View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Lund University Publications - Student Papers

ISSN 0280-5316
ISRN LUTFD2/TFRT--5907--SE

Modeling of Avionics Systems using JGrafchart and TrueTime

Anna Benktson
Sofia Dahlberg

Lund University
Department of Automatic Control
November 2012

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
November 2012

Document Number
ISRN LUTFD2/TFRT--5907--SE

Author(s)
Anna Benktson
Sofia Dahlberg

Supervisor
Eelco Scholte, UTC
Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Sponsoring organization

Title and subtitle
Modeling of Avionics Systems using JGrafchart and TrueTime (Modellering av flygsystem med JGrafchart och TrueTime)

Abstract
The first part of the thesis aims to investigate the applicability of JGrafchart and its associated Model of Computation (MoC) for describing sequential control in aircraft primary power distribution systems. The motivation behind this is the need for better modeling tools and in particular support for separation between nominal control and fault handling. Also, as system complexity increases, better structuring capabilities are required. The application for this part of the thesis is a typical primary power distribution system in a medium-sized aircraft, and JGrafchart is used as substitute for Stateflow for the sequential parts of the controller. Simulations were run to determine whether JGrafchart is suitable for these types of systems, and if it provided any additional value compared to Stateflow.

The second part focus around a different tool (TrueTime) to help assess the impact of embedded architecture on control performance. Today it is common for systems to be distributed over multi-tasking kernel nodes, which communicate on different networks. In these systems the nodes compete for the shared resources (The CPU and bandwidth) and the distribution of bandwidth is determined by the network protocol. Since the shared resources are limited in terms of bandwidth different kinds of delays arise, such as transmission delays and back-off times. The delays might lower the control performance significantly, which is why it is important to identify them early in the development process, preferably at the design stage. In the thesis, TrueTime is extended to support Avionics Full Duplex Switched Ehternet (AFDX) and applied to a typical aircraft electric power system.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-88

Recipient's notes

Security classification

http://www.control.lth.se/publications/

Modeling of Avionics Systems using JGrafchart and TrueTime

Anna Benktson and Sofia Dahlberg

December 3, 2012

Contents

1 Introduction **10**
1.1 Background . . . 10
1.2 Thesis Outline . . . 10
1.3 Related Work . . . 11
1.4 Individual Contributions . . . 11

I Sequential Control Systems **12**

2 Primary Power Distribution Systems **14**
2.1 Background . . . 14
2.1.1 Sources . . . 14
2.1.2 Electrical Components . . . 16
2.1.3 Control Algorithm . . . 16

3 Models of Computation for Sequential Control Systems **18**
3.1 Finite State Machine . . . 18
3.1.1 Moore Machine . . . 18
3.1.2 Mealy Machine . . . 18
3.2 Petri Net . . . 20
3.2.1 Background . . . 20
3.3 Different types of Petri nets . . . 23
3.3.1 Colored Petri nets . . . 23
3.3.2 Object Petri nets . . . 23
3.4 Stateflow . . . 25
3.4.1 State . . . 25
3.4.2 Transition . . . 25
3.4.3 Connective Junction . . . 27
3.4.4 History Junction . . . 27
3.5 Grafcet . . . 27
3.5.1 Step . . . 27
3.5.2 Transition . . . 27
3.5.3 Branching . . . 29
3.5.4 Macro Step . . . 29
3.6 Sequential Function Charts . . . 29
3.7 JGrafchart . . . 30
3.7.1 Background . . . 30
3.7.2 Step . . . 30
3.7.3 Transition . . . 31
3.7.4 Macro Step . . . 31
3.7.5 Exception Transition . . . 31
3.7.6 Procedure Step . . . 33

3.7.7 Process Step . . . 33
3.7.8 Step Fusion Set (SFS) . . . 33
3.7.9 Object-Oriented Features . . . 34
3.8 Choosing a Model of Computation . . . 34

4 Application of JGrafchart 36
4.1 Motivation . . . 36
4.2 An Existing Control System . . . 36
4.2.1 Stateflow blocks . . . 36
4.2.2 Interfaces . . . 37
4.3 JGrafchart Control System Architecture . . . 37
4.3.1 Sequential Control . . . 37
4.3.2 Sequential Control Integrated With Fault Handling . . . 38
4.3.3 Modifications Made to the JGrafchart Tool During Implementation . . . 50
4.4 Results . . . 53

5 Conclusion and Future Work 54
5.1 General Discussion of Results . . . 54
5.2 Future Work . . . 55

II Modeling of IMA Control Systems 58

6 Integrated Modular Avionics 60
6.1 Background and Introduction to IMA . . . 60
6.2 Theory . . . 60
6.2.1 Network Protocols . . . 60

7 TrueTime 64
7.0.2 TrueTime . . . 64

8 TrueTime Model of IMA Use Case 66
8.1 An Existing IMA System . . . 66
8.2 System modeled in TrueTime . . . 67
8.2.1 System Modeled With Full Duplex Switched Ethernet . . . 67
8.2.2 Implementation of AFDX . . . 67
8.3 Results . . . 70
8.3.1 System Modeled With Full Duplex Switched Ethernet . . . 70
8.3.2 System Modeled With AFDX . . . 70

9 Conclusion and Future Work 80

List of Figures

2.1 Typical Single Line Diagram 15
2.2 Priority Table 16

3.1 Moore machine 19
3.2 Mealy machine 19
3.3 A simple Petri net 21
3.4 A Petri net with corresponding reachability graph 21
3.5 A Petri net with coverability tree (upper) and coverability graph (lower) 22
3.6 Trucks and their corresponding Petri nets 24
3.7 Two identical systems modeled with a colored Petri net 24
3.8 Stateflow 26
3.9 State machine without connective junction (Left) and state machine with connective junction (Right) 27
3.10 A Grafcet chart 28
3.11 Grafcet Macro Step 29
3.12 Sequential Function Chart example 30
3.13 Step and initial step 31
3.14 Transition 31
3.15 Macro Step 32
3.16 Exception transition 32
3.17 Procedure step 33
3.18 Process step 33
3.19 State Machine Before Applying Step Fusion Set 34
3.20 State Machine After Applying Step Fusion Set 34

4.1 Request Handler, top layer 39
4.2 Request Handler, internal states 40
4.3 Configuration of electric system, top layer 41
4.4 Configuration of electric system, internal sequence 42
4.5 JGrafchart Design Concept 43
4.6 Generator State Machine, Top Level 44
4.7 Generator State Machine, Macro Step Body 45
4.8 Generator State Machine, Undervoltage Fault Detection Layer 46
4.9 TRU State Machine, Top Level 47
4.10 TRU State Machine, Macro Step Body 48
4.11 TRU State Machine, TRU Overcurrent Detection 49
4.12 Fault handling layer 50
4.13 State Machine for a normally open contactor 51
4.14 Workspace showing all contactors and their overall status (Error or not Error) 52

5.1 Concept Of Object Tokens In JGrafchart 56

7.1 Aircraft Electric Power System 65

8.1 Simplified Aircraft Electric Power System . . . 67
8.2 Simplified Aircraft Electric Power System modeled in TrueTime . . . 68
8.3 AFDX in TrueTime . . . 69
8.4 Network Schedule and Corresponding Control Performance With 0 % Bandwidth Assigned To High Priority CAN Interference Node . . . 71
8.5 Network Schedule and Corresponding Control Performance With 5 % Bandwidth Assigned To High Priority CAN Interference Node . . . 72
8.6 Network Schedule and Corresponding Control Performance With 10 % Bandwidth Assigned To High Priority CAN Interference Node . . . 73
8.7 Overshoot as A Function of Bandwidth Assigned To High Priority CAN Interference . . . 74
8.8 Control Performance With 1 ms Gateway Delay . . . 74
8.9 Control Performance With 5 ms Gateway Delay . . . 75
8.10 Control Performance With 10 ms Gateway Delay . . . 75
8.11 Overshoot as a Function of Gateway Delay . . . 76
8.12 Node 3 Executing Sporadic Task On Full Duplex Switched Ethernet . . . 77
8.13 Node 3 Executing Sporadic Task on AFDX . . . 78
8.14 Control Performance When Controller BAG is Equal to 1 ms . . . 79
8.15 Control Performance When Controller BAG is Equal to 10 ms . . . 79

List of Tables

4.1 Components and Examples of Respective Faults 41

Acronyms

AFDX Avionics Full Duplex Switched Ethernet

CAN Controller Area Network

EPDS Electric Power Distribution System

FSM Finite State Machine

HS Hamilton Sundstrand

IMA Integrated Modular Avionics

MIMO Multiple Input Multiple Output

MOC Model Of Computation

PN Petri Net

PPDS Primary Power Distribution System

SLD Single Line Diagram

SSRPC Solid State Relay Power Contactor

TDMA Time Division Multiple Access

Chapter 1

Introduction

1.1 Background

Modern aircraft rely on the integration of several systems such as the Electrical System, the Air Management System, Avionics, Vehicle Management and others. Most recent aircraft have seen a drive towards increased integration and more reliance on system control to achieve system functionality. Most control systems are implemented using a combination of hardware and software systems and are designed to work under normal and fault conditions to ensure system performance and availability.

The control system often is distributed across several computational nodes and using communication networks to exchange data in real-time. In the past this was often done using point-to-point communication networks, however recent platforms have started to shift towards high speed bus based architectures. Developing control algorithms for these systems is particularly challenging due to the need for fault handling and degraded performance requirements. It is therefore desirable to have a methodology (and supporting modeling framework) that allows for the designer to approach the design in a modular fashion, rather than a fully integrated approach from the start.

In addition, there is a need to predict the early impact of the embedded architecture on the control performance. The control performance in such Integrated Modular Avionics (IMA) based systems often is impacted by delay and jitter that are introduced by both the computational and communication elements.

To support both types of analysis, two existing toolsets will be evaluated for the application to current and future aircraft systems. These two toolsets, JGrafchart and TrueTime have been developed at Lund University since 1991 and 1999 respectively.

The thesis has been done at UTC Aerospace Systems (legacy Hamilton Sundstrand) in Windsor Locks (CT, U.S) which is a business division within the United Technologies Corporation (UTC). UTC consists of six business divisions. The aerospace businesses are Sikorsky which is the largest helicopter manufacturer in the world, Pratt & Whitney which develops engines, industrial gas turbines and space propulsion systems and UTC Aerospace systems. UTC Aerospace system is a major supplier to aerospace and defense systems as well as to international space programs. The company is the result of the merging of Hamilton Sundstrand and Goodrich, which was completed in 2012. UTC also has commercial businesses which are Otis elevators and UTC Climate, Controls & Security.

1.2 Thesis Outline

This thesis aims to address two applications of control and analysis tools to support development of aircraft control systems:

1. Investigate the applicability of JGrafchart and its associated Model of Computation (MoC) for describing the Primary Power Distribution Control System.

2. Extension of the TrueTime toolbox to typical aircraft control systems that employ Integrated Modular Avionics (IMA).

Within the first part a typical Electric Control System that forms the basis for the investigation of using JGrafchart is described. A brief comparison of different Models of Computation (MoC) is detailed in Chapter 3, and the application to Electric Systems is described in Chapter 4. Conclusions and future work for this part is summarized in Chapter 5.

A similar outline is followed for the second part of this thesis. The IMA concept is described in Chapter 6, followed by a description of the TrueTime background in Chapter 7 and a Use Case and the implementation of the necessary extensions in Chapter 8. The thesis is concluded with Chapter 9 for the second part of the thesis.

1.3 Related Work

Modeling of avionic systems has grown more difficult as system complexity has increased significantly over the past years. It becomes more and more important to construct models designed for verification, since these systems cannot be analyzed without the use of verification tools. Several large industry and government programs have developed methods for both design and verification of complex aircraft systems. Within the More Open Electrical Technologies (MOET) [1] program new modeling methodologies focused on the feasibility of more electric and integrated aircraft are investigated. In [2] the Modelica language is used as a common language for modeling aircraft systems in the different domains. An example of related work in the area of modeling and verification of avionic systems is provided in [3]. The authors propose an automated procedure for designing control protocols using Linear Temporal Logic to correctly describe the behavior of a system and its environment. The motivation behind this is to make the model amenable to formal analysis and create a hierarchical control structure.

1.4 Individual Contributions

The work behind this thesis has been done in close collaboration between the authors, and both have been working with both parts. However, Sofia had the main responsibility for the JGrafchart part whereas Anna had the main responsibility for the TrueTime part. Since the scope of the JGrafchart part exceeds the scope of the TrueTime part, Anna has also been responsible for the work done in Section 4.4.1.

Part I

Sequential Control Systems

Chapter 2

Primary Power Distribution Systems

2.1 Background

In today's aircraft powering is done dynamically by the Electric Power Distribution Systems (EPDS). Depending on which state the aircraft is in (take-off, landing etc) the routing will change accordingly by enabling/disabling a set of switches. In the physical system redundant paths are available for the controller to choose from. The controller has to choose a path and sequentialize the switches in a safe way, for example, it has to guarantee no electric power loss in the system. To describe the behavior of the sequentialized systems, different Models of Computation can be used [5]. Among them are Finite State Machines (FSM) and Petri Nets (PN), both of which have been used at Hamilton Sundstrand (HS) in the past for analysis purposes. The main objective of this part of the thesis is to model a PPDS control system in Simulink/JGrafchart and evaluate the expressiveness of the language, primarily to support fault handling.

An Electric Power Distribution Systems (EPDS) is divided into Power generation, Protections and Primary and Secondary power distribution, PPDS and SPDS respectively. The PPDS is responsible for protections, input/output processing and supplying electric power to loads. The topology is shown in the single line diagram (SLD), which can be seen in Figure 2.1.

The electric system consists of two sides, the left side and the right side. AC power is generated on each side to power the AC loads. The AC power is converted to DC power to power the DC loads. The reason for using AC power is that it works at a higher voltage, which corresponds to a lower current. A low current is preferred since it reduces losses such as power losses (proportional to current squared) as well as lowers the weight since current conductors are heavy. As an added safety measure there is an independent system in the middle, which in case of major failure provides power to the most critical loads in the system.

Listed below is a description of all sources and components shown in the SLD.

2.1.1 Sources

Left/Right Generator Since safety requirements on aircraft systems are rigid, redundancy in avionics systems is necessary to guarantee sufficient power supply. Therefore, two main generators are placed on each side of the aircraft. Under normal circumstances the generators power one side each, but depending on source availability one generator could power the entire system by itself. The generators are connected to two different main AC buses, from which power is distributed to the rest of the system [5].

Auxiliary Power Unit (APU) The APU is a gas turbine which primary function is to start the main engines. In case both the left and right generator break, the APU can be used to provide backup electricity to the system [5].

External Power (EP) As the name suggest, EP means providing the system with electricity from an external source. The difference between EP and the other sources is that EP can only be used while the airplane is on ground since electricity is supplied through a ground cart [5].

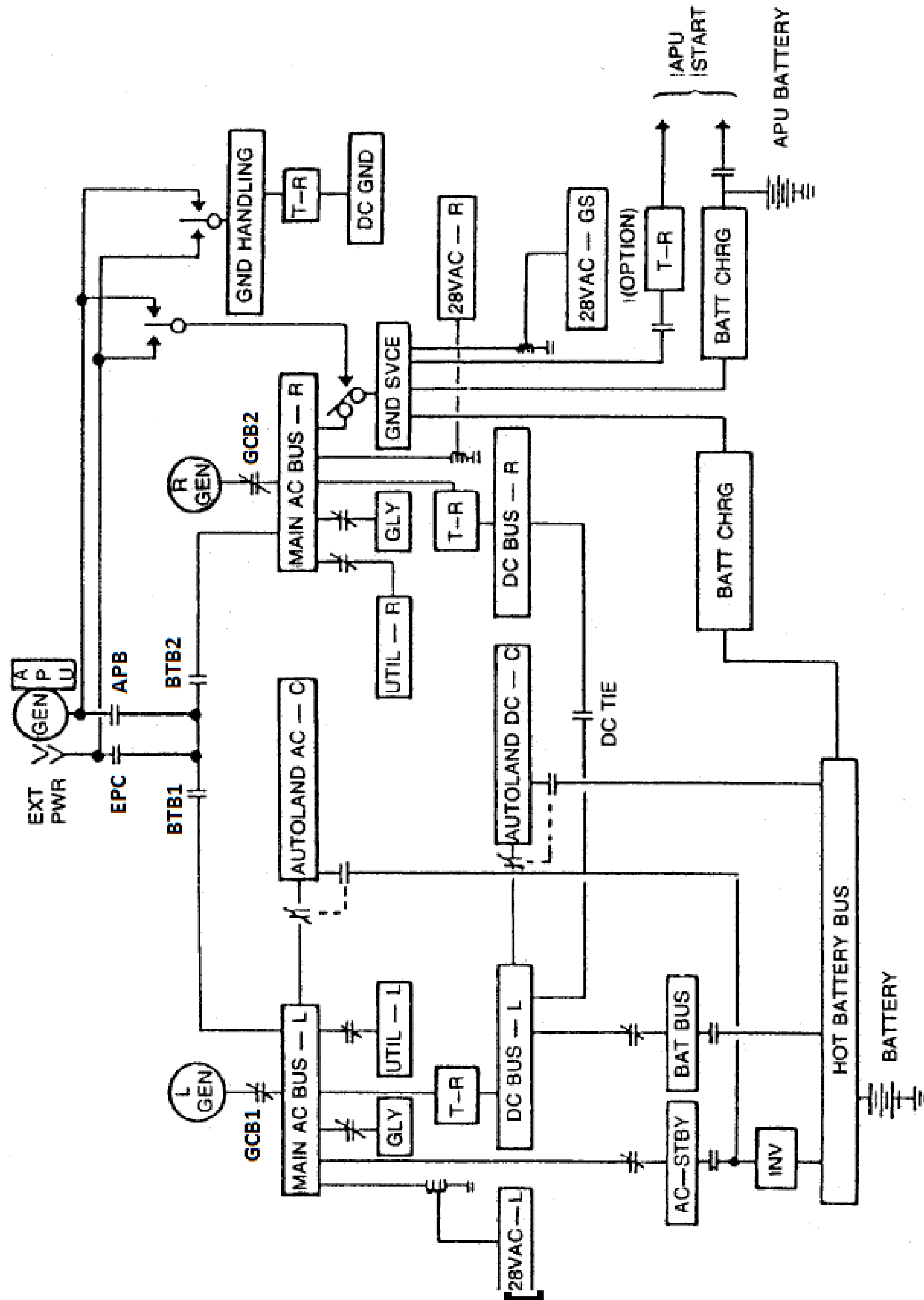

Figure 2.1: Typical Single Line Diagram

Priority	Path	Source
1	GCB1	Left Generator
2	BTB1, APB	APU
3	GCB2, BTB1, BTB2	Right Generator
4	UNPOWERED	

Figure 2.2: Priority Table

Ram Air Turbine (RAT) As one of the final protections the RAT exists to prevent power outage in the system and provide power to the hydraulics and electrical system. The RAT is used when most of all other power sources has broken down or are unavailable. It is an air-driven turbine connected to a small emergency generator. For a short period of time, the RAT is able to provide power to the most critical parts of the system and the most important flight instruments [5].

Batteries The batteries provide storage for electric power independent of the generators. They are used in system startup or emergency situations. In case of emergency, they supply short term powering (up to 30 minutes) while other sources are being prepared to take over [5].

2.1.2 Electrical Components

Transformer Rectifier Unit (TRU) A TRU is a conversion unit which transforms AC power to DC power [5].

Contactors Contactors are devices that can be either open or closed. The state of the contactor indicates whether electric power may pass through it or not. The configuration of the contactors determines the routing of electric power in the system. A contactor has several failure modes, two of them are the Failed To Close (FTC) and Failed To Open (FTO) state. A contactor is determined to be in failure mode if it does not respond to a command within a certain amount of time [5].

Buses The electrical buses distribute power to the loads in the system [5].

2.1.3 Control Algorithm

The controller is responsible for the routing of electric power to the buses by opening and closing contactors. In the system, there are redundant paths for the controller to choose from. The set of paths are listed in priority tables and are ranked by optimality. Based on priority tables and available sources and contactors, the controller determines from which source the bus will be powered.

In Figure 2.2, an example priority table for the left AC bus is shown. The first row corresponds to the most optimal power configuration for powering left AC bus. However, if GCB1 or Left generator are not available, the controller chooses the second row which is the second most optimal configuration. The control algorithm continues in this way until a valid configuration is found or all redundant paths are exhausted. The second column indicates which contactors need to be closed in order for the source to power the bus.

Chapter 3

Models of Computation for Sequential Control Systems

A model of computation is a formal, abstract description of a system and its behavior, showing how the pieces in it relate to each other. The gain of describing a system formally is that it can then be transformed into an analyzable format which makes it amenable to analysis using standard methods. The sections in this chapter describe different MoCs and modeling tool for sequential logic.

3.1 Finite State Machine

A finite state machine is a mathematical model used to describe an event-driven system. It consists of a finite number of states and a set of triggering conditions which cause the system to change state. The triggering conditions are called events. A finite state machine is often summarized in a transition table, which specifies the input and output for each transition. The output from a FSM may differ depending on what type of FSM is used, a Moore machine or a Mealy machine. These are described in short below [6].

3.1.1 Moore Machine

The output from a Moore Finite State Machine is only dependant on the state, not on the input. The actions are executed from inside a state. The advantage of the Moore machine is that its behavior is simple, but on the other hand the number of states are often larger [7]. In Figure 3.1, a Moore machine is displayed. The state machine has four states $q0, q1, q2, q3$ with actions a, b, c. The states are connected with transitions with conditions (events) x, y, z.

3.1.2 Mealy Machine

A Mealy machine calculates its output based on both state and input. All actions are tied to transitions. This tends to require less states, but also leads to a more complicated output logic than a corresponding Moore Machine [7]. A Mealy machine is depicted in Figure 3.2. It has the same states, actions and transitions as for Figure 3.1 (The Moore machine). The only difference is that the actions are performed when a transition fires, and not from within a state.

