Component-Based Software for the Avionics Domain

A dissertation submitted to the University of Manchester for the degree of Master of Science in the Faculty of Engineering and Physical Sciences

2010

By
Gurdeep Singh Kang

School of Computer Science
Table of Contents

Abstract

Declaration

Copyright

Acknowledgements

Chapter 1 Introduction .................................................................................................................. 10

Chapter 2 Aircraft Fuel Management Systems ......................................................................... 14

          2.1 Airbus A350 XWB Airliner’s Architecture ................................................................. 14
          2.2 Fuel System Components .............................................................................................. 16
          2.3 Fuel Management System Operating Modes ............................................................... 18

Chapter 3 Component-Based Software Engineering .................................................................. 21

          3.1 The Importance of Component-Based Software Engineering ................................. 21

          3.1.1 Software Components .............................................................................................. 21

          3.2 Component Models ....................................................................................................... 22

          3.2.1 Problems with Existing Software Component Models ........................................... 23

          3.2.2 Exogenous Connectors Component Model ............................................................... 24

          3.2.3 Justification for Using Exogenous Connectors Component Model ....................... 24

          3.3 Exogenous Connectors Component Model Tool ......................................................... 25

          3.3.1 Exogenous Connectors Component Model Structure ............................................. 25

          3.3.2 The Exogenous Connectors Component Model Interpreter ..................................... 30

Chapter 4 Fuel Management System Requirements ................................................................. 31

          4.1 Airbus Requirements Documentation ............................................................................ 31

          4.2 Requirements Analysis .................................................................................................. 33

          4.3 Setting the Scope of the Project ................................................................................... 34

          4.4 Functional Requirements Ordering ............................................................................. 34

          4.5 Specific Tasks to Perform ............................................................................................. 35

Chapter 5 Fuel Management System Design ............................................................................ 38

          5.1 Choosing a Suitable Diagram Technique ....................................................................... 38

          5.2 Designing the Fuel Operations ..................................................................................... 39

          5.3 The Need for a New Connector ..................................................................................... 48

          5.4 Mapping the Variables in the State Charts to the Design ............................................. 49
## Chapter 6: Fuel Management System Implementation

### 6.1 Technology Choices

- 6.1.1 Modelling Language Choice
- 6.1.2 Programming Language Choice

### 6.2 Implementation Discussion

- 6.2.1 Implementing the Fuel Management System Using the Tool
- 6.2.2 Writing the Code for the 'Evaluate_Conditions' Method
- 6.2.3 Limiting the Outputs from the 'Evaluate_Conditions' Method
- 6.2.4 Restructuring Certain Parts of the System
- 6.2.5 Implementing Composite Methods

## Chapter 7: Fuel Management System Testing

### 7.1 Testing Strategy

### 7.2 Test Plan

### 7.3 How Testing was Performed

### 7.4 Simulator Used for Testing

### 7.5 Testing the Fuel Management System

- 7.5.1 Unit Testing
- 7.5.2 Integration Testing
- 7.5.3 System Testing
- 7.5.4 Regression Testing

## Chapter 8: Implementation Discussion

### 8.1 Limitation of the Selector Connector Code in the Interpreter

### 8.2 Improving the Existing Selector Connector Code

## Chapter 9: Conclusion

### 9.1 Achievements

### 9.2 Limitations of the Component Model

### 9.3 Challenges

### 9.4 Future Work

### 9.5 Final Remarks

## References

## Appendices

### Appendix A: State Charts from the Requirements Documentation

### Appendix B: Hierarchical Tree Structure Designs for all Fuel Operations

### Appendix C: Fuel Management System Implementation Screenshots in GME

### Appendix D: Test Plan Showing all the Tests Performed for the Manual Refuel Operation
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The architecture of the Airbus A350 XWB, also showing the internal structure</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Airbus A340-600 fuel tank configuration</td>
<td>15</td>
</tr>
<tr>
<td>2.3</td>
<td>The schematics of an aircraft's engine</td>
<td>17</td>
</tr>
<tr>
<td>2.4</td>
<td>Aircraft fuel tank configuration with operating modes shown</td>
<td>19</td>
</tr>
<tr>
<td>3.1</td>
<td>A software component showing the two different types of interfaces</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>Direct method calls to other components</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>Connecting components using exogenous connectors</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>The GME toolkit which is used to create implementations of systems using the exogenous connectors' component model</td>
<td>26</td>
</tr>
<tr>
<td>3.5</td>
<td>Class diagram showing the Exogenous Connectors Component Model</td>
<td>27</td>
</tr>
<tr>
<td>3.6</td>
<td>A diagrammatic representation of an Atomic Component and a Composite Component</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>Behaviour of (a) sequencer, (b) pipe and (c) selector exogenous connectors</td>
<td>30</td>
</tr>
<tr>
<td>4.1</td>
<td>Textual functional requirement for the ‘Manual Refuel’ operation</td>
<td>31</td>
</tr>
<tr>
<td>4.2</td>
<td>Detailed functional requirement for the ‘Abort’ sub-mode</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>State chart for the ‘Manual Refuel’ operation</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Sub-states of the 'In-Progress' sub-state</td>
<td>33</td>
</tr>
<tr>
<td>4.5</td>
<td>Hierarchical representation of the ‘Manual Refuel’ state</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>An example of a system developed using components shown in a hierarchical tree structure</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>State chart for the ‘Manual Refuel’ operation based on the actual Airbus diagrammatic state chart requirements</td>
<td>40</td>
</tr>
<tr>
<td>5.3</td>
<td>Hierarchical tree structure design for the 'Manual Refuel' operation</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>Showing the (a) incorrect tree structure and the (b) improved correct tree structure for the 'Manual Refuel Op' composite component</td>
<td>44</td>
</tr>
<tr>
<td>5.5</td>
<td>Showing the (a) incorrect tree structure and the (b) improved correct tree structure for the 'Manual Refuel Execution' composite component</td>
<td>46</td>
</tr>
<tr>
<td>5.6</td>
<td>A decision node being used in the state charts</td>
<td>47</td>
</tr>
<tr>
<td>5.7</td>
<td>Showing the sub-states that can be executed concurrently for the ‘In_Progress’ state chart</td>
<td>48</td>
</tr>
<tr>
<td>5.8</td>
<td>Showing the code inside the ‘Evaluate_Conditions’ method from Figure 5.2</td>
<td>50</td>
</tr>
<tr>
<td>5.9</td>
<td>Showing how variables were defined by listing the variables that belonged to a component under the name heading for that component</td>
<td>51</td>
</tr>
<tr>
<td>5.10</td>
<td>Manual Refuel Left Wing Tank (MR_TK_LWT) state chart from the diagrammatic state chart requirements</td>
<td>52</td>
</tr>
<tr>
<td>5.11</td>
<td>Hierarchical tree structure design based on the Manual Refuel Left Wing Tank (MR_TK_LWT) state chart in Figure 5.10</td>
<td>52</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>5.12</td>
<td>Showing how variables required at the lowest level are passed down from the topmost components as input parameters</td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Showing the implementation of the 'Manual Refuel Process' composite component in the GME toolkit</td>
<td></td>
</tr>
<tr>
<td>6.2</td>
<td>Showing the simulator GUI which appears when the interpreter button in GME is clicked</td>
<td></td>
</tr>
<tr>
<td>6.3</td>
<td>The interpreter code which processes the GME implementation of the system</td>
<td></td>
</tr>
<tr>
<td>6.4</td>
<td>The 'Evaluate_Conditions' method for the 'Manual Refuel' state chart</td>
<td></td>
</tr>
<tr>
<td>6.5</td>
<td>C++ code written for the 'Evaluate_Conditions' method in Figure 6.4</td>
<td></td>
</tr>
<tr>
<td>6.6</td>
<td>Showing the 'Manual Refuel Process' composite component implementation in GME</td>
<td></td>
</tr>
<tr>
<td>6.7</td>
<td>Showing the sub-states and 'Evaluate_Conditions' method inside the 'Manual Refuel' state chart</td>
<td></td>
</tr>
<tr>
<td>6.8</td>
<td>Showing how one variable determines which sub-component to execute in the 'Manual Refuel Process' composite component</td>
<td></td>
</tr>
<tr>
<td>6.9</td>
<td>Showing the sub-components of the 'MR In_Progress' composite component</td>
<td></td>
</tr>
<tr>
<td>6.10</td>
<td>Showing the contents of the 'MR_Surge_Protection' sub-component</td>
<td></td>
</tr>
<tr>
<td>6.11</td>
<td>Showing the contents of the 'SP_Process' composite component</td>
<td></td>
</tr>
<tr>
<td>6.12</td>
<td>Showing the C++ code for the 'SP_Eval_Status' atomic component</td>
<td></td>
</tr>
<tr>
<td>6.13</td>
<td>Showing composite methods for composite components</td>
<td></td>
</tr>
<tr>
<td>6.14</td>
<td>Showing how parameters in GME are ordered in the composite method for the 'MR In_Progress' composite component so they can be supplied in that order to the sub-components</td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>Showing how testing was performed in a bottom-up manner [15]</td>
<td></td>
</tr>
<tr>
<td>7.2</td>
<td>GUI of the simulator used to test each aspect of the fuel management system</td>
<td></td>
</tr>
<tr>
<td>7.3</td>
<td>Using the simulator to perform the integration test 'I2' from Table 7.3</td>
<td></td>
</tr>
<tr>
<td>7.4</td>
<td>GME implementation of the 'MR In_Progress' composite component</td>
<td></td>
</tr>
<tr>
<td>8.1</td>
<td>State chart for the 'Manual Refuel' operation</td>
<td></td>
</tr>
<tr>
<td>8.2</td>
<td>The C++ selector connector code for the interpreter</td>
<td></td>
</tr>
<tr>
<td>8.3</td>
<td>Improved C++ selector connector code</td>
<td></td>
</tr>
</tbody>
</table>
List of Tables

4.1 Table showing the implementation order of all the ground fuel operations .......... 35
4.2 Table showing all the tasks performed for the entire project .................................. 36

7.1 Test plan structure ........................................................................................................ 71
7.2 Examples of 2 unit tests performed on the 'MR_LWT_Idle' and 'MR_LWT_Active' atomic components ......................................................................................................................... 73
7.3 Examples of 2 integration tests performed on the 'MR_TK_LWT' and 'MR_In_Progress' composite components ................................................................................................................................. 74
7.4 An example of a system test performed on the 'Manual Refuel Op' composite component 76
Abstract

The complexity of avionics software has increased over the years since modern airliners are now taking on more larger and complex tasks without human intervention. Due to this inherent complexity, the time scale and cost of development is relatively huge. One methodology that has been proposed to tackle this common problem is component-based software development, which is becoming popular in the avionics domain. This methodology focuses on reusing software components from previously built software systems to develop new software systems. As a result of reusing such components, the overall production time, risks and costs are significantly reduced.

The aim of this research project is to design and implement a fuel management system for the upcoming Airbus A350 XWB airliner which will be responsible for monitoring and controlling the fuel levels. This will be achieved by using a component-based methodology based on exogenous connectors to prove that reliable avionics software can be developed by reusing software components.
Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

I. Copyright in text of this dissertation rests with the Author. Copies (by any process) either in full, or of extracts, may be made only in accordance with instructions given by the Author. Details may be obtained from the appropriate Graduate Office. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the Author.

II. The ownership of any intellectual property rights which may be described in this thesis is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

III. Further information on the conditions under which disclosures and exploitation may take place is available from the Head of the School of Computer Science.
I would firstly like to thank my supervisor Dr Kung-Kiu Lau for his constant support and inspiration throughout the course and for allowing me to do this project. I would also like to thank Cuong M. Tran for his guidance and support throughout the project.
The complexity of avionics (aviation electronics) software for modern airliners is ever increasing with today's software taking on larger, more complex aviation tasks and playing a greater role in aviation safety [1]. Typical software in the latest jets have over 5 million lines of code, compared to about 1 million lines of code in older generation aircrafts. The complexity of avionics software is further increased since it is real time software whose behaviour must be modelled as timed system states [2]. Furthermore, the software must interact with the hardware systems, commonly known as fly-by-wire control systems in modern aircrafts, in order to allow manoeuvring. As a result, this adds to the complexity of the software. Since avionics software is safety-critical, rigorous testing methods must be applied to ensure that such software meets the strict reliability and performance requirements to avoid aircraft disasters which could result in fatalities [3, 4]. Thus, the complex and safety-critical nature of avionics software increases the costs of development since more time and effort must be invested in its development.

A modern commercial aircraft, such as the Airbus A380, consists of around 50 systems and 100 computers [3]. This is a clear indication of the importance software plays in today's aircrafts. It would be fair to say that a modern aircraft simply would not be able to fly without all the computer systems that provide the crew with vital data in order to control the aircraft.

Avionics software has traditionally been developed entirely from scratch. Due to the safety-critical nature of avionics software, such software must meet certain safety standards before the software can be put into operation, which is approved by the FAA (Federal Aviation Administration). This standard is known as DO-178B, which is a de-facto standard for the development and management of safety-critical software used in aviation [4]. Developing avionics software requires strict development procedures to be lawfully followed and therefore the steps involved in the development are much more detailed and vigorous. However, the common problem with safety-critical software is that they are extremely time consuming and expensive to build. Since software in the avionics domain is of a safety-critical nature, it must be verified rigorously before it can be used in practice to avoid catastrophic failures. The competitive nature of the avionics domain is also impacted by the prolonged software development period for commercial aircrafts and in today's dynamic business worlds, competition is fierce and competitors are eager to unveil their latest aircrafts before their own competitors. Such competition is often seen between the big players of the avionics domain: Airbus and Boeing.

The lengthy period and high expense associated with safety-critical software development has been an issue for over a decade and it is also a common problem with software developed in other domains. This is especially a problem in a dynamic business world where customer requirements are changing rapidly. Although many software development processes have been proposed to tackle such problems, such as Rapid Application Development, these development processes are
inadequate for avionics software due to the strict standards that must be adhered to because of their safety-critical nature [5]. However, one software development technique that has been proposed and has been gaining popularity in the avionics domain is component-based software engineering (CBSE) [5]. The notion behind CBSE is simple; develop software by reusing a set of predefined software components. Ian Sommerville [5] defines CBSE as:

“The process of defining, implementing and integrating or composing loosely coupled independent components into systems”. [5]

The emphasis of CBSE is on reusing existing software components to build new software systems. Szyperski (Szyperski, 2002) [5] defines software components as follows:

“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties”. [5]

The key emphasis of software components is that they are completely independent and are defined by their interfaces. A component can be considered a service provider in which it offers its services to other components [5]. These services are known as the methods of the component that are invoked by other components when a particular task is required to be executed. A given component expresses its interfaces in terms of parameterised methods only; the internal state of the component is never exposed [5]. This means that the implementation details are hidden from other components who can only communicate via the interfaces (methods) of the component.

It is clear why CBSE is seen as an attractive option to software development since avionics software is becoming much larger and complex in new generation aircrafts and clients are demanding software to be developed more quickly so that they can remain ahead of competitors. The only way to achieve this is to reuse existing software components rather than re-implement them from scratch.

Traditional avionics software did not make use of reusable software components (RSC) because all software had to comply with the DO-178B standard which contained no standard way of reusing software components. In addition to this, the DO-178B made reuse more difficult because it often required expensive recertification efforts [6]. The FAA realised the importance of allowing RSC and granted RSC acceptance as part of their standard certification process, without having to go through the costly and risky process of recertification. As a result of this, developers were able to take advantage of CBSE during the implementation of avionics software, thus reducing overall engineering labour, program risk and cost [6].

To aid the development of complex software, component models are often used. A component model is a framework that allows you to develop complex software systems from small reusable software components. In existing component models, control originates inside the component itself and connectors, which are the connections between the software components, are responsible for passing on control (method calls) to other components [7]. This is often referred to as message passing whereby one component is able to invoke the methods of another component either directly or indirectly. In such component models, computation and control are mixed inside components. Thus, in terms of control, components are not loosely coupled since direct or indirect dependencies exist between the components which make it harder to reuse software components. However, the
Software component model that is based on exogenous connectors differs from existing component models in that these exogenous connectors initiate the method calls of software components [7]. This means that components do not invoke methods in other components, instead this is done by the exogenous connectors. As a result, coupling is maximised in terms of data and control in components [7], thus making it easier to reuse software components to build other aircraft systems. This particular exogenous connector component model will be used to implement the proposed system.

The benefits of CBSE will be clearly demonstrated to show how avionics software can effectively be developed using reusable software components by applying the given software component model. The benefits of why this particular software component model is more desirable than other component models currently on the market will be discussed and also how exogenous connectors allow you to separate computation from control in software components, thus making software components more loosely coupled and truly independent.

The aim of the proposed research is to investigate how CBSE can be applied effectively to design and implement a fuel management system for the Airbus A350 XWB (Extra Wide Body) commercial airliner which will manage the fuel operations of the aircraft such as automatic refuel, defuel, and ground transfer and so on.

The primary objective of the proposed research is to implement the system using predefined reusable software components so that a functional system is the result. The research will involve first analysing Airbus data1 for their upcoming A350 (XWB) airliner’s fuel management system and then applying a software component model that is based on exogenous connectors [7] to implement the fuel system. This will offer the benefit of encapsulating control and separating this from computation of software components, thus making software components truly independent.

The structure of the remainder of this report can briefly be described as follows:

Chapter 2 – Aircraft Fuel Management Systems: Discusses the background research conducted into the field of avionics, primarily fuel management systems.

Chapter 3 – Component-Based Software Engineering: Discusses the background research conducted in software reuse which is followed by a discussion of a software tool that was analysed thoroughly in order to allow the design and implementation of the fuel management system.

Chapter 4 – Fuel Management System Requirements: Discusses the requirements supplied by Airbus and how the scope of the project was set.

Chapter 5 – Fuel Management System Design: Discusses the process of designing the fuel management system.

Chapter 6 – Fuel Management System Implementation: Discusses the important and interesting aspects of the fuel management system implementation.

---

1 This data has been made available to us via a non-disclosure agreement with Airbus
Chapter 7 - Fuel Management System Testing: Discusses the process of how the fuel management system was tested.

Chapter 8 - Implementation Discussion: Discusses the relevant issues that arose during implementation and the additional work carried out after the fuel management system was fully implemented.

Chapter 9 - Conclusion: Discusses the evaluation of the project as a whole in terms of achievements, limitations, challenges faced and future work.
A fuel management system in a commercial airliner is used to maintain, control and monitor fuel levels of the aircraft \[9, 10\]. The primary purpose of these systems is to provide reliable supply of fuel to the engines of the aircraft during all phases of flight which include changes in altitude (aircraft height changes), violent manoeuvres and sudden acceleration and deceleration \[11, 12\]. The fuel management system is considered to be one of the most essential systems on any form of aircraft, since without it the aircraft would be unable to sustain flight.

### 2.1 Airbus A350 XWB Airliner's Architecture

The focus of this dissertation is on the upcoming Airbus A350 XWB airliner’s fuel management system. The Airbus A350 XWB is a mid-size airliner due to be released around 2013 whose architecture is shown in Figure 2.1.

*Figure 2.1: The architecture of the Airbus A350 XWB, also showing the internal structure* [8]
Since this dissertation’s focus is on the fuel management system, the attention will only be paid to the areas of where fuel is stored in the airliner. These locations are referred to as fuel tanks and modern aircrafts often have several fuel tanks installed. Figure 2.1 highlights the fuel tank locations in red numerical values and arrows. Number 39 refers to the left and right fuel tanks which are located in the left and right wings of the airliner. These are the fuel tanks that transfer fuel to the primary collector cells (Figure 2.2) which in turn feed fuel to the engines. Number 29 refers to the central fuel tank which is an additional fuel tank that stores fuel which is transferred using pressurisation to the left and right fuel tanks inside the wings once the fuel in the wing tanks falls below a certain point.

Figure 2.2 shows an aerial view of the fuel tank configuration. This is a more detailed view of the fuel tanks for all Airbus aircraft series A340-600 [9].

![Airbus A340-600 fuel tank configuration](image)

**Figure 2.2:** Airbus A340-600 fuel tank configuration [9]

Figure 2.2 clearly demonstrates that several fuel tanks exist in the aircraft. These fuel tanks include [9]:

- Four collector cells or tanks – one for each engine. In the case of the Airbus A350 XWB, there would only be two collector tanks since it only consists of two engines. These collector tanks are the main tanks that feed fuel to the engines.

- A centre tank – used to store additional fuel.

- Three tanks in the left wing: outer, inner 1 and inner 2 tanks – used to store fuel that is fed to the collector tanks which in turn is fed to the engines.
- Three tanks in the right wing: outer, inner 3 and inner 4 tanks.
- Trim tank located in the tail of the aircraft. This tank is only present in certain aircrafts and is not compulsory. It is often used to control the aircraft centre of gravity.

As aircrafts have advanced over the years, so have their fuel management systems. The arrival of jet turbine powered aircrafts meant that the engines for these aircrafts were much fuel thirstier than their predecessors (piston-engine aircrafts) [9]. As the complexity of such aircrafts increased, so did the complexity of their fuel management systems. Due to the high consumption of fuel for today's modern airliners, the need for extra fuel tanks became crucial. In addition to fuel tanks being installed in the left and right wings of the aircraft, further fuel tanks were necessary in the body of the aircraft which would store additional fuel (Figure 2.2). As a result, these additional fuel tanks added to the complexity of the fuel system since they are installed in the body of the aircraft which meant that additional fuel system components were necessary in order to transfer the fuel from these additional tanks to the primary wing fuel tanks [9].

### 2.2 Fuel System Components

The fuel system components refer to the components that make up the entire hardware part of the fuel management system.

In order to ensure that tanks cannot be damaged due to the excessive pressure during fuel transfer, valves are installed to control the pressurisation. A valve (also known as fuel transfer valves) is a device that is used in aircraft fuel systems to allow fuel to flow from one location to another. They are analogous to doors in a house. When doors are opened people can move between rooms however when doors are closed these people are unable to move to another room. In this case the doors represent the valves and the people represent the fuel. The typical Airbus A340-600 airliner series may contain around 36 different fuel valves in the system [9].

Fuel transfer valves play a critical role in fuel management systems, especially since modern airliners now have several fuel tanks. There are many types of fuel transfer valves such as shut-off valves, cross-feed valves, and fuel dump valves [9, 11, 12]. Shut-off valves are responsible for shutting off fuel flow when necessary. This may occur when, for example, fuel is being transferred from the central fuel tank to the left wing fuel tank. Before this process begins the valves in both the central and left wing tanks will open and fuel will be transferred using fuel pumps and pressurisation if necessary. Once the necessary levels of fuel have been transferred, the valves at both ends will be shut off. Cross-feed valves are used to feed fuel from one side of the aircraft to the other while fuel dump valves are used during emergencies to dump excess fuel from the aircraft. Therefore, these multi-valve systems allow the pilots to move fuel around the fuel tanks according to their needs at any given time.

Fuel transfer pumps also play a critical role in fuel management systems. These are responsible for transferring fuel between the aircraft's fuel tanks. Fuel transfer pumps have three main functions
which include transferring fuel from one tank to another, transferring fuel from the tanks to the engines and finally from the engines back to the tanks. [12]

The fuel lines refer to the metal tubing or rubber cables that are used to transmit the fuel between the tanks. They connect all the tanks together so that fuel can be transferred easily when necessary. [12]. Figure 2.3 shows the schematics of an aircraft’s engine. The blue lines connecting the fuel tanks and engines are the fuel lines and the fuel transfer valves, pumps, tanks and fuel gauges are clearly labelled.

![Diagram of aircraft fuel system](image)

Figure 2.3: The schematics of an aircraft’s engine [13]

In the cockpit², pilots use fuel gauging systems to help determine the levels of fuel in the aircraft and this helps them make decisions for when fuel may need to be transferred from the additional tanks to the wing tanks [9]. The fuel gauging systems provide real time data on the levels of fuel consumed by the aircraft and how much fuel remains in each wing’s tank and other fuel tanks. Another important mechanism which is often handled automatically is ensuring that the fuel levels in the left and right wing tanks are equivalent. Obviously if one wing tank contains more fuel than the other this could affect the aircraft’s attitude relative to the horizon, resulting in the wings of the aircraft to become unlevel. In the event that one wing’s fuel tank contains more fuel than the other, valves in both wings are opened to allow fuel to be transferred from one wing tank to the other until both are holding equivalent quantities.

Due to the complex nature of fuel systems, it is critical that the fuel gauging systems are accurate in representing the fuel levels in the tanks. This is achieved by measuring the fuel levels in the tanks using fuel probes or sensors which are placed at various locations in the tanks [9]. Typical fuel tanks consist of around 30 to 40 fuel probes to measure the fuel contents accurately, which have accuracy

---

² The area located at the front of the aircraft where pilots control the aircraft.
in the region of 1-2% [9]. As the fuel levels fall below each fuel probe, this is represented accurately on the fuel gauge systems allowing pilots to determine the amount of fuel remaining, thus enabling them to take decisive action when necessary.

It is important to note that the fuel management system is composed of both hardware and software. The fuel transfer pumps, valves, fuel lines, and fuel tanks refer to the hardware part of the system where as the software part of the system is responsible for communicating with these hardware components to allow operations to take place, such as transferring fuel from the central tank to the left wing tank. In this case the software would instruct the valves in both tanks to open and the fuel pumps and the pressurisation system would be instructed to transfer the fuel between the tanks. Once the desired level of fuel is transferred, the software system would instruct the fuel pumps to stop pumping fuel and the valves in both tanks would be closed. This is a clear indication of how complex fuel management systems have become as they have evolved over the years.

2.3 Fuel Management System Operating Modes

The fuel management system operating modes refers to the different kinds of functions or tasks that are performed by the fuel management system which include the following:

- Fuel pressurisation
- Engine feed
- Fuel transfer
- Fuel Jettison
- Refuel
- Defuel

Before each of these operating modes are briefly discussed, it is important to categorise them into ground or flight operations. Ground operations refer to the operations that take place when the aircraft is on the ground and stationary [10]. Refuel and defuel are both ground operations. Flight operations refer to operations that take place when the aircraft is airborne [10]. Fuel pressurisation, engine feed, fuel transfer and fuel jettison are all flight operations. It is important to note that flight operations cannot occur during ground operations and vice-versa.

Fuel pressurisation refers to forcing fuel under relatively low pressure from certain tanks to others [9]. This may occur when, for example, fuel may need to be transferred from the central fuel tank to one of the tanks in the wings. Pressurisation is necessary in this case since the central fuel tank is located in a different location to the wing fuel tanks and so a low pressure ensures fuel can be transferred effectively.

