Institutionen för systemteknik
Department of Electrical Engineering
Examensarbete

Decision Support System for Fault Isolation of JAS 39 Gripen - Development and Implementation

Examensarbete utfört i Fordonssystem
av

Anders Holmberg
Per-Erik Eriksson

Rapport 3839
Linköping 2006

Decision Support System for Fault Isolation of JAS 39 Gripen

- Development and Implementation

Master Thesis
Department of Electrical Engineering
Linköping University

Anders Holmberg
Per-Erik Eriksson

LITH-ISY-EX--YY/3839--SE

Supervisor: **Carolina Romare**
Johan Rättvall
Jonas Biteus
Examiner: **Erik Frisk**
Linköping, 20 June 2006

Abstract

This thesis is a result of the increased requirements on availability and costs of the aircraft Jas 39 Gripen. The work has been to specify demands and to find methods suitable for development of a decision support system for the fault isolation of the aircraft. The work has also been to implement the chosen method. Two different methods are presented and a detailed comparison is performed with the demands as a starting point. The chosen method handle multiple faults in $O(N^2)$-time where N is the number of components. The implementation shows how all demands are fulfilled and how new tests can be added during execution. Since the thesis covers the development of a prototype no practical evaluation with compare of manually isolation is done.

Acknowledgment

We would like to thank our supervisor Jonas Biteus and examiner Erik Frisk for the guidance and discussion that led to the results. We like to thank Carolina Romare and Johan Rättvall for all guidance at Saab, setting up meetings with several people, and also for the interesting discussions of how Gripen and a large company like Saab works.

Anders
Finally would I like to send a thought to Gunnar Fogelberg, my former teacher in Analys A at MAI, Linköping University. He encouraged me and made me realize the fun and usability of mathematics. I would not be sitting here writing this thesis without the inspiration I got from you. Rest in peace.

Anders Holmberg and Per-Erik Eriksson, Linköping, 2006

Contents

Chapter 1

Introduction

The Aircraft Service Division is a part of Saab Aerosystems which is a business area within Saab AB. It runs development, modification and also flight and maintenance service of civil and military aircrafts. Our work has been carried out at the section of maintenance and service engineering.

1.1 Background

The requirements of increased availability and reduced costs of the aircraft Jas 39 Gripen are continuously being raised. Both the time and the accuracy to perform fault isolation have to be improved. A lot of time is consumed since fault isolation is often made by hand by an experienced technician. To fulfill the increased requirements a workstation that does the fault isolation automatically is highly desirable.

1.2 Purpose

The purpose of this thesis is to develop a decision support system for fault isolation of Jas 39 Gripen. This includes the evaluation of possibilities, specifying demands and building a prototype.

1.3 Limitations

The purpose of this thesis is to develop a prototype of a decision support system. There are no intentions of building a system for the complete aircraft, and there are no intentions of collecting the probability of failure for every single component. The intention *is* to investigate the possibilities of a decision support system for fault isolation and how this system can be further developed for the entire aircraft.

1.4 Thesis Outline

The thesis starts with four introductorily chapters: Chapter 2 gives an introduction to the aircraft, its fuel system and its diagnostic monitoring equipment; Chapter 3 describes the available documentation of Jas 39 Gripen and measuring data collected during flight. It also contains the demands we have specified for the decision support system; Chapter 4 explains the field of fault detection and isolation; Chapter 5 explains the field of probabilistic reasoning systems used in decisions support systems.

Our work is mainly described in the following four chapters:
Chapter 6 contains two different methods invented to fit the requirements. In Chapter 7 the methods are examined against the demands and each other. It ends with a conclusion of which one is the most suitable to the demands. In Chapter 8 method 2 has been implemented. Our work ends with Chapter 9 that contains a conclusion of the system and its possibilities.

1.5 Contributions

Our contribution to the scientific community with this thesis is:

- Interpreting Saab's wishes on the development of a decision support system. Chapter 3.
- Accumulating the demands for a decision support system. Explaining what abilities and functionality the system must have to fulfill the wishes. Chapter 3.
- Development of two methods suitable for the problem:
  Method 1: Agents.
  Method 2: Extended Structured Hypothesis tests. Chapter 6.
- Evaluation of the methods and explaining why Method 1 is not enough for the decision support system. Chapter 7.
- Implementation of Method 2 and the hypothesis tests. Chapter 8.
- Summarizing the work and suggesting how to continue the development of the system. Chapter 9-10.

Chapter 2

Introduction to the Aircraft

This chapter is an introduction to the aircraft, its fuel system and its diagnostic monitoring equipment. Its purpose is to give the reader a deeper understanding of the components and functions of the aircraft. The information may be needed when reading Chapter 8 and a suggestion is to read this chapter lightly and when reading Chapter 3 to Chapter 9 take a peek in this chapter to get the deeper understanding. Since the work has been concentrated to the fuel system, it is only that system that is described.

2.1 Components in the Fuel System

Before a more comprehensive description of the fuel systems structure and functionality is made, there is a need to describe some of the components in the system. Following is a short survey of the most important components in the fuel system. The fuel system in Gripen consists of many more components than what is presented in this chapter. The ones described below are the basic components for understanding this thesis. To have a basic understanding of the fuel system and the components within it, also helps when trying to understand the different hypothesis tests that are presented later on in this thesis.

When a component is mentioned in the thesis it means a mechanical unit that can be of different size and extent. A component is not necessarily the smallest part in the aircraft and one component can consist of other components. An example of this the ARTU, that contains valves. Both the ARTU and valves are referred to as components. The components that were just mentioned are described later in this chapter.

2.1.1 Forward Refueling/Transfer Unit

The forward refueling/transfer unit, abbreviated FRTU, can in short terms be described as a unit for transferring fuel between different tanks in the aircraft. The tanks that are connected to the FRTU are the fuselage tanks. This is illustrated in Appendix A.

2.1.2 Afterward Refueling/Transfer Unit

Just like the FRTU, the afterward refueling/transfer unit (ARTU) is a unit for transferring fuel in the aircraft. However it is not as advanced and does not have as big area of responsibility as the FRTU. The ARTU is located at the rear end of the fuel system (for more details see Appendix A) and it has two main purposes. The first one is to supply fuel to the tank T1A (see Appendix A for location), the wing tanks and the drop tanks during refueling. The second purpose is to control the fuel during the transfer from the wing tanks and the drop tanks to the FRTU. The ARTU has seven inlets/outlets and six of these each have a vent valve connected to the inlet/outlet..

2.1.3 Probes

To be able to measure the amount of fuel that a tank contains there are probes located in each tank in the aircraft. In some tanks there are two probes, like the wing tanks and the tank T2A (see Appendix A for location). There are a total of 16 probes in the aircraft, not counting the drop tanks which have three each.

2.1.4 Valve

Valves are found at different places in the fuel system but foremost they are located in the refueling/transfer units and in the Controlled Vent Unit. A valves purpose is to control the flow of the fuel (which means letting through or turning off the flow) through a pipe.

2.1.5 Sensors and Switches

There are two kinds of sensors in the fuel system, low level sensors (LLS) and high level sensors (HLS). Sensors and switches are both binary units that has two states. A low level sensor indicates if a tank is empty and a high level sensor indicates if it is full. There are three LLS and one HLS distributed among the aircrafts different fuel tanks. The tanks containing sensors are VT, T1F and the wing tanks. When it comes to switches there are a few different types in the fuel system, but only one is of any interest concerning this thesis. The interesting type is the float switch that is located in the drop tanks. These switches have the functionality as LLS and indicate if a tank is empty or not.

2.2 Fuel Tanks

The fuel in Gripen is stored in several different tanks that are placed in different parts of the aircraft body. The fuel tanks placement in Gripen is shown in Figure 2-1. The fuel system also consists of a cooling system and a pressure system. In Gripen the fuel is, apart from running the engine and some smaller units, also used to cool different devices. The purpose of the pressure system is to keep most of the tanks pressurized. This is to ease the fuel transfer and to avoid cavitations in the pumps. [1]

There are ten different fuel tanks in Gripen, including the drop tanks. Some of these tanks are divided into two smaller tanks, mostly a front and a rear part. The ten different tanks are: tank 1 (T1), tank 2 (T2), tank 3 (T3), vent tank (VT), negative-g tank (NGT), left wing tank, right wing tank and the centre, right and left drop tank. In accordance with Figure 2-1, T2 is located in the front of the aircraft followed by VT, T1 and T3 furthest to the rear in the aircraft. The drop tanks are not shown in the figure but are, if used, hung underneath the fuselage and wings. T1 and T2 are the tanks divided into a smaller front and rear tank, which are called forward tank and aft tank (T1 = T1F + T1A and T2 = T2F + T2A). In Gripen version B and D (twin seaters), the T2F has been removed to make room for the extra seat. The wing tanks are also divided into two different tanks. They are called tank 4 (T4) and tank 5 (T5). The collector tank constitutes of tank T1 and the NGT.

For the aircraft to be able to measure the quantity of fuel left, there is a number of contents probes in each fuel tank. Every tank in the fuselage has one contents probe, except for tank T2A what has two. There are four probes in each wing and three in each drop tank.

The fuel system is controlled by the GECU (General systems Electronic Control Unit), except for some functions that are controlled by the AIU (Aircraft Interface Unit). For more details about which functions the AIU control, see section 2.4.1. The GECU is an integrated digital control unit that controls three systems in the aircraft:

- Hydraulic System (HS)
- Environmental Control System (ECS)
- Fuel System (FS)

The GECU is located behind tank T3 and its function is to measure, monitor and control the three systems it is responsible for. [2]
The GECU communicates with a system computer (SysC) whose main tasks are to calculate the center of gravity, calculate the load vector and perform part of the Safety Check. [1]

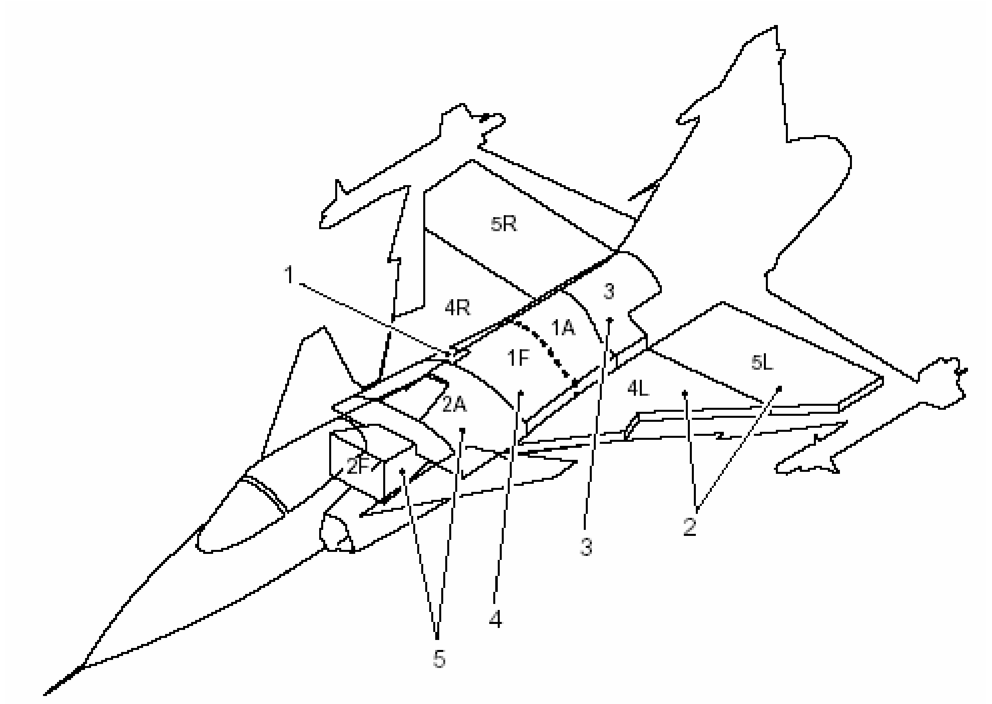


Figure 2-1. The different fuel tanks in Gripen

1: Vent tank, 2: Wing tanks, 3: Rear tank, 4: Collector tank, 5: Forward tank

The engine in the aircraft is fed with fuel from the boost pump that is located in the NGT. In addition to supplying the engine with fuel the boost pump also has to supply the heat exchangers with fuel and the jet pumps with fuel flow. If the boost pump should malfunction the transfer pump does the feeding of fuel to the engine instead and if the transfer pump also should break, tank T1 is pressurized and the engine can suck fuel itself. Even without pressurization the engine can suck fuel itself, as long as the aircraft is at a low altitude and with limited fuel consumption. [1]

2.3 Fuel Transfer

The fuel used in Gripen is always taken from the collective tank, i.e. the negative-g tank (NGT) plus tank T1. This conveys that the aircraft have to transfer fuel between different tanks to make sure that the collective tank never runs out of fuel. [1]

The fuel transfer is mostly done by the transfer pump and the jet pumps. A jet pump is a device in which a small jet of fluid in rapid motion moves, by its impulse, a larger quantity of the fluid. In Gripen there are five jet pumps and they are located in the tanks T1, T2 and NGT. The main purpose of the transfer pump in the fuel system is to transfer fuel between different tanks in the aircraft. As seen in appendix A, the transfer pump is located in the Forward Refueling Transfer Unit (FRTU). The GECU is able to limit the maximum speed of the transfer pump. Such a limit will occur at high altitudes, high pitch angles and high load factors. It can also occur if the hydraulic pressure decreases because of a malfunction in the hydraulic system. [1]

When the engine has a large output thrust, the jet pumps in fuel tank T2 and T3 operate in parallel with the transfer pump. The transfer pump will stop when all tanks except for T1 are empty. [1]

2.3.1 The Order of Fuel Transfer

Since the engine gets its fuel from the NGT, the aircraft has to make sure that it always stays full. This is done by transferring fuel between the different tanks in a specific order. The drop tanks (if any) are emptied first, in the order: the left and right drop tank first and then the center drop tank. When the drop tanks are empty, fuel is taken from tank T2 down to 200 kilograms and after that the fuel is taken from the wing tanks. When the wing tanks have been emptied the fuel is taken from tank 2, 3 and finally also from tank 1. [1]

The load factor of an aircraft is a measure of the aircrafts external load. The value of the factor is the same as the length of the load vector, which is shown in Figure 2-3. During flight conditions with a high load factor, the transfer pump cannot supply sufficient fuel from the drop tanks to T1. Therefore the fuel is moved from the wing tanks instead of the drop tanks, even though the drop tanks may contain fuel. The reason for this is that it is easier to transfer fuel from the wings than from the drop tanks. Another reason is the risk of cavitations of the transfer pump. The definition of high load factor is when the load factor is more than 3 g or when it is more than 1.5 g in combination with a greater altitude than 9 km. [3]

2.4 Monitoring and Measuring

One of the objects with the measuring system in Gripen is to control the fuel transfer and torrent. The fuel quantity is measured with contents probes individually in every tank. The measured signal is processed in the GECU, where the total remaining fuel quantity and aircrafts center of gravity is calculated. [1]

A fault that can occur is failure with the cable to the probe. As a result of this fault an incorrect amount of fuel will be displayed.

2.4.1 Function Monitoring

Function monitoring (FM) is the internal supervision in the fuel system. FM is automatically conducted continuously and its primary purpose is to monitor the system during operation and also to warn the pilot of malfunctions. In the fuel system, FM is mostly performed by the GECU but some parts are done by the AIU. [4]

The AIU has the following functions for the fuel systems:

- Start and stop the boost pump
- Start and stop the RCS (Radar Cooling System) pump
- Function Monitoring of the LP cock operation and the RCS

- Control of shut-off valve to EWS and leak monitoring of the cooling circuit for the FPU
- Fault warnings

Warnings are sent from the GECU to the AIU on the data bus if faults occur during the FM. [1]

2.4.2 Safety Check

The purpose of the Safety Check (SC) in the fuel system is to check the status of the fuel system at aircraft startup. The SC is done by the SysC in collaboration with the GECU. [4]

2.4.3 Fuel Measure

The measurement equipment in the fuel system measures and monitors the following:

- Fuel level
- Fuel temperature
- Air pressure in the fuel tanks
- Pump pressure

The fuel quantity that is being displayed to the pilot is shown in percent. From the beginning 100% was equal to full internal tanks. This is still true for the versions B and D (twin seaters) of the Gripen aircraft. Today, full internal tanks are in total 112% (for the single seated versions of Gripen). The reason for the extra 12% is the added tank T2F, located first in the aircraft fuselage. With three extra drop tanks the fuel quantity can come up to more than 112%. The quantity of 1% of fuel is same for all versions of aircraft 39. [1]

2.4.4 Probe Failure

The fuel quantity indication only displays fuel that is available. If the fuel in a tank isn't available the GECU consider that tank empty. If a contents probe malfunctions the fuel quantity will be displayed in accordance to Figure 2-2 below. Because of this, the data about the remaining fuel quantity can change quickly when there is a probe failure. [1]

Tank	Electrical Identification	Effects
T2F	1QB	If the probe malfunctions, the tank is considered empty.
T2A	2QB	While T2F has fuel, T2A is considered full. In other condition, the quantity in T2A is calculated from 3QB with decreased precision
T2A	3QB	While T2F has fuel, T2A is considered full. In other condition, the quantity in T2A is calculated from 2QB with decreased precision
VT	4QB	If the probe malfunctions, the tank is considered empty.
T1F		While VT has fuel, TF1 is considered full. In other conditions, the tank is considered empty.
T1A	6QB	If there is fuel in T1F, a value is calculated for T1A (57% of the quantity in T1F)
NGT	7QB	While T1A has fuel, NGT is considered full. In other conditions, the tank is considered empty.
T3	8QB	If the probe malfunctions, the tank is considered empty.
T4	11QB, 12QB 13QB, 14QB	If one of the two probes in T4 malfunctions, the quantity is calculated from the remaining probe with decreased precision. If both probes malfunction, the tank is considered empty.
T5	21QB, 22QB 23QB, 24QB	If one of the two probes in T5 malfunctions, the quantity is calculated from the remaining probe with decreased precision. If both probes malfunction, the tank is considered empty.
Drop Tank		If the probe malfunctions, the tank is considered empty.

