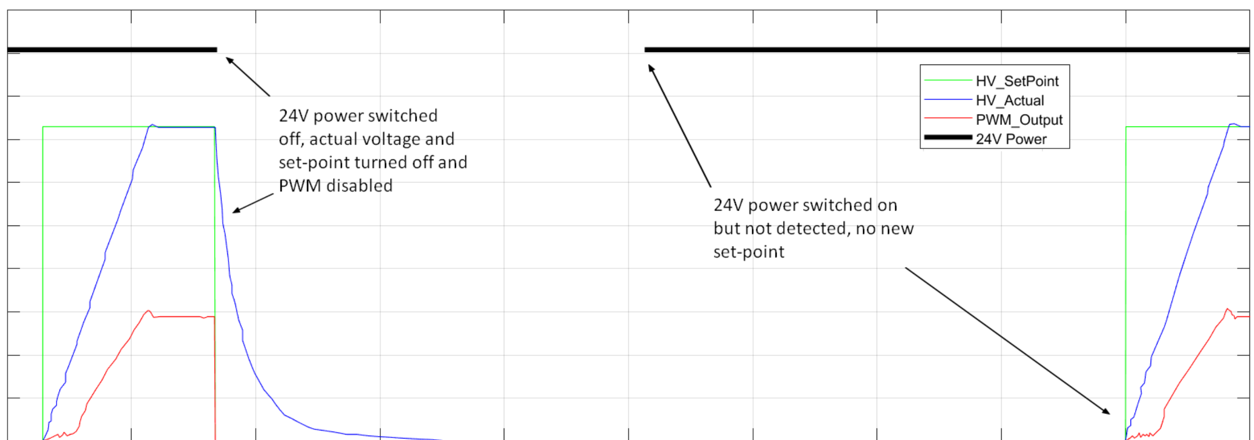


Figure 1.8: Error when the 24V power signal was falsely reported missing

Chapter 2

Theory

This chapter will give an overview of the central theory surrounding model checking and formal verification. Some of the theory in this chapter have not been directly applied in the work, but are still crucial in order to get a better understanding of the principles of model checking and can be considered a bridge to the theory that has been directly applied.

2.1 Formal Verification and Model Checking

The need for formal verification stem from the complexity of systems that are being designed for the industry. The increasing interactions between humans and cyber-physical systems imposes disciplined standards, processes and procedures during development and manufacturing of such products. The main goals of the disciplined rules are to prevent loss of lives and economic loss.

In 1995, statistics mentioned that people encounter about 25 ICT devices on a daily basis[3]. However, what is still really limiting our ability to exploit these systems and design them is our insufficient confidence in their correctness under all circumstances. Therefore, formal verification becomes increasingly significant in designing complex systems, particularly when it comes to safety critical systems[4].

2.2 Verification and Standard Verification Techniques

To make the definition of verification understandable and easily digestible, it can be formulated as one simple question. Is the product built correctly? From this question, it can be inferred that verification is meant to verify specifications of the model for the system. Hence, verification is to verify certain properties for the targeted design and thus the correctness of the system. The specification of targeted design represents the properties in formal verification which must be verified. In a case where all properties are met, the targeted design will be considered a success and the correctness of the complex system has been verified. Figure 2.1 demonstrates the systematic approach for the verification and validation process in industrial products.

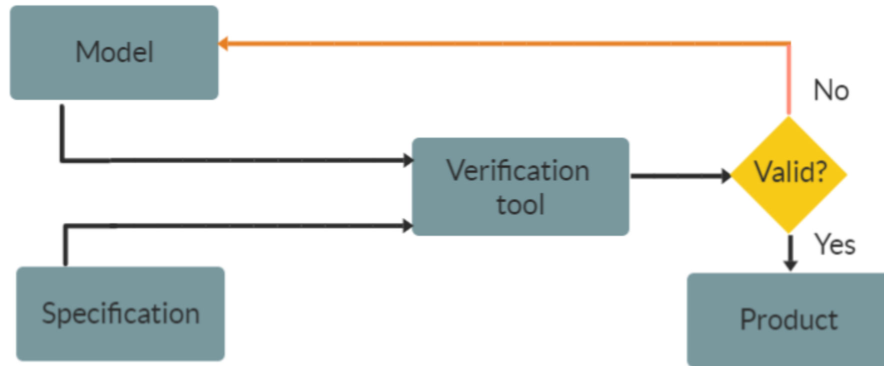


Figure 2.1: Flowchart showing the verification process

The principal validation techniques for complex systems are simulation, testing and formal verification that contains model checking and deductive verification[3]. Simulation and testing are performed by feeding inputs into the system and observing the corresponding output. Though these methods are effective particularly in the early stages of development, finding more complex errors can be a challenge. Only relying on input-driven testing methods may result in errors or bugs going undetected, especially when it comes to highly complex systems. Some techniques are well-known in certain fields, for example peer review is used in software verification, while simulation is mostly used for hardware verification[3].

The definition of formal verification is to verify the correctness of the system by testing certain properties that describe the system behaviour through an abstract mathematical model[4]. This verification approach is conducted through an exhaustive exploration of all possible executions of the complex system[3]. While simulation and testing techniques consider specific scenarios or behaviors, the formal verification technique can be seen as a comprehensive verifying method for covering all behaviours of the system.

The focus in the next chapters will be related to the formal verification procedure, particularly with regards to model checking. Several steps have to be taken in order to carry out the formal verification process, such as defining the formal model and stating the formal properties.

Stating the formal specifications is the initial step before performing any verification. It is vital to accurately understand the engineering requirements and specifications for the targeted system. This understanding is very helpful during conversion of the design into an abstract model to use by the model checking tools. Accordingly, this technique is called the model based verification technique[3]. In view of what has been introduced, the model has to be representative of the system in order for the automatic verification results to be valid. It is necessary to know which properties are important and which are not. Simplification of the model plays a significant key role in the verification process, since any irrelevant information should be simplified[4].

"Any verification using model-based techniques is only as good as the model of the system"[3].

2.2.1 Modelling Systems

This thesis is devoted to embedded systems. Such systems can be modelled by for instance a finite state machine. Finite state machines (FSM) are the preferred structure to model systems for model checking. The essential idea behind FSM is to model the system according to its internal states. The internal states specify the system status that can be held for a period of time and transition to another internal state under certain conditions. One of the simplest examples of a state machine is built up with two states, *on* and *off*. In this simple system, transitions from one state to the other happens when the transition condition occurs, which is when the power is switched on or off.

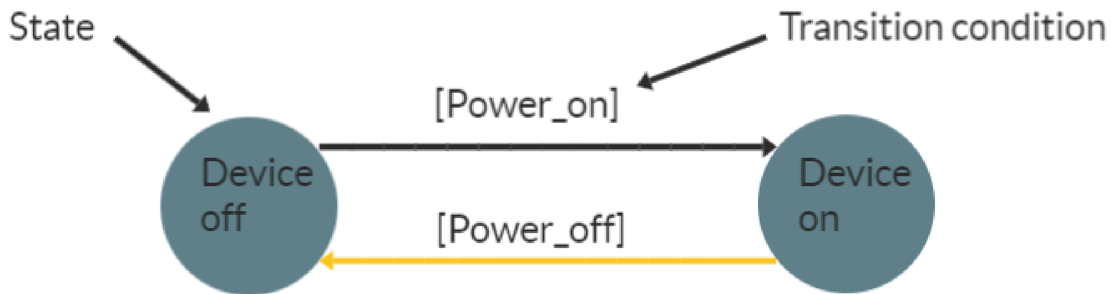


Figure 2.2: Finite state machine representation

2.2.2 Modelling Programs

Finite state machines can be used to model both software and hardware systems. Software can be modelled differently according to their structure whether they are sequential or concurrent programs. Sequential and concurrent models constitute the base for any system modelling. In sequential models, as shown in Figure 2.3, each state has a unique entry point and a unique exit point. States are arranged in a linear order, each state is preceded and followed by another state sequentially. Sequential models are usually used for describing a simple logical program. Only one state can be active at any given time and the rest are inactive. Sequential processes are always terminated by what is called a terminal state[3].

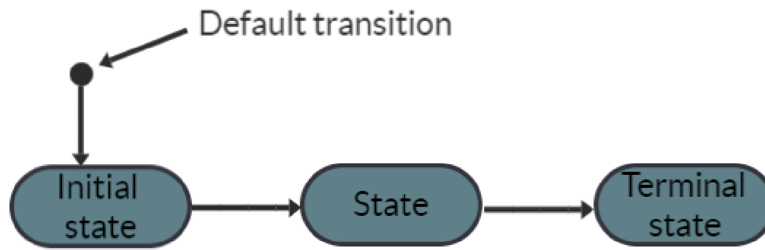


Figure 2.3: Representation of sequential states

Concurrent programs are composed of several sequential processes that can be executed individually at the same time. In asynchronous executions, one process is executed at a time, while in synchronous executions all processes executes at the same time. Processes communicate by different means such as using shared variables or by exchanging action messages between the processes, known as message passing[3].

2.2.3 Model Checking

Model checking is one of the formal verification techniques. It verifies finite state machines for sequential or concurrent systems. Model checking is an exhaustive procedure for systematically and automatically searching the entire state machine to verify if a certain property is satisfied. Even though the model checking procedure is constrained to finite states, model checking is applicable to important systems such as hardware controllers and other communication protocols[4]. Model checking benefits from being an automatic procedure, so it is preferred to be conducted whenever it can be applied[3]. In some cases where systems are infinite, the model checking technique can be used in combination with other techniques such as deductive verification[14]. Other limitations are processors and memory. The available model checkers can process a state space of 10^8 - 10^9 states, however larger state spaces can be treated by implementing algorithms with a reasonable efficiency[3].

The model of the system in model checking is an abstract, high level model, it is not a detailed model of the actual components of the system. This is an important aspect of formal verification. Converting the system into a representative abstract model is an important aspect and can be a challenge. Additionally, properties must be stated precisely and unambiguously.

Safety properties and fairness are among the properties that can be examined by model checking tools. If any violation occurs within any execution path, the model checker tool provides a counter example. The counter example shows how the property is violated and the execution path from the initial state to the state that violates the property is highlighted[3].

2.2.4 Model Checking Process

Using model checking requires a systematic process to ensure the accuracy of the verification. The model checking process evolves through organized steps. It begins with setting up the model for the system, stating the properties formally, running the model checker and analysing the results.

1. Modelling the System

The first step in the model checking process is to model the system in a way that the model checker tool can handle, it is simply a compilation task. Due to constraints on the computational power of the model checker, the possibility of eliminating or ignoring irrelevant details should be considered. Simulation can be used before model checking in order to enhance the quality of the model checking by detecting trivial or simpler errors. Discovering these errors early can save time and effort for model checking [4]. However, formal verification methods are usually applied before simulation, to rule out more high level errors.

2. Specification and Properties Definition

The model represents the behaviour of the system, the properties describes intended behaviours that should be verified within the system. Clear definitions of the properties is crucial to the model checking process. These properties are usually specified by using temporal logic, which is called a property specification language. The system specifications can be formally written with temporal logic to form different properties such as fairness properties, reachability properties and safety properties[3].

3. Verification

Verification is to verify whether the specifications are met for the abstract model of the system. The results are issued by the model checker tools in different forms depending on the tool in question.

4. Analysing the Result

Verification of the model can lead to three outcomes. Whether the property is valid, not valid or if the model of the system is beyond the computational power of the model checker. A valid result means the system satisfies the desired property. Discrepancies between the model and the actual system will cause inaccurate results. Insufficient memory due to the model being too complex leads to an error because of limitations on the model checker's computational power[3]. Figure 2.4 demonstrates the model checking process.

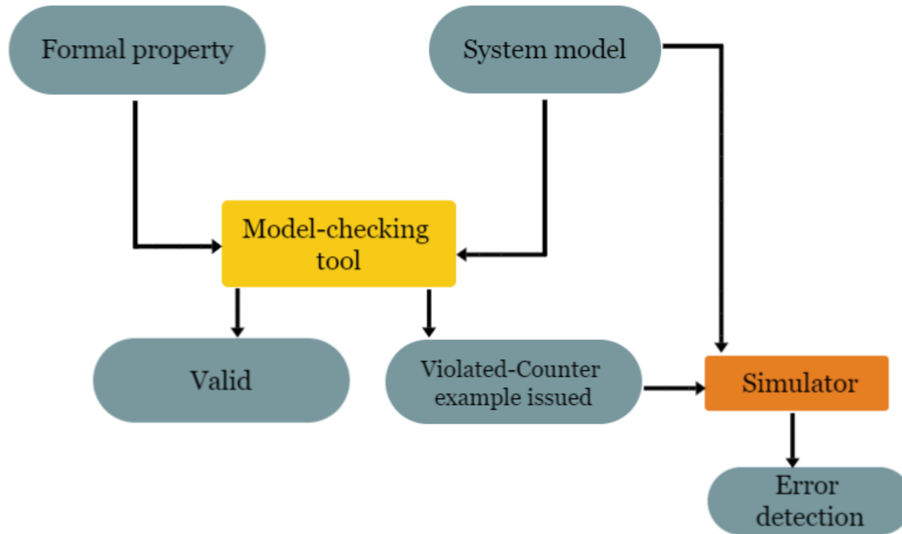


Figure 2.4: Representation of a general model checking procedure additionally showing how it is a complement to simulation

2.3 Temporal Logic

Temporal logic is particularly important for concurrent or reactive systems. The importance stems from the need to describe event order in time without mentioning time explicitly. Consequently, temporal logic is classified as two types. The first is linear temporal logic and the second is branching temporal logic, also called computational tree logic. In the former, time can be seen as if every present moment has one unique possible moment in the future. In the latter, the recent moments are split into a tree branch pattern (branching) into many possible moments in the future. Several temporal logic operators are used such as G, U, and X, these are explained in Sections 2.4 and 2.5. However, the G operator is the one most frequently used. The linear logic formula consists of a set of Atomic Propositions and the temporal logic operators previously mentioned. For instance, G(p) where p is a specific property that has been defined by using a set of atomic propositions[4]. Atomic Proposition (AP) is a statement or assertion that must be true or false.

Quantifiers such as universal and existential are used in combination with linear temporal logic in order to formalize branching temporal logic. For instance A(p), where A represents the universal quantifier and p represents the process. The concept of temporal logic was introduced a long time ago by several people, but Amir Pnueli was the first who used temporal logic in reasoning about concurrency[3]. In the 1980's[4], Clarke and Emerson have introduced the temporal logic model checking algorithm which allowed temporal logic model checking to be automated.

2.4 Linear Temporal Logic

Linear temporal logic (LTL), as the name suggests, is used for analysing linear structures. These can simply be a linear structure on their own, or a linear unwrapping of a tree structure or state machine.

LTL Grammar of Atomic Propositions (AP):

$$\phi := \text{true} \mid p_i \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid X\phi \mid \phi_1 \cup \phi_2 \tag{2.1}$$

ϕ contains all of the atomic propositions of LTL. The first expression "true", is always true. p_i is true if the first state in the linear model contains p_i, $p_i \in$ AP. $\phi_1 \wedge \phi_2$ is true if both ϕ_1 and ϕ_2 is true in the first state, ϕ_1 and ϕ_2 are both LTL formulas. $\neg\phi$ means not ϕ, which means that it this expression is true if the first state does not contain ϕ. Xϕ is true if the next state contains ϕ. The last expression $\phi_1 \cup \phi_2$ is true if ϕ_1 is true until ϕ_2 becomes true[24]. These expressions can be combined in order to make more complex properties. The expressions containing ϕ, ϕ_1 or ϕ_2 can be exchanged for any of the expressions within ϕ. For instance, $\phi_1 \wedge \phi_2$ can be used to create the expression $p_1 \wedge p_2$, where as mentioned before p_i is an atomic proposition. Some examples of linear executions that satisfy certain LTL properties can be seen in Figure 2.5.

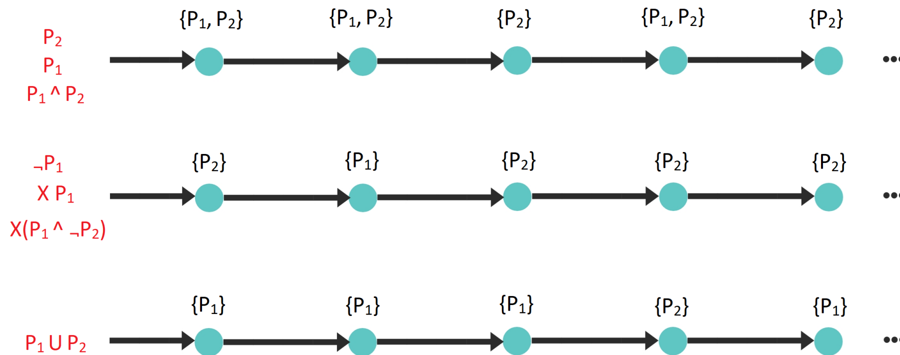


Figure 2.5: Figure showing examples of linear executions that satisfies the requirements described on the left[11]

Most of the previous expressions consider the linear execution from the point of view of the first state. They can however be used used to describe more global properties by combining some of the grammar into one statement. For instance, "true $\cup$ p_1" means true until p_1 is true. Which means that at some point in time p_1 will become true. This can be rephrased as "p_1 is eventually true in the future". Another property that can be derived is "$\neg$(true $\cup$ $\neg p_1$)". This expression is a bit more complex, but by analysing parts of the expression individually makes it more intuitive. The expression inside the parenthesis means true until not p_1. By itself, this is true if p_1 is never true.

The negation before the parenthesis means that the meaning of the expression is opposite. Which means that the whole expression means that p_1 is always true. These two combinations have their own operators[11].

Elementary temporal logic operators:

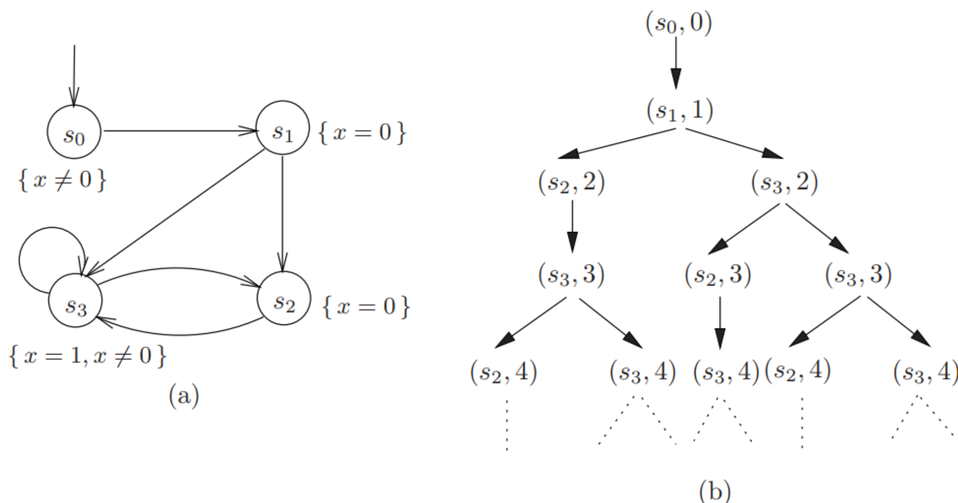
$$F/\diamond \text{ "eventually" - eventually true in the future} \tag{2.2}$$

$$G/\Box \text{ "always" - true now and in the future} \tag{2.3}$$

Whether F or $\diamond$, or G or $\Box$ is used depends on the source. In the report G and F will be used to describe these operators. G and F can also be used together. For instance GFp_1 means that "p_1 will always eventually happen" which means that p_1 will appear "infinitely often" throughout the sequence[11].

2.5 Computational Tree Logic

Computational tree logic (CTL) is used on systems that can be expressed in the form of a computational tree. A computational tree consists of several possible executions, as opposed to a linear execution which only contains one. State machines with several possible executions can be unwrapped into a computational tree. An example of this is shown in Figure 2.6, where a state machine is unwrapped into an infinite computational tree. Particular branches of the computational tree can be isolated and viewed as a single linear execution[3].



(a) A transition system and (b) a prefix of its infinite computation tree

Figure 2.6: Example of a computational tree extracted from a state machine[3]

The temporal logic operators F and G described in chapter 2.4, can also be used in CTL[3]. CTL also distinguishes between between state formulae and path formulae[7].

State formulae:

$$\phi := \text{true} \mid p_i \mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid E\alpha \mid A\alpha \tag{2.4}$$

Path formulae:

$$\alpha := X\phi_1 \mid \phi_1 \cup \phi_2 \mid F\phi_1 \mid G\phi_1 \tag{2.5}$$

Some of the expressions are similar to the ones explained in Chapter 2.4, and have the same meaning. In the state formulae, there are two new operators E and A. Eα means that there exists a path, where α holds true. Aα means that, in all paths, α is true. The α after the expressions E and A in the state formulae, means that α can only be an expression from the path formulae. Likewise, the ϕ in the path formulae must be an expression from the state formulae[7].

