SAM

Towards a context and situation aware cockpit

Harreman, Richard (0521597)
Roest, Maikel van der (0518922)

Technical Report DKS-03-06 / ICE 06
Version 1.0, 30 June, 2003
Mediamatics / Data and Knowledge Systems group
Department of Information Technology and Systems
Delft University of Technology, The Netherlands

Harreman, Richard R.A. (0521597) (dateq@warhammer.xs4all.nl)
Roest van der, Maikel M. (0518922) (maikel@roparun.nl)

“SAM: Towards a context and situation aware cockpit”

Technical Report DKS-03-06 / ICE 06
Version 1.0, 30 June,

Mediamatics / Data and Knowledge Systems group
Department of Information Technology and Systems
Delft University of Technology, The Netherlands

http://www.kbs.twi.tudelft.nl/Research/Projects/ICE/

Keywords: ICE project, knowledge based system,

Preface

This thesis is related to our research activities on the development of the Flight Gear Simulator Situation Awareness Module (SAM for short) at the Knowledge Based Systems (KBS for short) group of the Delft University of Technology.
We developed the 'Flight Gear Situation Awareness Module' for one of the projects of the KBS group, which is called the Intelligent Cockpit Environment (ICE) project.
The goal of this thesis is to introduce the reader to situation awareness, explain the concept and provide a detailed overview of how SAM is implemented.
The target audience are computer scientists and the graduation committee. To fully comprehend this thesis, some basic knowledge of artificial intelligence and JAVA is required.

Richard Harreman,
Maikel van der Roest,
Delft, June 2003

Summary

Computers and computer programs are becoming more complex as time progresses. The digital revolution has also made an entrance into the cockpit of an aircraft. A pilot gets more information than ever before in an ever decreasing amount of time. This could lead to an "information overload", which in essence is a concept of too much information provided in a very short time. An "intelligent cockpit" should solve this problem.

In order to achieve an intelligent cockpit the following project goals are set:

- Literature study
- Cognitive and system model
- Artificial Intelligence techniques research
- Demonstrator
- Evaluation, tests and validation

All this combined should form a template or framework for future development and research in the study of situation recognition.

Thesis overview

This report is separated into three parts. The first part, chapters 1 to 3, contains the project description and preliminary research in the field of intelligence, development tools and different kinds of reasoning techniques.
The second part, chapters 4 to 6 contains the design by description, schematic representations and artificial intelligence.
The last part, chapters 7 and 8, contains the project review and the evaluation.

Used abbreviations

CLIPS	C Language Integrated Production Systems
DTD	Document Type Definition
ICE	Intelligent Cockpit Environment
JESS	Java Expert System Shell
KBS	Knowledge Base Systems (Group)
NASA	National Aeronautics and Space Administration
PERL	Practical Extraction and Report Language
PHP	PHP: Hypertext PreProcessor
SAM	Situation Awareness Module
SDW	System Developer Workbench
XML	eXtensible Markup Language

Acknowledgements

This project could not have come into being if it had not been for the help of ir. Mohammed Abdelghany and Drs. dr. Leon Rothkrantz
Mr. Abdelghany was instrumental in finding an organization for our traineeship and giving advice. Drs. Dr. Rothkrantz was very helpful as a guidance and advisory person and for giving us the opportunity to graduate with this project.
We also would like to thank Quint Mouthaan and Patrick Ehlert for their help.

(This page has been left blank intentionally)

Table of contents

Table of figures

(This page has been left blank intentionally)

Chapter 1: Introduction

1.1 Project description

This thesis was written as part of our graduation project at the Hogeschool Rotterdam. Our graduation project is part of the Intelligent Cockpit Environment (ICE) project, started by Drs. dr. L. Rothkrantz who is appointed at the Delft University of Technology. On of the goals of the ICE project is to gather knowledge and experiment with adaptive interfaces for aircrafts. In other words, the research on different methods to give a pilot the correct information at the right moment.

Figure 1 Part of a cockpit of a Boeing 737.

As a result of the digital revolution within the aviation industry a pilot gets an increasing amount of information in a decreasing amount of time. This could lead to a so-called "information overload". If this is the case, the pilot is so busy, both mentally and physically, that he or she misses potentially important information.

The ICE system could help out in these kinds of situations. The system will monitor the behavior and the tasks the pilot is performing, analyze what the current situation of the aircraft and environment is, and keeps a track on what parts of the flight plan have finished. In the event of the pilot missing some potentially important information, or when the pilot fails to perform certain tasks, the ICE system issues a warning. In short, the final goal of the ICE project is to increase the "situation awareness" of the pilot, and to decrease his mental workload.

1.2 System overview

To develop a system which is capable of "situation awareness", the following data is needed to make an as accurately as possible assessment of the situation the pilot is in at a certain moment:

- The current state of the plane
- The current state of the environment
- The current state of the pilot
- The flightplan

This result in what the system should be able to do. The flight plan is loaded into the ICE system, just as the pilot would make a real flight plan that is used in flight. During the flight, the state of the aircraft is continuously assessed using the aircraft's instruments and measuring systems. These systems are also used to assess the environment the aircraft is in.

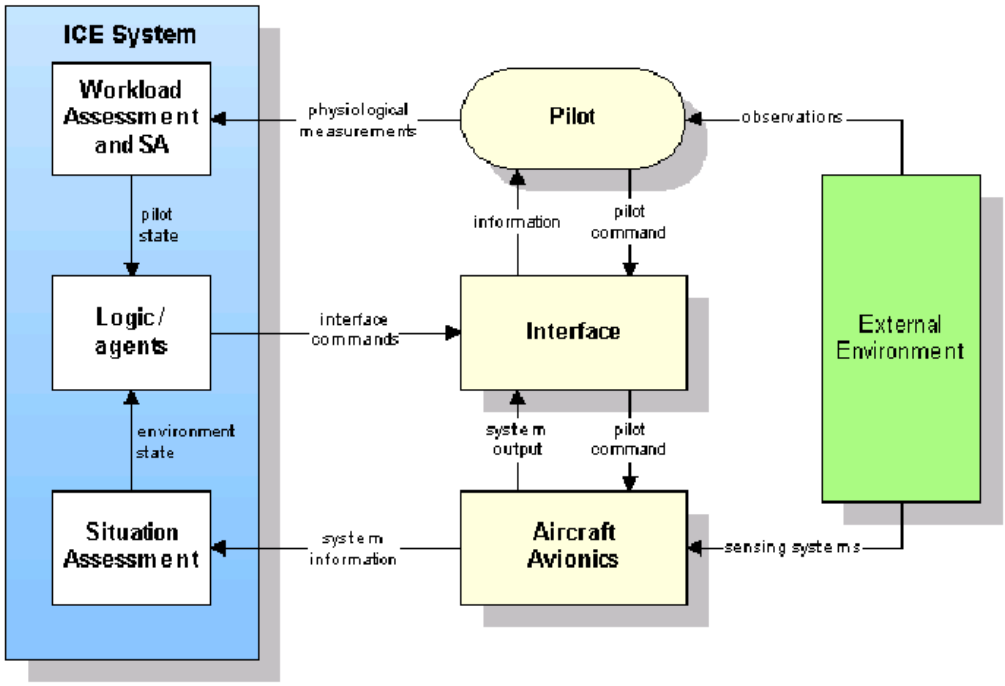


Figure 2 A system overview of the ICE project.

The assessment and detection of the state the pilot is in, is more complicated. With registering and analyzing all input that the pilot gives to the plane, the system can reason what the pilot is doing, but it gives very little information about the mental workload of the pilot. A possible solution to acquire the information about the mental workload of the pilot is, the use of a "gaze tracker" system.
A gaze tracker is a camera system that uses the reflection of infrared light on someone's eyes to assess what the person is looking at. Simultaneously, the gaze tracker measures the size of the pupils and the frequency of the eye-blinking.
These two factors combined give an indication of the stress level of the pilot.

As a third point of information the gaze tracker can also detect the view direction of the eyes; this way the system knows what the pilot is looking at.

Once the states of the pilot, environment and the aircraft are known, reasoning can start. The short response time of the pilot to situations (for example in an emergency situation) has a great influence on the system, which should reason and act in real-time. To simplify the problems, and to make parallel processing possible, the project is divided into several subprojects.

1.3 Sub-projects

The entirety of the ICE project, has been divided into several subprojects, which all contribute to the main project. Currently students are working on the following projects:

- *Flight Simulator Artificial Pilot by M. Andriambololona and P. Lefeuvre*
  The current project aims at implementing a bot for a flight simulator. By bot, it is meant an intelligent virtual player emulating a human player in the game environment. The bot requires being able to understand the rules of the game. The bot also needs other capabilities like flying through the game environment, and facing aircraft failures.
- *Automating the cockpit by M. Tamerius*
  The ultimate goal is to develop an autonomous, computer-driven pilot. Such a pilot, also called a bot, should be capable of flying an aircraft in a simulated environment according a flightplan. In addition, it should be capable of making in-flight decisions to adjust the actual flight path to reach the destination of the flightplan.
- *Situation Recognizer for F16 fighters by Q. Mouthaan*
  To create a system that is able to detect the current situation based on the state of the F16 fighter jet and the actions the pilot is performing. This is quite a challenging problem because, if the airplane is descending for example, this might mean the pilot is landing or that he is descending to the right altitude to perform an attack on a ground target. The system must be able to determine in real time which of these situations is actually occurring. In this document the reasoning process that is used to determine the situation is described.
- *Situation Awareness Module by R. Harreman and M. van der Roest*
  The SAM project aims at developing a situation recognizer that works in real-time. It incorporates the use of knowledge bases and the Java Expert System Shell (JESS). Using this technique it will be possible to work with dynamically adaptive rule sets to get accurate data about the situation the plane is in.
- *Workload assessment in the cockpit by P. Ehlert*
  To help a pilot deal with information processing and decision-making, avoid information overload, and optimize flight performance, a crew-assistant system or intelligent pilot-vehicle interface (PVI) has been proposed. To correctly assess the amount of information that a pilot can handle we need to know his (mental) workload. Therefore, we need to design a workload assessment module as part of the PVI system.