Figure 2-2 Probe failure effects

2.5 Load Vector

During flight, the different accelerations and gravity forces have an effect on the fuel system. This causes the fuel surface to tilt and therefore to effect the operation of the system. The different forces affecting the aircraft are summarized in the load vector, ñ, which must be considered during fuel transfer and measuring. The load vector, ñ, is calculated with a coordinate system that originates from the aircraft as reference. The pitch degree is derived from the angle between the z-axis and the load vector in x-line. In the same way the roll degree is calculated from angle between the z-axis and load vector in y-line. This is illustrated in Figure 2-3. In the illustrations the load vector and the reference coordinate system is shown. It is the SysC that calculates ñ and it can have three different states.

- ñ is within measurable range.
- ñ is out of measurable range but within transferable range.
- ñ is out of transferable range.

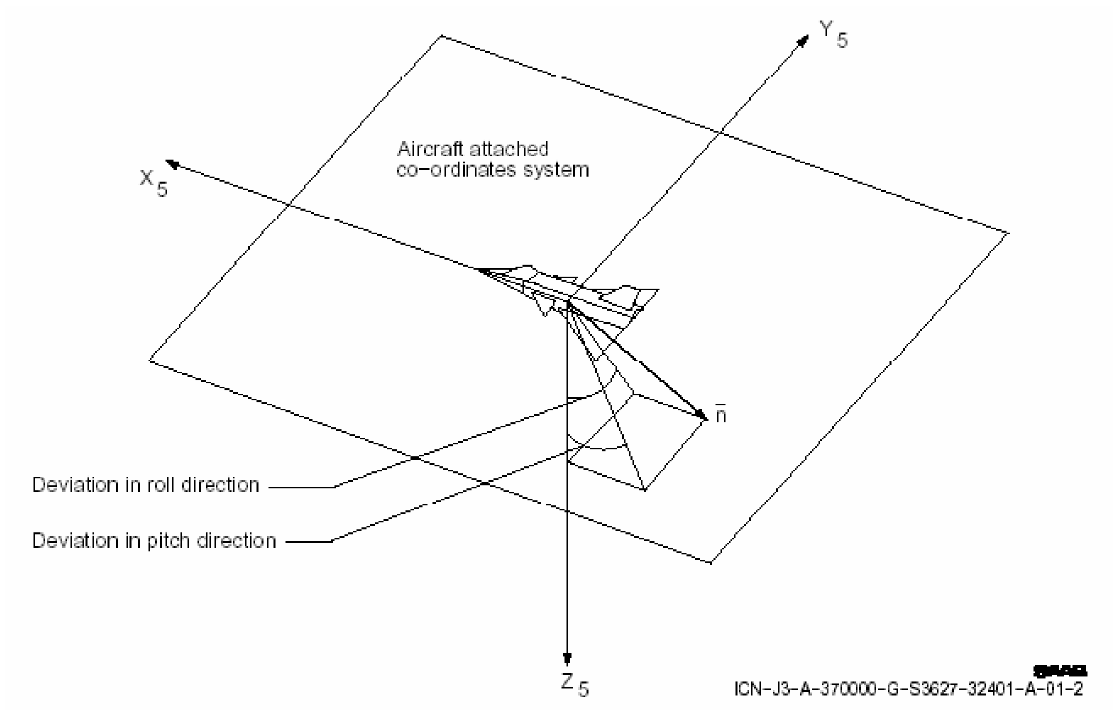
The different states are illustrated in Figure 2-4, where the inner filled box shows the restrictions for measurable range and the outer box shows the restrictions for transferable range. It is possible to measure fuel quantity only when ñ is within the measurable range. The measurable range is specified so that the direction of ñ in relation to Z_5 is not more than:

- - 5 degrees to + 20 degrees in pitch.
- ± 3 degrees in roll.

When ñ is out of the measurable range but within the transferable range the fuel tanks will still transfer fuel to T1 in the commonly set sequence. When ñ is out of the transferable range, the transfer pump stops and the larger part of the fuel transfer also stops. The transferable range is defined as:

- 5 degrees to + 80 degrees in pitch
- ± 10 degrees in roll

[1]



Y5
Aircraft attached co-ordinates system
X5
n̄
Deviation in roll direction
Deviation in pitch direction
Z5
ICN-J3-A-370000-G-S3627-32401-A-01-2

Figure 2-3 The load vector and its reference coordinate system

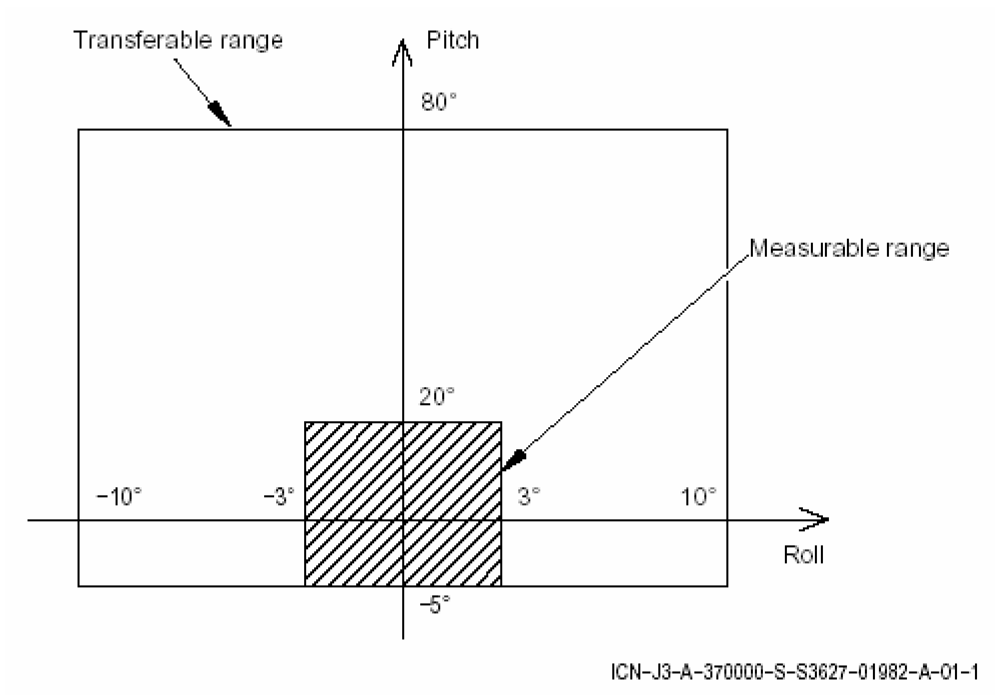


Figure 2-4 Measurable and transferable range

2.6 Fuel Air Pressure

During flight, most of the fuel tanks are supplied with an overpressure in relation to the ambient pressure. Tank T1 and the NGT are generally not kept pressurized. The reason for this is to ease the fuel transfer to tank T1 at lower altitudes. However there are a few exceptions to this rule. For example when all available fuel has been moved to fuel tank T1. Then T1 is pressurized to make sure that the supply to the engine operates as usual. Another exception is at high altitudes where there is a risk of cavitations in the boost pump and transfer pump. This can also occur at high fuel temperature. [1]

2.7 Existing Fault Isolation

The existing Fault Isolation system lists components depending on which alarms are raised. This system is unfortunately not developed for fault isolation but rather fault detection. This is done in order to inform the pilot whether he/she can fulfill the mission, abort it or switch to another mission when faults are detected. When the Functional Monitoring, FM, discovers faults in a subsystem, the subsystem can be shut down or blocked so that other subsystems can take care of the functionality. This gives a graceful degradation. A list of possible explanations to the faults is made up on basis of FM. This list ranks components after mean time between failure (MBTF) and costs and time for replacing the component. These alarms are based on strict logic and are handled in section 5.2.

Chapter 3

Prerequisites and Demands

This chapter describes the prerequisites for the work and the demands specified for the system that will be developed during this thesis.

3.1 Prerequisites

3.1.1 Documents

Besides the publications used in the former chapter, there is a large amount of documents covering JAS 39 Gripen. We have only used them to increase our own understanding of the aircraft and its functions. In a further development several important facts about dependencies between components and possibilities of failure can be found in FMEA (Failure Mode Effects Analysis), FTA (Fault Tree Analysis) and SSDD (Subsystem Design Description).

3.1.2 Data

Jas 39 Gripen has an onboard data storage system collecting measurement data for over 5000 variables. This system is referred to as RUF, Registration Used for maintenance and Flight security. Included in the RUF-data is a flight report which contains information about safety checks, function checks, and the risen alarms.
For the purpose of fault isolation using RUF-data a software toolkit called RUF-PD39 exists. This software toolkit is used by technicians to manually detect and isolate faults. [5]

Data is recorded in two ways: Continuous recording and conditional recording. The first continuously records some variables such as fuel quantity, altitude and mach. The latter starts recording when some condition is fulfilled, for example an extra altitude sensor is recorded during flight on low altitude. In both cases data compression is used to avoid running out of memory. Every variable that is recorded has a sampling frequency, often 1 Hz. If two or more following samples give almost the same value only the first value is recorded. The data compression is exemplified in Figure 3-1. In the figure, the value of the sensor is shown on the y-axis and the first measurement to be recorded is A. The following two samples do not vary enough from A and are therefore not recorded. The fourth sample, B, differs enough from A and is recorded. From this time on are further samples compared to B instead of A. How much a value can differ before it is recorded is called a window. [5]

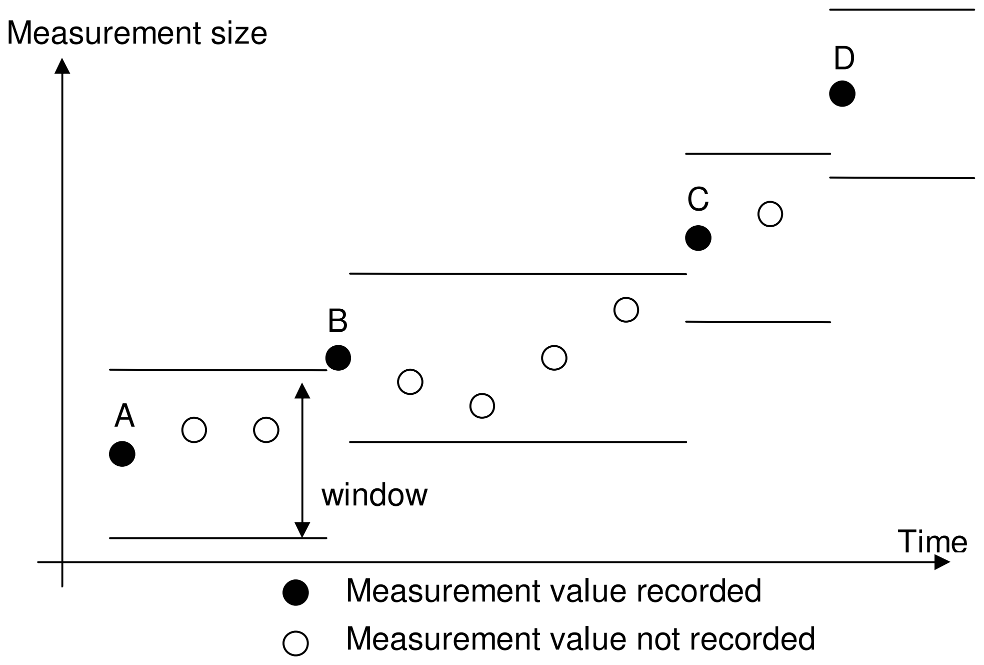

Measurement size
D
C
B
A
window
Time
● Measurement value recorded
○ Measurement value not recorded

Figure 3-1. Measurement values recorded with data compression

Conventional signal processing methods like mean value and filtering is not applicable on signals stored with data compression. To solve this, a sample and hold-function has been used with the signals originally frequency to estimate the samples that have not been recorded. After this the signals can be processed as ordinary signals without data compression. [6]

3.2 Demands on the System

3.2.1 Deterministic Fault Isolation

When hardware and software in the development of an aircraft has been tested and considered working, it is packaged to something called an *edition*. Two aircrafts from the same edition have to work equally. The same goes for software outside the aircraft and two fault isolation systems from the same edition fed with the same flight data have to result in the same output. Therefore it is not an option to have a system that could be altered after it has been packaged to an edition. This means that the system has to contain all knowledge from delivery and can not be trained by the end user. Technicians at Saab can however train the system to a certain level and package it to an additional edition.

3.2.2 Usable for a Less Experienced Technician

For advanced manual fault isolation in Gripen the experts use RUF-PD39 which is a software toolkit and there is no need for alternative software for them. The purpose of this thesis is to deliver a system for technicians less experienced than these experts, and therefore shall usage of the system require a low level of knowledge about the aircraft and RUF-PD39.

3.2.3 Application vs. Information

The information containing all knowledge about dependencies and probabilities of components and tests has to be updated when new information is gathered. A demand is that no new release of the application has to be installed during the update, but rather just the replacement of the files containing the information.

3.2.4 Configuration Management

Every aircraft is built up on a set of components. Due to service and modifications no aircrafts are identical. The decision support system has to be able to manage different configurations because of this.

3.2.5 Maintenance

The system has to be easy to maintain. It can not be built on ad hoc solutions and unstructured function calls. The information has to be handled in one place and not be spread out in several functions. The procedure of adding extra tests shall be equal for all tests and easy to handle, i.e. the insertion of new tests shall be handled the same independent of what the tests do.

3.2.6 Expansion

The system must be flexible and have potential for extension. It can not be built on a dead end that is not improvable.

3.2.7 Multiple Faults Isolation

The system obviously has to handle at least single faults otherwise it would not be a fault isolation system. A highly desirable feature is the ability to isolate multiple faults; we therefore consider that the system has to be able to isolate at least double faults.

3.2.8 Ranking of Components

If several components seem to be broken the system has to produce a list containing a score or probability for each component. With this score the list can be sorted in order to decide which component to replace first. This demand is a sub-demand of 3.2.2 since a less experienced technician does not know where to start if a list without scores is produced.

Chapter 4

Introduction to FDI, Fault Detection and Isolation

FDI is an abbreviation for Fault Detection and Isolation. This chapter is an introduction to terms used in this field. Most of this chapter is influenced by [7].

4.1 Fault Detection

The first step in a diagnosis and surveillance system is to detect if faults are present in the system. This can be done by limit checking, i.e. by raising an alarm when a value reaches a threshold. A common example is the lamp in a car indicating that the fuel level is low. The electronics does not tell you why the fuel is low, just that this is the case.

4.2 Fault Isolation

The second step in a diagnosis and surveillance system is to isolate the fault to a specific component by figuring out what could cause the system to react the way it does. In the previous example with low fuel level, this can be done by examining data from several sensors. By using a model of the fuel consumption fed with data of the engine speed you can calculate the fuel consumption. This way you can figure out if the fuel level is supposed to be low because of consumption or some other reason. If the fuel level sinks even when the engine speed is low there probably is some kind of leakage in the fuel system. One other thing to investigate is if the fuel level suddenly increases without any good reason. In this case you can suspect that the fuel sensor is broken and that the fuel level is lower than the one told by the instruments. A statement like this that can explain the measured sensor data is called a *diagnosis*. If several diagnoses are present it is important to have some method to rank them in order of possible failures.

4.3 Analytical Redundancy

If there are two or more ways of deciding a variable x using only observed variables z, i.e. $x=f_1(z)$ and $x=f_2(z)$, where $f_1(z)$ and $f_2(z)$ are different functions, then there exists an *analytical redundancy*.

The example above mentioned a model for fuel consumption. The outcome of the model was compared to a deduced fuel consumption based on measured fuel levels over the time. When there is a possibility to calculate the same thing in two different ways there is *analytical redundancy* in the system. This is one of the cornerstones of FDI and when the two ways end up with different values you can conclude that the system contains a faulty component.

4.4 Residuals

A function constructed the way that it is close to zero when the system is in a fault free mode, and apart from zero when a fault is present is called a *residual*. By using the functions mentioned earlier a residual *r* can be $r = f_1 - f_2$. When *r* is far from zero it can be concluded that either f_1 or f_2 use values inconsistent with the model.

4.5 Structured Hypothesis Tests

By examining several residuals it is possible to decide which component that raised the residuals. A *(binary) Hypothesis test* is defined as the problem to choose one of two unique states. One example is to choose between a hypothesis, $H_0^1 = \textit{no fault present}$ and another hypothesis, $H_1^1 = \textit{fault present}$. The upper index indicates the number of the hypothesis test and the lower index separates the two hypothesis in a hypothesis test. The Hypothesis Test decides which hypothesis is true. To create a hypothesis test a *test quantity* is needed. A test quantity is a function that is close to zero in the fault free case and apart from zero when faults are present. A residual is a good example of a test quantity. A test quantity, T_1, is close to zero when H_0^1 is true and non-zero when H_1^1 is true.