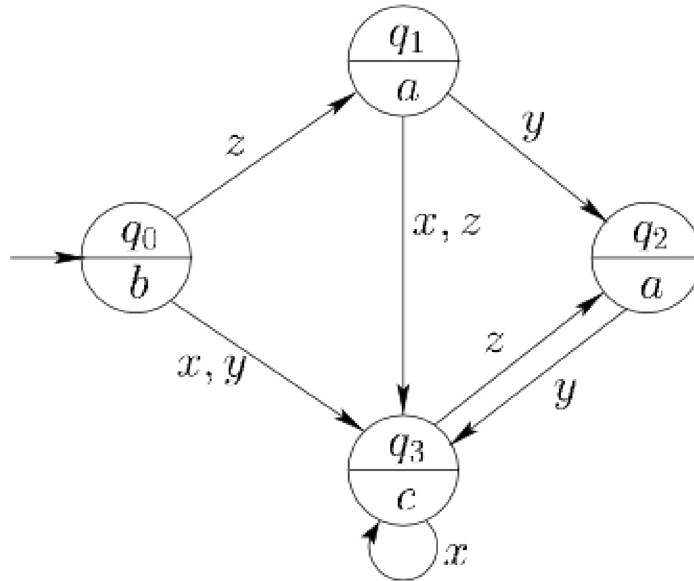

Figure 3.1: Moore machine

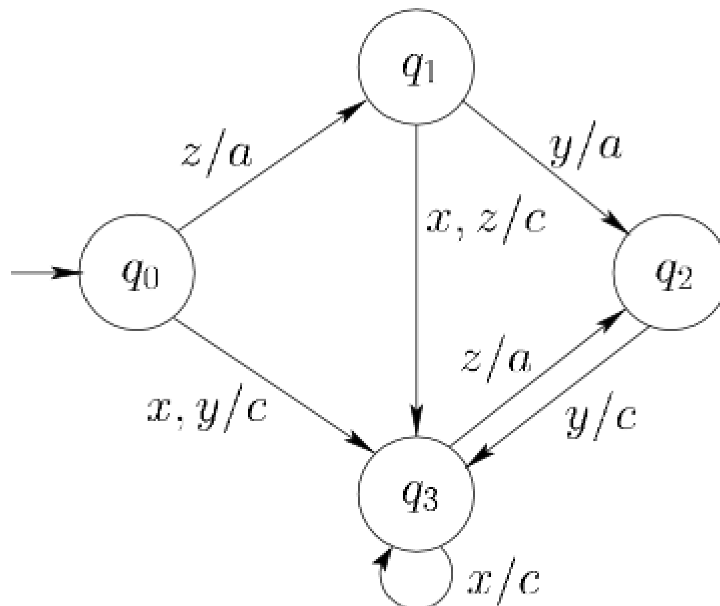

Figure 3.2: Mealy machine

3.2 Petri Net

3.2.1 Background

A Petri net is a mathematical and graphical modeling tool which can be used to model systems of concurrent, sequential, asynchronous, distributed, nondeterministic and/or stochastic character [9].

A Petri net is a bipartite directed graph, made up by three main components; places, transitions and arcs. A Bipartite graph is a graph where the nodes (places and transitions for this case) can be divided into two separate subsets such that each arc connects a node from one subset to a node in the other subset. If the arcs are directed it is called a bipartite directed graph [10].

The places, transitions and arcs, together with tokens, are used to simulate the flow of activities in the system. In the graph, places and transitions are called nodes which are connected with edges (arcs). An arc can only connect nodes of different types, i.e. a place with a transition or vice versa. Arcs are marked with a positive integer k which is called the weight of the arc. A k-weighted arc can transfer k tokens, and places can carry multiple tokens [9]. The flow is generated when a transition fires and tokens move from one place to another. A transition is enabled if all its input places carry at least as many tokens as the weight of the arc from the place to the transition. For a transition to fire, it first has to be enabled. When enabled, it may fire but does not have to. Both autonomous(untimed) and non-autonomous (timed) Petri nets exist. In the non-autonomous Petri net a condition from an external event and/or time may be assigned to the firing of a transition, when this condition becomes true the transition fires [11]. The tokens are removed from every input arc and added to every output arc when a transition fires [12].

In the informal definition the components of a Petri net are named as follows:

- P is the set of all places.
- T is the set of all transitions.
- M is the marking of the net.
- $M(P_i)$ or m_i is the number of tokens contained in one place

Each marking M is represented by a column vector with a row for each place. The element in the p:th row in the vector denotes the number of tokens in place p. With this said, the marking shows the state of the system [11]. When transitions fire the token distribution, also called the marking M, in the net is altered. A sequence of firing will cause a sequence of markings [9].

Analyzability Petri nets are of major use for analysis of concurrent systems, that is, to find out if different properties of the Petri net hold. There exist different types of analyzing methods which apply to Petri nets, where the main methods are reachability- and coverability graphs, linear algebra methods and reduction methods, which will be described in the following sections [11].

Reachability Reachability is used to examine the dynamical properties of a system. Mainly this is done by drawing a reachability graph. The graph can be used to see whether a given marking M_n can be reached from a different marking M_0 by a sequence of firings which transforms M_0 into M_n, if this holds M_n is said to be reachable from M_0. In the reachability graph (Figure 3.4) one is able to see in how many different ways this can be done [9].

Boundedness A Petri net is said to be bounded if the number of tokens in each place is less than a nonnegative number k for any marking reachable from a certain marking [9].

Coverability If the Petri net that is to be analyzed is unbounded the reachability graph becomes undefined and a coverability graph is used instead. This graph is obtained from the reachability graph by marking the places which carry an infinite number of tokens by ω and then merging the nodes that correspond to the same marking, see Figure 3.2.1 [11].

Deadlocks A deadlock occurs if a marking is reached where no transitions are enabled [11].

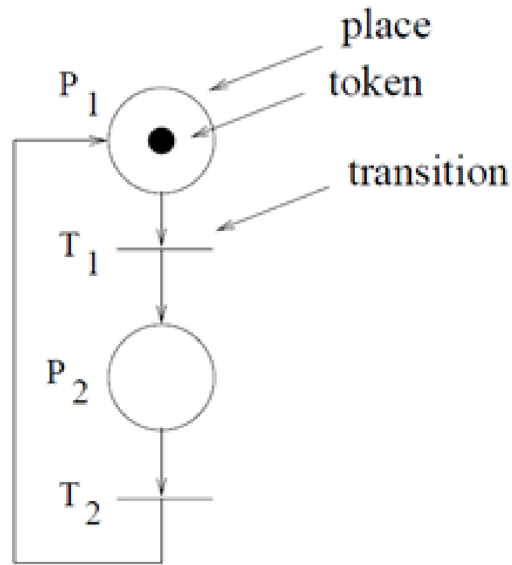

Figure 3.3: A simple Petri net

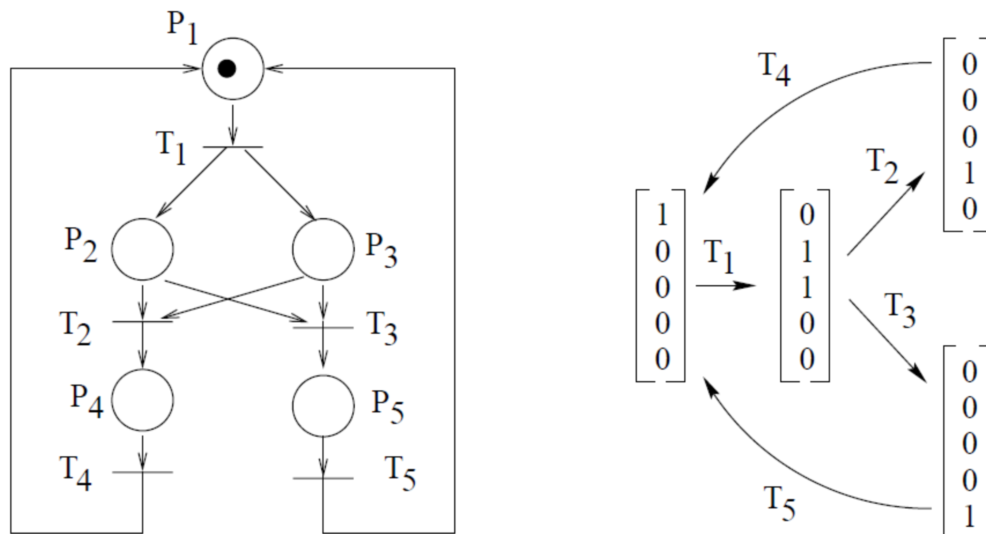

Figure 3.4: A Petri net with corresponding reachability graph

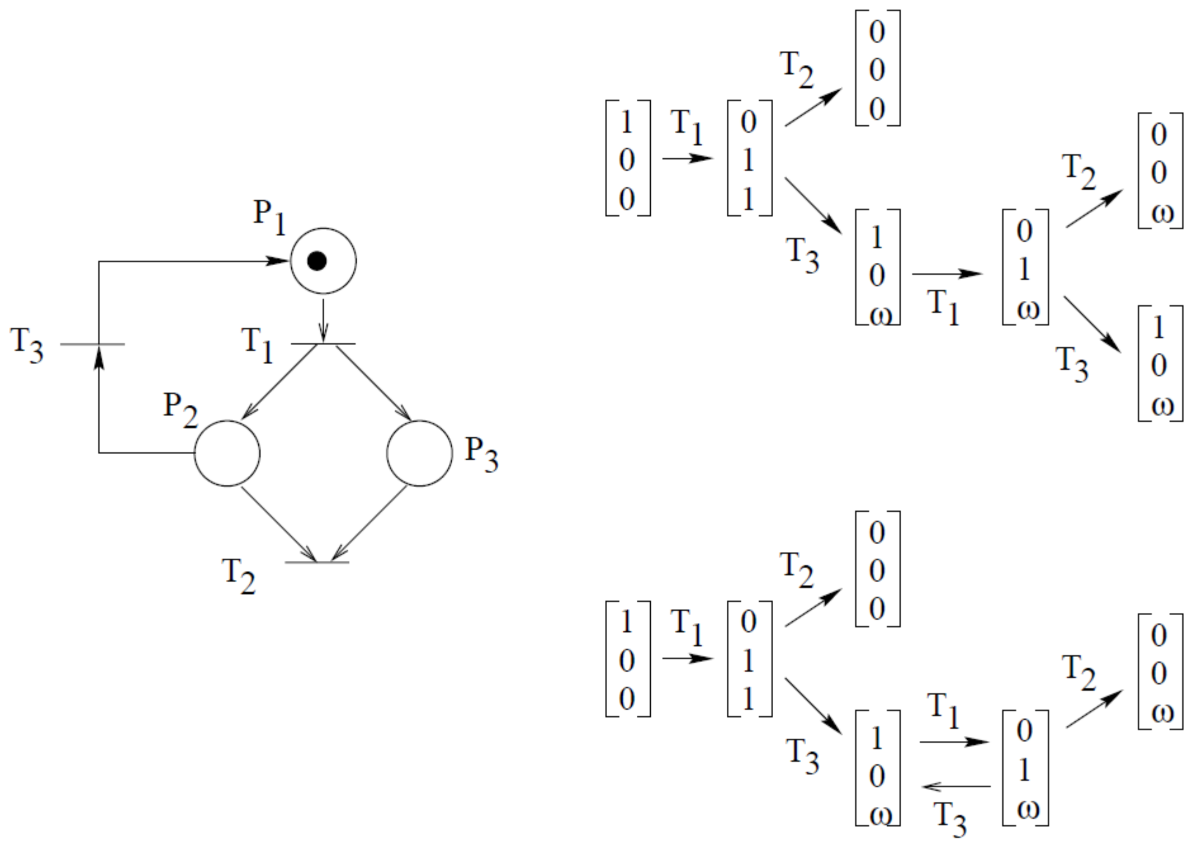

Figure 3.5: A Petri net with coverability tree (upper) and coverability graph (lower)

Linear algebra The properties of a Petri net can be determined by the use of mathematical methods in order to find out about the invariants of the net. If no deadlocks exist there will be an infinite number of firings, however this is most often not the case, all markings cannot be reached and not all sequences of firing can be done. This kind of restrictions is represented by the invariants of the net[11].

Reduction methods The reachability and coverability graphs are good analysis methods for small Petri nets, however they are not applicable to large systems. Therefore system models are often reduced to simpler ones. There exist a lot of techniques which reduce large Petri nets into smaller ones without altering the properties of the original net[11].

3.3 Different types of Petri nets

There exist both Petri nets and high-level Petri nets. The main and more informal difference between the two is that in the high-level Petri net different tokens can be distinguished and calculated with, which they cannot be in Petri nets. The following two paragraphs give an overview of two important high-level Petri nets; the colored Petri net and the object Petri net.

3.3.1 Colored Petri nets

The colored Petri net is a high-level petri net developed during the 1980s. The main idea is to assign each token a color, which serves as an identifier for that token. Any Petri net can be transformed into a colored Petri net, an action which makes them more compact in structure and easier to read and comprehend.

A simple example can be used to illustrate the benefits with colored Petri nets. Figure 3.6 describes two systems that are identical, except for the direction in which the trucks are moving, and their corresponding Petri nets. In this arrangement the system is modeled with separate nets each containing one token. The system could also be modeled with a colored Petri net. The Petri net would have two tokens whose label (or color) would mark the direction, see Fig 3.7. It is possible to transform a colored Petri net into a standard Petri net. This allows for analysis using the standard methods[11].

3.3.2 Object Petri nets

The concept of object Petri nets are inspired by high-level object-oriented programming languages, although only two classes exist; one for tokens and one for modules/subnets. The main idea is to give the token itself a Petri net structure which results in a net within a net structure. The layered approach is better suited for modeling of real system since they are not often ”flat”, but have internal structures that are of interest for the modeler[13].

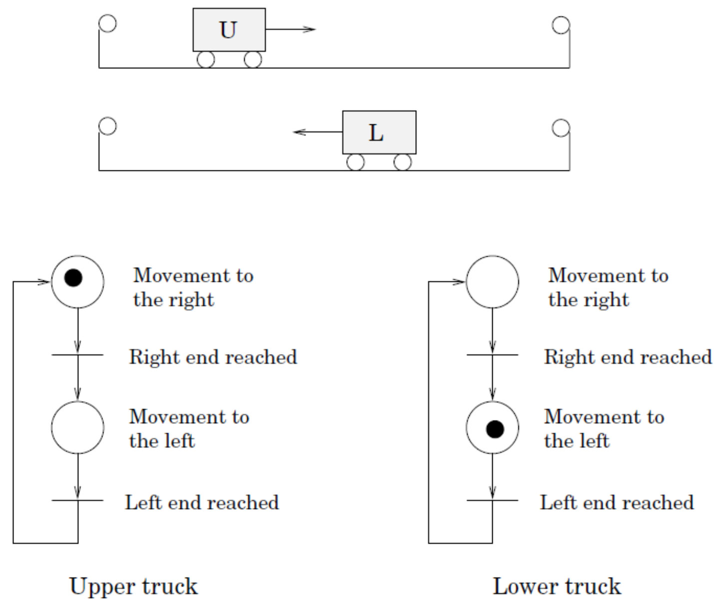

Figure 3.6: Trucks and their corresponding Petri nets

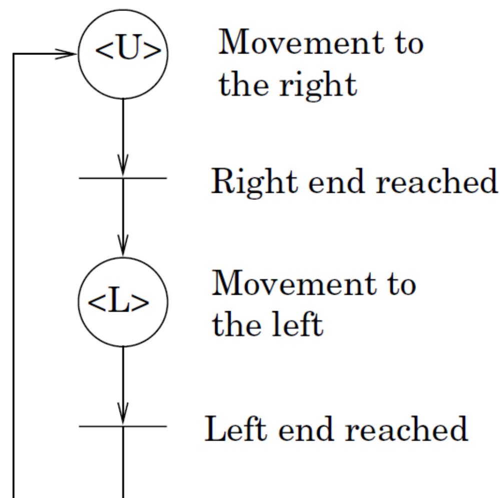

Figure 3.7: Two identical systems modeled with a colored Petri net

3.4 Stateflow

Stateflow is an extension of Simulink, which supports sequential control through the use of flow diagrams and state charts. In Stateflow it is possible to have both Moore and Mealy FSM, which in combination with its integration with Simulink and Matlab makes it a powerful tool for modeling of sequential control applications. Stateflow also has built-in C-code generation.

Stateflow is made up by a set of graphical components which are shown in Figure 3.8 and described in short in the following sections.

3.4.1 State

As the name suggest, a Stateflow state models a state and is graphically represented as a rectangle. The current state is marked with a thicker border which implies that the state is active. Stateflow supports two types of states, exclusive (OR) states and parallel (AND) states. Exclusive states are used to represent states that are mutually exclusive, which can be seen in Figure 3.8. $StateA1a$, $StateA1b$ and $StateA1c$ are exclusive, which implies that the system cannot be in more than one of them at a time. Exclusive states are marked with a solid border line. The parallel state, on the other hand, is used to model system behavior where the system can be in multiple states at the same time. Parallel states are marked with a dashed line and can be seen in the picture as $StateA2a$ and $StateA2b$. As a structuring mechanism, states can have internal states. A state which contains other states is called a superstate, and the inner states are called substates [8]. Superstates and substates can be seen in Figure 3.8, where for example $StateA1$ is superstate to $StateA1a$, $StateA1b$ and $StateA1c$ as well as a substate to $StateA$.

For each state, it is possible to specify actions. There are different types of actions and the type determines when the action will be executed. The types are; entry actions, during actions, exit actions, on event_name actions and bind actions. Entry and exit actions are executed once when a state is entered or exited respectively. During actions are executed periodically while the state is active. On event_name actions are bound to an event in the system and are performed when the active state receives that event, that is, events occurring in the system can trigger events in states. Bind actions ties data and events to the state where it is declared, meaning only that state and its substates are allowed to change the tied data and broadcast the events. Other states can only read the data and listen to the events[8].

3.4.2 Transition

Transitions mark the changing of state in the system. They are directed arcs that link one state to another. It is possible to prioritize transitions, meaning that if a state has more than one transition connected and two or more of them become true simultaneously, the one with the highest priority will fire. To mark where execution starts, default transitions exist. They can be connected to states and connective junctions. If connected to a state, they have no source state, only a destination state. The default transition is shown in Figure 3.8.

Transitions are defined by a label with the following syntax $event[condition]conditionaction/transitionaction$. Thus, a transition label is made up of four parts, each of which is optional.

Event The event part refers to a specified event in the system. If this event occurs and the condition, if any, is true the transition is taken. If no event is specified, the transition will be taken at any event.

Condition Conditions are simply boolean expressions which must be true in order for a transition to be taken. If both event and condition are specified, the condition must be true and the event must occur simultaneously for the transition to be valid.

Condition Action Condition actions are performed immediately if the condition is true. In absence of a condition, it will be assumed to be true (Implied condition). A condition action is performed before knowing whether the destination state is valid or not. A destination is valid if the preceding state is active and the transition connecting them is true.

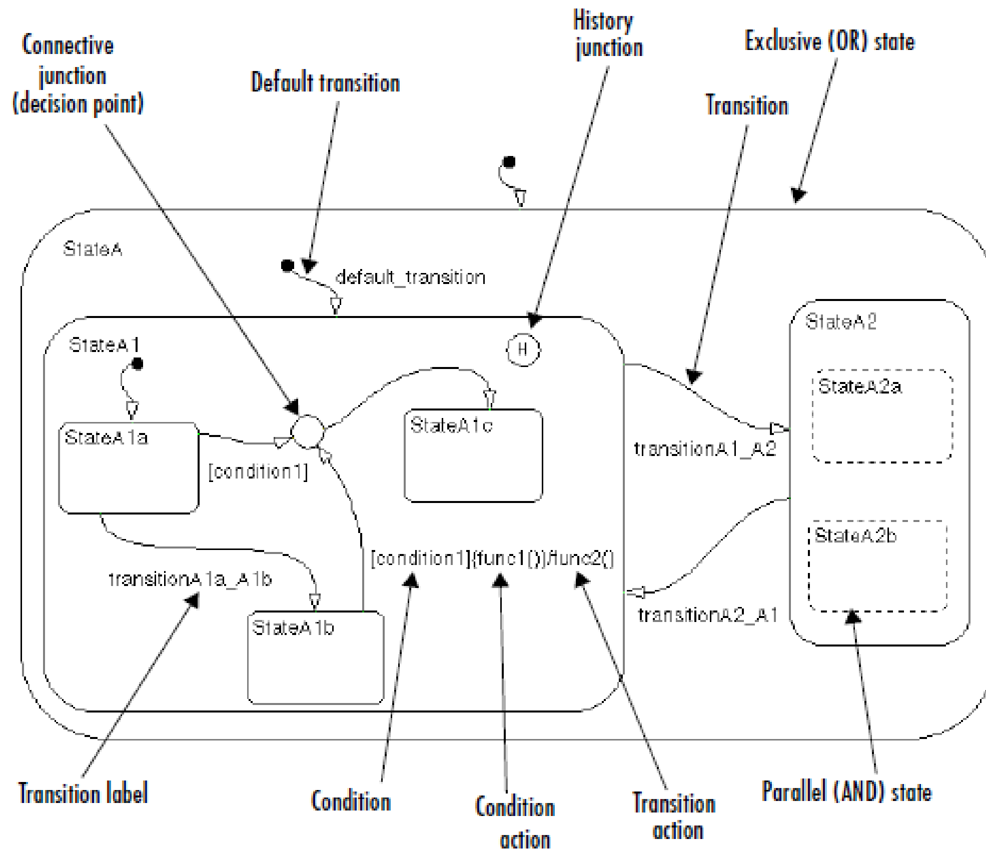

Figure 3.8: Stateflow

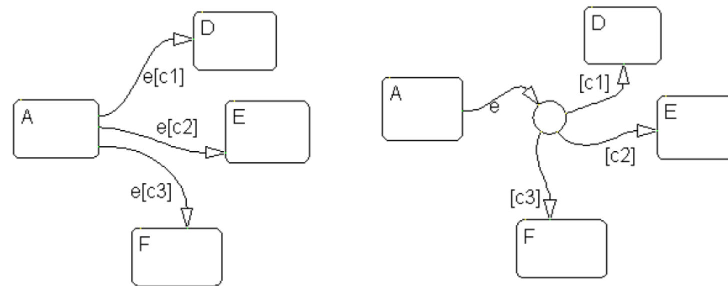

Figure 3.9: State machine without connective junction (Left) and state machine with connective junction (Right)

Transition Action The difference between a condition action and a transition action is that a transition action is not executed before the destination state is proven to be valid [8].

3.4.3 Connective Junction

A connective junction is a way of representing multiple transition paths originating from a single transition, i.e. a source state can have several destination states even though it only has one outgoing transition. In Figure 3.9 the concept of the connective junction can be seen. The two state machines are equivalent, but the state machine on the right hand side uses a connective junction instead of having three outgoing transitions. Upon the occurrence of event e, the state machine moves to the connective junction and evaluates the conditions $c1$, $c2$ and $c3$. If none of them are true, state A continues to be active[8].

3.4.4 History Junction

A history junction is used in a superstate to remember which substate was last active. When exiting the superstate, the history port registers the currently active substate and once it becomes active again, execution is resumed from that substate[8].

3.5 Grafcet

Originally developed in France 1977, Grafcet is a formal method for graphically describing logical controllers as state machines. Grafcet is based on Petri Nets and supports both Moore and Mealy state machines[14]. Figure 3.10 shows a simple Grafcet application. The components are described in short below.

3.5.1 Step

The states in Grafcet are called steps and are drawn as squares. A step can be either active or inactive depending on whether there is a token present or not. Initial steps are a special kind of step which are activated when execution starts, and are graphically represented as a double square.

It is possible to tie actions to a step, which are executed when the step is active. The actions can be one of two types, level actions or impulse actions. Level actions are continuous actions which are performed while the step they are tied to is active. A level action can be either conditional or unconditional. Impulse actions are executed immediately when the step becomes active, and are only performed once[11].