1.4 Project goal

The goal of the ICE project is to design, test, and evaluate computational techniques that can be used in the development of intelligent situation-aware flight crew assistance systems. Using methods from artificial intelligence, ICE focuses primarily on the data fusion, data processing and reasoning part of these systems. Special issues addressed in the ICE project are:

- Situation recognition
- Mission or flightplan monitoring
- Attack management
- Pilot workload monitoring

In the SAM sub-project of ICE we focused on the "Situation recognition" part. This resulted into the following goals of our project:

- Literature study
- Cognitive and system model
- Artificial Intelligence techniques research
- Demonstrator
- Evaluation, tests and validation

Chapter 2: Project file

2.1 Description

The project file will describe our approach to the problem setting. In this chapter the aspects to the project will be described. Things like the plan of approach, the project team and project planning will also be discussed.

2.2 Plan of approach

Constituent:	TU Delft
Approved by:	Ir. M. Abdelghany, Project council
Date:	Hogeschool Rotterdam
Author:	Maikel van der Roest
Mark:	"Situation Awareness Module"
Date:	3/3/2003

2.2.1 Background

After four years of study at the Hogeschool Rotterdam, our career as students was coming to an end, as we had to graduate. After a meeting with our graduation council Ir. M. Abdelghany, we came in touch with Drs. Dr. L.J.M. Rothkrantz. He invited us over to have a meeting with him, regarding the then-called "Adaptive Cockpit Environment" (ACE) project. This project was started by Drs. Dr. Rothkrantz, and he suggested that we could help out with this project, and so the idea for the SAM project was born. To avoid any misconceptions: ACE is the same as ICE. The name was changed to better suit the project.

2.2.2 Project description

2.2.2.1 Problem setting

Ever since man has walked on the earth he has dreamt about flying. The discovery of the kite by the Chinese started humans thinking about flying. Kites were used by the Chinese in religious ceremonies. For many centuries, humans have tried to fly just like birds and have studied the flight of birds.
The results were often disastrous, as the muscles of the human arms cannot move with the strength of a bird. Leonardo da Vinci made the first real studies of flight in the 1480's. He had over 100 drawings that illustrated his theories on bird and mechanical flight.
.

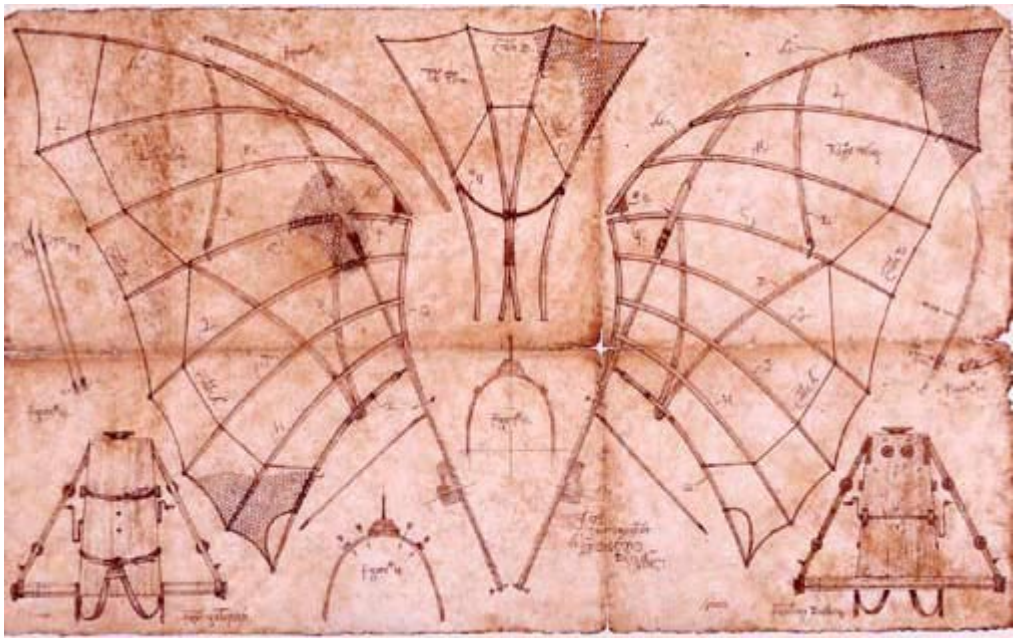


Figure 3 A drawing of the Ornithopter of Leonardo Da Vinci.

The Ornithopter flying machine (Figure 3) was never actually created. It was a design that Leonardo da Vinci created to show how man could fly. Leonardo da Vinci's notebooks on flight were re-examined in the 19th century by aviation pioneers. Based on the concept of Da Vinci the Wright brothers made the first successful manned flight in 1903. It lasted for 12 seconds.

Since those early days of aviation, the technology used in planes has evolved immensely whereas the principles of flight have remained the same. Alongside with this immense progress in technology used in planes, designed to reduce the skills needed to fly a plane, it inevitably also introduced an unexpected problem. The increased amount of information available to the pilot has decreased some of the physical workload, but has increased the mental workload.

Figure 4 Cockpit of the Concorde.

Figures 4 and 5 illustrate this problem. As the cockpit of the Concorde has many gauges and instruments the pilot should look at, the cockpit of the Boeing 737-700, as show in figure 5, uses information displays to feed the pilot with information. Although this is a big step forward into information management, and resulted into reducing the people needed to fly a plane to two persons, instead of three, the pilot still needs to choose the right screens to watch at, and the information available to him/her is much greater, as the technology evolves.

Figure 5 The glass cockpit of a Boeing 737-700.

2.2.2.2 Project goal

The ultimate project goal is to achieve a prototype system that assesses in real-time the situation the pilot is in and the workload of the pilot. The system will use a dynamically adaptable rule set to assess the current status of the flight, and predict what situations are likely to follow. Also adaptive logic needs to be defined which will be responsible for the information the system gives to the pilot.
To achieve this goal, the project has been divided up into three phases:

1st phase
- Literature study

2nd phase
- Artificial Intelligence techniques research
- Cognitive and system model

3rd phase
- Demonstrator
- Evaluation, tests and validation

Are more detailed summery of the different phases is the SAM project can be found in section 2.2.8.1 of this chapter.

2.2.2.3 Project result

The end result of the project should be a software program that works alongside the Flight Gear Simulator. This program uses knowledge bases and neural network technologies to comprehend and anticipate what the pilot is doing at the current moment, and what he or she is going to do in the near future.

2.2.3 Project limits

Taking in consideration the huge scale of the ICE and SAM project, the complexity, and the limited time frame, it is likely that the SAM program will not have all the features and functions as described in the plan of approach.

2.2.4 Project lifecycle

The project lifecycle describes the steps that need to be completed during the project.

- Preliminary research
- Draft of the plan of approach
- Analysis which results into a time planning
- Draft of the functional specifications
- Research into System Awareness
- Proof of concept SAM model
- Research into data communication
- Research into artificial intelligence
- Research into expert systems
- Research into further expansion of the SAM
- Draft of the requirements of the SAM
- Draft of the knowledge base layout
- Draft of the system model
- Visual prototype
- Programming of the Data server
- Programming of the Data modeler
- Programming of the Data client
- 1st modular prototype
- Implementation of the knowledge base into the program
- Implementation of JESS within the program
- 2nd modular prototype
- Delivery of SAM
- Draft report
- Draft of thesis

Further on in this thesis you can find a detailed time planning which chronologically describes which project members are responsible for the different aspects of the SAM project.

2.2.5 Project organization

2.2.5.1 Tasks and responsibilities

The team should cooperate in creating the Situation Awareness Module as described in the project goal

2.2.5.2 Execution conditions and competences

To properly complete the project, the team assumes the following engagements and acceptances:

- Each project members is responsible to completely fulfill his tasks.
- The self-reliant acquiring of information needed for the tasks is also a responsibility of the project member.
- All progress made within a task should be reported on weekly bases to the other project members.
- Applied methods should be discussed with the other project members, before they are used.
- Eventual delay should be reported as once, just as errors that could lead to delays.
- Any adaptations during the construction of the product should only be done by the responsible team member. In any other case the team and councils must always be consulted first.

2.2.5.3 Time schedule

The project is carried out from 3 October 2002 till 4 July 2003. For this project the hours used by the project members are estimated on 840 hours per person.

2.2.5.4 Information and reporting

All information, documents and software are stored on the TU server. This server allows the project members to share the new information with the other project members. All information is placed in the designated directory, and made public, when the information is ready for release.

Reporting the progress of the project is done ad hoc one time a week with the whole ICE project group. Every two weeks a progress report is given to the Hogeschool Rotterdam. The reporting is done to Ir. M. Abdelghany.