Since noise and model faults exist it is not feasible to demand the test quantity to be zero in the fault free case. Instead it is interpreted as zero as long as the value is below a certain level or threshold. Another test quantity T_2, can decide whether the hypothesis $H_0^2 = \textit{no fault or only fault } F_1 \textit{ is present}$ or $H_1^2 = \textit{any of the other faults are present}$ are true. By using several test quantities, fault detection and isolation can be performed. One way to do so is to set up a matrix over the available tests and the components to supervise. Figure 4-1 shows a matrix of dependencies between components and tests. The matrix is called a decision table, or decision matrix, and a cell containing ‘X’ indicates that this component can make the test of that row react. A cell containing ‘0’ indicates the opposite; that the component in no way can make the test react.

Example:
Test1 is influenced by Comp1, Comp2 and Comp3. Test2 is influenced by Comp2 and Comp4. Test3 is influenced by Comp3 and Comp4. When Test2 and Test3 have reacted, and Test1 has not, Comp2-4 can be broken. This is indicated by the circles, Test1 is grayed out because it has not reacted.

Dependency	Comp1	Comp2	Comp3	Comp4	Reacted
Test1	X	X	X	0	False
Test2	0	X	0	X	True
Test3	0	0	X	X	True

Figure 4-1 Example of decision table showing connection between components and tests

Structural hypothesis tests are used to find single-faults and the only component that can explain this test result is Comp4 since it affects both Test2 and Test3.

Chapter 5

Introduction to Probabilistic Reasoning Systems

In an ideal world there would be no reason not to trust the test quantities mentioned in Chapter 4. An absence of false alarms or missed alarms would be a comfortable environment for fault isolation. This chapter explains *uncertainty* to highlight the difficulties that arise when we leave the ideal world. It also covers two systems, one that handles uncertainty, and one that does not. Section 5.1 deals with uncertainty and explains the difficulties when signals are not reliable. Section 5.2 deals with theories not handling uncertainty and section 5.3 deals with theories that does. Most of this information is influenced by [8] and [9].

5.1 Uncertainty

A test is supposed to decide if some event has occurred, if some signal is within reasonable levels, if the fuel level drops according to the fuel consumption etcetera. For all tests a limit has to be set up to separate faulty cases from fault free cases. Figure 5-1 shows the upper and lower thresholds for a test that reacts if the sensor for the fuel level claims that there is more fuel left than the tank can contain, or that the level is lower than zero.

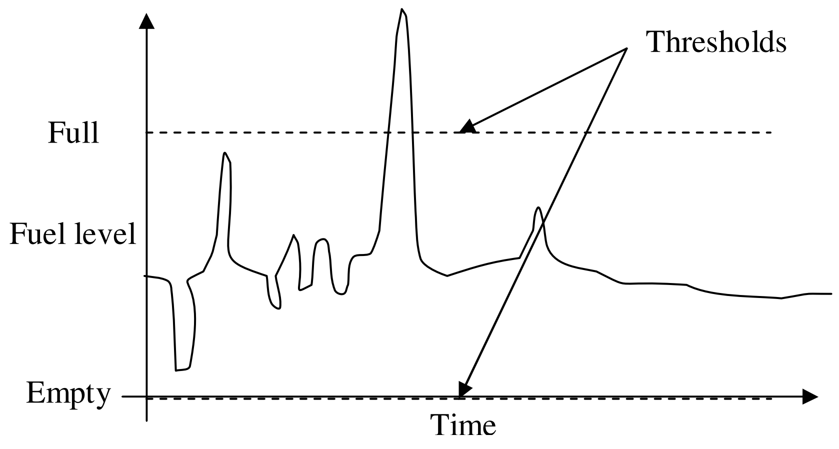


Figure 5-1 Thresholds for some sensor data.

This limit is called *threshold*. The work of setting the thresholds for tests is a large theory on its own, that's for example uses likelihood ratio on statistics and adaptive thresholds that change the limit depending on the environment. We shall not loose our self in this more than to establish two certain rules that always holds:

- If the threshold is set too low the test will react on normal behavior in the fault free case.
- If the threshold is set too high the test will not react even when a fault is present.

The first leads to *False alarms* and the latter to *Missed alarms* and bring uncertainty to the system.

5.1.1 False Alarms

If a system that trusts its alarms is exposed to false alarms, the wrong diagnosis will be deduced. When all tests react correct the broken component is isolated. If some tests react false, i.e. reacts when they should not have reacted, some other, possibly functioning, component will be isolated.

5.1.2 Missed Alarm

If a system has logical rules based on test results a rule will never be used as long as the test results do not suite the rule. If a certain test has to be true for a component to be considered broken by a rule, the component will never be considered broken as long as this test is false. If this test actually should be *true* but anyhow returns *false* a missed alarm is present.

This uncertainty has to be handled in order to build a good working decision support system. One possibility that has proven to be useful in [8] is Bayesian models which is a subset of the bigger theory of Bayesian network, also known as Belief network. To understand this possibility, the simpler theory of strict logical reasoning has to be studied first.

5.2 Strict Logical Reasoning

Strict logical reasoning is a propositional logic that never questions earlier decisions [8]. A set of logical rules are put together to give the output. Figure 5-2 shows an example of four rules specifying the output. The rule-based logic used in this example is *exclusive or*.

Rule nr	Test1	Test2	Output
1	True	True	False
2	True	False	True
3	False	True	True
4	False	False	False

Figure 5-2 Example of rule-based logic

As seen in the figure rule nr 1 specifies that if both test1 and test2 have reacted the output is false. If one of the tests have reacted the output is true, according to rule nr2 and nr3. Rule nr4 says that if both tests are false, the output is also false.

Rule-based logic like this will be used in section 6.1 to determine if components are broken. The drawback is that the logic gets really vulnerable for false and missed alarms. One can see that if Test1 or Test2 gives the wrong answer, the output also will be wrong. To handle false alarms as well as missed alarms a theory called uncertain reasoning can be used.

5.3 Uncertain Reasoning

The earliest expert systems developed in diagnosis are based on strict logical reasoning and did not handle any uncertainty. Rather soon the developers realized that this was insufficient for large systems. Expert systems presented later on all contain techniques for handling uncertainty. Belief network is the approach we have chosen to use. Some other approaches will be mentioned in section 5.3.2.

5.3.1 Belief Network

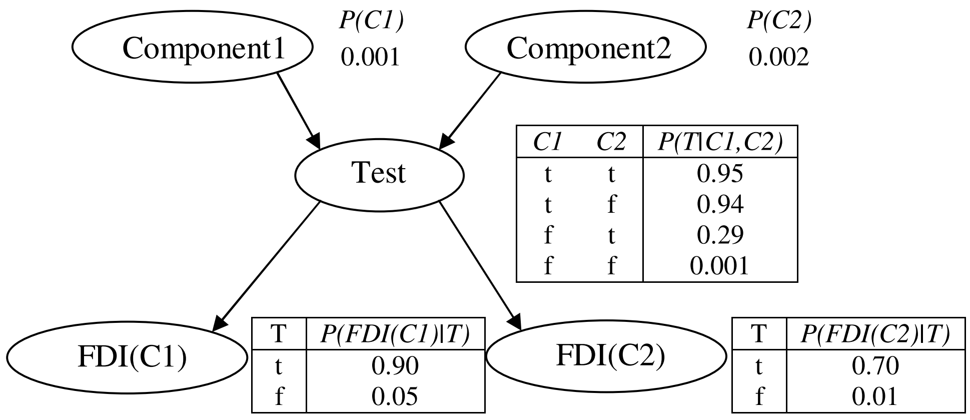
Belief networks are about specifying how possibilities for *query* are influenced by earlier facts, *evidence*. The notation used is $P(query \mid evidens)$ and *P(query)*. The latter is used for unconditional, prior probability that the proposition *query* is true. It is important to remember that this probability only is applicable when no evidence is known. As soon as other evidence B are known *conditional probability P(A|B)* should be used instead in order to get a more correct calculation. As soon as further evidence C is known the conditional probability $P(A \mid B \wedge C)$ should be used. The prior probability can be seen as a special case of conditional probability when no evidences are known. If C does not affect A when B is known, A and C are said to be conditional independent and *P(A|B)* can be used anyway.

A probabilistic inference system is used to calculate the posterior probability for a set of query variables, given values for the evidence variables. This means that the system calculates *P(query | evidence)*. A *Conditional Probability Table* that states the probability for a special event given the depending evidence can be set up. Figure 5-3 shows the probability that *Test* reacts given that *Component1* or *Component2* is broken or not. The condition nr1 specify that *Test* will react with a certainty of 95% if both *Component1* and *Component2* are broken=true.

Condition nr	Component1	Component2	P(Test|Component1, Component2)
1	True	True	0.950
2	True	False	0.940
3	False	True	0.290
4	False	False	0.001

Figure 5-3 Conditional Probability Table

These values and their origins can be drawn in a topology showing how components and test influence the fault detection and isolation. Figure 5-4 displays the topology of Fault Detection and Isolation of two components using one test. *P(C1)* is the probability that component1 is broken. *P(FDI(C1))* is the probability that C1 will be the considered broken by the Fault Detection and Isolation system. The table of *C1, C2* and *P(T)* contains conditional probabilities that the test will react given the four combination of t=true and f=false. The table of T=test and *P(FDI(C1))* contains probabilities that C1 will be considered broken given that the test has reacted=t or not=f.



C1	*C2*	*P(T|C1,C2)*
t	t	0.95
t	f	0.94
f	t	0.29
f	f	0.001

T	*P(FDI(C1)|T)*
t	0.90
f	0.05

T	*P(FDI(C2)|T)*
t	0.70
f	0.01

Figure 5-4 Bayesian network with topology and the conditional probability tables

The science of Bayes rules is large and need to be read in full in for example [9]. Because of that, a further description is left out and the short introduction is only present to show that we build our discussion of probabilities for false and missed alarms on solid ground.

5.3.2 Other Approaches

Several theories for handling uncertainty have been introduced in the field of probabilistic reasoning. For the interested readers are four of them are mentioned here:

- Default reasoning
- Rule-based method for uncertain reasoning
- Representing ignorance with Dempster-Shafer
- Representing vagueness with Fuzzy Logic

Descriptions can be found in [8] and [9].

Chapter 6

Two Different FDI Methods

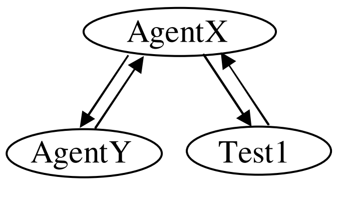
Two fundamentally different approaches to fault isolation will be discussed in this chapter. The first approach starts with the list of faulty components generated from the existing fault isolation. For each component it uses an agent to investigate the status of the component. The agents' task is to decide whether the component is broken or not. The second approach starts by looking at all available tests and tries to find out what component that can explain most of the test results.

6.1 Method 1: Agents

As described section 2.7, a ranked list of components is generated when faults are detected by the existing fault isolation system. The accuracy of this list has to be increased and to do this method 1 is invented.

The fundamental part of the method is the construction of one diagnostic system for each component. Each diagnostic system is denoted an agent. For example do AgentX handle componentX. The objective for the agent is to decide if the associated component is working or not. The output from an agent is *true* if the component is considered working, and *false* if it is considered to be broken. If the agent has not been able to decide whether the component is broken or not, output is *unknown*.

A problem is that in order to decide if a component C_x is working; some facts about the surrounding components are needed. If C_x uses output from another component C_y it is of importance to know that C_y is working. In this case the Agent for C_x can call the agent for C_y to get the status of C_y. Figure 6-1 shows how the rules decide the outcome of AgentX depending on the outcome of AgentY and Test1.



Rule nr	AgentY	Test1	AgentX
1	True	True	False
2	True	False	False
3	False	True	True
4	False	False	False

Figure 6-1 Decision table of AgentX based on AgentY and Test1

Another example is that Figure 6-1 shows how AgentX decides if C_x is broken by calling Test1. Test1 uses sensor data from C_y and therefore AgentY has to be called to verify that C_y is working. One possibility would be that Test1 calls AgentY instead but this would lead to unmanageable cyclic calls and are therefore not allowed.

6.1.1 Cyclic Calls

With a cyclic call is meant that a function calls another function which in turn calls back to the first function. This can be done directly or indirectly. When that is done indirectly there can be several functions that constitute the cycle and a direct cyclic call is between only two functions. The direct cyclic call is illustrated in Figure 6-2, where Agent1 calls Agent2 that in turn calls back to Agent1 and thus making an undesired cycle. Figure 6-3 shows an indirect cyclic call where four agents call each other in a manor that forms a cycle. In both the figures an arrow indicates a direct call.

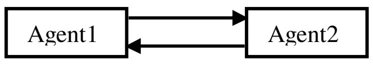


Figure 6-2 Direct cyclic call

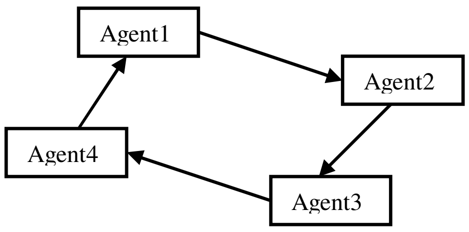


Figure 6-3 Indirect cyclic call

The direct cyclic calls are for obvious reasons easy to discover and avoid. The indirect cyclic calls can however cause a problem. It is to avoid these calls that the hierarchy in method 1 exists. To avoid cyclic calls there is a rule who says that calls can only be made to functions that are located in a lower level in the hierarchy than the caller. Despite this, problems can occur in large applications where it can be difficult to know where in the hierarchy functions are. Sometimes it also demands a certain amount of redundancy to avoid the cyclic calls. If for example an agent needs to call another agent at the same level it would not be allowed to do this and the first agent would instead have to call the second agents tests directly. This would accomplish the task as a call to the second agent, but with some redundancy necessary. The redundancy that becomes necessary is that all handling of the results from the tests that are done in the second agent also has to be done in the first. This example is shown in Figure 6-4. The figure contains two agents and a set of tests that are being called by the agents. The two complete arrows indicate the allowed calls and the dotted arrow represents the illicit call that can not be made.

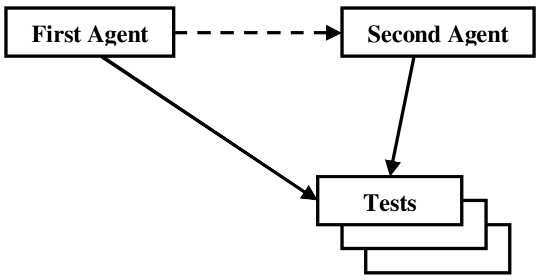


Figure 6-4 Redundancy in the agents

6.1.2 The Process of Method 1

Below follows a description of the process for method 1. This is illustrated in Figure 6-5 where RUF data is input to the process and a list of components is output. The illustration is divided into three layers where layer 3 is the deepest with all the different tests. Layer 2 contains all the agents and layer 1 is the comprehensive process that controls the underlying layers.

In the process picture it is shown that layer 1 consist of a function called *FDI()* and it is this one who control which agents that are called. *FDI()* also handles the answers from the called agents and uses these answers to come to conclusions that are needed for a good fault isolation.

All agents exist in layer 2 and they are sorted into a hierarchy. The hierarchy is divided according to which component an agent is connected to and how the components relate to each other. Some components can contain other components which in turn can consist of more components, as is explained in section 2.1. The hierarchy is important in order to avoid cyclic calls that otherwise would be a problem. To avoid cyclic calls there is a rule that no agent is allowed to call another agent what is on the same level in the hierarchy as the caller. Nor is an agent allowed to call other agents at a higher level in the hierarchy. It is only allowed to call functions downwards in the hierarchy. For an agent to be able to decide if its component is faulty it has to call all agents connected to its sub components. A reason for agents to call each other is if an agent needs to know if the component connected to another agent is faulty or not in order to self be able to decide if it is faulty.

The agents' task is to decide if its component is faulty or not and there are different tests that they use to accomplish this. These tests are all located in layer 3. When the agents have received answers from the tests, they will make a decision based on these answers and some rules.