2.6 Communicating Sequential Processes

Communicating Sequential Processes (CSP) was first introduced by C.A.R. Hoare, in his book "Communicating sequential processes"[5]. CSP is a mathematical language for describing concurrent systems that communicate by message passing. It is important to understand the distinction between concurrency and parallelism. In parallelism, the processes work simultaneously, this can be achieved by using multiple processors or processor cores. While concurrent systems work concurrently. Concurrent systems are not parallel, the sequential processes may be executed in any order, and the processor might jump between actions in the processes. Such systems can become quite complex and introduce situations such as non-determinism, deadlocks or livelocks. This can make analysis of such a system challenging. The CSP notation can be used to ease the analysis of such systems.

Due to CSP having a basis in mathematical notation, it is a robust notation. In Hoare's book "Communicating sequential processes"[5], mathematical proofs are used to prove the validity of the language. The math behind this language is the basis for the functional language CSP_M used by the verification tool for the RoboTool model. This functional language is further described in Section 3.3.1.

Chapter 3

Modelling and Verification Tools

This chapter will go through the capabilities of the different tools in greater detail. Some of the differences between the tools will become apparent in this chapter. The first part of the chapter will discuss Simulink, Stateflow and Simulink Design Verifier. The second part will go through RoboTool and FDR4. The modelling frameworks are built with different purposes in mind, this will become apparent in this chapter. Simulink is a tool mainly focused on models for simulation, whereas RoboChart models are designed with formal verification of robotic systems in mind.

3.1 Verification and Validation in Simulink

MathWorks uses model-based design to verify and validate the Simulink models by using a variety of toolboxes such as Simulink Design Verifier, Simulink coverage and Polyspace. Using these toolboxes increases the confidence in the design and reduces the time needed for debugging the design[25]. Simulink Design Verifier statically analyses the properties and discovers whether the model satisfies the requirements or not. Simulink coverage shows how much of the model that can be analysed by the verification tool, and displays it as a percentage. Polyspace debugs the generated code to catch the run-time errors and static errors. The formal verification process starts by defining the system requirements and then building a model. The model is verified according to the specification. If all properties are met, the model is considered successfully built. Figure 3.1 demonstrates the formal verification and validation process in MathWorks[15].

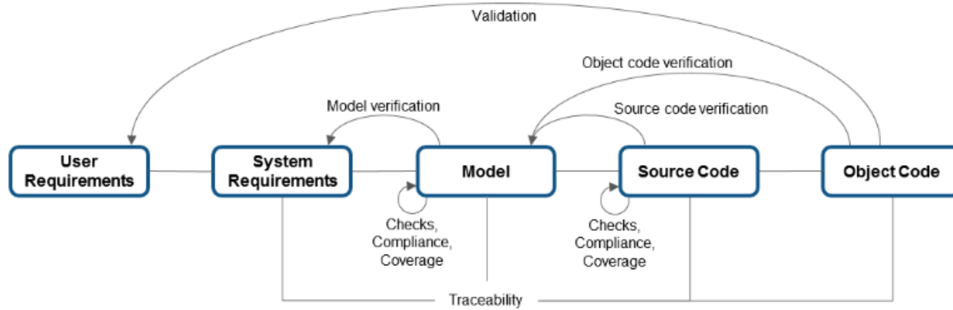


Figure 3.1: Formal verification process in Mathworks[15]

3.1.1 Simulink

Simulink is a MathWorks product and add-in product to MATLAB. It is an interactive tool to simulate, model and analyse dynamic systems. The environment of Simulink has different blocks that can be added via drag and drop to the Simulink canvas. Blocks are graphically represented to ease the work in Simulink. Primitive blocks and predefined library blocks are incorporated in the Simulink library browser. Customizable blocks and predefined user functions are included in Simulink as well. Results from Simulink or inputs from the MATLAB environment can be transferred interchangeably. A graphical editor and solvers (discrete-and continuous) are provided. Parameters of predefined blocks and sample time of model and blocks can be modified according to the user's requirements or preferences. It is widely used in design of systems, especially reactive systems which is the issue of interest in this case study. Code generation, system verification and simulation are part of Simulink's capabilities[12].

3.1.2 Stateflow

Stateflow is a graphical language integrated in the environment of Simulink. It is utilized to represent the control logic of reactive systems via finite state machines. State transitions, transition conditions, and temporal logic can all be expressed graphically in Stateflow. It facilitates the control logic in models of high abstraction, where the functionality, deterministic supervisory control and task scheduling are emphasized. Additionally, it has access to Simulink functions to represent the whole system of continuous and discrete combination. Figure 3.2 illustrates the use of Stateflow to represent the logic of switching the power of a device on or off, expressed as a state machine.

Many features and traits are incorporated into Stateflow, such as graphical error debugging during execution, inconsistency in states and logic, and static and run time analysis. Stateflow has the ability to generate code in the C language from its flowchart model. This code can run in another application that the designer might try to build, such as embedded systems [18].

In the concurrent system where the cyclic fashion always occurs, sharing of resources usually happens. Stateflow in Simulink can not deal with sharing resources. Stateflow follows the 12 o'clock rules and therefore if there are 2 states sharing the same resource, the first state checked is the one closest to the 12 position of the clock. As a result, Stateflow is unable to handle non-deterministic

states[18].

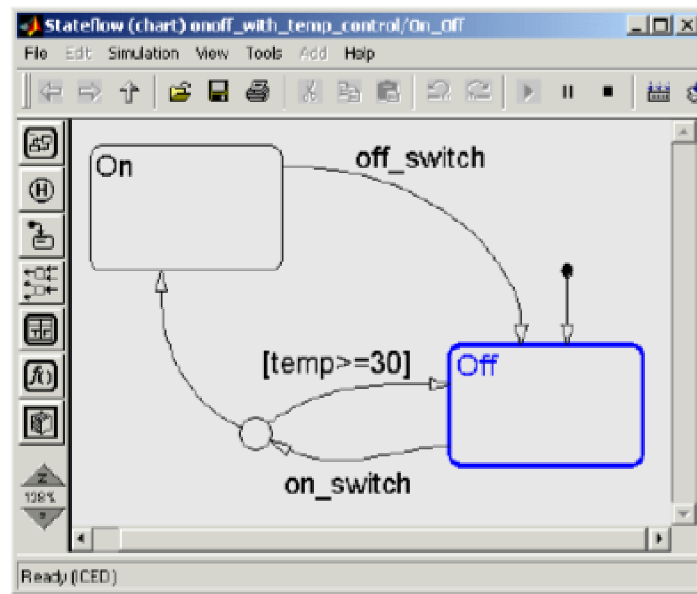


Figure 3.2: Representation of the Stateflow environment[18]

3.1.3 Simulink Desing Verifier

Simulink Design Verifier (SLDV) is a commercial toolbox produced by MathWorks with support for Simulink models. It utilizes formal methods to identify errors in the model without running a simulation. Simulink Design Verifier identifies design errors, generates test cases for model coverage, and verifies the design against the requirements. Figure 3.3 shows the functionality of SLDV. Detectable errors by SLDV include division by zero, integer overflow, dead logic and assertion violations[17].

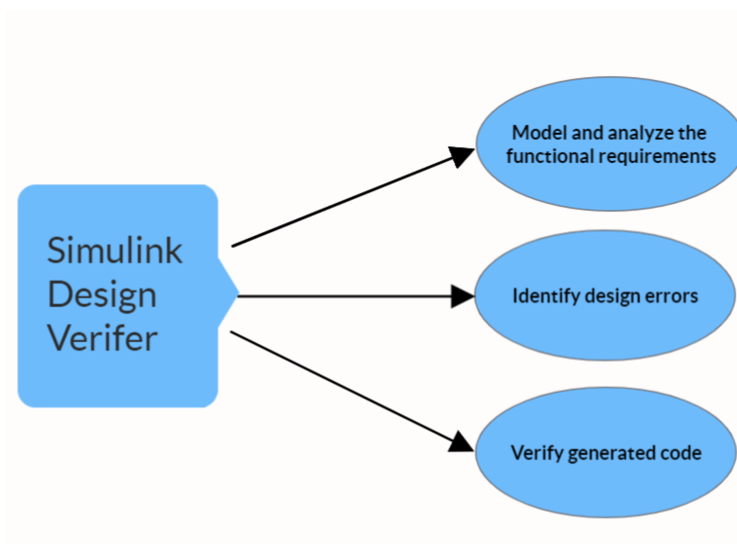


Figure 3.3: Simulink Design Verifer functions

The user can specify the properties in SLDV by using the verification sub-system block from the Simulink library. In that block, the assertions can be written to be utilized in the verification

analysis. The input and output signals from the model are connected to the verification subsystem. Block connections into the model's input and output does not affect the simulation result or code generation. Figure 3.4 shows the architecture of the verification subsystem block and the Simulink sub-model.

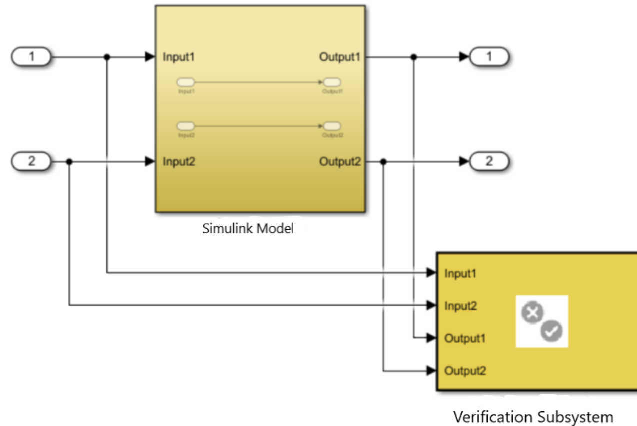


Figure 3.4: Demonstration of the architecture of the verification subsystem block and Simulink sub-model

Error Identification in Simulink Design Verifier

SLDV analyses the model based on mathematical equations. For that reason, simulation is not required. The analysis process checks all possible combinations of scenarios under all possible circumstances, and verifies whether the model meets the requirements. It points out the logical errors and inefficiencies early, therefore the requirements can be refined, errors can be detected and the design can be verified[25].

Error Mode in the Simulink Design Verfier

When the error detection mode is chosen, SLDV generates a progress dialogue box. In the dialogue box, processed numbers of valid items, falsified items and elapsed time can be seen. The elapsed time is normally limited to 300 seconds by default in SLDV.

Detailed analysis reports are available in PDF or HTML format, which includes the whole process of the error detection. Another option is to highlight the errors on the model. Error highlighting is of huge benefit during debugging of complex logical structures[25]. Figure 3.5 shows the error detection box during the error detection mode.

In the highlighted model, the coloured boxes, branches or logical conditions with red means that an error is detected. If the process is highlihgted as green, it means that the process is valid and error-free. Figure 3.6 shows highlighted boxes with green and red.

The type of errors that can be detected in SLDV includes static run-time errors (division by zero and integer overflow), detecting dead logic, reachability and assertion violation detection.

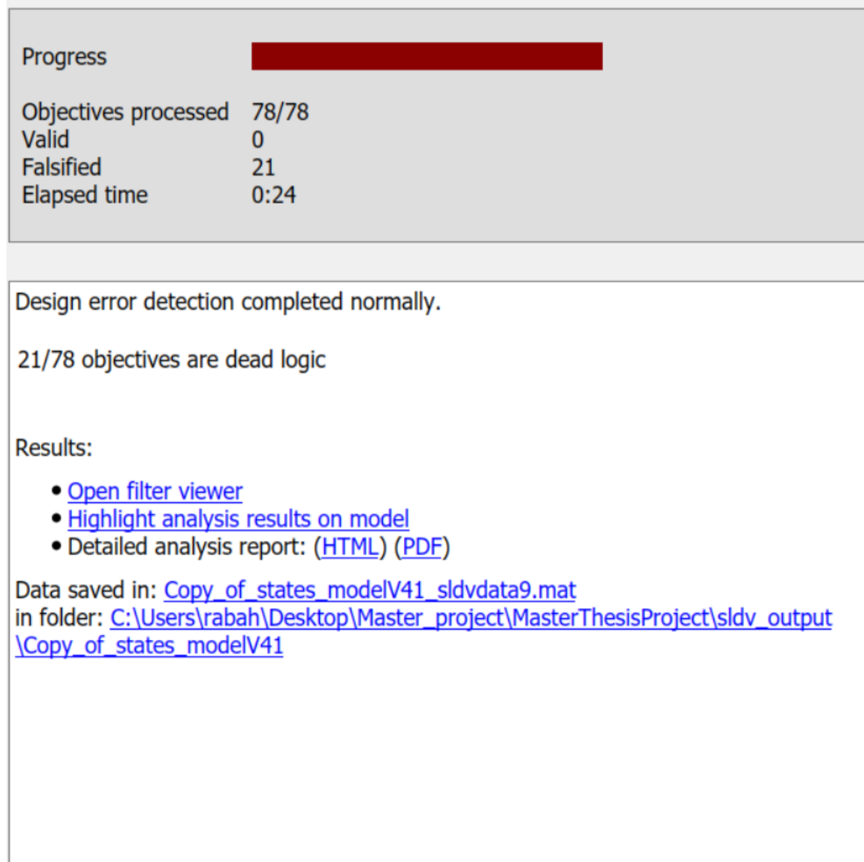


Figure 3.5: Design error detection progress box

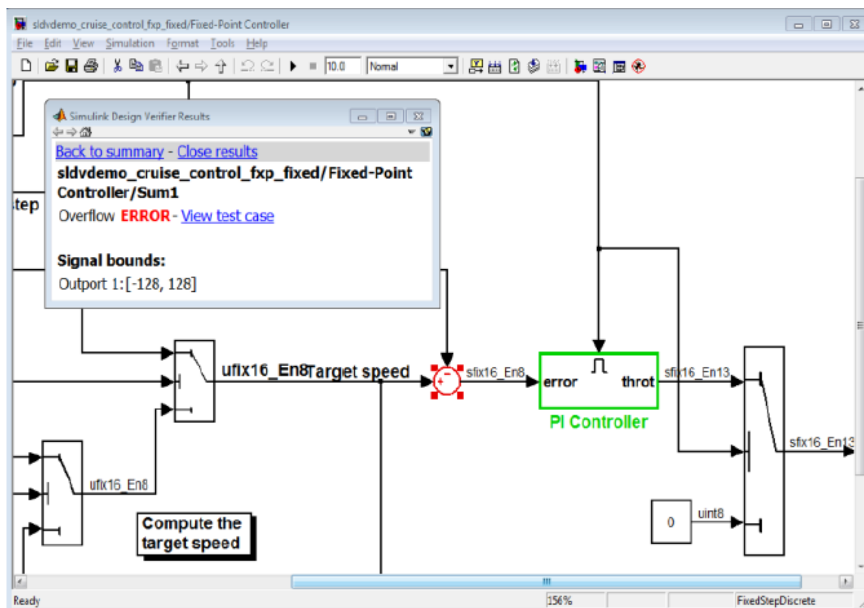


Figure 3.6: Error detection and model highlighting[25]

Integer Overflow Detection and Division by Zero

Integer overflow occurs when the result of an arithmetic operation does not fit into the allocated memory space. SLDV uses the error detection analysis to identify both overflow and zero division in the model. SLDV checks all possible paths inside the model and returns valid in case no error was discovered during the error detection, or else generates test cases that demonstrates the integer overflow and zero division.

Dead Logic Error Detection

Dead logic occurs when some states or logic remain inactive during the execution, or is obsolete. Design errors or requirement errors causes the occurrence of dead logic. If the code is generated while dead logic is present, it generates a dead code. The simulation test is not enough to uncover dead logic in the model even if exhaustively used in the model test, and is not adequate to prove that a specific part of the model is kept inactive during the execution test. To detect dead logic in the model, the test generation mode in Simulink Design Verifier is used. When the test generation analysis is ended, the objects inside the model are highlighted red or green depending on the presence of dead logic. Red color refers to the objects that cannot be active in the simulation, green color means fully active objects. Finally, a test case is generated by Simulink Design Verifier to reproduce the dead logic in simulation[16].

Assertion Violation Detection

By activating assertion violation detection in the property-proving mode, Simulink Design Verifier becomes able to test all valid scenarios that can trigger an assertion violation during the simulation. Any valid scenario that leads to assertion violations are highlighted as red, and the test vector that triggered the assertion violation is generated by Simulink Design Verifier[16].

3.1.4 Verifying Design Against Safety Requirements

In order to prove the correctness of the system against the safety requirements and formalize the safety properties, both the assumption objective block and proof objective block are used. By utilizing the assumption objective block, the input vector can be constrained to certain input values. In proof objective blocks the *value* parameters are set to acceptable values of the block's input signal. Any deviation from that value causes a property violation and the property is disproved. SLDV verifies all associated requirements as properties with the designed system under all possible input values and provides a counter example if a property is disproved[17]. Figure 3.7 illustrates how to write a safety property in SLDV, where signal 2 needs to be false for the condition to be met.

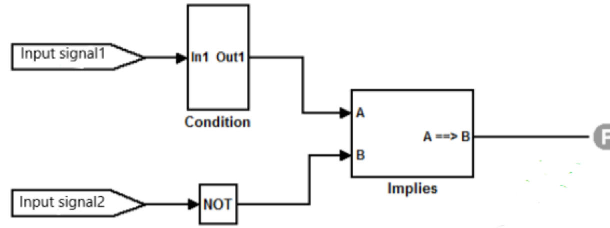


Figure 3.7: Safety property in Simulink Design Verifer

3.2 RoboTool

RoboTool is an Eclipse plugin that supports graphical modelling and validation of RoboChart models[22]. The software is developed by the RoboStar research group[22]. RoboChart is a notation inside RoboTool, used for creating models which are specifically designed for robotic systems in the form of state transition diagrams. RoboTool supports the CSP model checker FDR4[10]. Figure 3.8 shows the interface of RoboTool and provides a few explanations.

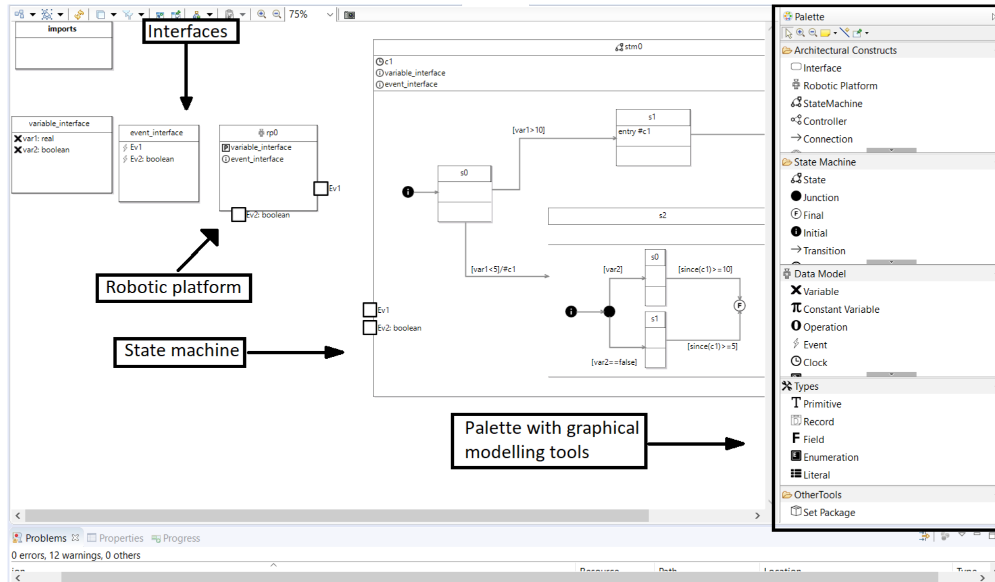


Figure 3.8: The graphical user interface of RoboTool

In this project, RoboTool is one of the tools used to model the system. There are certain limitations to using RoboChart to make the model. First of all, simulation is not possible in RoboTool as of yet, which makes the analysis more generalized and focused around the architecture rather than a numerical analysis through simulation. RoboChart models are intended to have a higher abstraction than models designed for simulation tools. However, it is possible to define more specific tasks and *operations*, mainly through the *operation* module. Adding such functionality will highly increase the complexity of the model, and may cause problems for the verification tool FDR4. This will be discussed further in Section 4.3.

RoboChart can be used both for modelling sequential systems and parallel systems. In general, the *state machines* inside the RoboChart notation are used for sequential behaviour, and *controllers* are used to model parallel behaviour. The top level component in RoboChart is a *module*, which is used for representing the whole robotic system[20]. A *robotic platform* contains the *variables*,

events and *operations* that represents the hardware of the robot.