3.5.2 Transition

Transitions are represented as horizontal bars and connect steps. When the step(s) preceding the transition is/are active, the transition is said to be enabled. To a transition, a condition, an event or both can be tied. When an enabled transition becomes enabled, it is also fireable. It will fire instantly and tokens are moved

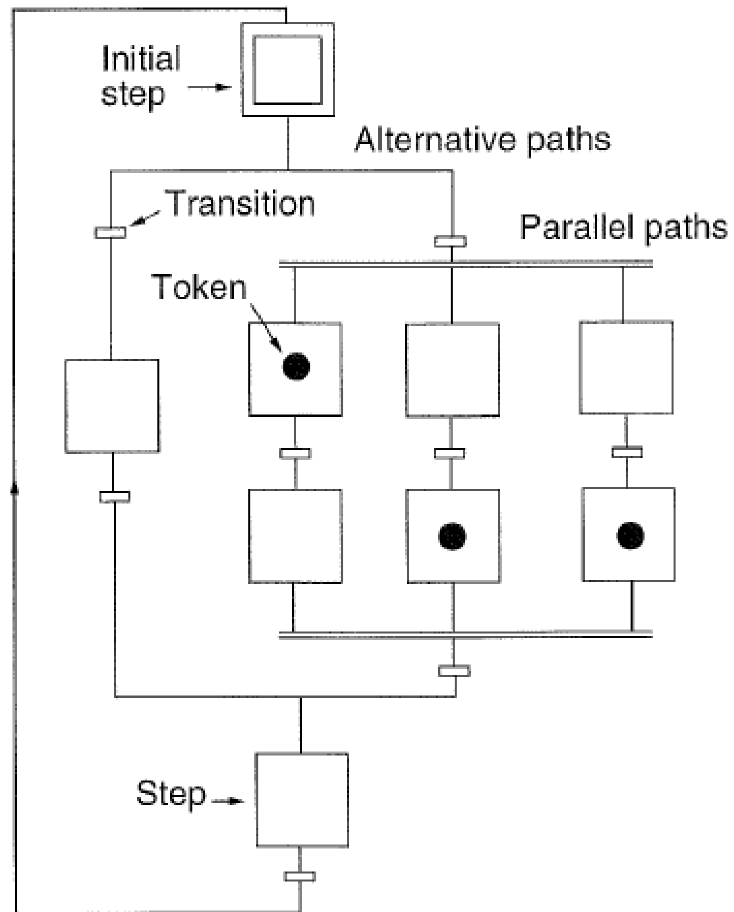

Figure 3.10: A Grafcet chart

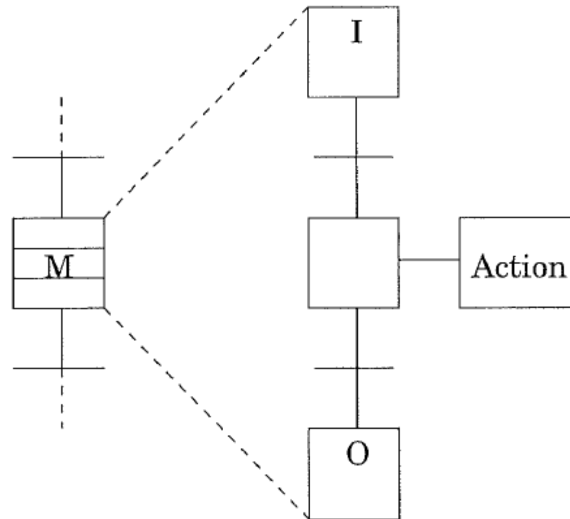

Figure 3.11: Grafcet Macro Step

to the succeeding step(s). In Grafcet, there is no support for prioritizing transitions. If a step has more than one outgoing transition and several of them are fireable at the same time, all fireable transitions will fire simultaneously. If, when a step is entered, its outgoing transition fires immediately the situation is said to be unstable. In this case, only the impulse actions are performed[11].

3.5.3 Branching

In Grafcet, both alternative and parallel branches can be used. If an ingoing transition which connects more than one step through the use of a parallel path fires, all steps succeeding it will become active. Furthermore, for outgoing transitions connected to more than one preceding step, all steps preceding the transition must be active for the transition to enable[11]. The concept of branching is visualized in Figure 3.10.

3.5.4 Macro Step

Macro steps is a way of structuring large applications in Grafcet. A macro step contains an internal sequence of steps and has one input and one output step. The transition following the macro step is not enabled until the output step is reached[11]. The macro step architecture can be seen in Figure 3.11.

3.6 Sequential Function Charts

Sequential Function Charts (SFC) is a programming language used to graphically model sequential systems and is standardized in IEC 61131-3. The language was derived from, and is essentially the same as, Grafcet. The Grafcet/SFC standard is well accepted in the industry due to its graphical interface. SFC consists of steps connected to transitions via directed arcs. It is possible to tie three types of actions to steps, set actions, reset actions and continuous actions. A continuous action is executed periodically while the step is active. A step becomes active if the transition(s) connected to it is/are fireable, i.e. its/their condition(s) is/are true, and the preceding step(s) is/are active [11]. A simple SFC example is shown in Figure 3.12.

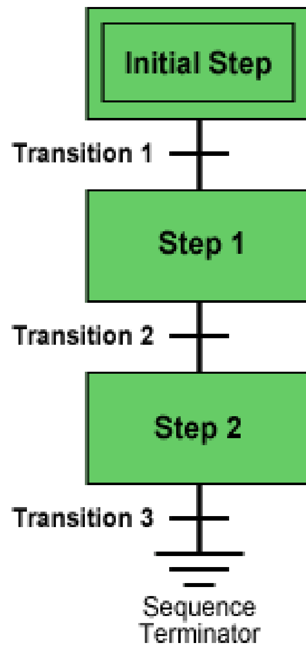

Figure 3.12: Sequential Function Chart example

3.7 JGrafchart

3.7.1 Background

Grafchart is a mathematical modeling tool based on Grafcet/SFC and Petri nets. It also features concepts from object-oriented programming languages. The aim of Grafchart is to provide a high-level language for control applications.

Originally there existed two versions of Grafchart, one low level version derived from Grafcet, as well as a high level version which resembles colored Petri nets. When the implementation platform was switched from G2 to Java, only support for the low level version was implemented. This version was called JGrafchart and is the version that is currently in use. This means that JGrafchart runs on every platform supporting Java. One feature of JGrafchart is the ability to create extensive graphical user interfaces, including for example animated icons, plotters and message browsers [15]. JGrafchart features several graphical elements which together form the system one wish to model. These components are briefly described in the following paragraphs.

3.7.2 Step

A step represents one of the states in the system. It is possible to tie actions to a step, which will be executed when the step is active, i.e. when it contains a token. The actions can be one out of four types; initially, finally, always or abortive. Final and initial actions are performed only once when a step is entered or exited respectively. Always actions are executed periodically while the step is active. Abortive actions are executed once when an exception transition fires.

In order to find out for how long a step has been active, two methods exist. The call syntax is either $< step - name > .s$, which returns the number of seconds the step has been active, or $< step - name > .t$, which returns the number of periods. Periods in JGrafchart are called Scan Cycles and can be set manually for each chart.

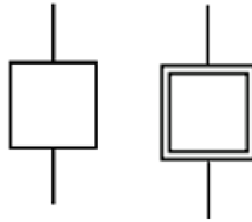

Figure 3.13: Step and initial step

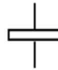

Figure 3.14: Transition

To mark where execution starts, initial steps are used. These are marked graphically by a double square, see Figure 3.13[15].

3.7.3 Transition

A transition models the changing of system state. They are controlled by Boolean expressions and are enabled once the expression becomes true. If there are tokens in the preceding step(s) connected to the transition and the transition is enabled, it will fire and tokens will move to the succeeding step(s)[15]. If a step has more than one outgoing transition, it is possible to prioritize them. Consequently, if several of the outgoing transitions are fireable at the same time, only the transition with the highest priority will fire. Priorities range from one, which corresponds to the highest priority, and up. If no priority is set, the transition automatically receives the lowest priority.

The JGrafchart transition can be seen in Figure 3.14.

3.7.4 Macro Step

A macro step is used to mark an internal sequence of steps and transitions. The macro step architecture provides the option of having a layered structure, which enhances readability of the model. It is possible to have multiple entries and exits to and from a macro step. In addition to steps and transitions, the body of the macro step also contains enter and exit steps. The number of enter and exit steps are equal to the number of transitions connected to the macro step, meaning that each transition corresponds to a certain enter or exit step. The transition(s) following the macro step cannot fire until its exit step is active.

The macro step has two additional ports, the exception transition port and the history port. The exception port has priority over the transitions in the body and aborts the macro step execution if it becomes true. The history port makes it possible to continue to execute from the state the system was in before abortion[15].

The macro step architecture is shown in Figure 3.15.

3.7.5 Exception Transition

The exception transition is a high priority transition which can only be connected to a macro step or procedure step. It is only active during the execution of the step it is connected to. If the exception condition becomes true, the transition fires and forces abortion of the current step. It is possible to add abortive actions that should be executed in case of exception[17]. The exception transition is depicted in Figure 3.16.

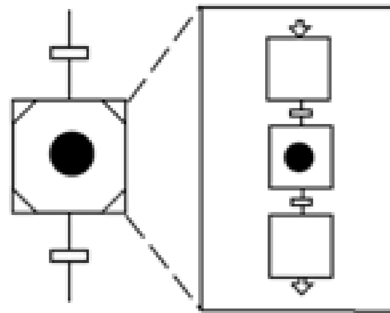

Figure 3.15: Macro Step

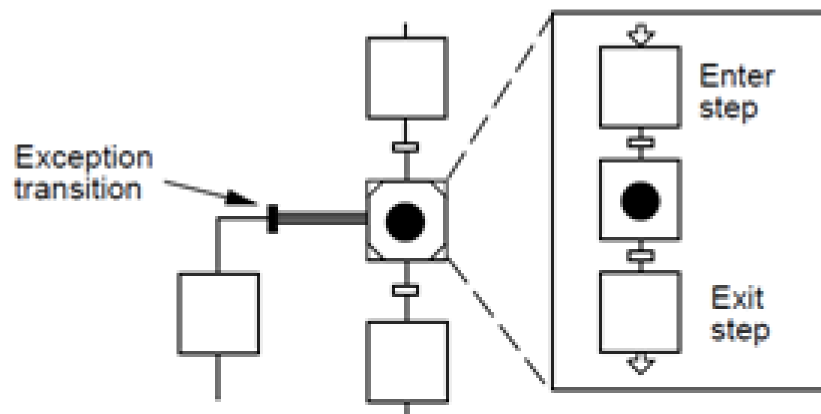

Figure 3.16: Exception transition

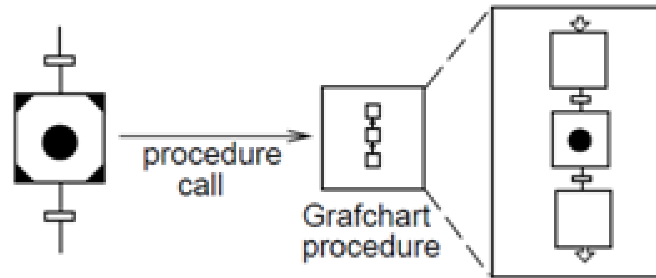

Figure 3.17: Procedure step

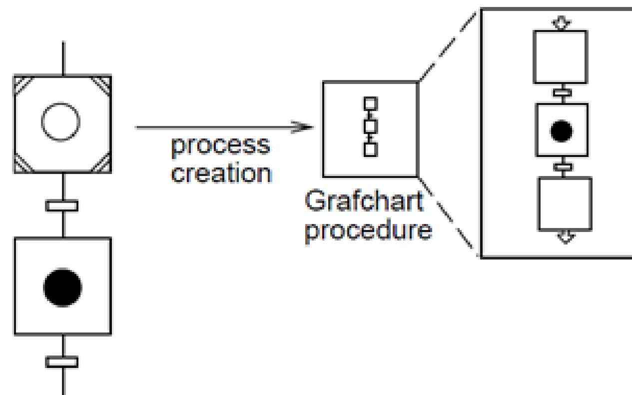

Figure 3.18: Process step

3.7.6 Procedure Step

If the same sequence of steps and transitions are repeated in the model, they can be extracted to a procedure. Procedures in JGrafchart are similar to methods in object-oriented programming. When a procedure step is entered, it calls a certain procedure. Similar to the macro step, procedures start with an enter step and end with an exit step. However, the procedure step does not allow multiple entries and exits. It is also possible to connect an exception transition to a procedure, but there is no history port. The transition following a procedure will not enable until the exit step is reached[15].

It is possible to pass values to and from a procedure. Parameters are assigned values either by call by value or call by reference, for which the syntax is $V < internalvariable >=< expression >$ and $R < internalvariable >=< variable >$ respectively[15]. The procedure step and its internal structure (body) can be seen in Figure 3.17.

3.7.7 Process Step

The process step is basically a procedure step except that the procedure is started in a separate execution thread, meaning that the transition following the process step may fire even if the process step has not reached its exit step[15]. The process step is shown in Figure 3.18.

3.7.8 Step Fusion Set (SFS)

The step fusion set is a way of representing different views of the same step. This means that if several steps are linked by a step fusion set, whenever one of them becomes active, the other ones activate as well. Step fusion sets can be either abortive or non-abortive. In an abortive step fusion set, for the transition connected

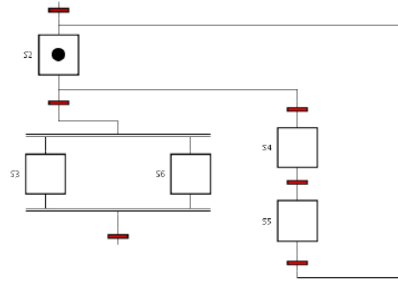

Figure 3.19: State Machine Before Applying Step Fusion Set

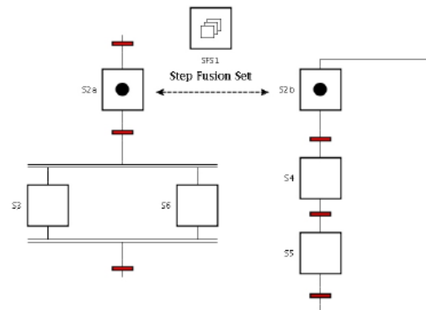

Figure 3.20: State Machine After Applying Step Fusion Set

to the output of the step fusion set to fire, it is only required that one of the views is in its last step. When an abortive step fusion set is exited, abortive actions, if any, are executed in the views that did not reach their last step. Non-abortive fusion sets however, require all of the views to have reached their last step before the transition connected to the output may fire [16]. The concept of the step fusion set is shown in Figures 3.19 and 3.20. In Figure 3.19, the state $S2$ is connected to states $S3$, $S6$ via one transition and to state $S4$ via a different transition. However, in Figure 3.20 state $S2$ is divided into $S2a$ and $S2b$, where state $S2a$ is connected to $S3$ and $S6$ and state $S2b$ is connected to $S4$. In that way, $S2a$ and $S2b$ represents the two views of the same state $S2$, and the state machine in Figure 3.19 has been separated into two different state machines in Figure 3.20. However, the two figures are equivalent in terms of functionality.

3.7.9 Object-Oriented Features

As mentioned, JGrafchart resembles object-oriented programming languages in the sense that objects and methods can be modeled. To represent objects, the workspace component can be used. A workspace contains a subworkspace and can be used in different ways, e.g. as a way to structure large JGrafchart applications or to model objects. If a workspace is used to model objects, its subworkspace contains only attributes represented as JGrafchart variables and methods represented as JGrafchart procedures.

However, the object-orientation in JGrafchart is limited since there is no support for classes, inheritance etc[15].

3.8 Choosing a Model of Computation

In the previous section, several MoCs and tools suitable for sequential control systems have been presented. When choosing a MoC, its applicability to the domain as well as its formality have to be taken into account. For example, the Finite State Machine is very good for sequential logic, but not very expressive and the number of states grow large even for trivial systems. The structuring capabilities in JGrafchart, where each state recursively can contain a whole state machine, together with the object oriented concepts and the fact

that it is transformable to a Petri net and can therefore be analyzed using standard methods makes it an interesting tool to investigate.

Chapter 4

Application of JGrafchart

4.1 Motivation

One of the main reasons for giving a system a model-based design is to increase traceability and analyzability. JGrafchart combines ideas from Finite State Machines and object-oriented programming languages to provide a more powerful structuring mechanism for designing complex systems. The motivation for using JGrafchart instead of Stateflow can be compared to the motivation for using a high-level programming language such as Java instead of a low-level language. It enables the designers to easier build larger and more complex systems.

4.2 An Existing Control System

In this chapter the implementation process is described. Due to proprietary reasons, specific details are removed and replaced with generic substitutes. The investigated control system models the Primary Power Distribution System (PPDS) in a medium sized jet airliner. The PPDS is divided into several subsystems, among which are the Power Transfer system which handles the routing of power from the sources to the loads, and the Protection system which exists to isolate faulty components.

The main control objectives are to make sure that the system operates safely both during normal operation and faults, as well as maximize the bus availability for all system states. This means that the most fundamental safety requirements always shall be met and that the system should utilize the available power in the best possible way.

The following paragraphs describe the Stateflow implementation of the sequential parts of the controller, followed by a brief description of the interfaces necessary for communication between Simulink and JGrafchart as well as the implementation of the sequential control in JGrafchart.

4.2.1 Stateflow blocks

In the Power Transfer subsystem the configuration of the contactors takes place. Part of this is sequential control, represented in the system as Stateflow charts. The rest of the subsystem is mainly logical expressions, which for obvious reasons are left in Simulink since it is better suited to handle that kind of computations.

The first part of the sequential control determines which part of the electrical system to configure based on requests. The requests are generated based on the status of sources and contactors. The requests can be either to turn a source on, turn a source off or to reconfigure the system. In the first Stateflow block, all of the requests have priorities as they can be triggered simultaneously and only one request can be processed at a time.

The second Stateflow block handles the sequence of steps taken when a request is triggered. The sequence is executed sequentially to reach a new target AC and DC configuration, and may differ slightly depending on which request is triggered and what state the system was in before the request. If the target configuration

cannot be achieved due to unavailable contactors, the controller will re-execute the second Stateflow chart to find an alternative path.

4.2.2 Interfaces

Communicator between Simulink and JGrafchart

Since JGrafchart is not integrated with Simulink an interface had to be developed in order for them to communicate. This work was done at Lund University and resulted in a TCP/IP server allowing Simulink and JGrafchart to exchange data over sockets. The server is placed in the Simulink model as a Simulink block. The server block receives and transmits messages using socket blocks in JGrafchart. The socket blocks in JGrafchart can be one of the following types; Real, Integer, Boolean or String. The format of the message being sent starts with an identifier followed by the value. The identifier maps the Simulink signals to the JGrafchart I/O Sockets, i.e. the signal name and the socket name must correspond.

The server has four arguments; InputPortMapping, OutputPortMapping, sampling time period and port. The InputPortMapping is a Simulink vector containing the identifiers for the input sockets in JGrafchart. If the size of the vector is larger than one, i.e. more than one signal is sent to JGrafchart, a multiplexer in Simulink is needed since the server only has one input port. The first element in the vector corresponds to the first port in the multiplexer, the second element to the second port and so on. This means that the order of the identifiers are of significance. The same structure applies to the OutputPortMapping vector, except that a demultiplexer is used instead of a multiplexer at the output port on the server. The sample time period is given in seconds and denotes how often the input signals to JGrafchart are read. The port is an integer indicating over which port communication takes place.

In addition to the server block, a block which controls the simulation time in Simulink was also implemented at Lund University. This was necessary since JGrafchart runs in real-time while Simulink runs in simulated time. The realtimer block synchronizes Simulink time with real time by slowing down the simulated time periodically.

4.3 JGrafchart Control System Architecture

The first part of the implementation process focused only on the sequential control in the Stateflow blocks. The aim was to investigate whether an equivalent implementation in JGrafchart could be achieved, i.e. given the same input the Simulink state machines and the JGrafchart state machines would produce the same output in terms of contactor configuration and, if any, lockouts.

The second part further extends the implementation by moving a larger part of the system from Simulink to JGrafchart. A fault handling layer was implemented in JGrafchart to substitute the protection layer in the Simulink model. To detect faults, components in the system were modeled with individual state machines. This enables the controller to keep track of the state of each component in parallel with the control algorithm. If a fault occurs in any of the components, that state machine moves into a fault state which triggers the fault handling.

4.3.1 Sequential Control

Communication

As the name suggest, the communication layer handles the communication between Simulink and JGrafchart. This layer has no sequential control functionality but simply contains all the I/O sockets.

In the Simulink model, a fixed step size is used. The fixed step size puts constraints on the sampling time in the sense that it must be a multiple of the fixed step size. In order for the model to execute properly, the sampling time in the Simulink model must be the same as that in the server and realtimer block. The sampling time indicates how often Simulink reads the server output port, meaning that if an output socket value in JGrafchart has changed more than once during one sample period, only the last change will be registered by Simulink. To avoid this problem, some delay is introduced at concerned transitions in JGrafchart.

Sequential Control

The JGrafchart model of the first Stateflow chart consists of two top level states, one state where the system awaits requests and one where requests are processed. The second state has internal states, since there are several different requests that may arrive. The layered structure introduced due to internal states is represented as a Multiple Input Multiple Output (MIMO) Macro step. The JGrafchart representation of the first Stateflow chart is shown in Figure 4.1 and 4.2.

Each transition in the top level corresponds to one request and listens to an input socket at the communication layer, which represents its request. Since the Simulink model only processes one request at a time, the transitions between the two top level states are prioritized. When a transition at the top level fires a trigger is sent to the second sequential control block, which handles the sequence of steps taken to achieve the configuration needed to satisfy the request.

The highest level of abstraction when looking at the system as a whole are modeled by two states, one steady state and one transient state. When no requests are processed, the system is in steady state and otherwise it is in transient, meaning that the electrical system is being configured. In the JGrafchart implementation the steady state and transient state are placed in a macro step. The macro step in turn is linked to a detection step at a fault handling layer by an abortive step fusion set. Thus, error detection is provided for all system states. Figure 4.3 and 4.4 show steady state and transient as well as the internal states in transient. If an error occurs, the execution of the second part of the sequential control will be aborted. When the error has been resolved at the fault handling layer it will return to its detection step and the sequential control will be re-activated and continue to execute from the state that was last active, according to the step fusion set logic. At the beginning of execution, the state machine is in steady state and remains there until triggered by the chart handling the requests. When the AC and DC target configurations are reached, the state machine returns to its steady state and waits for the next trigger.

In order for the communication to work properly, delays are introduced at the transitions to make sure the input socket values are updated.

4.3.2 Sequential Control Integrated With Fault Handling

To determine whether there is a fault in the system, the state of each of its components is evaluated. State machines were implemented for each component to keep track of their current state. If an error occurs, the state machine for the faulty component transits into an error state. The JGrafchart step fusion set feature makes it possible to have a separate fault handling layer which monitors the component state machines in parallel with the execution of the control algorithm. Whenever a fault in a component is detected, a high priority transition at the fault handling layer will fire and the exception actions will be performed. Different components experience different faults which are summarized in Table 4.1. Faults are of different severity and may inhibit requests as well as other fault handling. The design concept is shown in Figure 4.5.

As the aim of this thesis is to only investigate whether the MOC behind JGrafchart can be applied and provide value to aircraft electric systems, only the components and their respective faults necessary to show a proof of concept were implemented.

Implementation of Components

In JGrafchart, a component is modeled as an object through the use of a workspace. All workspaces contain a state machine which models the behavior of the component. The transitions in the state machine listen to input sockets at the Communication layer. The input are either data from the Plant indicating the current state of the component or commands from the controller. Furthermore, components of the same type ("instances of the same class") are collected in a workspace together with a state machine showing if there is a fault in any of them. The state machine consists of two states, the initial state AllOK and the Error state, which the state machine transits into whenever one of the components are in a faulty state. This structure can be seen in Figure 4.14.