Communication can be done by telephone, e-mail and meetings. Evaluation of the project is found in the appendix of this thesis.

2.2.6 Projects risks

The following risks are taken into consideration. Per risk measures are taken to prevent them from happening.

- Fire
- Theft
- Specification which are not conform the end result
- Failed agreements
- Environmental disasters

To limit these risks a daily backup is made of all data.

2.2.7 Project budget and costs

Since there is no budget for this project, costs are left out of this thesis.

2.2.8 Project planning

The SAM project is divided in several phases to make an incremental approach of the problem setting. Using this incremental approach, the project goal can be adjusted during the project, as the project progress is not as planned.

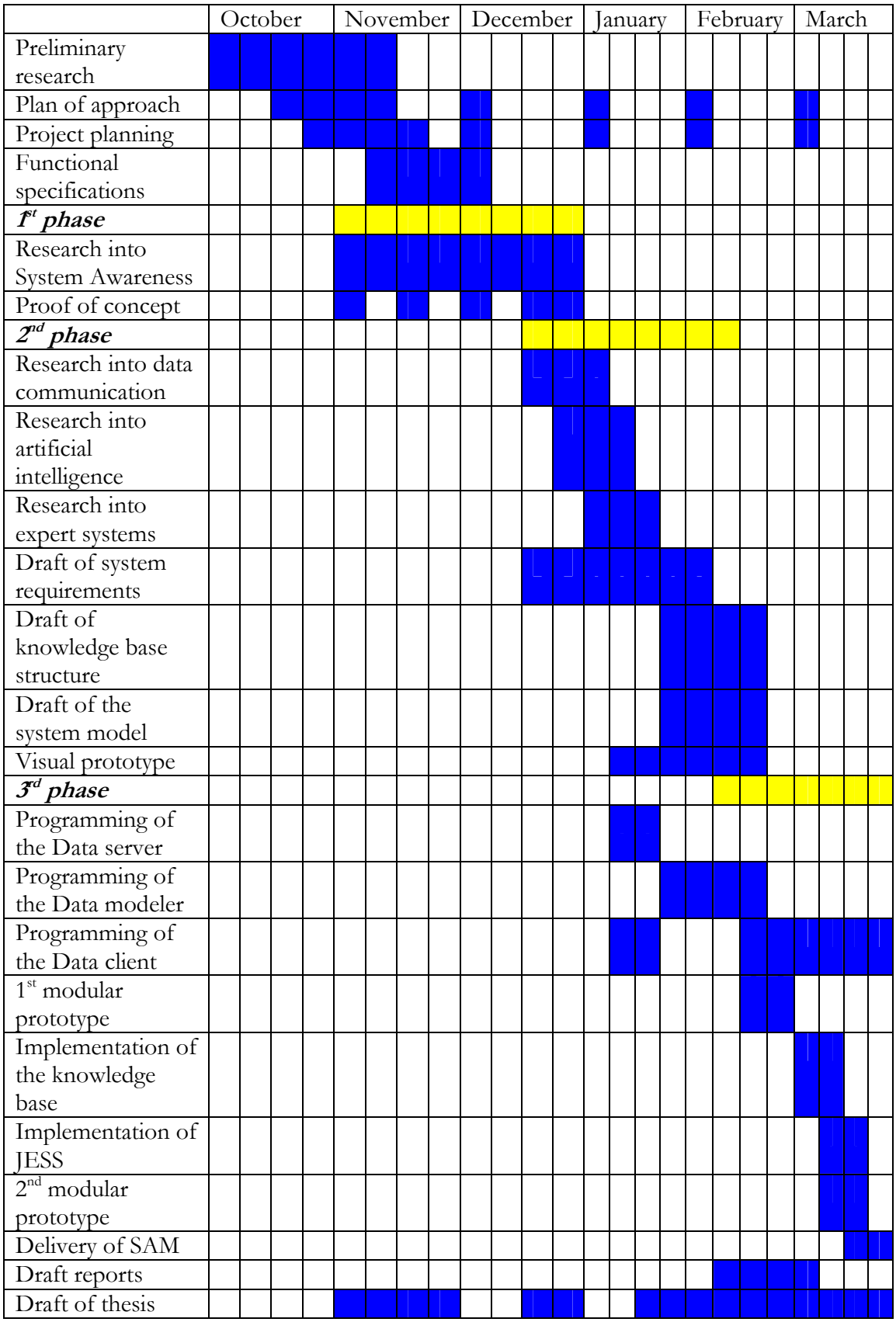
2.2.8.1 Definitions

• Preliminary research	Initial orientation into the ICE project and System Awareness
• Plan of approach	Complete description of the project
• Project planning	Time scheme of the complete project
• Functional specifications	Basic requirements of the Situation Awareness Module
1st phase	
• Research into System Awareness	Research into System Awareness and the application of it within the SAM project
• Proof of concept	A preliminary prototype which proves the concept described in our plan of approach
2nd phase	
• Research into data communication	Orientation and research into the communication between the SAM program and the flight simulator
• Research into artificial intelligence	Orientation and research into artificial intelligence and how it can be used in our program

• Research into expert systems	Orientation and research into expert system and how it can be integrated in the SAM program
• Draft of system requirements	All demands on the SAM program written down and fixed
• Draft of knowledge base structure	The layout and structure of the knowledge base written down a separate report
• Draft of the system model	The layout and structure of the SAM program defined
• Visual prototype	An extension to the proof of concept, which now uses the layout of the final prototype
3rd phase	
• Programming of the Data server	Creation of the code necessary to make the Data server
• Programming of the Data modeler	Creation of the code necessary to make the Data modeler
• Programming of the Data client	Creation of the code necessary to make the Data client
• 1st modular prototype	A first preliminary version of the program based on the functional specification
• Implementation of the knowledge base	Integration of the knowledge base into the SAM program
• Implementation of JESS	Integration of JESS into the SAM program
• 2nd modular prototype	A second preliminary version of the program which now complies to at least 80% of the requirements
• Delivery of SAM	A final prototype which complies for at least 90% to the requirements
• Draft reports	Creation of reports that cover parts of the SAM project which stand on them selves
• Draft of thesis	Creation of the final thesis which covers every aspect of the SAM project

2.2.8.2 Project time scheme

This is the latest project time scheme based on the original planning. As you can see the original date to end the project was March, but that date was too far-fetched for our project, as we were not happy with the results. Chapter 8 describes why this happened.



2.2.8.3 Project research areas and responsible project members

• Preliminary research	Maikel and Richard
• Plan of approach	Maikel
• Project planning	Maikel
• Functional specifications	Maikel
1st phase	
• Research into System Awareness	Maikel and Richard
• Proof of concept	Maikel and Richard
2nd phase	
• Research into data communication	Richard
• Research into artificial intelligence	Maikel and Richard
• Research into expert systems	Maikel and Richard
• Draft of system requirements	Maikel
• Draft of knowledge base structure	Maikel
• Draft of the system model	Maikel
• Visual prototype	Richard
3rd phase	
• Programming of the Data server	Richard
• Programming of the Data modeler	Richard
• Programming of the Data client	Richard
• 1st modular prototype	Richard
• Implementation of the knowledge base	Maikel and Richard
• Implementation of JESS	Richard
• 2nd modular prototype	Richard
• Delivery of SAM	Maikel and Richard
• Draft reports	Maikel and Richard
• Draft of thesis	Maikel and Richard

2.2.9 Project activities and to be delivered results

The following overview gives a specification of the tasks of the project group and the results that need to be delivered.

2.2.9.1 Tasks

- Drafting the plan of approach
- Drafting requirements
- Drafting specifications
- Drafting a test plan
- Creation of the knowledge base containing data on the Cessna 172RG
- Creation of the Data server
- Creation of the Data client
- Creation of the Data modeler
- Creation of prototypes, with increasing complexity
- Drafting of report on the knowledge base
- Drafting of report on the final prototype
- Writing the thesis

2.2.9.2 To be delivered results

- Report which describes the SAM prototype
- Report which describes the knowledge base
- Several prototypes of the SAM
- SAM which complies to the requirements
- Project thesis

2.3 Team

The SAM project team consists of 2 persons:	Richard Harreman Maikel van der Roest
The project council at the TU Delft:	Drs. Dr. Leon Rothkrantz
1st Project council of Hogeschool Rotterdam:	Ir. M. Mohammed Abdelghany
2nd Project council of Hogeschool Rotterdam:	Ing. Hans Manni

Chapter 3: Preliminary research

3.1 Description

This chapter discusses the preliminary research that was done as a primary analysis before the project started. Within this preliminary research we focused on the technical and reasoning aspects of the project. This preliminary research is used as framework for further analysis and has been of great importance for the eventual implementation of SAM. Because this chapter just describes the preliminary research, it is possible that certain aspects of the project have been discarded, which came to light with the draft of the requirements. It is of great importance that the final requirements are considered as the project goal.

The first step of the preliminary research is the transposing of the problem-setting onto a central question. The central question runs as follows:
"What demands need to be met for the development and implementation of SAM"
The objective of the research is to supply a user-friendly and functional model of SAM. This model needs to be usable by anyone who has some affinity with system- and situation awareness.

To make the initial research more accessible and more focused we have divided the central question into three shared questions. The first section of this chapter will cover the following shared question:
1. *"What demands need to be met in developing the system model?"*
The second section of this chapter will cover the following question:
2. *"What intelligence demands need to be met by the model, and which aspects are of use to the user of the program?"*
The last section of this chapter will address the last shared question, namely:
3. *"What technical demands need to be met by the model, and which aspects are of use for the user of the program?"*

3.2 System model aspects

Shared question
"What demands need to be met in developing the system model?"