The interfaces in RoboChart are used to store *variables*, *events* or *operation signatures*[20]. Figure 3.9 shows examples of such interfaces. The definitions of *operations* however, are defined in its own module, *operation definition*. Inside an *operation definition*, an internal state transition diagram can be made to define the behaviour of the *operation*. The *operation definition* module requires access to the *variables* and *events* that are being used in this diagram. The required interfaces containing these, can be provided to the *operation definition* by using the *required interface* feature in the top window of the *operation definition* module, see Figure 3.10. By using the *required interface* tool, the *variables* are considered shared *variables*. If the *defined interface* tool is used, the *operation* will create local *variables* specific to that module.

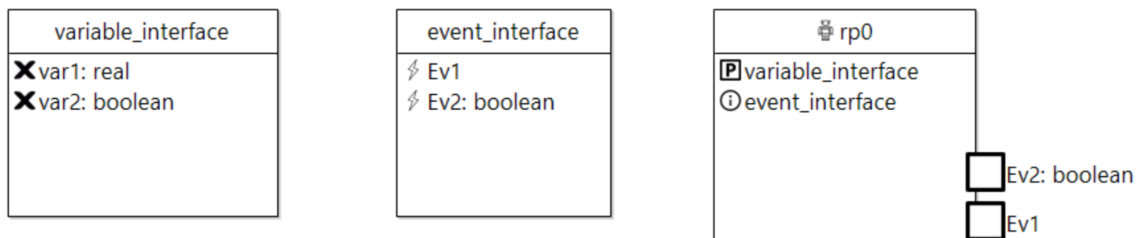


Figure 3.9: Examples of interfaces and a robotic platform

Events are the main form of communication between a *state machine* and its environment. The connection can be between other *state machines*, *controllers* or a *robotic platform*[20]. The connections between the *events* between *state machines* can be defined as synchronous or asynchronous. If the connection is defined as synchronous, one of the *state machines* have to wait for the recipient to receive the event. While on asynchronous communication, the event can simply be written, but the *state machine* writing the event, can simply proceed without waiting for the event to be read. *Events* can also be used to represent a physical aspect of the robot, such as a sensor signal. An example used in the RoboChart documentation for instance uses "obstacle" as an event, which would represent a sensor having detected an obstacle[19].

As mentioned earlier, certain modules may require access to an interface containing either *variables* or *events*, that have not been explicitly defined inside the module. For this purpose, the tools *required interface*, *provided interface* and *defined interface* can be used. Modules such as *controllers*, *state machines* or *operation definitions* have two windows. The top window is for variable declaration and providing interfaces using the tools mentioned above. Figure 3.9 shows an example of how these are used for a robotic platform. The interface containing *variables* is provided using the *provided interface* tool, while the interface with *events* are provided using *defined interface*. On the robotic platform, two boxes with the event names appear after providing it. These boxes are used for the connections described earlier. By using the *connection* tool, connections can be made between these and a *controller* or *state machine* where communication is required.

A *state machine*, as mentioned earlier, is used to describe the sequential behaviour of the robot. Inside the *state machine*, different tools can be used, such as states, transitions, initial junctions, junctions and final junctions. A state is the most basic and essential of these. It is stable, which

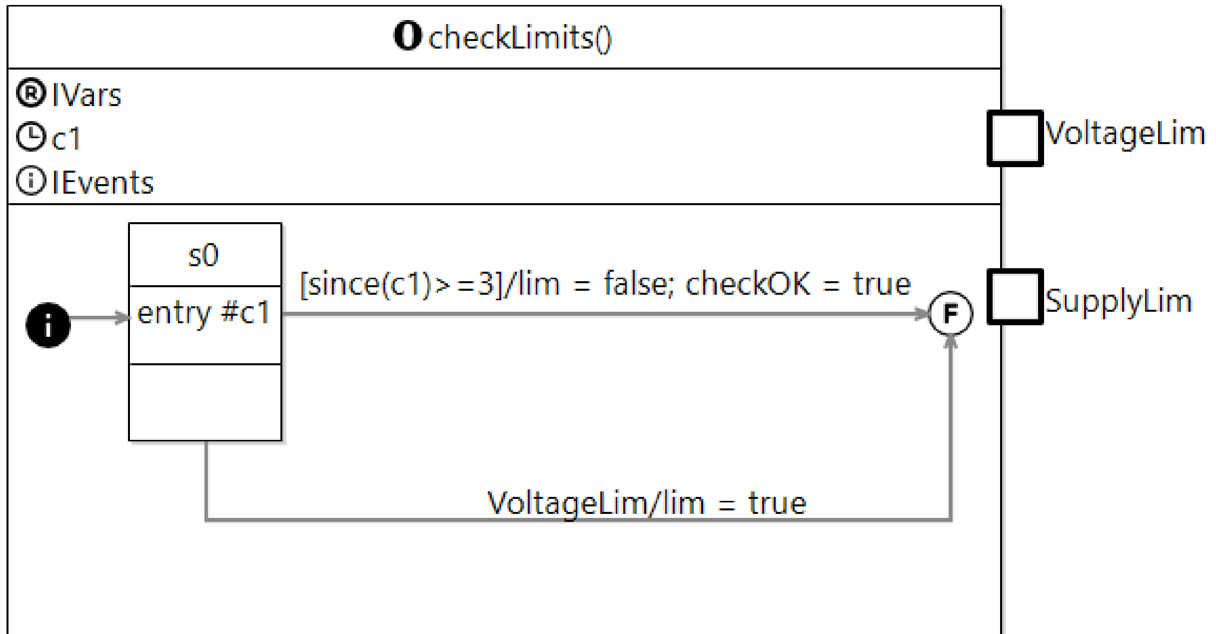


Figure 3.10: Example of an *operation*

means that time can be spent inside of it, and a transition from it does not have to happen instantly. Transitions are the connections made between states or junctions. A transition can be guarded by probability, an atomic proposition (a boolean true or false value) or an event. The transition can also perform an action or reset a clock. The stable property of a state mentioned earlier, means that one of the transitions from the state does not have to be true at all times, since it is possible to spend time in the state. For junctions, this is not the case. The initial junction can only have one outgoing transition, this can not be guarded, but it can perform an action or reset a clock. A *state machine* must have exactly one initial node. A regular junction can have several outgoing transitions, but at least one of the outgoing transitions has to be true at all times, since a junction is unstable, and no time can be spent in a junction. A final junction marks the end of a state transition diagram. It can not have any outgoing transitions, but it can have multiple transitions going into it.

3.3 FDR4

Failures-Divergences Refinement (FDR4) is a refinement checker that uses CSP algebra[9]. FDR4 has a graphical interface and allows to check assertions written in CSP_M. If the assertions do not pass, the refinement checker shows a trace of the instance where the property in the assertion has not been fulfilled. This feature can be used as a means of debugging the model. By identifying where the property is violated, this can give insight to how this violation might be resolved. FDR4 also has a probe tool available, where the user can manually go through the state transitions and pick which sequences to look through. This allows for the user to look for specific transitions in the models in order to get a better understanding of how the model may be executed. Figure 3.12 shows the interface of FDR4, the left margin shows the assertions written in the RoboChart assertions file.

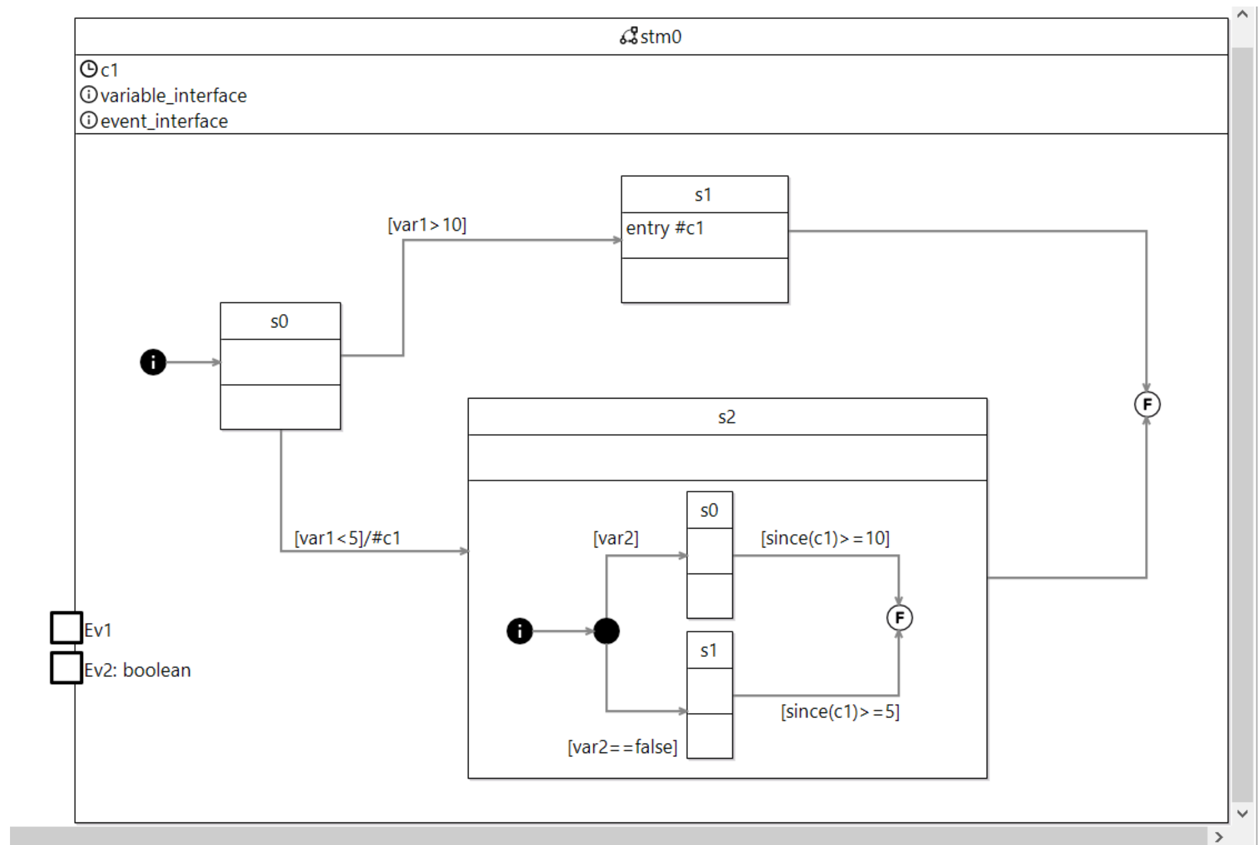


Figure 3.11: Example of a state diagram, showing some of the properties described in Chapter 3.2

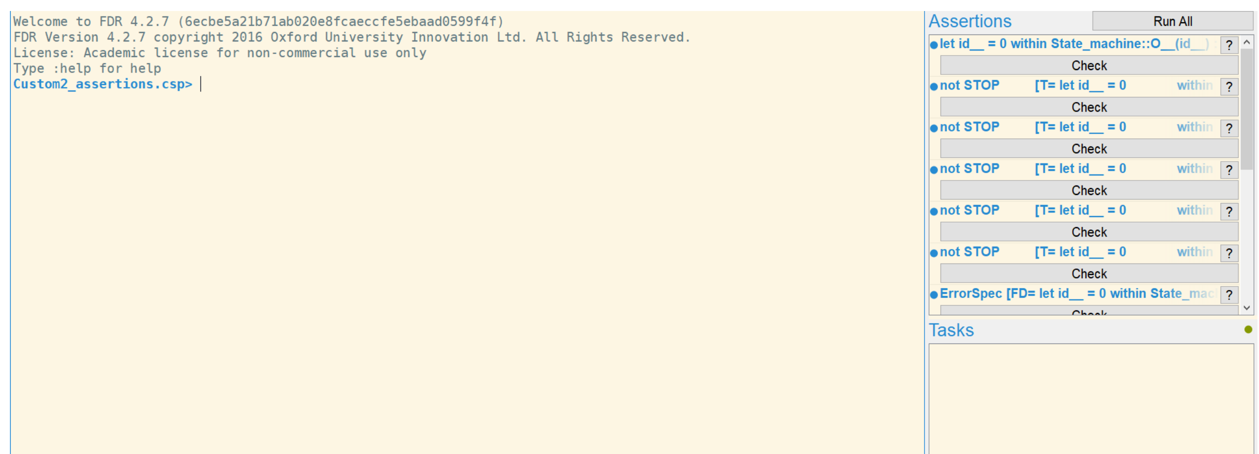


Figure 3.12: The interface of FDR4, with assertions in the right side margin

Models with a high level of complexity may encounter problems when assertions are checked with the FDR4 tool. An increased number of states, *variables* or large data types greatly increases the complexity of the model. When FDR4 attempts to verify an assertion, it will run all possible executions of the model, which means that a few states may still lead to the verification tool having to check a great number of transitions for each assertion. This has to be considered when modelling, and can be a great challenge. Trivial functionality that will not affect the properties of interest should be disregarded. This is one of the challenges when doing the formal verification, because the programmer will have to be able to identify which aspects of the model are vital and which aspects should be neglected. Floating point numbers are not currently supported by the FDR4 tool, this must be considered when creating the RoboChart model[23]. User defined data types can be defined within the assertion file, which can then be used in the model. Care should be taken here, as creating large data types, or using floating point data types can cause trouble for the FDR4 refinement checker due to computational limitations.

Assertions in FDR4 are the properties to be verified by the FDR4 verification tool. Some of the most common assertions such as deadlock, reachability and timelock are included in the FDR4 semantics. The tool also accepts custom assertions, which will have to be defined by the user. FDR4 only accepts CSP files for verifications. Assertions written in the assertion file automatically gets saved within CSP files both within the project's csp-gen and csp-gen/timed folders. It is however possible to edit these files directly, but it is recommended to use the assertion file. By using the assertions file, the syntax will be generated automatically, which requires less familiarity with CSP_M. CSP specifications written within the assertion file can be specified as timed, untimed, or not be specified. If the type is not specified, the specification will be compiled within both the timed and untimed CSP files. Whereas if this has been specified, it will only be compiled to the appropriate file. CSP specifications in the assertion file must be encapsulated with a csp-begin and csp-end statement. The CSP specifications are used to create custom specification, assertions can later be written based on these specifications. Built in specifications can directly be written as assertions.

3.3.1 $\mathbf{CSP_M}$

CSP_M is a lazy functional language with support for CSP notation, and can be used for writing assertions[6]. It was developed as a means of encouraging development of CSP tools. In RoboTool, FDR4 is used for the model checking, FDR4 supports assertions written in the CSP_M language, and was the first tool to utilize this dialect[23]. CSP_M is an important part of the verifications using RoboTool, since it is the language used for writing the assertions. The language is used for the formal verification of the RoboChart model. The CSP_M reference manual describes programming languages as being more generally used for describing algorithms in a form that can be executed[23]. CSP_M differs from other types of programming, since it is intended as a way of describing parallel systems which can be analysed and manipulated automatically. Therefore, it is more accurate to consider it as processes rather than an executable program in the traditional sense[23]. CSP_M has no restriction on the use of upper and lower-case letters. That means that the user of the language may freely adopt a naming convention for separating types such as processes, constructors or channels. CSP_M assertions can be tested in three different types of models, the traces model, the

failures model and the failures-divergences model. In the traces model, a process is represented by a finite number of traces[8]. The failures model is represented by traces as well as it failures[8]. Failures come in pairs of (s, X) where s is a finite trace of the process and X is a set of *events* that the model can refuse after s[8]. In the failures divergences model, the failures model is included, but additionally it includes the divergences model[8]. The divergence part of the model is a finite trace during or after which, an infinite sequence or consequitive internal actions occur[8]. When using FDR4 for verification, these different types of models can be specified, if no model has been specified, the failures-divergences model will be chosen by default.

Chapter 4

Modelling

This chapter will go through the modelling of the abstract models. The first section will go through the general modelling procudure and discuss some of the differences between the tools. The two next sections will go through the specific modelling procedure for Stateflow and RoboTool respectively.

4.1 General Modelling Procedure

As mentioned earlier, this case study is in a quite unique situation when it comes to the workflow. In the regular workflow, the model checking is done before any code has been written and before any testing has been done. The model checking procedure is therefore used very early in the development phase. The standard workflow of such a development phase is demonstrated in Figure 4.1. From the figure it is seen that the first step is making the model for verification. This model has the form of a state machine and should be of a higher abstraction than a model used for simulation. The steps will be slightly different between the Simulink and the RoboChart model, since Simulink offers the possibility of simulating the model. Additionally, in Simulink the model can have more details and mathmatical operations. RoboChart models are intended for a higher abstraction, and the properties to be verified are at a higher level.

Models for formal verification are designed to be output-driven while simulation models are input-driven[13]. A simulation model tests a certain set of input and the software simulates the behavior of the system according to these input. On a model for formal verification, one looks for particular properties regarding the outputs. The model checker will then verify whether the given property holds[13]. This concept differs greatly from input-driven models. It can be a challenge for an engineer with experience using simulation as a verification tool to adapt to this new way of modelling. That is something the group has experiences during the process, especially regarding the modelling of RoboChart models. StateFlow on the other hand allows more input-driven methods to be used when designing the model, while the result is still considering an output-driven approach.

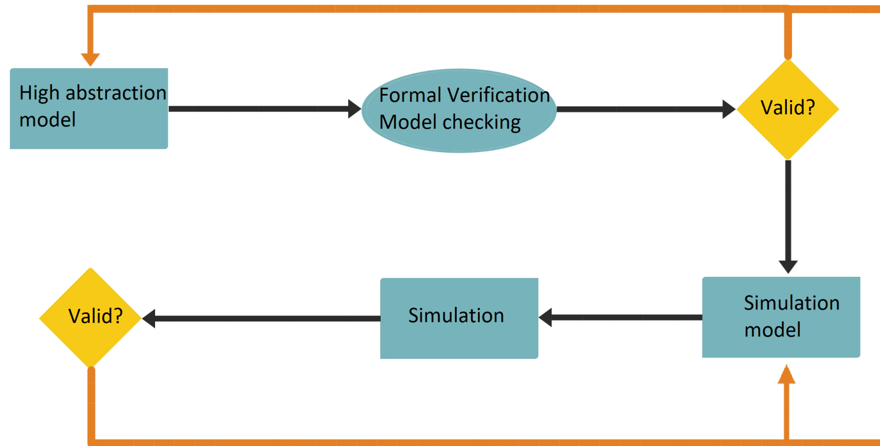


Figure 4.1: Representation of a general modelling procedure using model checking and visual representation of how model checking is related to simulation

4.2 Simulink Modelling

The Simulink model has been built according to the C++ code that was provided by ABB. The necessary requirements and specifications were extracted from the code. The first step in building the model was to determine which lines in the code were relevant to perform formal verification. The second step was to make a simplified Stateflow chart to describe the sequence of actions and events. The Stateflow chart is comprised of limited states and therefore a vital aspect is to determine and specify the number of states, transition conditions between those states, action conditions and the variables in the states. In Simulink, each state has its own variables, shared variables and the variables that result from the calculation inside the state itself. Variables can be local or global. The input and output into the system is considered as a global variable where the local variables can be for example a variable to enumerate the state in stateflow and use it later. The input variables in the Simulink environment could be used as a transition condition in the Stateflow environment. Moreover, the output of each state in Stateflow can be scoped or displayed in the Simulink canvas.

The functionality of the high voltage control module has been expressed in two sub models in the Simulink canvas. The Stateflow chart sub model and the controller sub model. The first sub model is a chart for the states of the state machine. It involves the logic for setting the *setPoint* by transitioning from one state to the next inside the flowchart. It constitutes the functionality of the HVC-02 module. Additionally, it monitors the limits of the variables, receiving the new setpoint, calculates the new limits and ramping the old setpoint value to the new setpoint. The second sub model is the controller where a first order transfer function is used to represent the high voltage output from the cascade. The functionality of the IPS in the HVC-02 is modelled by a PID controller. The output voltage is 10 percent less than the actual high voltage from the cascade. Figure 4.2 shows these two sub models. The green block is known as a *Subsystem Verification block* according to the Simulink Design Verifer library. Its functionality and use will be explained later in Chapter 5.