The following paragraphs describe the components implemented in JGrafchart.

Contactor As an initial state, a contactor can be either open or closed denoted as normally open or normally closed respectively. In Figure 4.13, the state machine for a normally open contactor is shown.

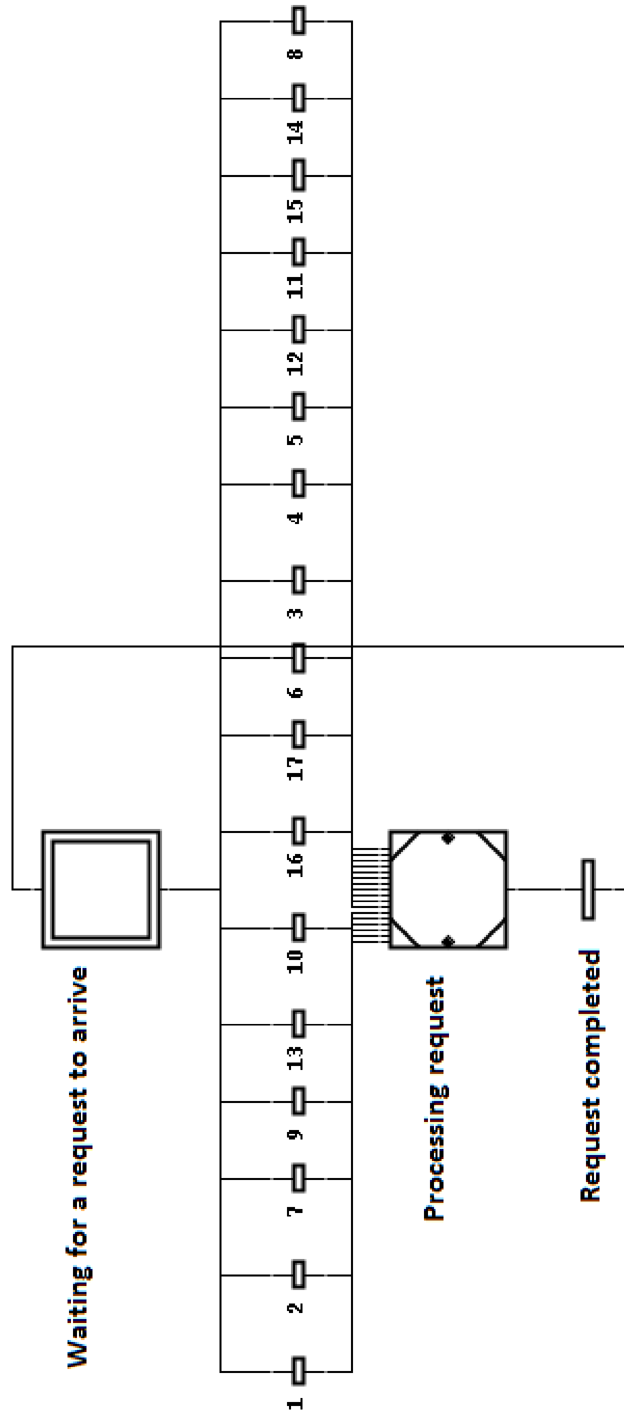

Figure 4.1: Request Handler, top layer

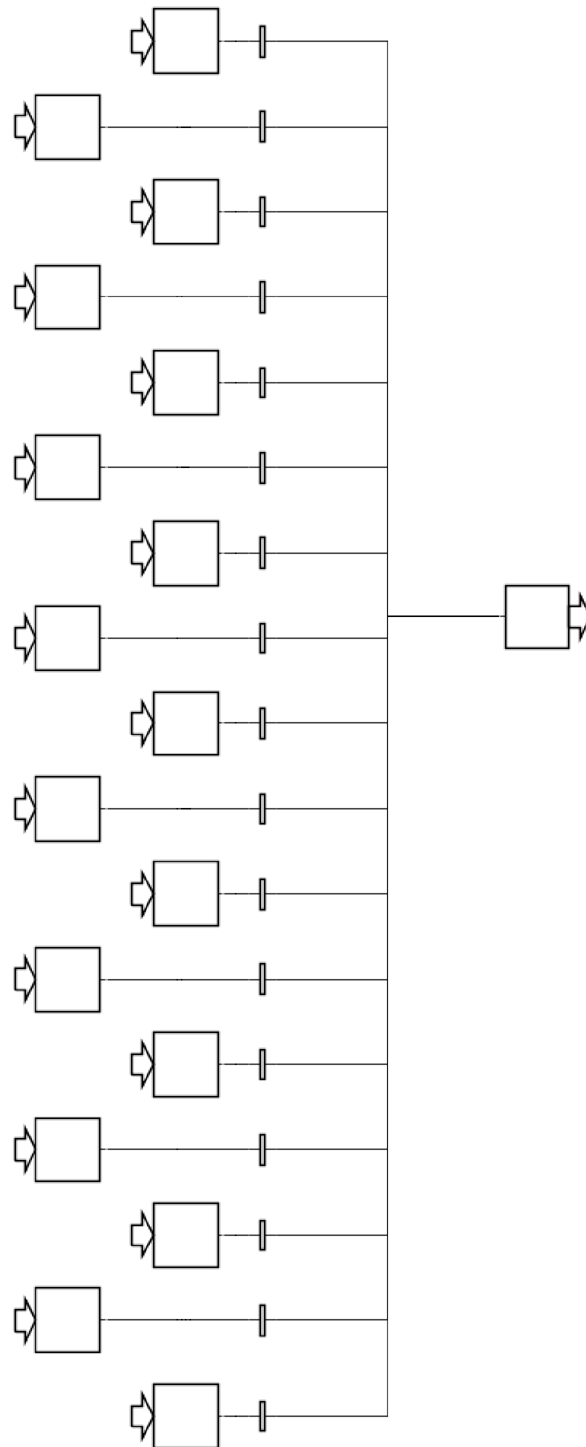

Figure 4.2: Request Handler, internal states

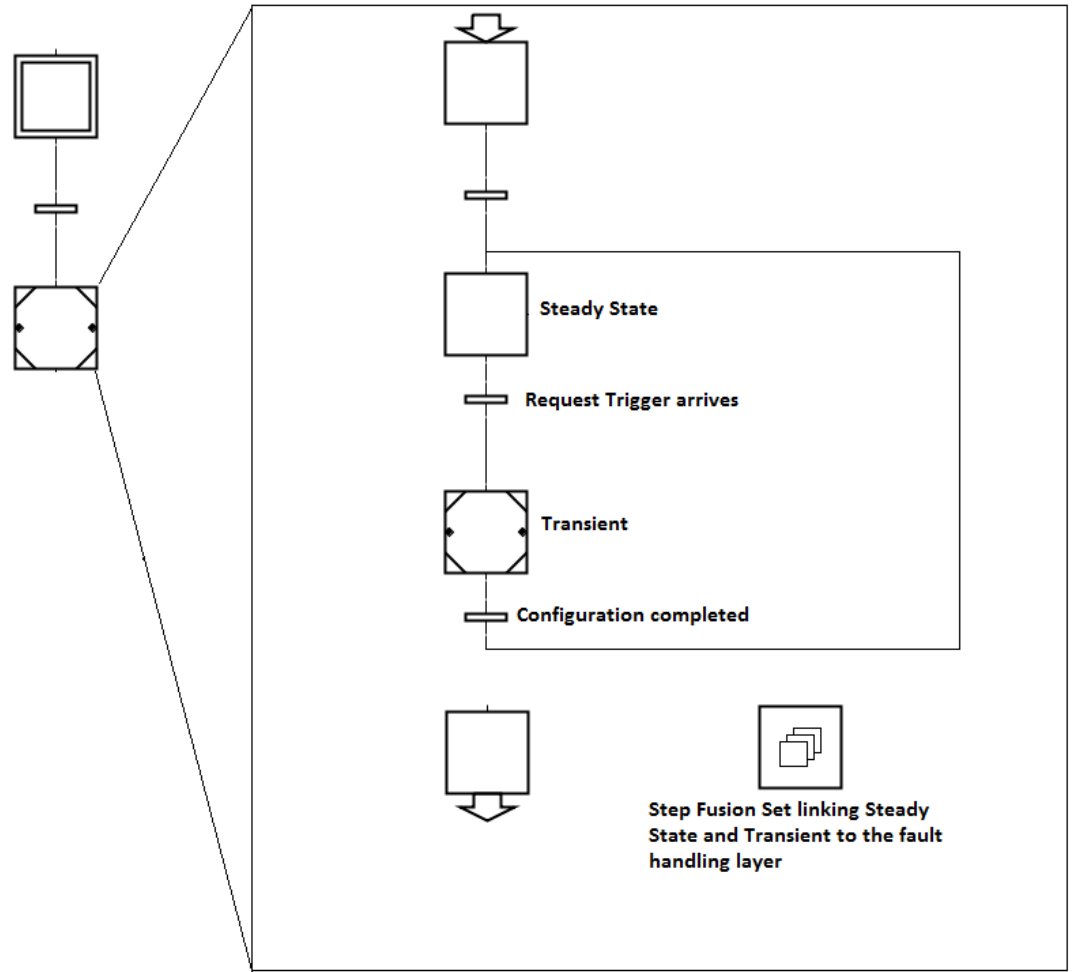

Figure 4.3: Configuration of electric system, top layer

Table 4.1: Components and Examples of Respective Faults

Component	**Fault**
Left/Right Generator	Over Current
	Over Frequency
	Over Voltage
	Under Frequency
	Under Voltage
APU	Over Current
	Over Frequency
	Over Voltage
	Under Frequency
	Under Voltage
EP	Over Current
	Over Frequency
	Over Voltage
	Under Frequency
	Under Voltage
Contactor	Failed To Open
	Failed To Close

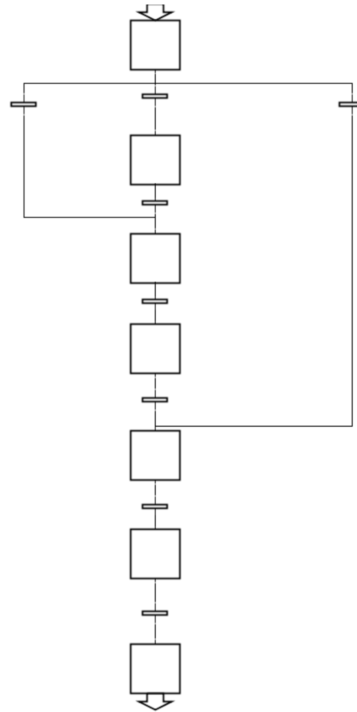

Figure 4.4: Configuration of electric system, internal sequence

The contactors implemented in the JGrafchart model are all normally open. Contactors are controlled by commands from the controller. If a contactor would fail to respond to a command within a given time limit, it will transit into a fault state. As an entry action in the fault states, the *Error* variable is set true which causes the state machine for all contactors to move to its error state. This in turn triggers a transition at the fault handling layer which handles the contactor error.

Each contactor object has two variables, *Name* and *AdjacentContactor*. These are retrieved at the fault handling layer using the call by reference functionality in the procedure step and are needed to determine where the fault is.

Left/Right Generator In the JGrafchart implementation, the generator state machine top layer consists of an Offline initial step, an Online macro step and one step for each fault described in Table 4.1. The transitions between the Online and Offline state listens to the boolean input socket < *Generatorname* > *Online* which is retrieving its value from the plant. Socket value 1 indicates that the generator is powering the system. The top level structure can be seen in Figure 4.6.

The Online macro step body is shown in Figure 4.7 and comprises of one Entry step and several macro and exit steps. When a generator is online, all faults described in Table 4.1 may occur. Therefore, the state machine for the generator must monitor all generator sensor values to detect abnormal circumstances when in the Online state. When detecting an abnormal sensor value it is handled by waiting for a specified amount of time and then re-checking the sensor values to determine whether there is a fault or not. Although the general idea of determining if there is a fault or not apply to all generator faults, they still differ significantly. Thus, the sequence of steps taken from the time an abnormal sensor value is detected until either a fault is confirmed or the sensor values have returned to normal, must be explicitly implemented for each fault. This results in a large number of steps and transitions, which is why each fault detection state and fault detection handling was implemented as a macro step to provide structure and enhance readability.

When entering the Online macro step, all fault detection macro steps activate, since theirthe Online entry step is connected to all fault detection macro steps via transitions that are true by default. This design enables a generator to simultaneously handle detection and determination of different faults. If a fault is

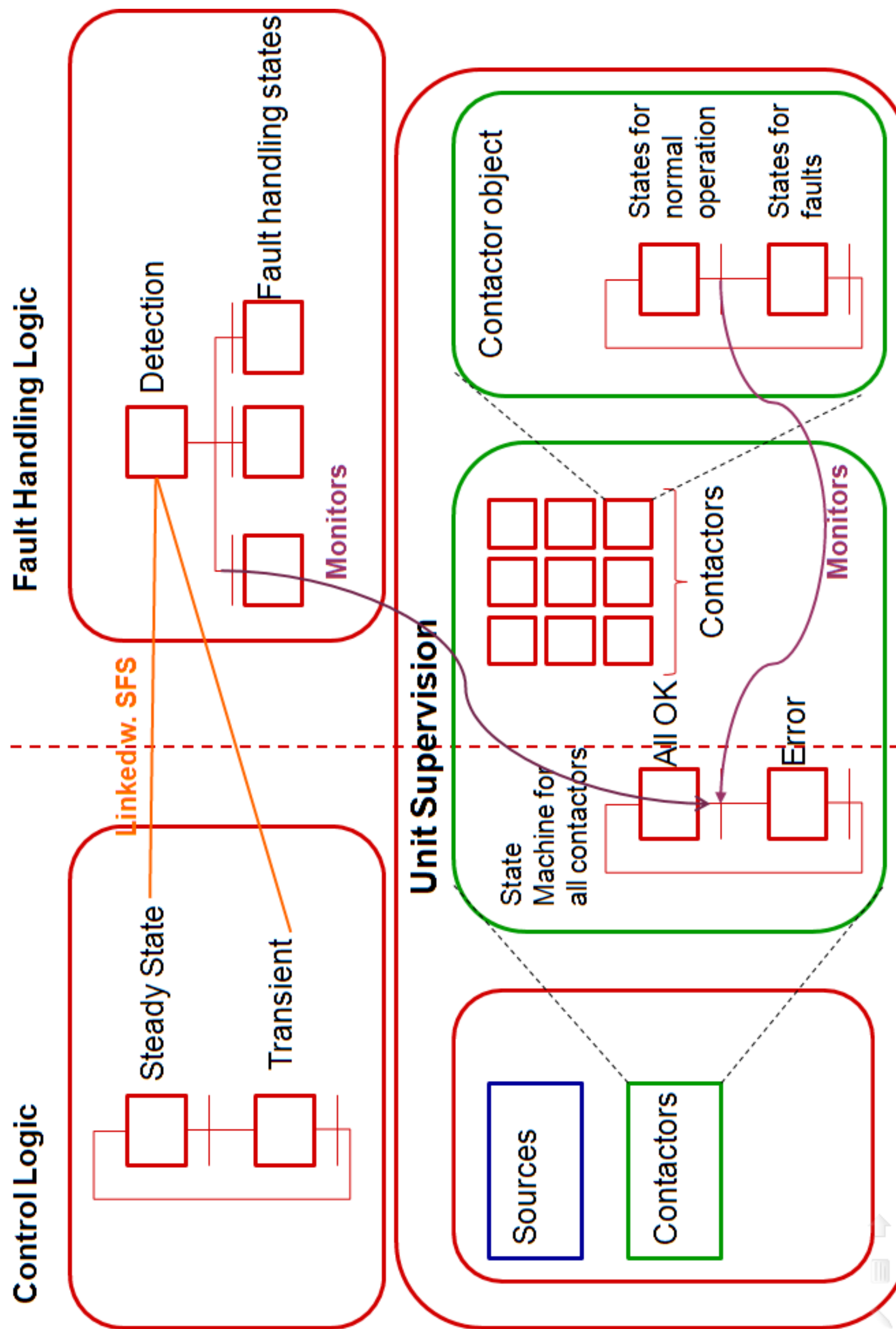

Figure 4.5: JGrafchart Design Concept

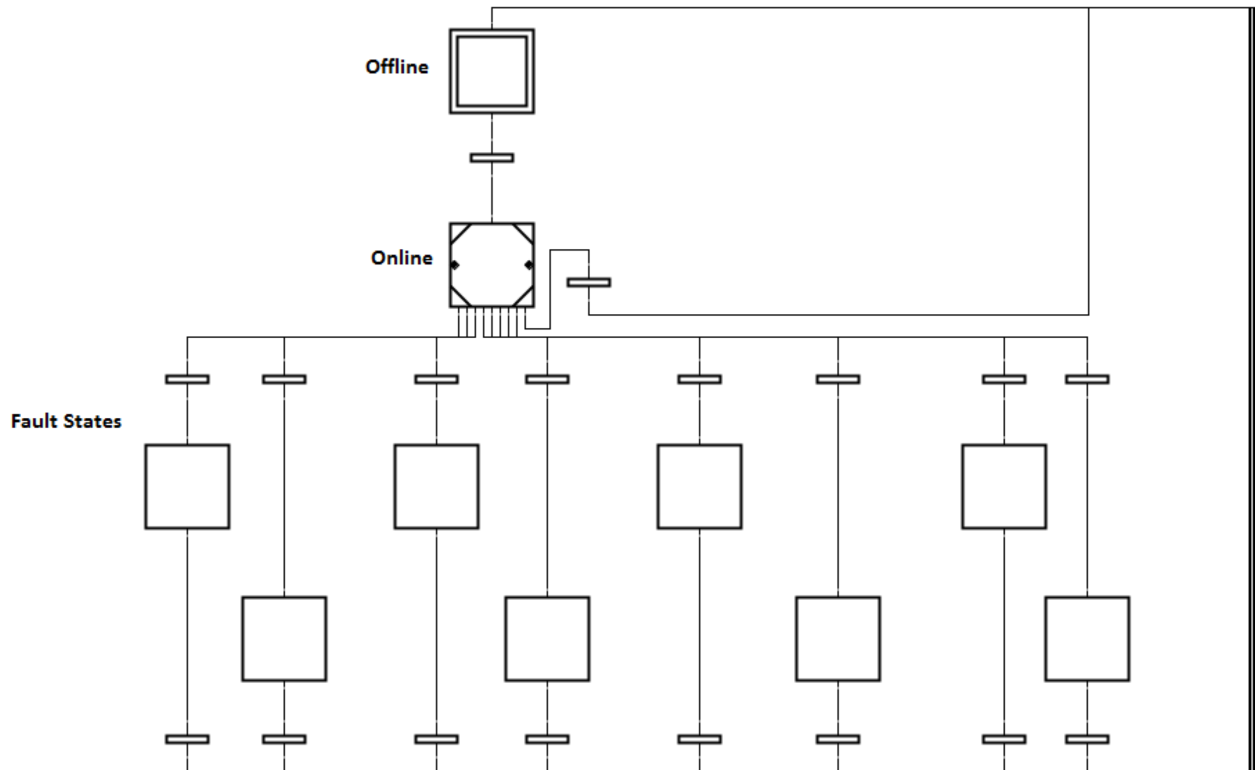

Figure 4.6: Generator State Machine, Top Level

confirmed in any of the fault detection macro steps, it will exit through the $To_ < FaultName > _Fault$ to its respective fault step at the top layer, see Figure 4.6. Each fault detection macro step has a second exit stub, which is fired if the generator for some reason is no longer online. The outgoing transitions connected to each fault detection macro step are set true by default, since all logic is handled internally in each fault detection macro step.

In Figure 4.8, the body of the Undervoltage fault detection macro step is displayed. When entered, the state machine immediately transits to the Online step through a default true transition. Thus, this step is activated as soon as the Online macro step at the top layer is entered. The transitions following the Online step in the Undervoltage fault detection macro step monitor the values of the generator voltage sensors as well as the generator status (Online/Offline). If an undervoltage is sensed and an undervoltage fault is not yet confirmed it is called an undervoltage condition. From the UnderVoltageCondition step it is possible to transient into an error state, if the timer expires, or go back to the Online step, depending on whether the sensor values are back to normal.

If the top level Online macro step is exited to one of the fault states, the state machine for all generators transits into its error state. This triggers a transition at the fault handling layer which is responsible for handling all generator faults.