3.2.1 Development tools

For the development of SAM we have chosen for a Microsoft Windows environment to do the final programming of SAM. One of the main requirements of SAM is that it is to be Operating System independent, so it could actually be any one Operating System, but for the Windows Operating System the most JAVA development tools are available.

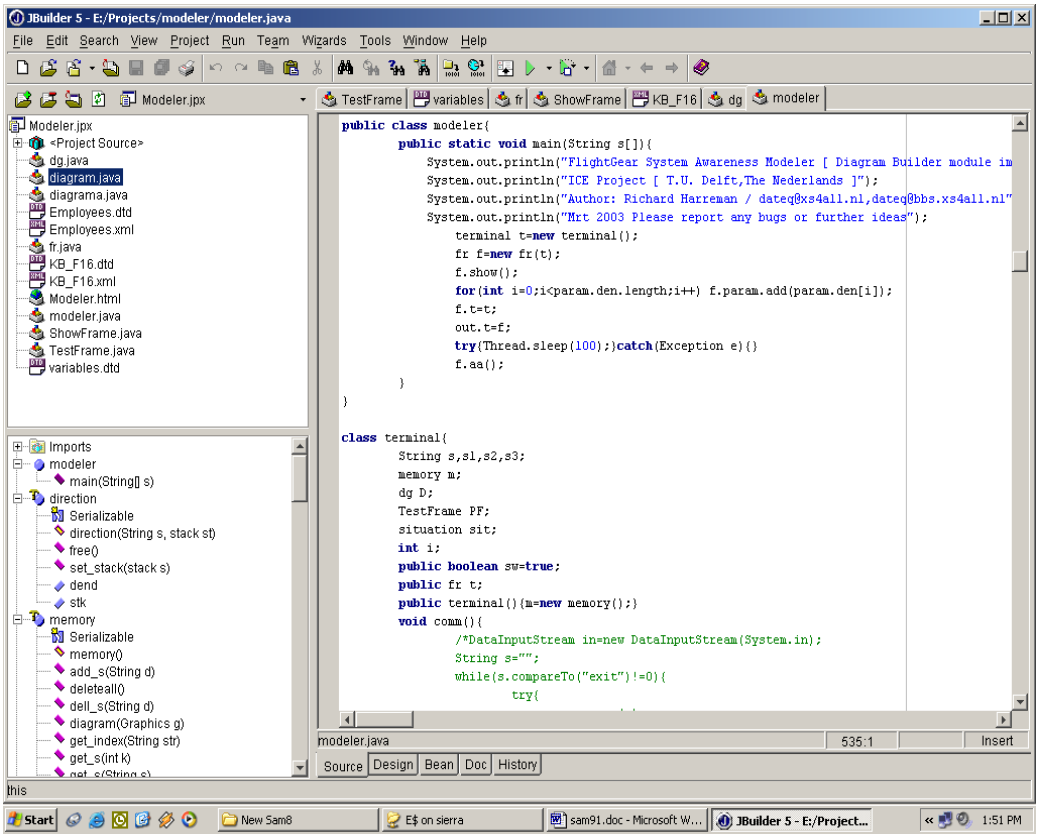


Figure 6 Development of Data Modeler in J-Builder environment.

The programming of SAM is done with the help of Borland J-Builder. This program has many similarities with the other Borland products, which the project team has used on many occasions in the past. The choice for J-builder will decrease the learning curve of the program, and will make the switch form C to JAVA more easy for the developers.

For the system development and analysis of the system in the first phase of the project, we have chosen for SDW. As the project progresses we will switch to Rational Rose, because SDW has some limits as the system gets more complex. To draw the system model we will use Microsoft Visio.

3.2.2 Development environment

As mentioned in the section above, the development environment is mainly Microsoft Windows. For the development we will use Windows NT 4.0 SP 6a, Windows 2000 and Windows XP, in combination with Borland J-Builder, SDW, Rational Rose 2000, JAVA SDK 2.0, Microsoft Visio 2002 and Microsoft Office XP.

3.2.2.1 Requirements for the user

It is essential not to have too many demands of the user. The main requirement of the user of SAM is that he or she has the JAVA Virtual Machine installed. If the user wants to do real time analyzing of the data, Flight Gear with at least version number 0.91 needs to be installed on the computer of the user.

3.2.2.2 Program testing

Whenever a prototype of SAM is finished, it will be tested. These tests are performed to evaluate the performance and accuracy of the prototype, which enables the way for improvements in following prototypes. The main points during testing are:

- Overall performance of the program
- Speed of data analysis
- Accuracy of the system
- Speed of the data transfer between the simulator and the program
- Validation of the model

3.2.3 Maintenance

Because of the modular build-up of the program and the use of external XML knowledge bases, SAM is relatively simple to maintain and improve. By simply changing some values in the knowledge base, the operation of the program can be manipulated. The program itself also gives the user the possibility to change certain aspects of the program trough the Data Modeler. For the more demanding users the source code of the program also uses the modular build-up, so that parts can be relatively simply changed, without having to compile the whole program again.

3.3 Intelligence aspects

Shared question
"What intelligence demands need to be met by the model, and which aspects are of use to the user of the program?"

3.3.1 Description

The most important aspect of the "SAM" project is the Artificial Intelligence aspect. The focal points for the reasoning are:

- Quality of reasoning
- Validation
- Transparency
- Explanation of results
- Efficiency

In order to accomplish this, the data that enters the system needs to be analyzed, and using this data, a situation needs to be recognized using a reasoning algorithm. To be able to process and analyze the incoming data a reasoning system is needed. Limiting ourselves to the context of our project and the knowledge already present at the Knowledge Base Group at TU delft, we had the choice between two systems. The following part of the report will cover those two systems and will discuss the advantages and disadvantages of the systems.

3.3.2 Neural networks

3.3.2.1 Description

A neural network is either a hardware implementation or a computer program that tries to accomplish the processing of data in the same way as its biological counterpart does. Usual neural networks are built up with a great number of artificially connected neurons.

In contradiction to a normal computer program, a neural network has some extra and unique features, like self learning, self organizing, error tolerance, and parallel information processing.

3.3.2.2 Operation

Neural networks are self learning because of the way they deal and react to the data the network provides it. Instead of telling the network how to react and interpret the data, a neural network has the possibilities to sort out the properties of the data. The neural network continuously learns when new data is available. When data is supplied to the neural network, the neural network organizes its structure to visualize the properties of the data it has found. Figure 5 is a representation of a feed forward neural network. A neural network like this can be used to train the system to find a relation between the input and output it encounters. In most cases the meaning of a self learning neural network is as following, the organization of data using the strength of connections between the neurons. The way how that organization takes place depends on the set learning algorithm. Differences between neural networks can be by different internal structures or the learning algorithm.

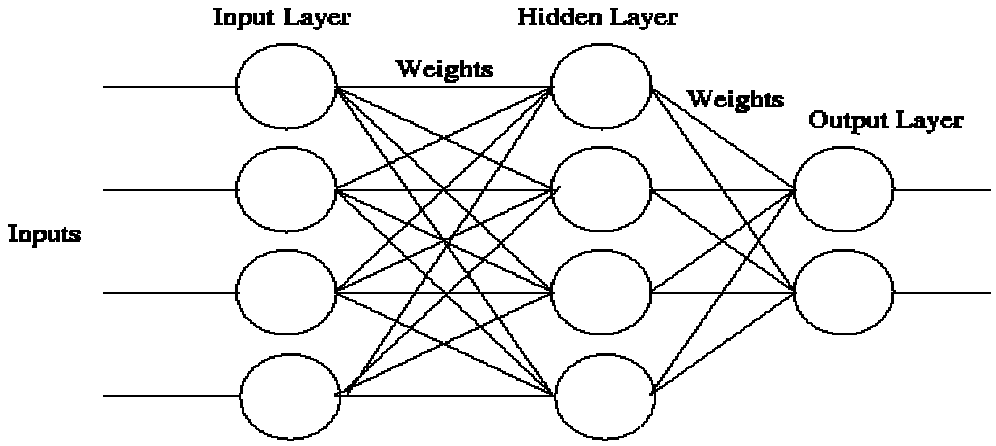


Figure 7 A feed forward neural network, trained by back propagation.

Error tolerance is an important aspect of a neural network. It reflects the property of the network to create models using the entered data. In other words, a neural network is able to find a generalization within the data.

3.3.2.3 Usability within the SAM project

In the original idea of the SAM project, neural networks would be the ideal choice. Nevertheless it was ruled out after the end of the preliminary research. The main problem with a neural network is to let it learn if it's right or wrong. This is on its own not a big problem, as long as the number of variables is limited. Within the SAM project we incorporate over 80 variables, which makes it at this point in time, impossible to use a neural network with the time span of the SAM project. Not only that, the simulator cannot provide so many variables on a real-time basis. Appendix D contains a example of what the source code of a feed forward neural network looks like.

3.3.3 Expert systems

3.3.3.1 Description

An expert system is used to solve problems for people which in normal cases need a specialized person to solve the problem, for example a doctor. To be able to build an expert system it is necessary to first gather the needed information in combination with the human expert. Much of this kind of information is kept up to date by the expert, using assumptions instead of actual facts. Especially because of this reason it is hard to gain the right information of the expert. There even is an own specialty for these experts: "Knowledge Engineer". A knowledge engineer has the task to gather information and to put it into the "knowledge base" of the expert system.