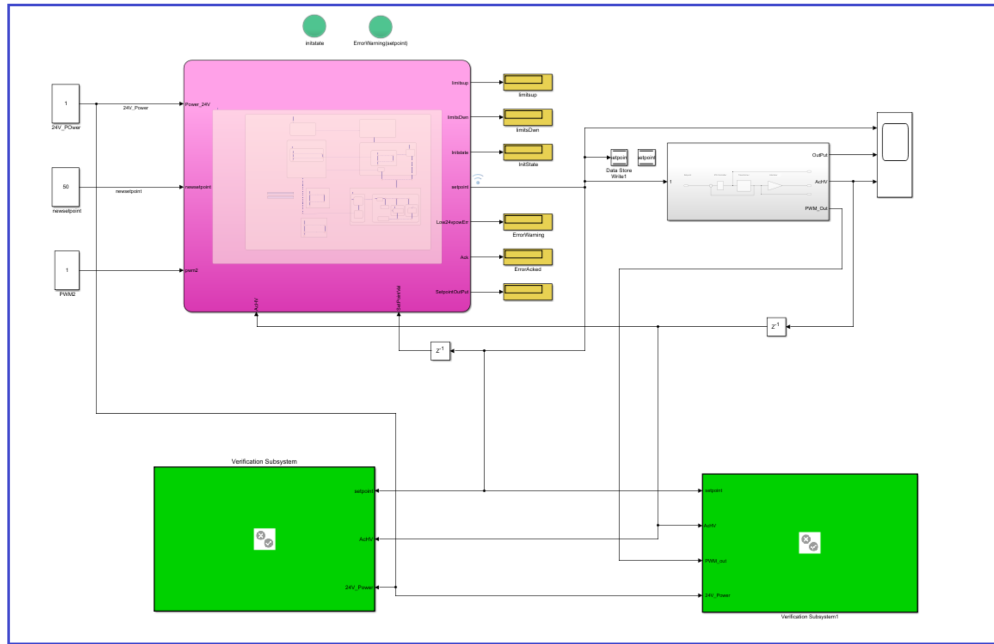


Figure 4.2: The model blocks configuration in the Simulink Environment

4.2.1 Input and Output Variables to the Simulink Model

The variables in the model have been classified as input variables and output variables. The input variables include the value of the set point that needs to be set and is called *newsetpoint* in the model. *pow24VStatus* is another input to the model which causes a transition from one state to another as it will be explained further later in this section. The variable *PWM_out* is a boolean variable used to represent the ramping action. If it is 1, it causes a transition to next state. The variable *ActualHV* is the actual high voltage. It is a representation of the high voltage at the output of the cascade. *setPoint* is the output of Flowstate chart to the Simulink canvas.

4.2.2 Stateflow - Sub model

The flowchart in the model contains the logic of the HVC-02 module. The sequence of the module's functionality is represented as states in the flowchart, so each state constitutes the module functionality at certain conditions. In the *Init* state, where the values are initialized, the acceptable high and low limit values for the variable *setPoint* or the output *ActualHV* of the HVC-02 module are set.

Figure 4.3 shows the *Init* state and its variables. Some of these variables are local variables and some are global variables. In the state, each variable that is preceded by the word *Entry* is classified as an output and can be displayed or monitored in the Simulink environment. For instance, the variable *limitsup* is an output variable and can be observed and used in Simulink. *undervoltVal30*, *undervoltVal90*, *overvoltVal30* and *overvoltVal90* are local variables.

Functions inside the states usually have different font colors. In the *Init* state, *overvoltlimit* and *undervoltlimit* are displayed in green because they are Matlab functions and the body of the function can be inside or outside the state depending on what is required. The previously mentioned

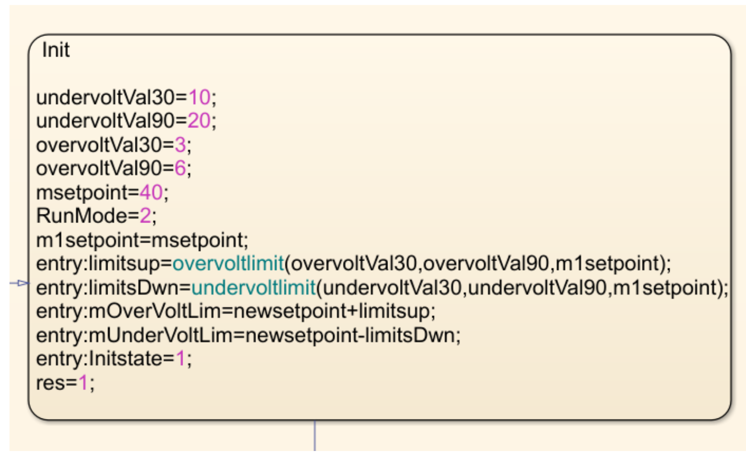


Figure 4.3: The *Init* state in flowchart

functions are used to calculate the upper and lower limits for the *setPoint*. The *CheckLimits* state reads these values during the *CheckLimit* function before ramping the *setpoint*.

Figure 4.4 shows the Matlab function blocks in the Stateflow canvas. The commands of the function are written in the Matlab environment as a m-file script.

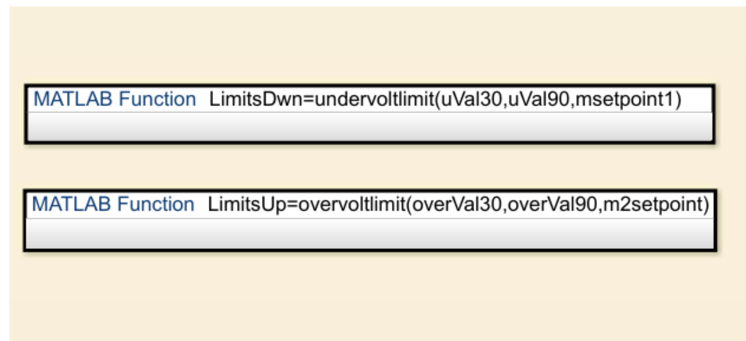


Figure 4.4: Matlab Body's Function Block

The variable *res* was introduced in the flowchart model. This variable is a boolean and is incorporated inside the function *DisableHV*. When the value of *res* is set to 1, the outcome of the function *DisableHV* is that the variable *disableHV* is set to zero. As a result, the *setpoint* is reset and becomes 0 and the actual high voltage goes to 0 as well. However, if the variable *res* is equal to 0, the *DisableHV* function output is 1. Consequently, the set point becomes 0 and the *Error Mode* state becomes active. The variable *res* is only added as a local variable to ease the building of the model and add more flexibility to call the DisableHV function during states when it is necessary.

Figure 4.5 illustrates the function *DisableHV*. Figure 4.5 is a graphical representation of the *if else* function in the Stateflow environment and it is a premade pattern to be used by the user.

After calculating the values in the *Init* state, the transition from the *Init* state to the *Wait24Vpower* state happens. When the *Wait24Vpower* state becomes active, the state checks the value of the *setPoint*. If the value is bigger than zero, the transition from the *Wait24Vpower* to the *ErrorMode* state occurs and the *setPoint* value is set to 0. The dummy variable *SetPointVal* reads the value of setpoint from the Simulink canvas. When the variable *pow24VStatus* is 1 and the setpoint is zero, the state *Wait24Vpower* transits into *ClosedLoop* state which become active. Figure 4.6 shows the *Wait24Vpower* state.

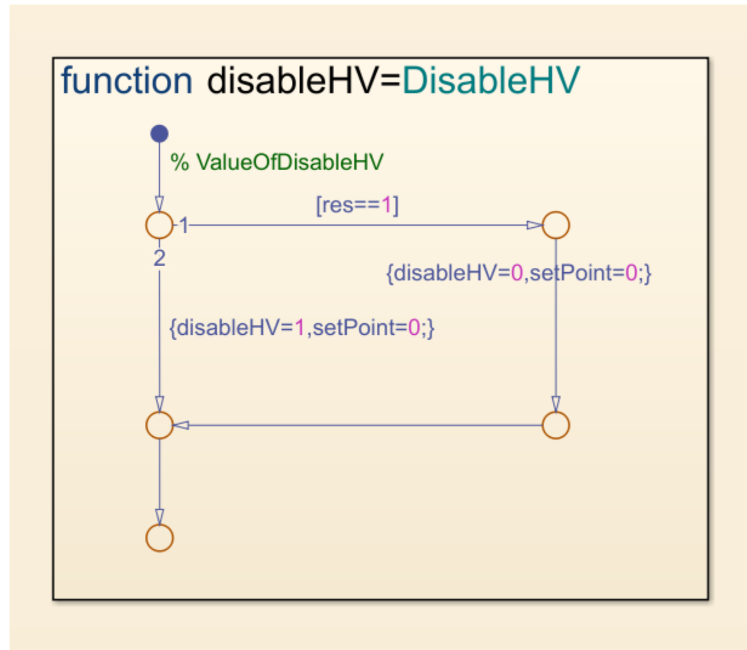


Figure 4.5: DisableHV function

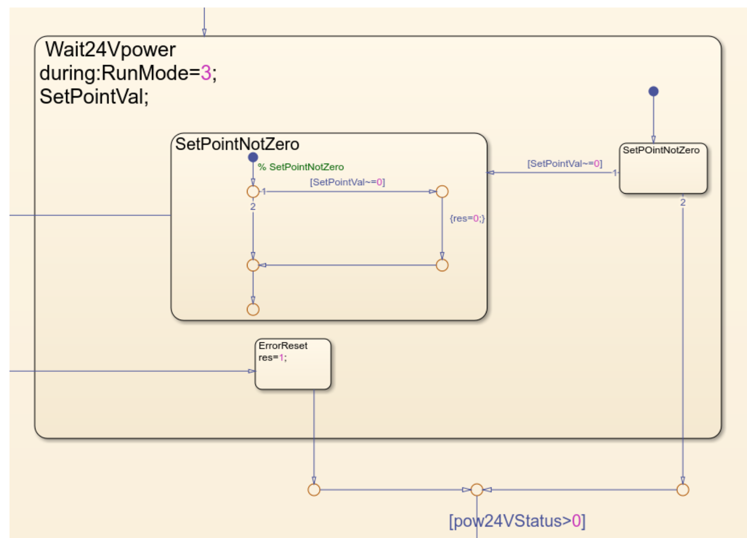


Figure 4.6: Wait24Vpower State

As can be seen from Figure 4.7, the *ClosedLoop* state contains two sub-states. One is called *CheckLimits*. It oversees the *ActualHV* value and resets the setpoint if it is out of the calculated limits from the *Init* state.

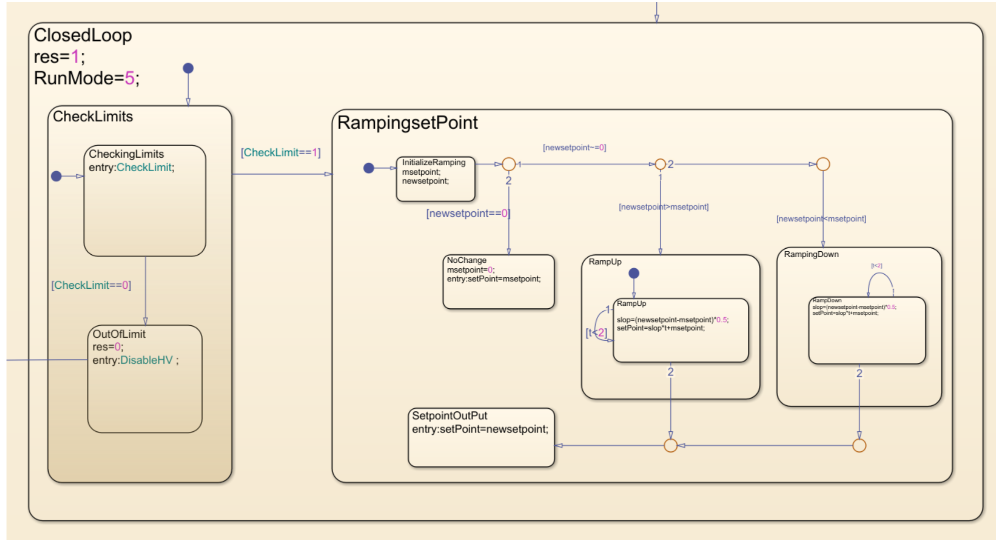


Figure 4.7: ClosedLoop State

The sub state *CheckLimits* includes two child states in sequential order. The first child state is the *CheckingLimits* state. In it, the graphical function *CheckLimit* is called. It returns 0 when its statement is false. When the checked values such as the actual high voltage are out of limits, the *CheckLimit* function returns 0 and hence the child state *OutOfLimit* becomes active.

Because the variable *res* equals 0 in the *OutOfLimit* child state, the *DisableHV* function that is inside it returns 1. Accordingly, the value of *setPoint* becomes 0 and the *ErrorMode* state is activated. The structure of the *CheckLimit* function is illustrated in Figure 4.8.

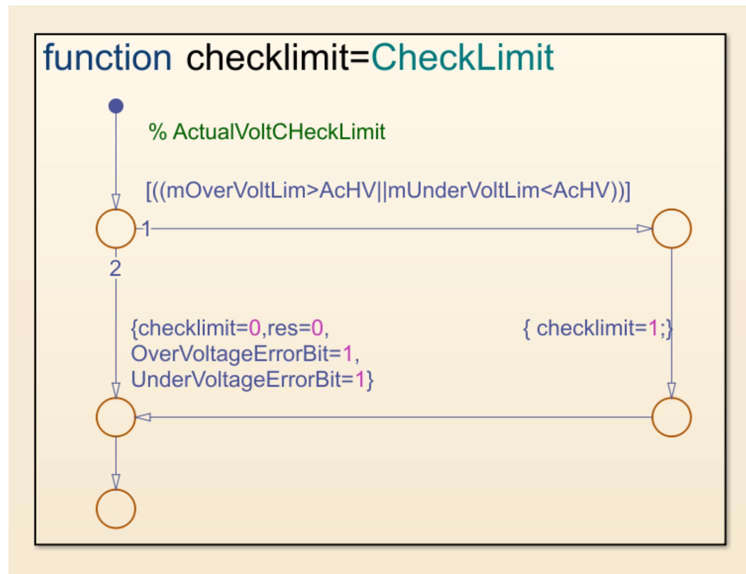


Figure 4.8: The Structure of CheckLimit Function

In case all values are within the limits, the transition condition [CheckLimit==1] is true and the transition from the sub state *CheckLimits* to the *RampingsetPoint* sub state occurs.

The *newsetpoint* variable has three possible values; 0, higher than the old *setPoint* or lower than the

recent *setPoint*. The *RampingsetPoint* sub state ramps the old value of a *setPoint* to the new value. The time of ramping to the new value must be within 2 seconds[2]. The local variable *msetpoint* is used as a dummy variable to store the old value of the *setPoint* in. After the completion of the ramping process, the *newsetpoint* value is assigned to the *setPoint* value that is being fed into the controller.

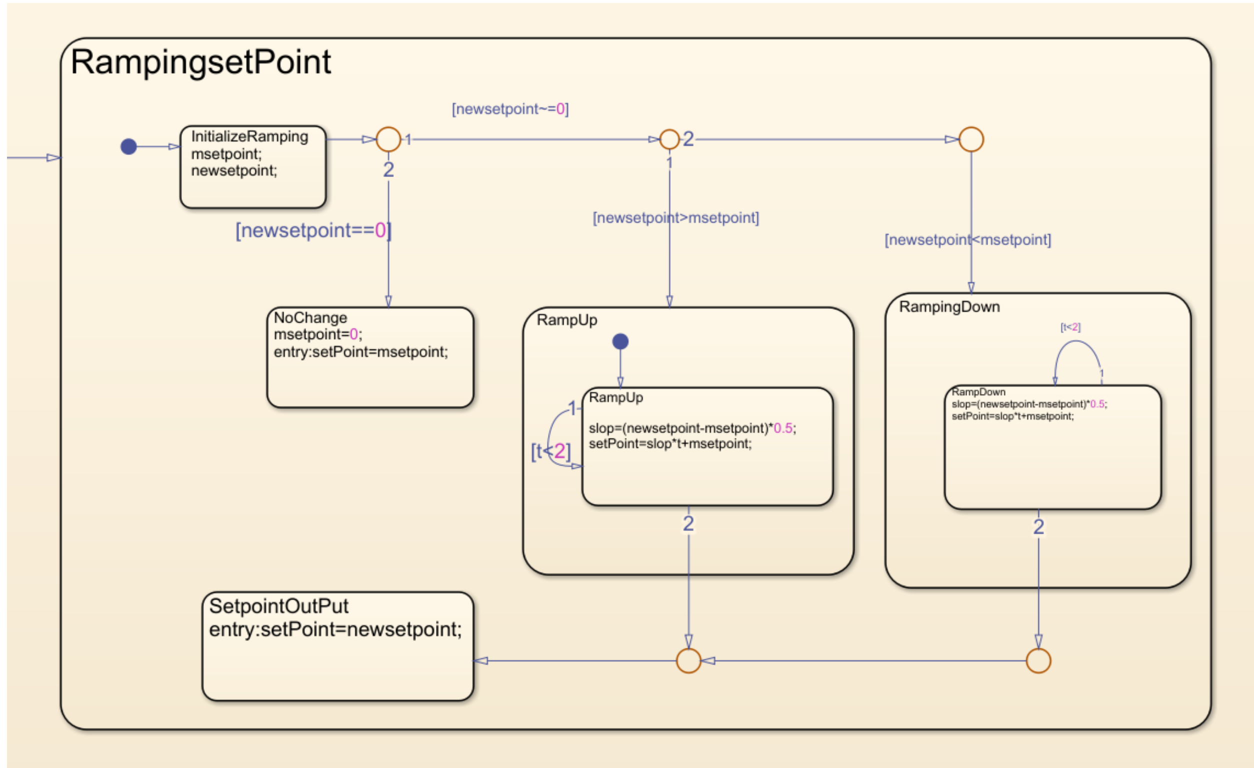


Figure 4.9: RampingsetPoint State

The *newsetpoint* has the potential to be 0 and according to the ramping function in the code, no change has to occur and the variable *msetpoint* will be set to 0.

The *ClosedLoop* state is kept active until the power is switched off, or the values in *Init* state go beyond the allowed limits, or *newsetpoint* is intended to be set again.

The *ErrorMode* state is activated when the *setPoint* goes over or under the voltage limits that are specified in the *Init* state.

The transition to the *ErrorMode* state is set in motion when the variable *res* equals 0 and hence the *DisableHV* function becomes active. Variable *res* can be 1 in the model of the system in two places. The first is in the *Wait24Vpower* state and the second is in the *ClosedLoop* state. Figure 4.10 illustrates the *ErrorMode* state.

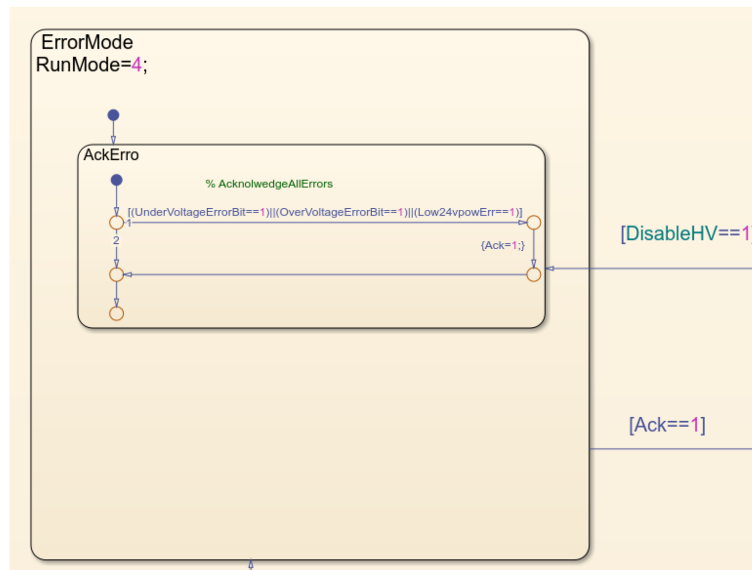


Figure 4.10: ErrorMode State State

4.2.3 Controller Sub-model

Due to lack of information about the PID parameters and the plant, a first order system is included as a replacement of the plant and hence the PID controller is tuned accordingly. The *setPoint* from the Stateflow chart is sent out to the PID controller. The controller eliminates the error between the value of the *setPoint* and the actual high voltage value, *ActualHV*.

In terms of functionality, the PID controller acts as the High Voltage Controller in the HVC-02 module, where the values are sent back from the cascade to the IPS. The output voltage from the applicator or atomizer is less than the *ActualHV* because of the resistance. This drop in the voltage is estimated by 10 percent in the model according to some elementary calculation from the ABB manual[2]. The signals *setPoint*, *PWM_out*, and *ActualHV* are logged and used during properties formalization. Figure 4.11 shows the components of the controller sub-model.

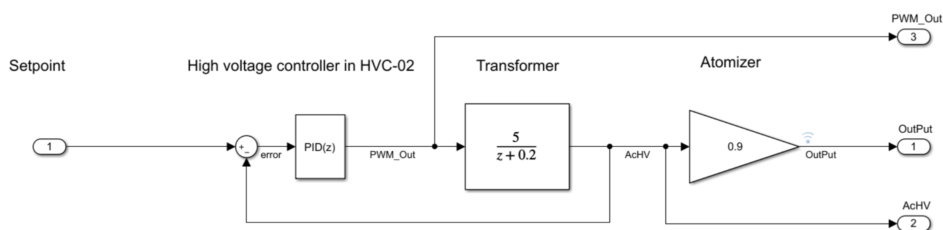


Figure 4.11: The Controller

4.3 RoboTool Modelling

This section will first describe a few changes that had to be done to the model in order to make it usable for verification. During the modelling procedure it is important to know the limitations of the FDR4 tool. A model with many states, large data types, many variables or many calculations can increase the verification time a lot. One of the biggest challenges of modelling in RoboTool has to do with the group's ability to adapt to the more abstract form of modelling. In the final part of the chapter, the latest version of the model will be presented, highlighting the most vital parts, and explaining why it was modelled a certain way.