TRU Similar to the generator model, the TRU state machine consists of an initial state Offline, a macro step Online and steps representing the different errors. The state machine transits to Online when the TRU status, which is retreived from the plant, is equal to one. This structure is shown in Figure 4.9. Since a TRU may experience two kinds of faults, the same macro step architecture as for generators is applied. Thus, when a TRU is brought online, both OverCurrentDetection and OverTempDetection in the top level online macro body activate immediately. The top level online macro step body can be seen in Figure 4.10. If the system

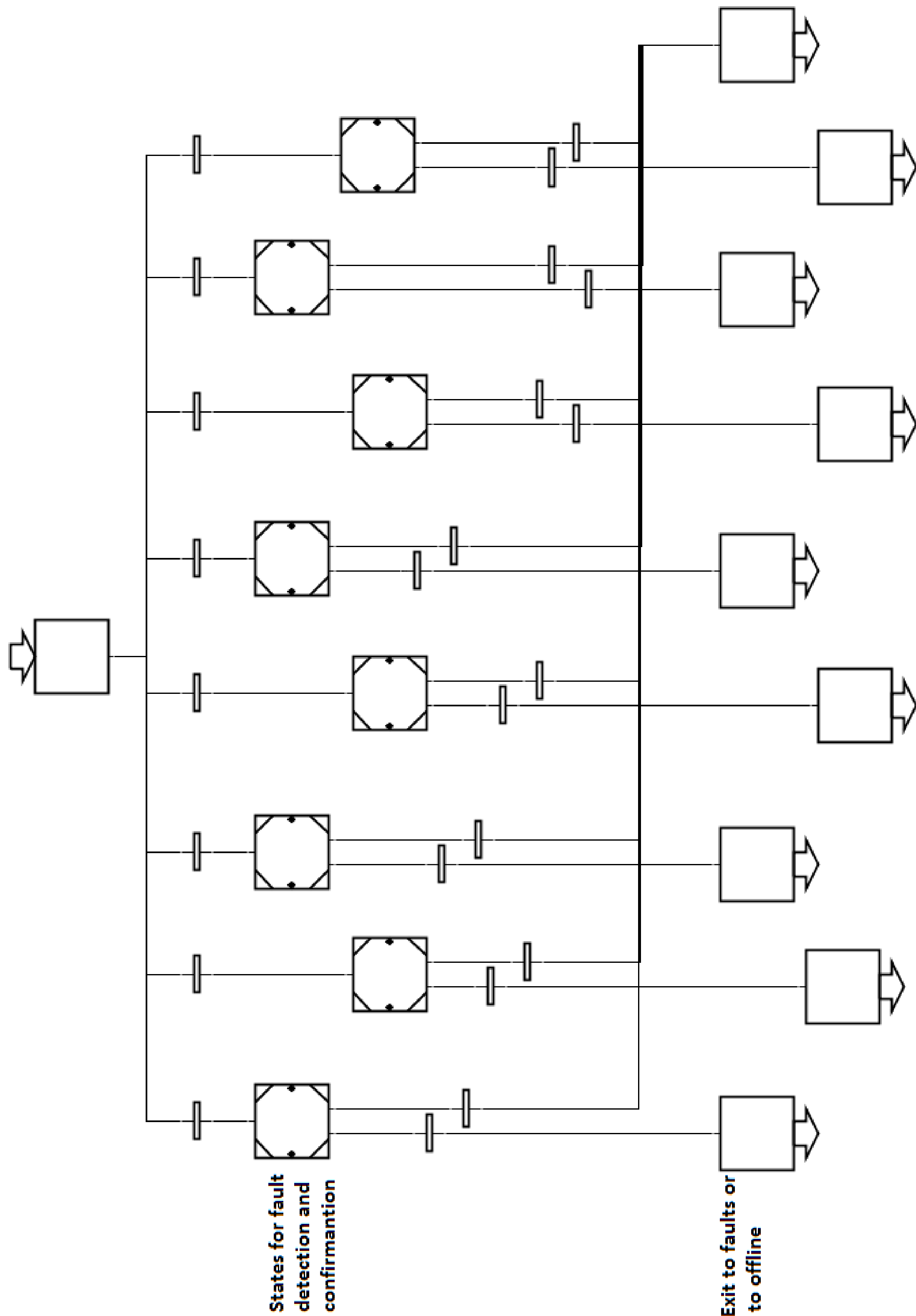

Figure 4.7: Generator State Machine, Macro Step Body

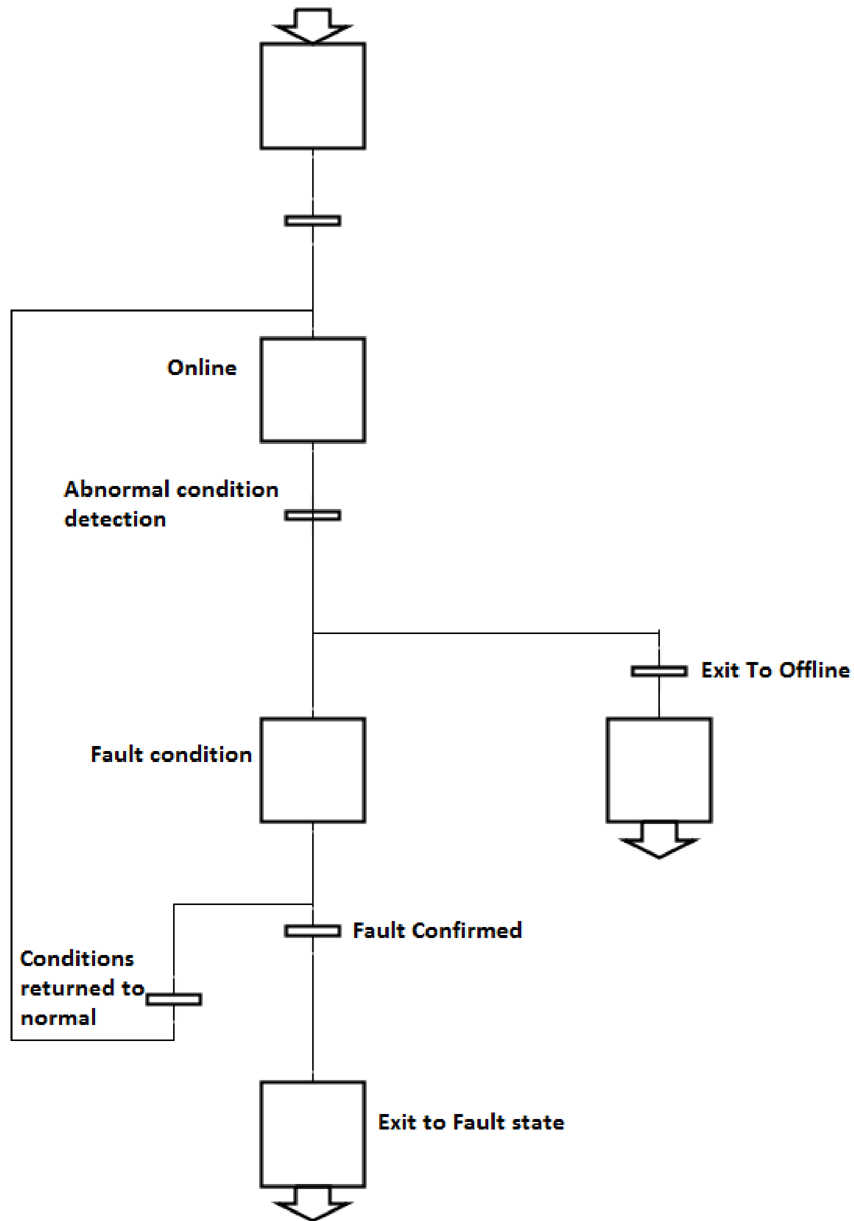

Figure 4.8: Generator State Machine, Undervoltage Fault Detection Layer

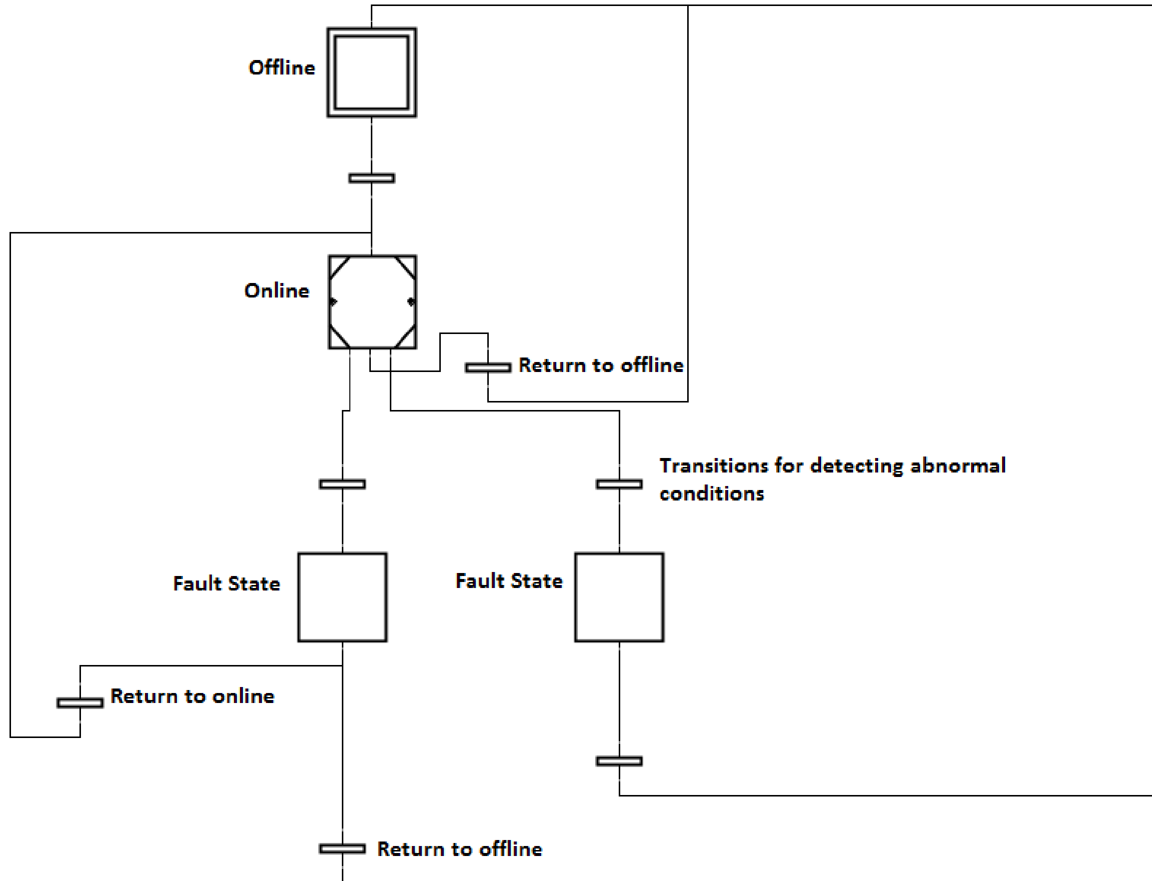

Figure 4.9: TRU State Machine, Top Level

experiences an overcurrent, depending on where it is located it is not always the case that the TRU itself is forced offline, which is why it is possible to move from OverCurrentFault back to Online. This descision is taken at the fault handling layer and is described thoroughly in the fault handling implementation section.

In Figure 4.11, the fault detection macro step for Overcurrent is displayed. Except for enter and exit steps, there are six steps each representing a level of overcurrent and one Online step modeling normal circumstances. The outgoing transitions from the Online step monitors the TRU current sensor as well as the TRU status sensor. The transitions connected to the $OC < Number >$ states enable once the TRU current sensor value is within a specified range. The reason for having multiple states modeling overcurrent is that depending on how much overcurrent the system is experiencing, the fault handling layer must be notified at different times, i.e. if the overcurrent is very high the system must handle the error fast. If an overcurrent is detected, the state machine leaves the Online step and the transition whose range includes the current sensor value fires.

Since overcurrent is considered to be a severe fault, system configuration is inhibited until the times expires and the fault handling layer isolates the overcurrent.

Implementation of Fault Handling Layer

The fault handling layer consists of one ordinary step, the detection step, and several procedure steps, one for each type of component, see Figure 4.12. A step fusion set links the detection step to the nominal control, meaning that whenever a request is processed the detection step runs in parallel and monitors the state machines in the component type workspaces. If, for example, a contactor get stuck open or closed

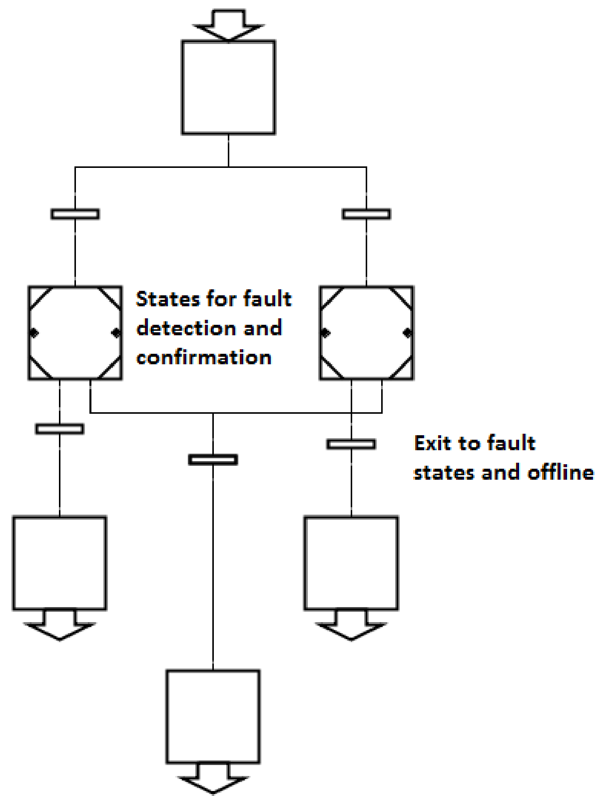

Figure 4.10: TRU State Machine, Macro Step Body

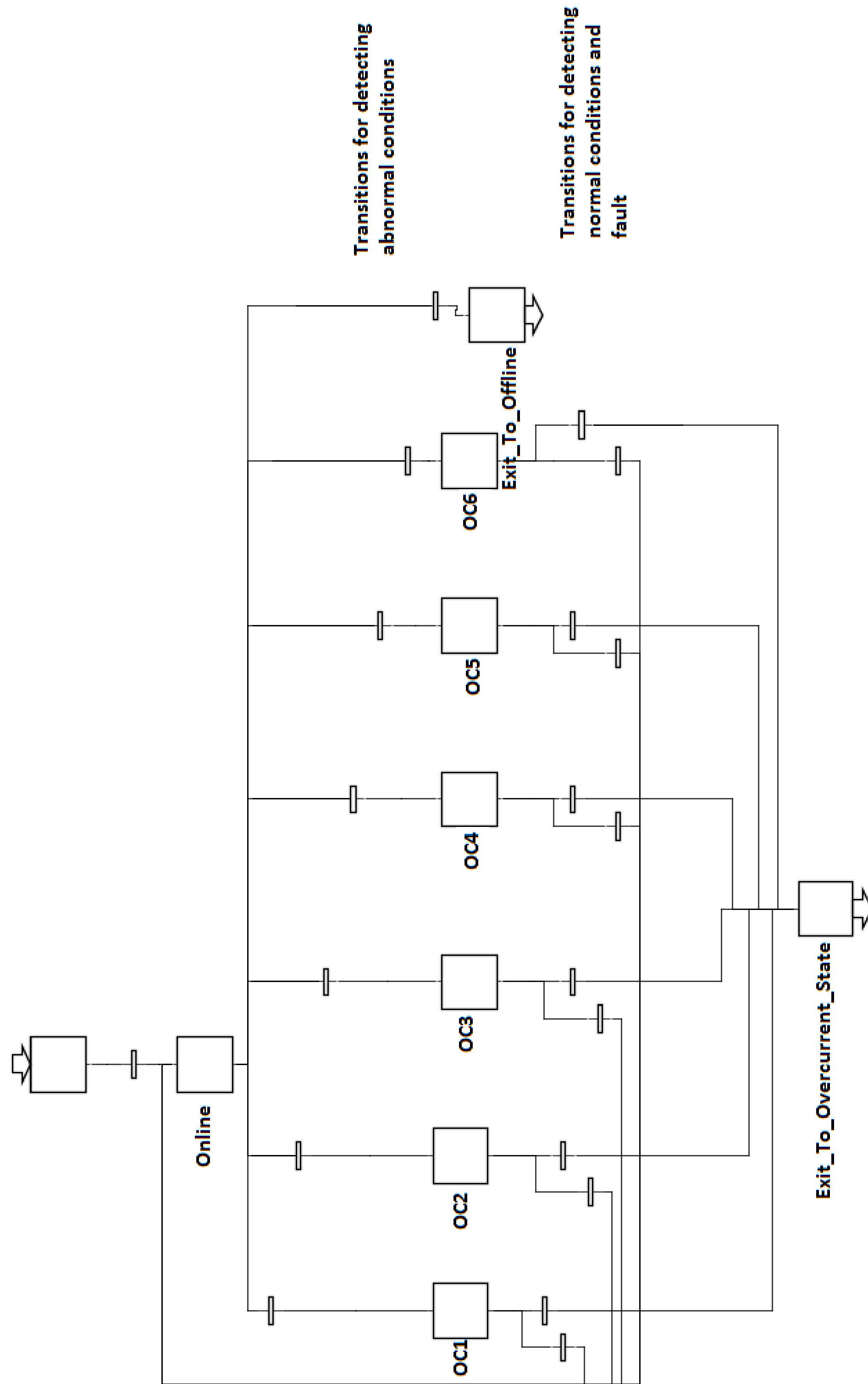

Figure 4.11: TRU State Machine, TRU Overcurrent Detection

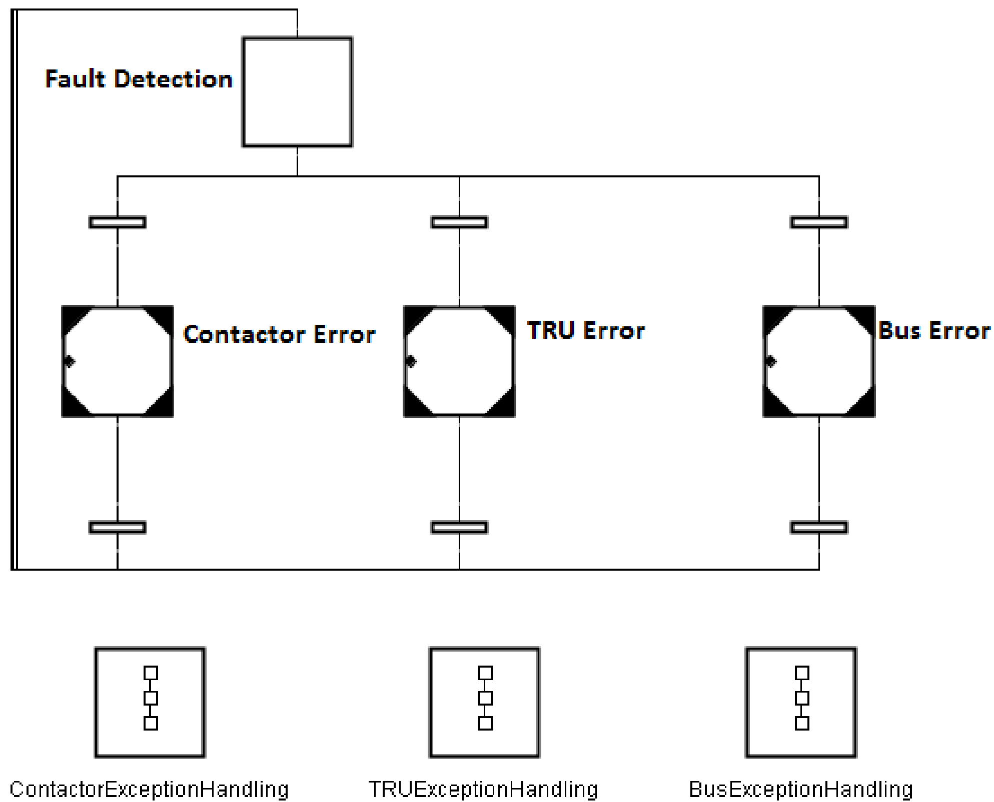

Figure 4.12: Fault handling layer

the state machine for all contactors will transient into its error state. This, in turn, will fire the transition monitoring the state machine for all contactors, the contactor exception handling procedure is called and the execution of the sequential control is aborted since the step fusion set is set to be abortive. When the fault handling procedure is exited, the state machine returns to the detection step and the execution of the sequential control is re-started according to the step fusion set definition.

4.3.3 Modifications Made to the JGrafchart Tool During Implementation

In order to build an equivalent model in JGrafchart, some new features of the JGrafchart tool were added. All changes were made by the developers of the tool at the Department of Automatic Control at Lund University. Two major changes was made, the possibility of having priorities on transitions and the macro step with resume. If a macro step is part of a SFS, it is now possible to resume execution from the state it was in before the SFS abortion.

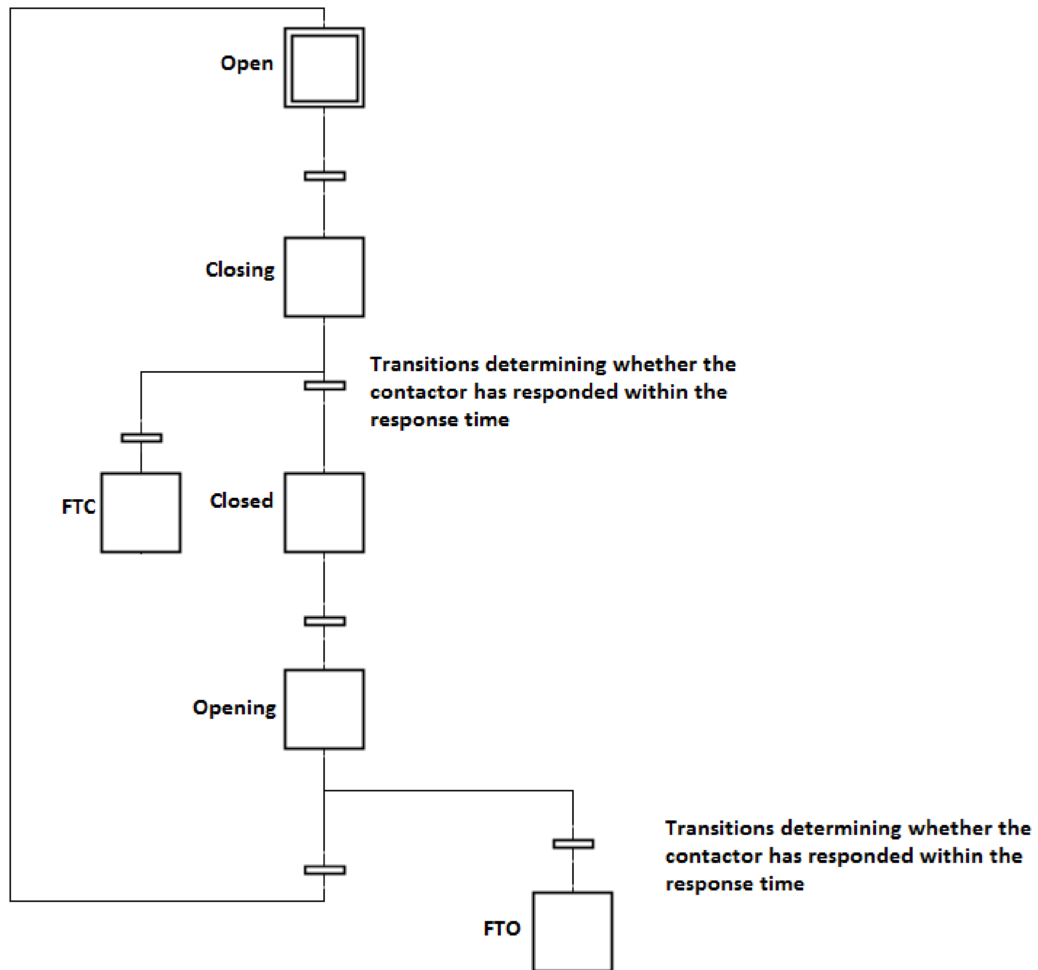

Figure 4.13: State Machine for a normally open contactor

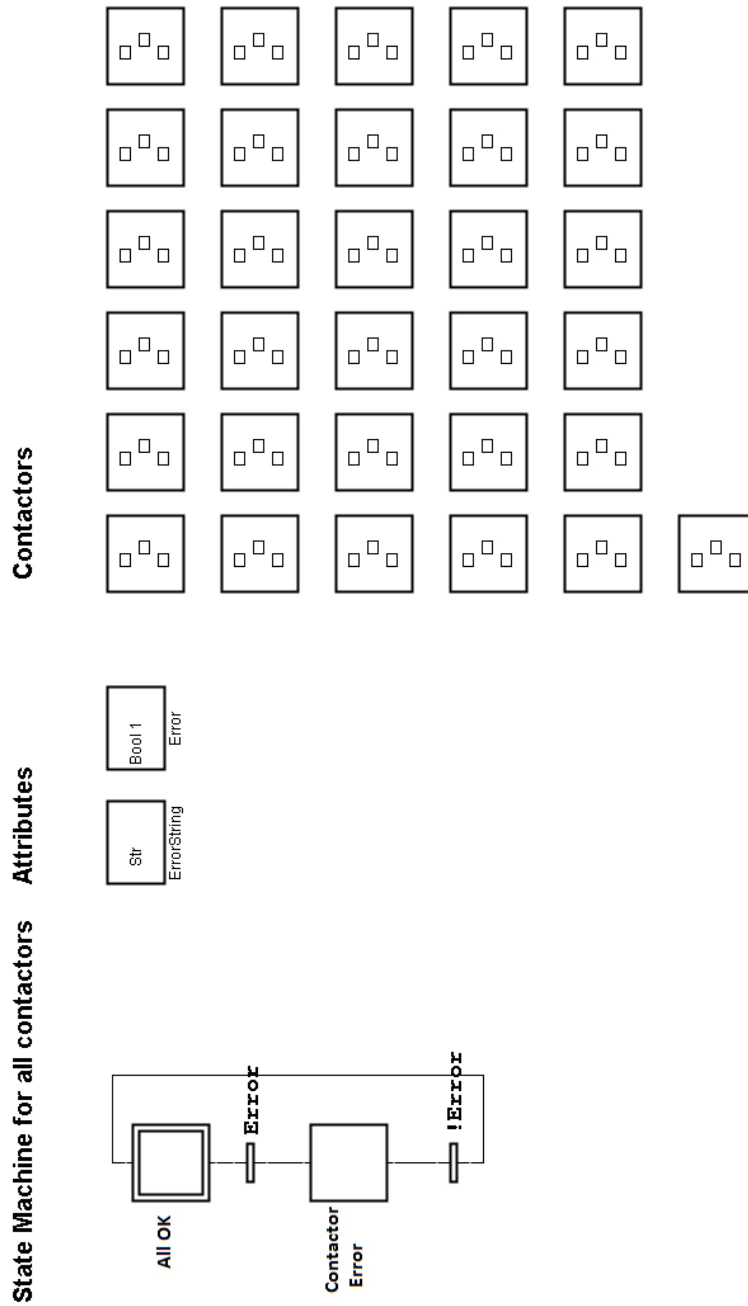

Figure 4.14: Workspace showing all contactors and their overall status (Error or not Error)

4.4 Results

In this section, the outcome of the Simulink/JGrafchart implementation is presented and compared to the original model. The criteria used to determine whether the implementations were equivalent or not was to run different scenarios and compare the outputs. The scenarios included turning sources on and off and inject faults. To determine whether the routing of electric power was equivalent in both models, the contactor states were compared. If the contactor configuration was the same it could be concluded that power was distributed in an equivalent manner. To confirm that the sequential control algorithm behaved consistently, i.e. that for each request the same sequence was executed in both models, the output signals both from Stateflow and JGrafchart were compared.