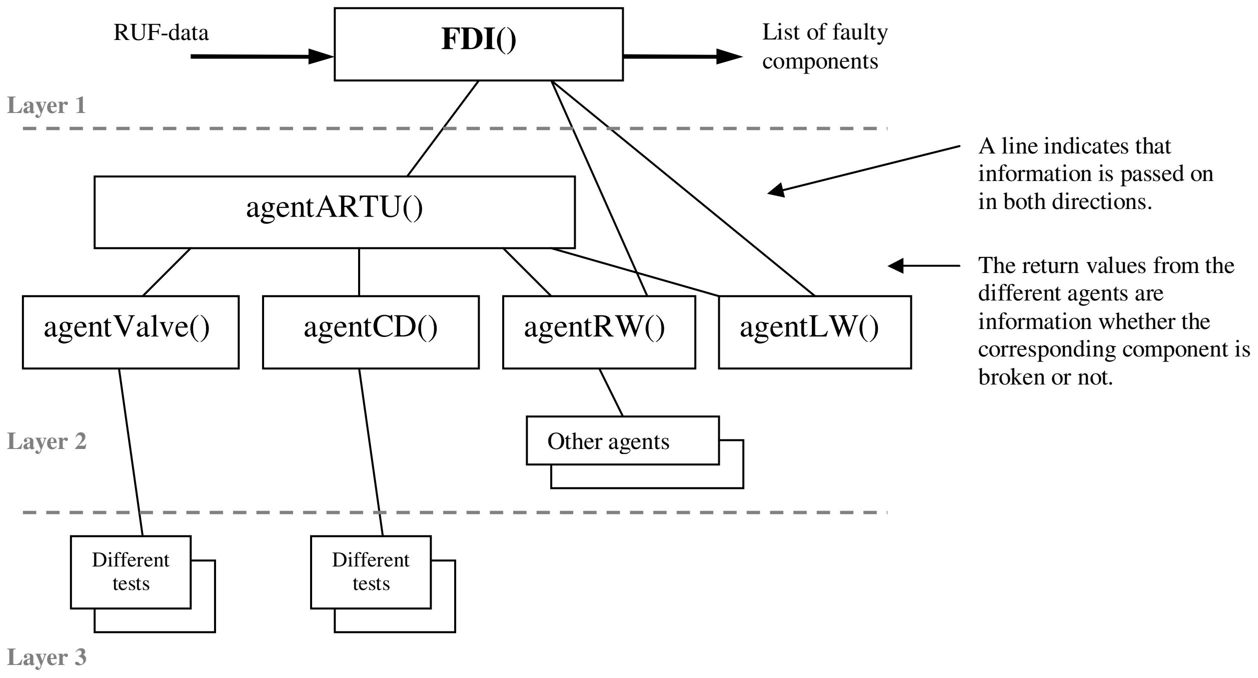

RUF-data
FDI()
List of faulty components
Layer 1
A line indicates that information is passed on in both directions.
agentARTU()
The return values from the different agents are information whether the corresponding component is broken or not.
agentValve()
agentCD()
agentRW()
agentLW()
Other agents
Layer 2
Different tests
Different tests
Layer 3

Figure 6-5 The process of method 1

The use of agents has a drawback when there are insufficient sensors. For an agent to be able to decide if the component is broken it must have sensors in the nearby. If a couple of components are placed between these sensors it is not always possible to say which component that is broken when a fault is detected. Figure 6-6 shows two components placed between two sensors, and if a fault is detected between the sensors it is not always possible to isolate the fault to one component.

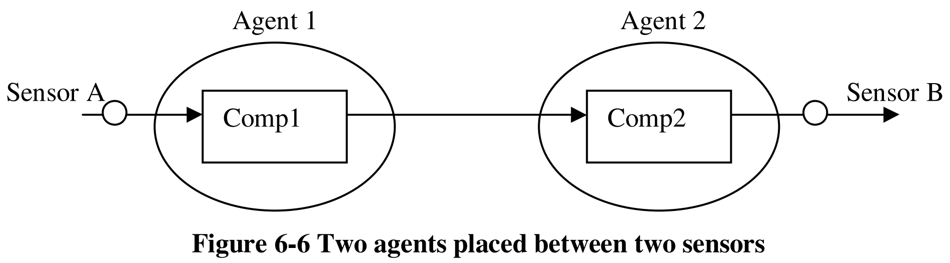


Figure 6-6 Two agents placed between two sensors

A possible solution is to see *Comp1* and *Comp2* as one unit. Figure 6-7 shows a large agent covering both *Comp1* and *Comp2*. When *Comp1* or *Comp2* is listed for examination Agent1.2 is called instead of Agent1 or Agent2. If Agent1.2 outputs *True* either *Comp1* or *Comp2* is broken, and it is time for statistics or other suitable method to decide whether Comp1 or Comp2 shall be replaced first.

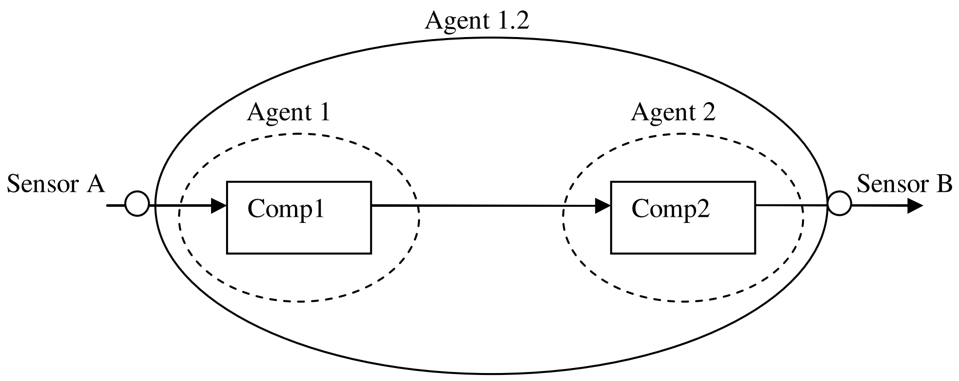


Figure 6-7 One agent placed between two sensors

6.1.3 Ranking of Components

The ability of ranking components after probability of failure given test results is one of the demands specified in section 3.2. If the agents indicate more than one component as possibly faulty, there is a need to rank these in good way. There is a variety of different information that can be considered for this ranking. Aspects that can be worth considering are statistics over earlier faults, mean time between failures, cost, time to change component and so on. These aspects only use information of how components usually failure. Doing like this every component gets a value and the component that has been considered broken is put on top of the list. A better ranking system would be to also look at how the tests have reacted.

This rank can be done since RUF-data that agents and tests work with is denoted with timestamps for every sample. The agents are able to specify at what time the component is considered broken. Figure 6-8 shows three agents claiming that their component is broken. A possible rank is to say that the one indicated first is most probably broken, and that this broken component disturbs the other agents to believe that their component is broken.

The demand of a list of components ranked by probability of failure is hard to fulfill. In the example above the three agents have answered true at different times. Is it really sure that the earliest found is broken? How sure is it and when is some other case more probable? Since no general procedure is available to build in knowledge about this, the demand is not achieved and instead it is the technician's job to rank the components.

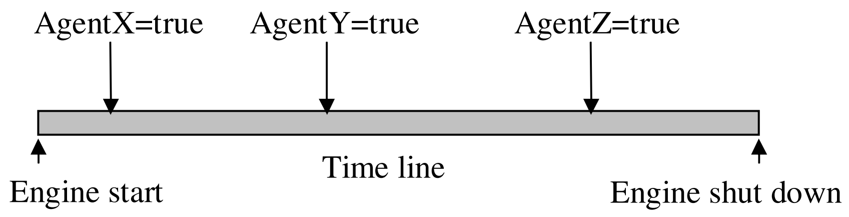


Figure 6-8 Agent answers in time

6.1.4 Advantages with Method 1

- It is easy to automatize the manual fault isolation procedure and do the same tests as a technician does.

6.1.5 Disadvantages with Method 1

- As will be presented under next headline, all test results has to be considered in order to do a correct isolation. This is not an impossible thing for agents but it gets rather inefficient since every agent has to contain the rules for all the other agents in order to determine if some other agent better explains the test results. This implementation ends up with something similar to ESH but in every agent, which will be explained in next chapter.

6.2 Method 2: Extended Structured Hypothesis Tests

The problem with agents is that more than one agent can answer *True* based on the same tests. This problem is avoided by using *Structured Hypothesis Tests*, described in section 4.5. Structured Hypothesis Tests evaluates all tests and tries to find one component, or a set of components, that could cause the test results. Practically it tries to find a component that has an 'X' marked for all tests that has reacted. If no component has 'X' marked for all test, there may be more than one faulty component i.e. some test have reacted because of one component and some test have reacted because of another, or there may be a test that has reacted wrong, and there are false alarms in the test results. The handling of multiple faults is done later and for now more focus is put on handling false alarms. To find a false alarm it is possible to search for a component that could cause all test results except one test. If a component can explain 3 of 4 reacted tests and no component can explain all 4, then the one explaining 3 are considered most probably broken. By doing this, more than one component may be able to explain 3 of 4 tests, but they explain different tests. Figure 6-9 shows the decision table of three components and four tests that have reacted. All three components can explain 3 of 4 tests. The question is how to pick the one most probably broken, out of these three.

Dependency	*ComponentX*	*ComponentY*	*ComponentZ*
Test1	0	X	X
Test2	X	0	X
Test3	X	X	0
Test4	X	X	X

Figure 6-9 Decision table of three random components

A solution is to look at the tests that could not be explained by the components, to see if any of these tests often react when no dependent component is broken or if any test almost never reacts this way. If for example Test1 often react without a broken dependent component, and Test2 and Test3 never do, it is probably ComponentX that is broken since it explains all tests except Test1, and Test1 is not trustable. To handle this new information we have extended the structured hypothesis tests with an extra matrix and decided to call the method for *Extended Structured Hypothesis tests*, abbreviated ESH. The ESH-matrix is an extra matrix specifying values for missed and false alarms. This extra matrix is a complement to the ordinary decision table shown above. For tests marked with '0' in the decision table the corresponding value in the ESH-matrix specifies the probability of false alarms when the component is working.

ESH	*ComponentX*	*ComponentY*	*ComponentZ*
Test1	0.9	0.3	0.1
Test2	0.2	0.1	0.23
Test3	0.35	0.4	0.1
Test4	0.4	0.3	0.1

Figure 6-10. ESH-matrix with values for missed and false alarms

False alarms are one part of uncertainty mentioned in section 5.1.2. *Missed alarms* are the other part. Figure 6-9 describes the decision table of reacted test, but it is still interesting to look at tests that have not reacted. Figure 6-11 shows the decision table for Test5 that did not react. If Test5 is strongly connected to ComponentX and always reacts when ComponentX is broken, it is not likely that ComponentX is broken if Test5 has not reacted. This information can be handled by Structured Hypothesis Tests by putting '1' in the cell corresponding to the component and the test. A '1' in a cell means that if the test has not reacted the component can not be broken. This is a very hard statement and it is not applicable especially often.

Dependency	*ComponentX*	*ComponentY*	*ComponentZ*
Test5	X	0	0

Figure 6-11 Continuation of Decision table in figure 6.9

For tests marked with an 'X' in the decision table the corresponding value in the ESH-matrix specifies the probability of missed alarms when the component is broken. This way the ESH-matrix handles the information about how probable false and missed alarms are. How to use this information will now be explained.

6.2.1 The Process of Method 2

Below follows a description of the process of method 2. This process is illustrated in Figure 6-12 and consists of four major steps. The parameters that are sent between each step are shown in connection with the arrows. The different steps are described in more detail.

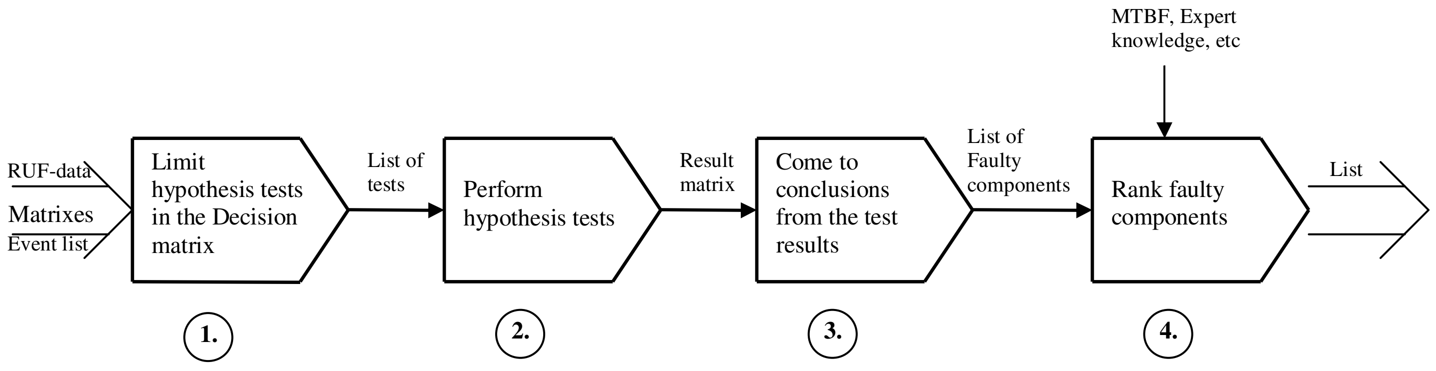

MTBF, Expert knowledge, etc
RUF-data
Matrixes
Event list
Limit hypothesis tests in the Decision matrix
List of tests
Perform hypothesis tests
Result matrix
Come to conclusions from the test results
List of Faulty components
Rank faulty components
List
1.
2.
3.
4.

Figure 6-12 The process of method 2

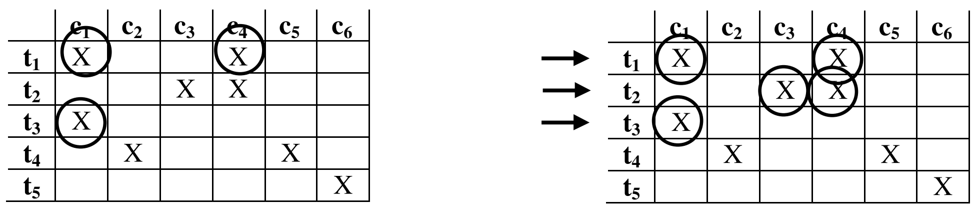
Step 1

In this part of the process the amount of hypothesis tests needed to be performed are limited. This is to not burden the system unnecessarily much and also to shrink the time it takes to perform a fault isolation. If time is not a critical aspect or if the tests are not too resource demanding, there is no need for this limitation.

The limitation is done by checking which hypothesis tests that provides any information to the diagnosis of the components in the list. Then only those tests are performed. Tests that provide information are first and foremost those that are directly affected by the components in the list, but also those that are connected to components that affect tests that in other ways contribute to the diagnosis. An example, pictured in

Figure 6-13, follows to clarify the limitation procedure. The figure shows the same decision matrix in two different steps in the limitation procedure. The arrows indicate which tests that in the end has to be performed.

Say that component c_1 is the only one in the list of possibly faulty components. First and foremost every test that affect c_1 must be performed (t_1 and t_3). Then a check is made for any further components that affect the so far chosen tests (the only new component is c_4, which comes from t_1). All tests that are affected by the new component are also added to the list of tests that has to be performed (test t_2 are affected by c_4). These additional tests contain information that can be used to dismiss components as faulty. So far the tests that have to be performed are t_1, t_2 and t_3. The latest added test (t_2) is affected by c_3 and c_4. Component c_3 are new and tests that affect that one must also be added to the list of tests that has to be performed. In this way the procedure continues until no new tests are found. In this example, c_3 does not result in any new tests and the procedure is finished.



	c_1	c_2	c_3	c_4	c_5	c_6
t_1	Ⓧ			Ⓧ		
t_2			X	X		
t_3	Ⓧ					
t_4		X			X	
t_5						X

		c_1	c_2	c_3	c_4	c_5	c_6
→	t_1	Ⓧ			Ⓧ		
→	t_2			Ⓧ	Ⓧ		
→	t_3	Ⓧ					
	t_4		X			X	
	t_5						X

Figure 6-13 An example of how the tests are limited

Step 2

In step 2 the chosen hypothesis tests are performed and information about when and which tests that reacted are sent to step 3 in the isolation process. This step also includes some sort of handling of the time aspect. A detailed description about the time aspect can be found in section 6.2.3. The handling of the time aspect is necessary so that the next step in the process can make an easy and flawless isolation.

Step 3

In this step of the process, conclusions are made with help from the results from the hypothesis tests. The result of this step is a ranked list of components with a corresponding score that states how likely it is that a component is faulty. The list of components can contain additional information, like for example number of false alarms and which they are. The number of false alarms for each component can be calculated by comparing which hypothesis tests that has reacted to which components they are affected by. If a test has reacted that is not affected by a component, that component has a false alarm. What is meant here is that if it is this component that is faulty there has been a false alarm. If there is a component that affects every test that has reacted, this component has no false alarms. In this way the number of false alarms for each component can be calculated. Missed alarms can also be calculated in a similar way. If a component affects a test that has not reacted, that component has a missed alarm. For every test that has not reacted and affects a component, that component has a missed alarm.

The information about missed and false alarms is then used to rank the components in order of most probable faulty. How the ranking of components is done is presented in section 6.2.2.

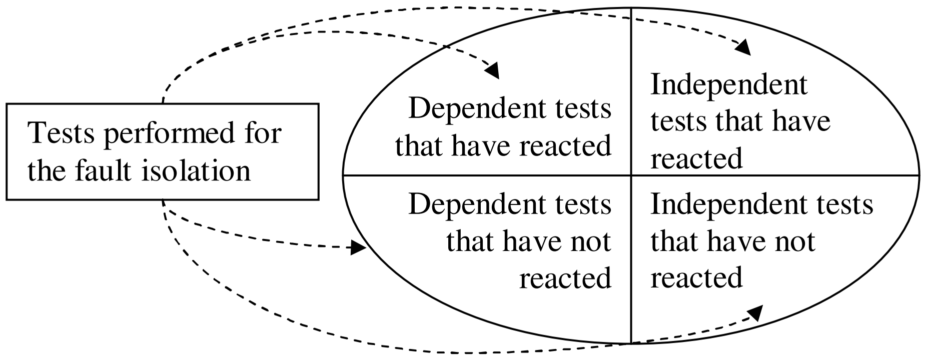
Step 4

Since it can be desirable to have more information than just the score from step 3 in mind when the components are ranked, step 4 exists to take care of this. Further information that can be used for this part of the ranking is for example MTBF, expert knowledge, statistics about earlier maintenance and so on. Here different weights are added to different information and everything is weighed together to sort the list of components in the order of which to change or inspect first. This step in the process is not implemented in our application, but is still included here to show a probable continuation on the treatment of the data returned from step 3. The reason step 4 is not implemented is that it has no direct connection to the fault isolation itself or to the method that is used in this thesis. The fault isolation process has already generated a list of components with belonging scores and if one chooses to trust it or not is a different issue. Naturally it can be in Saabs interest to include other aspects when they decide which component in the aircraft that should be changed, but this is outside the scope of the fault isolation process.