As mentioned earlier, RoboChart models are intended to be of a higher abstraction than models for simulation. Initially in the modelling process, the idea was to model the system as similarly as possible to the actual code that the model is representing. This means that in the first version of the model, some lower level behaviour and calculations were used in the states. As an example of this, the system has a state responsible for ramping up the duty cycle from 0 to 45 in 135 milliseconds, i.e., increments the value by 1 every 3 milliseconds. This state was indeed made in the RoboTool model, and can be seen in Figure 4.12. In the Figure it can be seen that the initial node transitions to the ramping state, this is because it is the first state in the sequential part of the state machine. The outgoing transition compares the continuously increasing *dutyCycle* variable to a constant *rampThresh* that was defined with an initial value of 45. The behaviour of the state is captured using an internal state machine within the ramping state. The internal state transitions back to itself every 3 time units, the transition increments *dutyCycle* in the transition action every time a transition occurs. The time units in RoboTool do not represent any particular time unit, and the interpretation of it is decided by the programmer, in this case it represents milliseconds.

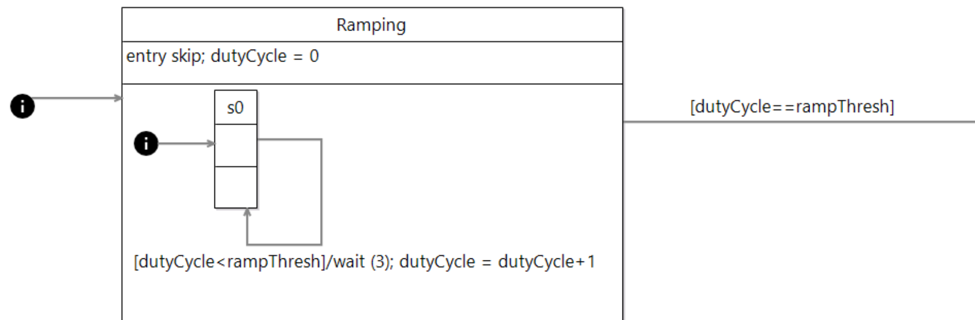


Figure 4.12: The original version of the ramping state

The main reason why lower level functionality is not ideal to capture within RoboChart is because of the verification tool FDR4, this will be described in more detail later. The original ramping state did cause some trouble for the verification, and verification on the model could not be done within a reasonable time frame. This behaviour is not crucial to any of the assertions to be checked, and could therefore safely be neglected. The simplification of the state removes the *dutyCycle* variable, the transition from the simplified ramping state is only based on time in the new implementation. The new version of the state can be seen in Figure 4.13. Not all the states will be presented in such detail, the purpose of presenting changes in the ramping state was an example of simplifications that had to be done in order to reduce the verification time.

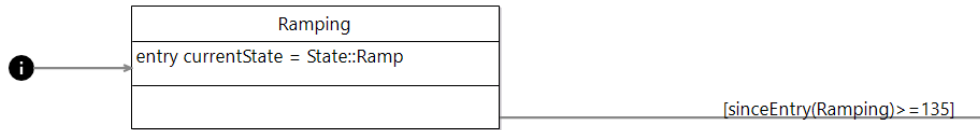


Figure 4.13: The new version of the ramping state

Most of the changes that had to be made on the model was due to misconceptions regarding the use of events. Chapter 3.2 describes the purpose of events, which is mainly used for communications between state machines or to represent physical attributes of the robotic system. Events can be used to cause a state transition. The misconception about events was that initially, the group thought that when an event was called in an action, this would cause a transition if the event was used in a transition from the state where this action was made. An example where this mistake was made was in the original implementation of the *disableHV* operation. The *disableHV* operation will turn off the *setPoint* and the high voltage power *ActualHV*. *disableHV* also requires a boolean argument when the operation is called. If the argument is *true*, the operation should cause the system to enter the *ErrorMode* state. If it is *false*, no transition will happen, but the power and set point is still turned off. The original implementation of the state can be seen in Figure 4.14. A state from the old implementation calling the operation is shown in Figure 4.15.

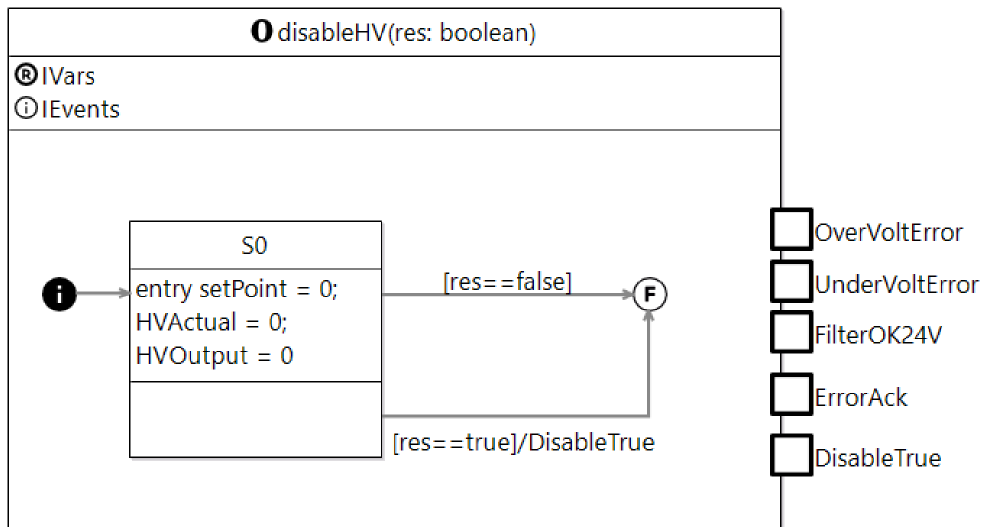


Figure 4.14: The old version of the disableHV operation

In this implementation of the operation, the *DisableTrue* event is used in an action in the operation. The same event as seen in Figure 4.15, is used on the guard between the *Wait24Vpower* and the *ErrorMode* state. The operation call in the internal state *s1* in the *Wait24Vpower* state will not cause a transition, and will not reflect the intended behaviour of the system. A solution to this is to simply use a boolean variable as a guard instead, since the transition should only happen during the specific conditions. The new implementation of the operation is shown in Figure 4.16, the new implementation uses shared variables. The boolean used in the transition is set true or false based on the argument to the operation as described earlier. Figure 4.17 shows the boolean used in a transition, the image also shows that the boolean is set to false during the transition so that if the

system enters the state again later, the transition will not happen when it should not.

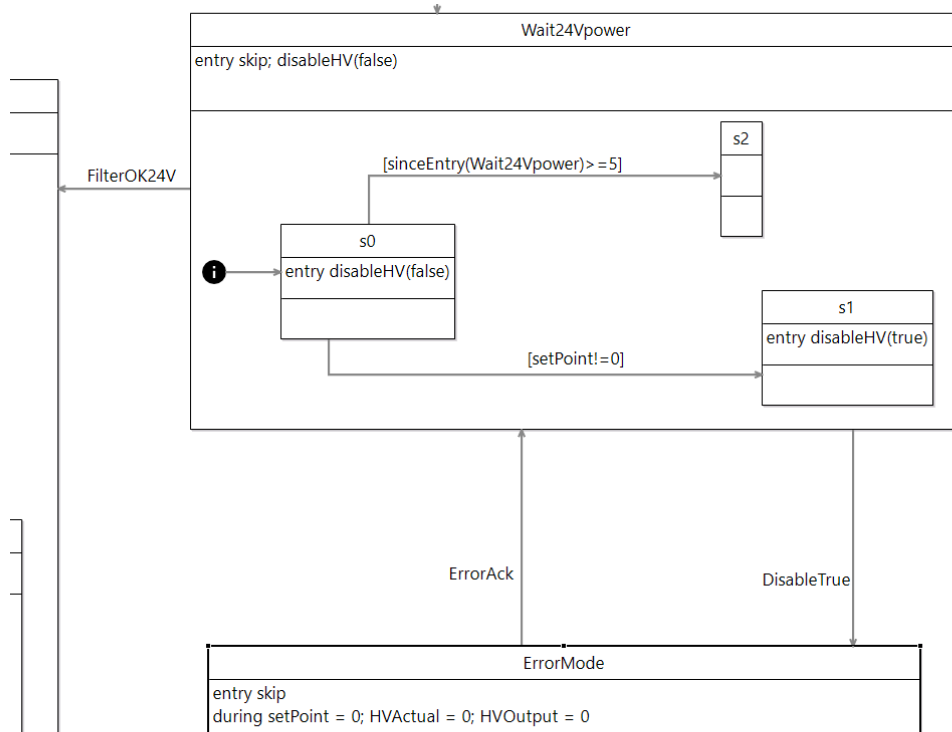


Figure 4.15: Demonstrates where the DisableTrue event is used for a transition

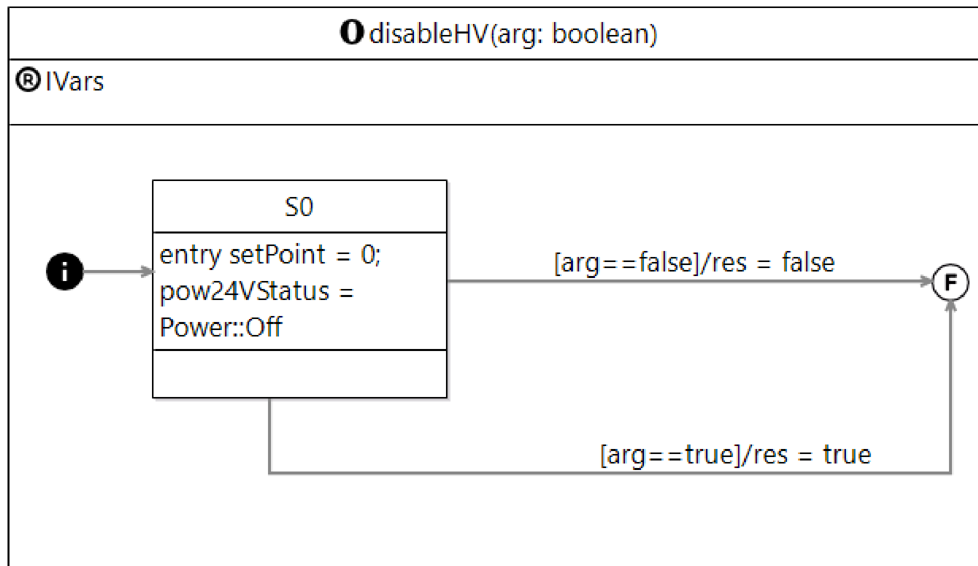


Figure 4.16: The new version of the disableHV operation

The final model does use an event as a transition guard in two instances. An event, as mentioned earlier, can represent a physical attribute of the robotic system, such as a sensor signal. This is used in the *checkLimits* operation. The purpose of this operation initially is to measure the actual high voltage of the system, and compare it to the different limits calculated in the *Init* state. These limits and the comparison have been omitted in the model for the sake of simplifying the model for verification. Capturing the behaviour could be challenging in an abstract model like RoboChart. Unlike a simulation model, the system does not compile and dynamically run through the system. The model is instead analysed by the verification tool, that goes through all possible transitions in order to verify or disprove a property. Specific values could be used, but this would defeat the

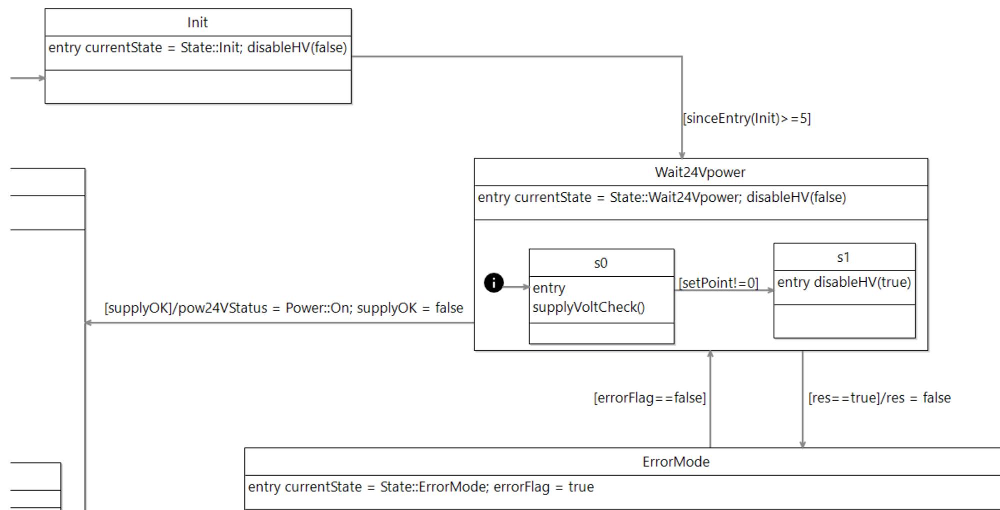


Figure 4.17: Demonstrating the boolean used for a transition

purpose of such a model. The specific value would cause a predefined route the transition would follow, the verification would then only prove the properties for that exact execution. An event *VoltageLim* is used instead of using actual values to do this. Designing it this way will allow both executions to be possible. The *operation* checkLimits can be seen in Figure 4.18. The *operation* supplyVoltCheck is shown in Figure 4.19. This operation works in a similar manner, but uses the event *SupplyLim* to represent whether the supply voltage has breached a limit.

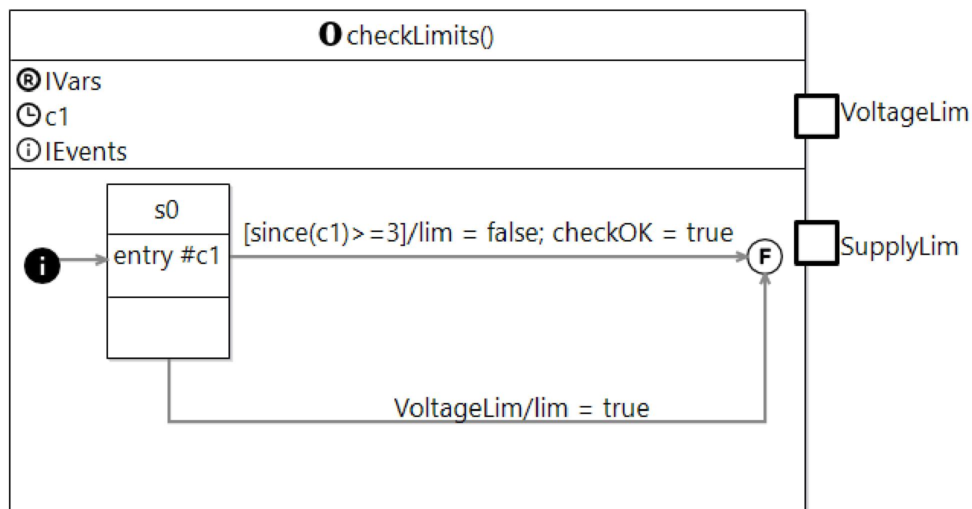


Figure 4.18: Visual representation of the check limits operation

If the variables for *setPoint* and the *ActualHV* are given initial values, they may cause specific transitions to occur, so these values are used a bit differently. The set point is represented as a variable real *setPoint* without an initial value. Instead of having a variable for the high voltage, a custom enumerated variable *ActualHV* is used which can have the values *Power_On* or *Power_Off*. This implementation causes a deviation between the RoboChart model and the behaviour of the actual system and does mean that some of the properties of interest may not be accurately verified. One of the assertions regarding the closed loop state has been represented through using enumerated variables. The *ClosedLoop* state represents a closed loop PID controller. Due to this implementation, actual increase or decrease of the *ActualHV* variable are not modelled.

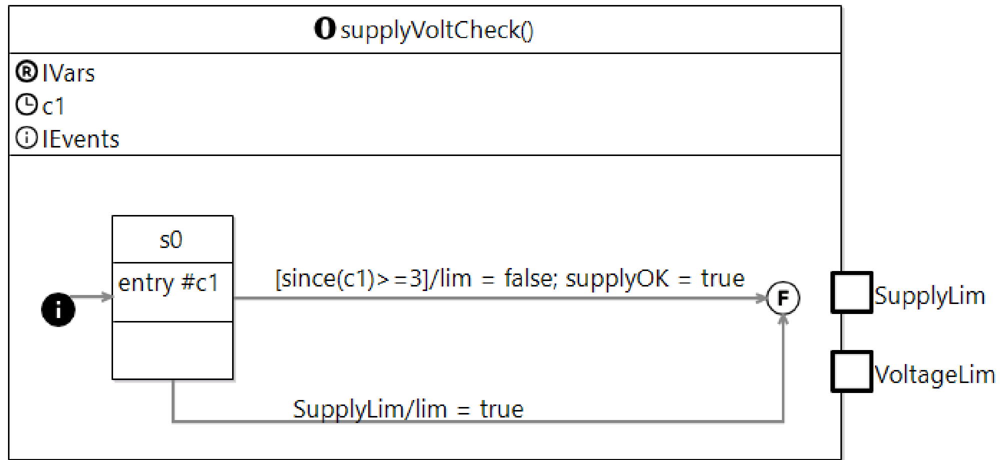


Figure 4.19: Visual representation of the supplyVoltageCheck operation

In the current implementation, the variable simply represents whether the system is controlling the voltage, or if the voltage has been turned off in the *ErrorMode* state. The *ClosedLoop* state can be seen in Figure 4.20.

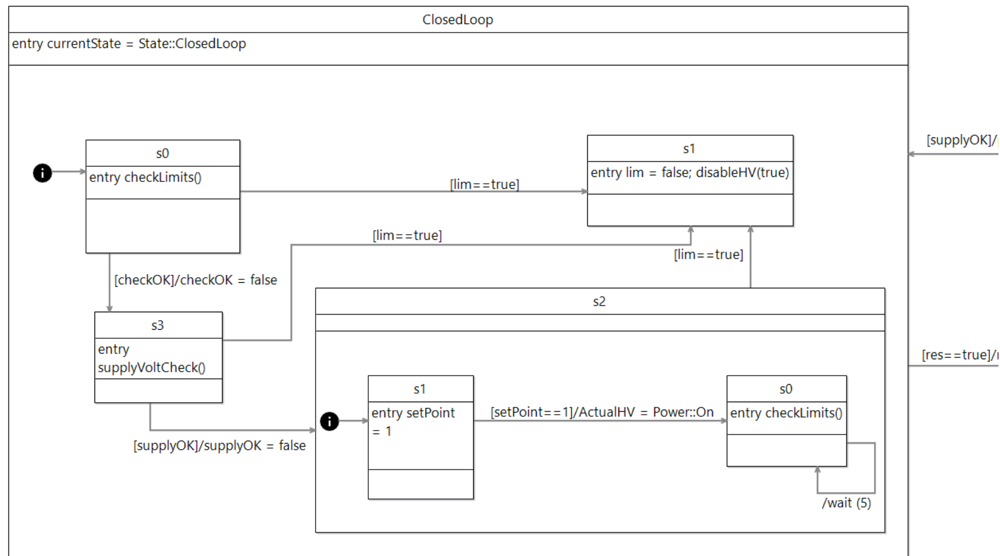


Figure 4.20: Representation of the *ClosedLoop* state

The first thing every state in the system does is to set the enumerated variable *currentState* and gives it the appropriate value. The enumerated variable has a name for every state in the state machine. This is set as an *entry action* to the state, so that it will be set as the states are entered. In the *ClosedLoop* state the value is simply set as: *currentState = State::ClosedLoop* see Figure 4.20. *State* is the enumerated types containing the possible values for the currentState variable, see Figure 4.21.

The behaviour inside the *ClosedLoop* state is designed using an internal state diagram. In the first state, the *checkLimit* operation is called, this is also done in the actual system. If the operation call results in the voltage being outside the limit, represented by the *VoltageLim* event, a boolean variable *lim* is set to true which causes a transition to internal state *s1*. *s1* calls the *disableHV* operation with the argument true, which will cause a transition to the *ErrorMode* state. However, if these set of events do not occur, the operation will set the boolean *checkOK* to *true* and it proceeds

to *s3* which calls the *supplyVoltCheck* operation. The *supplyVoltCheck* may cause a transition to *s1* which causes a transition to *ErrorMode*. If the event *SupplyLim* does not trigger a transition inside the operation, *supplyOK* is set true, and the state machine proceeds to *s2*. Internal state *s2* emulates the controller by setting the *setPoint* variable to 1, then if the *setPoint* is 1, it changes *powStatus* to *On*, while calling *checkLimit* every 5 time units. This operation call may cause a transition from the internal state which will cause the system to transition to the *ErrorMode* state.