Engine feed refers to supplying the engines with the fuel. This takes place via the collector fuel tanks in the wings since the engines are located just below the outer wing. In this case no pressurisation is necessary since gravity is sufficient to supply the fuel to the engines [9].
Fuel transfer refers to moving the fuel from the central fuel tanks and main wing tanks to the collector tanks. The collector tanks are the main tanks that feed fuel to the engines [9]. These tanks are small compared to the other tanks as can be seen from Figure 2.2. It is important to note that valves also play an important role here to ensure that these tanks are only filled with fuel from the wing fuel tanks once their levels have fallen below a certain point.

Fuel jettison is the process of dumping fuel from the aircraft. This operation would only ever be executed in emergencies when, for example, an aircraft suffers a malfunction or emergency shortly after take-off. In such an event, fuel would need to be dumped out of the aircraft to reduce the weight of the aircraft rapidly to a level that is acceptable for landing [9]. Another case in which fuel may be dumped is when an engine fails. In this case reducing the weight of the aircraft is critical so that the aircraft can remain airborne until a nearby airport is located and it is acceptable to land.

Refuel is the process of filling up the fuel tanks of the aircraft with fuel. Before refuelling can take place a shut-off test is critical to ensure the aircraft has been shut off correctly since failure to do this could have devastating effects [9, 10].

Defuel is the opposite process of refuel which involves removing fuel from the aircraft’s fuel tanks. It may be necessary to defuel the aircraft for maintenance purposes [9, 10]. Generally, defueling is not carried out as frequently as refuelling. However, when defueling is performed the entire fuel tanks must be completely emptied and purged with air so that the tanks are a safe place to operate in [9].

Figure 2.4 shows another aircraft’s fuel tank configuration similar to Figure 2.2 except this fuel tank configuration shows all the operations discussed above and exactly where they take place in the aircraft.

**Figure 2.4:** Aircraft fuel tank configuration with operating modes shown [9]
There are many other system operating modes for fuel management systems however these will not be discussed since they are often specific to certain aircrafts. The above operations take place in all forms of aircraft and in particular the Airbus A350 XWB which is where the focus of this dissertation lies.
Chapter 3

Component-Based Software Engineering

The complexity of avionics software is ever increasing with aircrafts nowadays incorporating more sophisticated features that require less input from pilots during flight. The major concern associated with today’s complex avionics software is the prolonged time period required in its implementation since such software has traditionally been developed from scratch. Since avionics software is of a safety-critical nature, all software must meet the DO-178B standard before the software can be put into operation. Since this standard traditionally did not specify any means of reusing existing software components from previous software systems, this meant it was always compulsory to develop software from scratch. Reusing software components was difficult because it often required expensive recertification efforts from the authorities and thus reuse was always avoided [6]. Once the Federal Aviation Administration (FAA) recognised the importance of reuse, standards for reusing software components were eventually added to the existing DO-178B standard thus increasing the popularity of software reuse in the avionics domain. As a result of reuse the time, costs and risks associated with development are significantly reduced [6].

3.1 The Importance of Component-Based Software Engineering

Component-based software engineering (CBSE) has been around since the late 1990s [5] however its popularity has just began to increase recently, especially within the avionics domain. CBSE is a process of developing software by reusing software components from previously defined systems to develop new systems. These components are connected together to develop new systems and as a result a composite component is formed. These composite components can then be combined with other composite components to form even larger composite components.

3.1.1 Software Components

A software component is a unit of software that has a well defined purpose and is loosely coupled\(^3\) and highly cohesive\(^4\). A given software component should be as independent as possible so that it can easily be reused.

---

\(3\) Coupling describes the interactions between components. The looser the coupling, the less interaction between the components [14].

\(4\) Cohesion describes the interactions within components. The more cohesive a component, the more related the internal parts of the component to each other and to its whole purpose [14].
To get a clearer picture of what a component does, it is useful to consider a component as a service provider. Reusable software components have two characteristics which are as follows [5]:

1. The component is an independent executable entity. This means it should be possible to deploy this component without having to use other components.

2. The services offered by the component are made available through an interface and all the communication takes place via the interfaces. The interfaces refer to the methods of the component. This means that the internal details of a component are always hidden from other components [14].

Software components commonly have two different types of interfaces [5]:

1. A **provides** interface specifies the services that are provided by the component. This interface defines the methods that can be called by another component.

2. A **requires** interface specifies what services must be provided by other software components. This interface specifies the methods that this component requires in order to function.

Figure 3.1 shows an example of a software component along with the two interfaces. The provides interface is represented by a circle at the end of a line where as the requires interface is represented by a semi-circle at the end of a line. When components are connected together they fit together as a ball and socket [5].

![Diagram of Software Component Interfaces](image)

**Figure 3.1:** A software component showing the two different types of interfaces [5]

### 3.2 Component Models

Ian Sommerville describes a component model as follows:

“A component model is a definition of standards for component implementation, documentation and deployment. These standards are for component developers to ensure that components can interoperate.” [5]

A component model is simply a framework that is used to develop complex software systems, such as avionics software, from small reusable software components. Many component models exist on
the market; however the most popular component models are the CORBA component model from the OMG, Sun's Enterprise Java Beans model and Microsoft's COM+ model [5].

The component model to be used in the design and implementation of the fuel management system will be different to those listed above. This software component model is based on exogenous connectors which is discussed in more detail in the following sections.

### 3.2.1 Problems with Existing Software Component Models

Existing software component models, such as Sun's Enterprise Java Beans model, all have a common disadvantage. The software components that are defined with these models mix computation with control [7]. In this case computation refers to any processing code in a component, such as some code that calculates the average of a set of numbers input. On the other hand, the control refers to the method calls that a given component invokes. The emphasis here is on the control because when one component invokes a method of another component, control moves to the other component and this means that a dependency exists between the two components. As a result, components are not loosely coupled. Therefore components are not truly independent and this hinders the ability to reuse software components easily to develop other software systems.

When a set of individual components are defined and combined, the lines that connect all these components are referred to as connectors. In existing component models the components encapsulate computation where as the connectors encapsulate the communication between components. In these existing component models, control is initiated inside the components themselves and the connectors between the components handle the control flow [7]. Thus, these connectors handle method calls and their return values. The main drawback here is that method calls are directly mixed inside the rest of the code of the components. This means that the given component and the component whose method is invoked are tightly coupled which makes reusing these software components difficult. Figure 3.2 shows an example of direct method calls being made from inside the components. The method call a() to component B has been hard coded inside component A. This is what we mean by control being mixed with computation of the component. This direct dependency between the two components is what makes it difficult to reuse software components easily without having to make significant changes to its structure.

![Figure 3.2: Direct method calls to other components [7]](image)
Therefore a mechanism is required whereby the connector’s code can be isolated into separate entities, thus making the components more loosely coupled. By achieving this, control (method calls) is now encapsulated in separate entities while computation remains inside the components. This is exactly what the proposed software component model based on exogenous connectors achieves. This is discussed further in the next section.

3.2.2 Exogenous Connectors Component Model

The component model based on exogenous connectors is a unique component model since it uses exogenous connectors to connect all software components. These connectors are different to connectors used in existing component models because this is where all the control (method calls) is initiated and coordinated [7]. This means that components do not invoke methods of other components directly inside themselves; instead this is all performed inside the connectors which are separate entities that link two or more components together. Thus, these exogenous connectors encapsulate control entirely and this makes components truly independent as they are loosely coupled since no direct dependencies exist between them. Exogenous connectors therefore maximise loose coupling not only in terms of data and functions but also in terms of control which existing component models are incapable of achieving [7]. Figure 3.3 shows an example of exogenous connectors being used to connect components. When compared to Figure 3.2, you can see that now the exogenous connectors (blue boxes) are responsible for initiating the method calls as opposed to the components themselves. The components only encapsulate the computation whereas the connectors now encapsulate the control entirely. Since no direct dependencies exist between components, all the components can now easily be reused to develop other software systems.

![Figure 3.3: Connecting components using exogenous connectors](image)

3.2.3 Justification for Using Exogenous Connectors Component Model

Due to the increased popularity of reusing software components in the avionics domain, it was important to choose a component model that offered maximum loose coupling since avionics software is inherently complex and time consuming to build due to its safety-critical nature. For
these reasons, the component model based on exogenous connectors was chosen to be the most appropriate model to use for the design and implementation of the fuel management system for the Airbus A350 XWB airliner. This component model offers the benefits discussed above which are important for the implementation of this system. Being able to define new software components which can then be reused easily to implement other systems is essential in this case and the software component model based on exogenous connectors is guaranteed to offer this since computation and control are encapsulated separately. This means that new avionics software systems can be easily developed without the need of making significant changes to adapt components to fit the new system requirements when they are reused.

3.3 Exogenous Connectors Component Model Tool

To aid the development of the fuel management system an in-depth understanding of a tool that was supplied by the School of Computer Science was required. This tool consists of 2 major parts. The first part is the component model based on exogenous connectors. As mentioned earlier, this component model is used to develop software systems. Any software system can be developed using this component model since it is a general specification. In this case it will be used to implement the fuel management system since our focus is on developing such a system using reusable software components. The second part of the tool refers to the interpreter. This is a piece of software that was written by a colleague at the School of Computer science and its sole purpose is to process the fuel management system that has been implemented using the component model and it involves executing a simulator that will test the fuel management system and generate some outputs to check the validity of the system. Both these parts of the tool will be discussed in detail in the following sections.

3.3.1 Exogenous Connectors Component Model Structure

The exogenous connectors’ component model was defined in a modelling environment called Generic Modelling Environment (GME). GME is a graphical based software toolkit that allows you to create software models for specific domains [18] which is shown in Figure 3.4. The toolkit is configurable which means that developers can define their own metamodels\(^5\) and use these metamodels to create specific instance models (implementations) for a given domain. To avoid any confusion of terms, the component model based on exogenous connectors is actually a metamodel itself although it is referred to as a component model.

\(^5\) Specifies the properties that specific instance models have such as what concepts will be used to construct models, what relationships can exist between these concepts and rules that govern the construction of such models. All models conform to its metamodel.
Figure 3.4: The GME toolkit which is used to create implementations of systems using the exogenous connectors’ component model.

Figure 3.5 shows a class diagram of the basic structure of the exogenous connectors’ component model which will be used to implement the fuel management system. Certain details have been omitted from the diagram to simplify understanding. Each item in the diagram will be explained briefly to give a more thorough understanding of the component model. Notice how some of the elements shown in Figure 3.5 are represented on the left hand side of the GME window in Figure 3.4. These elements are used to create the system implementation.
Figure 3.5: Class diagram showing the Exogenous Connectors Component Model.

In Figure 3.5, the Component class is used to define any component in a system such as a checkout component that is responsible for processing a customer’s order and charging them. Notice that the
component class has a single attribute called ‘Order’ of an integer type. This refers to the order of the component at the given level. For example at any given level there could be several components and each component needs to specify their order to help the connectors determine which components to execute in which order. The Component class actually serves the purpose of generalising components that have something in common; in this case all components will have an ‘Order’ attribute. The Component class has two subclasses called ‘Atomic Component’ and ‘Composite Component’.

The Atomic Component class refers to a component that cannot be composed of other components. This is the lowest level component found in a system and can only consist of a ‘Computation Unit’ and an ‘Invocation Connector’. A ‘Computation Unit’ encapsulates all the data and computation of the unit. It provides the methods that the component could offer and therefore it can consist of one or more ‘Method’ objects which are responsible for processing the code for the given unit. It is important to note that a ‘Computation Unit’ does not invoke any computation outside itself [15]. An ‘Invocation Connector’ serves as an interface for the component and is used to invoke the methods in the ‘Computation Unit’ [16]. This is the only point of access to the methods of the ‘Computation Unit’ and therefore the ‘Invocation Connector’ serves this only purpose. You will notice that the Computation Unit class has 1 attribute called ‘ExecutableCode’. This is where the executable code for this component is supplied. For example, if this computation unit is responsible for processing the average of a set of numbers input, then the programming code that deals with this processing is supplied here.

A composite component is a component that can consist of one or more (atomic or composite) components. These components are used to develop the high level structures of the system. For example, in an implementation the top most component is usually a composite component that will consist of one or more other (atomic or composite) components. As you traverse down the structure of the system there will be other composite components. Figure 3.6 shows a diagrammatic representation of an atomic component and a composite component.

![Figure 3.6: A diagrammatic representation of an Atomic Component and a Composite Component](image)

The Method class refers to the method that deals with some form of computational processing just like a method written in any programming language. It has two attributes called ‘Name’ and ‘MethodSequence’. The ‘Name’ attribute is just the name of the method given by the developer.
Every single method must be supplied a name. The ‘MethodSequence’ refers to the order in which a number of methods are executed. This second attribute applies mainly to composite methods which are methods defined inside a ‘Composite Component’. In a ‘Composite Component’ there may be several components defined each of which have their own method defined inside. So a composite method defined at this level would specify the order in which these sub-component’s methods are going to be executed. This just helps the connectors determine the order in which the methods should be executed so that the necessary input parameters can be supplied to each sub-component. If only one (atomic or composite) component has been defined at a given level then the name of the method defined inside this component is supplied to the attribute ‘MethodSequence’ else if multiple components have been defined at the given level then the order in which you want these component’s methods to be invoked is specified by just listing the name of the methods.

Each ‘Method’ can consist of one or more ‘Inparam’ and ‘Outparam’ objects. Both ‘Inparam’ and ‘Outparam’ have two attributes called ‘Order’ and ‘ValueType’. The ‘Order’ attribute refers to the order of the parameter inside the method. Each parameter must have an order value specified so that the connectors can determine which parameters to pass to which components. The ‘ValueType’ attribute specifies the type of the value in the ‘Order’ attribute which can be a string, integer or any other type. It is important to note that the order values specified for output parameters do not continue on from the order values of the input parameters. For example if method A has 3 inputs (a, b, c) and 2 outputs (x, y) then the order values for these will be 1, 2, 3 and 1, 2 respectively, assuming the integer type has been selected for the second attribute. So the outputs do not continue from the inputs, they always start from 1.

The Connector class has 2 types of connectors: ‘Invocation Connector’, which has already been discussed, and a ‘Composition Connector’ (Figure 3.6) which refers to the exogenous connectors that encapsulate control. It is these connectors that make this component model unique. There are 3 different types of exogenous connectors each of which has a specific purpose. It is important to note that these 3 exogenous connectors can only be used in composite components.

When a ‘Sequencer’ connector is used in a composite component it invokes the methods of the sub-components (components inside the composite component) in sequential order, one after another [17].

When a ‘Pipe’ connector is used in a composite component it invokes the methods of the sub-components sequentially one after another similar to the Sequencer connector except that the output of one sub-component’s computation is used as the input into the next sub-component’s execution and so on [17].

When a ‘Selector’ connector is used in a composite component, only one sub-component’s method is executed based on the evaluation of a boolean expression [17]. Figure 3.7 shows a diagrammatic representation of these 3 different exogenous connectors in use.
3.3.2 The Exogenous Connectors Component Model Interpreter

The interpreter that was supplied is a piece of software that contains the code for processing the implementation of the fuel management system developed in GME by using the exogenous connectors’ component model. Because you cannot directly write code for the components and connectors inside GME (except for computation units), an Interpreter is required to process the implementation of the fuel system and a simulator is used which allows you to supply some input values to test the system which in turn generates some output values. The way the interpreter works is each time an element (such as a component or a method in Figure 3.5) is encountered in GME, some code is executed to process that element.

The interpreter was written in the C++ programming language by a colleague at the School of Computer Science using the Microsoft Visual Studio environment. Before any design and implementation of the fuel management system could begin, it was essential to gain a thorough understanding of this interpreter since the behaviour of all the classes in Figure 3.5 have been defined here. This was especially important for the three connectors (sequencer, pipe and selector) since the interpreter improves the component model further because coding allows you to express much more than the GME modelling toolkit. So it was important to understand each connector’s code clearly so that the implementation of the fuel management system would be accurate.

Figure 3.7: Behaviour of (a) sequencer, (b) pipe and (c) selector exogenous connectors [17]
The purpose of the project was to design and implement a fuel management system for the upcoming Airbus A350 XWB commercial airliner. Before any design and implementation activities could begin it was essential to understand the requirements of the fuel management system. These requirements were made available to us by Airbus via a non-disclosure agreement. The requirements included two sets of documentation that outlined the requirements for the proposed fuel management system, both in textual format and in diagrammatic state charts format.

4.1 Airbus Requirements Documentation

The first requirements documentation specifies the functional requirements of the fuel management system in textual format. An example of a functional requirement for the 'Manual Refuel' operation is given in Figure 4.1. Due to the non-disclosure agreement with Airbus the textual requirement in Figure 4.1 has been slightly altered to ensure privacy of their data is preserved.

The Manual Refuel mode shall consist of 3 exclusive sub-modes:

- Idle
- In Progress
- Abort

![Figure 4.1: Textual functional requirement for the 'Manual Refuel' operation.](image)

The textual requirements are written in a hierarchical manner, starting with the high level sub-mode descriptions and progressively moving down to the low level sub-mode descriptions which contain more detail. The textual requirement in Figure 4.1 is the top level requirement and is fairly simple to understand. However, as we move down to each sub mode’s functional requirements, they become more detailed. For example, the requirements of the 'Abort' sub-mode are shown in Figure 4.2. Notice how the requirement is more detailed.
Manual Refuel shall enter the 'Abort' sub-mode when any of the following conditions are true:
- There is an overflow condition.
- Any Jettison Valve (J1 or J2) is failed OPEN.
- Any valve connecting the refuel gallery and engine feed gallery is failed OPEN.
- Any Wing Tank Transfer pushbutton on the cockpit ICP is selected ON.

**Figure 4.2:** Detailed functional requirement for the 'Abort' sub-mode.

The second requirements documentation is in diagrammatic state charts format. This documentation specifies the functional requirements for the fuel management system in the form of state charts which is a diagram technique. State charts are exactly the same as state transition diagrams in the UML modelling language. They describe the behaviour of systems by showing the states of the system and how the system can move from one state to the next based on a given condition. Figure 4.3 shows the state chart for the 'Manual Refuel' operation.

**Figure 4.3:** State chart for the 'Manual Refuel' operation.

Similar to the textual requirements documentation, the diagrammatic state charts requirements documentation is also written in a hierarchical manner. The state chart in Figure 4.3 is a high level state which contains 3 sub-states ('Idle', 'In_Progress' and 'Abort'). The 'Idle' and 'Abort' sub-states are simple and do not contain additional sub-states instead they only contain variables that are initialised to true or false. However, the 'In_Progress' sub-state does contain additional sub-states which are shown in Figure 4.4.
The requirements documentation was read in a hierarchical manner starting with the high level states first and progressively moving down to the lower level states. Figure 4.5 shows what a hierarchical representation of the states in Figure 4.3 and Figure 4.4 would look like. This helps ease the understanding of how these requirements documentation were read. The variable names and methods have been omitted for ease of readability. The full set of state charts for all the fuel operations can be viewed in appendix A. Note that all these state charts have been slightly altered due to the non-disclosure agreement with airbus.

### 4.2 Requirements Analysis

Both requirements documentations were read in conjunction since they both contained the necessary information to understand what was required. The textual requirements documentation contained more detail that was not present in the diagrammatic state charts requirements documentation. The diagrammatic state charts requirements documentation only contained state
charts and therefore any textual descriptions were limited to avoid cumbersome diagrams, hence the need for the textual requirements documentation. Therefore during requirements analysis we were required to read both sets of documentation side-by-side. For example, when trying to understand the ‘Manual Refuel’ operation functional requirements, the diagrammatic state chart requirements documentation was read first to get an idea of the structure of the system and once this was clear the textual requirements for the given state charts were read to gain a more detailed understanding of what was required.

Prior to and during the development of the fuel management system, the requirements for the fuel management system of the Airbus A350 XWB were read thoroughly since the system was based on these requirements. It was essential to analyse these requirements in great detail continuously due to the safety-critical nature of avionics software and to ensure the fuel operations were implemented correctly.

4.3 Setting the Scope of the Project

In general a fuel management system consists of two categories of operations: ground and flight operations. Ground operations refer to the operations that take place when the aircraft is on the ground and stationary where as flight operations take place when the aircraft is airborne. The textual and diagrammatic state chart requirements documentation consists of functional requirements for both ground and flight operations. The ground operations for the Airbus A350 XWB include Automatic Refuel, Manual Refuel, Defuel, Ground Transfer, Shut-off Test and Off. This project will only focus on ground operations due to a limited development time period and since the scale of the fuel management system is large and complex, only certain aspects of the fuel management system can be concentrated on.

4.4 Functional Requirements Ordering

Once the scope of the project was determined it was important to specify the order in which the functional requirements would need to be implemented. In total there were 6 ground operations. Each ground operation is listed below with a brief description of its purpose:

- **Automatic Refuel (AR)** – This operation is responsible for automatically refuelling the aircraft’s fuel tanks with fuel.
- **Manual Refuel (MR)** – This operation is responsible for manually refuelling the aircraft’s fuel tanks with fuel.
- **Defuel (DF)** – This operation is responsible for safely removing fuel from the aircraft’s fuel tanks.
- **Ground Transfer (GT)** – This operation is responsible for safely transferring fuel from one fuel tank to another within the aircraft.
- **Shut-Off Test (SOT)** – This operation is responsible for ensuring that the aircraft has safely been turned off before any refuelling can commence.
- **Off** – This operation is responsible for ensuring all fluid mechanical equipment is reset and no ground operation is inadvertently active.

Note that the entire fuel management system is composed of these 6 fuel operations. Table 4.1 shows the order in which each fuel operation was set to be implemented. The order of implementation was based on the complexity of each fuel operation and so the simplest operations were set to be implemented first because this would increase our understanding of designing and implementing each fuel operation, making the complex fuel operations less of a challenge. This was an important consideration to make due to the limited development time, since any errors could impact the ability to complete the fuel management system.

<table>
<thead>
<tr>
<th>Order Number</th>
<th>Fuel Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Off</td>
</tr>
<tr>
<td>2</td>
<td>Shut-Off Test (SOT)</td>
</tr>
<tr>
<td>3</td>
<td>Ground Transfer (GT)</td>
</tr>
<tr>
<td>4</td>
<td>Manual Refuel (MR)</td>
</tr>
<tr>
<td>5</td>
<td>Defuel (DF)</td>
</tr>
<tr>
<td>6</td>
<td>Automatic Refuel (AR)</td>
</tr>
</tbody>
</table>

**Table 4.1**: Table showing the implementation order of all the ground fuel operations.

### 4.5 Specific Tasks to Perform

The objective of the proposed project is as follows:

- Design and implement a fuel management system using the predefined reusable software components so that a functional system is the result, which is capable of managing the fuel for the Airbus A350 XWB commercial airliner. This will be achieved using a component model based on exogenous connectors.

In order to achieve this objective the tasks shown in Table 4.2 were initially drawn up to be performed.
<table>
<thead>
<tr>
<th>Task ID</th>
<th>Task Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Invest time in learning the GME toolkit. This will involve learning how to create meta-models and how to use these to create implementations of systems.</td>
</tr>
<tr>
<td>2</td>
<td>Invest time in learning new programming languages, primarily C and C++. This will be necessary to write the code in computation units of the fuel management system implementation in GME and also to understand the interpreter mentioned in task 3.</td>
</tr>
<tr>
<td>3</td>
<td>Invest time in learning the tool supplied by the School of Computer Science. This tool consists of the component model based on exogenous connectors and the interpreter which is responsible for processing the implementation developed in GME.</td>
</tr>
<tr>
<td>4</td>
<td>Invest time in analysing and understanding the textual and diagrammatic state chart requirement documentations of the fuel management system for the upcoming Airbus A350 XWB airliner.</td>
</tr>
<tr>
<td>5</td>
<td>The understanding gained from analysing and understanding the exogenous connectors’ component model, the interpreter and the fuel management system requirement documentations will be used to design the system by identifying the atomic and composite components and the necessary exogenous connectors required. Each ground fuel operation will be designed individually.</td>
</tr>
<tr>
<td>6</td>
<td>Once the system has been designed it will be implemented in GME using the component model based on exogenous connectors. Each ground fuel operation will be implemented individually, one after another.</td>
</tr>
<tr>
<td>7</td>
<td>Each ground fuel operation will be processed by the interpreter to ensure it conforms to the exogenous connectors’ component model. A simulator will then be used to test each fuel operation which will be supplied with some test input values to ensure each fuel operation is valid.</td>
</tr>
<tr>
<td>8</td>
<td>Once all the ground fuel operations have been implemented and tested then the entire fuel management system (composed of all the ground fuel operations) will be tested using the interpreter and simulator.</td>
</tr>
<tr>
<td>9</td>
<td>The final system will be evaluated to ensure the outlined objective has been met successfully.</td>
</tr>
</tbody>
</table>

Table 4.2: Table showing all the tasks performed for the entire project.

The tasks in Table 4.2 refer to everything that was performed in order to implement the fuel management system. It is important to note that all the ground fuel operations outlined in Table 4.1 are part of task 5 and 6 of Table 4.2 because it was in these tasks where each ground fuel operation was designed and implemented. Although the tasks in Table 4.2 have been listed in numerical order, some of these tasks were carried out concurrently. For example tasks 1 and 2 were done in parallel since at this stage it was important to gain a good understanding of the GME toolkit but at the same
time acquire good knowledge of the C and C++ programming languages so that the code for the interpreter in task 3 could be understood clearly and the code inside the computation units of the fuel management system in GME could be written during implementation.
The design phase involved designing each fuel operation of the fuel management system so that it could easily be converted into an implementation in the GME toolkit. Each fuel operation was designed individually by analysing the requirements documentation thoroughly to ensure all requirements were incorporated in the design. This chapter first discusses a suitable diagram technique that was selected to design the fuel operations and an example of how a particular fuel operation was designed is demonstrated followed by how the variables from the diagrammatic state chart requirements documentation were mapped to the design.

5.1 Choosing a Suitable Diagram Technique

Before any of the fuel operations could be designed it was important to choose a suitable diagram technique that would allow us to design the structure of each fuel operation in such a way that it would be simple to convert this into an implementation in GME. Many diagram techniques and languages were considered such as the Unified Modelling Language (UML), however all these techniques and languages seemed to be insufficient for our needs. The main reason to this was the way systems are implemented in GME using the component model based on exogenous connectors. GME follows a hierarchical structure for implementation. So if a developer wanted to create an implementation in GME they would do this in a hierarchical manner. Figure 5.1 shows an example of a system implementation built using components. In GME the composite component ‘A’ would be at the root level. It contains 2 sub-components called ‘A1’ and ‘A2’. So in GME if the developer wanted to access the contents of composite component ‘A’ they would double-click on the component. Sub-component ‘A1’ is also a composite component because it contains a sub-component called ‘A1.1’. Again, if the developer wanted to access this sub-component then they would double-click on the composite component ‘A1’ to view its contents. As you can see a hierarchical tree structure begins to form. During the time we spent learning the GME toolkit we felt that using a hierarchical tree structure diagram would be the most suitable diagram technique to design each fuel operation and hence the entire fuel management system since it would easily allow us to convert the design into an implementation in GME because GME itself follows this hierarchical tree structure. Thus, all 6 fuel operations and hence the entire fuel management system was designed using this diagram technique.
5.2 Designing the Fuel Operations

Each fuel operation was designed individually on paper using a hierarchical tree structure diagram technique as shown in Figure 5.1. For each fuel operation the requirements documentation (both textual and diagrammatic state charts) were analysed carefully to help determine how to create the overall structure using the exogenous connectors’ component model. When designing the fuel operations we had to make sure that the design conformed to this component model since the whole objective of the project was to develop a fuel management system using reusable software components.