6.2.2 Ranking of Components

A list of components to change has to be produced and a score shall belong to each component. The component with the highest score is the one to change first and shall be put on top of the list. Different ways of giving the components its score are available, here are two ways mentioned and one of them is used in Method 2. Both of them use the test results, the dependency matrix mentioned in section 4.5 and the new ESH-matrix.

How all tests are split up into four subsets is shown in Figure 6-14. This is done for each component. If a test is dependent on the component it is put in the left half, otherwise it is put in the right. If the test has reacted it is put in the upper half, otherwise the lower.



Tests performed for the fault isolation
Dependent tests that have reacted
Independent tests that have reacted
Dependent tests that have not reacted
Independent tests that have not reacted

Figure 6-14 Set of tests divided into four subsets

Rewarding scoring

The first of the two scoring system is a rewarding system. It starts with the initial belief that every component is working, and the value for each component is initially set to zero. When a test indicates that a component may be is broken the value for that component is increased. The fault isolation is only used when a fault has been detected and the assumption of present faults in the system can be made. When a test has reacted it is an indication that one of the components depending on the test is broken. Therefore the values for all depending components are increased. When a test not has reacted it is an indication that one component that is not depending on the test is broken, and the values are therefore increased for all components that are not depending on the test. Figure 6-15 shows the subsets of tests used in rewarding scoring.

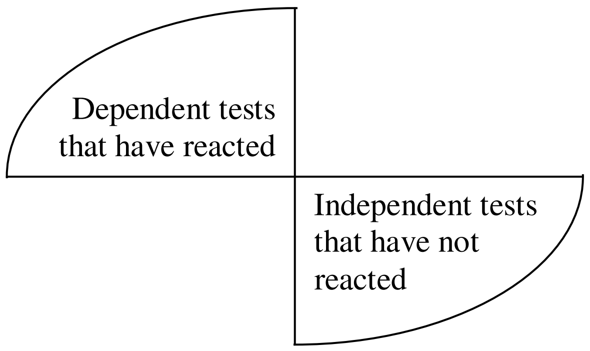

Dependent tests that have reacted
Independent tests that have not reacted

Figure 6-15 Subset used in rewarding score

Punishing scoring
The second of the two scoring system is a punishing scoring. This is the one we have chosen to use. It starts with the initial belief that every component is broken, and the value for each component is initially set to one. This is in the range of one to zero. When a test indicates that a component may be working the value for that component is decreased. The assumption is done that there are present faults in the system. When a test has reacted it is an indication that a dependent component is broken and therefore it is also an indication that an independent component is working and the value is decreased for all independent components. If a component is broken its dependent tests would react, therefore it is an indication that a component is working if its dependent tests have not reacted. Consecutively the value for all dependent components is decreased. Figure 6-16 shows the subsets of tests used in punishing scoring.

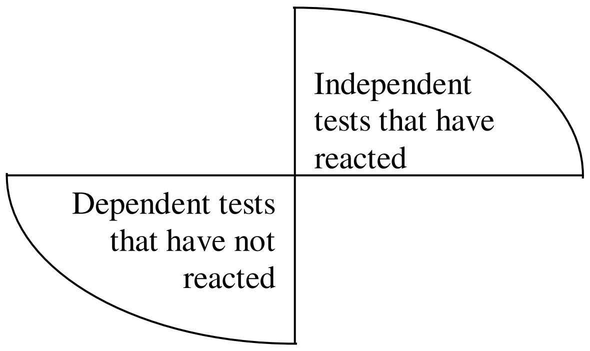

Independent tests that have reacted
Dependent tests that have not reacted

Figure 6-16 Subsets used in punishing score

Since it is known from Chapter 5 that false and missed alarms are of interest is the punishing scoring is suitable. Missed alarms are strongly connected to *dependent tests that have not reacted*, and false alarms are strongly connected to *independent tests that have reacted.*

To get a scoring system that grade the component from 0 to 100 percent, i.e. from zero to one, we have choose to multiply the values belonging to the tests in the two subsets. All values is within the interval [0, 1]. This gives

$$Value(C_i \mid Test\ results) = 1 \cdot \prod_{j \subset \{missed\ alarms\}} Test_j \cdot \prod_{k \subset \{false\ alarms\}} Test_k \tag{1}$$

the '1' after the equal sign is the initial value.
Figure 6-17 shows the same subsets as Figure 6-14 but rewritten with a new notation.

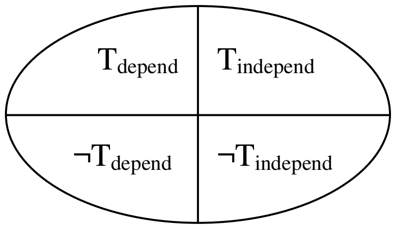


Figure 6-17 New notation of subsets

Every part is defined as:

T_{depend}	Set of reacted tests depending on C_i.
$\neg T_{depend}$	Set of *not* reacted tests depending on C_i.
$T_{independ}$	Set of reacted tests *not* depending on C_i.
$\neg T_{independ}$	Set of *not* reacted tests *not* depending on C_i.

This new notation is used in (1):

$$Value(Ci \mid Test\ results) = 1 \cdot \prod P(C_i \mid \neg T_{depend}) \cdot \prod P(C_i \mid T_{independ}) \tag{2}$$

where $P(C_i|\neg T_{depend})$ and $P(C_i|T_{independ})$ will be derived. To do this and to make the fault isolation system working we need knowledge from the world it is going to work in. The knowledge has to be specified by experts on the aircraft, experts who created the tests, and statistics from earlier repairs and flights. To keep it simple they only need to specify one sort of probability and it is the probability that a certain test reacts when a certain component is broken:

$$P(T_j \mid C_i) \quad \forall i, j \tag{3}$$

$P(T_j|C_i)$ is the value that shall be stored in the ESH-matrix.
Bayes rule used on (3) gives the following two equations:

$$P(C_i \mid T_j) = \frac{P(T_j \mid C_i)P(C_i)}{P(T_j)} \tag{4}$$

$$P(C_i \mid \neg T_j) = \frac{P(\neg T_j \mid C_i)P(C_i)}{P(\neg T_j)} = \frac{(1 - P(T_j \mid C_i))P(C_i)}{1 - P(T_j)} \tag{5}$$

Where only $P(C_i)$ and $P(T_j)$ are unknown. $P(C_i)$ is the probability that a component is broken and the value can be calculated by statistics. We have chosen to use the value α (alfa) for every component to not favorise or punish any component. This means that it is equally possible that component-x breaks as if component-y breaks. If this choice is proven to be bad it can easily be changed later on in the development. To get $P(T_j)$ the following discussion is used: Assume that tests are rather well designed and that false and missed alarms are unusual exceptions. If so it is fair to say that the probability of a test is the same as the probabilities that the components that influence the test break:

$$P(T)=1-\prod_i (1-P(C_i))=1-(1-P(C_1))(1-P(C_2))(1-P(C_3))\ldots= \\ =1-1+P(C_1)+P(C_2)+\ldots+P(C_1)P(C_2)+\ldots+P(C_1)P(C_2)P(C_3)+\ldots \tag{6}$$

Since the components are chosen to have a low probability of failure P(C_i) are assumed to be very small. This assumption gives:

$$1-1+P(C_1)+P(C_2)+\ldots+P(C_1)P(C_2)+\ldots+P(C_1)P(C_2)P(C_3)+\ldots\approx \\ \approx\sum_i^n P(C_i)\approx n\alpha \tag{7}$$

because all products are neglected.

To summarize (1) is written as

$$Value(C_i \mid Test\ results)=\prod_j \frac{(1-P(T_j \mid C_i))P(C_i)}{n\alpha}\cdot\prod_k \frac{P(T_k \mid C_i)P(C_i)}{1-n\alpha} \tag{8}$$

An analysis shows that if the number of factors in any of the two products is increased, the total value of Ci is decreased. This is desirable since more terms come from more missed or false alarms.

6.2.3 Consequences of Similar Tests

If two tests are very similar, uses the same sensors and always react the same, the same probability is multiplied several times. If the tests reacts as false or missed alarms they will be multiplied together with the other false or missed alarms and a square term of the two tests are received. A solution is to change the tests probability depending on the dependency and the value moves closer to $\sqrt{probability}$ than *probability*. When several tests are dependent the probability value can be distributed among them so the product will be the original value used if the tests were grouped together as one single test.

6.2.4 Time Aspects

All tests are performed within a given time interval. If a test react during the interval the answer is *true*, if it does not the answer is *false*. Whether or not the test would react outside the interval is not of interest. If the data used in the test are not available for some reason, i.e. this kind of data are only recorded during some conditions, the answer is *Unknown.* An interesting problem is to choose what time interval to use. Shall the entire flight be used, just a couple of minutes surrounding the time when a fault was detected by the aircraft or maybe the ten latest flights for the aircraft. If a too short interval is used the possibility of missed alarms is increased. If a too long interval is used the possibility of false alarms is increased. The latter depends on that during a long interval several different flight conditions do occur and within each one can different tests react without any faults in the system. The time aspect is therefore another possible value of adjustment that has to be adjusted when the full scale implementation is done. It should be mentioned that if short intervals are used another problem appears. At what time shall we start and stop and shall we look at several intervals? Figure 6–12 show the time for one flight split into four intervals. Intervals like these are often called *time windows.* A fault has been detected by the aircraft in the second window.

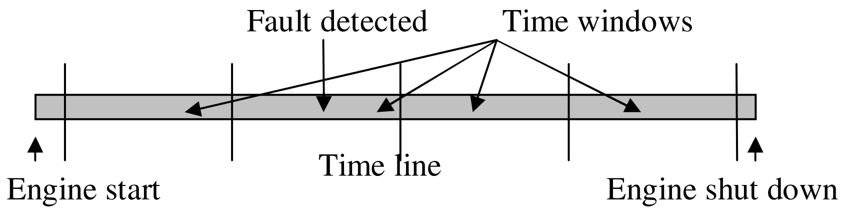


Figure 6-18 Problems with time windows

There are no guarantees that the fault actually did happen during the second window. It could be the case that it just was not able to be detected earlier. In this case several windows have to be taken in account and every window can produce different diagnosis based on the reacted tests, and like the agents it is now up to the technician to decide which component to replace. We choose to recommend one time window starting when the engine starts and ending when the engine is shut down. This is to simplify the fault isolation and to avoid getting different diagnosis's for different part of a flight.

6.2.5 Advantages with Method 2

- Since all information is gathered in one place, it is practical to make the decisions in one place.
- It is possible to calculate and prove which components those are able to be isolated and with this information add extra tests to isolate more components. The adding of tests can be done both in the aircraft and in the post-process.

6.2.6 Disadvantages with Method 2

- Structured Hypothesis tests are actually not made for components with several failure modes (when a component can break down in different ways). The presence of different failure modes force tests for the different failure modes. When a component is in one failure mode, tests for the other modes will not react and appear as missed alarms. This will lead to more detailed decision matrices with the components divided into failure modes, but the theory should still be working.

6.3 Similarities to Strict Logic and Uncertain Reasoning

It is now time to connect chapter 3 and 4 by explaining the similarities between Agents and strict logic as well as between ESH and uncertain reasoning.

Regardless of the inner design of agents they act as logic units. They get the input from surrounding agents and tests in form of true/false/unknown. Figure 6-19 shows a rule-based logic used to decide the agents output depending on the inputs. Rule nr 1 specifies that if Test1=True and if Test2=True, the agents output is False. If an agent wrongfully accuses its component to be broken, i.e. wrong rule is used, no other agent will discover this mistake and instead it will be handled as a fact that the component is broken.

Rule nr	Test1	Test2	Agent output
1	True	True	False
2	True	False	True
3	False	True	True
4	False	False	False

Figure 6-19 Rule-based logic for agent

Extended Structural Hypothesis Tests has a matrix with values for all possible combinations of tests and components with specified values of how probable false and missed alarms are. Method 2 handles all this information and weighs presence of false and missed alarms against each other when deciding which component to repair. The uncertainty is therefore fully handled.

Chapter 7

Method 1 vs. Method 2

To evaluate the two approaches an examination of how they fulfill the demands specified in section 3.2 is described below. When one method is considered better than the other, the better one is described first to easily understand the drawback of the other. M1 is short for Method 1, and M2 is short for Method 2.

7.1 Demands on the Approaches

7.1.1 Deterministic Fault Isolation

Both methods can be handled in editions and fulfill this demand. In Method 1 an edition will consist of new and updated agents that will be added to the program. An new edition for Method 2 consists of new hypothesis tests and new updated matrices. For information about which matrices that is used in Method 2, see section 8.3. Method 2 has the ability to be trained in field if the demand will be changed.

7.1.2 Usable for a Less Experienced Technician

Method 2: The system does all decisions by it self as long as the time interval is adjusted correctly. It needs no extra interpretation of a technician. The component ranked as number one is the one to change.

Method 1: A technician is needed to decide if the first detected component is the one most proper to repair. This means that the technician must be more experienced to use M1 than M2, and that M1 does not fulfill this requirement.

Which one of the methods has the best ability to convince the technician that the fault isolation isolate the correct component?
Method 2: It is possible to show which tests that have reacted and to show which can be explained by different components. It is also possible to show that a component is not broken since some tests did not react. To show how many of the reacted tests that every component can explain is a good measure of how accurate the diagnoses are.

Method 1: Every agent that has reacted explains the reason for this by itself. This gives good information of each component but very poor information of the dependency between components. It is hard to choose between two components, especially if the agents show results from the same tests. It is even hard to show that a component is considered working with the only evidence that the agent did not deliver a *true*.

7.1.3 Application vs. Information

Method 2: The information is kept in a matrix of crosses and a matrix of probabilities. If a component is upgraded to a later version some crosses and some probabilities may have to be changed. All information can be kept in Microsoft Excel sheets and loaded when the application starts.

Method 1: The information is kept in the agents. Each agent has information about its component. Since the agents are just rule-based logic, these rules can be kept in a separate file and loaded to the agents upon start. The upgrade of a component leads to new rules in this file and would be rather straight forward. In a later development smart agents with AI can be used and the information is built into each agent and is no longer separated from the application.

7.1.4 Configuration Management

Method 2: The components in the present examined aircraft control which tests to run. In the beginning the program has access to all tests but sorts out the tests not applicable for this aircraft.

Method 1: The components in the present examined aircraft control which agents to load. This results in some problems in the logic rules. We may need some special rules for special sets of components. This is easy as long as only one component in each rule is upgraded but otherwise this could lead to a combinatorial explosion of rules.

7.1.5 Maintenance

Method 2: Maintenance is to create new tests, update the matrix of crosses when realized that some additional components are dependent of some test, and update the matrix of probabilities, the probability table, to tune the accuracy of the system.
The creation of a new test requires knowledge about the aircraft and RUF but not on the program. The insertion of the new test into the system is done by a beautiful solution explained in Chapter 8 that makes it possible to insert tests without writing one single line of code. In fact, as long as the technician can make new tests he can also add them to the fault isolation.

Method 1: Maintenance is to create new tests, update the rules for the agents, and to create new agents. The first two is rather simple but to insert a new agent a lot of knowledge both about aircraft and the FDI is required. A new agent has to be called by other agents and these agents' rules must be updated to use the information from the new agent.

7.1.6 Expansion

Method 2: To gather tests for all subsystems in the aircraft to a large ESH is possible. It is also possible to split up the tests in an ESH for each subsystem, and for each subsystem add a dummy component called *fault in subsystem-X*. If *fault in subsystem-X* explains most test results the ESH for subsystem-X is used instead.

Method 1: Building agents for every component is possible. It is also possible that they call agents in other subsystems. Similar to ESH, *agent-Subsystem-Y* can be called by all agents in subsystem-X if they suspect a failure in subsystem-Y.

7.1.7 Multiple Faults Isolation

Method 2: ESH can handle multiple faults but it is not easy to do it in an efficient way. For each component a check against the other components is done. If tests have reacted that for component c_i is $T_{independ}$, then a component c_j that has these tests in T_{depend} can explain these test results. If now c_j has some reacted test is $T_{independ}$ that is in T_{depend} for c_i it is fair to suspect that both components are broken since the set of $T_{independ}$ becomes smaller. The complexity is $O(N^2)$ since there are two nested loops through the components, where N is the number of components. When these two components have been found, a third loop can be done to find triple faults with complexity $O(N^2)+ O(N)\approx O(N^2)$.

Method 1: The agents react independently so several agents can explain its component to be broken. This means that the system can handle multiple faults. The problem is that the system can not tell whether there exist single or double faults so the technician will probably always believe that only a single fault is present and that some agents have reacted incorrect.

7.1.8 Ranking of Components

This demand has already been handled by 6.1.3 and 0 and will therefore not be repeated.

7.2 Conclusion from Chapter 3-7

The agent structure is a reasonable first thought. It resembles the way technicians manually do fault isolation. Unfortunately this structure has several drawbacks in many of the demands. Several agents can react upon the same signals and even present the same facts when they explain their results. No good way has been found to build in all knowledge in the system so the technician does not need to do half of the isolation manually. The advantage with agents is that event flow can be followed: *First the tank pressure regulator broke, secondly the valve unit, and third the transfer pump stopped.* The technician has to realize that this event flow depends on that if the tank pressure regulator breaks, the valve unit can work properly and will not conduct the fuel the right way and if no fuel arrives to the transfer pump it stops. In some case the agents can contain this knowledge but the links are not always as simple as this.