Due to prorietary reasons, the simulation plots cannot be published in this thesis. However, the sequential control behaved consistently in both models.

Chapter 5

Conclusion and Future Work

5.1 General Discussion of Results

The aim of the first part of the thesis was to investigate the suitability of the MoC behind JGrafchart for aircraft electric systems. Since JGrafchart is intended for sequential control applications, as an initial approach only the sequentialized parts were targeted. Once proven feasible to use JGrafchart instead of Stateflow for the sequential parts, the JGrafchart implementation was extended to include fault handling and make use of object oriented features. General conclusions, remarks and ideas for future improvements are discussed in this chapter.

After implementing the first part where only the sequential control was considered it could be concluded that application of JGrafchart provided satisfactory results. Replacing the Stateflow blocks with JGrafchart equivalents produced the same output given a specific scenario. However, since the Stateflow blocks were rather simple the completeness of the comparison is questionable. If, for example, the Stateflow state machines had been Mealy machines the transformation to JGrafchart would most likely require more effort and result in a larger number of states and transitions. Furthermore, if the system modeled by the Mealy machine had been particularly time critical, the JGrafchart Moore machine equivalent might have been too slow to produce an equivalent output. In conclusion, one point of interest in this thesis was to investigate whether JGrafchart is suitable for modeling the sequentialized control in aircraft electric systems or not. For the system considered in this thesis it was feasible to replace Stateflow blocks with JGrafchart charts, however if that still holds for more complex Stateflow blocks cannot be commented on since that lies outside the scope of this project.

The implementation of the fault handling layer and component state machines made use of more features in JGrafchart and approached the fault handling differently compared to the Simulink/Stateflow model. During execution the states of all individual components are monitored. All component state machines have states for normal operation as well as for faults which together with transitions defines the possible paths between states. This puts restrictions on system behavior in the sense that all transitions are not reachable from each state. In the Simulink/Stateflow model where no state machines exist for the components, the state of a component is represented as a boolean value. This puts higher demands on the developer since it is possible to move between states in a way contradicting the natural behavior of the component. An example of this, which was found when the scenarios were run, is the generator under voltage fault handling. In the Simulink/Stateflow model, the protection layer erroneously assumes an under voltage fault when a generator is offline. Since the generator is offline, the voltage is equal to zero which triggers the under voltage protection actions without taking the generator state into account. In the JGrafchart implementation it is not possible to transient from offline to any kind of error, without first passing through the online state which is why this error never occurs. The bottom line is that the component state machine defines what actions are allowed depending on current state whereas the Simulink/Stateflow model relies on boolean variables representing states that may contradict each other. Thus, less constraints are put on the Simulink/Stateflow model which leaves more room for implementation errors.

In the implementation of the fault handling layer and component state machines, object oriented features are used to some extent. All components are modeled as objects using workspaces. However, since the object

orientation is limited in terms of classes, inheritance, etc. it is not possible for each component to be an instance of its class, i.e. it would be desirable if a contactor object could belong to the class contactor. As it is now, each contactor in the system is represented by a JGrafchart workspace, see Figure 4.14. In turn, each workspace contains a state machine modeling the behavior of the contactor, which is the same for all contactors except for some small customizations. Thus, there is a lot of redundancy throughout the model. It would be possible to rid the system of redundancy by using process calls. This way the component state machine would only appear once, and the customized parameters such as name, input sockets and output sockets would be passed a parameters to the procedure. However, this approach was not chosen since it reduces readability and the graphical representation becomes less intuitive.

An additional aspect in favor of having component state machines is reusability. Since all aircrafts basically consist of the same components, e.g. generators, buses, contactors etc., the backbone of the state machine implementation and concept of fault handling can be used for different aircrafts with some customization.

A recurring issue during implementation was the timing. Since JGrafchart is not integrated with Simulink, an interface had to be implemented for them to communicate. The sending of signals back and forth introduced a small delay in the Simulink/JGrafchart implementation.

Also, the way of addressing the problem formulation is discussed. It is obviously not ideal to investigate the suitability of a certain tool when the starting point is an already finished implementation in a similar tool. The problem is that instead of utilizing the new tools functionality, the implementation tailored for the original tool is to a large extent copied not doing the new tool justice. When presented with an already working solution it is hard rethink the design and come up with alternate approaches.

As a final comment, an intuitive design is easy to achieve by making use of the object oriented features in JGrafchart. Also, the structuring capabilities in JGrafchart decrease the probability of programming errors and ease implementation of complex applications. The structuring capabilities in Stateflow are primitive compared to JGrafchart, which except for chart layering also supports more refined functionalities such as the step fusion set, exception transitions and object orientation. However, the tool needs to be further developed to better handle large control applications. It might be possible to apply JGrafchart early in the development phase to build an abstracted model, which can be used early on for analysis using model checking.

5.2 Future Work

In order for JGrafchart to provide more value the high level version needs to be implemented. If it was possible to have token objects a lot of redundant code could be avoided while still maintaining the graphical clarity. The main conclusion drawn from comparing the different implementations (Simulink/Stateflow and Simulink/JGrafchart) was that by using component state machines the model became easier to comprehend and some erroneous behavior was avoided. The drawback was that each component required its own chart which makes it hard to monitor the state of all components simultaneously. If, however, object tokens were introduced, components of the same type could be collected in one state machine and monitoring the states of all components belonging to that type would only require looking at one chart. Thus, readability would improve significantly. In Figure 5.1 the concept is shown.

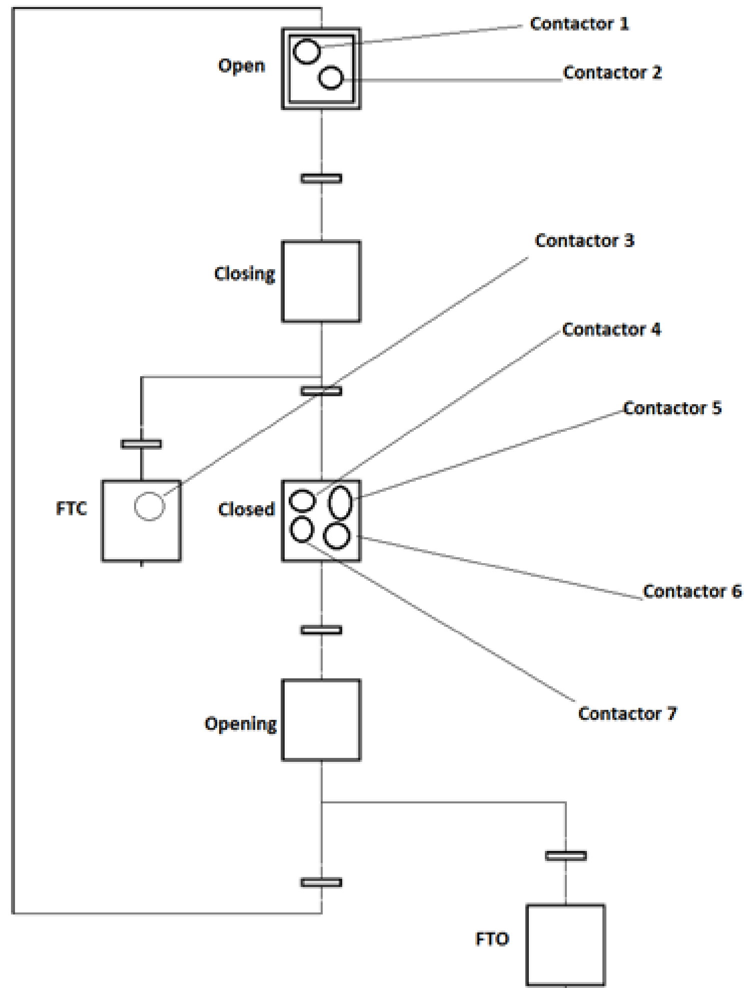

Figure 5.1: Concept Of Object Tokens In JGrafchart

Part II

Modeling of IMA Control Systems

Chapter 6

Integrated Modular Avionics

6.1 Background and Introduction to IMA

Integrated Modular Avionics (IMA) is a term used for distributed computer networks in aircrafts and a concept describing how to integrate distributed modules. In the aircraft industry today there is an increasing drive towards IMA. One reason for this is that it replaces the point-to-point networking with virtual links and thus reduces the required wiring. This is an important improvement since the wires together with isolation are the main contributor to aircraft weight. Furthermore the weight of the aircraft is a key point of cost. Another important reason for using IMA is that it increases system reliance since the network protocol used is deterministic and redundant.

In distributed computer networks it is not unusual for actuators, sensors and controllers to be on different networks with different protocols. As control system complexity increases the ability of evaluation at an early design stage is highly desirable. By being aware of the constraints both on control algorithm and software, developers can early on comment on system restrictions. From a cost perspective it is valuable to be able to analyze a system before it is a finished product, since adjustments grow more expensive with each development phase.

In order to understand the effect of delay and jitter that might occur due to distribution, simulation tools are needed. Simulation tools can be used to investigate the constraints and possibilities of the system, with respect to the control algorithm as well as the software design. Delay and jitter might lead to poor control performance, and therefore it is important to have an idea about how much interference the system can handle before becoming unstable.

As for the first part of the thesis, this part also aims to investigate the suitability of a certain tool on HS systems, namely TrueTime. TrueTime is a Simulink-based simulation tool for real-time control applications. The main objectives of this part are to apply TrueTime to HS systems and to extend the TrueTime library to typical aircraft control systems that employ IMA. The motivation for doing this is that TrueTime then could be used to build an abstracted model containing all critical elements, run simulations and analyze the impact of delay and interference traffic on control performance. Furthermore, the analysis result can be used to help drive requirements which normally are difficult to assess, already at a design stage [18].

6.2 Theory

6.2.1 Network Protocols

As mentioned, control systems are often distributed over different networks with different network protocols. This section aims to give a background on the protocols present in the system that was modeled in TrueTime.

TDMA

The Time Division Multiple Access (TDMA) is a communication channel access method which makes it possible for several nodes to exchange data over the same frequency channel. The nodes on the network send

messages one after another since each node on the network has a dedicated timeslot for when it is allowed to send. In this way each node is able to use the entire transmission channel in its specified timeslot thus making TDMA a very efficient transmission method. This results in a stream of messages which are being sent in a fast sequence over the network and collisions are avoided [19].

CAN

The Controller Area Network (CAN) protocol is a bus standard used for communication between micro controllers and devices. The transmission technique on a CAN bus is broadcast, which means every node hears all messages being sent on the network. However, it is possible to introduce filtering so each node only cares about message intended for them [20].

The order in which nodes are allowed to transmit is determined by an identifier in the message. If two or more nodes send at the same time the message with the lowest identifier will be transmitted, since a low identifier equals a high priority [21].

Bit rates in CAN varies up to a maximum of 1 Mbps, which makes it suitable for real time control. This bit rate however can only be achieved if the nodes are distributed within 40 meters from each other. When the distance between the nodes increases the network speed decreases [22].

CAN features extensive error checking, which makes the protocol reliable. The different error checks are stuffing error, bit error, checksum error, frame error and acknowledgement error [22].

Ethernet

The network protocol used in Ethernet is called Carrier Sense, Multiple Access and Collision Detection (CSMA/CD). It allows several nodes to communicate over a shared medium. The specifications for this protocol are that each node senses the network and whenever it is idle the nodes send messages if they have any. If the message being sent collides with another message the nodes detect this and retransmit later using some appropriate back-off strategy. The drawback with this protocol is that no centralized coordination exists and collisions will result in a non-deterministic behavior.

Collisions may lead to large delays and theoretically a message might collide an infinite number of times and never be transmitted. This kind of situation needs to be avoided in avionics networks.

Switched Ethernet

In Full-duplex Switched Ethernet each avionic subsystem (e.g. the autopilot) has an End System which connects it to a switch using a full-duplex link. The switch comprises one transmit buffer and one receive buffer for each End System, one memory bus connecting all the buffers, one I/O Processing Unit (CPU) as well as a Forwarding table. The I/O Processing Unit transfers the packages from the receive buffers to the intended transmit buffers. The destination address of the package is the virtual link identifier. By knowing the virtual link identifier and using the forwarding table, it can be determined which transmit buffer the packages should be forwarded to. The package is copied and via the memory bus sent to the transmit buffer in FIFO order. With this type of store and forward technique collisions are avoided. The disadvantage is that it might result in jitter and delays since packages may have to wait for other packages to be transmitted [23].

AFDX

In order to provide a high Quality of Service (QoS) in Aircraft Data Networks (ADNs) the network protocol needs to ensure limited jitter, guaranteed bandwidth, upper bounded transmit latency and a low Bit Error Ratio (BER).

Avionics full duplex switched Ethernet (AFDX) was developed to satisfy the requirements on the ADNs. It is made up of three main components; End systems (ESs), Virtual Links (VLs) and switches. Each end system is connected to a switch through a cable. This means that two end systems communicate with each other over a single physical link. However, it is possible to have multiple logical communication links who behave like separate physical links, even though they are using the same physical link. These are called Virtual Links (VLs).

AFDX is based on switched Ethernet, with two important differences. It is deterministic instead of probabilistic and redundant to increase reliability. The deterministic behavior is assured by traffic control. This means that each logical communication channel is promised a certain bandwidth, and if a channel exceeds its bandwidth it will be dropped by the switch so that it does not affect the other channels. The presence of two independent networks ensure that the same data stream is sent simultaneously. This lowers the probability for erroneous data since only the uncorrupted stream is forwarded.

A message is uniquely identified by its UDP port source, IP source, the VL MAC address, the IP destination and the UDP port destination. The MAC destination address contains the VL.

The AFDX has two different port types, communication ports and SAP ports. Both of these can receive and transmit data. A communication port can either be a sampling port or a queuing port, which are both based on UDP. The sampling port buffer contains only one message, which will be overwritten when a new message arrives. The queuing port however, has a FIFO message queue, and the messages are queued instead of overwritten. The SAP port communicates with other AFDX networks or LANs.

In AFDX, all frames from the VLs must be multiplexed onto the physical link. This is handled by the VL scheduler, who uses time multiplexing, i.e. it divides the time domain into slots. The slot time is called Bandwidth Allocation Gap (BAG) and is unique for each VL. The BAG range is between 1-128 ms and must be a power of two [24].

Brief Comparison Between Switched Ethernet and AFDX

This section briefly discusses the differences between switched ethernet and AFDX. Naturally, there are many similarities since AFDX is derived from switched ethernet but some important extensions have been implemented to provide network characteristics satisfactory for avionics applications.

Determinism Determinism is highly desirable in avionics and in AFDX this is guaranteed by traffic control. Traffic control means dedicating each virtual link a certain bandwidth, and if the virtual link exceeds that bandwidth it will be dropped by the switch. This limits delays and jitter in the system [24].

Redundancy Since safety is critical in avionics, all packets sent between two end systems are sent on two independent networks, i.e. the same data stream is sent twice. The receiving end system therefore gets two copies of each packet. If a packet gets corrupted, the copy of that packet sent over the other network can be used instead. If both copies are uncorrupted on the receiver side, only one of them are passed along by what is called redundancy management [24].

Chapter 7

TrueTime

7.0.2 TrueTime

Today it is common for systems to be distributed over multi-tasking kernel nodes, which communicate on different networks. In these systems the nodes compete for the shared resources (The CPU and bandwidth) and the distribution of bandwidth is determined by the network protocol. Since the shared resources are limited in terms of bandwidth different kinds of delays arise, such as transmission delays and back-off times. The delays might lower the control performance significantly, which is why it is important to identify them early in the development process, preferably at the design stage. The bottom line is that both the control algorithm and control software need to be studied and taken into account simultaneously early on in the development process.

TrueTime is a simulation tool used for real-time systems and has been developed at Lund University since 1999. Its main purpose is to analyze the impact of latency on control performance in networked control applications. TrueTime is based on Matlab/Simulink and applications can be built both as .cpp and .m files[18].

The TrueTime library comprises of several blocks which are briefly described below, as well as shown in Figure 7.1.

TrueTime Kernel In the kernel node, the user can define tasks which are either periodic or sporadic. For the sporadic tasks, there also need to be an interrupt handler defined since they are event-based. The scheduling policy of the kernel is set by the user [18].

TrueTime Network The network blocks enable nodes to interchange data with each other using a specific network protocol. The supported protocols for wired communication are Ethernet, CAN, TDMA, FDMA, Round Robin, Switched Ethernet, FlexRay and PROFINET [18].

TrueTime Wireless Network TrueTime also features a block for wireless communication. The protocols supported are 802.11b WLAN and 802.15.4 ZigBee [18].

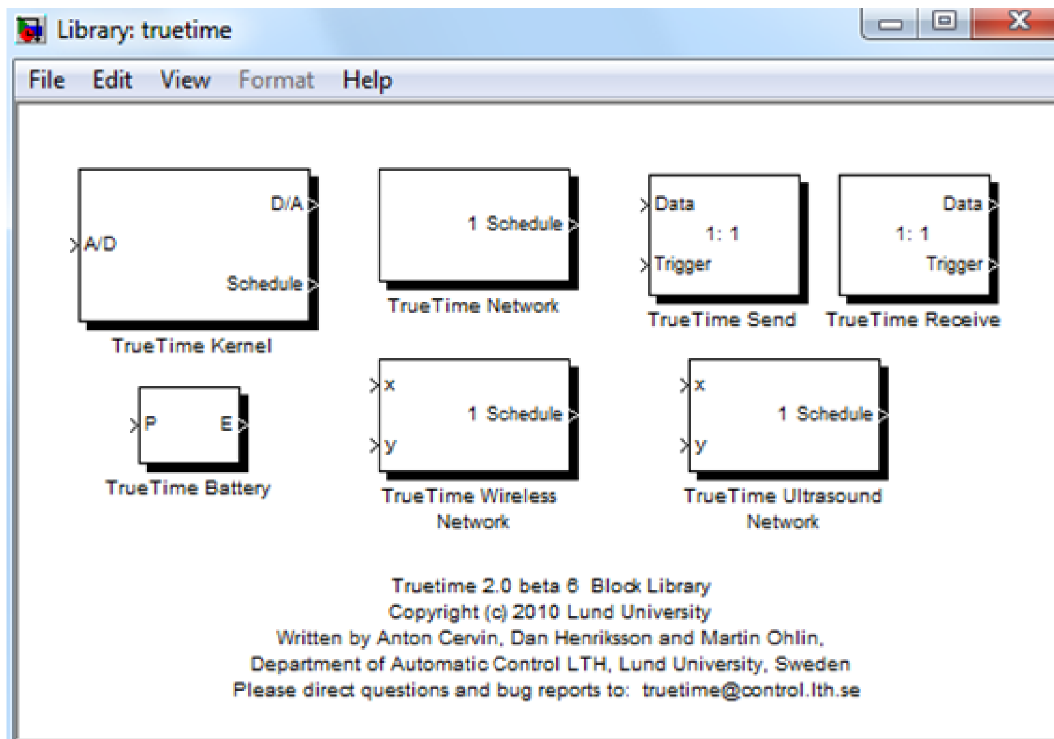

Figure 7.1: Aircraft Electric Power System

Chapter 8

TrueTime Model of IMA Use Case

8.1 An Existing IMA System

As a starting point, a typical aircraft electric power system was studied. In the system, communication takes place over three different networks, TTP, AFDX and CAN. The original system was reduced to a simpler one with similar architecture and fewer nodes, which can be seen in Figure 8.1. The reason for this was to quicker get an idea of TrueTimes suitability for such systems. In the distributed control loop, the sensor which samples the plant is on the CAN network. The sensor values are sent to the controller on the AFDX network where the control signal is computed. In turn, the control signal is sent to the actuator on the TTP network. Two gateways are needed, one between CAN and AFDX and one between AFDX and TTP.

It is assumed that the AFDX and CAN networks have background traffic and that gateways simply map payloads from frame to frame.

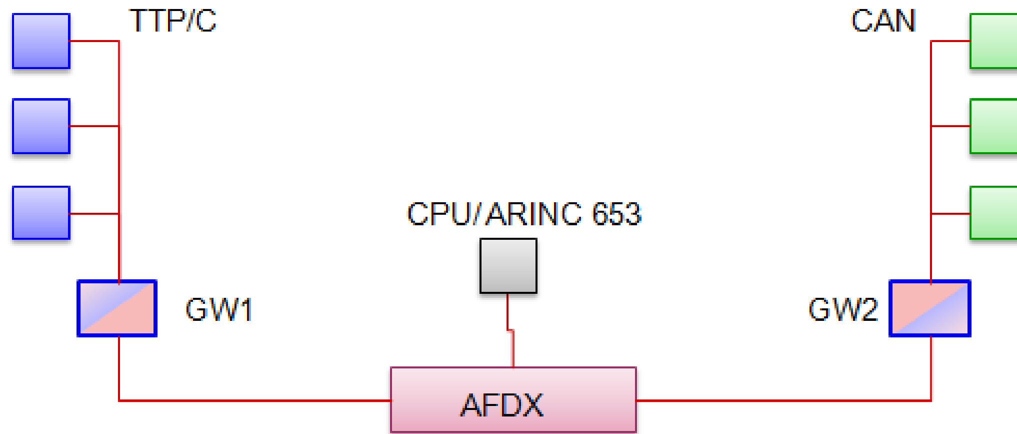

Figure 8.1: Simplified Aircraft Electric Power System

8.2 System modeled in TrueTime

8.2.1 System Modeled With Full Duplex Switched Ethernet

As described in the previous sections, communication in the system takes place over three different networks. These are modeled as TrueTime network blocks. Since AFDX is not among the supported network protocols in TrueTime, it is approximated with a full duplex switched Ethernet. The sensor, actuator, controller, interference and gateways are all implemented as TrueTime kernels. The plant is modeled as a simple second order system. In order to simulate background traffic on the CAN network, interference nodes sending dummy messages to themselves are introduced, one node sending high-priority messages and one sending low priority messages. On the AFDX network, the interfering node has to send its messages to the controller for it to have any impact. All messages sent to the controller are therefore marked, so the controller can distinguish interference messages and discard interfering traffic.