The structure of each fuel operation was designed in a top-down manner, starting at the root level and progressively moving down to the lower levels. Figure 5.2 shows an example of a state chart for the ‘Manual Refuel’ operation from the diagrammatic state chart requirements documentation. Due to the non-disclosure agreement of data with Airbus, the actual state chart from their requirements documentation could not be used and therefore the state chart shown in Figure 5.2 has been slightly altered showing only the relevant details. Figure 5.3 shows the hierarchical tree structure design of the ‘Manual Refuel’ operation we created, which conforms to the exogenous connectors’ component model, based on the state chart in Figure 5.2. The process of how we created the hierarchical tree structure design in Figure 5.3 will be explained in detail later when we discuss how the state chart in Figure 5.2 was mapped to this design.
**Figure 5.2**: State chart for the ‘Manuel Refuel’ operation based on the actual Airbus diagrammatic state chart requirements.
Notice that in Figure 5.2 the top level state is called 'Manual Refuel' and this state consists of 3 sub-states called 'Idle', 'In_Progress' and 'Abort'. The 'Manual Refuel' state also consists of an 'Initialise Variables' action and an 'Evaluate_Conditions' method. Before discussing how this fuel operation was designed in Figure 5.3, it would be useful to briefly mention the purpose of the 'Initialise Variables' action and the 'Evaluate_Conditions' method as these are present in all the fuel operation state charts in the diagrammatic state chart requirements documentation.

The 'Initialise Variables' action refers to what happens as soon as the 'Manual Refuel' state becomes active. When we say active we mean the state will be executed. The keyword 'On Entry to State' indicates that as soon as this state is active, initialise some variables. These variables are system state variables and determine which fuel operation is active at any time and only one fuel operation can be active at any time. These system state variables are of boolean type and if the 'Manual Refuel' state has been selected then on entry to this state when the 'Initialise Variables' action is executed the system state variable called 'SS_MR_ACTIVE' will be set to true and the other 5 system state variables that represent the other fuel operations will be set to false, as is clearly demonstrated in Figure 5.2. It is important to note that the 'Initialise Variables' action is only present at the top level state of each fuel operation, it does not appear elsewhere in the lower sub-states. Its purpose is to just initialise the system state variables so that the current fuel operation being executed is known.
The 'Evaluate_Conditions' method is simply a method that determines which sub-state ('Idle', 'In_Progress' or 'Abort') in the 'Manual Refuel' state will be active. The method performs some computation based on the values of particular variables (not shown in Figure 5.2) that have been supplied to the method and determines which sub-state should be executed. The actual sub-state that is executed depends on the value of the 3 different variables that are located near the transition lines of the states in square brackets of Figure 5.2. These 3 variables are set and output when the 'Evaluate_Conditions' method is computed. Whichever variable is true is the sub-state that is executed.

The requirements documentation of the 'Manual Refuel' state indicates that only one sub-state at any time can execute. This is indicated by a solid line around the state box in Figure 5.2. If multiple states can execute concurrently then this is represented by a dotted line around the state box. An example of this will be discussed later. Notice that the 'Evaluate_Conditions' method is executed on entry to the 'Manual Refuel' state and during the time the 'Manual Refuel' state is active. The keyword 'During the state is active' in Figure 5.2 indicates this. The 'Evaluate_Conditions' method must constantly be executed whilst the 'Manual Refuel' state is active in order to determine which sub-state to execute next.

Now that the notation of the state charts has been briefly explained we will discuss how the design for one of the fuel operations ('Manual Refuel') shown in Figure 5.3 was created based on the state chart in Figure 5.2.

When it came to designing the 'Manual Refuel' operation, creating a hierarchical tree structure design was not entirely straight-forward. Due to the way the exogenous connectors’ component model was developed, it was not always possible to directly map the requirements in the state charts to the design. In this case we had to use our own initiative to find a way in which we could design the structure without affecting the functionality originally outlined in the requirements.

Each state in Figure 5.2 and each component in Figure 5.3 have a corresponding colour code to help understand the mapping process of how each state was mapped and represented in the hierarchical tree structure design. This will help simplify how the mapping process was undertaken for the 'Manual Refuel' operation which will be discussed in a step-by-step manner.

Mapping the 'Manual Refuel' state in Figure 5.2 was relatively straight-forward since this is the top level state for this fuel operation. It was mapped to the 'Manual Refuel Op' composite component in Figure 5.3. Since the 'Initialise Variables' action in Figure 5.2 would take place as soon as the 'Manual Refuel' state became active it was important to ensure that this was represented as the first sub-component of the 'Manual Refuel Op' composite component in the hierarchical tree structure shown in Figure 5.3. This component was named 'Manual Refuel Init' and this is an atomic component because it will not be composed of other sub-components. Note that although the 'Initialise Variables' action is responsible for initialising the system state variables, this had to be represented inside an atomic component in Figure 5.3 because this is the rule of the exogenous connectors’ component model and so our design had to conform to this. The actual computation code that initialises the system state variables would be written inside the computation unit for this atomic component.
The second sub-component of the 'Manual Refuel Op' composite component in Figure 5.3 is called 'Manual Refuel Execution'. This is a composite component because it consists of other sub-components. Although this component does not exist as a state in Figure 5.2, the reason for its existence was due to what was mentioned earlier about the exogenous connectors' component model and how due to the way it was developed it was not always possible to map the state charts directly when creating the design for the fuel operations.

Before explaining the need for the 'Manual Refuel Execution' composite component it is important to note that a sequencer connector was used to connect the 'Manual Refuel Init' atomic component and 'Manual Refuel Execution' composite component, as can be seen in Figure 5.3. This was a decision that had to be made during design. Recall that the sequencer connector invokes the methods of the sub-components in sequential order, one after another. The 'Manual Refuel Init' atomic component will output some values but these will not be used by any other sub-components, instead they will need to be passed out of the sequencer connector and out of the component. However, the 'MR Evaluate Conditions' atomic component in Figure 5.3 (which corresponds to 'Evaluate_Conditions' method in Figure 5.2) will need to output some values that will help the 'Manual Refuel Process' composite component to determine which sub-component to execute (either 'MR Idle', 'MR In_Progress' or 'MR Abort'). So in this case the 'MR Evaluate Conditions' atomic component will require a pipe connector. However, the rules of the exogenous connectors component model state that two connectors cannot be used at the same level and therefore the structure of the system had to be designed differently to that specified in the state chart requirements documentation (where Figure 5.2 shows that the 'Initialise Variables' action and the 'Evaluate_Conditions' method are both represented at the same level in the 'Manual Refuel' state).

Figure 5.4(a) shows what the hierarchical tree structure would look like if both the 'Manual Refuel Init' atomic component and the 'MR Evaluate Conditions' atomic component were represented at the same level. Notice the conflict of the connectors where the 'Manual Refuel Init' atomic component requires a sequencer connector but the 'MR Evaluate Conditions' atomic component requires a pipe connector. Based on the rules of the exogenous connectors' component model, you cannot have more than one exogenous connector on the same level and so the structure had to be changed to conform to the rules of this component model and this corrected structure is shown in Figure 5.4(b). Notice how in Figure 5.4(b) the conflicting connectors’ problem has now been resolved by adding a new composite component called 'Manual Refuel Execution' which now contains the 'MR Evaluate Conditions' atomic component and this composite component now uses a pipe connector to connect to this atomic component. So the exogenous connectors have been separated and only one exists on each level making the structure valid.
Figure 5.4: Showing the (a) incorrect tree structure and the (b) improved correct tree structure for the 'Manual Refuel Op' composite component.

So based on Figure 5.4(b) a sequencer connector is used to connect the 'Manual Refuel Init' atomic component and 'Manual Refuel Execution' composite component. In this case the 'Manual Refuel Init' atomic component will be executed first and the given system state values output will be passed out of the connector and then the second component called 'Manual Refuel Execution' will be executed and this composite component consists of a pipe connector that connects the 'MR Evaluate Conditions' atomic component with other components that have been omitted.

The next part of the design for the 'Manual Refuel' operation was to design the structure of the 3 sub-states of the 'Manual Refuel' state called 'Idle', 'In_Progress' and 'Abort' as shown in Figure 5.2. At this point a problem was encountered similar to the one discussed above. The requirements for the 'Manual Refuel' operation specify that at any time only one of these 3 sub-states can be active.
So it was clear that a selector connector would be most appropriate to use here however there would be a conflict of representing these 3 sub-states as components in the hierarchical tree structure since these 3 sub-states appear in the same state as the ‘Evaluate_Conditions’ method in Figure 5.2. Figure 5.5(a) shows this conflict when these states are mapped as components in the hierarchical tree structure. Notice how the ‘MR Evaluate Conditions’ atomic component requires a pipe connector, since it outputs some values, however the ‘MR Idle’, ‘MR In_Progress’ and ‘MR Abort’ components (which correspond to the sub-states ‘Idle’, ‘In_Progress’ and ‘Abort’ in Figure 5.2) require a selector connector. Thus, representing the structure as shown in Figure 5.5(a) would be invalid and so it had to be modified to reflect the rules of the exogenous connectors’ component model which is shown in Figure 5.5(b). So now a new composite component called ‘Manual Refuel Process’ was created at the same level as the ‘MR Evaluate Conditions’ atomic component which would contain the selector connector along with the 3 sub-components. Notice how the conflicting connectors problem has now been resolved by creating this new composite component called ‘Manual Refuel Process’. Note that the higher level components shown in Figure 5.4 have been omitted in Figure 5.5 to simplify understanding since the focus here is now on the ‘Manual Refuel Execution’ composite component.

You will notice that the ‘Evaluate_Conditions’ method in Figure 5.2 has been represented as an atomic component called ‘MR Evaluate Conditions’ in the design. Again, due to the way the exogenous connectors’ component model has been developed, we had to represent this as an atomic component. The actual computation code that evaluates the conditions is written inside the computation unit (which is located inside the atomic component). This will be discussed more in later sections.
So in Figure 5.5(b) when the ‘MR Evaluate Conditions’ atomic component executes and values are output from it, these values are passed as inputs into the ‘Manual Refuel Process’ composite component to help determine which sub-component to execute from ‘MR Idle’, ‘MR In_Progress’ and ‘MR Abort’. This design decision of having to change the structure of each fuel operation when the state chart was mapped to the design had to be made numerous times for each fuel operation. This was an important design decision because we had to ensure that our design conformed to the exogenous connector's component model but at the same time we had to ensure this design incorporated all the functionality specified in the requirements documentation and that the system behaved as outlined in these requirements when implemented.

Only a sample of the ‘Manual Refuel’ operation hierarchical tree structure design was shown in Figures 5.3, 5.4 and 5.5 for discussion purposes. The complete hierarchal tree structure for the ‘Manual Refuel’ operation can be viewed in appendix B. All the other sub-states for the ‘Manual
Refuel' operation do not require further discussion since the design decisions made were very similar to the ones discussed above.

Designing each fuel operation was an iterative process. We first had to consider which components had to exist at a given level and then an appropriate exogenous connector had to be selected which would connect these components together. If multiple connectors were required at the same level mainly because one component required the use of one connector while the other required the use of another, then the components had to be separated so that they were not on the same level as described above with the 'Manual Refuel Init' atomic component and the 'MR Evaluate Conditions' atomic component. In the state charts requirements documentation, both these exist in the same state ('Manual Refuel') in Figure 5.2. However, when it came to mapping them using the exogenous connectors' component model to create our hierarchical tree structure design, the 'Manual Refuel Init' atomic component and the 'MR Evaluate Conditions' atomic component could not be represented on the same level because they both required a different exogenous connector and due to the rules of the exogenous connectors' component model this is not allowed. Hence the reasons for separating them onto different levels in the hierarchical tree structure design as shown in Figure 5.4(b).

One feature in the state charts requirements documentation which really helped determine when a composite component was required was the decision node. Figure 5.6 shows what a decision node looks like (grey circle).

![Decision Node in State Chart](image)

**Figure 5.6**: A decision node being used in the state charts.

Whenever a decision node was encountered, this gave us a clue that a composite component was required. For example, if we look at the 'Manual Refuel' state in Figure 5.2, you will notice that a decision node exists that connects the 3 sub-states 'Idle', 'In_Progress' and 'Abort'. Since only one of these sub-states can be active at any time, it was clear that a selector connector would be required for these sub-states. However, since the 'Evaluate_Conditions' state will need to be represented using a pipe connector (as the outputs generated will help determine which sub-state to execute), there would be a conflict in representing both these states as components since at any level no more than one exogenous connector can be used. Therefore the 3 sub-states are encapsulated in a separate composite component called 'Manual Refuel Process' along with the selector connector as
can be seen in Figure 5.5(b) and this composite component can now be represented at the same level as the 'MR Evaluate Conditions' atomic component connected by a pipe connector. The decision nodes helped speed up the design process and also made the mapping process much simpler.

All the fuel operations were designed similarly in terms of representing them as a hierarchical tree structure and the design decisions made at each stage were similar to those discussed above therefore for this reason the rest of the designs will not be discussed however the full hierarchical tree structure designs for all the fuel operations can be viewed in appendix B.

5.3 The Need for a New Connector

During the process of designing the hierarchical tree structures for each fuel operation, it became apparent that when attempting to map certain state charts in the requirements documentation, the 3 exogenous connectors did not meet our needs entirely in some cases. For example, when we attempted to map the 'In_Progress' state chart shown in Figure 5.7 (which is actually a sub-state of the 'Manual Refuel' state in Figure 5.2), we noticed that multiple states could be active at the same time. The dotted lines around the state boxes indicate that multiple states can execute concurrently. The problem was that the exogenous connectors' component model we did not offer this type of connector. The connector which closely resembled it was the sequencer connector since it could be used to execute the states in order and eventually all the states would be executed, although not concurrently.

Therefore during design we thought that it might be a good idea to design this type of connector however due to the limited development time it could not be implemented. The name given to this type of connector was 'Concurrent connector'. This connector would allow multiple components to be executed at the same time and could be added to the future work for the project.
It is important to note that this concurrent connector was not used for the implementation of this fuel management system since it was not fully designed and implemented. It was only mentioned here since the need arose for another type of connector when faced with a situation where multiple components can execute concurrently like in Figure 5.7. So in place of where the ‘Concurrent connector’ could have been used we felt that the sequencer connector was sufficient since all the components would get executed regardless, although not concurrently.

5.4 Mapping the Variables in the State Charts to the Design

Once each fuel operation was designed in a hierarchical tree structure manner the next step was to map the variables shown in the state charts requirements documentation. A unique feature of the exogenous connectors’ component model is that no global variables can exist. A global variable is a variable that is accessible in every scope [19]. Therefore the only way a variable can be used in the exogenous connectors’ component model is if it is passed as a parameter to a method. If you recall the class diagram representation of the exogenous connectors’ component model in Figure 3.5 of Chapter 3, you will see that the ‘Method’ class has 2 parameters: input (‘Inparam’ class) and output (‘Outparam’ class) parameters. This is the only mechanism by which variables can be passed to the component’s methods and since our design must conform to the exogenous connectors’ component model we had to make sure that the variables in the state charts were mapped in accordance to these rules. However, the problem was that the diagrammatic state chart requirements documentation used global variables in all the fuel operations and therefore we had to ensure that during mapping these global variables were represented as input and output parameters.

Before the variables could be mapped to the design, all the variables in the state chart requirements documentation for a given fuel operation had to be identified. All the variables that are used throughout each fuel operation state chart are located in the ‘Evaluate_Conditions’ method. In addition to this the system state variables shown in the ‘Idle’ and ‘Abort’ sub-states in Figure 5.2 are also used. Figure 5.8 shows the contents of the ‘Evaluate_Conditions’ method for the ‘Manual Refuel’ state chart from Figure 5.2. It shows the code that deals with evaluating the conditions. The ‘Evaluate_Conditions’ method in Figure 5.8 has been modified slightly to preserve the privacy of Airbus’s data.

You will notice that the code written inside the ‘Evaluate_Conditions’ method is not language specific and so can essentially be classed as pseudocode⁶. The code inside the method will be briefly discussed so that its purpose is clear. As mentioned earlier the purpose of the ‘Evaluate_Conditions’ method is to carry out some computations on some variables and determine which sub-state to execute based on the value of the variables. The ‘Evaluate_Conditions’ method is present in all the fuel operations at the top most level states. Notice that the structure of the method is similar to any method written in any programming language. All the local variable names have been highlighted in red and they all have a boolean type. If we look at the code for the first local variable called ‘ANY_TIV_SWITCH_OPEN’ you will notice that if the ‘IRP_MAN_VALVES_TIV_LWT’ or

---

⁶ A high level description of a computer programming algorithm which uses conventions similar to a programming language but is intended to be read by a human rather than a machine [19].
‘IRP_MAN_VALVES_TIV_RWT’ or ‘IRP_MAN_VALVES_TIV_CT’ variables have a value of ‘OPEN’ then the value of the local variable ‘ANY_TIV_SWITCH_OPEN’ evaluates to true. The rest of the code is similar to this so will not be explained. The last three variables (‘Abort’, ‘Idle’ and ‘In_Progress’) are output of this method and used by the ‘Manual Refuel Process’ composite component to determine what sub-component to execute which can be seen dearly in Figure 5.5(b).

```
Method name = Evaluate_Conditions
{
    Boolean ANY_TIV_SWITCH_OPEN =
    IRP_MAN_VALVES_TIV_LWT == OPEN OR
    IRP_MAN_VALVES_TIV_RWT == OPEN OR
    IRP_MAN_VALVES_TIV_CT == OPEN;

    Boolean MR_ANY_TIV_OPEN =
    RES_EQUIP_STATE_V_RL == OPEN OR
    RES_EQUIP_STATE_V_RR_ == OPEN OR
    RES_EQUIP_STATE_V_RC1 == OPEN OR
    RES_EQUIP_STATE_V_RC2 == OPEN;

    Boolean MR_ANY_JTSN_VLV_OPEN =
    PIN_PROGRAMMING_OPT_JETTISON == TRUE AND (RES_EQUIP_STATE_V_J1 == OPEN OR
    RES_EQUIP_STATE_V_J2 == OPEN);

    Boolean MR_ANY_CONNCTNG_VLV_FAILED_OPEN = ...
    (RES_EQUIP_STATE_V_D == OPEN AND RES_EQUIP_STATUS_V_D == FAILED) OR
    (PIN_PROGRAMMING_OPT_JETTISON == TRUE AND
    ((RES_EQUIP_STATE_V_B == OPEN AND RES_EQUIP_STATUS_V_B == FAILED) OR
    (RES_EQUIP_STATE_V_JL == OPEN AND RES_EQUIP_STATUS_V_JL == FAILED) OR
    (RES_EQUIP_STATE_V_JR == OPEN AND RES_EQUIP_STATUS_V_JR == FAILED)));

    Boolean MR_MAN_XFR_SELCTD =
    AFDX_ICP_CMDS_ICP_LWT_XFR_PB == ON OR AFDX_ICP_CMDS_ICP_RWT_XFR_PB == ON;

    Boolean Abort =
    MR_ANY_JTSN_VLV_OPEN OR MR_ANY_CONNCTNG_VLV_FAILED_OPEN OR
    MR_MAN_XFR_SELCTD;

    Boolean Idle = ANY_TIV_SWITCH_OPEN == FALSE AND Abort == FALSE;

    Boolean In_Progress =
    ANY_TIV_SWITCH_OPEN == TRUE AND Abort == FALSE;
}
```

Figure 5.8: Showing the code inside the ‘Evaluate_Conditions’ method from Figure 5.2.

All the variables which are used in the entire ‘Manual Refuel’ state chart are listed in Figure 5.8 and the additional system state variables are also used as shown in Figure 5.2 in the ‘Idle’ and ‘Abort’ sub-states. Once all these were known then the mapping process of the variables could begin.
The mapping process for the variables was quite simple since the structure of each fuel operation had already been designed. So all we had to do was map the variables as parameters and add them inside the given component’s method in the hierarchical tree structure diagrams. But to avoid cluttering these diagrams we decided to add the list of required parameters for given components on a separate piece of paper under a heading for each component. An example of how this was done is shown in Figure 5.9. The red headings refer to the component names and the black text refers to the parameter names for that component. The blue text indicates whether the given parameter will be an input or output parameter. It was important to indicate this during design so that it would simplify the process of implementation as there were many variables to deal with for each fuel operation.

Figure 5.9: Showing how variables were defined by listing the variables that belonged to a component under the name heading for that component.

To help explain the mapping of the variables we will use several diagrams. Figure 5.10 shows the state chart for the manual refuel left wing tank (MR_TK_LWT) which is located at the lowest level of the 'Manual Refuel' state. Figure 5.11 shows the hierarchical tree structure design based on the manual refuel left wing tank state chart (MR_TK_LWT) in Figure 5.10. In Figure 5.11 the components between the composite component 'Manual Refuel Op' and the 'MR_TK_LWT' composite component have been omitted since the level of the tree structure is relatively large. The full structure can be viewed in appendix B.

Notice how in Figure 5.10 the system state variable called 'SS_MR_LWT' is used by the 'Idle' and 'Active' states. This system state variable was mapped as an output parameter in the 'MR_LWT_Idle' and 'MR_LWT_Active' atomic components in Figure 5.11. As mentioned earlier we did not want to clutter the hierarchical tree structure designs (Figure 5.11) by adding all the parameters and so this parameter is shown in Figure 5.9 under the headings 'MR_LWT_Idle' and 'MR_LWT_Active' which refer to the purple components in Figure 5.11. The reason this variable was mapped as an output parameter is due to the rules of the exogenous connectors’ component model which states that no global variables can exist as all variables must be passed into and out of components as input or/and output parameters respectively. Because the system state variable 'SS_MR_LWT' is used by other higher level atomic components called 'MR_Idle' and 'MR_Abort' located towards the top level of
the hierarchical tree structure, as can be seen in the corresponding state chart in Figure 5.2, it had to be specified as an output parameter so that it could be passed all the way up. Therefore, once this variable has been set by either the ‘MR_LWT_Idle’ or ‘MR_LWT_Active’ atomic components it will need to be output so that other components can use it in case they need to reset the variable. Recall that each fuel operation can execute multiple times in an iterative manner because as the ‘Evaluate_Conditions’ method is computed, a different component may need to be executed.

**Figure 5.10:** Manual Refuel Left Wing Tank (MR_TK_LWT) state chart from the diagrammatic state chart requirements.

**Figure 5.11:** Hierarchical tree structure design based on the Manual Refuel Left Wing Tank (MR_TK_LWT) state chart in Figure 5.10.
Since no global variables can exist in the design and implementation of the fuel system due to the rules of the exogenous connectors’ component model, this also changes how variables at the lower level are accessed. Figure 5.10 shows that the ‘Evaluate_status’ method requires 3 different variables for computation: ‘IRP_MAN_VALVES_TIV_LWT’, ‘HIGH_LEVEL_STATE_TK_LWT’ and ‘HIGH_LEVEL_STATUS_TK_LWT’. If the ‘MR_LWT_Eval_Status’ atomic component in Figure 5.11 (which corresponds to the ‘Evaluate_status’ method in Figure 5.10) needs to use these variables they must be passed all the way down from the top most level component (‘Manual Refuel Op’) as input parameters because this is where the variables originate and therefore they must be passed all the way down as input parameters. Figure 5.12 shows a diagramatic representation of this.

Figure 5.12: Showing how variables required at the lowest level are passed down from the top most components as input parameters.

Notice how in Figure 5.12, the 3 variables ‘IRP_MAN_VALVES_TIV_LWT’, ‘HIGH_LEVEL_STATE_TK_LWT’ and ‘HIGH_LEVEL_STATUS_TK_LWT’ originate at the top most level inside the ‘Manual Refuel Op’ composite component and are passed all the way down to the ‘MR_LWT_Eval_Status’ atomic component as input parameters. Once they reach this point they can then be used. This is how the exogenous connectors’ component model works. Any variables that are required at any level must be passed down as input parameters through the component’s composite method because global variables are not permitted. If any variables are required at both the lower levels and higher levels of the hierarchical tree structure then these variables must be passed as output parameters so they can be passed back up to the higher levels from the lower levels once the values have been set.
The implementation phase involved converting the hierarchical tree structure designs created for all the 6 fuel operations during the design phase into a working software artefact using the Generic Modelling Environment (GME) toolkit. This chapter focuses on the important aspects of the fuel management system implementation.

6.1 Technology Choices

Appropriate software tools were required to aid the development of the fuel management system. This involved using certain modelling and programming languages that were suitable for avionics software development. This section discusses the tools that were selected.

6.1.1 Modelling Language Choice

There were various modelling languages on the market that could have been selected for the implementation of the fuel management system, however GME was specifically chosen since it is a recommended modelling environment in the avionics industry and since this project involves implementing a piece of avionics software, specifically a fuel management system, it seemed more appropriate to use GME as the primary modelling language.

Another modelling language that was considered is EMF (Eclipse Modelling Framework). EMF is similar to GME in that it is a modelling framework that allows you to develop software based on a structured model [20]. However, the difference between the two modelling languages is that GME is based on the C++ programming language where as EMF is based on the Java programming language. Since our experience in programming mainly lies entirely in Java, EMF was an attractive modelling language to begin with, however a majority of safety-critical software is programmed in C or C++ and since the fuel management system is itself a safety-critical piece of software, it was important to use a modelling language that was suitable for the given domain. Therefore it was decided that GME would be the most suitable modelling language for this project.
6.1.2 Programming Language Choice

The main programming language that had to be selected was C++ since the tool that was supplied to us by the School of Computer Science (composed of the exogenous connectors' component model and the interpreter) was written in the C++ programming language and GME itself is based on C++. Therefore we had to gain a good understanding of C++ in order to first understand the tool and then implement our fuel management system in GME. However, the programming language C was learnt first since around 80% of C++ is based on C and therefore acquiring this knowledge beforehand was essential. Although this meant that additional time had to be invested in learning entirely new programming languages (C and C++), this was considered more of an exciting challenge for us rather than a drawback. Learning new languages would be motivational since it would broaden our programming skills and knowledge and this is especially beneficial in industry.