Method 2 solves all problems. The tests for the valve unit and the transfer pump know that they can react if the tank pressure regulator is broken. The test for the regulator can not react when the valve unit or the pump is broken so the only valid diagnosis that can explain the test results is the regulator, and the other “faults” will not even be proposed.

Method 2 solves the problem of what to display for the technician. It will be the list of components, the probability of each component and a number of false alarms that has to be present for the component to be a diagnosis. We can present the tests that have and have not reacted for a component and we can also explain why a component is not broken.

Finally possibility to detect multiple faults in the ESH and the elegance of introducing new tests the must be repeated. These are two major advantages with ESH that makes the fault isolation much more accurate than even the best technician with unlimited time can produce.

Chapter 8

Implementation

This chapter contains a closer description about how method 2 was implemented. This method did not have the drawbacks that method 1 had and that is the reason we chose to implement method 2. Even though they are two different methods, they would still have pretty much in common in the implementation. The common parts are those at a lower level in the implementation hierarchy, i.e. the specific tests that are performed on the RUF data. The hypothesis tests used in method 2 are described in more detail in section 8.2.

8.1 Implementation of the Framework

This section contains a description of how method 2 was implemented and each step in the process is described in more detail. In Figure 8-1, the main functions in the process are illustrated and also the parameters sent between the functions are shown in form of arrows. The dotted arrow in the beginning of the process indicates that there are no variables sent between the first two main functions. But it is still there to represent the flow in the process. Each main function calls some important subfunctions and these are shown under respective main function. How the functions relate to the different steps in the process is also shown in the figure. For more information about the variables used in the implementation see section 8.3.

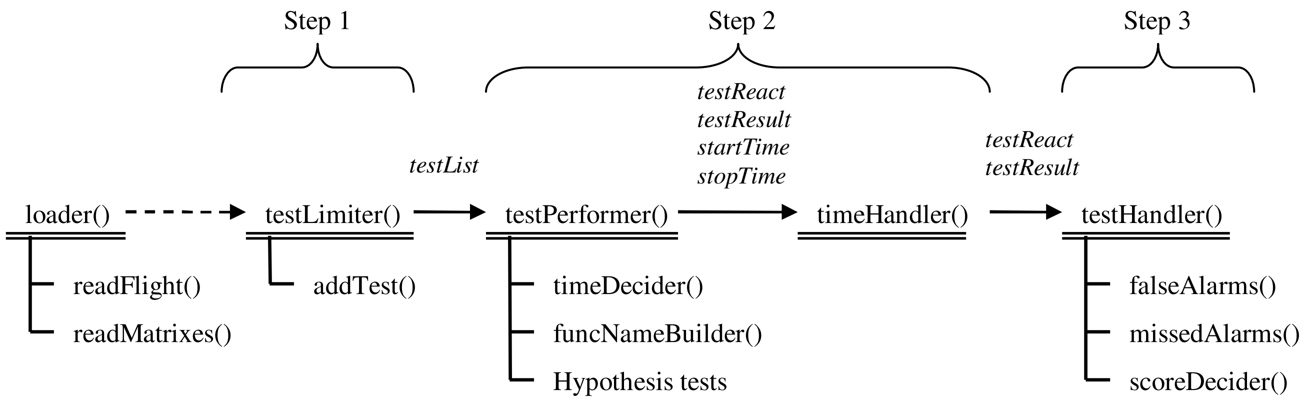


Figure 8-1 Implementation of the framework

The RUF data and the Event list are global parameters and are available in all the steps in the process. However they are only used in step 1 and 2, which will later be explained in the process pictures of the respective steps.

Just like the RUF data and the Event list, the four different matrices (*SHmatrix*, *SH2matrix*, *function2NameMatrix* and *alarm2CompMatrix*) are read into the process before step 1 and are all global variables. The four matrices are explained in detail in section 8.3. Reading of all the global variables are handled by the function *loader()*, which in turn uses the two functions named *readFlight()* and *readMatrixes()* to do the job. *readFlight()* reads the RUF data and the Event list, while the matrices are read by the other function.

8.1.1 Step 1

In this step the alarms that have been triggered are extracted from the Event list, which in turn are used to make a list of possibly faulty components. This list is in the application called *compList*. The mapping between alarms and components are specified in *alarm2CompMatrix*. This is a matrix that simply holds information about which components that can be faulty when specific alarms are raised. A limitation of the amount of tests that needs to be performed is made with help from the variable *compList*. The function *addTest()* is used to add hypothesis tests to the list of test that shall be performed. This is done in the way described in section 6.2.1.

In Figure 8-2 a more detailed picture of step 1 in the isolation process is presented. The dotted arrow has the same meaning as in Figure 8-1. The list of tests that shall be performed are in this picture called *testList* and are sent from *testLimiter()* to *testPerformer()*.

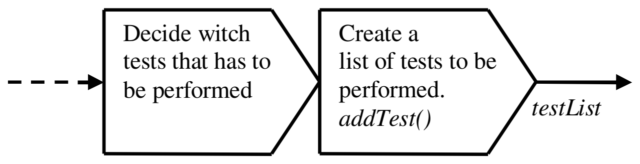
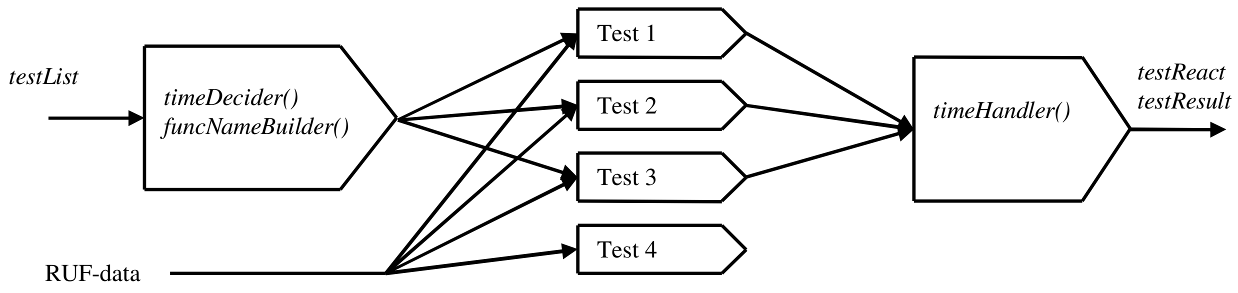

Decide witch tests that has to be performed
Create a list of tests to be performed.
addTest()
testList

Figure 8-2 A detailed picture of step 1 in the implementation

8.1.2 Step 2

In Figure 8-3 below, there is a more detailed illustration of step 2 where test 1, 2 and 3 has been performed. Therefore there are only arrows from the first function to these



three tests. There are two major parts in step 2. The first one is a comprehensive function that controls which hypothesis tests to call on the basis of the *testList*. The second one is a function that handles the time aspect of the results from the tests. The first part includes two functions, *timeDecider()* and *funcNameBuilder()*. *timeDecider()* determines a start and a stop time for every test that is to be performed and for more information about how this is done see the time variables in section 8.3. The other function builds the function names for the tests and evaluates them in order to execute the tests. The hypothesis tests have among other things RUF data and a time interval as input. The output from a test is a *timeArray*, which indicates if the test has reacted or not. This variable is further described in section 8.3. After all the tests have been performed, the results are put together in a variable of type *testResult*. This variable is first treated with regard to the time aspect in the function timeHandler(), then it is sent on to *testHandler()*. For more details on how the time aspect is handled see section 8.4.

Figure 8-3 A detailed picture of step 2 in the implementation

8.1.3 Step 3.

In the first part of step 3 the number of false alarms and missed alarms for each component is calculated. Not just only the number of missed and false alarms are calculated, but also which alarms it is. This information is stored in a temporary variable and later used to calculate a score for each component.

The variable *reactList* is used when the false alarms is determined. This is done in the following way. The content of each row in *reactList* is compared to the *testReact* list. The difference between the row and the list is the false alarms for the corresponding component. This is true because the row contains every test that has reacted and also is affected by that component, and the list contains every test that has reacted. The difference between these two sets is the false alarms. This is the set of independent alarms that has reacted, as mentioned in section 6.2.2.

reactList is also used when the missed alarms are determined. This is done by again comparing a row in *reactList* with the set of all depending tests for a component. As mentioned, a row in the matrix contains the set of all dependent tests that has reacted for a component and the set of every dependant test is received from *SHmatrix*. Tests that is affected by a component is represented by a value in the corresponding column in *SHmatrix*.

In the second part of step 3 a score is calculated for each component and a ranking of the components are done. This ranking is done with the use of the false and missed alarms in the way described in section 6.2.2. The ranking and score is then stored in the variable *fdiResult* which is presented to the user. A detailed description about the variable *fdiResult* is found in the section 8.3

In Figure 8-4 is a detailed illustration of step 3. The figure shows how the finding of false and missed alarms is done before the score for each component is calculated. This is necessary because the result from the first part is used in the second part where the score is decided and the ranking is done.

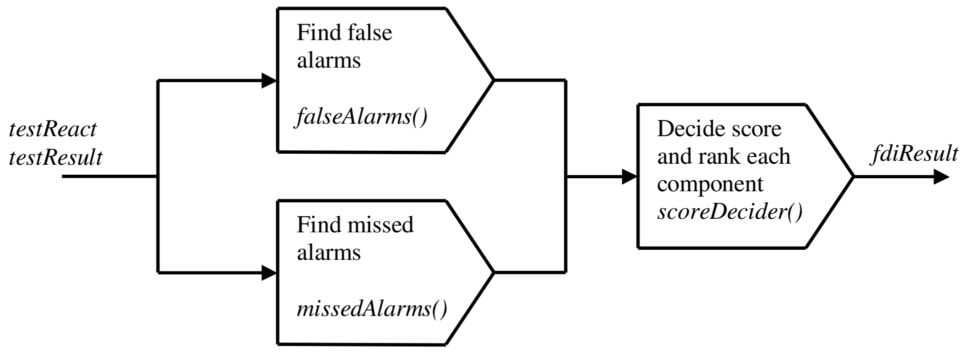


Figure 8-4 A detailed picture of step 3 in the implementation

8.2 Hypothesis Tests

This section contains a description of the different hypothesis tests that were implemented. Some of the tests are more complicated than others and a declaration is therefore required to understand the tests. A hypothesis test always returns a variable of type *timeArray* and all tests have the same kind of input parameters. These input parameters are always received in the same order, which is first *data* then an Id variable and finally a *startTime* followed by a *stopTime*. A test is structured like this: `timeArray = TestX(data, id, startTime, stopTime).` The reason for the fixed order is to simplify the insertion of new tests and to separate the application from the information.

checkTankOrder()

This function consists of three tests that checks three different tank empty orders depending on which tank that was stated in the call. With tank empty order means a certain order in which the fuel tanks in the aircraft are emptied. The order in which the fuel tanks in the aircraft should be emptied is gathered from section 2.3.1, and is: LD + RD → CD → LW + RW → T2 + T3 → T1. The reason why the tank empty order are divided into three tests are to avoid one big test that is affected by many components. Tests that are affected by many components make it harder to isolate the correct component and preferably avoided. The first test checks if the left and right drop tank is emptied before the central drop tank starts to defuel and if the drop tanks are empty before the wing tanks are starting to defuel. The second test checks if the wing tanks are emptied before T2 and T3 becomes empty. The third test checks if T2 and T3 are empty before T1 is starting to defuel.

findPlateau()

The function *findPlateau()* checks if there is a plateau in the graph that shows the fuel quantity for a tank, i.e. if the fuel quantity remains constant over a period of time. A plateau in a tank does not necessarily need to be due to a faulty probe, it can also depend on a faulty valve. Because of this, tests are only implemented for the different tanks that are connected to a valve in ARTU. It is no problem to extend this for other tanks that are connected to FRTU.

findJump()
The hypothesis tests for detecting jumps in the graphs that shows fuel quantity are divided into two categories. These are *findJumpUp()*, that finds a jump upwards in the graph, and *findJumpDown()*, that finds a jump downwards. With a jump up or down is meant an unnaturally big increase or decrease of fuel (y value) during a shorter period of time (x-interval). The reason that the tests are divided into up and down is that it is not the same components that affects the tests. For example a faulty valve conveys that a tank connected to it becomes immeasurable, which in turn leads that the remaining amount of fuel in the tank is withdrawn from the displayed fuel quantity and results in a downward jump in the graph. This is why *findJumpDown()* is affected by a faulty valve, while *findJumpUp()* is not.

checkProbeFW()
This function conveys of different tests that checks if a Failure Word for a probe has been set. These tests do not exist for the drop tanks because of the simple fact that the RUF data does not contain any variables that indicate FW for those tanks. There are however tests for every other fuel tank in the aircraft. A RUF variable that indicates a FW for a probe is probably set if the probe gives values outside of certain boundaries. Because of this a probe can be faulty without *checkProbeFW()* reacting.

checkProbeAndSensor()
The tests that are included in this function checks if the displayed fuel quantity matches with the indications from the current sensor of a fuel tank. There are two types of sensors, HLS (High Level Sensor) and LLS (Low Level Sensor), and they indicate if there is too much fuel in a tank respectively if a tank is almost empty. The low level indicators in the drop tanks are switches instead of sensors, but their functionality is the same. It is not every tank in the aircraft that has one of these sensors and the tests are only implemented for the ones that does. If the sensors' value does not match with the fuel quantity the test reacts. The *checkProbeAndSensor()* reacts for example if the RUF variable for a HLS is set and the tank is not full or if a variable connected to LLS indicates that a tank is almost empty but the graph that shows fuel quantity says it is not.

checkOpenFault()
This function checks a RUF variable that contains a FW for ARTU. This FW indicates if there is an open fault at any valve in ARTU and are therefore affected by all its valves. It is uncertain what an open fault exactly indicates and this is discussed in section 8.7.

checkAccessOK()
The tests in this function checks if the aircraft has access to all fuel tanks that are connected to ARTU. If a tank is not accessible this can be due to a faulty valve. The tests only include fuel tanks that are connected to the ARTU, since there only exists RUF variables for these.

checkMeasurable()
These tests check a RUF variable that indicates if a fuel tank is measurable or not.

With measurable means that the amount of fuel in a tank can be measured and if this is not possible it could be due to a faulty probe. There can be other reasons to why a tank is not measurable and perhaps they should be prerequisite to this function. This is an issue that is deeper discussed in section 8.7.

checkOutOfBounds()
This function contains tests for every fuel tank in the aircraft and what is tested is if a fuel quantity is illicit. An illicit fuel quantity exists if a tank shows to contain more fuel than it is physically possible or if the fuel quantity is shown to be less than zero. The tests in *checkOutOfBounds()* are affected by faulty probes.

checkValveFW()
The tests contained in the function *checkValveFW()* checks if a FW for a valve in ARTU has been set. Since this thesis is limited towards the ARTU, no tests for the other valves in the fuel system have been implemented. It would be possible to extend the function for every valve since RUF contains variables for them all.

checkIdFW()
These tests check a FW that indicates if any drop tank has an id failure. An id failure occurs if the aircraft can not identify a drop tank and this can be due to a faulty connection between the aircraft and the drop tank. There is also a RUF variable that indicates a general id failure for all the drop tanks.

checkSensorFW()
The tests in this function checks if a FW for any sensor or switch has been set. There is a high level sensor in the Vent Tank (VT) that is tested and also the low level switch in the drop tanks is tested. For a location of the VT, see Appendix A. Further on the low level sensors in the wing tanks and in T1 are tested.

8.3 Variables, Constants and Data Types.

This section contains a description of the variables and constants that are used in the application. Some of the variables are also own defined data types and has a more significant role in the program. How the variables and constants are passed and used in the application is described in section 8.1. The variables that are own defined data types are *timeArray*, *nrOfZero* and the variables that contain the results from the hypothesis tests.

Data and Event list
Data is a global constant that contains all the RUF data. It can be said that *data* contains variables since it contains all the RUF variables. It is the variables in *data* that the hypothesis tests examine when they are performed. The constant *data* is global for the entire process but are mostly used by the hypothesis tests.

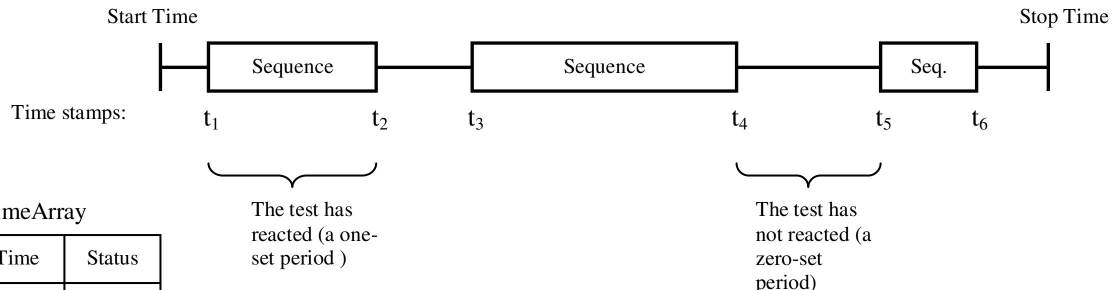
Event list is a constant that is strongly connected to *data* and it contains comprehensive information about the flight. This information consists of times when different events occurred. The events can for example be time for take off, the time when different alarms happened and time for touch down.