The most widely used way to extract knowledge out of an expert system is by the use of rules. Usually these rules do not have a specific solution. There is more certainty that the solution is right when the conditions do not change. Statistics are used to predict the certainty of the solutions of the expert system. Rule based expert systems are easy to manipulate because of the use of knowledge bases.

Expert systems are used to solve a variety of problems, for example in the areas of informatics, law and mathematics. Within each area the expert system can solve different sets of problems.

3.3.3.2 Operation

An expert system works by means of reasoning. The expert system matches the incoming data against knowledge rules, and then determines which situation the user is in. The knowledge rules need to be registered into the expert system before it is used. With the use of algorithms the knowledge "rules" in combination with the incoming data will be converted into "facts".
The simplest algorithm that can be used within an expert system is the "Boolean" algorithm. A Boolean is nothing more than an "if" statement. It checks if the stated rule is true or false. An expert system is basically based on this way of reasoning. The way of how data is interpreted and handled differs per system. The biggest advantage of this system is that the conclusion is always correct, as long as the rules are stated correctly. The disadvantage of the Boolean approach is that the system is not flexible, because the rules are always based on "hard data".

To give an example within the context of our project: If an airplane is on final approach the speed needs to be 70 knots. When the speed is 71, or 69, the system will pass the variable as false, and will conclude that the airplane is not on final approach. More complex expert system circumvents this disadvantage with algorithms and logics, which enables a certain flexibility, but will also decrease the certainty factor.

3.3.3.3 Expert Systems

There is a wide variety of expert systems shells available. Within the context of the SAM project we have chosen to take a closer look at the following expert systems shells.

CLIPS

Clips (C Language Integrated Production System) is an expert system shell developed by NASA in 1986. Since those days' CLIPS has improved, and has gotten a lot more sophisticated. CLIPS is a "tool" that is designed to make the development of assessment systems easier. Outside the fact that CLIPS is a stand-alone tool, it can also be used as function call within a programming language. CLIPS is designed to work with programming languages like C and C++.

JESS

JESS (Java Expert System Shell) is in its original form a clone of CLIPS, developed in 1995. Trough time JESS has evolved to a complete stand-alone expert system shell. Because JESS is based on CLIPS, it is possible to use CLIPS rules in JESS. One of the problems a programmer may encounter is that some facts can activate many if-then rules. In order to select the most relevant rule to "fire", JESS uses the Rete algorithm. JESS not only has the complete CLIPS function set, but also some extra features of it own, like backwards chaining and memory queries. That makes JESS an efficient, small, and one of the fastest expert systems available.

3.3.3.4 Usability within the SAM project

After the initial research and rule out of a neural network to use within the SAM project, expert system shells came into view. Taking the big amount of variables into consideration and that we wanted to be able to follow the reasoning process; expert systems were a better solution to our problem. Incorporating the use of knowledge bases, it is possible to make a dynamic and accurate system. That left us with the question of which expert system shell to use. We had narrowed it down to two choices, JESS or CLIPS. The choice between JESS and CLIPS was not a big one to make. CLIPS has stopped with further development since 2001, which inevitable also rules out any updates in cases of bugs. JESS is still under development, and has an active user and support group.
Therefore, JESS was our choice.

3.3.4 Prototypes

During our preliminary research into artificial intelligence we started to make the first prototypes. We made both a PHP based, and a PERL based prototype, dependent on the programming skills of the team members. Both prototypes use simple if-then statements to do the assessment of the situations. The following code example illustrates this:

```
function Takeoff ($Airspeed,$Rpm,$Throttle,$Brakes,$ParkingBrakes,$Pitch,$VerticalSpeed,$Elevator, $Altitude)
{
//Takeoff functions
//At takeoff the flaps can be set to 0,10,20 or 30 degrees, depending on the runway situation
//The elevator needs to increase at about 50 knots(v1), and the takeoff will take place at 70/80 knots(v2)
if($Brakes == "Off" && $ParkingBrakes == "Off") {
  //Checking whether we are taxiing or taking off
  //Implementing of groundlevel is needed, to see if we're of the ground or not
  //Formula: /enviroment/ground-level-m / /steam/attitude-ft = 3.14
  //So if the groundlevel == 3.14 then were still cruising on the deck, instead of in-air
  if($Rpm >= "1200" && $Airspeed >= "20") {
     $Vertical = round ($VerticalSpeed);
     echo $Vertical;
     if($Airspeed <= "50" && Vertical == "0") {
       echo "Takeoff: Rolling $eol";
       }
       if($Airspeed >= "50" && $Airspeed <= "64" && $Vertical == "0") {
           echo "Takeoff: V1 speed $eol";
           }
       if($Airspeed >= "65" && $Altitude <= "100") {
           //For this concept we only work with the kfso airport
           if($Elevator <= "0" && $Pitch >= 2 && $VerticalSpeed >= 2) {
              echo "Takeoff: Taking off $eol";
              }
           else {
              if($Airspeed <= "80") {
                 echo "Takeoff: V2 speed $eol";
                 }
              }
            }
          }
        }
      }
```

Figure 8 A PHP code example of the first prototype.

The previous code describes the 'Takeoff' function. It analyzes the data that was inputted into the function, and uses the variables to give back a situation. As you can see all the data is hard, which means, it cannot deviate from it. In real life, all pilots have to obey the rules, but minor deviations from the rules are allowed. This means that a following prototype should have a feature to overcome this problem. The connection between the prototype and the simulator was done by a TELNET connection. This worked in the following way:

```
$cfgServer    = "127.0.0.1";
$cfgPort    = 5555;
$cfgTimeOut    = 10;
//Newlines? <br> or \n
$eol    = "\n";
//Printout the variables?
$Test = false;

// open a socket
if(!$cfgTimeOut)
  // without timeout
  $fgfs_handle = fsockopen($cfgServer, $cfgPort);
else
  // with timeout
  $fgfs_handle = fsockopen($cfgServer, $cfgPort, &$errno, &$errstr, $cfgTimeOut);

if(!$fgfs_handle) {
  echo "Connection failed$eol";
  exit();
  }
else {
   echo "Connected $eol";
   //sleep(10);
   echo "Running script, this may take a while $eol";

   //Switch to raw mode
   fputs($fgfs_handle, "data \015\012");

   //Get the Flightdata
   //Get Airspeed
      fputs($fgfs_handle, "get /steam/airspeed-kt \015\012");
      $Airspeed = fgets($fgfs_handle,1024);
   //Get Altitude
      fputs($fgfs_handle, "get /steam/altitude-ft \015\012");
      $Altitude = fgets($fgfs_handle,1024);
   //Get Heading
      fputs($fgfs_handle, "get /steam/gyro-compass-deg \015\012");
      $Heading = fgets($fgfs_handle,1024);
   //Get Aircraft pitch
      fputs($fgfs_handle, "get /orientation/pitch-deg \015\012");
      $Pitch = fgets($fgfs_handle,1024);
   //Get Aircraft roll
      fputs($fgfs_handle, "get /orientation/roll-deg \015\012");
      $Roll = fgets($fgfs_handle,1024);
   //Get Aircraft yaw
      fputs($fgfs_handle, "get /velocities/side-slip-deg \015\012");
      $Yaw = fgets($fgfs_handle,1024);
   //Get Engine rpm
      fputs($fgfs_handle, "get /engines/engine/rpm \015\012");
      $Rpm = fgets($fgfs_handle,1024);
   //Get Vertical Speed
      fputs($fgfs_handle, "get /velocities/vertical-speed-fps \015\012");
      $VerticalSpeed = fgets($fgfs_handle,1024);
   //Get Ground Level
      fputs($fgfs_handle, "get /fdm/jsbsim/ic/terrain-altitude-ft \015\012");
      $GroundLevel = fgets($fgfs_handle,1024);
```

Figure 9 The connection between the program and the simulator.

As one can probably see this is a pretty basic function. Although this is an effective and easy way to obtain data, the socket connection with Flight Gear was slow.

Switching back to an earlier version of Flight Gear solved this problem, but reduced the amount of variables, which ruled out that solution. The bug was submitted to the Flight Gear user group, and will hopefully be solved in upcoming versions of Flight Gear.

3.3.5 SAM reasoning

To conclude the artificial intelligence part of our preliminary research we had to make some choices. As neural networks were ruled out quite early, and we chose to use an expert system, we did some more research into the use of JESS. A second prototype revealed the same problem as we encountered with the first prototype. If the situation does not have the ideal variables, it is possible that the system analyzes the wrong situation. To minimize the chances for this error, because it's virtually impossible to prevent it from happening, we have taken the following precautions.

3.3.5.1 XML Knowledge base

To keep the program as dynamically as possible and easy maintainable, all knowledge needed to reason is contained into a knowledge base. The knowledge base itself is written in XML, an easy to learn language, and contains all the situations and rules the program can recognize. The report on this knowledge base is included in Appendix A.

The idea of the knowledge base was inspired by Quint Mouthaan, who made a XML knowledge base for a F-16. This knowledge base contains information about which instruments the pilot should look at, and in what situation the plane is. Looking at the fact that not all actions will be carried out, but they actually should be, all actions are given a priority and probability value. To give a basic example: when a pilot is in the "Take Off" phase, it's most likely that the pilot will open up the throttle to 100%, thus the probability is 1, but the chance that the pilot will set his transponder to code 1200 is only 0.5. This all has to do with the priority of actions. If the pilot wants to take off, he has to open up the throttle, but it is not needed to set the transponder at 1200, thus the priority values differs.