8.2.2 Implementation of AFDX

In order for TrueTime to provide value for HS, an implementation of the AFDX protocol was necessary. Since AFDX is very similar to a full duplex switched Ethernet, no new protocol was implemented. Instead modifications were made in the user defined code in the kernel nodes belonging to the AFDX network. As mentioned, the main differences between AFDX and switched Ethernet are traffic control and redundant networks. In this implementation only the traffic control aspect was considered since the redundancy does not affect control performance. Traffic control was achieved by modeling virtual links, including communication ports and bandwidth allocation gap (BAG).

In the implementation without AFDX, application tasks directly send messages to receiver nodes by using ttSendMsg. However, in AFDX a node is not allowed to send messages more frequently than what is given by the BAG for the virtual link in question. When an application task wants to send a message, it posts it to a mailbox. A separate task then reads from the mailbox, and forwards the messages to the receiving nodes with a time interval that is given by the BAG. Thus, no sending node will send more frequently than allowed. To simulate the communication ports in AFDX, monitors and mailboxes are used on the receiving side. When a message arrives, the interrupt handler checks whether the virtual link it was sent over is associated with a sampling port or queuing port. If it is a sampling message, the interrupt handler writes the message content into shared memory, which is modeled with a TrueTime monitor. Finally, application tasks simply read from the monitor. In this way, several tasks can read the content. A message will be overwritten as soon as a new message arrives, in accordance with the AFDX sampling port definition. A queuing port on the other hand, is modeled by an additional mailbox in which the messages are queued. Separate application tasks then read from the mailbox to access the content.

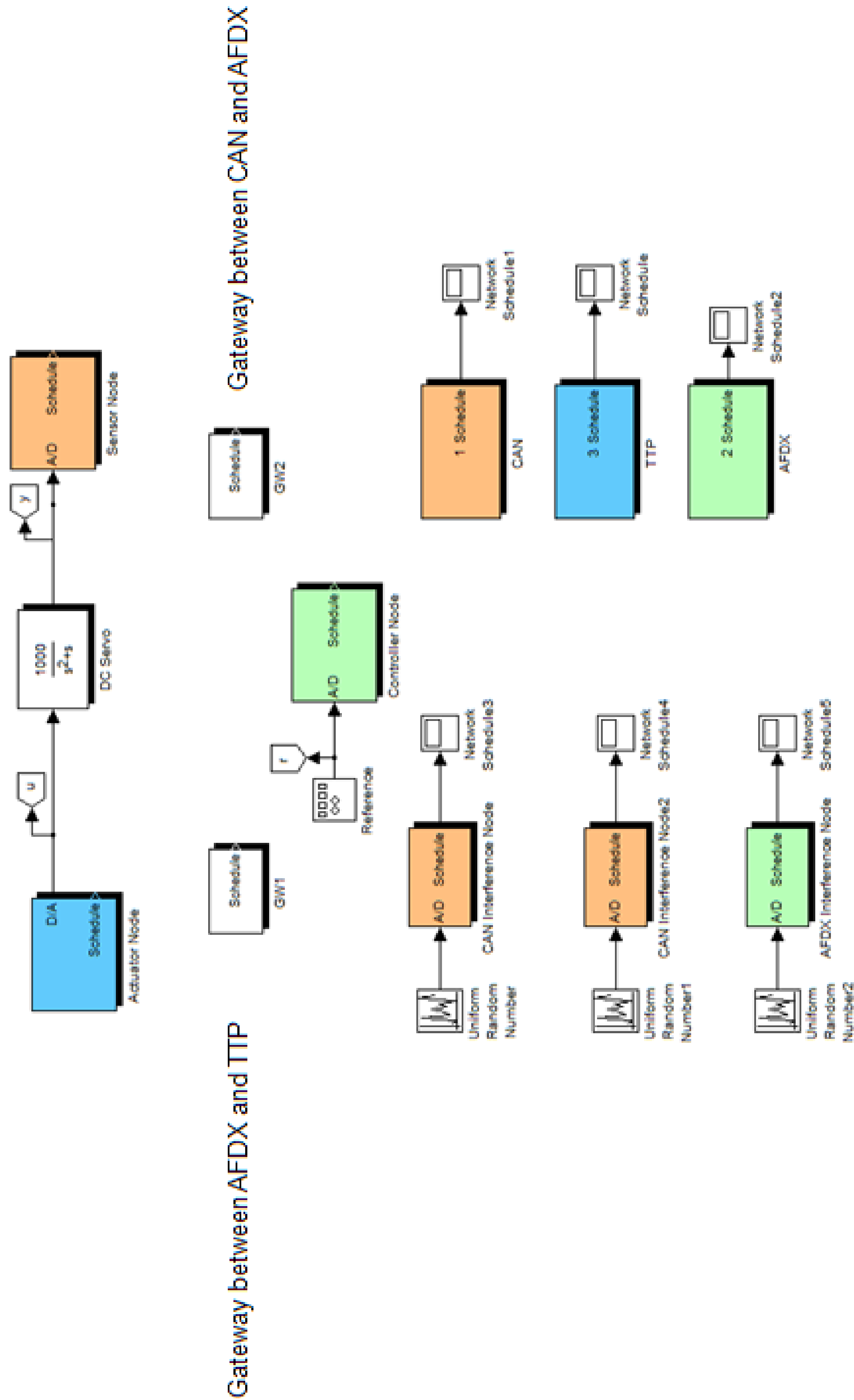

Figure 8.2: Simplified Aircraft Electric Power System modeled in TrueTime

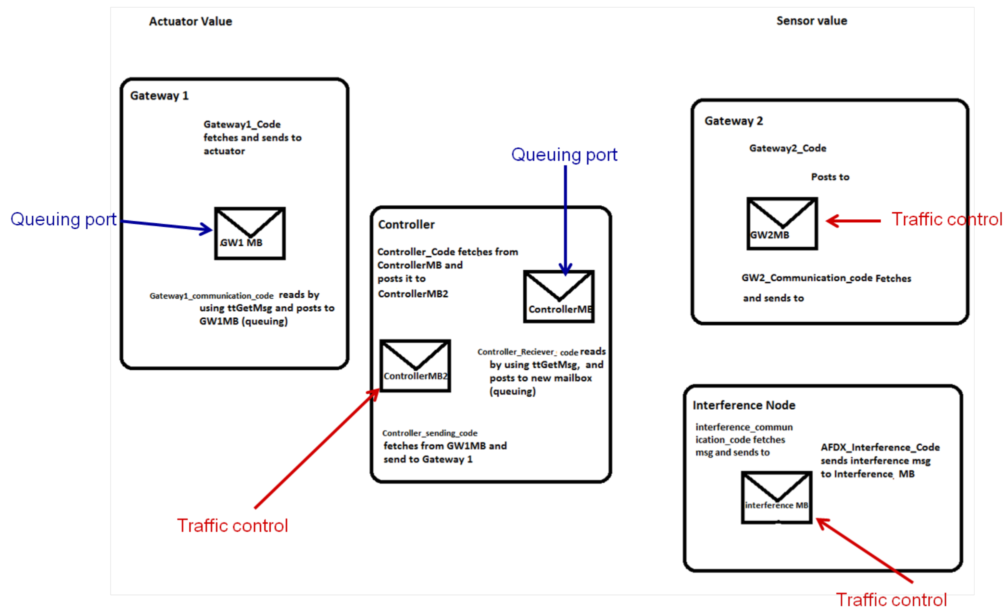

Figure 8.3: AFDX in TrueTime

In Figure 8.2.2, the message passing between nodes on the AFDX network can be seen.

8.3 Results

8.3.1 System Modeled With Full Duplex Switched Ethernet

As a first approach and proof of concept, AFDX was modeled with a Full Duplex Switched Ethernet and simulations were run to illustrate how TrueTime could be used to determine impact of delay and jitter on control performance. The scenarios included introducing interfering traffic on the networks as well as changing the delay in the gateways, both of which are areas of interest for Hamilton.

Impact of CAN network interference On Control Performance

The CAN network protocol is priority based, which means that the node with the highest priority gets to send if two nodes are competing for the bandwidth. As can be seen in Figure 8.2.1, there are three sending node on the CAN network, the sensor node and two interfering background traffic nodes. The interfering nodes have the highest and the lowest priority respectively and the sensor has the second highest priority. Figure 8.3.1, 8.3.1 and 8.3.1 show the CAN network schedule and the impact of a high priority interference node on the control performance and Figure 8.3.1 shows how the overshoot varies as a function of bandwidth used by high priority CAN interference.

For the network schedules, a signal ranges between $[0,1]* < NodeNumber >$, where a low signal indicates idle, a medium signal that the node is waiting to send and a high signal means the node is currently executing.

Impact of Gateway Delay On Control Performance

Except for investigating the impact of interfering traffic on control performance, TrueTime was used to determine how much gateway delay the system could handle before becoming unstable. The gateways in Figure 8.2.1 simply map payload from frame to frame, and are modeled as a constant delay. In Figure 8.3.1, 8.3.1 and 8.3.1 the impact of different gateway delays on control performance is shown. Figure 8.3.1 illustrates how the overshoot increases as a function of gateway delay.

8.3.2 System Modeled With AFDX

After traffic control and communication ports were implemented in the application tasks in the kernel nodes, simulations were run to show how the AFDX implementation differ from the Full Duplex Switched Ethernet implementation. In Figure 8.3.2, node three is sending messages in bursts and there is no lower limit on the time interval between two consecutive sendings. However, the same task behaves periodically on the AFDX network due to traffic shaping, which can be seen in Figure 8.3.2. With the AFDX implementation messages are spread out in time where the minimal interval between subsequent sendings are equal to the BAG. Since the BAG controls the frequency with which nodes on the AFDX network are allowed to transmit, different values of the BAG obviously affect control performance.

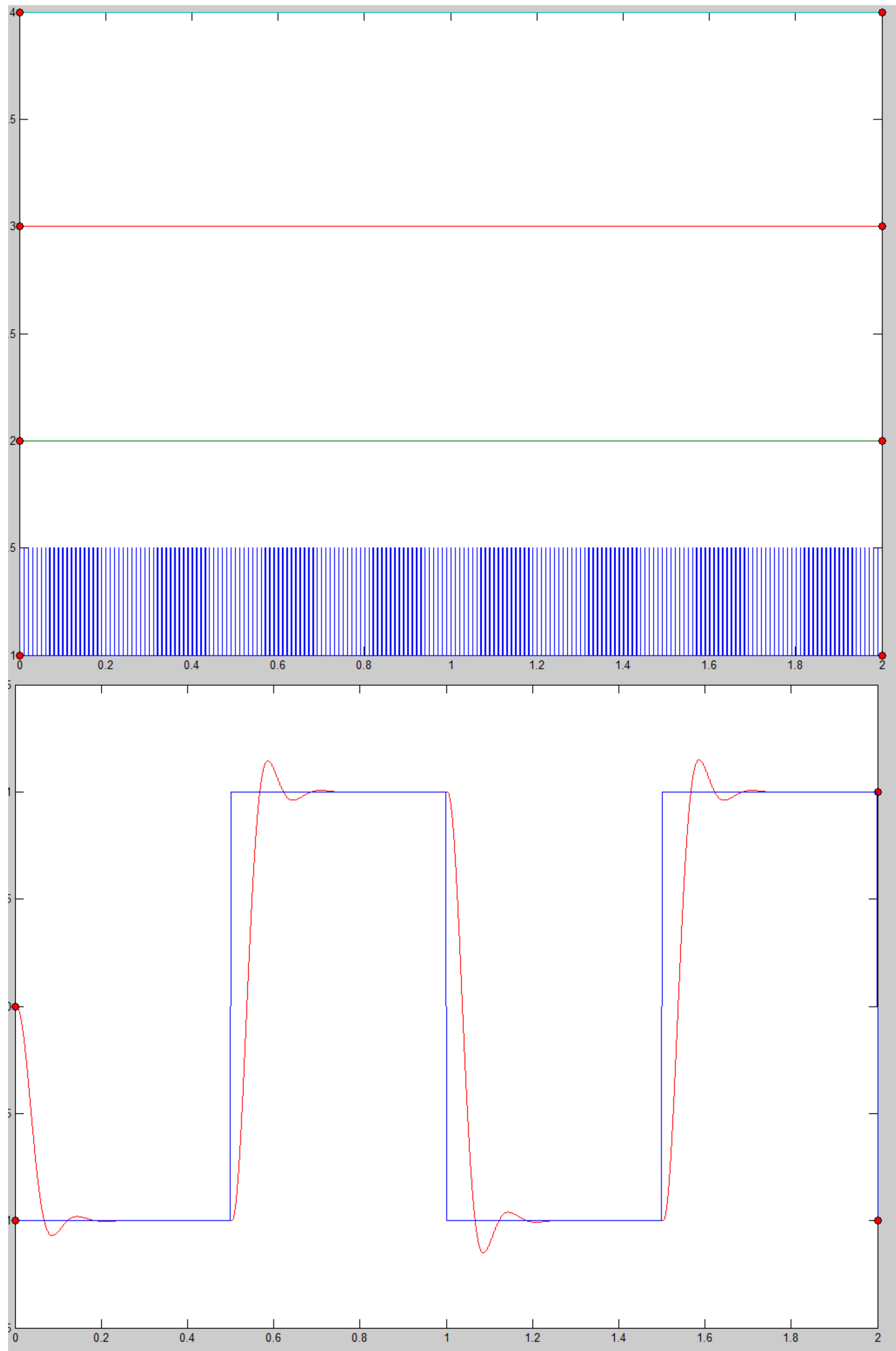

Figure 8.4: Network Schedule and Corresponding Control Performance With 0 % Bandwidth Assigned To High Priority CAN Interference Node

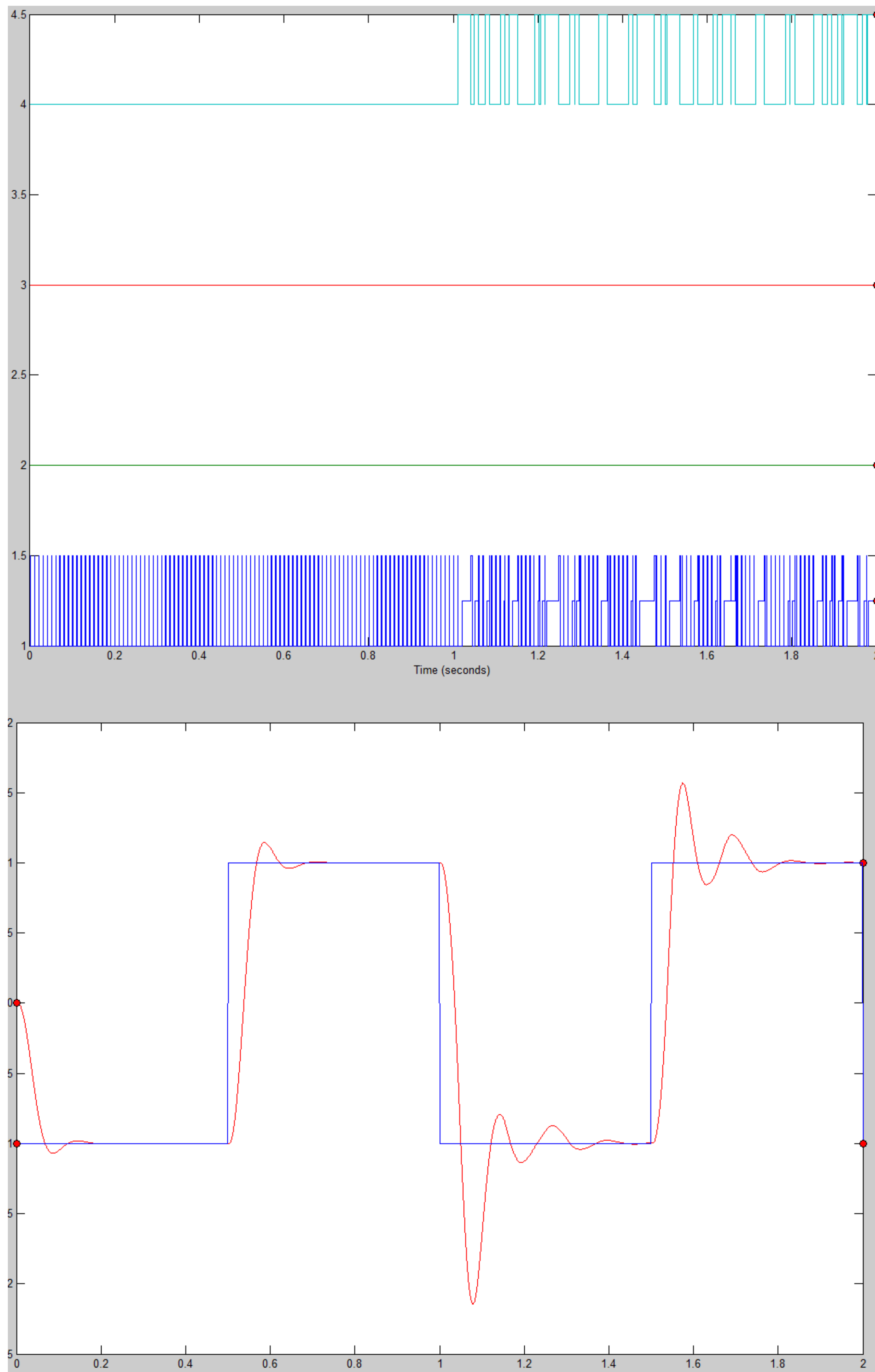

Figure 8.5: Network Schedule and Corresponding Control Performance With 5 % Bandwidth Assigned To High Priority CAN Interference Node

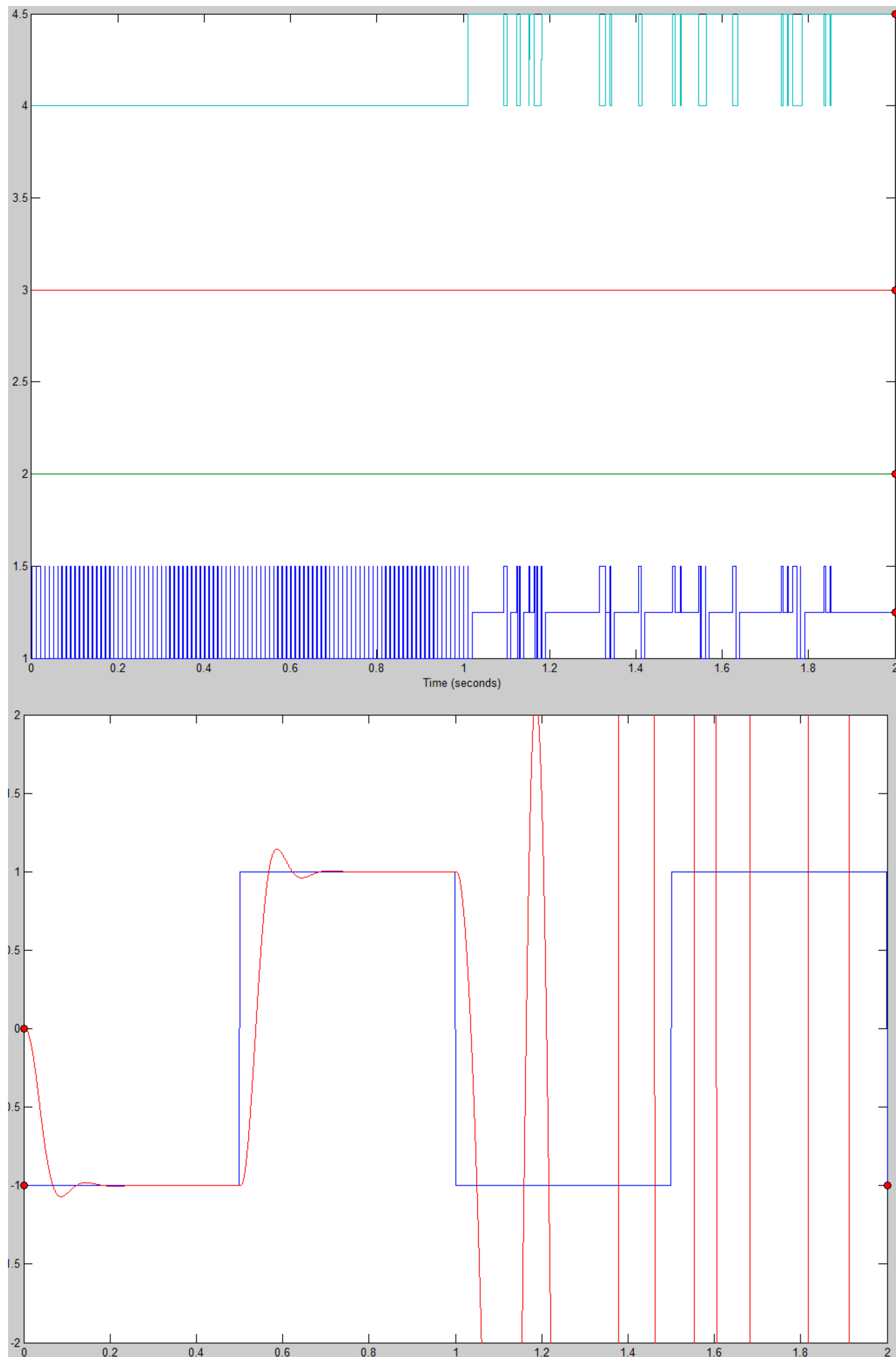

Figure 8.6: Network Schedule and Corresponding Control Performance With 10 % Bandwidth Assigned To High Priority CAN Interference Node

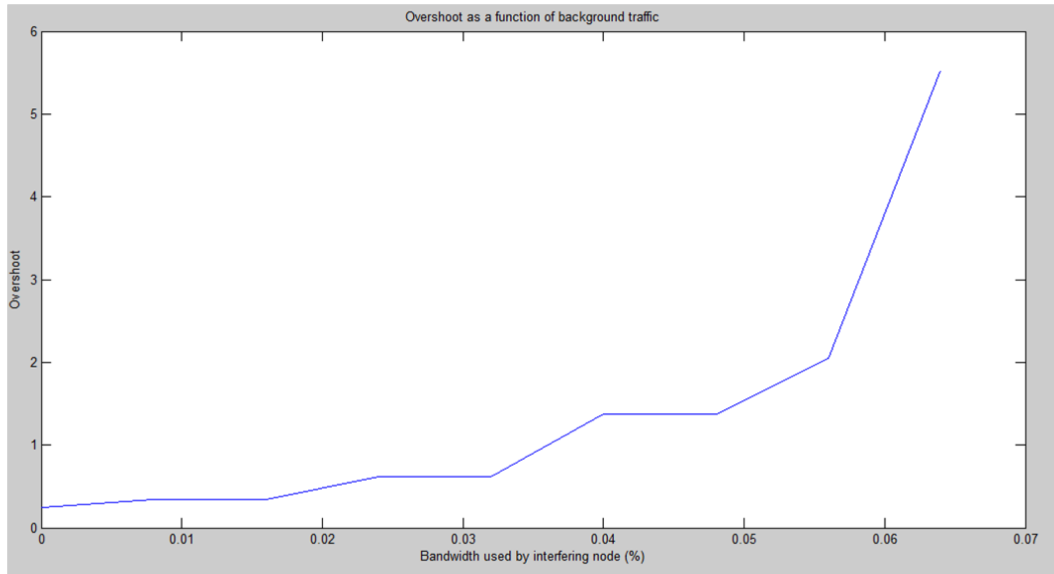

Figure 8.7: Overshoot as A Function of Bandwidth Assigned To High Priority CAN Interference