Time variables
There are two variables that states what time during the flight that a hypothesis test

should begin and end. In other words which part of the flight that should be tested. The variables are called *startTime* and *stopTime* and are as default set to time for take off respectively time for touch down. This can however be changed to an arbitrary value by the user. The variables are input parameters to every hypothesis test and are set in a function called *timeDecider()*. The time in the variables are given in milliseconds.

timeArray

timeArray, also called TA, is a self defined data type that contains the answer from a hypothesis test. The output from the tests is for that reason always a TA. The variable includes two fields; one is called *time* and it contains a time stamp for when a sequence begins or ends, the other is called *status* and it indicates if the time stamp is the beginning or the end of a sequence. With a sequence means a time period when the test connected to the TA has reacted. For each time stamp in TA there is a corresponding *status* variable and if it contains a ‘1’ this indicates the beginning of a sequence. A ‘0’ in the *status* variable indicates the end of a sequence. A *timeArray* is illustrated in Figure 8-5. The figure contains two different approaches to illustrate a TA. The first is displayed along a time axis with the sequences represented as blocks. The second is in matrix form and closer to the implementation. If a TA contains a sequence it means that the test that returned the variable has reacted during the time of the sequence.


Start Time
Stop Time
Sequence
Sequence
Seq.
Time stamps:
t1
t2
t3
t4
t5
t6
The test has reacted (a one-set period)
The test has not reacted (a zero-set period)

timeArray

Time	Status
t_1	1
t_2	0
t_3	1
t_4	0
t_5	1
t_6	0

The timeArray can be seen as a two column matrix, where the first column contains the time stamps and the second column holds the status.

Figure 8-5 timeArray

Lists

There are a few different lists that are used in the application. One of these lists is *testList* which holds the number of every hypothesis test that shall be performed. After the function *testLimiter()* has limited the number of tests that shall be performed this information is stored in the *testList* and later used when these tests are performed.

There is a list that is called *compList* and it contains the components that are possibly faulty. This list is determined on the basis of the alarms that is raised and is later used by *testLimiter()* when it decides which tests that has to be performed.

testReact is a list that holds which hypothesis tests that has reacted during an isolation process. This list is among other things used to map a test against a result from the variable *testResult*. The first position in *testReact* contains information about which hypothesis test that has given the result that is in the first position in *testResult*.

Matrices
There are four different matrices that are used in the application and they are all loaded into the program from Microsoft Excel.

SHmatrix
The most important matrix of them all is the decision matrix, which contains information about probability between depending components and tests. You could say that the values in the matrix states how probable it is for a component to be faulty when a depending test has not reacted (a missed alarm), or in other words how good a test is at detecting a faulty component. This is described in section 6.2.2 in more detail. *SHmatrix* also holds information about which components that affect different tests. If a component affects a test, this is represented with a value at the corresponding position in the matrix.

SH2marix
Just like the matrix above, this also contains probability values between components and tests. The difference is that the values in *SH2matrix* are probabilities for independent test that has reacted, in other words the probability that a component is faulty when an independent test has reacted. The subsets represented in both *SHmatrix* and *SH2matrix* are illustrated in Figure 6-16. The solution with *SHmatrix* and *SH2matrix* differs from the theory described earlier using a decision table and an ESH-matrix. The information in the SH-matrices is the same as in the theory but it is only represented in another way.

function2NameMatrix
This matrix contains a list of text strings with parts of the function names for the hypothesis tests. Every row in the matrix holds a part of a function name that corresponds to a test in any of the SH matrices. This string is used in the application to build a complete function name and is later evaluated in order to make a function call to the test that shall be performed.

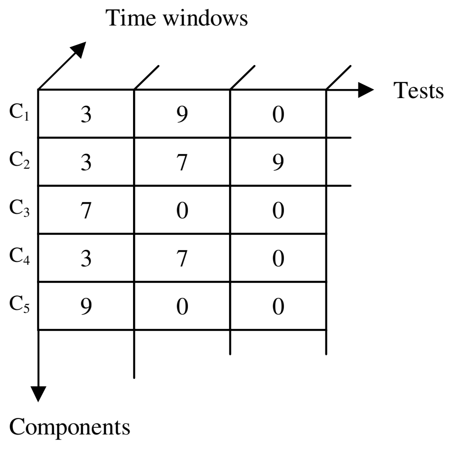
alarm2CompMatrix
When the application shall limit the number of tests it has to perform, it uses the *alarm2CompMatrix*. This matrix holds information about components that possibly can be faulty when a certain alarm has been raised. This connection is described in the Aircraft Maintenance Publications (AMP37). Earlier fault isolations have shown that the connection in AMP37 is not entirely complete. This leads to that the list of possibly faulty components must be increased with additional components with the help from people with expert knowledge.

Test result

The results from the different hypothesis tests are saved in a variable that is called *testResult*. This variable is a vector that contains a *timeArray* in every position, which is a natural structure since every position corresponds to the result from a test that has reacted. It is only the result from reacted tests that are stored. *testResult* is created in a function called *testPerformer()* and are sent to a function called *timeHandler()* where it is modified to *testResultTH*. The difference between the two variables is that the later is adapted in accordance to how the application handles the time aspect. In other words the *testResultTH* is a time handled *testResult*. The last two letters (*TH*) in the variable name stands for Time Handled. This time adaptation is necessary for an easy interpretation of the test results.

reactList

This variable is a matrix with three dimensions that corresponds to component, tests that has reacted and are affected by a component and finally time. The number of columns in the matrix is equal to the number of tests that has reacted, which is the same as the size of *testReact*. The size of the dimension for time is equal to the number of time windows for the flight and the size of the dimension for components is equal to the number of components in the decision matrix. It would be possible to make a reduction of the dimension for components since the only ones interesting is the components that affect any of the tests that has reacted. This would not lead to any greater benefits, so it has not been done. *reactList* is as mentioned a matrix and it contains for every component which tests that has reacted and are influenced by the component. An example of how *reactList* can look like is shown in Figure 8-6. For each test that has reacted but does not affect a component in the matrix there is a zero in that components row. The matrix is among other used to calculate the number of false alarms for a component. How this is done is described in section 8.1.



In this example *reactList* contains three tests (3, 7 and 9) and five components. It is only the second component that affects all three of the reacted tests. This is seen by the fact that it has no zeros in its row.

Figure 8-6 reactList

nrOfZeros

This is an own defined data type that in large can be described as a summary of *reactList*. The information stored in *nrOfZeros* is actually the number of zeros that each component have in its row in *reactList*. In other words the number of zeros in each row in *reactList* is counted and the sum is stored in *nrOfZeros*. The variable consists of three different fields and they contain information about how many zeros each component in *reactList* has. They are also sorted in ascending order with the lowest number of zeros first. The structure of the variable looks as follows: *nrOfZeros(TW).rank(x).comp / .FA*. The foundation of *nrOfZeros* is a vector where each position corresponds to a time window and contains another vector that is a sorted list of the components with least amount of zeros first. Since the application uses one time window, the first vector (*nrOfZeros(TW)*) has no real function and exists only for extension possibilities. The vector *rank(x)* has two fields. One of them is called *comp* and specifies which component the current position in *rank(x)* holds. The other one is called *FA* and contains the number of zeros for the current component.

fdiResult
The variable *fdiResult* contains the final result of the fault isolation process and it is this variables content that is presented to the user of the program. fdiResult is a vector that contains five fields in each position. Every position in the vector corresponds to a components and the fields hold information about that component. The five fields are: *score*, *comp*, *falseAlarms*, *missedAlarms* and *nrOfFalseAlarms*. The field *comp* indicates which component the position in the vector corresponds to and *score* holds the components' score. The score are the value the components are sorted by. The fields *falseAlarms* and *missedAlarms* contain which alarms that are false respectively missed. Finally the field *nrOfFalseAlarms* contains the number of false alarms. The variable *fdiResult* is constructed for easy storage and access to the final result of fault isolation. The structure of the variable looks like this: *fdiResult(x).score / .comp / .falseAlarms(y) / .missedAlarms(z) / .nrOfFalseAlarms.*

Id variables
The id variables are a collection of three types of variables, *tankId*, *probeId* and *valveId*, which identifies different components. It is usually one of these variables that decide which of the hypothesis tests in a function that should be performed. The id variables are used as input parameters for the tests. *tankId* identifies an individual tank or a collection of tanks, like for example RW for the tanks in the right wing. *probeId* indicates a probe in a fuel tank and *valveId* identifies a valve in the aircraft. *checkValveFW()* can be taken as an example. The call to the function looks like `timeArray = checkValveFW(data, valveId, startTime, stopTime)` and as second input parameter the id variable *valveId* is used. The function contains several hypothesis tests and *valveId* determines which one of them that is performed. If *valveId* contains the Id for the valve connected to the left wing, the test checks a FW for that valve.

8.4 Time Aspect

As mentioned in section 6.2.3 the hypothesis tests has a dimension in time what has to be handled in an appropriate way. This part of the thesis report is about how the time aspect has been handled in the program. There are several different ways to handle the aspect, even if they do not differ that much. What needs to be done is to treat the different sequences, which indicates that a test has reacted, so that they in a good way can be compared.

In this thesis the time aspect has been handled in a way that stretches the sequences in every *timeArray* both forward and backward in time. In order to make easy comparisons between all TA and to have fixed start and stop times for the stretching, one time window (TW) is used for the entire flight. This time windows constitute of the period in time between start and stop time. Every sequence in the test results are stretched across the entire time window. To have just one time window helps simplifying the method without any major drawbacks. A condition for the time handling is that every test result has the same start and stop time. If this is not the case different sequences can end and start at different time and the time windows will then have different length. Then the advantages of the time windows will be lost. The application has been implemented in a way that sets every start time to the same value and every stop times to the same value.

The stretch of the test results forward in time can be motivated with the reasoning that a fault never can heal by itself. If a fault once has occurred it will be in the system for the rest of the flight even if a test that reacted to the fault stops reacting before the end of the flight. When the time aspect is handled a stretch backwards in time is also made. This stretch is made to the beginning of the time window, which begins at start time. Start time is in the application set to the time for take off. If a *timeArray* contain more than one sequence, it would still be stretched as one sequence during the entire TW. How a test result is handled with regard to the time aspect is illustrated in Figure 8-7 below. In the figure the sequence in the *timeArray* is the test result before it is handled and is shown closest to the axis. The test has reacted from approximately 2500 sec to little more than 4000 sec. How the test result looks after it has been handled with regard to the time aspect is shown above the original result. It has been stretched across the entire time window. The TW goes from start time to stop time, which as shown in the figure is not from time zero to the end of the flight. Instead it is from take off to touch down.

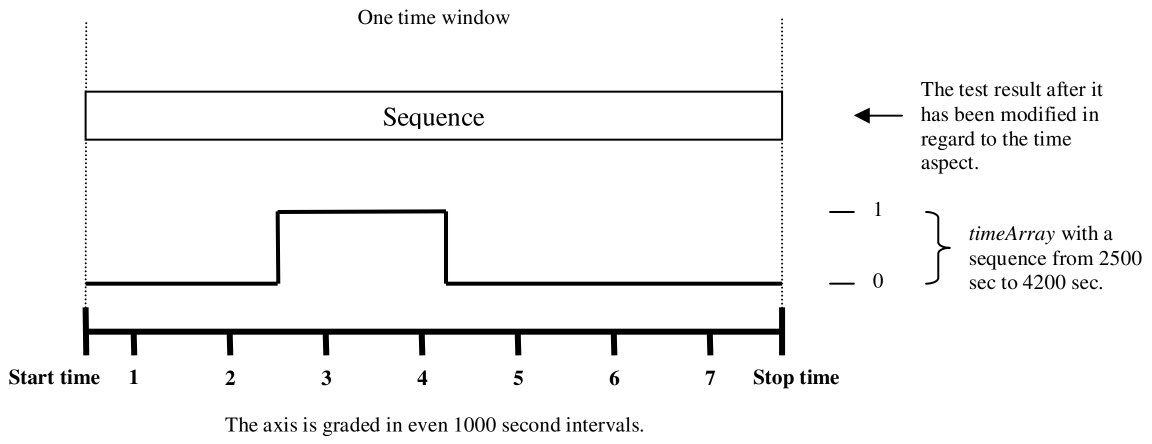

One time window
Sequence
The test result after it has been modified in regard to the time aspect.
1
0
timeArray with a sequence from 2500 sec to 4200 sec.
Start time
1
2
3
4
5
6
7
Stop time
The axis is graded in even 1000 second intervals.

Figure 8-7 A test result handled with regard to the time aspect

The function that handles the time aspect is called *timeHandler()* and the variable it modifies is *testResult*, that contains the answers from the hypothesis tests. The handling that is described above is done once for every test result, i.e. once for every position in *testResult*. The modified result is saved in the variable *testResultTH*.

8.5 Thresholds

Thresholds are a constantly recurring issue in the application and foremost among the hypothesis tests. Since all data that is being treated in the application has a time dimension, the thresholds are often over a period in time. However this is not always the case and the threshold can also be in a value of a variable. An example of where a threshold in time is used is in the findPlateau(). This function checks if the represented fuel quantity stays constant during a period of time. The length of this period is a threshold in time. Thresholding can affect the tests in different ways, like increase or decrease their sensitivity. There is a risk of false alarms if a test becomes too sensitive and on the contrary a risk of missed alarms if it becomes too insensitive. We have not used any method or put any effort in trying to make smart decisions of the thresholds in this thesis. Thresholding is a big area and it would take to much time to use smart methods for it. Instead the thresholds have been set to arbitrary values that seam to work good during the testing of the application. Among the hypothesis tests described in section 8.2 it is *checkTankOrder()*, *findPlateau()*, *findJump()*, *checkProbeAndSensor()* and *checkOutOfBounds()* that uses any kind of thresholds.

There is also built in thresholds in the RUF variables. An example of this is the variables for the sensors that indicate if a fuel tank is empty or not. In these variables there is a threshold for when the sensor considers a tank to be empty. The test C compares the variable for a low level sensor with a test of the represented fuel quantity. The test of the represented fuel quantity checks if the tank is empty and has a threshold for when the tank is considered empty. Both the LLS variable and the fuel quantity test has thresholds and in order to make the comparison as good as possible they should be set to the same value.

8.6 Application vs. Information

Below follows a deeper description about which modifications that are needed when the aircraft configuration is changed. This includes a description about how the application is separated from the information and how easy it is to adapt the application to different aircraft configurations.

When maintenance is done to the application there are actually not many changes that need to be done. As it was explained in section 7.1.5 maintenance is mainly constituted of two operations; the first is implementing the function that performs the hypothesis test and the second is modifying the different matrices.

As mentioned in section 7.1.5 knowledge about the aircraft and RUF is needed to implement a test, but almost no knowledge is needed about the application. The knowledge needed about the application is limited to a data type and how a function call for a test should look like. A requirement made by the application is that every test has the same input and output when it is called. Since the output from a hypothesis test always consists of the data type *timeArray*, the maintenance demands knowledge about how a *timeArray* is constructed and what information it holds. The input to a function that performs a hypothesis test is always of the same type and sent in a specified order. This demands knowledge about the input variables, their order and their content. The reason for the specified order of the input is that the application builds the function name based on different variables and the information in a matrix. The function name is built in a certain order that have to match the order in the function itself. These function names are later evaluated in the program when a hypothesis test should be performed.

When it comes to the modification of the matrices the implemented solution differs some what from the theory presented in section 7.1.3. The theory presented two matrices that needed change during maintenance, but in the application it is actually four matrices. First and foremost change is needed in the *SHmatrix* and the *SH2matrix* which contains probabilities and connections between tests and components that affects them. The required changes are described in section 7.1.3. Another matrix that needs to be changed if a new test is added is the *function2NameMatrix*. The fourth matrix that could need to be modified is the one that contains information about which component that are possibly faulty when an alarm has been raised, i.e. the *alarm2CompMatrix*.

A detail that has been included in the development phase of the application is the ability to make all the changes and modifications in the maintenance during run time of the program. This conveys that no restart or new compilation of the program is needed when maintenance is done. A great deal of this simplicity comes from the way that the functions are evaluated and that all information exists in external files that are separated from the application. A contributing fact is also that the matrices used in the program are read before an isolation is to be performed and in that way updated before every fault isolation is performed.

When the program is at the end-user, the maintenance will not be done in run time and updates of the program will come in packages. A package will include new versions of the matrices and a complete set of functions for the hypothesis tests. This package is added to the application and an update has been done.

As seen the required knowledge about the application during maintenance is keep to a minimum. This is because the application is separated from the information in a clear and structured way.

8.7 Problems

A problem with the application is that all the information about the RUF variables is not known. This can for example be how the variables are decided or in some cases even what they indicate. The information have not emerged from the documents that where studied during the thesis. Most likely Saab has this information somewhere, but due to lack of time we have not been able to find it.

There are several problems that come up when complete information about the variables that the tests are performed on is not available. One of them is that the prerequisites for the hypothesis tests can be hard to decide and this can lead to unnecessary double tests. When it comes to the prerequisitions the problem can be described with the case for the function *checkMeasurable()*. The tests in the function checks a variable that indicates if the fuel quantity in a tank is measurable or not. As described in section 2.5, the quantity of fuel in an aircraft is not measurable during certain flying conditions. These conditions are determined as restrictions on LV (load vector). The question is now if the variables that indicate if a tank is measurable is a test for these restrictions on LV or if that should be a prerequisite for the function *checkMeasurable()*. To answer that question deeper knowledge about what the variables indicate is necessary. A consequence of having a test on LV as a prerequisite and that it is the same test that the variable itself indicates is an unnecessary redundancy. The variable indicating measurable does probably not perform a test on LV since there are several such variables for different fuel tanks. If LV were to exceed its restrictions, every fuel tank in the aircraft except NGT would become immeasurable. It is for this reason not necessary with several variables that indicate if a tank is measurable.