3.3.5.2 Temporal reasoning

One of the main additions to the SAM project, as well as the ICE project, should be temporal reasoning. This mechanism will be used to check the situation analyzed by the expert system shell, and to anticipate on upcoming situations.

The main idea behind temporal reasoning is the human reactions to any kind of problem. A pilot knows what to do when he encounters an anomaly or an emergency, but a computer program does not. The pilot can also dynamically adapt the situation, if his or her solution is not working. Temporal reasoning as thought of in the SAM project, is a first setup to try to solve this problem.

The principle behind this is a timeline. A pilot takes off, and lands the plane, which can be marked as start- and endpoint. Between these two points he or she will complete some tasks, which are predefined in the flight plan. These situations will

be converted into a table, which contains a summary of the actions, and the following logical actions which can happen after it. Using a timeline, the system can anticipate the logical next situation the pilot will encounter, and the system can if necessary advice the pilot.

Parallel to this timeline runs a second table, which contains possible alternatives for the standard situations. In this second timeline the program searches for environment variables which could have a negative effect on the flight. This function runs recursively, and checks the first timeline for any alternatives.

Looking at the fact that a plane can also encounter emergency situations, there is a third alternate timeline implemented. In this timeline all emergency situations are included, with the possible situations that solve the emergency, as is described in the knowledge base. If a situation of this timeline is detected, then the system gives it top priority to get back into a normal flight situation as fast as possible. In future projects this third timeline can also be used to let the program perform the tasks needed, and to deny the pilot access to the flight control systems.

A more detailed description of the temporal recursion system is found in chapter 6.

3.4 Technical aspects

Shared question

"What technical demands need to be met by the model, and which aspects are of use for the user of the program?"

3.4.1 Description

This part of our thesis will cover all the technical aspects involving the SAM project. As described in the previous section of this thesis, one of the main focal points of the SAM project is "speed". In order to meet that demand the program itself needs to comply with certain aspects, that enable the quick handling of data.

3.4.2 Data processing

Data is being delivered by the Flight Gear simulator and will be processed and analyzed by the various modules of the program.

The simulator uses two kinds of protocols to deliver data to any kind of program, namely HTTP (Web-protocol) and TELNET, respectively on ports 80 and 23. The problem we faced quite early on in the project was the "slowness" of the data being delivered. If you want for example 40 variables delivered by the simulator it can take up to 5 seconds for all variables to be sent. Then of course the program needs to "translate" and analyze those variables, which will take some amount of time. In the worst-case scenario you will be looking at up to 7 seconds of time that is lost every time a request is made of the simulator for data.

3.4.3 Real-time analyzing

In order to be able to do real-time analyses within the program so that accurate information is given to the pilot, the team counteracted the 7 seconds as stated in chapter 3.4.2 by making a selection for the data needed at any given situation in flight. After all, when a pilot is taking off, there is no need for the variables that are used while taxiing from the runway i.e. the program does not need data from the wheel breaks, parking brakes and so on.

To realize this, the team introduced a "time-line" which specifies the natural order of flight and all that goes with it. This sped up the program for about 5 seconds, which still left the team with 2 seconds delay in the analyses of data supplied by the simulator. Since the program (luckily) does not concern a real airplane at this period in time, the team decided to discard the 2 seconds.
This is not to say that the end product will be "behind" in data for 2 seconds throughout the entirety of the flight, but means that when a lot of data is needed, for instance when the pilot is in transition between descend and landing, the program will slow down, to speed up again when you are in mid-flight or when not much data is needed.

Another selection made by the team was the use of data protocol. The TELNET protocol was preferred above the HTTP protocol, simply because it was faster. The TELNET protocol sends the data to the program in ASCII characters while HTTP does the same thing but in such a manner that it can be read by Internet Explorer or Netscape, which in turn means that entire sentences need to be sent to the receiving party. In future versions of the Flight Gear simulator the TELNET connection will undoubtedly be faster, since a few programmers have assured the team that that piece of software will be looked at in the near future.

(This page has been left blank intentionally)

Chapter 4: System requirements

4.1 Description

The following system requirements will cover all demands of the SAM program. SAM is the first concept to fully realize an Intelligent Cockpit Environment. In the following system requirements, all aspects will be covered according to subjects.

4.2 Subject

The Intelligent Cockpit Environment (ICE) project focuses on the use of new techniques and technology for human-machine interaction within the cockpit. A context aware system, like SAM, should monitor the actions of the pilot, the plane and the environment. This system should enhance the communication between the aircraft (machine) and the pilot (human), by giving the right information at a time the pilot needs or requests it. With the use of a context aware system it is also possible to take over some of the visual systems of the pilot, and to give him or her a warning in case of a defect, pilot error, or upcoming hazard.

4.3 Abbreviations

In the following system requirements some abbreviations are used. The following summation will cover those abbreviations:

I.C.E	Intelligent Cockpit Environment
J.D.K.	Java Developer Kit
J.E.S.S	Java Expert System Shell
S.A.	System Assessment
S.A.M	Situation Awareness Module
V.M.	Virtual Machine
X.M.L.	eXtensible Markup Language

4.4 General description

SAM is a prototype system that assesses the situation of the pilot, using real-time techniques. The system will be built-up out of several modules, to meet up with the specific demands of the user. Using this modular built-up of the program, it is easy to change and enhance the program, without having to re-compile the whole source code.

The context and situation awareness of the system will be realized with the use of JESS. This expert system uses 18 different algorithms after context rules are provided to come to a conclusion. In contrary to "normal" expert and reasoning systems, which works with the Boolean principle (YES-NO), JESS can also work with variables, which comply for a percentage to the rule, without being dismissed.

With this "fuzzy logic" technique it is possible to make a first prototype, which analyzes and anticipates to human behavior.
This is specifically of interest in a situation where a pilot has a flight plan, which states that normal cruise altitude will be 4500 feet. In real life, 4300 and 4700 feet are also acceptable, but when working with the Boolean principle ONLY 4500 feet is the correct altitude.

4.5 Context of the product

SAM is an operating system independent program. The program needs to run on every operating system, as long as it has a Java Virtual Machine installed on it. In order to let the program work as specified in these requirements, at least version 1.41 of the V.M. needs to be installed, or JDK 2.0.

4.6 Functions

The main function demand is that all application software runs smoothly and stable, and that all the modules can communicate with one another. The main application needs to obtain its data in the shortest amount of time as possible, and analyze it in real-time to generate a conclusion.

4.7 Users

SAM only knows one type of user so far. This user group has all rights within SAM. It can manipulate the interface, and load modules. Any user restrictions are not implemented in SAM at this time.

4.8 General limitations

Even though SAM must be able to run on every operating system, it is bound to some limitations. SAM will only run on a computer if the user has installed a Java Virtual Machine, version 1.41 at least. For real-time data analysis SAM depends on the input rate of the data. If the data stream fluctuates, real-time analysis is no longer possible. I.e. SAM can only perform a static analysis on the supplied data if this happens.

The level of advice and conclusion the system generates, depends on the quality of the knowledge loaded into the program. SAM uses this knowledge to make rules for JESS, which then evaluates those rules. If this data is inaccurate or incomplete, the forthcoming advice or conclusion is also inaccurate or incomplete.

4.9 Descriptions of the product

The SAM project has to meet up to some specific demands. These demands are categorized in the modules of the project.

SAM Modeler

The modeler program of the SAM project visualizes the knowledge inputted in the modeler program. It has to have the ability to load, save, edit and visualize the knowledge. The SAM modeler should have the following abilities:

- Load Context data
- Save Context data
- View Context diagram
- Parse XML file
- View variables
- Clear memory
- Exit program
- Context Editor

Load Context data

Using the mouse pointer to open up the file menu, the user sees the "Load Context" submenu. Clicking on this item will open up a file browser in which the user must input a file containing Context data. The file inputted by the user must be a gzipped binary file, containing user data. The program then checks if the file complies to the specified layout, and generates an error message if it does not. If the inputted file is of the correct format, the program extracts the context data out of the file, and passes the variables to the main program.

Save Context data

Using the mouse pointer to open up the file menu, the user sees the "Save Context" submenu. Clicking on this item will open up a file browser in which the user can save a file containing Context data. The user will enter a filename, and thus the program will generate a gzipped binary file, and saves it to the inputted file location.

View diagram

Clicking on this item will open up a window which visualizes the context previously inputted into the program. The context visualized in this function displays a set of inputted situations. Moving over such a situation will display its individual steps and its possible new situations that follow this situation.
Displayed without moving the mouse are several lines that help the user determine which logical new situation is run through at the end of any other situation.