6.2 Implementation Discussion

The fuel management system was developed in an incremental manner. First the entire structure of each fuel operation was implemented in GME based on the hierarchical tree structure designs in a top-down manner and the necessary variables were added as parameters in the component's methods in a bottom-up manner. The reason for adding the parameters in a bottom-up manner were because the data is passed via parameters and so each component had to be tested once it was implemented using the simulator (which is part of the tool) to ensure that the implementation was valid and conformed to the exogenous connectors' component model. Finally the code for each computation unit (located in every atomic component) was written using C++. This section focuses on the important and interesting aspects of the system's implementation.

6.2.1 Implementing the Fuel Management System Using the Tool

The fuel management system was implemented in GME using the tool that was supplied to us by the School of Computer Science. This tool was composed of the exogenous connectors' component model, which was used to create the implementation of the system, and an interpreter, which was used to interpret the implementation of the fuel management system in GME.

This tool played an important role in development since our implementation had to conform to the exogenous connectors' component model. When implementing the fuel management system in GME we used the hierarchical tree structure designs created during the design phase.

Figure 6.1 shows the implementation of the 'Manual Refuel Process' composite component in the GME toolkit. When defining the structure of this composite component we used the elements located on the left hand side of the GME toolkit as can be seen in Figure 6.1. These elements belong to the exogenous connectors' component model and were used to implement the entire fuel management system by dragging and dropping them onto the development area. The steps undertaken in developing the system in GME were similar for all the fuel operations. We first added the necessary exogenous connector followed by the sub-components (atomic and/or composite) and then all these components were connected to the given exogenous connector. The exogenous connector and all the sub-components were given appropriate names based on the names given in
the hierarchical tree structure designs. The next stage involved defining the composite method (‘proc’ composite method in Figure 6.1) inside the composite components and for the atomic components a method, invocation connector and a computation unit was defined. Once the entire fuel operation structure was complete the input and output parameters (as shown inside the ‘Proc’ composite method in Figure 6.1) were added to the methods in a bottom-up manner which will be discussed clearly in later sections. Finally the C++ code was written inside all the computation units for each fuel operation.

Figure 6.1: Showing the implementation of the ‘Manual Refuel Process’ composite component in the GME toolkit.

The interpreter was used to process the implementation of the fuel operations in GME. At each point as we developed the lower level components of the fuel operation, we used the interpreter to process the components to check that the implementation was valid and that it conformed to the exogenous connectors’ component model. Figure 6.1 shows the interpreter button in the GME toolkit. When this button is clicked a simulator GUI appears (Figure 6.2) which allows you to supply some input values for the input parameters that were defined for the composite methods of the components. Once the interpreter has executed and processed the implementation in GME some output values are generated. The output values depend on the values supplied as inputs. The values output refer to the system state variables which indicate which system state variable is currently true (1) or false (0). For example if the system state variable ‘SS_MR_LWT’ is output with a value of 1, this means that the left wing tank sub-component was executed to allow manual refuelling of the left wing tank. The simulator is discussed in more detail in the next chapter during testing.
It is important to note that the 2 parts that make up the tool (exogenous connectors’ component model and the interpreter) go hand-in-hand. The exogenous connectors’ component model was used to create the implementation of our fuel management system (recall that our objective was to use this component model to implement such a system) and the interpreter was used to interpret the implementation in GME based on the exogenous connectors’ component model.

GME is primarily a modelling language and so does not allow you to write code directly for each element of the exogenous connectors’ component model except in computation units. This is why an interpreter is required so that it can process the implementation in GME based on the exogenous connectors’ component model. Figure 6.3 shows the code for the interpreter which was used. It was developed by a colleague at the School of Computer Science in the Microsoft Visual Studio environment. It currently shows a method that contains the code for the selector connector but this is where the code for all the elements that appear in the exogenous connectors’ component model is defined. Each time the interpreter button in Figure 6.1 is clicked the simulator in Figure 6.2 appears. As soon as the input values have been supplied and the ‘Start’ button of the simulator is clicked the code in Figure 6.3 (interpreter) is executed and it processes the implementation of the fuel management system in GME. As the interpreter is executing and it encounters a particular element in GME, such as a selector connector, this triggers a particular method to be executed inside the interpreter depending on which element has been encountered. Each element in the exogenous connectors’ component model has a corresponding method in the interpreter (Figure 6.3) which is invoked each time the given element is encountered in GME. The code inside this method is responsible for processing the given element in the exogenous connectors’ component model.
6.2.2 Writing the Code for the ‘Evaluate_Conditions’ Method

Recall that the ‘Evaluate_Conditions’ method exists in all the fuel operation state charts from the diagrammatic state chart requirements. This method appears at the top most state and determines which sub-states are executed by computing some variables.

Figure 6.4 shows the ‘Evaluate_Conditions’ method for the ‘Manual Refuel’ state chart from the diagrammatic state chart requirements. When attempting to write the code for this in C++ it was important to make a few decisions beforehand. Firstly the code in Figure 6.4 cannot be directly mapped to C++ since it is not language specific; it is mainly pseudocode so it was up to us to convert the code into C++ code.

During the analysis of the diagrammatic state chart requirements documentation we noticed that all the variables (except those that accepted integer values for fuel quantities) used throughout the state charts for every fuel operation only held one of two possible values. For example, in Figure 6.4 if we look at the variable ‘IRP_MAN_VALVES_TIV_LWT’ it can only be assigned a value of either ‘OPEN’ or ‘SHUT’. Currently the code here is testing if the variable has a value of ‘OPEN’. For some variables the boolean type is used. For example, the code in Figure 6.4 tests if the variable ‘PIN_PROGRAMMING_OPT_JETTISON’ has a value of ‘TRUE’ and so the only other possible value is ‘FALSE’. Therefore we felt that it would be ideal to represent all the variables as boolean types since boolean values consist of either ‘true’ or ‘false’ as this would allow us to ensure consistency was maintained throughout the system. However, during the analysis of the interpreter (which is responsible for processing the implementation of the system to ensure it conforms to the exogenous connectors’ component model) we found that all variables had to be represented as integer types.
(either with a value of 1 which represents true or a value of 0 which represents false) due to the choice of the type made by the developers during the development of the tool. So we had to make sure our implementation conformed to this. Therefore when attempting to indicate that a variable will have a value of 'true' this was represented as 1 and a value of 'false' was represented as 0. Figure 6.5 shows the C++ code that was written for the 'Evaluate_Conditions' method in Figure 6.4.

![Method code]

**Figure 6.4:** The 'Evaluate_Conditions' method for the 'Manual Refuel' state chart.

There are several important points to discuss for the code in Figure 6.5 and so we will refer to each part of the code by number.

Number 1 refers to the method signature. Here we have named the method 'eval'. As a convention we decided that all the 'Evaluate_Conditions' methods would have the same name throughout the implementation of the system. Notice the long list of input parameters that have been supplied here. These are all the variables that are being used by the method in Figure 6.4 and if we recall one of the rules of the exogenous connectors’ component model no global variables can exist and so all variables had to be passed via input parameters to this method.

Number 2 refers to an additional parameter defined here called 'COMP_TO_EXECUTE' because recall that the whole purpose of the 'Evaluate_Conditions' method is to compute some variables to determine which sub-component to execute and this variable will hold the value of which component to execute. This is explained more clearly when the code at number 5 is discussed.
Notice the ampersand at the beginning of the 'COMP_TO_EXECUTE' parameter. This indicates that this is a reference variable. What this means is that we are passing the address location of this variable into the method as an input parameter as opposed to its actual value. This is good programming practice to follow if a variable's value is going to be altered during execution, as is the case here.

```cpp
class Fig6_5:
    def __init__(self):
        pass

    def Evaluate_Conditions(self, IRP_MAN_VALVES_TIV_LWT, IRP_MAN_VALVES_TIV_RWT, IRP_MAN_VALVES_TIV_CT, PIN_PROGRAMMING_OPT_JETTISON, RES_EQUIP_STATE_V_J1, RES_EQUIP_STATE_V_J2, RES_EQUIP_STATE_V_D, RES_EQUIP_STATE_V_B, RES_EQUIP_State_V_JL, RES_EQUIP_STATE_V_JR, AFDX_ICP_CMDS_ICP_LWT_XFR_Pb, AFDX_ICP_CMDS_ICP_RWT_XFR_Pb, &COMP_TO_EXECUTE):
        int ANY_TIV_SWITCH_OPEN, MR_ANY_JTSN_VLV_OPEN, MR_ANY_CNCTNGL_VLV_FAILED_OPEN, MR_MAN_XFR_SELCTD, IDLE, IN_PROGRESS, ABORT;
        if(IRP_MAN_VALVES_TIV_LWT == 1 || IRP_MAN_VALVES_TIV_RWT == 1 || IRP_MAN_VALVES_TIV_CT == 1)
            ANY_TIV_SWITCH_OPEN = 1;
        else
            ANY_TIV_SWITCH_OPEN = 0;
        if(PIN_PROGRAMMING_OPT_JETTISON == 1 && (RES_EQUIP_STATE_V_J1 == 1 || RES_EQUIP_STATE_V_J2 == 1))
            MR_ANY_JTSN_VLV_OPEN = 1;
        else
            MR_ANY_JTSN_VLV_OPEN = 0;
        if((RES_EQUIP_STATE_V_D == 1 && RES_EQUIP_STATE_V_B == 0) && (RES_EQUIP_STATE_V_JL == 1 && RES_EQUIP_STATE_V_JR == 0))
            MR_ANY_CNCTNGL_VLV_FAILED_OPEN = 1;
        else
            MR_ANY_CNCTNGL_VLV_FAILED_OPEN = 0;
        if(AFDX_ICP_CMDS_ICP_LWT_XFR_Pb == 1 || AFDX_ICP_CMDS_ICP_RWT_XFR_Pb == 1)
            MR_MAN_XFR_SELCTD = 1;
        else
            MR_MAN_XFR_SELCTD = 0;
        if(MR_ANY_JTSN_VLV_OPEN == 1 || MR_ANY_CNCTNGL_VLV_FAILED_OPEN == 1 || MR_MAN_XFR_SELCTD == 1)
            ABORT = 1;
        else
            ABORT = 0;
        if(IDLE == 1 && IN_PROGRESS == 0 && ABORT == 0)
            IDLE = 1;
        else
            IDLE = 0;
        if(IN_PROGRESS == 1 && ABORT == 0)
            IN_PROGRESS = 1;
        else
            IN_PROGRESS = 0;
        if(IDLE == 0 && IN_PROGRESS == 0 && ABORT == 0)
            COMP_TO_EXECUTE = 1;
        else
            COMP_TO_EXECUTE = 2;
        else
            COMP_TO_EXECUTE = 3;
```
Number 3 refers to the local variables that have been defined which are exactly the same as those in Figure 6.4 (variable names in red text). These local variables are set when the other variables passed as input parameters are computed inside the ‘if-else’ blocks (Number 4).

Number 4 is where the computation of the variables that are passed as input parameters to the method is performed. Notice how several ‘if-else’ blocks are used here to determine the values of the local variables. For example, the first ‘if’ block in Figure 6.5 determines if the variables ‘IRP_MAN_VALVES_TIV_LWT’, ‘IRP_MAN_VALVES_TIV_RWT’ and ‘IRP_MAN_VALVES_TIV_CT’ have a value of 1 (which corresponds to the value of ‘OPEN’ in Figure 6.4 for the first set of statements) and if this is the case then the value of the local variable ‘ANY_TIV_SWITCH_OPEN’ is set to 1 to indicate this is true else it is set to 0 to indicate this is false. It was also important to add ‘else’ blocks to the code here to ensure that the value of the local variables had at least one value (either 1 or 0) because these local variables are used towards the bottom of the code to determine the value of the ‘IDLE’, ‘IN_PROGRESS’ and ‘ABORT’ local variables. You will notice that we have represented the values of the variables used in Figure 6.4 as integer values (either 1 for true or 0 for false) in the ‘if’ statements in Figure 6.5. As mentioned earlier this was necessary since the interpreter was designed to read only integer values. The rest of the ‘if’ statements are similar to the one discussed so do not require further discussion.

Number 5 refers to new code that was added which is not present in the code in Figure 6.4. This code determines which sub-component will be executed in the ‘Manual Refuel Process’ composite component (Figure 6.6) by setting the value of the reference variable ‘COMP_TO_EXECUTE’ specified in the method signature. The value of this reference variable is based on the values of the local variables ‘IDLE’, ‘IN_PROGRESS’ and ‘ABORT’. If the value of the ‘IDLE’ local variable is 1 (which represents true) and the values of the ‘IN_PROGRESS’ and ‘ABORT’ local variables is 0 then the value of the ‘COMP_TO_EXECUTE’ variable is set to 1 which means that the ‘MR Idle’ sub-component will be executed located inside the ‘Manual Refuel Process’ composite component (Figure 6.6). If the local variable ‘IN_PROGRESS’ has a value of 1 and the ‘IDLE’ and ‘ABORT’ local variables have a value of 0 then the ‘COMP_TO_EXECUTE’ variable is set to 2 which means that the ‘MR In_Progress’ sub-component is executed else the ‘COMP_TO_EXECUTE’ variable is set to 3 which means that the ‘MR Abort’ sub-component is executed.

Notice how the reference variable ‘COMP_TO_EXECUTE’ stores an integer value to determine which sub-component is executed from ‘MR Idle’, ‘MR In_Progress’ and ‘MR Abort’. The reason for this is because each sub-component defined in GME has an order number. If we recall the class diagram showing the exogenous connectors component model in Figure 3.5 from Chapter 3 each component has an order number that helps the exogenous connector determine which sub-component to execute. So the ‘COMP_TO_EXECUTE’ variable will be used by the selector connector in the ‘Manual Refuel Process’ composite component to determine which sub-component to execute from ‘MR Idle’, ‘MR In_Progress’ and ‘MR Abort’. Figure 6.6 shows the GME implementation of the ‘Manual Refuel Process’ composite component. The attribute called ‘Order’ indicates the order value of the component. It currently shows a value of ‘1’ because the first sub-component called ‘MR Idle’ has been selected.

It is important to point out that the code in Figure 6.5 was written inside the computation unit located inside the ‘MR Evaluate Conditions’ atomic component. The code is always defined inside the
computation unit and this was done for all the fuel operations where conditions were evaluated or where system state values were updated.

![Image of GME implementation](image)

**Figure 6.6**: Showing the ‘Manual Refuel Process’ composite component implementation in GME.

### 6.2.3 Limiting the Outputs from the ‘Evaluate_Conditions’ Method

When it came to implementing the ‘Evaluate_Conditions’ method (Figure 6.4) in GME, the way the method operated in the state charts requirements documentation could not be mapped directly. Figure 6.7 shows a simplified version of what the inside of the ‘Manual Refuel’ state chart looks like. The ‘Manual Refuel’ state and system state variables have been omitted for ease of readability. Notice how the 3 variables (‘Idle’, ‘In_Progress’ and ‘Abort’) that are computed in the ‘Evaluate_Conditions’ method in Figure 6.4 are output from the method in Figure 6.7. Here they are used to determine which sub-state to execute.

When it came to implementing this in GME we could not directly map this due to the way the interpreter, which processes the implementation, was written. In GME we used a selector connector to connect the 3 sub-components (as can be seen in Figure 6.6) since only one of these sub-components could execute. When the interpreter processes the selector connector, it is programmed to accept only one variable that would help determine which sub-component to execute. For this reason we could not output 3 variables out of the ‘Evaluate_Conditions’ method as shown in Figure 6.7 since this would result in an invalid implementation when it is processed by the interpreter. So to ensure our implementation conformed to the way the interpreter was written we ensured that only one variable was output from the ‘Evaluate_Conditions’ method and this variable is called ‘COMP_TO_EXECUTE’ which was discussed in the previous section. Therefore the value assigned to this variable determines which sub-component to execute. So notice how the state chart in Figure 6.7 determines which sub-state to execute by first executing the ‘Evaluate_Conditions’ method, outputting 3 variables and determining which of these 3 variables is true once the decision node is
encountered. However, in our implementation we determine which sub-component to execute inside the ‘Evaluate_Conditions’ method itself as can be seen in Figure 6.5 (number 5) and we output a single variable ('COMP_TO_EXECUTE') that determines the sub-component to execute. Figure 6.8 shows a diagrammatic representation of this. Notice how even though the implementation has been structured differently, the behaviour is unchanged and the system will still operate as outlined in the requirements documentation. The only difference is instead of outputting 3 variables (Figure 6.7) we output only one variable that determines which sub-component to execute (Figure 6.8). This feature was implemented for every fuel operation where an ‘Evaluate_Conditions’ method was encountered.

**Figure 6.7:** Showing the sub-states and ‘Evaluate_Conditions’ method inside the ‘Manual Refuel’ state chart.

**Figure 6.8:** Showing how one variable determines which sub-component to execute in the ‘Manual Refuel Process’ composite component.
6.2.4 Restructuring Certain Parts of the System

There were occasions during implementation when the structure of a fuel operation for a particular component had to be redesigned due to good system development practice. If you look at Figure 6.4 you will notice that the 4 variables under the boolean variable ‘MR_ANY_TIV_OPEN’ are not implemented in the code in Figure 6.5. The reason for this was because these variables are actually required by the ‘MR_Surge_Protection’ sub-component and the problem was this component is located inside the ‘MR_In_Progress’ composite component (Figure 6.6). Figure 6.9 shows the contents of the ‘MR_In_Progress’ composite component in GME.

We could have easily implemented the 4 variables associated with the boolean variable ‘MR_ANY_TIV_OPEN’ shown in Figure 6.4 inside the code in Figure 6.5 and passed them out of the ‘Evaluate_Conditions’ method as output parameters and then pass them as input parameters into the ‘Manual_Refuel_Process’ composite component so that they could be forwarded to the ‘MR_In_Progress’ sub-component and used by the ‘MR_Surge_Protection’ sub-component. However, this was avoided because due to good software development practice we felt it was more ideal to isolate these variables and place them inside the component that actually requires them which in this case is the ‘MR_Surge_Protection’ sub-component. Recall that a software component is a unit of software that has a well defined purpose and is loosely coupled and highly cohesive. Cohesion describes the interactions within components. The more cohesive a component, the more related the internal parts of the component to each other and to its whole purpose [14]. Therefore, for this reason we felt it was important to isolate these variables and place them inside the component that actually requires them which in this case is where they are used. If we were to implement these variables in the ‘Evaluate_Conditions’ method (Figure 6.5) then the ‘MR_Evaluate_Conditions’ atomic component (Figure 6.8) that contains this method would not be considered highly cohesive since the 4 variables associated with the boolean variable ‘MR_ANY_TIV_OPEN’ in Figure 6.4 would be unrelated to the rest of the code as they are not used by any sub-component except the ‘MR_Surge_Protection’ sub-component, which is located further down the hierarchy of components.

Therefore the system was restructured by adding a new atomic component called ‘SP_Eval_Status’ inside the ‘MR_Surge_Protection’ sub-component and connecting this atomic component with a composite component called ‘SP_Process’ using a pipe connector. Figure 6.10 shows this. The method inside the ‘SP_Eval_Status’ atomic component would perform some computation, similar to the computation performed in the ‘Evaluate_Conditions’ method, by using the 4 variables associated with the boolean variable ‘MR_ANY_TIV_OPEN’ in Figure 6.4 and it will output a variable to determine which atomic component from ‘SP_Open’ and ‘SP_Shut’ to execute (Figure 6.11). These two components are actually located inside the ‘SP_Process’ composite component (Figure 6.10). Notice that because the ‘SP_Eval_Status’ atomic component will output a variable that will help determine which sub-component to execute in the ‘SP_Process’ composite component, we have used a pipe connector to connect these two components because the pipe connector allows values output from one component to be passed as inputs into the next component which is exactly what was required here.
Figure 6.9: Showing the sub-components of the ‘MR In_Progress’ composite component.

Figure 6.10: Showing the contents of the ‘MR_Surge_Protection’ sub-component.
Figure 6.11: Showing the contents of the 'SP_Process' composite component.

The code that determines which sub-component inside the 'SP_Process' composite component (Figure 6.11) will be executed is shown in Figure 6.12. This code is defined in a computation unit inside the 'SP_Eval_Status' atomic component (Figure 6.10).

```cpp
void eval(int RES_EQUIP_STATE_V_RL, int RES_EQUIP_STATE_V_RR, int RES_EQUIP_STATE_V_RC1, int RES_EQUIP_STATE_V_RC2, int &SURGE_PROT_COMP_TO_EXEC)
{
  int MR_ANY_TIV_OPEN;
  if (RES_EQUIP_STATE_V_RL == 1 || RES_EQUIP_STATE_V_RR == 1 || RES_EQUIP_STATE_V_RC1 == 1 || RES_EQUIP_STATE_V_RC2 == 1)
  {
    MR_ANY_TIV_OPEN = 1;
  }
  else
  {
    MR_ANY_TIV_OPEN = 0;
  }
  if (MR_ANY_TIV_OPEN == 1)
  {
    SURGE_PROT_COMP_TO_EXEC = 1;
  }
  else
  {
    SURGE_PROT_COMP_TO_EXEC = 2;
  }
}
```

Figure 6.12: Showing the C++ code for the 'SP_Eval_Status' atomic component.
The code in Figure 6.12 will be explained by referring to each part of the code by number. Number 1 refers to the method signature. All the variables associated with the boolean variable ‘MR_ANY_TIV_OPEN’ in Figure 6.4 are being passed as input parameters into the method here.

Number 2 refers to a reference variable called ‘SURGE_PROT_COMP_TO_EXEC’ which is going to determine which sub-component to execute out of ‘SP_Open’ and ‘SP_Shut’ (Figure 6.11). In GME this variable will be passed out of the ‘SP_Eval_Status’ atomic component as an output parameter and passed into the ‘SP_Process’ composite component as an input parameter (Figure 6.10) to determine which sub-component to execute in Figure 6.11.

Number 3 refers to a local variable being defined called ‘MR_ANY_TIV_OPEN’, exactly the same as the variable in Figure 6.4.

Number 4 refers to the code where the computation is performed. If either of the variables passed in as input parameters to the method have a value of 1 (true) then the ‘MR_ANY_TIV_OPEN’ variable is set to 1 (true) else it is set to 0 (false).

Number 5 refers to the code which is similar to the ending of the code in Figure 6.5. Here if the local variable ‘MR_ANY_TIV_OPEN’ is equivalent to 1 (true) then the ‘SURGE_PROT_COMP_TO_EXEC’ variable is set to 1 which means the ‘SP_Open’ atomic component located inside the ‘SP_Process’ composite component (Figure 6.11) will be executed, since it has an ‘Order’ attribute value of 1, else the ‘SURGE_PROT_COMP_TO_EXEC’ variable is set to 2 which means the ‘SP_Shut’ atomic component will be executed because it has an ‘Order’ attribute value of 2.

Notice how although the structure of the system has been altered the behaviour is unchanged because the system will still operate as outlined in the requirements documentation. The only difference here is that instead of computing the 4 variables associated with the boolean variable ‘MR_ANY_TIV_OPEN’ inside the ‘Evaluate_Conditions’ method (Figure 6.5), they are now computed in the ‘SP_Eval_Status’ atomic component’s method located inside the ‘MR_Surge_Protection’ component (Figure 6.10) and the method that performs this computation is shown in Figure 6.12. Since these 4 variables were only required by the ‘MR_Surge_Protection’ composite component, this justified the need to restructure the system in this manner to ensure our implementation was highly cohesive.

### 6.2.5 Implementing Composite Methods

A composite method is a method that has been defined inside a composite component. It is mainly used by an exogenous connector to determine which parameters to supply to which sub-components and what parameters to output of the composite component. Figure 6.13 shows an example of composite methods for 3 composite components using a sequencer connector.
Figure 6.13: Showing composite methods for composite components.

Figure 6.13 shows a fictitious system just to help explain the idea of composite methods. A real example from the implementation will be shown later.

‘Method A’ is a composite method defined inside composite component ‘A’. Since this component is composed of ‘A1’ and ‘A2’ sub-components, this composite method specifies what input parameters these sub-component’s methods will require. Since a sequencer connector is being used, all sub-components will be executed sequentially. So sub-component ‘A1’ will be executed first and it requires 3 input parameters (P1, P2 and P3) and sub-component ‘A2’ is executed second and it also requires 3 input parameters (P4, P5, and P6). So when the composite method ‘Method A’ is defined, the order of the parameters must be defined in the order in which they are used by the sub-components since the interpreter is designed to pass parameters to sub-components in this way. Notice how ‘Method A’ specifies the parameters in the order in which the sub-components require them.

During the implementation of the fuel operations it was important to ensure that the parameters specified inside the composite methods were ordered correctly based on the order in which the sub-components would require them as discussed above so that the system implementation would be valid. The composite methods for each fuel operation in GME were defined in a bottom-up manner starting at the most lowest level components and progressively moving up to higher level components. The reason we defined them like this was due to simplicity. If you look at Figure 6.13, defining the composite methods by starting at the lower level is much simpler because once you have defined the lower level component’s composite methods the parent component’s composite method will consist of all the lower level component’s parameters. Since a sequencer connector is used, we know that ‘Method A’ must define the ‘A1’ sub-component’s input parameters first since this is the component that will be executed first followed by the input parameters for the ‘A2’ sub-component.
Figure 6.14 shows the GME implementation for the 'MR In_Progress' composite component. It shows how input parameters have been ordered in the composite method 'proc' so that they can be supplied to the sub-components in that order when the interpreter processes this GME implementation. Notice how a sequencer connector has been used. This means that all the sub-components at this level are going to be executed sequentially one after another. The first sub-component (MR_In_Progress_Init) does not require any input parameters as it uses output parameters and outputs them from itself. These output parameters are listed on the right hand side of the composite method 'proc' in Figure 6.14 which means they are passed out of the 'MR In_Progress' composite component.

When composite methods were implemented in GME they were double checked to ensure the order of the parameters were correct since a slight error in the order could result in failure when the interpreter was executed.

*Figure 6.14:* Showing how parameters in GME are ordered in the composite method for the 'MR In_Progress' composite component so they can be supplied in that order to the sub-components.

Only samples of screenshots for the fuel management system implementation were shown in this chapter. The full set of screenshots for the fuel management system implementation in GME can be viewed in appendix C.
During and after the implementation, it was vital to ensure that the fuel management system was tested so that it conformed to the tool (which is composed of the exogenous connectors’ component model and interpreter) supplied to us by the School of Computer Science so that our implementation of the system would be considered valid. Numerous testing techniques were performed which are discussed in this chapter.

### 7.1 Testing Strategy

The testing strategy refers to the plan of how testing was approached. It was developed before implementation began so that it was clear how the system would be tested. The following outlines the different testing techniques used:

- **Unit Testing** - During the development of the system, unit testing was continuously performed on individual atomic components.
- **Integration Testing** - This form of testing was necessary when two or more components were combined (composite component). It was continuously performed during implementation.
- **System Testing** - This involved testing each fuel operation of the system as a whole, once it was fully implemented. Each fuel operation was composed of multiple composite and atomic components and composite methods.
- **Regression Testing** - This form of testing involved checking that no errors were introduced as a result of making changes to certain parts of the system’s structure in GME during implementation. It involved rerunning all the tests for a given composite component to ensure the sub-components still operated correctly.