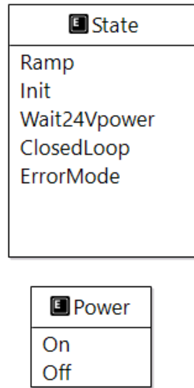


Figure 4.21: Representation of the enumerated interfaces for *State* and *Power*

The *ErrorMode* state is entered whenever the excecution encounters an error. There errors can be tied to voltage or current limits being exceeded, or if there is a fault in the execution which gives *setPoint* a value when it should be zero, among other things. Most of the circumstances that can lead to transitions to the error state occurs in the sequential part of the model that has been modelled in RoboChart. In the actual system there are three different *watchdogs* monitoring certain conditions and operating concurrently with the sequential part. In RoboChart, these *watchdogs* have not been modelled. In order to include them, they would each be created within their own *state machine* within the same *controller* where the sequential *state machine* is. These state machines would then communicate based on events. The *watchdogs* would then be able to affect the sequential system in order to cause transitions to the *ErrorMode* state. However, in the current model, only the sequential part is included. The state is shown in Figure 4.22.

When the sequential part of the system is isolated from the *watchdogs*, transitions to the *ErrorMode* state can only occur from the *Wait24Vpower* or *ClosedLoop* state. The purpose of the *ErrorMode* state is to shut off the power and set the *setPoint* to zero when an error occurs. Once this has been done, and all errors have been acknowledged, the system will proceed the the *Wait24Vpower* state. In the RoboChart model, this is somewhat simplified. For instance, there is nothing to represent that the errors have been acknowledged. In order to do this, a variable could be used to keep track of the type of error that has occured. The boolean variable *errorFlag* is used to signify if the system is in the *ErrorMode*, which means that an error has occured. The flag is set true in the entry action to the state, and then set false when it exits. This variable can also be used in order to analyse the behaviour in the state.

An overview of the *state machine* is shown in Figure 4.23. The functionality within the states is collapsed in order to display the architecture better.

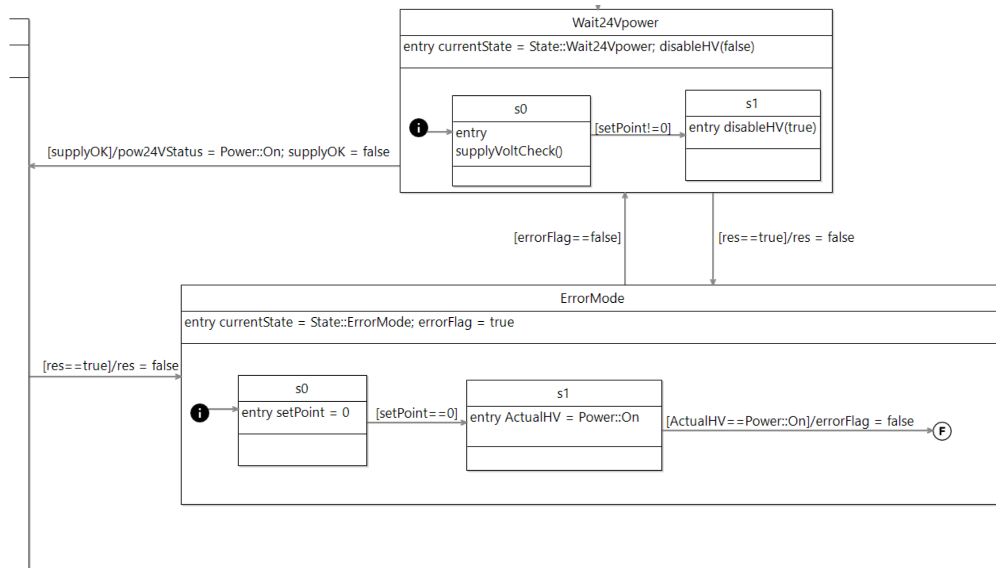


Figure 4.22: Demonstration of the *ErrorMode* state

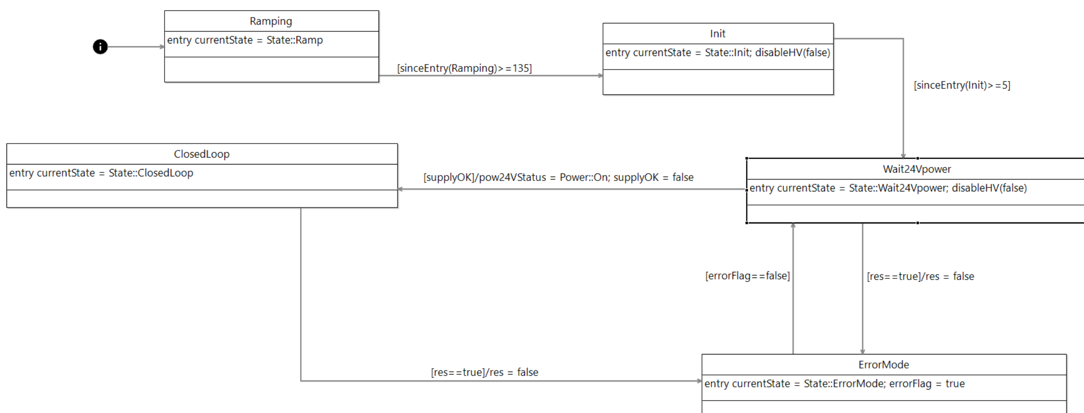


Figure 4.23: An overview of the state machine and its transitions

The final *interfaces* for storing *operations*, *variables*, *events* and the interface for the *robotic platform* is shown in Figure 4.24. The structure of the system is made seperately from the file containing *interfaces* and *operations* and the file containing the *state machine*. This is done in order to have a structured overview of the hierarchy of the model and also to make the file for the *state machine* more readable. The structure of the system is shown in Figure 4.25.

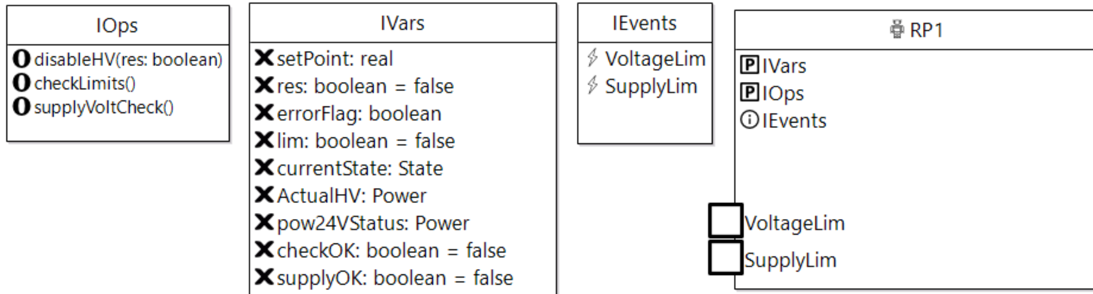


Figure 4.24: The different *interfaces* used in the RoboChart model

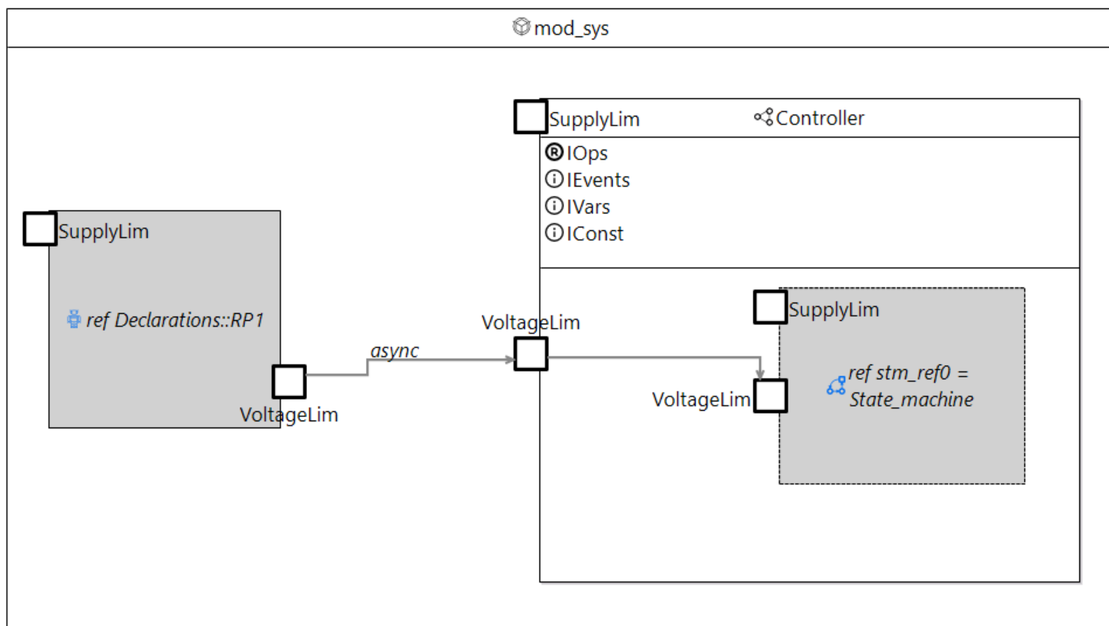


Figure 4.25: Displays the architecture of the RoboChart model

Chapter 5

Formal Verification

In contrast to Chapter 4, which focuses on the modelling capabilities of the tools as well as presenting the model of the system, this chapter will specifically be focused around the model checking and formal verification of the tools. This chapter will show the implementation of these concepts on the models created during the project. Differences between the model checking with the different tools will become apparent. Section 5.1 will go through the verification in the Simulink Design Verifier, Section 5.2 will explain the verification process used by RoboTool and FDR4. The properties of interest can be simplified as:

P1: The actual high voltage should always follow the setpoint.

P2: The PWM output signal should be set to 0 when the 24V power signal is switched off.

P3: The system should be deadlock free.

5.1 Verification in Simulink Design Verifier

The method to verify the requirements of the system and how to formalize them in Simulink Design Verifier will be discussed and presented in this section. Property verification blocks and assertion writing technique will be showed and discussed here. Proofs of the defined properties will be shown in the result chapter. Under the Simulink Design Verifier library, there are three sub items that helps the user to write and formalize the requirements in Simulink. These items are:

- Objective and constraints.
- Temporal operators.
- Verification utilities.

A combination from the first and last items are used during the process of writing the formal requirements for this thesis. The second item, temporal operators, is comprised of *Detector*, *Extender* and *Within Implies* blocks. The temporal operator function is mainly to shift the signal steps in

time or detect the signal after certain steps in time. The temporal operators are not used in this thesis. *Assumption objective*, *proof objective*, *implies* and *verification subsystem* blocks were briefly explained in Chapter 3, so the point of focus here will be on their implementation to prove the properties. Two verification subsystems are placed in the model. In order to function correctly, the block may not have any output signals. The inputs to the verification block comprise the inputs and the outputs from the Simulink model. For instance, the input variable *pow24VStatus* and the output variable *ActualHV* are inputs to the verification subsystem block. Figure 5.1 shows the input variables to the block and it should be noted that there are no outputs from the block to the Simulink canvas.

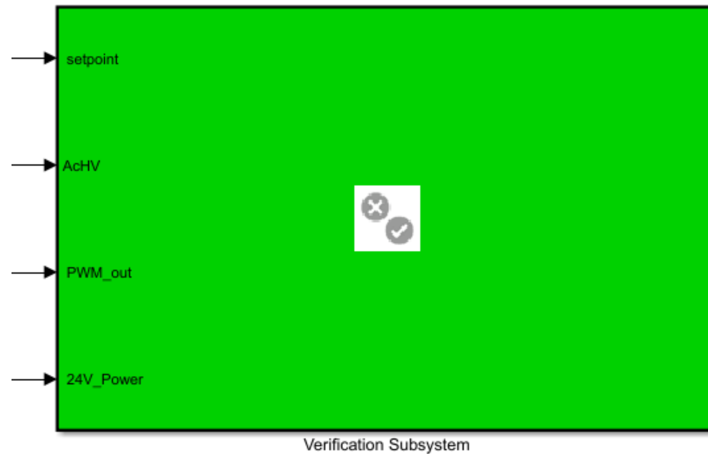


Figure 5.1: Inputs to the verification subsystem block in the model

One of the formal properties for the system requirements inside the block in Figure 5.1 is shown in Figure 5.2. P1 is the first property to be proven. The actual high voltage *ActualHV* must always follow the *setPoint* value. Accordingly, if the value of *setPoint* equals 0, the value of *ActualHV* variable must go to zero. In contrast, when the value of the *setPoint* is higher than zero this implies that the value of *ActualHV* is bigger than zero as well.

In logical math, the property P1 can be written in two parts. The first part expresses the relation between *setPoint* and the actual high voltage *ActualHV* when the *setPoint* value equal 0. The second part represents the implication relation between them when their values are above 0.

$$(setPoint = 0 \implies ActualHV = 0) \equiv \neg(setPoint = 0) \vee (ActualHV = 0)$$

$$(setPoint > 0 \implies (ActualHV > 0) \equiv \neg(setPoint > 0) \vee (ActualHV > 0)$$

Both sides of the equivalence relation can be verified by truth tables. However, the relation is added only to demonstrate the functionality of the *implies block* in Simulink and make it more sensible. The previously mentioned property P1 was carried out in verification subsystem block in Simulink as seen in Figure 5.2.

In the second property P2, the variable *pow24VStatus* is a boolean input to the Simulink model.

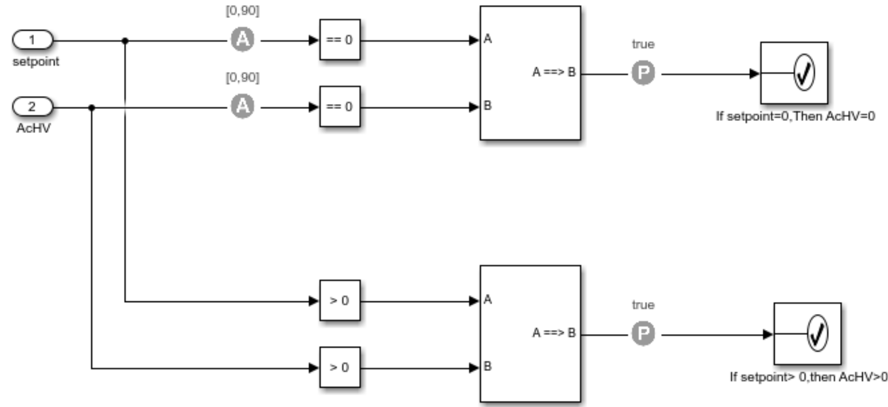


Figure 5.2: Illustrates the implementation of P1 in the verification subsystem block

From Figure 5.3, condition A [*pow24VStatus*==0] has to be true before it implies B. The B statement is a formalization for that *PWM_out* must be 0 when the condition has been met. The interpretation for A implies B according to the model is that the *PWM_out* must be equal to 0 if the input variable *pow24VStatus* is 0.

In logical math property P2 can be formulated as:

$$(pow24VStatus\ =\ 0\ \implies\ PWM_out = 0) \equiv \neg(pow24VStatus\ =\ 0) \vee (PWM_out = 0)$$

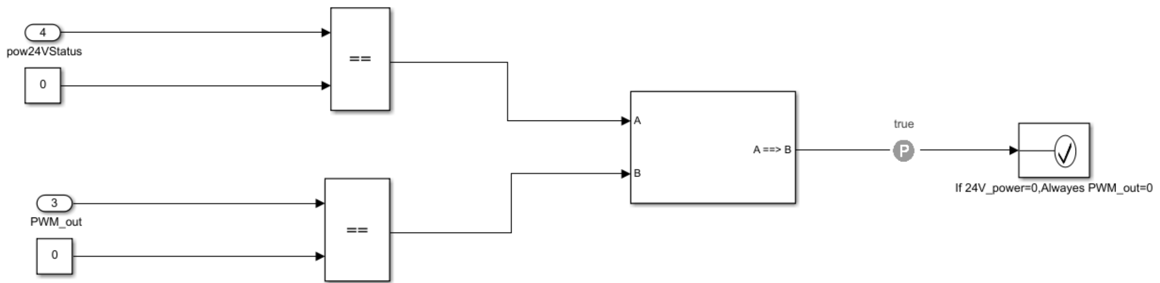


Figure 5.3: Formalization of second property P2 in the verification subsystem block

The last official requirement in the thesis is P3. It is to check whether the system is deadlock free or not. It is not possible to check for deadlock freedom in Simulink Design Verifier, but reachability verification for the states is possible. The reason for this impossibility is because the Simulnik Design Verifier can not handle non-deterministic systems as was explained in Subsection 3.1.2.

The successful proofs for the previously mentioned assertions motivates the group to conduct more verification to discover more of Simulink's capabilities in the field of formal verification. These tests are to prove to what extent the model is accurate and to examine hidden errors. The next requirement to be verified is related to the *setPoint* and *newsetpoint* in the model.

Ramping the *newsetpoint*t in the C++ code is achieved by assigning the value of *newsetpoint*

to *msetpoint* and ramp it gradually within a limited time. The exception happens when the *newsetpoint* value is equal to zero, then the value of *msetpoint* must be reset to zero and accordingly the *setPoint* is set to zero and no change occurs. In the model the same strategy is followed and implemented. If the value of the *newsetpoint* equals zero, the *setPoint* must equal zero. Figure 5.4 illustrates the formalization of the above mentioned property. Though the value of the *setPoint* is initialized between 0 and 90, its value equals 0 when the *newsetpoint* is set to 0.

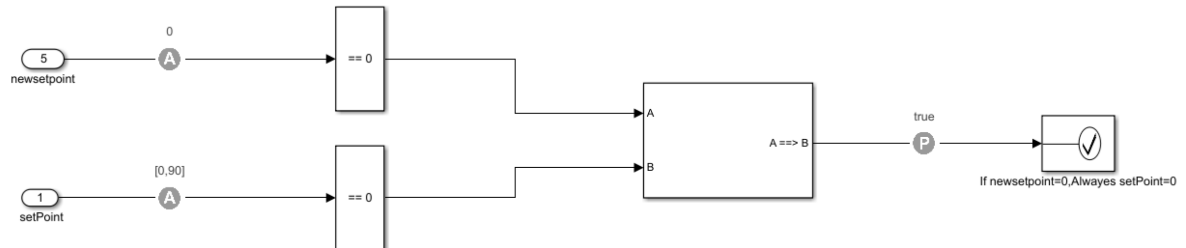


Figure 5.4: Newsetpoint and Setpoint property Assertion

5.2 Verification in RoboTool and FDR4

Verification of RoboChart models is done using the FDR4 refinement checker. The assertions for the model are written in the *assertions* file in the RoboTool project. FDR4 only accepts files of the type *CSP*, when saving the *assertions* file, it will automatically be compiled into *CSP* files, seperating timed and untimed assertions. The semantics contain some built in assertions for checking properties such as reachability and deadlock. One of the properties of interest in the thesis is that the model should be deadlock free, specified in property P3. The reachability property is also important, if not all the states were reachable, that could be an indication of a flaw in the design. The built in assertions written for checking deadlock and reachability and how they appear in the *assertions* file is shown in Figure 5.5.

```
82 //Checks if the model is deadlock free
83 assertion A0 : State_machine is deadlock-free
84
85 //Checks for reachability of states:
86 assertion A1 : State_machine::Ramping is reachable in State_machine
87 assertion A2 : State_machine::Init is reachable in State_machine
88 assertion A3 : State_machine::Wait24Vpower is reachable in State_machine
89 assertion A4 : State_machine::ClosedLoop is reachable in State_machine
90 assertion A5 : State_machine::ErrorMode is reachable in State_machine
91
```

Figure 5.5: The built in assertions in the *assertions* file

These assertions are automatically compiled to *CSP*, in the *CSP* file they will appear as in Figure 5.6, which shows the deadlock as well as the two first reachability assertions. Though it is possible to manually edit the *CSP* file, it is quite apparent that it is a lot easier to edit the assertions file and let the *CSP* syntax be compiled automatically.

```
1 -- A0 - deadlock free
2 assert let
3     id__ = 0
4
5
6
7 within
8     State_machine::O__(id__) :[deadlock free]
9
10 -- A1 - reachable
11 assert not STOP [T= let
12     id__ = 0
13
14
15
16 within
17     State_machine::VS_O__(id__) |\ {|State_machine::enteredV.State_machine::SID_State_machine_Ramping|}
18
19 -- A2 - reachable
20 assert not STOP [T= let
21     id__ = 0
22
23
24
25 within
26     State_machine::VS_O__(id__) |\ {|State_machine::enteredV.State_machine::SID_State_machine_Init|}
27
28 -- A3 - reachable
29 assert not STOP [T= let
30     id__ = 0
31
```

Figure 5.6: The built in assertions as they appear in the *CSP* file

The built in assertions are trivial and easy to check, but verifying these properties are quite important, which is why they have to be verified. Figure 5.7 shows that these assertions were verified in FDR4. In Section 4.3, it is mentioned that not all of the non-trivial properties can be verified in

the current model, because of how it is designed. With certain changes they could be, but due to changes in the model there has not been time to make the necessary changes. As a proof of concept, a few custom *CSP specifications* have been written and asserted to FDR4. A *CSP specification* describes a custom behaviour or property. These need to be written as an *event* followed by a *process*, described in the CSP language as "$e \rightarrow P$", pronounced "e then P", where e is an event an P is a process.