Parse XML file

Clicking on this item will open up a new window, which displays the rules that are run trough in a fixed XML file, called KB_Cessna.xml. This XML file uses a separate file called variables.dtd, which incorporates the values for each variable that is in the XML file. At the same time the XML file is parsed the main window of the modeler displays what situations are being created and how many individual steps each situation has.
The XML file is parsed on a line-by-line bases. As each line is parsed the modeler translates it into a language that the modeler and therefore the client understands. For example:
The XML file contains the following line:
<action name="brakes" priority="1" probability="VSP">&ON;</action>
The Modeler transforms this line into:
Add step [parking brakes] on situation [Startup] where [controls/parking-brake==1]

More about the language used by the Data Modeler can be found in the section "**Context Editor**" further on in the chapter.

```
<situation name="Pre startup" timewindow="20">
 <phase name="prestartup">
  <action name="parking brakes" priority="1" probability="MP">&ON;</action>
  <action name="throttle" priority="1" probability="VSP">&IDLE;</action>
  <action name="ignition switch" priority="1" probability="BP">&OFF;</action>
  <action name="avionics power" priority="1" probability="BP">&OFF;</action>
  <action name="master switch" priority="1" probability="BP">&ON;</action>
  <action name="pitot heat" priority="1" probability="BP">&ON;</action>
  <action name="avionics master" priority="1" probability="BP">&OFF;</action>
  <action name="static pressure" priority="1" probability="BP">&OFF;</action>
  <action name="fuel selector" priority="1" probability="BP">&BOTH;</action>
  <action name="flaps" priority="1" probability="BP">&FULL;</action>
  <action name="pitot heat" priority="1" probability="BP">&OFF;</action>
  <action name="master switch" priority="1" probability="BP">&OFF;</action>
  <action name="fuel shutoff" priority="1" probability="BP">&ON;</action>
  <action name="lights" priority="1" probability="BP">&OFF;</action>
  <action name="beacon" priority="1" probability="BP">&OFF;</action>
  <action name="strobes" priority="1" probability="BP">&OFF;</action>
  <action name="nav. lights" priority="1" probability="BP">&OFF;</action>
  <action name="trim" priority="1" probability="BP">&SET;</action>
 </phase>
</situation>
```

Figure 10 Snippet from KB_Cessna.xml.

View variables

Clicking on this item displays the variables that need to be set in the server program. Without these variables, or parameters if you will, the client cannot get the necessary variables from the server to make an accurate guesstimate of the situation.

Clear memory

Clicking on this item clears the memory of the program so that a new context can be entered or loaded. In essence clicking this item resets the program back to its run-state.

Exit Program

Clicking on this item exits the program.

Context Editor

Besides the functions that have just been described there is one other function. This function allows you to add, edit, and delete rules of an empty or already existing context. This is done in a simple "language" to make it easy for the user to understand and use.
The commands used in this language are:

new situation [situation name]
This command creates a new situation in the memory.

add step [step name] when [boolean expression] on [situation name]
This command creates a new step for a given situation. The Boolean expression is something in the order of [controls/parking-brake == 1]

change step [step name] when [boolean expresion] on [situation name]
This command makes changes to the already existing step in a particular situation.

add direction [direction name] when [boolean expresion] on [situation name]
This command adds a direction (such as [Take-off]) on a particular situation when a Boolean expression has been reached.

show situation [situation name]
This command displays an already existing situation.

show step [step name] on [situation name]
This command displays a step with name [step name] in an already existing situation.

show direction [direction name] on [situation name]
This command displays an already existing direction from an existing situation.

show all
This command displays everything that is in memory.

clear memory
This command deletes all rules from memory.

delete step [step name] on [situation name]
This command deletes a step with name [step name] from a specific situation.

delete direction [direction name] on [situation name]
This command deletes a direction from a specific situation

delete situation [situation name]
This command deletes a situation.

SAM Data server

The SAM Data Server should gather the data from the flight simulator, store it into memory and send it to any number of connections made by any or one Data Client. Note however that at this point in time, the Data Server has to be given a list of parameters of flight values to get from the flight simulator in order to provide data to the client. This is because of the enormous amount of time it would take for the Data Server to gather all parameters from the simulator.
The SAM Data server should have the following abilities:

- Parameters
- Connect
- Disconnect & Exit

Parameters

Clicking this item lists a set of fixed built-in parameters which needs to be selected in order to provide the variables for the SAM Client. Without these parameters set, the program will produce an error-message.

Connect

Clicking on this item opens a TELNET connection to the Flight Gear Simulator and extracts all the inputted variables from the Simulator. The IP address and the port number of the Simulator can and must be entered in two textboxes. If these textboxes are not correctly entered or when no Simulator has been found on the provided address, the program will produce an error-message.
After a successful connection has been made, the parameters button will be disabled since there is no more need for it now, and the Server will start polling the Simulator at an interval set at 600 milliseconds. Any SAM Client that is connected to the SAM Data Server can change this interval.
Also, after a successful connection has been made, the Server will be able to start accepting connections from clients. Every xxx milliseconds (400 if it's not altered by any client) the Data server requests the list of parameters from the Flight Gear Simulator and then stores them in memory.

Disconnect & Exit

Clicking this item will disconnect the Data Server from all clients and the Flight Gear Simulator and then exits the program.

SAM Data Client

The Data Client should do all of the actual processing and determination of the situations according to the variables that are provided by the Data Server.
The Data Client should have the following abilities:

- Load Context
- Connect
- Blackboard

Load Context
The Load Context function opens a file dialog where one can select a context file that is to be used in concordance with the Flight Gear Simulator. The context file is the file that was saved by the modeler and thus is a gzipped binary file. If some variables are missing, i.e. not selected on the server before connecting to the Flight Gear Simulator an error message is displayed on the main window along with the variables that need to be added. If no connection has been made to the Data Server another error message is displayed stating that a connection to the Data Server has to be made.

Connect
The Connect function uses the IP Server textbox as reference for its network connection and the Frequency textbox for the interval at which the Data Server is supposed to be polling the Flight Gear Simulator for its variables.
While connected the Frequency can be changed at any time by simply entering a number and pressing the ENTER button on the users' keyboard.
After the IP address for the Data Server has been checked, the Data Client tries to open a network connection to the Data Server and requests the variables that the Data Server has available.
Once connected, the Data Client first requests all the variables names and then infinitely requests the individual values of each variable. This is done because there is no need to keep repeating the variable names over the network again and again which would slow down a network and the program because it would constantly be busy allocating memory for the variables.
After all values of the list of variables have been obtained, the program passes the values to certain JESS rules and uses the results that JESS returns to draw a conclusion.

Blackboard

The Blackboard function opens up a window, which displays the values that have been provided by the Data Server. This is done in a table where the x-axis represents the variable names and the y-axis the amount of steps that have been run through.
Either clicking the Close button or clicking the 'x' at the top-right position closes this window.

4.10 Functional demands of the product

The following demands need to be met by SAM. These demands are limited to the functionality that SAM needs to have towards the user:

- The user must be able to get a advice from the system
- The user must be able to ask for an advice to the system
- The user must be able to enter knowledge into the system
- The user must be able to export knowledge from the system
- The system must be able to obtain data from the Flight Gear Simulator
- The Flight Gear Simulator must be able to give data to the system

4.11 Properties of the external connections

4.11.1 Users dialog

SAM

SAM must be presented with a clear and obvious menu structure. The program itself needs to consist of three separate modules, which can communicate with one another, after user input. The modeler needs to visualize the knowledge that is loaded in from an XML file. This knowledge needs to be easily manageable with use of graphical tools. All data must have the ability to be saved.

SAM Client

The SAM client needs to give a clear overview of the situation the pilot is in. The display and refresh ratio of the situation needs to be as good as real-time, with a maximal time-lapse of 1 second. The knowledge needed for the analysis needs to be loaded in, by opening the menu. The user dialog needs to be built out of four separate boxes with additional information about the situation the pilot is in. This additional information can be the following:

- Used variables
- Conditions that are met
- Predicted next possible situations
- Input for the address of the Data Server
- Frequency rate the data is refreshed

SAM Modeler

The primary task of the SAM modeler is to visualize the knowledge, inputted by the user, or loaded in from an XML file. With the means of an input box, the user can manipulate the knowledge, or if he or she wants to, enter new knowledge. Through a diagram button, the knowledge will be visualized into a clear diagram, which will display situations, steps and possible following situations.

SAM Data Server

The Data Server module provides an interface in which the user must enter the address and port number of the Flight Gear Simulator. The user can select variables trough a dialog, which will be requested from the Simulator and in turn are sent to the SAM client. The interface itself displays the parsed variables, the address of the Flight Gear simulator, and a Connect and, after a connection has been made, a Disconnect button.

4.11.2 Apparatus connections

If the user wishes to run a Flight Gear Simulator on a computer other than on the computer the Data Server is running on, or when the Data Server runs on another Computer than the Data Client, it is necessary to have a network connection available.

4.11.3 Program connections

For the use of the SAM, the following other programs are needed:

- Operating system
- Java Virtual Machine or JDK1.2.4
- Flight Gear

Any other software is not needed.

4.11.4 Communication connections

To use the SAM program, a computer needs to be equipped with a working TCP/IP stack, even when all programs are run on one computer.

4.11.5 Presentation demands

SAM needs to have a clear and attractive interface, which visualizes the most important data in one blink of an eye. By simple steps, the interface needs to be able to be manipulated, so that the user can choose what kind of data he or she wants to see.

4.11.6 Design limitations

The most important design limitation of the SAM project is time.

4.11.7 Apparatus limitations

SAM needs to be able to run on the computer of the user. The minimum requirements of SAM are the same as the requirements for the programs SAM needs to run, to make SAM able to work on a great variety of computers.

4.11.8 Quality criteria

SAM needs to be operating system independent. It should be able to work in any environment. The analysis of the data by SAM needs to be an accurate and actual representation of the reality.

4.11.9 Maintenance ability

SAM has to have a modular type of built-up, in order to be able to make changes or improvements to the program, without having to recompile the source code of the entire program.