The testing strategy helped guide the process of testing to ensure that all aspects of the system had been tested thoroughly, which was crucial since the nature of the system was safety-critical.

### 7.2 Test Plan

Before testing could be undertaken, it was important to have a test plan in place which would guide the testing process. The test plan was based on the testing strategy which defined a set of test cases to be carried out and the expected and actual outcomes of the tests. The test plan was created

---

7 A set of conditions under which a tester will determine whether a system meets its specification.
during the design phase once the structure of the system was determined. Table 7.1 shows the structure of the test plan.

<table>
<thead>
<tr>
<th>Test Ref No.</th>
<th>Comment</th>
<th>Test Input</th>
<th>Expected Outcome</th>
<th>Actual Outcome</th>
<th>Actions Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>A unique identifier for the test</td>
<td>A comment explaining the purpose of the test</td>
<td>The actual data to be used in the test</td>
<td>The outcome of the test predicted</td>
<td>The actual result of the test</td>
<td>Any actions taken if the outcome was not as expected</td>
</tr>
</tbody>
</table>

**Table 7.1: Test plan structure.**

### 7.3 How Testing was Performed

During implementation the main testing techniques that were used were unit and integration testing. Each fuel operation was tested in a bottom-up manner starting at the lowest level components and progressively moving up to higher level components. The lowest level components were mainly atomic components that contained a computation unit, invocation connector and a method and the higher level components were composite components that were composed of a combination of atomic and composite components and composite methods including exogenous connectors.

Figure 7.1 shows a diagrammatic representation of how testing was performed. The numbers in the figure indicate the order of testing, where lower level components were tested first before higher level components. Note that Figure 7.1 only shows an example of a system. The actual fuel management system was much larger in terms of the number of components.

---

**Figure 7.1:** Showing how testing was performed in a bottom-up manner [15].
7.4 Simulator Used for Testing

To help test the system during and after implementation, a simulator was used which was part of the tool supplied to us by the School of Computer Science. This simulator accepts a set of input values and generates a set of output values. The input values are mainly the values of the variables specified in the state chart requirements documentation. As mentioned during design a majority of these variables take one of two possible values and so the integer type was used to represent true (1) or false (0). However, in some cases the value may be an integer value specifying the amount of fuel in the aircraft but this was mainly for the 'Automatic Refuel' operation. Either way integer values were passed as inputs to the simulator which represent the values of the parameters for a given composite method in a given component. The output values generated refer to the values of the system state variables. The values output depend on the values passed as inputs. Figure 7.2 shows the GUI of the simulator that was used to test the fuel management system.

![Figure 7.2: GUI of the simulator used to test each aspect of the fuel management system.](image)

Based on Figure 7.2, the method drop down list is where the composite method is selected from. It is important to note that the method selected from here is always a composite method. Every time a component needed testing, it had to be enclosed in a composite component with a composite method so that the necessary inputs can be supplied to the contained components. All input values are passed via the composite methods. Input values can either be entered in the white 'Inputs' text box or they can be imported through a text file by selecting the button 'Import inputs'. The 'Outputs' white text box is where the outputs are generated based on the values input. Once the inputs have been supplied, you simply click the 'Start' button and the interpreter processes the implementation of the system in GME and generates the necessary outputs and displays them in the 'Outputs' text box.

During implementation this simulator was used continuously to determine if the structure of our fuel management system conformed to the exogenous connectors' component model and the interpreter because if the simulator successfully generated outputs then this meant the structure of our
implementation was valid. If the structure was invalid an error message dialog would appear which meant that the structure required correcting.

7.5 Testing the Fuel Management System

The number of tests that were performed overall was relatively large and so only examples of the tests will be shown here for each testing technique.

7.5.1 Unit Testing

Unit testing involved performing tests on every single atomic component for each fuel operation. This form of testing was performed constantly during implementation. Table 7.2 shows an example of 2 unit tests performed for the ‘MR_LWT_Idle’ and ‘MR_LWT_Active’ atomic components for the ‘Manual Refuel’ operation.

<table>
<thead>
<tr>
<th>Test Ref No.</th>
<th>Comment</th>
<th>Test Input</th>
<th>Expected Outcome</th>
<th>Actual Outcome</th>
<th>Actions Taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>U1</td>
<td>Check that the 'MR_LWT_Idle' atomic component outputs the correct system state value.</td>
<td>Component name: MR_LWT_Idle Input Value: 1</td>
<td>Output value = 0 for system state variable ‘SS_MR_LWT’</td>
<td>Test Passed</td>
<td>None</td>
</tr>
<tr>
<td>U2</td>
<td>Check that the 'MR_LWT_Active' atomic component outputs the correct system state value.</td>
<td>Component name: MR_LWT_Active Input Value: 0</td>
<td>Output value = 1 for system state variable ‘SS_MR_LWT’</td>
<td>Test Passed</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 7.2: Examples of 2 unit tests performed on the 'MR_LWT_Idle' and 'MR_LWT_Active' atomic components.

7.5.2 Integration Testing

Integration testing involved testing composite components that were composed of a combination of atomic and composite components. Once the lower level atomic components were tested, the parent component of these sub-components was tested since this parent component is composed of these sub-components. Table 7.3 shows an example of the integration tests performed.
<table>
<thead>
<tr>
<th>Test Ref No.</th>
<th>Comment</th>
<th>Test Input</th>
<th>Expected Outcome</th>
<th>Actual Outcome</th>
<th>Actions Taken</th>
</tr>
</thead>
</table>
| I1 | Check that when the status of the LWT is evaluated the correct sub-component is executed and the correct system state value is output. | Component name: MR_TK_LWT  
Input Values: 1, 0, 1 | MR_LWT_Eval_Status component outputs value 2.  
MR_LWT_Process component uses this value to execute sub-component 2 (MR_LWT_Active)  
Output value = 1 for system state variable ‘SS_MR_LWT’ | Test Passed  
MR_LWT_Eval_Status component outputs value 2.  
MR_LWT_Process component executes sub-component 2 (MR_LWT_Active)  
Value of 1 was output for system state variable ‘SS_MR_LWT’ | None |
| I2 | Check that the ‘MR_In_Progress’ composite component executes correctly and all system state values output are correct. | Component name: MR_In_Progress  
Input Values: 1, 0, 1, 0, 0, 0, 1, 1, 1, 0 | All sub-components are executed in sequence.  
Output 6 values = 0, 1, 0, 1, 1, 0, 1 | Test Passed  
All sub-components were executed in sequence.  
6 values were output = 0, 1, 0, 1, 1, 0, 1 | None |

**Table 7.3:** Examples of 2 integration tests performed on the ‘MR_TK_LWT’ and ‘MR_In_Progress’ composite components.

Figure 7.3 shows the integration test ‘I2’ being performed from Table 7.3 using the simulator. The input values from Table 7.3 are shown in the ‘Inputs’ text box in Figure 7.3 and the generated outputs are shown in the ‘Outputs’ text box. It is important to note that the output values generated are for the output parameters that appear on the right-hand side of the composite method ‘proc’ in the ‘MR_In_Progress’ composite component (Figure 7.4). These output values are passed from the components, once set, to the composite method and out as output parameters. Figure 7.4 shows the GME implementation of the ‘MR_In_Progress’ composite component. The output parameters shown in the ‘proc’ composite method correspond to the output values shown in the ‘Outputs’ text box of the simulator in Figure 7.3. It is difficult to visualise this using the simulator since the simulator only outputs the integer values and not the names of the output parameters. By looking at Figure 7.3 and 7.4 notice how the ‘SS_MR_IDLE’ and ‘SS_MR_ABORT’ output parameter values are 0 but the ‘SS_MR_IN_PROGRESS’ parameter value is 1 because the ‘MR_In_Progress’ composite component was executed and so this system state variable was set to true (1). The rest of the output parameter values were determined based on the values we input.
Figure 7.3: Using the simulator to perform the integration test ‘I2’ from Table 7.3.

Figure 7.4: GME implementation of the ‘MR In_Progress’ composite component.
7.5.3 System Testing

System testing involved performing tests on the system as a whole. Since each fuel operation is independent of the other we tested each fuel operation here. System testing was performed once each fuel operation was fully implemented and then again when all the fuel operations were fully implemented. Table 7.4 shows an example of a system test performed on the 'Manual Refuel Op' composite component which corresponds to the 'Manual Refuel' operation in the state chart requirements documentation.

<table>
<thead>
<tr>
<th>Test Ref No.</th>
<th>Comment</th>
<th>Test Input</th>
<th>Expected Outcome</th>
<th>Actual Outcome</th>
<th>Actions Taken</th>
</tr>
</thead>
</table>
| S1           | Check that the 'Manual Refuel Op' composite component fully works as expected when given values are input into the simulator. | **Component name:** Manual Refuel Op  
**Input Values:** 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1 | The 'Manual Refuel Op' composite component should execute successfully without any error.  
Output 14 values = 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1 | **Test Passed**  
The 'Manual Refuel Op' composite component executed successfully without any error.  
14 values were output = 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1 | None |

Table 7.4: An example of a system test performed on the 'Manual Refuel Op' composite component.

7.5.4 Regression Testing

Regression testing was performed rigorously throughout development by rerunning all the tests, even as a small change was made to the fuel management system in GME, to ensure that a slight change did not cause the system implementation to fail when it was tested using the simulator and interpreter.

It is important to note that the examples of tests shown above were only a sample of the tests performed. The full test plan showing every single test performed could not be shown due to space limitations. The full set of test cases can be viewed in Appendix D. Please note that these test cases only include the tests for the 'Manual Refuel' operation because all the other tests for the other fuel operations were similar to this.
This chapter involves discussing relevant issues that were considered interesting which arose during the development of the fuel management system. It focuses on the additional work that was carried out after the implementation of the fuel management system.

8.1 Limitation of the Selector Connector Code in the Interpreter

During the analysis of the interpreter (which is responsible for processing the fuel management system implementation in GME to ensure it conforms to the exogenous connectors’ component model and also allows the system to be tested via a simulator) we felt that the code that was responsible for processing a selector connector in GME had its limitations. This selector connector was programmed to accept only a single value to determine which sub-component to execute. However, the state charts in the requirements documentation are modelled differently to the way the selector connector behaves. Figure 8.1 shows the ‘Manual Refuel’ state chart from the state chart requirements documentation. Once the ‘Evaluate_Conditions’ method is executed 3 variables are output and a single sub-state from ‘Idle’, ‘In_Progress’ or ‘Abort’ is executed depending on whichever of these variables has a value of ‘true’. However, when it came to mapping the output of these 3 variables from the ‘Evaluate_Conditions’ method in GME during implementation so they could be used to determine which sub-component to execute from ‘MR_Idle’, ‘MR_In_Progress’ and ‘MR_Abort’, it could not be mapped directly since the interpreter only allowed one value to be passed to the selector connector to determine the sub-component to execute in GME.

Therefore, after the fuel management system had been fully implemented in GME we felt that it would be ideal to improve the way the selector connector behaved and we attempted to improve the implementation for the selector connector so that it would behave similarly to the way the state charts are represented in the requirements documentation. This would mean that we would be able to map the state chart shown in Figure 8.1 directly, in terms of the 3 variables output to determine which sub-component to execute, when implementing the system in GME.
Figure 8.1: State chart for the 'Manual Refuel' operation

8.2 Improving the Existing Selector Connector Code

Figure 8.2 shows the code written for the current selector connector. Note that this is the existing code that is used in the interpreter to process the selector connector. Figure 8.3 shows our improved version of the code for the selector connector. Note that in both figures only the relevant code is being shown, the code that was irrelevant has been omitted.
Figure 8.2: The C++ selector connector code for the interpreter.

Only the important parts of the code will be discussed here since the code is self explanatory as it has been commented. The code in Figure 8.2 will be explained by referring to it by number. Number 1 refers to the first input parameter (which is the value passed from the 'Evaluate_Conditions' method to help determine which sub-component to execute) which is retrieved and stored in a variable called 'param'. Number 2 retrieves each sub-component connected to the selector connector and the getOrder() method is invoked to retrieve each sub-component's 'Order' value (recall that this is an attribute given to each component in GME which determines the order of the component). This 'Order' value is compared against the value of the variable 'param' and if they match then this is the sub-component that is executed.

The code in Figure 8.3 shows the improved version of the selector connector code to reflect the way the state charts are represented in the requirements documentation, which will mean that they can be mapped directly if this code is used instead of the code in Figure 8.2.
Figure 8.3: Improved C++ selector connector code.
The code in Figure 8.3 will be explained by referring to each part by number. In number 1 we first get all the sub-components connected to the selector connector and order all these sub-components based on the value of their ‘Order’ attribute in GME and then we count the total number of sub-components connected to this selector connector. In number 2 the first set of input parameters are retrieved based on the total number of sub-components. For example, if 3 sub-components exist then the first 3 input parameters are retrieved. These 3 input parameters correspond to the 3 variables output from the ‘Evaluate_Conditions’ method which are shown on the transition lines near the decision node in Figure 8.1. In number 3 we check which input parameter has a value of 1 (which represents true) and get the index position of this input parameter in the vector and assign this index position to a variable. In number 4 this index position is then used to locate the sub-component that needs to be executed. The index position of the input parameter will have the same value as the order value of the sub-component to be executed which is why we can use it to locate the sub-component to be executed.

Notice how this code reflects exactly what is happening in the state chart in Figure 8.1. The decision node in Figure 8.1 checks which variables out of the 3 have a value of ‘true’ and then it executes that sub-state. In our code (Figure 8.3) we check which input parameter (which corresponds to the variables on the transition lines in Figure 8.1) has a value of 1 (true) and we then get the index position of this input parameter to help determine which sub-component to execute as the ‘Order’ value for the sub-component will be the same as the index position of the input parameter.

However, if we look at the code in Figure 8.2 for how the selector connector currently works, only one variable is used to determine which sub-component needs to be executed. As a result of this the developers have to write additional code inside the ‘Evaluate_Conditions’ method to help determine which sub-component to execute. If we use our code in Figure 8.3 for the selector connector instead, no additional code needs to be written in the ‘Evaluate_Conditions’ method because the number of variables output is based on the number of sub-components and each of these variables is checked to determine which has a value of 1 (true) and this helps determine which sub-component to execute. So now developers can map the state chart in Figure 8.1 directly when creating an implementation in GME and this reduces development time. Note that although the code in Figure 8.3 is much longer in comparison to the code in Figure 8.2, this is not a problem since we have provided a way of allowing state charts to be mapped directly in GME wherever a selector connector is required.

It is important to note that the improved code for the selector connector in Figure 8.3 was not used in our implementation of the fuel system. Instead our system was based on the code in Figure 8.2. The reason for this was that the code for the selector connector was not improved until we had implemented the entire fuel management system. Although we realised during implementation that it could be improved, we left this improvement until the end since the focus of the project was to design and implement a fuel management system using the exogenous connectors’ component model and interpreter and so it was important to ensure the entire fuel management system was fully implemented before any additional work was considered.
This chapter focuses on evaluating the project as a whole, examining whether the implemented fuel management system successfully met the objective originally outlined, the limitations of the exogenous connectors’ component model, what challenges were encountered during the development and any possible future work for the system.

9.1 Achievements

The aim of the project was to design and implement a fuel management system for the upcoming Airbus A350 XWB commercial airliner by using a component model that is based on exogenous connectors. Existing component models such as Microsoft's COM+ model have a common disadvantage in that when systems are developed using these component models they mix computation and control (method calls) inside the components. As a result of this components are not loosely coupled since direct dependencies exist between components due to the way control has been mixed with computation. However, the component model based on exogenous connectors is unique since it uses exogenous connectors to connect all software components and inside these connectors is where all the control (method calls) is initiated and coordinated [7]. So components themselves are not responsible for invoking the methods of other components, instead this is performed by the exogenous connectors themselves. Thus, these exogenous connectors encapsulate control entirely and this makes components truly independent as they are loosely coupled since no direct dependencies exist between them.

Overall, a successful implementation of a fuel management system was developed which was based on the exogenous connectors’ component model that successfully met the objective originally outlined. A fully complete fuel management system was implemented which consisted of 6 fully functional ground fuel operations (Automatic Refuel, Manual Refuel, Defuel, Ground Transfer, Shut-Off Test and Off). The implemented system conformed to the rules of both the exogenous connectors’ component model and interpreter and this was confirmed by testing the system using the simulator. In addition to this, the requirements of Airbus were fully met since the system implementation behaved exactly as outlined in the requirements documentation even though the system was slightly different structurally (due to the exogenous connectors’ component model). Overall we managed to develop a fuel management system that not only met the requirements of Airbus but was also based on the exogenous connectors’ component model which proved that a successful system could be developed using this component model. As a result we were able to clearly demonstrate that component-based software engineering can effectively be applied to design and implement a fuel management system by using the exogenous connectors’ component model.
9.2 Limitations of the Component model

Although a successful implementation of the fuel management system was developed using the exogenous connectors' component model, there were a few limitations identified with this component model during the design and implementation stages.

Firstly, the exogenous connectors (sequencer, pipe and selector) did not meet our needs in some cases during design and implementation. When implementing the fuel management system based on Airbus's requirements documentation, we found that there were cases where multiple states could be executed concurrently. However, when it came to mapping this in our implementation in GME there was no exogenous connector that supported concurrent execution of components. As a result we had to use a sequencer connector in such situations that would execute all the components in sequence but not concurrently which was sufficient in this case as all components would be executed eventually.

A limitation was also found with the way the interpreter dealt with parameter values in composite methods. For example, if a composite method were to output 3 parameters (a, b, c) then the developer would have to ensure that this order was maintained throughout all the higher level composite methods as the values of these output parameters are passed upwards in the defined order. So, for example, if composite method ‘Z’ output 3 parameters (a, b, c) and the developer defined them in this order, then for the composite method that is defined at the level above this (called composite method ‘X’), the developer has to ensure these 3 output parameters are defined in this same order (a, b, c) because the interpreter is not designed to read the names of the parameters to determine which value is assigned to which output parameter, instead the interpreter just passes the values of outputs to parameters in the order in which they are defined in the composite method. The problem with this is if the developer passes 3 output values in the order ‘c, b, a’ the interpreter will assign these values to the output parameters ‘a, b, c’ respectively even though this is incorrect because the last value of ‘a’ should be assigned to the first variable ‘a’ since this value corresponds to the variable ‘a’. However, the interpreter is not designed to check that the correct value is assigned to the correct output parameter. This is left up to the developer to check. Therefore the developer has to ensure that all parameters are ordered correctly at all levels since an incorrect order will mean incorrect values will be output for certain parameters.

9.3 Challenges

Although the main objective of the project was met successfully, there were many challenges faced prior to and during development.

Since we were new to the GME modelling language it was a huge challenge to learn this modelling language since the fuel management system needed to be implemented using this toolkit. Therefore, prior to development it was important to ensure that the right learning resources were used to learn and understand the environment so that our ability to develop the fuel management system would be easy without introducing any errors. This meant accessing online tutorials from the main GME...
A website which offered two types of tutorials: short tutorial lessons and long tutorial lessons [21]. The short tutorial lessons were used first to get familiar with the basics of the GME environment and the long tutorial lessons were followed later once the GME tutorial was familiar to us, thus allowing us to grasp the advanced features.

Our background in programming was mainly Java. We had no prior knowledge of the C/C++ programming languages and therefore we had to gain this knowledge relatively quickly in a short period of time because the interpreter for the exogenous connectors' component model that is responsible for processing the implementation of systems in GME was written in these programming languages and before we could understand the interpreter we had to become familiar with these programming languages. Although this was a huge challenge, since it required more time to be invested in learning entirely new programming languages, this was extremely motivating since new programming languages helped broaden our programming skills and knowledge and this would be invaluable in industry. As for learning the new programming languages C and C++, it was important to find resources of learning that were not too cumbersome since we only had a limited time to learn new programming languages. Therefore our focus was on finding books on C and C++ programming that were relatively small in size but had enough content to allow us to gain a good understanding of the language. During research it was found that the most suitable book available which would allow us to gain a good understanding of C in a short period of time was a book called "The C Programming Language" [22] and for C++ the recommended book by many developers was "Accelerated C++: Practical Programming by Example" [23]. Both these books covered the basics and advanced concepts of C and C++ which gave us enough knowledge to understand the interpreter clearly and implement a good implementation of the fuel management system. Both these books were used prior to and during implementation to aid the development process.

In addition to this we were also required to invest a large amount of time in understanding the state charts in Airbus's state charts requirements documentation since the implementation of the system was based entirely on these state charts. So it was essential to understand the notation used for these state charts to ensure the requirements were captured and implemented accurately in the implementation. The basics of state chart diagrams was acquired using a book called "UML Distilled: A brief guide to the standard modelling language" [24]. This book was helpful in that it covered the basics and advanced concepts of state charts in several pages.

Finally, implementing a complete fuel management system in GME based on the requirements of Airbus and also ensuring that the implementation conformed to the exogenous connectors' component model was a huge challenge in itself due to the development time constraint. As a result this helped improve our project management skills which will be valuable for the future.

9.4 Future Work

Although a successful implementation of a fuel management system was developed, there are many ways in which the system could have been improved. These improvements not only refer to our implementation of the fuel management system in GME but also include the tool (exogenous
connectors’ component model and interpreter) that was supplied to us by the School of Computer Science.

Firstly, we could have written our own interpreter to process the implementation in GME. For our implementation we used an interpreter that was defined by a colleague at the School of Computer Science. In Chapter 8 (Implementation Discussion) we discussed how the selector connector could have been improved and we implemented an improved version of this connector although it was not used due to the limited development time. Future works could include embedding this selector connector in the existing interpreter. This would allow the state charts in the requirements documentation to be mapped directly to the implementation in GME, reducing development time.

The concurrent connector that was mentioned in Chapter 5 (Fuel Management System Design) could be implemented in the exogenous connectors’ component model and interpreter so that it can be used in situations where components need to be executed concurrently. As a result, the usefulness of the exogenous connectors’ component model would be increased.

The graphical user interface (GUI) of the simulator that was used to test the fuel management system in GME could be improved. As we mentioned in Chapter 7 (Fuel Management System Testing), the simulator had its limitations since the names of the input and output parameters were not displayed when the input and output values were displayed in the text boxes. It might be ideal to add the names of the input and output parameters to the values in the text boxes so that the tester is aware of which value they are looking at.

During the analysis of the requirements documentation, we found that some variables in the state charts requirements documentation we actually sensors, although in the state charts requirements documentation they were represented as variables. It might be more ideal to represent these sensors as atomic components and have them pass the variables that store the values from these sensors as output parameters and back into the next components that require them as input parameters. Due to the development time constraints it was not possible to represent the sensors as atomic components in our implementation.

9.5 Final Remarks

The opportunity to develop a complete fuel management system using an exogenous connectors’ component model was extremely motivational since this system was based on real life requirements for an actual aircraft manufacturer called Airbus. The project also helped us realise how effective component-based software engineering is and how it allows you to develop systems using components and also how the reuse of software components can save considerable development time. This is an important factor to take into account when developing avionics software since the major problem in avionics software development is the prolonged development time which can span many years because software has originally been developed from scratch.
Many lessons were learnt throughout development such as the importance of preserving the privacy of confidential data during development and also learning that in reality there is always a time constraint when developing systems and even though the fuel management system was fully completed there were many additional add-ons that could have been added to improve our system and also the interpreter that was used to implement the system in GME.

Since this was our first opportunity in developing a system of a safety-critical nature it has given us an insight into how these types of systems differ from other kinds of systems such as web applications or desktop applications. Due to the safety-critical nature of these systems, more emphasis has to be placed on ensuring that the design and implementation of the system accurately reflects the requirements and that the system is verified vigorously during testing.
References


   http://chess.eecs.berkeley.edu/hcssas/papers/Sha%20Avionics%202.0%20doc.pdf [Accessed 12/03/10].

   the safety of air navigation. [Online].
   http://www.eurocontrol.int/oca/gallery/content/public/docs/OATA-P2-D4.2.11-03-01%20Study%20Report%20on%20Avionics%20Systems.pdf [Accessed 12/03/10].


   Development, Chapter 19 – Component-Based Software Engineering, Addison Wesley.

6. ‘LynxOS-178 RSC (Reusable Software Components)’, (2006), [Online].
   http://www.lynuxworks.com/rsc/lynxos-178-rsc.pdf [Accessed 14/03/10].


8. Image showing the architecture of the Airbus A350 XWB (publicly available), [Online].


13. Image showing the schematics of a general aircraft's engine, [Online].


18. 'Overview of Generic Modelling Environment (GME)', [Online].
   http://www.isis.vanderbilt.edu/Projects/gme/ [Accessed 2503'10].


21. ‘Introduction to GME Tutorials’ [Online].
   http://w3.isis.vanderbilt.edu/Projects/gme/Tutorials/ [Accessed 25/03/10].