There is another case when it in greater extent has been uncertain what the variable indicates. This is for the function *checkOpenFault()* that checks a variable that indicates an open fault for a valve in ARTU. There has not been any information available about what a open fault even is. Is it a valve that can not be opened or is it a valve that are stuck in an open position? This test can however contribute to the fault isolation without knowing this information. The variable that is checked is a failure word and if a variable like that reacts, something is faulty. Even if more information would have been better, the little that exists can still bring information to the diagnosis.

There are some obvious difficulties in deciding how the decision matrix should look like, that is which crosses a test should have. There are two kinds of problems when it comes to the decision matrix. Either there can be a component that affects a test and do not get a cross, or there can be a cross for a component that do not affect a test. In both cases it adds flaws to the fault isolation and a consequence can be faulty components that are not isolated. It can also lead to the wrong component being pointed as faulty. A great deal of the accuracy and reliability of the isolation lays in minimizing these sorts of errors. In order to accomplish this exhaustive information about the variables in RUF are required. Since the decision matrix in this thesis is done without all the necessary information it can contain faults like the ones mentioned above.

Chapter 9

Conclusions

Method 1 using one agent for each component was at first thought of as a good idea with the possibility of implementing the most crucial agent first and later finishing with the less important. This procedure works fine as long as the component does not interact with each other. The interaction forces an agent to gather information of how the surrounding components work before it can specify how the agent's component works. Rather soon is realized that this method has no way of structuring the knowledge of how components interacts and how the information may be modified when needed.

Method 2 is invented from *Structured Hypothesis Tests* that has a dependency matrix specifying some if the information of how components interact: It specifies how a test can be raised from several components even if the test intentionally where designed for one single component. One demand on the system is to get a ranked list of components most probable to be broken. To manage this, the theory of probabilistic reasoning system that handles uncertainty is used. Uncertainty is represented as false and missed alarms, which can be rather frequent depending on how thresholds for automatized test are set. This led us to an extension of structured hypothesis tests that even has the ability of multiple fault detection.

The intention was to implement both of the methods but method 1 was after a time proven to be full of flaws and all problems were not even possible to solve. Therefore only a prototype using method 2 implemented fully.

Unfortunate there has not been enough time or data available to conduct a complete testing of the implementation. A few minor tests have been made and they show good results, but a more thorough testing has to be performed before any deeper conclusions can be drawn about the implementation.

Did the prototype fulfill the demands?

As stated in Chapter 6 and Chapter 7, method 2 fulfills all eight demands. So does the prototype building on method 2. The multiple fault detection has not been implemented due to time. We believe that the theory works and it would be interesting to see multiple fault detection in action, since it is not so many systems that do handle multiple faults.

Automatic vs. manual fault detection

Since we only have developed a prototype and not fed it with knowledge this comparison is not feasible. Depending on how the knowledge is put into the system it can produce diagnoses with different accuracy. This will be further explained in 9.1. One thing that is for certain is that multiple fault detection is almost impossible to do manually, and if it is proven to work well in our system it will have a great advantage to manual fault detection.

9.1 Future Work

Theoretical

We have shown how to fuse data from both present and created test to get a list of components containing a score of how each component may explain the test results. The next step is to put knowledge into the system. This can be done in two ways:

1. An expert of the aircraft specifies the probability $P(T_j \mid C_i) \quad \forall i, j$ as stated in section 6.2.2. The accuracy depends on how well the experts know the system and can specify the probabilities.
2. A learning framework is built outside the decision support system to train the system to give the components its score. In this case a less experienced expert can build up the knowledge by feeding the system with flight data and specifying what components are broken in each flight. This way the learning framework calculates the probabilities on its own. The accuracy depends on how many training flights the system is fed with. A fair guess is that at least one flight for each component, with the component broken in that flight, has to be used. It is rather hard to tell exactly what was broken during a flight, and this way the system will probably not get any better than the expert. [9]

These two ways are pretty different; the first one just tells the system how to calculate the scores and the latter one tells the system how to think. When taking over the development, the developer has to choose which way to go depending on the experts' knowledge.

Further on two interfaces to the user has to be built. One for the insertion of knowledge, used by experts and programmers, and one for the regular use of the decision support system. These two has to be tailor made according to other software used by technicians of the aircraft. This way it will be easy for the technicians to learn and use the program.

Practical

A good continuation of this thesis would be study the RUF variables in more detail. Example of questions that should be answered is: What is it a failure word truly indicated? How is it set? As an example a flight with a faulty probe can be mentioned. In this example a failure word for a valve reacted but it was actually a probe that was faulty. In order to understand and handle this, a deeper knowledge about the RUF data is required. Also the prerequisite for the RUF variables needs to be studied in more detail. This is directly linked to what the tests that decides the variables actually do. All this information is important when deciding how the hypothesis tests shall be designed. It is also important in order to get an understanding of how good a test is, which components it affects and also which tests that are needed in the isolation process.

Since step 4 in the process of method 2 is not implemented this could be done as future work. Step 4 consists of making a final ranking of the components with more information considered. An example of information that can be considered is: expert knowledge, mean time between failure, statistic about faulty components, cost for the components and time it takes to change the components. This extension is a question about how much information that should be considered and how they should be weighed between themselves. A suggestion is to show a list where all the information in present in its own column and to have one column that sums up the weighing of the information into a “change value”. The components can then be sorted after this “change value”.

Bibliography

[1] Saab AB, *General Description Publication, Fuel System*,
Linköping 2005, J3-A-37-00-00-00A-040E-A

[2] Saab AB, *General Description Publication, General systems electronic control unit,*
Linköping 2005, J3-A-37-00-00-00A-040E-A

[3] Saab AB, *Detailed Description Publication, Fuel system,*
Linköping 2005, J3-A-37-00-00-00A-040G-A

[4] Saab AB, *Testmetodbeskrivning,*
Linköping 2005 JSU2-37-TMB:4184

[5] Saab AB. *Detailed Description Publication, Maintenance data recording system*,
Linköping 2005, J3-A-69-00-00-00A-040G-A

[6] Gustafsson F., Ljung L, , Millnert M., *Signalbehandling*, Studentlitteratur,
Lund 2001, ISBN 914401709X

[7] Nyberg M., Frisk E, *Model Based Diagnosis of Technical Processes*,
Linköping 2005

[8] Russell S., Norvig P,, *Artificial Intelligence, A modern approach 1st ed*, Prentice
Hall, New Jersey, 1995. ISBN 0131038052

[9] Russell S., Norvig P., *Artificial Intelligence, A modern approach 2nd ed*, Prentice
Hall, New Jersey 2003. ISBN 0130803022

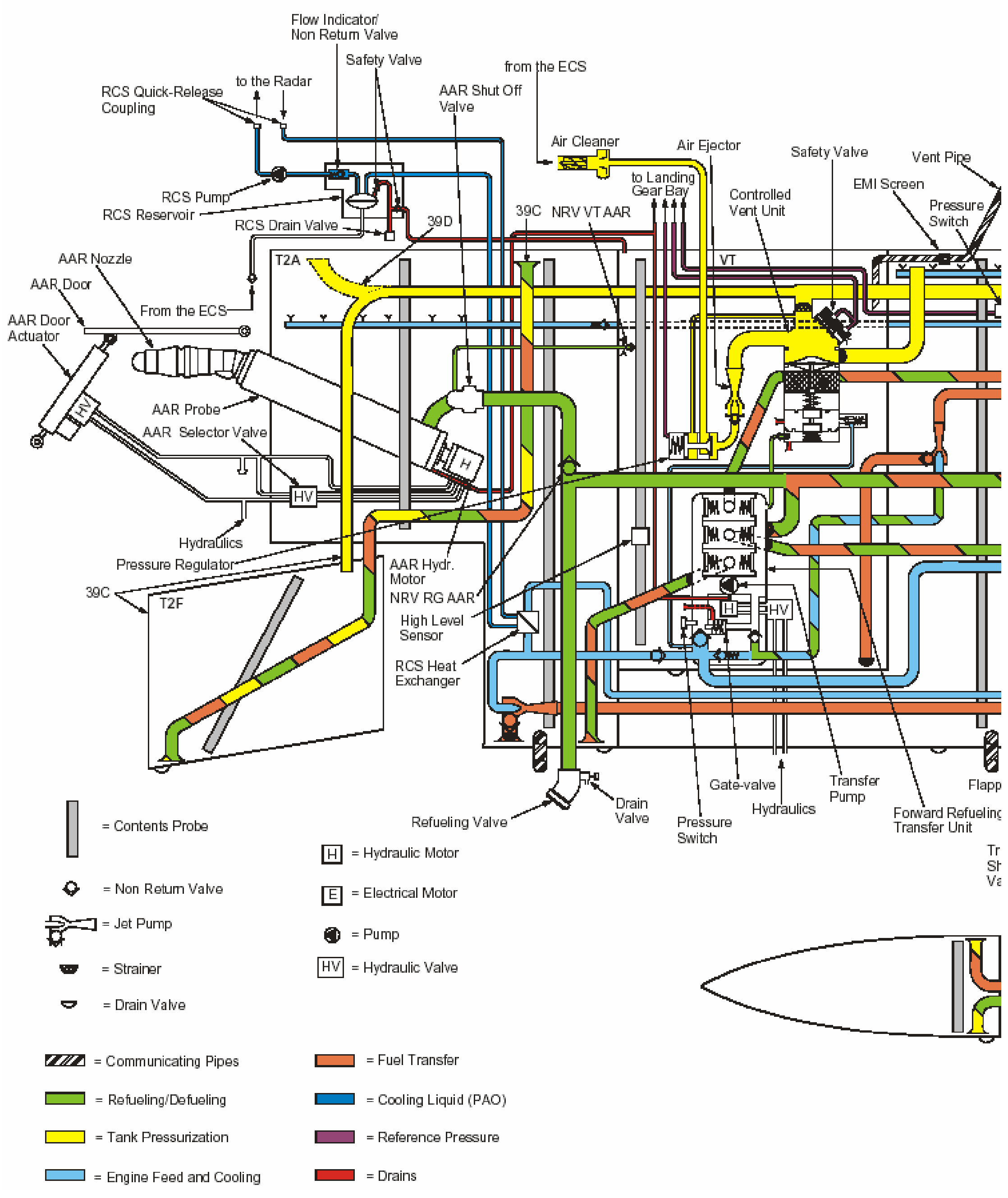
[10] Saab AB. *Aircraft Maintenance Publication, Fuel system*,
Linköping 2005, J3-A-37-00-00-00A-002A-A

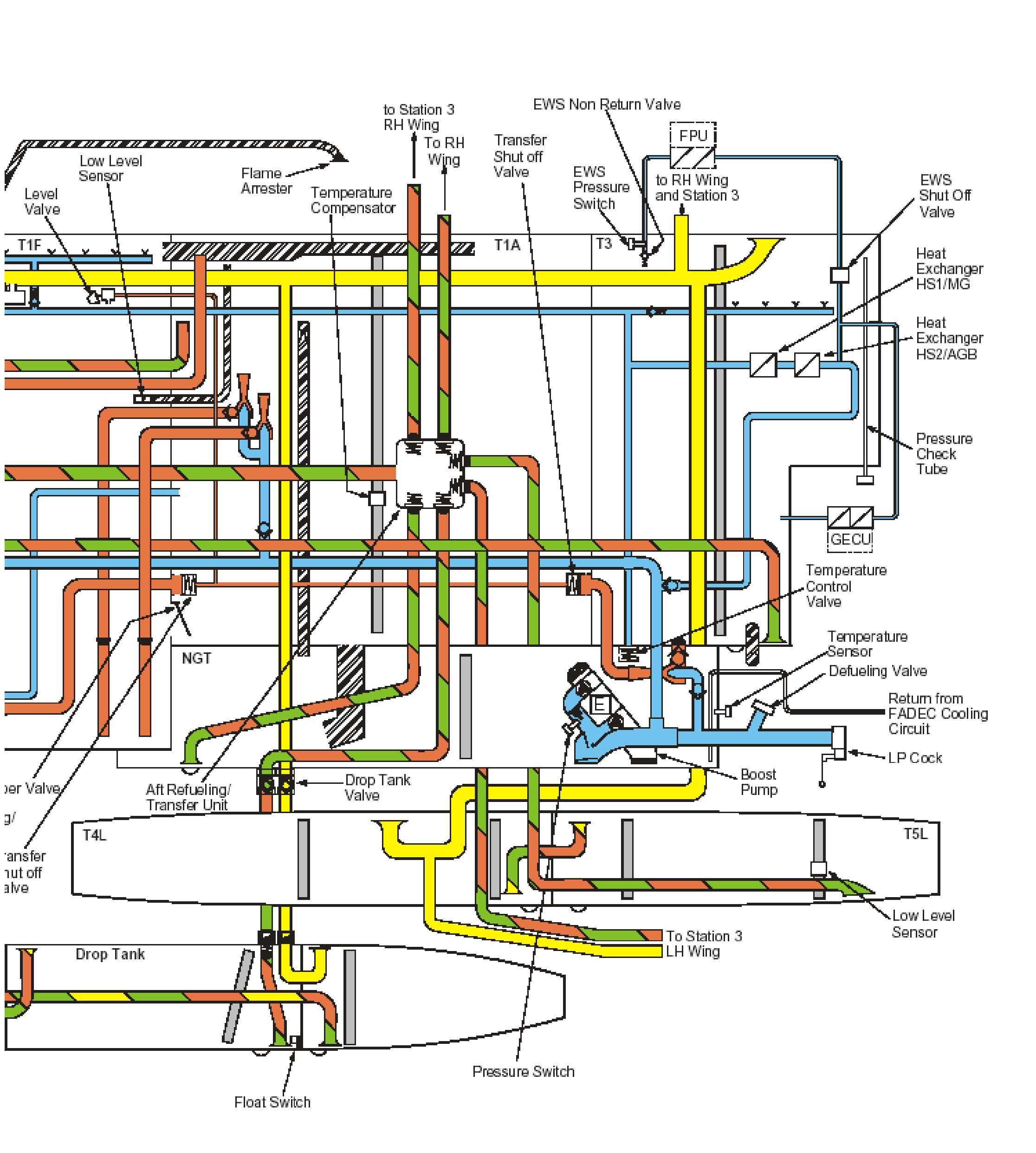
Abbreviations

AIU	–	Aircraft Interface Unit
ARTU	–	Afterward Refueling Transfer Unit
BIT	–	Built In Test
CVU	–	Controlled Vent Unit
ESH	–	Extended Structured Hypothesis tests
FDI	–	Fault Detection and Isolation
FM	–	Function Monitoring
FRTU	–	Forward Refueling Transfer Unit
GECU	–	General Electronic Control Unit
GUI	–	Graphical User Interface
JAS	–	Jakt Attack Spaning
NGT	–	Negative-G Tank
RUF	–	Registration Used for maintenance and Fight security
SC	–	Safety Check
SH	–	Structured Hypothesis tests
SysC	–	System Computer
TA	–	Time Array
TW	–	Time Window

Appendix A

Next pages contain a picture of the complete fuel system.


Flow Indicator/
Non Return Valve
Safety Valve
from the ECS
to the Radar
RCS Quick-Release
Coupling
AAR Shut Off
Valve
Air Cleaner
Air Ejector
Safety Valve
Vent Pipe
to Landing
Gear Bay
EMI Screen
RCS Pump
Controlled
Vent Unit
Pressure
Switch
RCS Reservoir
39C
NRV VT AAR
RCS Drain Valve
39D
AAR Nozzle
T2A
VT
AAR Door
From the ECS
AAR Door
Actuator
HV
AAR Probe
AAR Selector Valve
H
HV
Hydraulics
Pressure Regulator
AAR Hydr.
Motor
39C
NRV RG AAR
T2F
High Level
Sensor
HV
H
RCS Heat
Exchanger
Gate-valve
Transfer
Pump
Flapp
Drain
Valve
Hydraulics
Refueling Valve
Pressure
Switch
Forward Refueling
Transfer Unit
Tr
Sh
Va
= Contents Probe
= Non Return Valve
= Jet Pump
= Strainer
= Drain Valve
H = Hydraulic Motor
E = Electrical Motor
= Pump
HV = Hydraulic Valve
= Communicating Pipes
= Refueling/Defueling
= Tank Pressurization
= Engine Feed and Cooling
= Fuel Transfer
= Cooling Liquid (PAO)
= Reference Pressure
= Drains


to Station 3
RH Wing
To RH
Wing
EWS Non Return Valve
FPU
Transfer
Shut off
Valve
EWS
Pressure
Switch
to RH Wing
and Station 3
EWS
Shut Off
Valve
Low Level
Sensor
Level
Valve
Flame
Arrester
Temperature
Compensator
T1F
T1A
T3
Heat
Exchanger
HS1/MG
Heat
Exchanger
HS2/AGB
Pressure
Check
Tube
GECU
Temperature
Control
Valve
Temperature
Sensor
Defueling Valve
Return from
FADEC Cooling
Circuit
NGT
E
LP Cock
Boost
Pump
per Valve
Aft Refueling/
Transfer Unit
Drop Tank
Valve
g/
ansfer
hut off
alve
T4L
T5L
Low Level
Sensor
To Station 3
LH Wing
Drop Tank
Pressure Switch
Float Switch