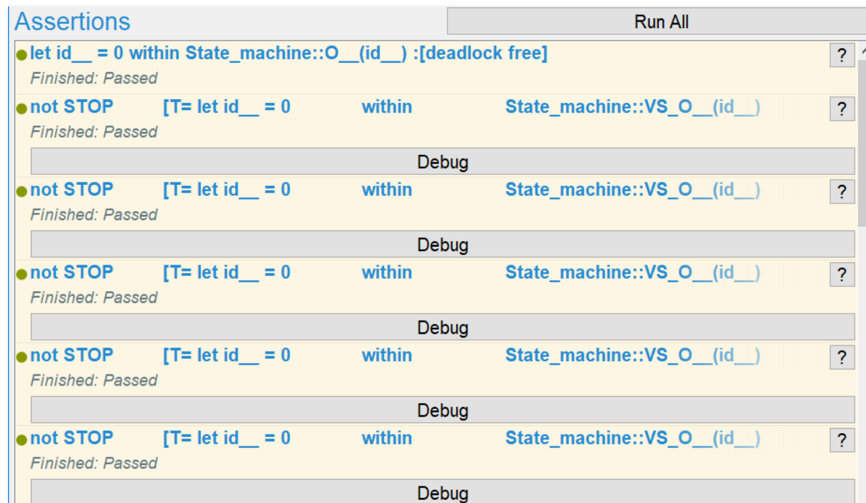


Figure 5.7: Shows that the built in assertion has been successfully verified by FDR4

Part of property P1 requires to check that when the *setPoint* is set to 0, *ActualHV* is set to *Off*. The *ClosedLoop* state may not be representative enough for the actual system. The property described in this paragraph is one of two parts used to verify whether *ActualHV* is following the *setPoint*. To verify this first part, the property describes that when the *setPoint* variable becomes 0, the *ActualHV* will be set *Off*. This represents the safety critical part of the requirement. This specification can be expressed as in Figure 5.8. Some of the steps described in the next paragraphs can be applied to several of the assertions.

```
183
184 untimed csp ErrorSpec7 csp-begin
185
186 ErrorSpec7 = CHAOS(Events) [| {|State_machine::set_setPoint.0|} |> State_machine::set_ActualHV.Power_Off -> ErrorSpec7
187
188 csp-end
189
```

Figure 5.8: Assertion for checking when setPoint is set to 0 it is followed by ActualHV being set off

Separate internal states are used within the *ErrorMode* state to set the values for *setPoint* and *ActualHV*. Asserting this specification on the *State_machine* using the *Failures-divergences* model fails and produces the counter example in Figure 5.9.

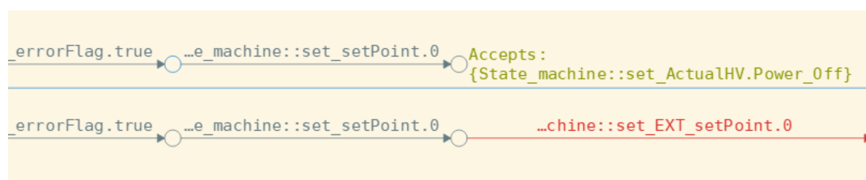


Figure 5.9: Counter example of the setpoint assertion

The counter example shows the instances where the specification is met, highlighted in green, and the situation where it is not met in red. The situation where the assertion fails is when the the

event *set_EXT_setPoint.0* occurs after the event *set_setPoint.0*. Since such external event calls does not interfere with the behaviour in the specification, another assertion is run while hiding the external events. This can be done by defining the formal CSP specification describing the relevant events and then specifying the events of interest, all events in the list except for the specified events of interest will be excluded in the analysis. The specification is shown in Figure 5.10.

```
129
130 untimed csp SMHiddenEvs csp-begin
131
132 SharedVarEvs = {| State_machine::set_EXT_currentState, State_machine::set_EXT_setPoint, State_machine::set_EXT_powStatus, State_machine::set_EXT_lim, State_machine::set_EXT_res, State_machine::set_EX
133
134 SMHiddenEvs = (State_machine::O__(0) [| SharedVarEvs |] SKIP) |\ {| State_machine::set_powStatus, State_machine::set_setPoint, State_machine::set_errorFlag |}
135
136 csp-end
137
```

Figure 5.10: The specification that hides events that are not of interest

SharedVarEvs contains the relevant events in this process, while *SMHiddenEvs* specifies which events are of interest. This is however not enough for the assertion to pass, A counter example is produced in FDR4 with an infinite sequence of τ events, which is an internal event call, the infinite τ event call is a divergence. As mentioned in Section 3.3.1 an assertion is by default tested on the *failures-divergences* model unless any other models has been specified. This is not relevant to the specification of interest, therefore the assertion can be tested using the *Failures* model, where it does pass. Figure 5.11 shows the different assertions mentioned and Figure 5.12 shows how FDR4 displays whether an assertion has passed or failed.

```
217
218 untimed assertion A19 : State_machine refines ErrorSpec7
219 untimed assertion A20 : SMHiddenEvs refines ErrorSpec7
220 untimed assertion A21 : SMHiddenEvs refines ErrorSpec7 in the failures model
```

Figure 5.11: The assertions for the setPoint specification

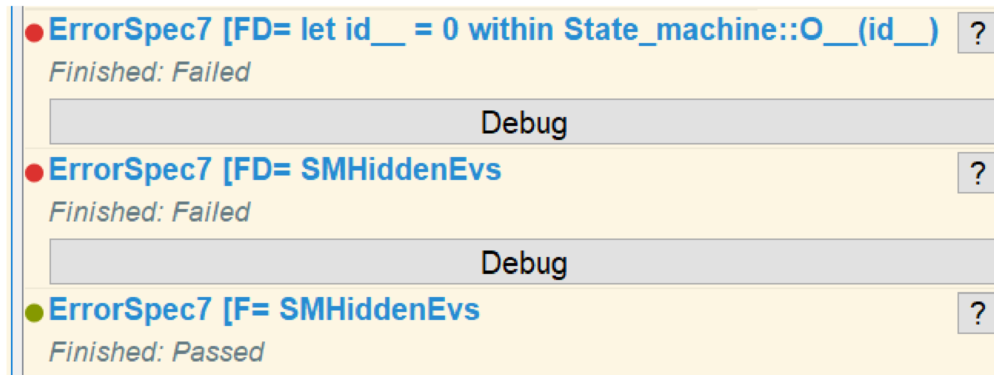


Figure 5.12: Shows assertion results for one of the *setPoint* assertions displayed in FDR4

Counter examples can only be produced if an assertion fails. The *Debug* button in Figure 5.12 brings up the counter example. By default, all internal events are displayed as τ events, which can make the counter example difficult to interpret, these can be hidden from the display by de-selecting the *View Taus* check box. A counter example displaying τ events is shown in Figure 5.13

Part of property P1 requires checking whether *ActualHV* is turned off after the *setPoint* is set to 0. This is one part of the specification stating that the actual voltage should always follow the *setPoint*. Since numerical modelling is not realistically feasible because the FDR4 computing time increases with the complexity of the model, this can not be checked directly. However, this behavior is still represented in the model. The second part of the specification is to verify that when the *setPoint* is set to 1, the power should be turned on. The specification for this is written as shown in

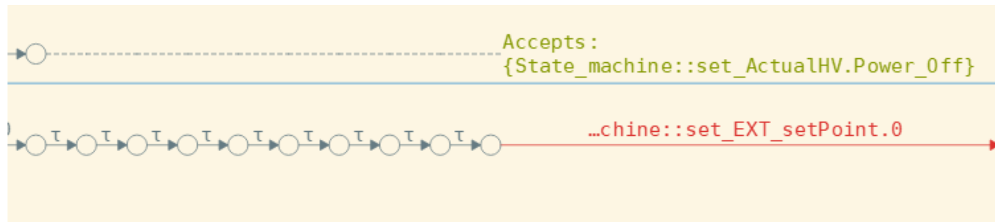


Figure 5.13: Counter example displaying τ events

Figure 5.14. In order for the assertion to pass the verification, the hidden events had to be used and the assertion was tested on the *failures* model. How the different custom assertions were written based on the custom specifications is shown in Figure 5.15. The assertions tested on *SMHiddenEvs* uses hidden events in the analysis. The assertions where no model has been specified, are tested on the *failures-divergences* model. Some of the assertions in Figure 5.15 are not discussed in the report, this is because some of these are used in order to analyse the model or simply test the capabilities of FDR4. Figure 5.16 shows that both assertions about *setPoint* tracking passes in FDR4.

```
155
156 //Checking that when the set point is set to 1, it is followed by the power being set to Power_on
157 untimed csp ErrorSpec4 csp-begin
158
159 ErrorSpec4 = CHAOS(Events) [| {|State_machine::set_setPoint.1|} |> State_machine::set_powStatus.Power_On -> ErrorSpec4
160
161 csp-end
```

Figure 5.14: Specification describing that after the event *set_setPoint.1* happens, it should be followed by *set_powStatus.Power_On*

```
203 untimed assertion A9 : State_machine refines ErrorSpec2
204
205 untimed assertion A10 : State_machine refines ErrorSpec3
206 untimed assertion A11 : SMHiddenEvs refines ErrorSpec3 in the failures model
207 untimed assertion A12 : State_machine refines ErrorSpec3 in the failures model
208 untimed assertion A13 : SMHiddenEvs refines ErrorSpec3
209
210 untimed assertion A14 : State_machine refines ErrorSpec4 in the failures model
211 untimed assertion A15 : SMHiddenEvs refines ErrorSpec4 in the failures model
212
213 untimed assertion A16 : State_machine refines ErrorSpec5 in the failures model
214
215 untimed assertion A17 : SMHiddenEvs refines ErrorSpec6 in the failures model
216 untimed assertion A18 : SMHiddenEvs refines ErrorSpec6
217
218 untimed assertion A19 : State_machine refines ErrorSpec7
219 untimed assertion A20 : SMHiddenEvs refines ErrorSpec7
220 untimed assertion A21 : SMHiddenEvs refines ErrorSpec7 in the failures model
221 untimed assertion A22 : State_machine refines ErrorSpec7 in the failures model
222
```

Figure 5.15: The custom assertions and how they are asserted to FDR4 in the *assertions* file

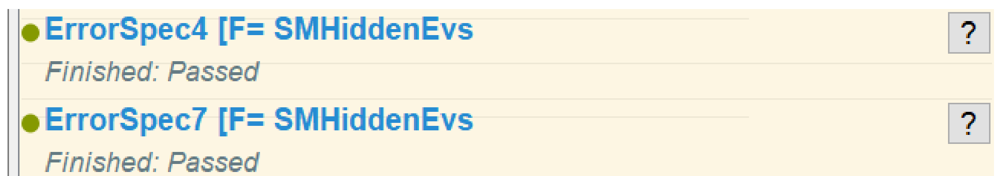


Figure 5.16: Both assertions for checking whether *ActualHV* is tracking *setPoint* passed in FDR4

Chapter 6

Results

The results for Simulink Design Verifier will be presented first, followed by the results from RoboTool and FDR4. The last part will present a few differences between the results of the different tools.

6.1 Simulink Design Verifier Results

The Simulink Design Verifier verifies all possible paths in the model. It searches for any scenario that can disprove a property and if it is the case, Simulink Design Verifier will generate a counter example.

Simulink Design Verifier examines all inputs of the system against all possible trajectories in the model. The result of the verification of a property is either valid or disproved. Simulink Design Verifier issues a report including all details from the verification process. In Chapter 5, several properties have been formally written in Simulink Design Verifier. The next step is to conduct the verification process. The *Prove Properties* button in the tool bar in Simulink Design Verifier starts the formal verification process. The user has the ability to switch between *error detection mode* or *property proving mode*. Moreover, in the *error detection mode* several options are available. For instance, the check can be held only for dead logic (reachability, not deadlock) detection, division by zero or choose and select all types of error detection in Simulink Design Verifier.

The formal verification result for the properties in Chapter 5 will be presented in this chapter. If an assertion block turns green, this means the property is verified against the requirement and it is valid.

In Figure 6.1, the first property P1 from Chapter 5 is verified and proved valid.

Figure 6.2 is the result of the formal verification for the second property P2 from Figure 5.3 in Chapter 5.

The next property to prove according to what has been introduced in Chapter 5 is that when *newsetpoint* equals 0, the *setPoint* is 0. The *newsetpoint* input value has been constrained to 0

while the value of *setPoint* has been investigated for the whole range from 0 to 90KV.

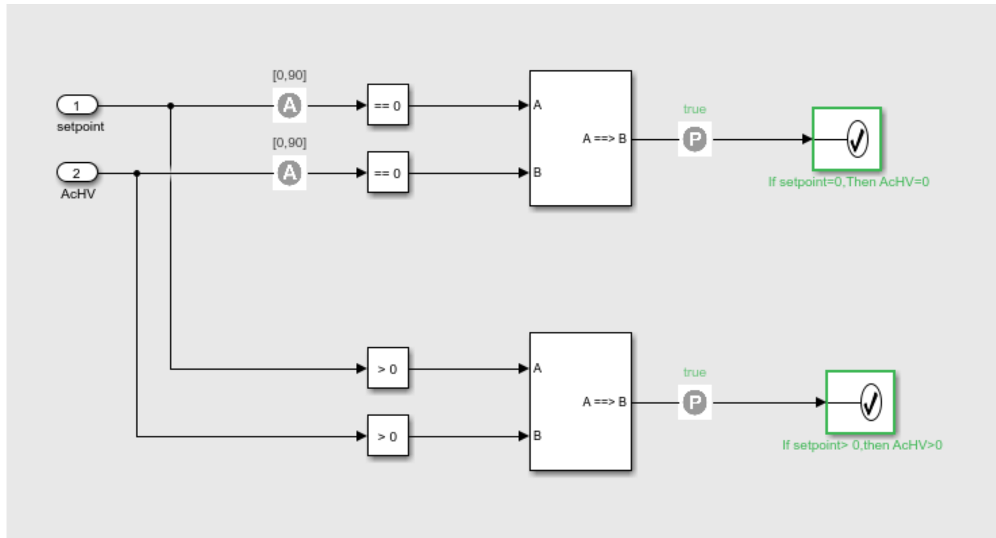


Figure 6.1: Proof of P1 - *ActualHV* follows *setPoint*

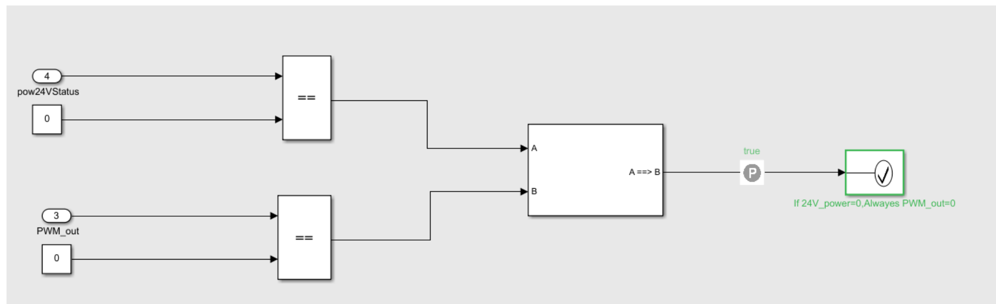


Figure 6.2: Proof of P2 - *PWM_out* is being turned off when *pow24VStatus* is disabled

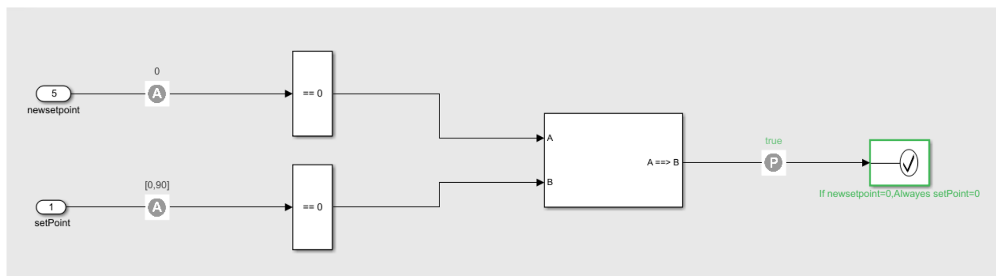


Figure 6.3: *setPoint* follows *newsetpoint* proof

As was mentioned in Chapter 5, Simulink Design Verifier is not able to detect the deadlock and therefore P3 can not be verified. Instead of that it detects the dead logic in the error mode and hence the result of the errors mode covers the dead logic, division by zero and an integer overflow error detection.

6.1.1 Results from Error Detection Mode

Error detection was performed in stages. Firstly, division by zero error detection was selected from the error detection settings in the tool bar, after that an integer overflow error check was used. Lastly, the dead logic error and identifying active logic boxes was ticked. The reason to detect the errors in stages is to investigate each type of errors individually and hence increase the efficiency of debugging the errors.

Because the model does not have any mathematical relation on the form of numerators and denominators, division by zero error detection is considered trivial. As a result, from the dialogue box in Figure 6.4, it is seen that there are not any object in the model to be processed. Like division by zero errors, there are no integer overflow errors in the model and the processed objectives are zero.

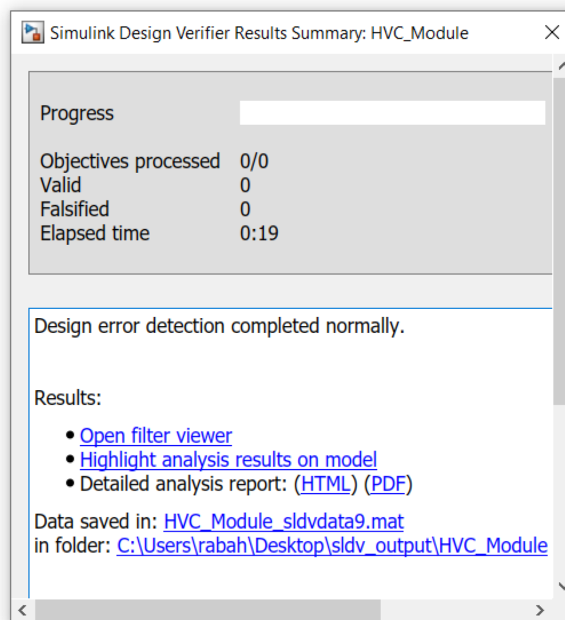


Figure 6.4: Result of the error detection for an integer overflow and division by zero errors

Reachability in Simulink Design verifier is immensely affected by the inputs to the model, parameters configuration and constraints on the states. Figure 6.5 shows the number of the dead logic-reachability in the model when the input variable *pow24VStatus* set to 0.

The number of dead logic is reduced dramatically when the variable *pow24VStatus* is on as it seen in Figure 6.6

The variable *pow24VStatus* is an input to the model and a transition condition between two states in the model, so when the condition is not true the transition does not occur and SLDV considers that the rest of the states are dead logic as Figure 6.5 illustrates. Moreover, reducing the constraints on states leads to less dead logic in the model.