Appendices

Appendix A – State Charts from the Requirements Documentation

Manual Refuel Operation State Charts
Method name = Evaluate_Conditions

Boolean ANY_TIV_SWITCH_OPEN =
IRP_MAN_VALVES_TIV_LWT == OPEN OR
IRP_MAN_VALVES_TIV_RWT == OPEN OR
IRP_MAN_VALVES_TIV_CT == OPEN;

Boolean MR_ANY_TIV_OPEN =
RES_EQUIP_STATE_V_RL == OPEN OR
RES_EQUIP_STATE_V_RR == OPEN OR
RES_EQUIP_STATE_V_R1 == OPEN OR
RES_EQUIP_STATE_V_RC2 == OPEN;

Boolean MR_ANY_JTSN_VLV_OPEN =
PIN_PROGRAMMING_OPT_JETTISON == TRUE AND (RES_EQUIP_STATE_V_J1 == OPEN OR RES_EQUIP_STATE_V_J2 == OPEN);

Boolean MR_ANY_CNNECTNG_VLV_FAILED_OPEN = ...
(RES_EQUIP_STATE_V_D == OPEN AND RES_EQUIP_STATUS_V_D == FAILED) OR
(PIN_PROGRAMMING_OPT_JETTISON == TRUE AND
((RES_EQUIP_STATE_V_B == OPEN AND RES_EQUIP_STATUS_V_B == FAILED) OR
(RES_EQUIP_STATE_V_JL == OPEN AND RES_EQUIP_STATUS_V_JL == FAILED) OR
(RES_EQUIP_STATE_V_JR == OPEN AND RES_EQUIP_STATUS_V_JR == FAILED)));

Boolean MR_MAN_XFR_SELCTD =
AFDX_ICP_CMDS_ICP_LWT_XFR_PB == ON OR AFDX_ICP_CMDS_ICP_RWT_XFR_PB == ON;

Boolean Abort =
MR_ANY_JTSN_VLV_OPEN OR MR_ANY_CNNECTNG_VLV_FAILED_OPEN OR
MR_MAN_XFR_SELCTD;

Boolean Idle = ANY_TIV_SWITCH_OPEN == FALSE AND
MANUAL_REFUEL_ABORT = FALSE;

Boolean In_Progress =
ANY_TIV_SWITCH_OPEN == TRUE AND MANUAL_REFUEL_ABORT = FALSE;
}
On entry to state:
Initialise Variables

Action: Initialise Variables
SS_MR_IDLE = false;
SS_MR_IN_PROGRESS = true;
SS_MR_ABORT = false;

Surge Protection

<table>
<thead>
<tr>
<th>SHUT</th>
<th>OPEN</th>
</tr>
</thead>
<tbody>
<tr>
<td>[MR_ANY_TIV_OPEN == true]</td>
<td>[SS_MR_SURGE = false;]</td>
</tr>
<tr>
<td>[SS_MR_SURGE = true;]</td>
<td>[SS_MR_SURGE = true;]</td>
</tr>
</tbody>
</table>

In_Progress

MR_TK_LWT

On entry to state:
evaluate_status();

During the state is active:
evaluate_status();

Method: Evaluate_status

Boolean ACTIVE =
(IRP_MAN_VALVES_TIV_LWT AND
(HIGH_LEVEL_STATE_TK_LWT == NOT_HIGH AND
HIGH_LEVEL_STATUS_TK_LWT == NORMAL) OR
HIGH_LEVEL_STATUS_TK_LWT == FAILED));

Boolean IDLE =
(IRP_MAN_VALVES_TIV_LWT == SHUT OR
HIGH_LEVEL_STATE_TK_LWT == HIGH AND
HIGH_LEVEL_STATUS_TK_LWT == NORMAL);

MR_TK_RWT

On entry to state:
evaluate_status();

During the state is active:
evaluate_status();

Method: Evaluate_status

Boolean ACTIVE =
(IRP_MAN_VALVES_TIV_RWT AND
(HIGH_LEVEL_STATE_TK_RWT == NOT_HIGH AND
HIGH_LEVEL_STATUS_TK_RWT == NORMAL) OR
HIGH_LEVEL_STATUS_TK_RWT == FAILED));

Boolean IDLE =
(IRP_MAN_VALVES_TIV_RWT == SHUT OR
HIGH_LEVEL_STATE_TK_RWT == HIGH AND
HIGH_LEVEL_STATUS_TK_RWT == NORMAL);
On entry to state:
evaluate_status();

During the state is active:
evaluate_status();

Method: Evaluate_status

Boolean ACTIVE = 
(IRP_MAN_VALVES_TIV_CT AND 
(HIGH_LEVEL_STATE_TK_CT == NOT_HIGH AND 
HIGH_LEVEL_STATUS_TK_CT == NORMAL) OR 
HIGH_LEVEL_STATUS_TK_CT == FAILED));

Boolean IDLE = 
(IRP_MAN_VALVES_TIV_CT == SHUT OR 
(HIGH_LEVEL_STATE_TK_CT == HIGH AND 
HIGH_LEVEL_STATUS_TK_CT == NORMAL));

Idle
On entry to state:
SS_MR_CT = false;

Active
On entry to state:
SS_MR_CT = true;

Please note that colour was only used for the ‘Manual Refuel’ state charts to guide the reader during the process of mapping these state charts to the hierarchical tree structure designs, which was discussed in the body of the report. The remaining state charts are in black and white.
**Automatic Refuel Operation State Charts**

**Automatic Refuel**

**State: On entry to state:**
- Evaluate_Failures();

**During the state is active:**
- Evaluate_Failures();

**Action:**
- Initialise Variables:
  - SS_AR_ACTIVE = true;
  - SS_MR_ACTIVE = false;
  - SS_DF_ACTIVE = false;
  - SS_GT_ACTIVE = false;
  - SS_SOT_ACTIVE = false;
  - SS_MODE_OFF = false;

**Method:**
- Evaluate_Failures

**Automatic Refuel Mode**

**State: On entry to state:**
- Evaluate_Conditions();

**During the state is active:**
- Evaluate_Conditions();

**Method:**
- Evaluate_Conditions

**_IDLE**

**State: On entry to state:**
- SS_AR_IDLE = true;
- SS_AR_IN_PROGRESS = false;
- SS_AR_ABORT = false;
- SS_AR_FUEL_UPLIFT_CONFIRMATION = false;
- SS_AR_FUEL_UPLIFT_COMPARISON = false;
- SS_AR_COMPLETE = false;

**FUEL_UPLIFT_CONFIRMATION**

**State: On entry to state:**
- SS_AR_IDLE = false;
- SS_AR_IN_PROGRESS = false;
- SS_AR_ABORT = false;
- SS_AR_FUEL_UPLIFT_CONFIRMATION = true;
- SS_AR_FUEL_UPLIFT_COMPARISON = false;
- SS_AR_COMPLETE = false;

**FUEL_UPLIFT_COMPLETE**

**State: On entry to state:**
- SS_AR_IDLE = true;
- SS_AR_IN_PROGRESS = false;
- SS_AR_ABORT = false;
- SS_AR_FUEL_UPLIFT_CONFIRMATION = false;
- SS_AR_FUEL_UPLIFT_COMPARISON = false;
- SS_AR_COMPLETE = true;

**Abort**

**State: On entry to state:**
- Evaluate_Failures();

**During the state is active:**
- Evaluate_Failures();

**Method:**
- Evaluate_Failures

**Method:**
- Evaluate_Conditions

**Boolean AR_ANY_TANK_INLET_VALVE_OPEN = RES_EQUIP_STATE_V_RL == OPEN OR RES_EQUIP_STATE_V_RR == OPEN OR RES_EQUIP_STATE_V_RC1 == OPEN OR RES_EQUIP_STATE_V_RC2 == OPEN;**

**Boolean ALL_TANK_INLET_VALVES_SHUT = RES_EQUIP_STATE_V_RL == SHUT AND RES_EQUIP_STATE_V_RR == SHUT AND RES_EQUIP_STATE_V_RC1 == SHUT AND RES_EQUIP_STATE_V_RC2 == SHUT;**

**Boolean ALL_TANK_INLET_VALVES_CMD_SHUT = SS_AR_LWT == FALSE AND SS_AR_RWT == FALSE AND SS_AR_CT == FALSE;**

**Boolean FOB_IN_TOLERANCE = (FOB >= FQMS_PFQ - FOB_ACCURACY) AND (FOB <= FQMS_PFQ + FOB_ACCURACY);**

**Boolean FQMS_PFQ_LOWER_THRESHOLD = (FQMS_PFQ - FOB_ACCURACY - THR_AUTO_REFUEL);**

**Boolean AUTO_REFUEL_ABORT == false;**

**Boolean AUTO_REFUEL_ABORT == true;**
Method name = Evaluate_Failures
{
    Boolean LWT_TARGETED = TARGET_TK_LWT > MASS_TK_LWT;
    Boolean RWT_TARGETED = TARGET_TK_RWT > MASS_TK_RWT;
    Boolean CT_TARGETED = TARGET_TK_CT > MASS_TK_CT;

    Boolean ALL_TANKS_HIGH_LEVEL =
        HIGH_LEVEL_STATE_TK_LWT == HIGH AND HIGH_LEVEL_STATE_TK_RWT == HIGH AND HIGH_LEVEL_STATE_TK_CT == HIGH;

    Boolean UNATTAINABLE_PFQ =
        ALL_TANKS_HIGH_LEVEL AND FQMS_PFQ > (FOB + FOB_ACCURACY);

    Boolean ANY_TANK_FQI_FAILED =
        FQI_STATUS_TK_LWT == FAILED OR FQI_STATUS_TK_RWT == FAILED OR FQI_STATUS_TK_CT == FAILED;

    Boolean TARGETED_HIGH_LEVEL_FAILED =
        LWT_TARGETED AND (HIGH_LEVEL_STATUS_TK_LWT == FAILED) OR
        RWT_TARGETED AND (HIGH_LEVEL_STATUS_TK_RWT == FAILED) OR
        CT_TARGETED AND (HIGH_LEVEL_STATUS_TK_CT == FAILED);

    Boolean TARGETED_TIV_FAILED_SHUT =
        LWT_TARGETED AND (RES_EQUIP_STATE_V_RL == SHUT) AND (RES_EQUIP_STATUS_V_RL == FAILED) OR
        RWT_TARGETED AND (RES_EQUIP_STATE_V_RR == SHUT) AND (RES_EQUIP_STATUS_V_RR == FAILED) OR
        CT_TARGETED AND (RES_EQUIP_STATE_V_RC1 == SHUT AND RES_EQUIP_STATUS_V_RC1 == FAILED) AND
        (RES_EQUIP_STATE_V_RC2 == SHUT AND RES_EQUIP_STATUS_V_RC2 == FAILED);

    Boolean ANY_TIV_FAILED_OPEN =
        (RES_EQUIP_STATE_V_RL == OPEN AND RES_EQUIP_STATUS_V_RL == FAILED) OR
        (RES_EQUIP_STATE_V_RR == OPEN AND RES_EQUIP_STATUS_V_RR == FAILED) OR
        (RES_EQUIP_STATE_V_RC1 == OPEN AND RES_EQUIP_STATUS_V_RC1 == FAILED) OR
        (RES_EQUIP_STATE_V_RC2 == OPEN AND RES_EQUIP_STATUS_V_RC2 == FAILED);

    Boolean ANY_JETTISON_VALVE_OPEN =
        PIN_PROGRAMMING_OPT_JETTISON == TRUE AND
        (RES_EQUIP_STATE_V_J1 == OPEN OR RES_EQUIP_STATE_V_J2 == OPEN);

    Boolean ANY_CONNECTING_VALVE_FAILED_OPEN =
        (RES_EQUIP_STATE_V_D == OPEN AND RES_EQUIP_STATUS_V_D == FAILED) OR
        (RES_EQUIP_STATE_V_JL == OPEN AND RES_EQUIP_STATUS_V_JL == FAILED) OR
        (RES_EQUIP_STATE_V_JR == OPEN AND RES_EQUIP_STATUS_V_JR == FAILED);

    Boolean MANUAL_TRANSFER_SELECTED =
        AFDX_ICP_CMDS_ICP_LWT_XFR_PB == ON OR AFDX_ICP_CMDS_ICP_RWT_XFR_PB == ON;

    Boolean AR_ABORTS_PRESENT =
        (UNATTAINABLE_PFQ OR ANY_TANK_FQI_FAILED OR OVERFLOW_CONDITION OR TARGETED_HIGH_LEVEL_FAILED OR
        TARGETED_TIV_FAILED_SHUT OR ANY_TIV_FAILED_OPEN OR ANY_JETTISON_VALVE_OPEN OR
        ANY_CONNECTING_VALVE_FAILED_OPEN OR MANUAL_TRANSFER_SELECTED));

    Boolean COMPLETED_BELOW_TOLERANCE =
        SS_AR_IN_PROGRESS AND
        (FQMS_PFQ < (FOB - FOB_ACCURACY)) AND AUTO_TOP_UP == TRUE;

    Boolean COMPLETED_ABOVE_TOLERANCE =
        SS_AR_IN_PROGRESS AND
        (FQMS_PFQ > (FOB + FOB_ACCURACY));

    Boolean ANY_TANK_MASS_GREATER_THAN_TARGET =
        SS_AR_IN_PROGRESS AND
        (MASS_TK_LWT > TARGET_TK_LWT + LWT_TARGET_TOLERANCE OR
        MASS_TK_RWT > TARGET_TK_RWT + RWT_TARGET_TOLERANCE OR
        MASS_TK_CT > TARGET_TK_CT + CT_TARGET_TOLERANCE);

    Boolean AUTO_REFUEL_ABORT =
        AR_ABORTS_PRESENT OR COMPLETED_BELOW_TOLERANCE OR COMPLETED_ABOVE_TOLERANCE OR
        ANY_TANK_MASS_GREATER_THAN_TARGET OR AR_ABORT_LATCH;
}
In_Progress

On entry to state:
Initialise Variables;

Action: Initialise Variables
SS_AR_IDLE = false;
SS_AR_IN_PROGRESS = true;
SS_AR_ABORT = false;
SS_AR_FUEL_UPLIFT_CONFIRMATION = false;
SS_AR_FUEL_UPLIFT_COMPARISON = false;
SS_AR_COMPLETE = false;

Surge Protection

On entry to state:
SS_AR_SURGE = false;

[AR_ANY_TANK_INLET_VALVE_OPEN = true]

On entry to state:
SS_AR_SURGE = true;

AR_TK_LWT

On entry to state:
evaluate_status();

During the state is active:
evaluate_status();

Method: Evaluate_status
Boolean Active =
((MASS_TK_LWT < (TARGET_TK_LWT - LWT_MASS_MARGIN)) AND
(HIGH_LEVEL_STATE_TK_LWT == NOT_HIGH));

Boolean Idle =
((MASS_TK_LWT >= (TARGET_TK_LWT - LWT_RED_VALUE)) OR
(HIGH_LEVEL_STATE_TK_LWT == HIGH));

AR_TK_RWT

On entry to state:
evaluate_status();

During the state is active:
evaluate_status();

Method: Evaluate_status
Boolean Active =
((MASS_TK_RWT < (TARGET_TK_RWT - RWT_MASS_MARGIN)) AND
(HIGH_LEVEL_STATE_TK_RWT == NOT_HIGH));

Boolean Idle =
((MASS_TK_RWT >= (TARGET_TK_RWT - RWT_RED_VALUE)) OR
(HIGH_LEVEL_STATE_TK_RWT == HIGH));
AR_TK_CT

On entry to state:
evaluate_status();

During the state is active:
evaluate_status();

Idle
On entry to state:
SS_AR_CT = false;
SS_AR_CT_SEQ = false;

Active
On entry to state:
SS_AR_CT = true;

Valve_sequencing
On entry to state:
SS_AR_CT = false;
SS_AR_CT_SEQ = true;

Method: Evaluate_status

Boolean Active =
(MASS_TK_CT <= (TARGET_TK_CT - CT_MASS_MARGIN)) AND
(HIGH_LEVEL_STATE_TK_CT == NOT_HIGH);

Boolean Start_Sequence =
(MASS_TK_CT >= (TARGET_TK_CT - CT_RED_VALUE - CT_VLV_SEQ_MASS));

Boolean Finish_Sequence =
(MASS_TK_CT <= (TARGET_TK_CT - CT_RED_VALUE))
OR
(HIGH_LEVEL_STATE_TK_CT == HIGH);

Boolean Idle =
(MASS_TK_CT >= (TARGET_TK_CT - CT_RED_VALUE))
OR
((HIGH_LEVEL_STATE_TK_CT == HIGH) OR
(FOB >= FQMS_PFQ))

Idle == true
Active == true
Start_Sequence == true
Finish_Sequence == true
Defuel Operation State Charts

Defuel

On entry to state:
Initialise Variables;
Evaluate_Conditions();

During the state is active:
Evaluate_Conditions();

Action: Initialise Variables
SS_AR_ACTIVE = false;
SS_MR_ACTIVE = false;
SS_DF_ACTIVE = true;
SS_GT_ACTIVE = false;
SS_SOT_ACTIVE = false;
SS_MODE_OFF = false;

Method:
Evaluate_Conditions

Idle

On entry to state:
SS_DF_IN_PROGRESS = false;
SS_PDF_IN_PROGRESS = false;
SS_DF_ABORT = false;
SS_DF_IDLE = true;
SS_SDF_LWT = false;
SS_SDF_RWT = false;
SS_DF_CT = false;
SS_PDF_PC1 = false;
SS_PDF_PC2 = false;

Abort

On entry to state:
SS_DF_IN_PROGRESS = false;
SS_PDF_IN_PROGRESS = false;
SS_SDF_IN_PROGRESS = false;
SS_DF_ABORT = true;
SS_DF_IDLE = false;
SS_SDF_LWT = false;
SS_SDF_RWT = false;
SS_SDF_CT = false;
SS_PDF_PC1 = false;
SS_PDF_PC2 = false;

In_Progress

On entry to state:
SS_DF_IN_PROGRESS = true;
SS_DF_ABORT = false;
SS_DF_IDLE = false;

Suction_Defuel_In_Progress

Pressure_Defuel_In_Progress

[ANY_PUMP_PB_ON== false]

[ANY_PUMP_PB_ON== true]

[ANY_PUMP_PB_ON== false]
Method name = Evaluate_Conditions
{
    Boolean DF_ANY_TIV_SW_OPEN =
        IRP_MAN_VALVES_TK_LWT == OPEN OR IRP_MAN_VALVES_TK_RWT == OPEN OR
        IRP_MAN_VALVES_TK_CT == OPEN;

    Boolean ANY_PUMP_PB_ON =
        AFDX_ICP_CMDS_ICP_P_C1_PB == ON OR AFDX_ICP_CMDS_ICP_P_C2_PB == ON OR
        AFDX_ICP_CMDS_ICP_P_M1_PB == ON OR AFDX_ICP_CMDS_ICP_P_M2_PB == ON OR
        AFDX_ICP_CMDS_ICP_P_S1_PB == ON OR AFDX_ICP_CMDS_ICP_P_S2_PB == ON;

    Boolean DF_ANY_JTSN_VLV_OPEN =
        PIN_PROGRAMMING_OPT_JETTISON == TRUE AND
        RES_EQUIP_STATE_V_J1 == OPEN OR RES_EQUIP_STATE_V_J2 == OPEN;

    Boolean DF_ANY_TIV_FAILED_OPEN =
        (RES_EQUIP_STATE_V_RL == OPEN AND RES_EQUIP_STATUS_V_RL == FAILED) OR
        (RES_EQUIP_STATE_V_RR == OPEN AND RES_EQUIP_STATUS_V_RR == FAILED) OR
        (RES_EQUIP_STATE_V_RC1 == OPEN AND RES_EQUIP_STATUS_V_RC1 == FAILED) OR
        (RES_EQUIP_STATE_V_RC2 == OPEN AND RES_EQUIP_STATUS_V_RC2 == FAILED);

    Boolean DF_MAN_XFR_SELCTD = AFDX_ICP_CMDS_ICP_LWT_XFR_PB == ON OR
        AFDX_ICP_CMDS_ICP_RWT_XFR_PB == ON;

    Boolean Abort = DF_ANY_JTSN_VLV_OPEN == TRUE OR DF_MAN_XFR_SELCTD == TRUE OR
        (DF_ANY_TIV_FAILED_OPEN == TRUE AND ANY_PUMP_PB_ON == TRUE) OR
        (DF_ANY_TIV_SW_OPEN == TRUE AND ANY_PUMP_PB_ON == FALSE) AND
        OVERFLOW_CONDITION;

    Boolean Idle = DF_ANY_TIV_SW_OPEN == FALSE AND ANY_PUMP_PB_ON == FALSE AND
        Abort == FALSE;

    Boolean In_Progress = (DF_ANY_TIV_SW_OPEN == TRUE OR ANY_PUMP_PB_ON == TRUE) AND
        Abort == FALSE;
}

Suction_Defuel_In_Progress

On entry to state:
SS_SDF_IN_PROGRESS = true;
SS_PDF_IN_PROGRESS = false;
SS_PDF_PC1 = false;
SS_PDF_PC2 = false;

SUCTION_DEFUEL_TK_LWT

SUCTION_DEFUEL_TK_RWT

SUCTION_DEFUEL_TK_CT

SUCTION_DEFUEL_TK_LWT

On entry to state:
evaluate_status();
During the state is active:
evaluate_status();

Method: Evaluate_status
Boolean Active = IRP_MAN_VALVES_TIV_LWT == TRUE;
Boolean Idle = IRP_MAN_VALVES_TIV_LWT == FALSE;

Idle
On entry to state:
SS_SDF_LWT = false;

[Active == true]
[Idle == true]

Active
On entry to state:
SS_SDF_LWT = true;

SUCTION_DEFUEL_TK_RWT

On entry to state:
evaluate_status();
During the state is active:
evaluate_status();

Method: Evaluate_status
Boolean Active = IRP_MAN_VALVES_TIV_RWT == TRUE;
Boolean Idle = IRP_MAN_VALVES_TIV_RWT == FALSE;

Idle
On entry to state:
SS_SDF_RWT = false;

[Active == true]
[Idle == true]

Active
On entry to state:
SS_SDF_RWT = true;
SUCTION_DEFUEL_TK_CT

On entry to state:
evaluate_status();
During the state is active:
evaluate_status();

Method: Evaluate_status

Boolean Active =
IRP_MAN_VALVES_TIV_CT == TRUE;
Boolean Idle =
IRP_MAN_VALVES_TIV_CT == FALSE;

Pressure_Defuel_In_Progress

On entry to state:
SS_SDF_IN_PROGRESS = false;
Active
On entry to state:
SS_SDF_IN_PROGRESS = true;

PRESSURE_DEFUEL_PC1

On entry to state:
evaluate_status();
During the state is active:
evaluate_status();

Method: Evaluate_status

Boolean Active =
AFDX_ICP_CMDS_ICP_P_C1_PB = TRUE AND
TK_Empty_TK_CT == FALSE;
Boolean Idle =
AFDX_ICP_CMDS_ICP_P_C1_PB
== FALSE OR TK_Empty_TK_CT == TRUE;

PRESSURE_DEFUEL_PC2

On entry to state:
evaluate_status();
During the state is active:
evaluate_status();

Method: Evaluate_status

Boolean Active =
AFDX_ICP_CMDS_ICP_P_C2_PB = TRUE AND
TK_Empty_TK_CT == FALSE;
Boolean Idle =
AFDX_ICP_CMDS_ICP_P_C2_PB
== FALSE OR TK_Empty_TK_CT == TRUE;
Ground Transfer Operation State Charts

Ground Transfer

On entry to state:
Initialise Variables;
Evaluate_Conditions();

During the state is active:
Evaluate_Conditions();

Action: Initialise Variables
SS_AR_ACTIVE = false;
SS_MR_ACTIVE = false;
SS_DF_ACTIVE = false;
SS_GT_ACTIVE = true;
SS_SOT_ACTIVE = false;
SS_MODE_OFF = false;

Method:
Evaluate_Conditions

Idle
On entry to state:
SS_GT_IDLE = true;
SS_GT_IN_PROGESS = false;
SS_GT_ABORT = false;
SS_GT_RUN_PC1 = false;
SS_GT_RUN_PC2 = false;
SS_GT_OPEN_VRL = false;
SS_GT_OPEN_VRR = false;
SS_GT_OPEN_CTIV = false;

Abort
On entry to state:
SS_GT_IDLE = false;
SS_GT_IN_PROGRESS = false;
SS_GT_ABORT = true;
SS_GT_RUN_PC1 = false;
SS_GT_RUN_PC2 = false;
SS_GT_OPEN_VRL = false;
SS_GT_OPEN_VRR = false;
SS_GT_OPEN_CTIV = false;

In_Progress

[Idle == true]
[Abort == false]
[Abort == true]
[In_Progress == true]
[In_Progress == false]
Method name = Evaluate_Conditions
{
    Boolean GT_ANY_TIV_SW_OPEN =
        IRP_MAN_VALVES_TIV_LWT == OPEN OR IRP_MAN_VALVES_TIV_RWT == OPEN
        OR IRP_MAN_VALVES_TIV_CT == OPEN;

    Boolean GT_ANY_PUMP_PB_ON =
        AFDX_ICP_CMDS_ICP_P_C1_PB == ON OR AFDX_ICP_CMDS_ICP_P_C2_PB == ON
        OR AFDX_ICP_CMDS_ICP_P_M1_PB == ON OR AFDX_ICP_CMDS_ICP_P_M2_PB == ON
        OR AFDX_ICP_CMDS_ICP_P_S1_PB == ON OR AFDX_ICP_CMDS_ICP_P_S2_PB == ON;

    Boolean GT_ANY_JTSN_VLV_OPEN =
        PIN_PROGRAMMING_OPT_JETTISON == TRUE AND
        (RES_EQUIP_STATE_V_J1 == OPEN OR RES_EQUIP_STATE_V_J2 == OPEN);

    Boolean GT_MAN_XFR_SELCTD = AFDX_ICP_CMDS_ICP_LWT_XFR_PB == ON OR
        AFDX_ICP_CMDS_ICP_RWT_XFR_PB == ON;

    Boolean Abort = OVERFLOW_CONDITION OR GT_ANY_JTSN_VLV_OPEN == TRUE
        OR GT_MAN_XFR_SELCTD == TRUE;

    Boolean Idle = (GT_ANY_TIV_SW_OPEN == FALSE OR GT_ANY_PUMP_PB_ON == FALSE) AND
        Abort == FALSE;

    Boolean In_Progress = GT_ANY_TIV_SW_OPEN == TRUE AND
        GT_ANY_PUMP_PB_ON == TRUE AND Abort == FALSE;
}
In_Progress

On entry to state:
SS_GT_IDLE = false;
SS_GT_IN_PROGRESS = true;
SS_GT_ABORT = false;

CENTRE_TANK_PUMP_CONTROL

P_C1

On entry to state:
evaluate_status();

During the state is active:
evaluate_status();

Method: Evaluate_status

Boolean Run_PC1 =
AFDX_ICP_CMDS_ICP_P_C1_PB == ON AND
TK_Empty_TK_CT == FALSE;

Boolean Stop_PC1 =
AFDX_ICP_CMDS_ICP_P_C1_PB == OFF OR
TK_Empty_TK_CT == TRUE;

TANK_INLET_VALVE_CONTROL

LWT_INLET_VALVE
RWT_INLET_VALVE
CT_INLET_VALVES

P_C2

On entry to state:
evaluate_status();

During the state is active:
evaluate_status();

Method: Evaluate_status

Boolean Run_PC2 =
AFDX_ICP_CMDS_ICP_P_C2_PB == ON AND
TK_Empty_TK_CT == FALSE;

Boolean Stop_PC2 =
AFDX_ICP_CMDS_ICP_P_C2_PB == OFF OR
TK_Empty_TK_CT == TRUE;
LWT_INLET_VALVE

On entry to state:
evaluate_status();

During the state is active:
evaluate_status();

Method: Evaluate_status

Boolean Open =
![IRP_MAN_VALVES_TIV_LWT AND
(HIGH_LEVEL_STATE_TK_LWT == NOT_HIGH AND
HIGH_LEVEL_STATUS_TK_LWT == NORMAL) OR
HIGH_LEVEL_STATUS_TK_LWT == FAILED)];

Boolean Shut =
![IRP_MAN_VALVES_TIV_LWT == SHUT OR
(HIGH_LEVEL_STATE_TK_LWT == HIGH AND
HIGH_LEVEL_STATUS_TK_LWT == NORMAL)];