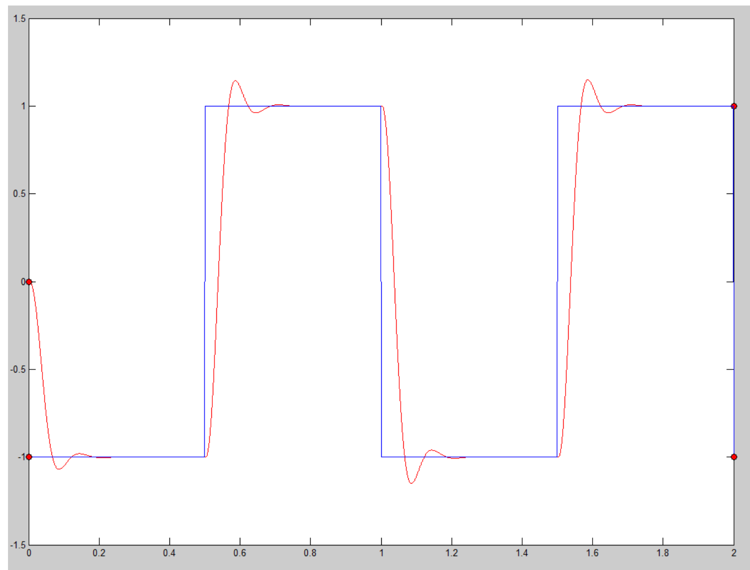

Figure 8.8: Control Performance With 1 ms Gateway Delay

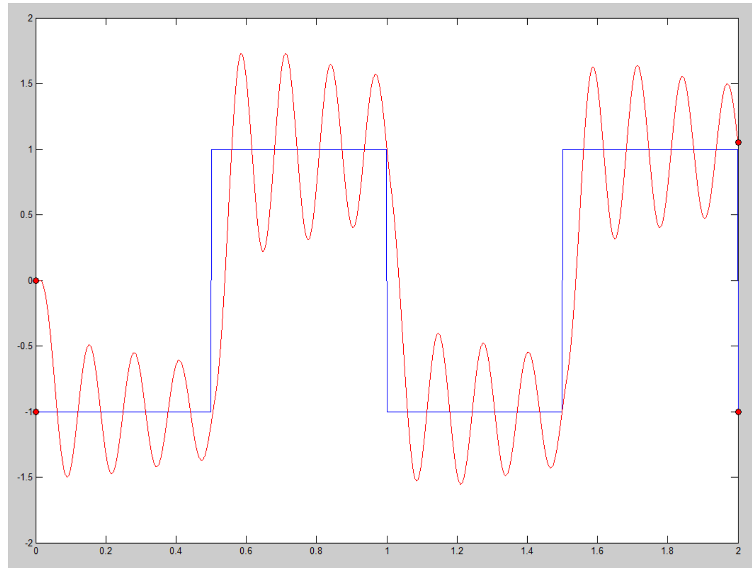

Figure 8.9: Control Performance With 5 ms Gateway Delay

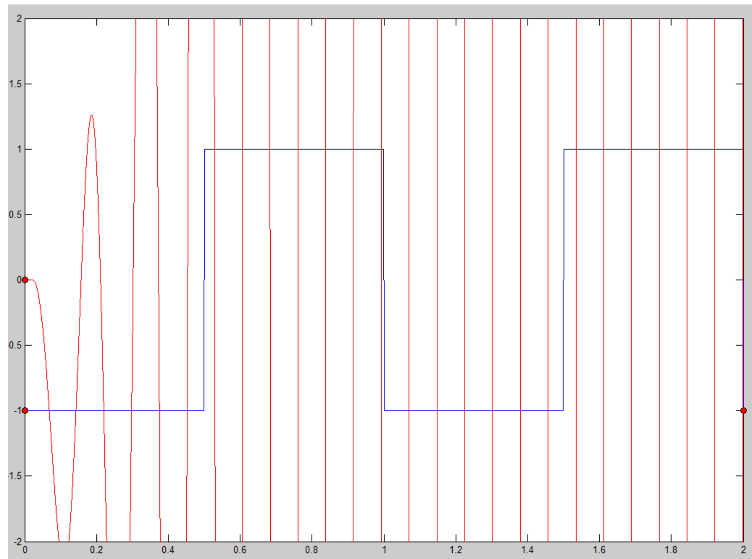

Figure 8.10: Control Performance With 10 ms Gateway Delay

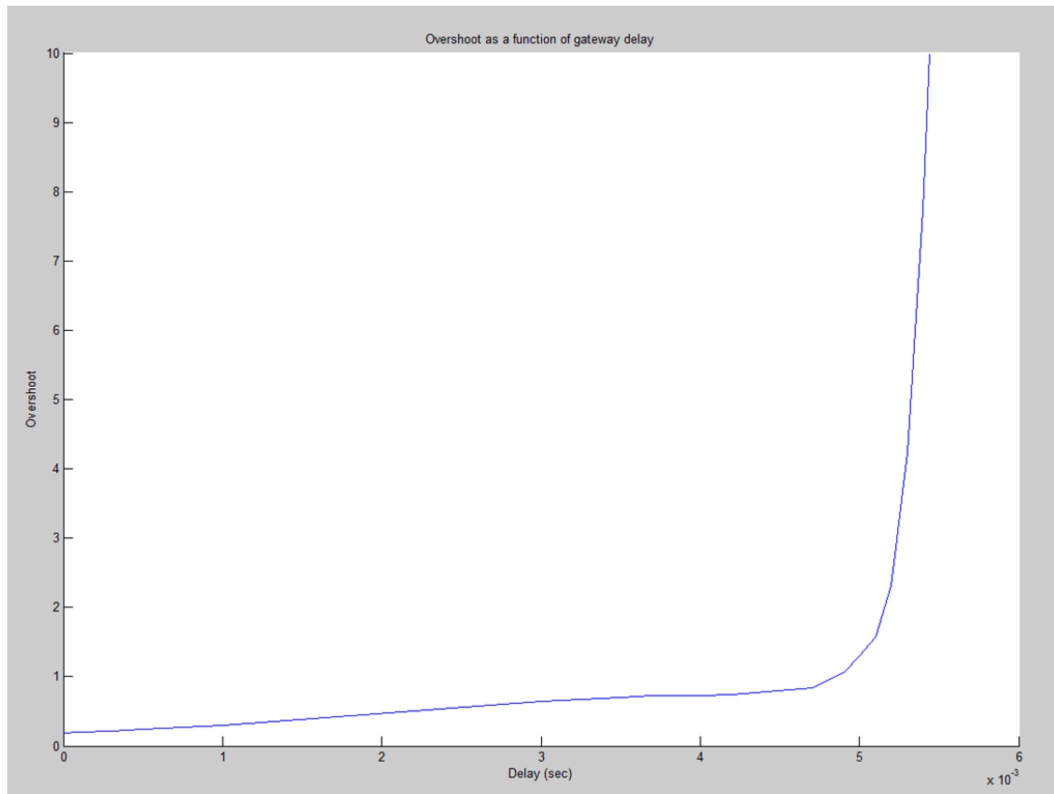

Figure 8.11: Overshoot as a Function of Gateway Delay

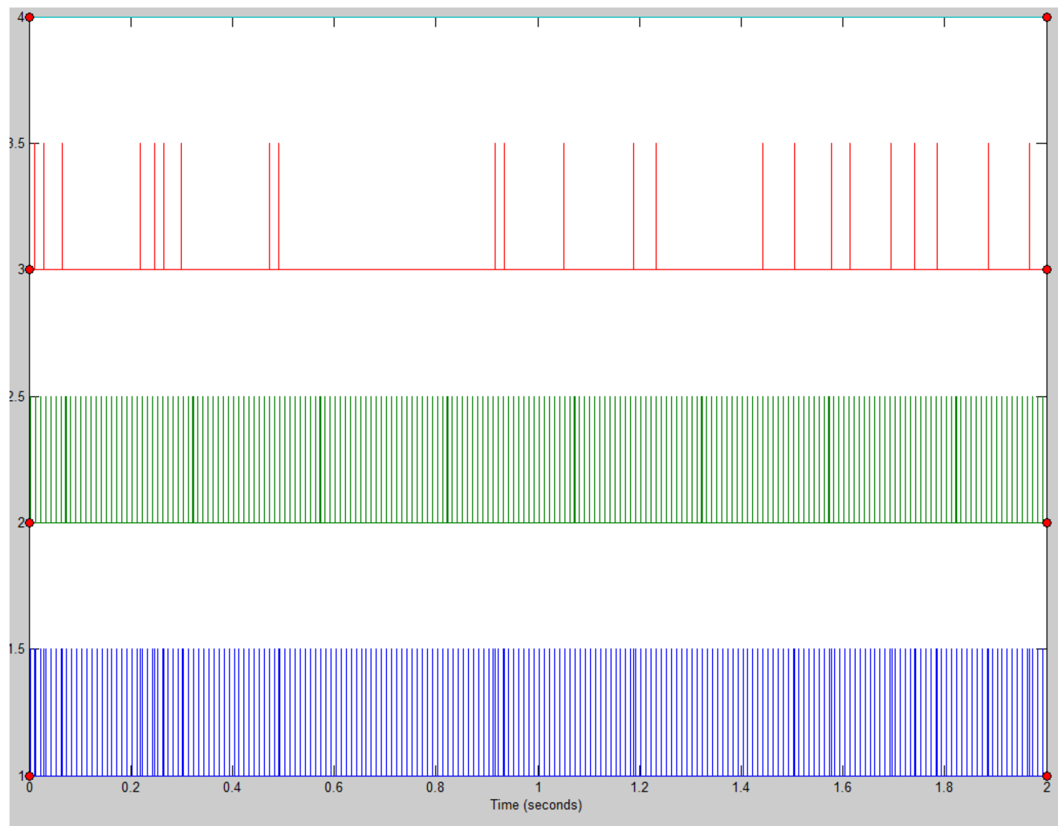

Figure 8.12: Node 3 Executing Sporadic Task On Full Duplex Switched Ethernet

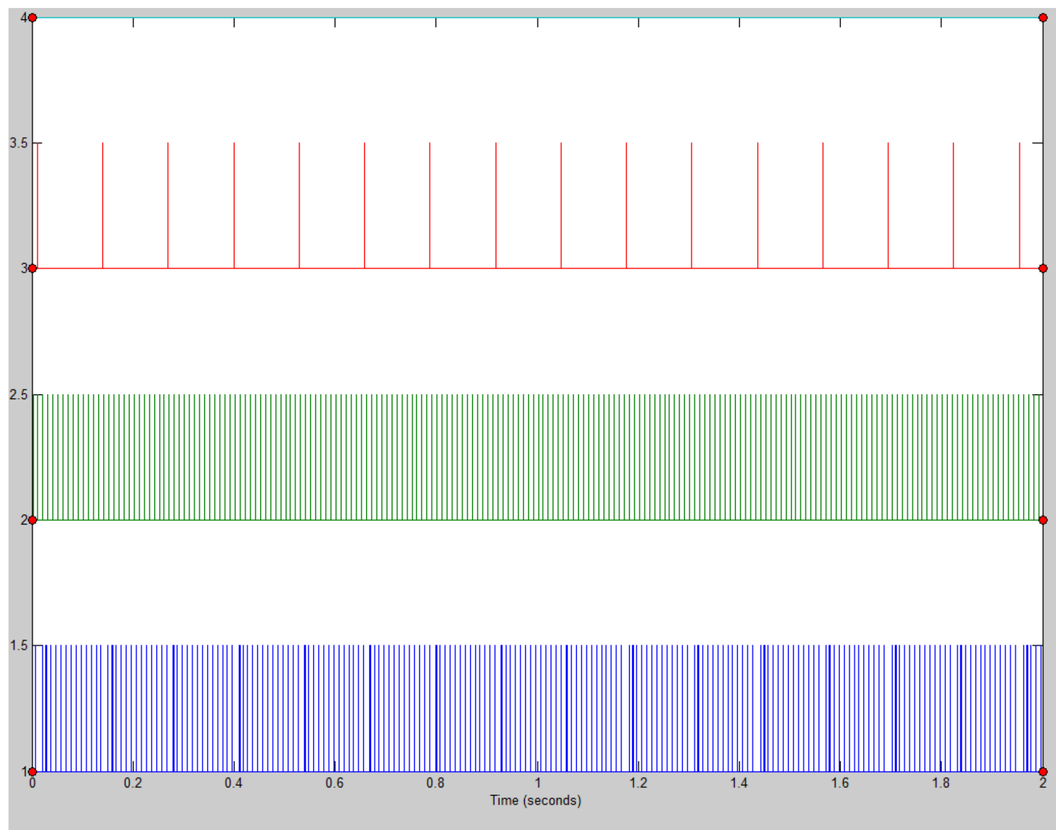

Figure 8.13: Node 3 Executing Sporadic Task on AFDX

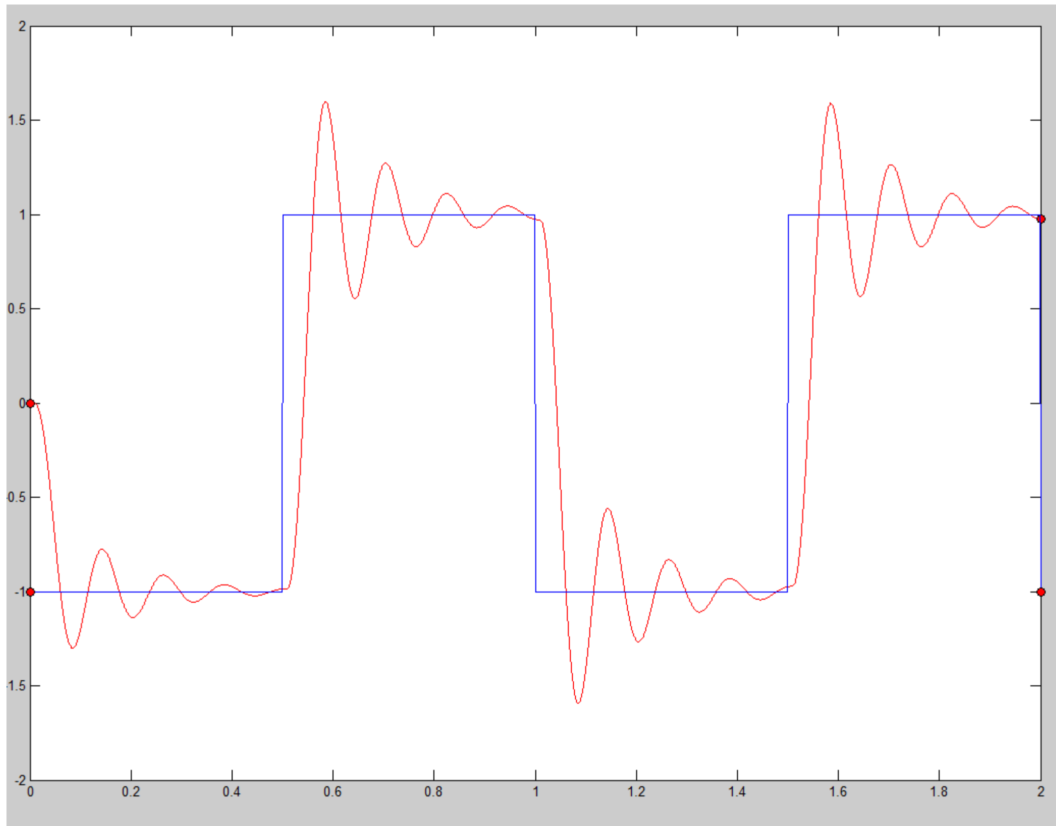

Figure 8.14: Control Performance When Controller BAG is Equal to 1 ms

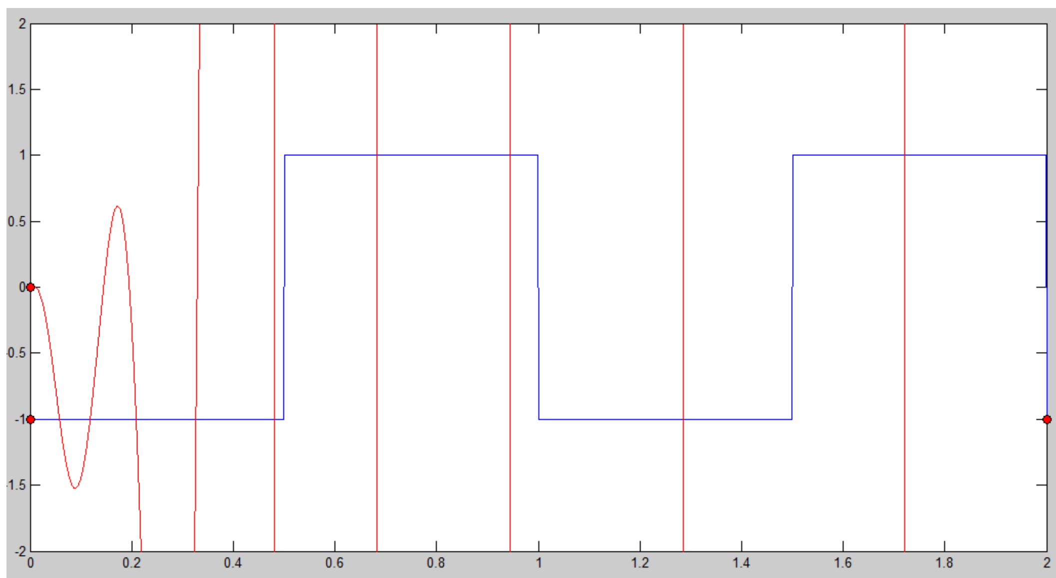

Figure 8.15: Control Performance When Controller BAG is Equal to 10 ms

Chapter 9

Conclusion and Future Work

From the first implementation of the simplified system it can be concluded that TrueTime provides value in terms of determining how much delay and background interference a Hamilton system can tolerate before becoming unstable. Also, the modeling in this part was quite straightforward and no major obstacles were encountered. Since AFDX was approximated with a switched Ethernet for this part, no extensions to TrueTime were necessary in order to provide a proof of concept.

For the second part were the AFDX traffic control was implemented, some effort was required to modify the code from the first implementation in order for it to work as intended. Only the key properties of AFDX were taken into account, to simulate a good enough abstraction of it. To implement AFDX fully, support for fragmentation and defragmentation, freshness indicators at the ports as well as the redundant networks needs to be added. Although the implementation was limited, the concept of traffic control was achieved and produced satisfactory results.

In conclusion, it appears to be feasible to apply TrueTime to Hamilton systems since, except for some adjustments to the AFDX protocol, the tool can be used as it is today. The impression is that TrueTime can be used to drive requirements and provide value in terms of assessing what reasonable limits for requirements could be already at a design stage, which is non-trivial to do and as of today, Hamilton has no similar tool which provides this functionality. A next step would be to further validate this work by applying TrueTime and the AFDX implementation to a real system and evaluate the outcome.

Bibliography

[1] *More Open Electrical Technologies (MOET)*, Final Report, Dec. 2009

[2] Johann Bals, Gerhard Hofer, Andreas Pfeiffer, Christian Schallert, *Object-Oriented Inverse Modelling of Multi-Domain Aircraft Equipment Systems and Assessment with Modelica*, pp. 377-384, Proceedings of the 3rd International Modelica Conference, Linkping, November 3-4, 2003,

[3] T. Wongpiromsarn, Ufuk Topcu, and R. Murray, *Formal synthesis of embedded control software for vehicle management systems*, AIAA Infotech Aerospace, 2011.

[4] "Hamilton Sundstrand History" [Online] Available at: http://www.hamiltonsundstrand.com/About+Us/Historical+Ti [Accessed June 13 2012]

[5] Moir, I., Seabridge, A., "Aircraft Systems" *Mechanical, electrical, and avionics subsystems integration*, pp. 139-153, Second Edition, Professional Engineering Publishing, UK 2001

[6] Holzmann, G., "Design and Validation of Computer Protocols" pp. 162-164, February 2007

[7] Amato,C., Bonet, B., Zilberstein, S., "Finite-State Controllers Based on Mealy Machines for Centralized and Decentralized POMDPs", pp 1-2, July 2010

[8] The Mathworks "Stateflow and Stateflow Coder" *For Complex Logic and State Diagram Modeling*, Version 5, pp. 2-6

[9] Murata, T., "Petri Nets; Properties, analysis and applications" pp. -, April 1989.

[10] Graph Theory Glossary [Online] Available at: http://www.utm.edu/departments/math/graph/glossary.html#d [Accessed March 12 2012]

[11] Johnsson, C., "A graphical language for batch control", pp 21-65, Ph.D dissertation, ISRN LUTFD2/TFRT–1051–SE, Department of Automatic Control, Lund University, Sweden, March 1999.

[12] Bobbio, A., "System modelling with Petri nets", Istituto Elettrotecnico Nazionale Galileo Ferraris, Italy, 1991

[13] Niu, J., Zou, J., Ren, A., "OOPN:AN OBJECT-ORIENTED PETRI NETS AND ITS INTEGRATED DEVELOPMENT ENVIRONMENT", 2010

[14] Arzen, K-E., Johnsson, C., "Grafchart and Grafcet" *A COMPARISON BETWEEN TWO GRAPHICAL LANGUAGES AIMED FOR SEQUENTIAL CONTROL APPLICATIONS*,

[15] Arzen, K-E., "JGrafchart user manual" , ver. 1.5, 2004

[16] Olsson, R., "Batch Control and Diagnosis", pp 50-51 , Ph.D dissertation, ISRN LUTFD2/TFRT–1073–SE, Department of Automatic Control, Lund University, Sweden, June 2005.

[17] Arzen, K-E., Johnsson, C., "Grafchart and its relation to Grafcet and Petri nets", 1998

[18] Cervin, A., Henriksson, D., Lincoln, B., Eker, J., Arzen, K-E., "How Does Control Timing Affect Performance?" *Analysis and Simulation of Timing Using Jitterbug and TrueTime*, pp. -, June 2003.

[19] Dhotre, I.A., Bagad, V.S., "Data Communication and Networking" ,pp. 287 , First Edition, Technical Publications Pune, August 2005

[20] The CAN protocol [Online] http://www.kvaser.com/en/about-can/the-can-protocol.html [Accessed: May 9 2012]

[21] Controller Area Network (CAN) [Online] http://www.ece.cmu.edu/ ece649/lectures/11_can.pdf [Accessed: May 9 2012]

[22] CAN - A Brief Tutorial [Online] http://www.computer-solutions.co.uk/info/Embedded_tutorials/can_tutorial.htm [Accessed: May 9 2012]

[23] General Electric FANUC, "AFDX/ARINC 664 Protocol" *Tutorial*

[24] Techsat, "AFDX/ARINC 664" *A Tutorial*, August 2008