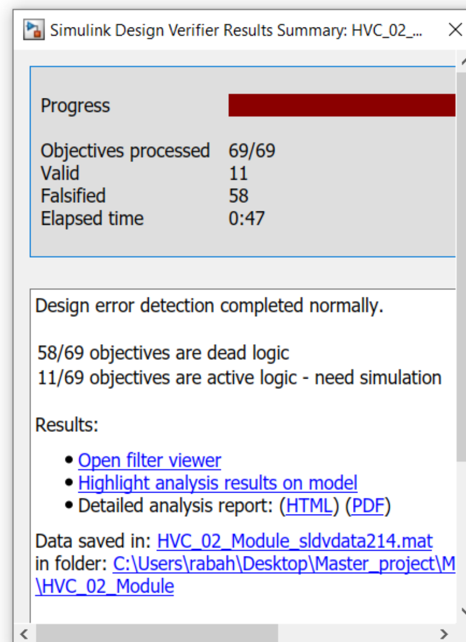


Figure 6.5: Errors detection when 24V Power is Off

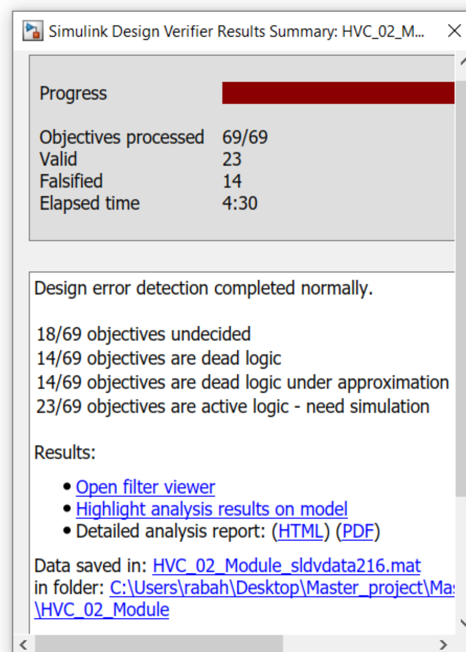


Figure 6.6: Errors detection when 24V Power is On

6.1.2 Usability Remarks

Simulink Design Verifier only supports discrete solvers and hence the Simulink model has to be discrete. As a result, Simulink Design Verifier is not applicable to continuous time blocks from Simulink. The first model of the system used continuous time blocks. For instance, the PID controller was of the continuous time type. The compatibility error is recognized when using SLDV with the continuous time controller. Similarly, more blocks such as the continuous integrator can not be used with SLDV. The above mentioned problem is considered a practical limitation of SLDV.

Simulink Design Verifier was not able to differentiate between the constants and the variables in the denominator. Therefore, many division by zero errors are detected by Simulink Design Verifier even when the denominator has a constant value in it.

6.2 FDR4 results

In FDR4, two of the main three properties were verified. Property P1 from Chapter 5 was that the *ActualHV* should follow the *setPoint*. This assertion was verified using two custom specifications. The variables were modelled as being either on or off, due to the fact that dynamic simulation is not possible in RoboChart. Using variables in this way is also more convenient for handling the specifications. Property P3 was that the system should be deadlock free, which was verified using the built-in semantics in RoboTool. The reachability of the states was also verified in order to verify that that all possible states could be reached, and that the transitions were built correctly. Figure 6.7 shows the verification of the state machine being deadlock-free and it also shows the reachability for all the states. Figure 6.8 shows that the custom specifications for the *ActualHV* following *setPoint* had to be verified using the failures model with hidden external events. Figure 6.9 shows that both *setPoint* tracking assertions passed in FDR4.

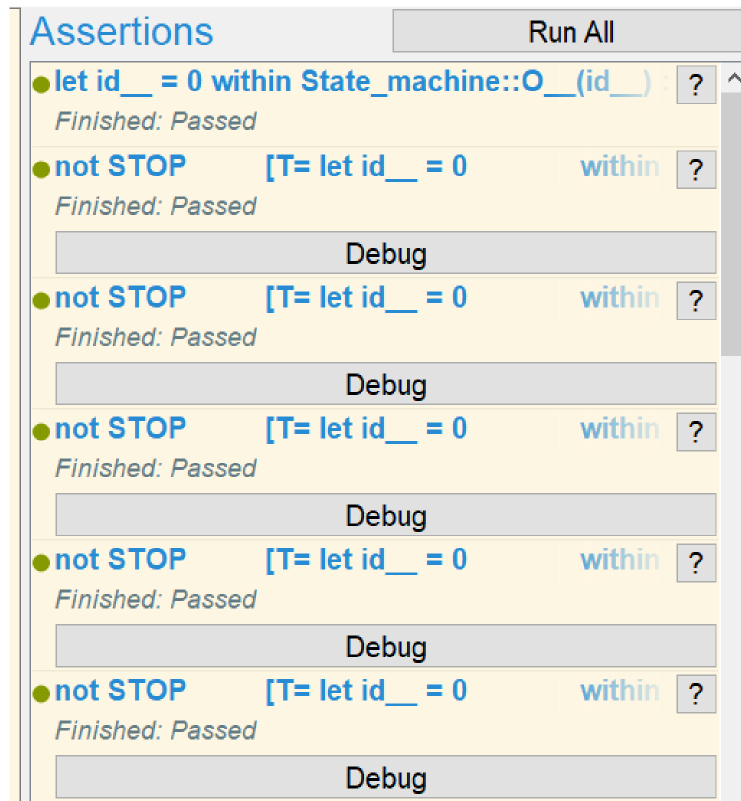


Figure 6.7: Results from the built in assertions, among these are the verification of property P3

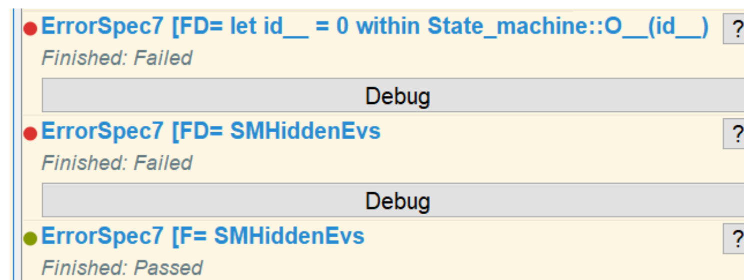


Figure 6.8: Results from the custom setpoint assertions which verifies part of P1

Figure 6.9: Results from the custom set point assertions which verify both assertions for property P1

6.3 Tool comparison

Two tables where made in order to systematically show some key differences in the tools. These differences will be further discussed in Section 7.1. Table 6.1 shows a few differences in capabilities between the verification tools. These capabilities are key factors of interest that was noted about the tools. Table 6.2 shows which properties could be verified within each of the tools. Reasoning for which properties could or could not be verified within each tool is discussed in Section 5.1 and 5.2.

Property	**SLDV**	**RoboTool**
Can check for deadlock	No	Yes
Can check for reachability	Yes	Yes
Allows dynamic simulation	Yes	No
Graphical modelling	Yes	Yes
Allows input-driven modelling	Yes	No

Table 6.1: Tool capability comparison

Verified properties	**SLDV**	**RoboTool**
P1: *ActualHV* tracking *setPoint*	Yes	Yes
P2: *setPoint* is turned off when 24V Power is turned off	Yes	No
P3: Deadlock-free	No	Yes

Table 6.2: Overview of which properties could be verified within each tool

Chapter 7

Discussion and Conclusion

The results from the previous chapter will be discussed here. The first section is general and discusses both tools. The final section of this chapter discusses future work. First future work regarding Stateflow and SLDV is presented, then the same is discussed for RoboTool and FDR4.

7.1 Conclusion

Throughout this report, it is clear that the tools are quite different, both in terms of modelling and verification. Simulink Design Verifier uses the Simulink framework for modelling, which allows for a more input-driven and dynamic way of modelling. This may also come at the cost of the abstraction level of the verification. Introducing an input-based approach means that certain outcomes may be excluded from the analysis if the input variables changes the execution. However, modelling in such a way can be more intuitive for an engineer who is new to formal verification methods, and who has more experience with dynamic simulations. Simulink Design Verifier also offers the capability of creating models of higher abstraction. Having the option of creating more lower level functionality can be helpful. This is something that has benefited the group during this thesis, because the group have had some previous experience with using Simulink for simulation of dynamic models.

RoboTool on the other hand, was created specifically for formal verification of robotic systems. The modelling in the software is therefore highly abstract, and dynamic simulations are not possible, as of yet. This can be a challenge for someone with more experience from simulation based models. This type of modelling can be regarded as the opposite of simulation models. This is because simulation models are input-driven whereas formal verification models are output-driven. The goal in formal verification is to look for a specific type of behavior in the model and to see whether the model satisfies it. Whereas in a simulation model, the input values and dynamics of the system is modelled, and the outputs are analysed.

Both tools posed different challenges, even if SLDV was more similar to previous methods the group had used, it was still challenging to use the tool for model checking. The verification in the different tools also had different capabilities. For instance in the specification for whether *ActualHV* followed the *setPoint*, the specification written in CSP_M for RoboTool could not be directly applied

to SLDV.

7.2 Future work

This section will discuss improvements that could be made in future work with this case study. It is divided so that the first part will discuss improvements that could be made with regards to Stateflow and SLDV, the second part discusses the same for RoboTool and FDR4.

7.2.1 Stateflow and Simulink Design Verifier

The future work for Stateflow and SLDV is spilt into two parts. The first part is about the model and the second is regarding the model checking tool.

The Model

In terms of building the model, the actual system is composed of several state machines that work simultaneously and communicate continuously by exchanging internal messages. The model has been simplified to alleviate the complexity of it and reduce the required computational power. To simplify, more constraints have been imposed on the main sequential state machine. The functionality of the interrupt state and supply voltage check state are compensated by resetting the *setPoint* in every state before the *wait24Vpower* state. This approach worked properly and led to satisfactory results. However, the previous strategy affects the reachability of the states. Reducing the number of constraints in the model will improve the reachability. One way to reduce the constraints in the model is by adding more details to the model instead of building a plain and highly abstract model.

In terms of accuracy, the behaviour of the model was captured from the given C++ code. The accuracy can be increased by adding more details to the model from the code.

The Verification Tool

Simulink Design Verifier is a toolbox incorporated into the Simulink environment, and has been used to verify the properties. Several issues were raised during the work. For instance, some of the Simulink blocks are not supported by Simulink Design Verifier. As a result, a compatibility error was generated when initializing the verification task. Simulink Design Verifier automatically checks the compatibility, and the result of the automatic check is compatible or not compatible. Sometimes the outcome of the automatic check is partially compatible, the result of the verification will be incomplete or it is not exactly what was expected. The group faced the case of a partially compatible model when trying to add a user-defined function block in the C language. Adding such a block would capture the behavior of the code better than converting the code into logic in Stateflow and would save considerable time and efforts during the conversion process.

Stateflow does not support non-deterministic systems and hence deadlock properties cannot be detected. Considering detection of deadlock errors in any future improvement will increase the capability of SLDV in the formal verification. However, PolySpace which is another toolbox from MathWorks, can be used in combination with Simulink Design Verifier to catch deadlock errors. C code can be generated from the model which can be verified for deadlock errors by using PolySpace.

Formalizing several properties at the same time requires a careful approach. If the properties contradict each other, an error will be generated and the process stops. The group faced this problem where in many instances, all properties were not satisfied because one of them contradicts the other during the property formalization and no counter example was issued.

Working to improve and solve the previously mentioned points will lead to more accurate results and will increase the flexibility of using the tool.

7.2.2 RoboTool and FDR4

For future work with the model in RoboTool, the model should be expanded in order to enable verification of the property that could not be verified. This is property P2 from Chapter 5. In order to get this verification to work, a variable representing the PWM_output signal would have to be implemented. Additionally, the system does not fully capture how the system behaves based on changes in the 24V power signal. Since the verification time quickly increases as the model is expanded, special care is necessary in order to not introduce too much of the type of behaviour that highly increases the verification time.

Another part of the model that is not quite complete, is that only the sequential part of the system has been modelled. The *watchdogs* continuously running concurrently with the sequential part have not been modelled. This also affects property P2, as the concurrent monitors can affect the transitions regarding the specification. In order to introduce this behaviour, the three different monitors would be modelled as separate state machines which would communicate with the main state machine through events. The main task of these monitors are to detect errors, and cause the system to transition to the *ErrorMode* state.

Acknowledgements

First of all would like to acknowledge our supervisors David Anisi and Yvonne Murray and thank them for their continuous support and supervision during our work with the thesis. Through weekly meetings they have helped us focus our aim to complete the thesis in a satisfactory manner.

We would also like to thank ABB Robotics for presenting the thesis. They have provided us with the necessary documentation and information about the system. ABB has also been providing guidance to help us better understand the functionality of the system and the C++ code that has been crucial to building the models.

We would also like to thank Pedro Ribeiro from the University of York, who is one of the developers of RoboTool. Through e-mail communication and Skype meetings, he has answered questions about the software and guided us in utilizing the tool properly. These guided sessions often led to new changes in the model and giving the group a better understanding of the tool's capabilities.

Bibliography

[1] *ABB HVManul pdf.* ABB.

[2] *ABB IPS HVC Manual pdf.* ABB.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of model checking.* The MIT Press, Cambridge, Massachusetts. URL: http://is.ifmo.ru/books/_principles_of_model_checking.pdf.

[4] Edmund M Clarke Jr et al. *Model checking book 2nd edition.* 2018.

[5] *Communicating Sequential Processes.* C.A.R. Hoare. URL: http://www.usingcsp.com/cspbook.pdf.

[6] *CSPM manual.* Oxford university. URL: https://cocotec.io/fdr/manual/cspm.html.

[7] *CTL.* A video lecture about CTL. 2015. URL: https://www.youtube.com/watch?v=Blh060Hgbm8&list=PLK50zIm6tHRiKFJvKu1a7q_z2tcXnBUHp&index=43.

[8] *Failures-Divergences Refinement.* University of Oxford, 2012. URL: https://www.cs.ox.ac.uk/projects/concurrency-tools/download/fdr2manual-2.94.pdf.

[9] Thomas Gibson-Robinson et al. *Failures Divergences Refinement (FDR) Version 3.* 2013. URL: https://www.cs.ox.ac.uk/projects/fdr/.

[10] Thomas Gibson-Robinson et al. 'FDR3 — A Modern Refinement Checker for CSP'. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by Erika Ábrahám and Klaus Havelund. Vol. 8413. Lecture Notes in Computer Science. 2014, pp. 187–201.

[11] *Introduction to LTL.* A video introduction to linear temporal logic. 2015. URL: https://www.youtube.com/watch?v=W5Q0DL9plns&list=PLK50zIm6tHRiKFJvKu1a7q_z2tcXnBUHp&index=33.

[12] Steven T. Karris. *Introduction to Simulnik with Engineering Application.* 2006.

[13] William K. Lam. *Simulation-Based Verification versus Formal Verification.* 2005. URL: https://www.informit.com/articles/article.aspx?p=392278&seqNum=4.

[14] Edmund M.Clarke et al. *Handbook of Model Checking.* Springer, Cham. URL: https://link.springer.com/book/10.1007/978-3-319-10575-8.

[15] Mathworks. *Algorithm Verifcation and Validdation in Matalb.* URL: https://www.mathworks.com/videos/algorithm-verification-and-tool-validation-in-matlab-1500578151023.html.

[16] Mathworks. *Simulnik design verifier catalog.* URL: https://pdf.directindustry.com/pdf/mathworks/simulink-design-verifier/12865-435443-_3.html.

[17] MathWroks. *Simulink design verifer*. URL: https://www.mathworks.com/products/simulink-design-verifier.html.

[18] MathWroks. *Statflow and Stateflow coder*. URL: https://edulab.unitn.it/nfs/Manualistica/Software/MathWorks%20Guide/stateflow/sf_ug.pdf.

[19] Alvaro Miyazawa et al. *Robochart reference*. Robostar, University of york. URL: https://www.cs.york.ac.uk/circus/publications/techreports/reports/robochart-reference.pdf.

[20] Alvaro Miyazawa et al. *Robochart reference manual*. Robostar, University of york. URL: https://www.cs.york.ac.uk/circus/RoboCalc/assets/RoboChart-manual.pdf.

[21] PCI. *The Next Big Thing in Robotic Painting: Smart Atomizers*. URL: https://www.pcimag.com/articles/106518-the-next-big-thing-in-robotic-painting-smart-atomizers.

[22] *Robostar notation tools*. Robostar, University of york. URL: https://www.cs.york.ac.uk/robostar/notations-tools/.

[23] Bryan Scattergood and Philip Armstrong. *CSPM : A Reference Manual*. Bryan Scattergood, Philip Armstrong. URL: http://www.cs.ox.ac.uk/ucs/cspm.pdf.

[24] *Semantics of LTL*. A video about the semantics of LTL. 2015. URL: https://www.youtube.com/watch?v=TLr0wq-8iDs&list=PLK50zIm6tHRiKFJvKu1a7q_z2tcXnBUHp&index=34.

[25] Shobhit Shanker and Prashant Hedge. *Improving the quality of complex control logic design using model verification*. URL: https://www.mathworks.com/content/dam/mathworks/mathworks-dot-com/solutions/automotive/files/in-expo-2013/improving-the-quality-of-complex-control-logic-design-using-model-verification-and-validation-techniques.pdf.

List of Figures

1.1 Robotic paint system from ABB[21] 1

1.2 Electrostatic painting principle[1] 2

1.3 Finite state diagram of the High Voltage Controller (HVC) 3

1.4 High voltage control system unit[1] 3

1.5 Block diagram of one part of the paint robot, containing the HVC[1] 3

1.6 Error concerning *HV.Actual* value 5

1.7 Error when the 24V power signal failed and the HVC froze 6

1.8 Error when the 24V power signal was falsely reported missing 6

2.1 Flowchart showing the verification process 8

2.2 Finite state machine representation 9

2.3 Representation of sequential states 10

2.4 Representation of a general model checking procedure additionally showing how it is a complement to simulation 12

2.5 Figure showing examples of linear executions that satisfies the requirements described on the left[11] 13

2.6 Example of a computational tree extracted from a state machine[3] 14

3.1 Formal verification process in Mathworks[15] 17

3.2 Representation of the Stateflow environment[18] 18

3.3 Simulink Design Verifer functions 18

3.4 Demonstration of the architecture of the verification subsystem block and Simulink sub-model 19

3.5 Design error detection progress box 20
3.6 Error detection and model highlighting[25] 20
3.7 Safety property in Simulink Design Verifer 22
3.8 The graphical user interface of RoboTool 22
3.9 Examples of interfaces and a robotic platform 23
3.10 Example of an *operation* 24
3.11 Example of a state diagram, showing some of the properties described in Chapter 3.2 25
3.12 The interface of FDR4, with assertions in the right side margin 25

4.1 Representation of a general modelling procedure using model checking and visual representation of how model checking is related to simulation 29
4.2 The model blocks configuration in the Simulink Environment 30
4.3 The *Init* state in flowchart 31
4.4 Matlab Body's Function Block 31
4.5 DisableHV function 32
4.6 Wait24Vpower State 32
4.7 ClosedLoop State 33
4.8 The Structure of CheckLimit Function 33
4.9 RampingsetPoint State 34
4.10 ErrorMode State State 35
4.11 The Controller 35
4.12 The original version of the ramping state 36
4.13 The new version of the ramping state 37
4.14 The old version of the disableHV operation 37
4.15 Demonstrates where the DisableTrue event is used for a transition 38
4.16 The new version of the disableHV operation 38
4.17 Demonstrating the boolean used for a transition 39
4.18 Visual representation of the check limits operation 39

4.19 Visual representation of the supplyVoltageCheck operation 40

4.20 Representation of the *ClosedLoop* state 40

4.21 Representation of the enumerated interfaces for *State* and *Power* 41

4.22 Demonstration of the *ErrorMode* state 42

4.23 An overview of the state machine and its transitions 42

4.24 The different *interfaces* used in the RoboChart model 43

4.25 Displays the architecture of the RoboChart model 43

5.1 Inputs to the verification subsystem block in the model 45

5.2 Illustrates the implementation of P1 in the verification subsystem block 46

5.3 Formalization of second property P2 in the verification subsystem block 46

5.4 Newsetpoint and Setpoint property Assertion 47

5.5 The built in assertions in the *assertions* file 48

5.6 The built in assertions as they appear in the *CSP* file 48

5.7 Shows that the built in assertion has been successfully verified by FDR4 49

5.8 Assertion for checking when setPoint is set to 0 it is followed by ActualHV being set off 49

5.9 Counter example of the setpoint assertion 49

5.10 The specification that hides events that are not of interest 50

5.11 The assertions for the setPoint specification 50

5.12 Shows assertion results for one of the *setPoint* assertions displayed in FDR4 50

5.13 Counter example displaying τ events 51

5.14 Specification describing that after the event *set_setPoint.1* happens, it should be followed by *set_powStatus.Power_On* 51

5.15 The custom assertions and how they are asserted to FDR4 in the *assertions* file . . . 51

5.16 Both assertions for checking whether *ActualHV* is tracking *setPoint* passed in FDR4 51

6.1 Proof of P1 - *ActualHV* follows *setPoint* 53

6.2 Proof of P2 - *PWM_out* is being turned off when *pow24VStatus* is disabled 53

6.3 *setPoint* follows *newsetpoint* proof 53

6.4 Result of the error detection for an integer overflow and division by zero errors . . . 54

6.5 Errors detection when 24V Power is Off 55

6.6 Errors detection when 24V Power is On 55

6.7 Results from the built in assertions, among these are the verification of property P3 57

6.8 Results from the custom setpoint assertions which verifies part of P1 57

6.9 Results from the custom set point assertions which verify both assertions for property P1 58

List of Tables

6.1 Tool capability comparison 58

6.2 Overview of which properties could be verified within each tool 58