Shut

On entry to state:
SS_GT_OPEN_VRL = false;

Open

On entry to state:
SS_GT_OPEN_VRL = true;

RWT_INLET_VALVE

On entry to state:
evaluate_status();

During the state is active:
evaluate_status();

Method: Evaluate_status

Boolean Open =
![IRP_MAN_VALVES_TIV_RWT AND
(HIGH_LEVEL_STATE_TK_RWT == NOT_HIGH AND
HIGH_LEVEL_STATUS_TK_RWT == NORMAL) OR
HIGH_LEVEL_STATUS_TK_RWT == FAILED)];

Boolean Shut =
![IRP_MAN_VALVES_TIV_RWT == SHUT OR
(HIGH_LEVEL_STATE_TK_RWT == HIGH AND
HIGH_LEVEL_STATUS_TK_RWT == NORMAL)];

Shut

On entry to state:
SS_GT_OPEN_VRR = false;

Open

On entry to state:
SS_GT_OPEN_VRR = true;

CT_INLET_VALVES

On entry to state:
evaluate_status();

During the state is active:
evaluate_status();

Method: Evaluate_status

Boolean Open =
![IRP_Man_VALVES_TIV_CT AND
(HIGH_LEVEL_STATE_TK_CT == NOT_HIGH AND
HIGH_LEVEL_STATUS_TK_CT == NORMAL) OR
HIGH_LEVEL_STATUS_TK_CT == FAILED)];

Boolean Shut =
![IRP_Man_VALVES_TIV_CT == SHUT OR
(HIGH_LEVEL_STATE_TK_CT == HIGH AND
HIGH_LEVEL_STATUS_TK_CT == NORMAL)];

Shut

On entry to state:
SS_GT_OPEN_CTIv = false;

Open

On entry to state:
SS_GT_OPEN_CTIv = true;
Shut-Off Test Operation State Chart

Manual Refuel

On entry to state:
Initialise Variables;
Evaluate_Conditions();

During the state is active:
Evaluate_Conditions();

Action: Initialise Variables
SS_AR_ACTIVE = false;
SS_MR_ACTIVE = false;
SS_DF_ACTIVE = false;
SS_GT_ACTIVE = false;
SS_SOT_ACTIVE = true;
SS_MODE_OFF = false;

Method:
Evaluate_Conditions

In_Progress
On entry to state:
SS_SOT_IN_PROGRESS = true;

[SOT_PASSED == true]
[SOT_FAILED == true]

SOT_Passed
On entry to state:
SS_SOT_IN_PROGRESS = true;
SS_SOT_PASSED = true;

SOT_Failed
On entry to state:
SS_SOT_IN_PROGRESS = true;
SS_SOT_FAILED = true;

[SOT_FAILED == true]

SOT_Complete
On entry to state:
SS_SOT_IN_PROGRESS = false;
SS_SOT_PASSED = false;
SS_SOT_FAILED = false;
SS_SOT_COMPLETE = true;
AR_AUTO_SOT = true;
Method name = Evaluate_Conditions
{
  Boolean ANY_ISOLATION_VALVE_FAILED_OPEN =
  (RES_EQUIP_STATE_V_SR1 == OPEN AND RES_EQUIP_STATUS_V_SR1 == FAILED)
  OR (PIN_PROGRAMMING_OPT_SR1 == TRUE AND
       RES_EQUIP_STATE_V_SR2 == OPEN AND RES_EQUIP_STATUS_V_SR2 == FAILED);

  Boolean SOT_TIV_FAILED_OPEN = (RES_EQUIP_STATE_V_RL == OPEN AND
                                  RES_EQUIP_STATUS_V_RL == FAILED) OR (RES_EQUIP_STATE_V_RR == OPEN AND
                                  RES_EQUIP_STATUS_V_RR == FAILED) OR (RES_EQUIP_STATE_V_RC1 == OPEN
                                  AND RES_EQUIP_STATUS_V_RC1 == FAILED) OR (RES_EQUIP_STATE_V_RC2 ==
                                  OPEN AND RES_EQUIP_STATUS_V_RC2 == FAILED);

  Boolean HIGH_LEVEL_DETECTION_FAILED =
  HIGH_LEVEL_STATUS_TK_LWT == FAILED OR
  HIGH_LEVEL_STATUS_TK_RWT == FAILED OR
  HIGH_LEVEL_STATUS_TK_CT == FAILED;

  Boolean OVERFLOW_DETECTION_FAILED =
  OVERFLOW_STATUS_SIGNAL_OFLOW_LST == FAILED OR
  OVERFLOW_STATUS_SIGNAL_OFLOW_RST == FAILED;

  Boolean SOT_FAILED =
  ANY_ISOLATION_VALVE_FAILED_OPEN OR
  SOT_TIV_FAILED_OPEN OR
  HIGH_LEVEL_DETECTION_FAILED OR
  OVERFLOW_DETECTION_FAILED;

  Boolean SOT_PASSED = SOT_FAILED == FALSE;
}
Off Operation State Chart

Off

**On entry to state:**
- SS_AR_ACTIVE = false;
- SS_MR_ACTIVE = false;
- SS_DF_ACTIVE = false;
- SS_GT_ACTIVE = false;
- SS_SOT_ACTIVE = false;
- SS_MODE_OFF = true;
- SYSTEM_BITEGROUND_ENABLE = true;
Appendix B – Hierarchical Tree Structure Designs for all Fuel Operations

Fuel Management System

- Manual_Refuel_Op
- Automatic_Refuel_Op
- Defuel_Op
- Ground_Transfer_Op
- Shut_Off_Test_Op
- Off_Op

Full structure of each fuel operation shown on the following pages
Inv = Invocation Connector
CU = Computation Unit
Appendix C – Fuel Management System Implementation Screenshots in GME

 Entire Fuel Management System

 Manual Refuel Operation
Executable code for computation unit
void eval(int IRP_MAN_VALVES_TIV_LWT, int IRP_MAN_VALVES_TIV_RWT, int IRP_MAN_VALVES_TIV_CT, int PIN_PROGRAMMING_OPT_JETTISON, int RES_EQUIP_STATE_V_J1, int RES_EQUIP_STATE_V_J2, int RES_EQUIP_STATE_V_D, int RES_EQUIP_STATE_V_B, int RES_EQUIP_STATE_V_JL, int RES_EQUIP_STATE_V_JR, int RES_EQUIP_STATUS_V_D, int RES_EQUIP_STATUS_V_B, int RES_EQUIP_STATUS_V_JL, int RES_EQUIP_STATUS_V_JR, int AFDX_ICP_CMDS_ICP_LWT_XFR_PB, int AFDX_ICP_CMDS_ICP_RWT_XFR_PB, int &COMP_TO_EXECUTE)
{
    int ANY_TIV_SWITCH_OPEN, MR_ANY_JTSN_VLV_OPEN, MR_ANY_CNNCTNG_VLV_FAILED_OPEN, MR_MAN_XFR_SELCTD, IDLE, IN_PROGRESS, ABORT;
    if(IRP_MAN_VALVES_TIV_LWT == 1 || IRP_MAN_VALVES_TIV_RWT == 1 || IRP_MAN_VALVES_TIV_CT == 1)
    {
        ANY_TIV_SWITCH_OPEN = 1;
    }
    else
    {
        ANY_TIV_SWITCH_OPEN = 0;
    }
    if(PIN_PROGRAMMING_OPT_JETTISON == 1 && (RES_EQUIP_STATE_V_J1 == 1 || RES_EQUIP_STATE_V_J2 == 1))
    {
        MR_ANY_JTSN_VLV_OPEN = 1;
    }
    else
    {
        MR_ANY_JTSN_VLV_OPEN = 0;
    }
    if((RES_EQUIP_STATE_V_D == 1 && RES_EQUIP_STATUS_V_D == 0) || (PIN_PROGRAMMING_OPT_JETTISON == 1 && ((RES_EQUIP_STATE_V_B == 1 && RES_EQUIP_STATUS_V_B == 0) || (RES_EQUIP_STATE_V_JL == 1 && RES_EQUIP_STATUS_V_JL == 0) || (RES_EQUIP_STATE_V_JR == 1 && RES_EQUIP_STATUS_V_JR == 0))))
    {
        MR_ANY_CNNCTNG_VLV_FAILED_OPEN = 1;
    }
    else
    {
        MR_ANY_CNNCTNG_VLV_FAILED_OPEN = 0;
    }
    if(AFDX_ICP_CMDS_ICP_LWT_XFR_PB == 1 || AFDX_ICP_CMDS_ICP_RWT_XFR_PB == 1)
    {
        MR_MAN_XFR_SELCTD = 1;
    }
    else
    {
        MR_MAN_XFR_SELCTD = 0;
    }
    if(MR_ANY_JTSN_VLV_OPEN == 1 || MR_ANY_CNNCTNG_VLV_FAILED_OPEN == 1 || MR_MAN_XFR_SELCTD == 1)
    {
        ABORT = 1;
    }
    else
    {
        ABORT = 0;
    }
    if(ANY_TIV_SWITCH_OPEN == 0 && ABORT == 0)
    {
        IDLE = 1;
    }
    else
    {
        IDLE = 0;
    }
    if(ANY_TIV_SWITCH_OPEN == 1 && ABORT == 0)
    {
        IN_PROGRESS = 1;
    }
    else
    {
        IN_PROGRESS = 0;
    }
    if(IDLE == 1 && IN_PROGRESS == 0 && ABORT == 0)
    {
        COMP_TO_EXECUTE = 1;
    }
    else if(IDLE == 0 && IN_PROGRESS == 1 && ABORT == 0)
    {
        COMP_TO_EXECUTE = 2;
    }
    else
    {
        COMP_TO_EXECUTE = 3;
    }
}
void eval(int IRP_MAN_VALVES_TIV_LWT, int HIGH_LEVEL_STATE_TK_LWT, int HIGH_LEVEL_STATUS_TK_LWT, int &COMP_TO_EXECUTE)
{
    int IDLE, ACTIVE;

    if(IRP_MAN_VALVES_TIV_LWT == 1 && HIGH_LEVEL_STATE_TK_LWT == 0 &&
       HIGH_LEVEL_STATUS_TK_LWT == 1 || HIGH_LEVEL_STATUS_TK_LWT == 0)
    {
        IDLE = 0;
        ACTIVE = 1;
    }

    if(IRP_MAN_VALVES_TIV_LWT == 0 || HIGH_LEVEL_STATE_TK_LWT == 1 &&
       HIGH_LEVEL_STATUS_TK_LWT == 1)
    {
        IDLE = 1;
        ACTIVE = 0;
    }

    if(ACTIVE == 0 && IDLE == 1)
    {
        COMP_TO_EXECUTE = 1;
    }
    else
    {
        COMP_TO_EXECUTE = 2;
    }
}
The screenshots for the ‘MR_TK_RWT’ and ‘MR_TK_CT’ composite components will not be shown because their structure is exactly the same as the MR_TK_LWT screenshots shown here.
void eval(int RES_EQUIP_STATE_V_RL, int RES_EQUIP_STATE_V_RR, int RES_EQUIP_STATE_V_RC1, int RES_EQUIP_STATE_V_RC2, int &SURGE_PROT_COMP_TO_EXEC)
{
    int MR_ANY_TIV_OPEN;

    if((RES_EQUIP_STATE_V_RL == 1 || RES_EQUIP_STATE_V_RR == 1 || RES_EQUIP_STATE_V_RC1 == 1 || RES_EQUIP_STATE_V_RC2 == 1))
    {
        MR_ANY_TIV_OPEN = 1;
    }
    else
    {
        MR_ANY_TIV_OPEN = 0;
    }

    if(MR_ANY_TIV_OPEN == 1)
    {
        SURGE_PROT_COMP_TO_EXEC = 1;
    }
    else
    {
        SURGE_PROT_COMP_TO_EXEC = 2;
    }
}

Executable code for the 'SP_Eval_Status' atomic component. This code is placed inside the computation unit of this atomic component.
Automatic Refuel Operation
void eval()
{
    TARGET_TK_LWT, int MASS_TK_LWT, int TARGET_TK_RWT, int MASS_TK_RWT, int TARGET_TK_CT, int MAES_TK_CT, int HIGH_LEVEL_STATE_TK_LWT, int HIGH_LEVEL_STATE_TK_RWT, int HIGH_LEVEL_STATE_TK_CT, int HIGH_LEVEL_STATUS_TK_LWT, int HIGH_LEVEL_STATUS_TK_RWT, int HIGH_LEVEL_STATUS_TK_CT, int RES_EQUIP_STATE_V_JL, int RES_EQUIP_STATE_V_JR, int RES_EQUIP_STATE_V_J1, int RES_EQUIP_STATE_V_J2, int RES_EQUIP_STATE_V_B, int RES_EQUIP_STATE_V_RL, int RES_EQUIP_STATE_V_RR, int RES_EQUIP_STATE_V_RC1, int RES_EQUIP_STATE_V_RC2, int PIN_PROGRAMMING_OPT_JETTISON, int RES_EQUIP_STATE_V_J1.INT RES_EQUIP_STATE_V_J2, int RES_EQUIP_STATE_V_B, int RES_EQUIP_STATE_V_RL, int RES_EQUIP_STATE_V_RR, int RES_EQUIP_STATE_V_RC1, int RES_EQUIP_STATE_V_RC2, int RES_EQUIP_STATE_VJR, int FQMS_PFQ, int AFDX_ICP_CMDS_ICP_LWT_XFR_PB, int AFDX_ICP_CMDS_ICP_RWT_XFR_PB.

    if (COMP_TO_EXECUTE == 2)
    {
        if (TARGET_TK_LWT > TARGET_TK_RWT)
        {
            LWT_Targeted = 1;
        }
        else
        {
            LWT_Targeted = 0;
        }
    }
    else
    {
        if (TARGET_TK_RWT > TARGET_TK_CT)
        {
            RWT_Targeted = 1;
        }
        else
        {
            RWT_Targeted = 0;
        }
    }
    else
    {
        if (TARGET_TK_CT > TARGET_TK_LWT)
        {
            CT_Targeted = 1;
        }
        else
        {
            CT_Targeted = 0;
        }
    }
    else
    {
        if (HIGH_LEVEL_STATE_TK_LWT == 1 && HIGH_LEVEL_STATE_TK_RWT == 1 && HIGH_LEVEL_STATE_TK_CT == 1)
        {
            ALL_TANKS_HIGH_LEVEL = 1;
        }
        else
        {
            ALL_TANKS_HIGH_LEVEL = 0;
        }
    }
    else
    {
        if (UNATTAINABLE_PFQ == 1)
        {
            TARGETED_TIV_FAILED_SHUT = 1;
        }
        else
        {
            TARGETED_TIV_FAILED_SHUT = 0;
        }
    }
    else
    {
        if (RES_EQUIP_STATE_V_B == 0)
        {
            ANY_TANK_MASS_GREATER_THAN_TARGET = 0;
        }
        else
        {
            ANY_TANK_MASS_GREATER_THAN_TARGET = 1;
        }
    }
    else
    {
        if (RES_EQUIP_STATE_V_J1 == 1 || RES_EQUIP_STATE_V_J2 == 1)
        {
            ANY_JETTISON_VALVE_OPEN = 1;
        }
        else
        {
            ANY_JETTISON_VALVE_OPEN = 0;
        }
    }
    else
    {
        if (TARGETED_HIGH_LEVEL_FAILED == 1)
        {
            TARGETED_TIV_FAILED_SHUT = 1;
        }
        else
        {
            TARGETED_TIV_FAILED_SHUT = 0;
        }
    }
    else
    {
        if (TARGETED_TIV_FAILED_SHUT == 1)
        {
            AUTO_REFUEL_ABORT = 1;
        }
        else
        {
            AUTO_REFUEL_ABORT = 0;
        }
    }
    else
    {
        if (ANY_TANK_MASS_GREATER_THAN_TARGET == 0)
        {
            COMP_TO_Execute = 1;
        }
        else
        {
            COMP_TO_Execute = 2;
        }
    }
}
void eval(int RES_EQUIP_STATE_V_RL, int RES_EQUIP_STATE_V_RR, int RES_EQUIP_STATE_V_RC1, int RES_EQUIP_STATE_V_RC2, int SS_AR_LWT, int SS_AR_RWT, int SS_AR_CT, int FOB, int FQMS_PFQ, int FOB_ACCURACY, int THR_AUTO_REFUEL, int MIN_UPLIFT_QTY, int &COMP_TO_EXECUTE)
{
    int ALL_TANK_INLET_VALVES_SHUT, ALL_TANK_INLET_VALVES_CMD_SHUT, FOB_IN_TOLERANCE, PFQ_LOWER_THRESHOLD;
    if (RES_EQUIP_STATE_V_RL == 0 && RES_EQUIP_STATE_V_RR == 0 && RES_EQUIP_STATE_V_RC1 == 0 && RES_EQUIP_STATE_V_RC2 == 0)
        ALL_TANK_INLET_VALVES_SHUT = 1;
    else
        ALL_TANK_INLET_VALVES_SHUT = 0;
    if (SS_AR_LWT == 0 && SS_AR_RWT == 0 && SS_AR_CT == 0)
        ALL_TANK_INLET_VALVES_CMD_SHUT = 1;
    else
        ALL_TANK_INLET_VALVES_CMD_SHUT = 0;
    if ((FOB >= FQMS_PFQ - FOB_ACCURACY) && (FOB <= FQMS_PFQ + FOB_ACCURACY))
        FOB_IN_TOLERANCE = 1;
    else
        FOB_IN_TOLERANCE = 0;
    PFQ_LOWER_THRESHOLD = FQMS_PFQ - FOB_ACCURACY - THR_AUTO_REFUEL;
    if (FQMS_PFQ < (FOB + MIN_UPLIFT_QTY))
        COMP_TO_EXECUTE = 1;
    else if (FQMS_PFQ < (FOB + MIN_UPLIFT_QTY))
        FOB < PFQ_LOWER_THRESHOLD || FQMS_PFQ < (FOB + MIN_UPLIFT_QTY)
            COMP_TO_EXECUTE = 2;
    else if (ALL_TANK_INLET_VALVES_CMD_SHUT == 1)
        COMP_TO_EXECUTE = 3;
    else if (FOB_IN_TOLERANCE == 1)
        COMP_TO_EXECUTE = 5;
}

Executable code for the 'AR_Mode_Evaluate_Conditions' atomic component.
This code is placed inside the computation unit of this atomic component.
The implementations of these sub-components are similar to those in the Manual Refuel operation so will not be shown.
void eval(int IRP_MAN_VALVES_TK_LWT, int IRP_MAN_VALVES_TK_RWT, int IRP_MAN_VALVES_TK_CT, int AFDX_ICP_CMDS_ICP_P_C1_PB, int AFDX_ICP_CMDS_ICP_P_C2_PB, int AFDX_ICP_CMDS_ICP_P_M1_PB, int AFDX_ICP_CMDS_ICP_P_M2_PB, int AFDX_ICP_CMDS_ICP_P_S1_PB, int AFDX_ICP_CMDS_ICP_P_S2_PB, int PIN_PROGRAMMING_OPT_JETTISON, int RES_EQUIP_STATE_V_J1, int RES_EQUIP_STATE_V_J2, int RES_EQUIP_STATE_V_RL, int RES_EQUIP_STATUS_V_RL, int RES_EQUIP_STATE_V_RR, int RES_EQUIP_STATUS_V_RR, int RES_EQUIP_STATE_V_RC1, int RES_EQUIP_STATUS_V_RC1, int RES_EQUIP_STATE_V_RC2, int RES_EQUIP_STATUS_V_RC2, int AFDX_ICP_CMDS_ICP_LWT_XFR_PB, int AFDX_ICP_CMDS_ICP_RWT_XFR_PB, int &COMP_TO_EXECUTE) {
    if (IRP_MAN_VALVES_TK_LWT == 1 || IRP_MAN_VALVES_TK_RWT == 1 || IRP_MAN_VALVES_TK_CT == 1) {
        DF_ANY_TIV_SW_OPEN = 1;
    } else {
        DF_ANY_TIV_SW_OPEN = 0;
    }
    if (AFDX_ICP_CMDS_ICP_P_C1_PB == 1 || AFDX_ICP_CMDS_ICP_P_C2_PB == 1 || AFDX_ICP_CMDS_ICP_P_M1_PB == 1 || AFDX_ICP_CMDS_ICP_P_M2_PB == 1 || AFDX_ICP_CMDS_ICP_P_S1_PB == 1 || AFDX_ICP_CMDS_ICP_P_S2_PB == 1) {
        ANY_PUMP_PB_ON = 1;
    } else {
        ANY_PUMP_PB_ON = 0;
    }
    if (PIN_PROGRAMMING_OPT_JETTISON == 1 && (RES_EQUIP_STATE_V_J1 == 1 || RES_EQUIP_STATE_V_J2 == 1)) {
        DF_ANY_JTSN_VLV_OPEN = 1;
    } else {
        DF_ANY_JTSN_VLV_OPEN = 0;
    }
    if (RES_EQUIP_STATE_V_RL == 1 && RES_EQUIP_STATUS_V_RL == 0) || (RES_EQUIP_STATE_V_RR == 1 && RES_EQUIP_STATUS_V_RR == 0) || (RES_EQUIP_STATE_V_RC1 == 1 && RES_EQUIP_STATUS_V_RC1 == 0) || (RES_EQUIP_STATE_V_RC2 == 1 && RES_EQUIP_STATUS_V_RC2 == 0) {
        DF_ANY_TIV_FAILED_OPEN = 1;
    } else {
        DF_ANY_TIV_FAILED_OPEN = 0;
    }
    if (AFDX_ICP_CMDS_ICP_LWT_XFR_PB == 1 || AFDX_ICP_CMDS_ICP_RWT_XFR_PB == 1) {
        DF_MAN_XFR_SELCTD = 1;
    } else {
        DF_MAN_XFR_SELCTD = 0;
    }
    if (DF_ANY_JTSN_VLV_OPEN == 1 || DF_MAN_XFR_SELCTD == 1 || DF_ANY_TIV_FAILED_OPEN == 1 || ANY_PUMP_PB_ON == 1) || (DF_ANY_TIV_SW_OPEN == 1 && ANY_PUMP_PB_ON == 0) {
        ABORT = 1;
    } else {
        ABORT = 0;
    }
    if (DF_ANY_TIV_SW_OPEN == 0 && ANY_PUMP_PB_ON == 0 && ABORT == 0) {
        IDLE = 1;
    } else {
        IDLE = 0;
    }
    if (DF_ANY_TIV_SW_OPEN == 1 && ANY_PUMP_PB_ON == 1 && ABORT == 0) {
        IN_PROGRESS = 1;
    } else {
        IN_PROGRESS = 0;
    }
    if (IDLE == 1 && IN_PROGRESS == 0 && ABORT == 0) {
        COMP_TO_EXECUTE = 1;
    } else if (IDLE == 0 && IN_PROGRESS == 1 && ABORT == 0) {
        COMP_TO_EXECUTE = 2;
    } else {
        COMP_TO_EXECUTE = 3;
    }
}

Executable code for the 'Defuel_Evaluate_Conditions' atomic component. This code is placed inside the computation unit of this atomic component.
```c
void eval(int AFDX_ICP_CMDS_ICP_P_C1_PB, int AFDX_ICP_CMDS_ICP_P_C2_PB, int AFDX_ICP_CMDS_ICP_P_M1_PB,
          int AFDX_ICP_CMDS_ICP_P_M2_PB, int AFDX_ICP_CMDS_ICP_P_S1_PB, int AFDX_ICP_CMDS_ICP_P_S2_PB,
          &DF_IN_PROG_TYPE_COMP_TO_EXEC)
{
    int ANY_PUMP_PB_ON;
    if((AFDX_ICP_CMDS_ICP_P_C1_PB == 1 || AFDX_ICP_CMDS_ICP_P_C2_PB == 1 || AFDX_ICP_CMDS_ICP_P_M1_PB == 1 ||
        AFDX_ICP_CMDS_ICP_P_M2_PB == 1 || AFDX_ICP_CMDS_ICP_P_S1_PB == 1 || AFDX_ICP_CMDS_ICP_P_S2_PB == 1))
    {
        ANY_PUMP_PB_ON = 1;
    }
    else
    {
        ANY_PUMP_PB_ON = 0;
    }
    if(ANY_PUMP_PB_ON == 0)
    {
        DF_IN_PROG_TYPE_COMP_TO_EXEC = 1;
    }
    else
    {
        DF_IN_PROG_TYPE_COMP_TO_EXEC = 2;
    }
}
```

Executable code for the 'Defuel_In_Progress_Type_Eval' atomic component. This code is placed inside the computation unit of this atomic component.
void eval(int IRP_MAN_VALVES_TIV_LWT, int &COMP_TO_EXECUTE)
{
    int IDLE, ACTIVE;
    if (IRP_MAN_VALVES_TIV_LWT == 1)
    {
        IDLE = 0;
        ACTIVE = 1;
    }
    else
    {
        IDLE = 1;
        ACTIVE = 0;
    }
    if (IDLE == 1 && ACTIVE == 0)
    {
        COMP_TO_EXECUTE = 1;
    }
    else
    {
        COMP_TO_EXECUTE = 2;
    }
}
The screenshots for the ‘SD_RWT’ and ‘SD_CT’ composite components will not be shown because their structure is exactly the same as the ‘SD_LWT’ screenshots shown here.
```c
void eval(int AFDX_ICP_CMDS_ICP_P_C1_PB, int TK_EMPTY_TK_CT, int &COMP_TO_EXECUTE)
{
    int IDLE, ACTIVE;

    if (AFDX_ICP_CMDS_ICP_P_C1_PB == 1 && TK_EMPTY_TK_CT == 0)
    {
        IDLE = 0;
        ACTIVE = 1;
    }
    else if (AFDX_ICP_CMDS_ICP_P_C1_PB == 0 || TK_EMPTY_TK_CT == 1)
    {
        IDLE = 1;
        ACTIVE = 0;
    }
    else if (IDLE == 1 && ACTIVE == 0)
    {
        COMP_TO_EXECUTE = 1;
    }
    else
    {
        COMP_TO_EXECUTE = 2;
    }
}
```

Executable code for the 'PD_PC1_Eval_Status' atomic component. This code is placed inside the computation unit of this atomic component.
The screenshots for the 'Pressure_Defuel_PC2' composite component will not be shown because its structure is exactly the same as the 'Pressure_Defuel_PC1' screenshots shown here.
Ground Transfer Operation
void eval(int IRP_MAN_VALVES_TIV_LWT, int IRP_MAN_VALVES_TIV_RWT, int IRP_MAN_VALVES_TIV_CT, int AFDX_ICP_CMDS_ICP_P_C1_PB, int AFDX_ICP_CMDS_ICP_P_C2_PB, int AFDX_ICP_CMDS_ICP_P_M1_PB, int AFDX_ICP_CMDS_ICP_P_M2_PB, int AFDX_ICP_CMDS_ICP_P_S1_PB, int AFDX_ICP_CMDS_ICP_P_S2_PB, int PIN_PROGRAMMING_OPT_JETTISON, int RES_EQUIP_STATE_V_J1, int RES_EQUIP_STATE_V_J2, int AFDX_ICP_CMDS_ICP_LWT_XFR_PB, int AFDX_ICP_CMDS_ICP_RWT_XFR_PB, int &COMP_TO_EXECUTE)
{
    int GT_ANY_TIV_SW_OPEN, GT_ANY_PUMP_PB_ON, GT_ANY_JTSN_VLV_OPEN, GT_MAN_XFR_SELCTD, IDLE, IN_PROGRESS, ABORT;

    if (IRP_MAN_VALVES_TIV_LWT == 1 || IRP_MAN_VALVES_TIV_RWT == 1 || IRP_MAN_VALVES_TIV_CT == 1)
    {
        GT_ANY_TIV_SW_OPEN = 1;
    }
    else
    {
        GT_ANY_TIV_SW_OPEN = 0;
    }

    if (AFDX_ICP_CMDS_ICP_P_C1_PB == 1 || AFDX_ICP_CMDS_ICP_P_C2_PB == 1 || AFDX_ICP_CMDS_ICP_P_M1_PB == 1 || AFDX_ICP_CMDS_ICP_P_M2_PB == 1 || AFDX_ICP_CMDS_ICP_P_S1_PB == 1 || AFDX_ICP_CMDS_ICP_P_S2_PB == 1)
    {
        GT_ANY_PUMP_PB_ON = 1;
    }
    else
    {
        GT_ANY_PUMP_PB_ON = 0;
    }

    if (PIN_PROGRAMMING_OPT_JETTISON == 1 && RES_EQUIP_STATE_V_J1 == 1 || RES_EQUIP_STATE_V_J2 == 1)
    {
        GT_ANY_JTSN_VLV_OPEN = 1;
    }
    else
    {
        GT_ANY_JTSN_VLV_OPEN = 0;
    }

    if (AFDX_ICP_CMDS_ICP_LWT_XFR_PB == 1 || AFDX_ICP_CMDS_ICP_RWT_XFR_PB == 1)
    {
        GT_MAN_XFR_SELCTD = 1;
    }
    else
    {
        GT_MAN_XFR_SELCTD = 0;
    }

    if (GT_ANY_JTSN_VLV_OPEN == 1 || GT_MAN_XFR_SELCTD == 1)
    {
        ABORT = 1;
    }
    else
    {
        ABORT = 0;
    }

    if (GT_ANY_TIV_SW_OPEN == 0 || GT_ANY_PUMP_PB_ON == 0 && ABORT == 0)
    {
        IDLE = 1;
    }
    else
    {
        IDLE = 0;
    }

    if (GT_ANY_TIV_SW_OPEN == 1 && GT_ANY_PUMP_PB_ON == 1 && ABORT == 0)
    {
        IN_PROGRESS = 1;
    }
    else
    {
        IN_PROGRESS = 0;
    }

    if (IDLE == 1 && IN_PROGRESS == 0 && ABORT == 0)
    {
        COMP_TO_EXECUTE = 1;
    }
    else if (IDLE == 0 && IN_PROGRESS == 1 && ABORT == 0)
    {
        COMP_TO_EXECUTE = 2;
    }
    else
    {
        COMP_TO_EXECUTE = 3;
    }
}
```c
void eval(int IRP_MAN_VALVES_TIV_LWT, int HIGH_LEVEL_STATE_TK_LWT, int HIGH_LEVEL_STATUS_TK_LWT, int &COMP_TO_EXECUTE) {
    int OPEN_VRL, SHUT_VRL;
    if (IRP_MAN_VALVES_TIV_LWT == 1 && (HIGH_LEVEL_STATE_TK_LWT == 0 && HIGH_LEVEL_STATUS_TK_LWT == 1 || HIGH_LEVEL_STATUS_TK_LWT == 0)) {
        OPEN_VRL = 1;
        SHUT_VRL = 0;
    } else if (IRP_MAN_VALVES_TIV_LWT == 0 || HIGH_LEVEL_STATE_TK_LWT == 1 && HIGH_LEVEL_STATUS_TK_LWT == 1) {
        OPEN_VRL = 0;
        SHUT_VRL = 1;
    }
    if (OPEN_VRL == 1 && SHUT_VRL == 0) {
        COMP_TO_EXECUTE = 1;
    } else {
        COMP_TO_EXECUTE = 2;
    }
}
```

Executable code for the 'LWT_Inlet_Valve_Eval_Status' atomic component. This code is placed inside the computation unit of this atomic component.
The screenshots for the 'RWT_Inlet_Valve' and 'CT_Inlet_Valve' composite components will not be shown because its structure is exactly the same as the 'LWT_Inlet_Valve' screenshots shown here.
void eval(int AFDX_ICP_CMDS_ICP_P_C1_PB, int TK_Empty_TK_CT, int &COMP_TO_EXECUTE)
{
    int RUN_PC1, STOP_PC1;
    if (AFDX_ICP_CMDS_ICP_P_C1_PB == 1 && TK_Empty_TK_CT == 0)
    {
        RUN_PC1 = 1;
        STOP_PC1 = 0;
    }
    else if (AFDX_ICP_CMDS_ICP_P_C1_PB == 0 || TK_Empty_TK_CT == 1)
    {
        RUN_PC1 = 0;
        STOP_PC1 = 1;
    }
    if (RUN_PC1 == 1 && STOP_PC1 == 0)
    {
        COMP_TO_EXECUTE = 1;
    }
    else
    {
        COMP_TO_EXECUTE = 2;
    }
}

Executable code for the 'P_C1_Eval_Status' atomic component. This code is placed inside the computation unit of this atomic component.
The screenshots for the 'P_C2' composite component will not be shown because its structure is exactly the same as the 'P_C1' screenshots shown here.
void eval(int RES_EQUIP_STATE_V_SR1, int RES_EQUIP_STATUS_V_SR1, int PIN_PROGRAMMING_OPT_SR1, int RES_EQUIP_STATE_V_SR2, int RES_EQUIP_STATUS_V_SR2, int RES_EQUIP_STATE_V_RL, int RES_EQUIP_STATUS_V_RL, int RES_EQUIP_STATE_V_RR, int RES_EQUIP_STATUS_V_RR, int RES_EQUIP_STATE_V_RC1, int RES_EQUIP_STATUS_V_RC1, int RES_EQUIP_STATE_V_RC2, int RES_EQUIP_STATUS_V_RC2, int HIGH_LEVEL_STATUS_TK_LWT, int HIGH_LEVEL_STATUS_TK_RWT, int HIGH_LEVEL_STATUS_TK_CT, int OVERFLOW_STATUS_SIGNAL_OFLOW_LST, int OVERFLOW_STATUS_SIGNAL_OFLOW_RST, int &COMP_TO_EXECUTE) 
{
    int ANY_ISOLATION_VALVE_FAILED_OPEN, SOT_TIV_FAILED_OPEN, HIGH_LEVEL_DETECTION_FAILED, OVERFLOW_DETECTION_FAILED, SOT_FAILED, SOT_PASSED;

    if((RES_EQUIP_STATE_V_SR1 == 1 && RES_EQUIP_STATUS_V_SR1 == 0) || (PIN_PROGRAMMING_OPT_SR1 == 1 && RES_EQUIP_STATE_V_SR2 == 1 && RES_EQUIP_STATUS_V_SR2 == 0))
    {
        ANY_ISOLATION_VALVE_FAILED_OPEN = 1;
    }
    else
    {
        ANY_ISOLATION_VALVE_FAILED_OPEN = 0;
    }

    if((RES_EQUIP_STATE_V_RL == 1 && RES_EQUIP_STATUS_V_RL == 0) || (RES_EQUIP_STATE_V_RR == 1 && RES_EQUIP_STATUS_V_RR == 0) || (RES_EQUIP_STATE_V_RC1 == 1 && RES_EQUIP_STATUS_V_RC1 == 0) || (RES_EQUIP_STATE_V_RC2 == 1 && RES_EQUIP_STATUS_V_RC2 == 0))
    {
        SOT_TIV_FAILED_OPEN = 1;
    }
    else
    {
        SOT_TIV_FAILED_OPEN = 0;
    }

    if((HIGH_LEVEL_STATUS_TK_LWT == 0 || HIGH_LEVEL_STATUS_TK_RWT == 0 || HIGH_LEVEL_STATUS_TK_CT == 0))
    {
        HIGH_LEVEL_DETECTION_FAILED = 1;
    }
    else
    {
        HIGH_LEVEL_DETECTION_FAILED = 0;
    }

    if(OVERFLOW_STATUS_SIGNAL_OFLOW_LST == 0 || OVERFLOW_STATUS_SIGNAL_OFLOW_RST == 0)
    {
        OVERFLOW_DETECTION_FAILED = 1;
    }
    else
    {
        OVERFLOW_DETECTION_FAILED = 0;
    }

    if(ANY_ISOLATION_VALVE_FAILED_OPEN == 1 || SOT_TIV_FAILED_OPEN == 1 || HIGH_LEVEL_DETECTION_FAILED == 1 || OVERFLOW_DETECTION_FAILED == 1)
    {
        SOT_FAILED = 1;
    }
    else
    {
        SOT_FAILED = 0;
    }

    if(SOT_FAILED == 0)
    {
        SOT_PASSED = 1;
    }
    else
    {
        SOT_PASSED = 0;
    }

    if(SOT_PASSED == 1 && SOT_FAILED == 0)
    {
        COMP_TO_EXECUTE = 1;
    }
    else
    {
        COMP_TO_EXECUTE = 2;
    }
}

Executable code for the 'SOT_Eval_Conditions' atomic component. This code is placed inside the computation unit of this atomic component.
Off Operation
# Appendix D – Test Plan Showing all the Tests Performed for the Manual Refuel Operation

## Unit Tests

<table>
<thead>
<tr>
<th>Test Ref No.</th>
<th>Comment</th>
<th>Test Input</th>
<th>Expected Outcome</th>
<th>Actual Outcome</th>
<th>Actions Taken</th>
</tr>
</thead>
</table>
| U1           | Check that the 'MR_LWT_Idle' atomic component outputs the correct system state value. | Component name: MR_LWT_Idle  
Input Value: 1 | Output value = 0 for system state variable 'SS_MR_LWT' | Test Passed  
Value of 0 was output for system state variable 'SS_MR_LWT' | None |
| U2           | Check that the 'MR_LWT_Active' atomic component outputs the correct system state value. | Component name: MR_LWT_Active  
Input Value: 0 | Output value = 1 for system state variable 'SS_MR_LWT' | Test Passed  
Value of 1 was output for system state variable 'SS_MR_LWT' | None |
| U3           | Check the 'MR_LWT_Eval_Status' atomic component outputs the correct value to indicate the 'MR_LWT_Idle' atomic component needs to be executed. | Component name: MR_LWT_Eval_Status  
Input values: 0, 1, 1 | Output value = 1 which indicates the 'MR_LWT_Idle' atomic component needs to be executed. | Test Passed  
Value of 1 was output which indicates 'MR_LWT_Idle' atomic component will be executed. | None |
| U4           | Check the 'MR_LWT_Eval_Status' atomic component outputs the correct value to indicate the 'MR_LWT_Active' atomic component needs to be executed. | Component name: MR_LWT_Eval_Status  
Input values: 1, 0, 1 | Output value = 2 which indicates the 'MR_LWT_Active' atomic component needs to be executed. | Test Passed  
Value of 2 was output which indicates 'MR_LWT_Active' atomic component will be executed. | None |
| U5           | Check that the 'MR_RWWT_Idle' atomic component outputs the correct system state value. | Component name: MR_RWWT_Idle  
Input Value: 1 | Output value = 0 for system state variable 'SS_MR_RWWT' | Test Passed  
Value of 0 was output for system state variable 'SS_MR_RWWT' | None |
| U6           | Check that the 'MR_RWWT_Active' atomic component outputs the correct system state value. | Component name: MR_RWWT_Active  
Input Value: 0 | Output value = 1 for system state variable 'SS_MR_RWWT' | Test Passed  
Value of 1 was output for system state variable 'SS_MR_RWWT' | None |
| U7   | Check the 'MR_RWT_Eval_Status' atomic component outputs the correct value to indicate the 'MR_RWT_Idle' atomic component needs to be executed. | Component name: MR_RWT_Eval_Status  
Input values: 0, 1, 1 | Output value = 1 which indicates the 'MR_RWT_Idle' atomic component needs to be executed.  
Test Passed  
Value of 1 was output which indicates 'MR_RWT_Idle' atomic component will be executed. | None |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| U8   | Check the 'MR_RWT_Eval_Status' atomic component outputs the correct value to indicate the 'MR_RWT_Active' atomic component needs to be executed. | Component name: MR_RWT_Eval_Status  
Input values: 1, 0, 1 | Output value = 2 which indicates the 'MR_RWT_Active' atomic component needs to be executed.  
Test Passed  
Value of 2 was output which indicates 'MR_RWT_Active' atomic component will be executed. | None |
| U9   | Check that the 'MR_CT_Idle' atomic component outputs the correct system state value. | Component name: MR_CT_Idle  
Input Value: 1 | Output value = 0 for system state variable 'SS_MR_CT'.  
Test Passed  
Value of 0 was output for system state variable 'SS_MR_CT'. | None |
| U10  | Check that the 'MR_CT_Active' atomic component outputs the correct system state value. | Component name: MR_CT_Active  
Input Value: 0 | Output value = 1 for system state variable 'SS_MR_CT'.  
Test Passed  
Value of 1 was output for system state variable 'SS_MR_CT'. | None |
| U11  | Check the 'MR_CT_Eval_Status' atomic component outputs the correct value to indicate the 'MR_CT_Idle' atomic component needs to be executed. | Component name: MR_CT_Eval_Status  
Input values: 0, 1, 1 | Output value = 1 which indicates the 'MR_CT_Idle' atomic component needs to be executed.  
Test Passed  
Value of 1 was output which indicates 'MR_CT_Idle' atomic component will be executed. | None |
| U12  | Check the 'MR_CT_Eval_Status' atomic component outputs the correct value to indicate the 'MR_CT_Active' atomic component needs to be executed. | Component name: MR_CT_Eval_Status  
Input values: 1, 0, 1 | Output value = 2 which indicates the 'MR_CT_Active' atomic component needs to be executed.  
Test Passed  
Value of 2 was output which indicates 'MR_CT_Active' atomic component will be executed. | None |
| U13  | Check the 'MR_In_Progress_Init' atomic component initialises all variables to the correct state. | Component name: MR_In_Progress_Init  
Input values: 1, 1, 1 | Output values = 0, 1, 0.  
Test passed  
Output values = 0, 1, 0. | None |
<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
<th>Component name</th>
<th>Input Values</th>
<th>Output Value</th>
<th>Test Passed</th>
<th>Explanation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>U14</td>
<td>Check that the 'SP_Open' atomic component outputs the correct system state value.</td>
<td>SP_Open</td>
<td>0</td>
<td>Output value = 1</td>
<td>Value of 1 was output for system state variable 'SS_MR_SURGE'</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>U15</td>
<td>Check that the 'SP_Shut' atomic component outputs the correct system state value.</td>
<td>SP_Shut</td>
<td>1</td>
<td>Output value = 0</td>
<td>Value of 0 was output for system state variable 'SS_MR_SURGE'</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>U16</td>
<td>Check the 'SP_Eval_Status' atomic component outputs the correct value to indicate the 'SP_Open' atomic component needs to be executed.</td>
<td>SP_Eval_Status</td>
<td>1, 0, 1, 1</td>
<td>Output value = 1</td>
<td>Value of 1 was output which indicates the 'SP_Open' atomic component needs to be executed.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>U17</td>
<td>Check the 'SP_Eval_Status' atomic component outputs the correct value to indicate the 'SP_Shut' atomic component needs to be executed.</td>
<td>SP_Eval_Status</td>
<td>0, 0, 0, 0</td>
<td>Output value = 2</td>
<td>Value of 2 was output which indicates the 'SP_Shut' atomic component needs to be executed.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>U18</td>
<td>Check the 'MR_Idle' atomic component sets all the system state variables correctly to ensure only the 'SS_MR_IDLE' variable is true (1).</td>
<td>MR_Idle</td>
<td>0, 0, 0, 0, 0</td>
<td>Output values = 1, 0, 0, 0, 0, 0, 0</td>
<td>Only the 'SS_MR_IDLE' system state variable should have a value of 1 which is the 1st output value below.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>U19</td>
<td>Check the 'MR_Abort' atomic component sets all the system state variables correctly to ensure only the 'SS_MR_ABORT' variable is true (1).</td>
<td>MR_Abort</td>
<td>0, 0, 0, 0, 0</td>
<td>Output values = 0, 0, 0, 0, 0, 0, 0</td>
<td>Only the 'SS_MR_ABORT' system state variable should have a value of 1 which is the 3rd output value below.</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>Test Case</td>
<td>Description</td>
<td>Component name:</td>
<td>Input values</td>
<td>Output value</td>
<td>Test Result</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>----------------</td>
<td>--------------</td>
<td>--------------</td>
<td>-------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>U20</td>
<td>Check that the 'MR_Evaluate_Conditions' atomic component outputs the correct value to indicate the 'MR_Idle' sub-component (which is the 1st sub-component) needs to be executed based on the values input.</td>
<td>MR_Evaluate_Conditions</td>
<td>0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1</td>
<td>1 which indicates the 1st sub-component (MR_Idle) will be executed.</td>
<td>Test Passed</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>U21</td>
<td>Check that the 'MR_Evaluate_Conditions' atomic component outputs the correct value to indicate the 'MR_In_Progress' sub-component (which is the 2nd sub-component) needs to be executed based on the values input.</td>
<td>MR_Evaluate_Conditions</td>
<td>1, 1, 1, 0, 0, 1, 1, 1, 1, 1</td>
<td>2 which indicates the 2nd sub-component (MR_In_Progress) will be executed.</td>
<td>Test Passed</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>U22</td>
<td>Check that the 'MR_Evaluate_Conditions' atomic component outputs the correct value to indicate the 'MR_Abort' sub-component (which is the 3rd sub-component) needs to be executed based on the values input.</td>
<td>MR_Evaluate_Conditions</td>
<td>0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1</td>
<td>3 which indicates the 3rd sub-component (MR_Abort) will be executed.</td>
<td>Test Passed</td>
<td>None</td>
<td></td>
</tr>
<tr>
<td>U23</td>
<td>Check the 'Manual_Refuel_Init' atomic component sets the values of the system state variables correctly by ensuring only the 'SS_MR_ACTIVE' system state variable is true (1).</td>
<td>Manual_Refuel_Init</td>
<td>0, 0, 0, 0, 0, 0</td>
<td>Only the SS_MR_ACTIVE system state variable should have a value of 1 which is the 2nd value output below.</td>
<td>Only the SS_MR_ACTIVE system state variable had a value of 1.</td>
<td>Test Passed</td>
<td>None</td>
</tr>
</tbody>
</table>

Output values = 0, 1, 0, 0, 0, 0, 0
**Integration Tests**

<table>
<thead>
<tr>
<th>Test Ref No.</th>
<th>Comment</th>
<th>Test Input</th>
<th>Expected Outcome</th>
<th>Actual Outcome</th>
<th>Actions Taken</th>
</tr>
</thead>
</table>
| I1           | Check that when the status of the LWT is evaluated the correct sub-component is executed and the correct system state value is output. | Component name: MR_TK_LWT  
Input Values: 1, 0, 1 | MR_LWT_Eval_Status component outputs value 2.  
MR_LWT_Process component uses this value to execute sub-component 2 (MR_LWT_Active)  
Output value = 1 for system state variable 'SS_MR_LWT' | Test Passed  
MR_LWT_Eval_Status component outputs value 2.  
MR_LWT_Process component executes sub-component 2 (MR_LWT_Active)  
Value of 1 was output for system state variable 'SS_MR_LWT' | None |
| I2           | Check that the 'MR_In_Progress' composite component executes correctly and all system state values output are correct. | Component name: MR_In_Progress  
Input Values: 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0 | All sub-components are executed in sequence.  
Output 6 values = 0, 1, 0, 1, 1, 0 | Test Passed  
All sub-components were executed in sequence.  
6 values were output = 0, 1, 0, 1, 1, 0 | None |
| I3           | Check that when the status of the LWT is evaluated the correct sub-component is executed and the correct system state value is output. | Component name: MR_TK_LWT  
Input Values: 0, 1, 1 | MR_LWT_Eval_Status component outputs value 1.  
MR_LWT_Process component uses this value to execute sub-component 1 (MR_LWT_Idle)  
Output value = 0 for system state variable 'SS_MR_LWT' | Test Passed  
MR_LWT_Eval_Status component outputs value 1.  
MR_LWT_Process component executes sub-component 1 (MR_LWT_Idle)  
Value of 0 was output for system state variable 'SS_MR_LWT' | None |
| I4           | Check that when the status of the RWT is evaluated the correct sub-component is executed and the correct system state value is output. | Component name: MR_TK_RWT  
Input Values: 1, 0, 1 | MR_RWT_Eval_Status component outputs value 2.  
MR_RWT_Process component uses this value to execute sub-component 2 (MR_RWT_Active)  
Output value = 1 for system state variable | Test Passed  
MR_RWT_Eval_Status component outputs value 2.  
MR_RWT_Process component executes sub-component 2 (MR_RWT_Active)  
Value of 1 was output | None |
<table>
<thead>
<tr>
<th></th>
<th>Component name:</th>
<th>Input Values:</th>
<th>Component outputs value</th>
<th>Component outputs value</th>
<th>Test Passed</th>
<th>Output value</th>
<th>System state variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>MR_TK_RWT</td>
<td>0, 1, 1</td>
<td>1</td>
<td>0</td>
<td>Test Passed</td>
<td>0</td>
<td>'SS_MR_RWT'</td>
</tr>
<tr>
<td>16</td>
<td>MR_TK_CT</td>
<td>1, 0, 1</td>
<td>2</td>
<td>1</td>
<td>Test Passed</td>
<td>1</td>
<td>'SS_MR_CT'</td>
</tr>
<tr>
<td>17</td>
<td>MR_TK_CT</td>
<td>0, 1, 1</td>
<td>1</td>
<td>0</td>
<td>Test Passed</td>
<td>0</td>
<td>'SS_MR_CT'</td>
</tr>
<tr>
<td>18</td>
<td>MR_Surge_Protection</td>
<td>1, 1, 1, 1</td>
<td>1</td>
<td>1</td>
<td>Test Passed</td>
<td>1</td>
<td>'SS_MR_RWT'</td>
</tr>
<tr>
<td>Component 1 (SP_Open)</td>
<td>executes sub-component 1 (SP_Open)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------</td>
<td>-------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output value = 1 for system state variable ‘SS_MR_SURGE’</td>
<td>Value of 1 was output for system state variable ‘SS_MR_SURGE’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| 19 | Check that when the status of surge protection is evaluated the correct sub-component is executed and the correct system state value is output. |
| Component name: MR_Surge_Protection | SP_Eval_Status component outputs value 2. |
| Input Values: 0, 0, 0 | SP_Process component uses this value to execute sub-component 2 (SP_Shut) |
| | Output value = 0 for system state variable ‘SS_MR_SURGE’ |
| Test Passed | SP_Eval_Status component outputs value 2. |
| | SP_Process component executes sub-component 2 (SP_Shut) |
| | Value of 0 was output for system state variable ‘SS_MR_SURGE’ |
| None |

| 110 | Check the ‘Manual Refuel Execution’ composite component executes correctly and outputs the correct system state values based on the values input. |
| Component name: Manual Refuel Execution | The ‘Manual Refuel Execution’ composite component should execute correctly. |
| Input Values: 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 | Output 14 values = 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1 |
| Test Passed | The ‘Manual Refuel Execution’ composite component executed correctly. |
| | 14 values output= 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1 | None |
## System Tests for all the Fuel Operations

<table>
<thead>
<tr>
<th>Test Ref No.</th>
<th>Comment</th>
<th>Test Input</th>
<th>Expected Outcome</th>
<th>Actual Outcome</th>
<th>Actions Taken</th>
</tr>
</thead>
</table>
| S1 | Check that the ‘Manual Refuel Op’ composite component fully works as expected when given values are input into the simulator. | Component name: Manual Refuel Op  
Input Values: 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1 | The ‘Manual Refuel Op’ composite component should execute successfully without any error.  
Output 14 values = 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1 | Test Passed  
The ‘Manual Refuel Op’ composite component executed successfully without any error.  
14 values were output = 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1 | None |
| S2 | Check that the ‘Automatic Refuel Op’ composite component fully works as expected when given values are input into the simulator. | Component name: Automatic Refuel Op  
Input Values: 1000, 500, 1000, 300, 1000, 400, 0, 0, 0, 800, 100, 5, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 800, 100, 5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 | The ‘Automatic Refuel Op’ composite component should execute successfully without any error.  
Output 18 values = 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1 | Test Passed  
The ‘Automatic Refuel Op’ composite component executed successfully without any error.  
18 values were output = 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1 | None |
| S3 | Check that the ‘Defuel Op’ composite component fully works as expected when given values are input into the simulator. | Component name: Defuel Op  
Input Values: 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 | The ‘Defuel Op’ composite component should execute successfully without any error.  
Output 17 values = 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1 | Test Passed  
The ‘Defuel Op’ composite component executed successfully without any error.  
17 values were output = 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1 | None |
| S4 | Check that the ‘Ground Transfer Op’ composite component fully works as expected when given values are input into the simulator. | Component name: Ground Transfer Op  
Input Values: 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1 | The ‘Ground Transfer Op’ composite component should execute successfully without any error.  
Output 15 values = 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1 | Test Passed  
The ‘Ground Transfer Op’ composite component executed successfully without any error.  
15 values were output = 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1 | None |
| 55 | Check that the ‘Shut-Off Test Op’ composite component fully works as expected when given values are input into the simulator. | **Component name:** Shut-Off Test Op  
**Input Values:** 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0 | The ‘Shut-Off Test Op’ composite component should execute successfully without any error.  
Output 12 values = 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0 | Test Passed  
The ‘Shut-Off Test Op’ composite component executed successfully without any error.  
12 values were output = 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1 |

| 56 | Check that the ‘Off Op’ composite component fully works as expected when given values are input into the simulator. | **Component name:** Off Op  
**Input Values:** 1, 0, 0, 0, 0, 0 | The ‘Off Op’ composite component should execute successfully without any error.  
Output 7 values = 0, 0, 0, 0, 1, 1 | Test Passed  
The ‘Off Op’ composite component executed successfully without any error.  
7 values were output = 0, 0, 0, 0, 1, 1 | None |