4.11.10 Portability

SAM needs to be operating system independent, in order to provide portability to any computer system that meets up with the program requirements of SAM.

Chapter 5: System model

5.1 Description

In this chapter the system model of SAM will be described. The first section (5.2) will give an overview on how SAM operates. Section 5.3 will cover the functional specification of SAM. The next 4 sections will describe the techniques used with the creation of SAM.

5.2 System overview

The main goal of SAM is to give an accurate reflection on what situation the pilot is in. To achieve this goal, a lot of testing is necessary. After some first prototypes it was concluded to make a modular model. Using this way knowledge could easily be implemented in the program, and due the modular model, code changes can be easily integrated in the project.

5.3 Functional specification

5.3.1 In general

5.3.1.1 Introduction

The functional specification is part of the graduation project SAM for the Technical University of Deft, based on the assignment of Maikel van der Roest and Richard Harreman.

The project consists of the research into Situation Awareness, and the following implementation of a prototype. This prototype must provide a base into further research towards an Intelligent Cockpit Environment. This section of the thesis forms the functional basis of SAM and covers the automation configuration and the functional specification of the Situation Awareness Module.

5.3.1.2 Functions of SAM

The main task of SAM is that all application software runs stable, and can communicate with one another fast and accurate. Application software covers: SAM Data server, SAM Data modeler, SAM Data Client. A supplementary task is maintenance of the software and knowledge.

The SAM Data server provides a quick and stable flow of simulation data for the SAM Client. The Data server also provides the user with the possibility to buffer and safe data, so if the connection with the client is lost, the data is not lost.

The SAM Data Modeler gives the user the ability to load XML files which provide the program with a context to work with. This context will be visualized on a screen. The user has also the ability to make its own contexts in the modeler which can be saved to an XML file.

The SAM Data Client is the main application of the SAM project. It provides the main functionality. The client uses context models provided by the data modeler to analyze simulation data provided by the data server. The client analyzes the incoming data and provides the user with a conclusion and predicts what the upcoming situations are.

5.3.1.3 System choice

The development of SAM takes place on a Windows NT and 2000 platform. This choice was made by the project members based on the facilities of the Technical University of Delft. Using JAVA technologies enables the program to run at virtually any pc, providing that it has a JAVA Virtual Machine installed.

5.3.2 Automation configuration

5.3.2.1 Introduction

In order to reduce the workload of a pilot, it is necessary for the program to run "on it's own". This means that once the button is pressed for the situation analysis, the pilot only has to look on the screen and do nothing else. In order to achieve this, it is vital for the program to be self-sufficient in data gathering and reasoning.

5.3.2.2 System overview

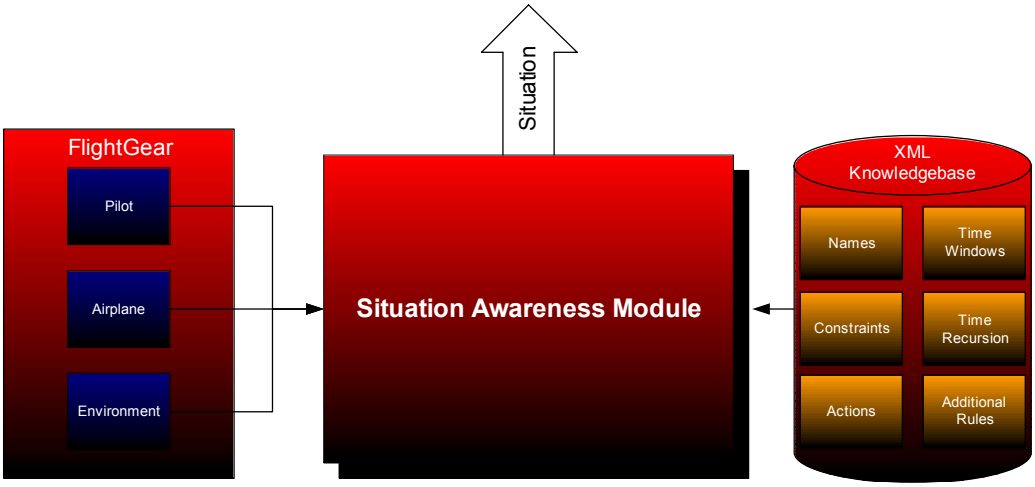


Figure 11 A global overview of SAM.

In the above shown picture there are basically three different sources. On the one hand there is the Flight Gear Simulator and on the other there is the XML knowledge base, which we will call the XML file from here on.

To make the SAM program easy accessible and easy to maintain, we split it into three parts. The figures below show a representation of the way the information is processed in the separate parts.

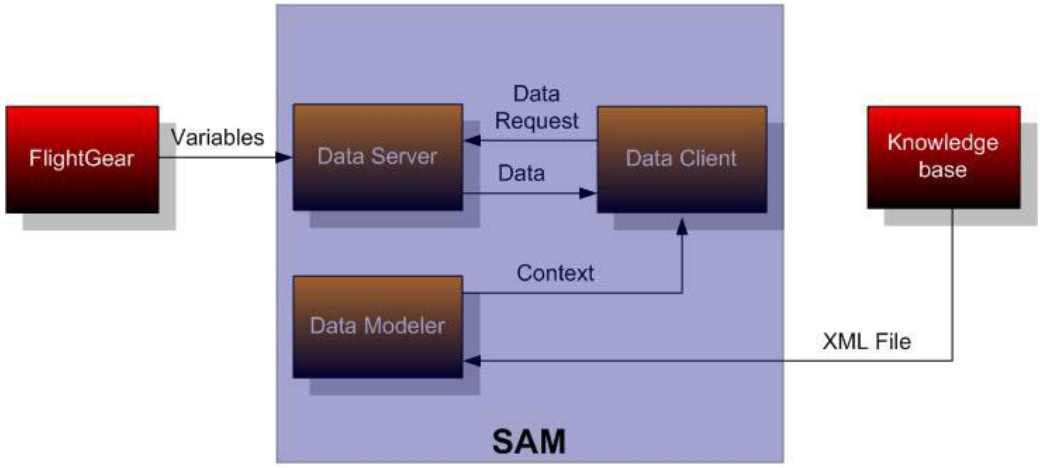


Figure 12 Modules within SAM.

The Data Modeler Parses the XML File to a context which can be written down and thereafter read by the Data Client. Once the XML File is parsed it can be viewed by the user on a subscreen. This subscreen displays the various situations, the links between the situations and the steps within each situation.
The variables that are needed from the Data Server can also be viewed on a separate subscreen. This is very handy, because it is necessary to select them in the Data Server.

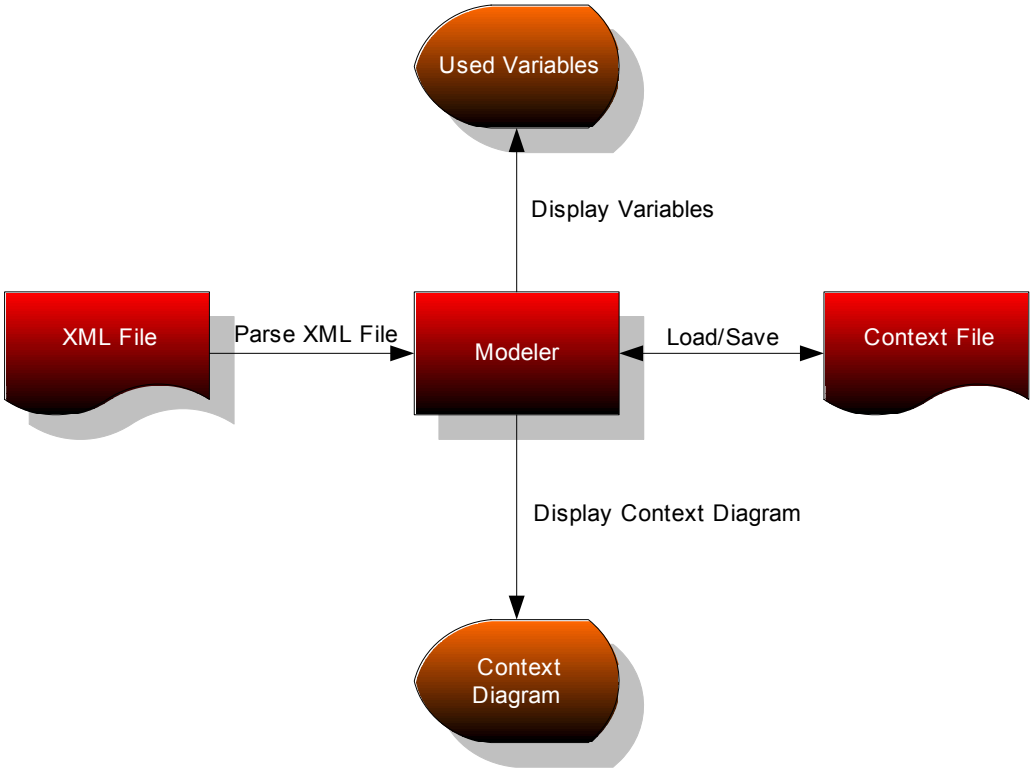


Figure 13 Flowchart of Data Modeler.

The Data Server gathers the variables selected in a separate subscreen from the Flight Gear Simulator and passes them on to the variable container in its memory. From there the Data Client gets the stored variables at a set interval. The variable container has been created to prevent data loss in the event of network collisions and temporary network severs.

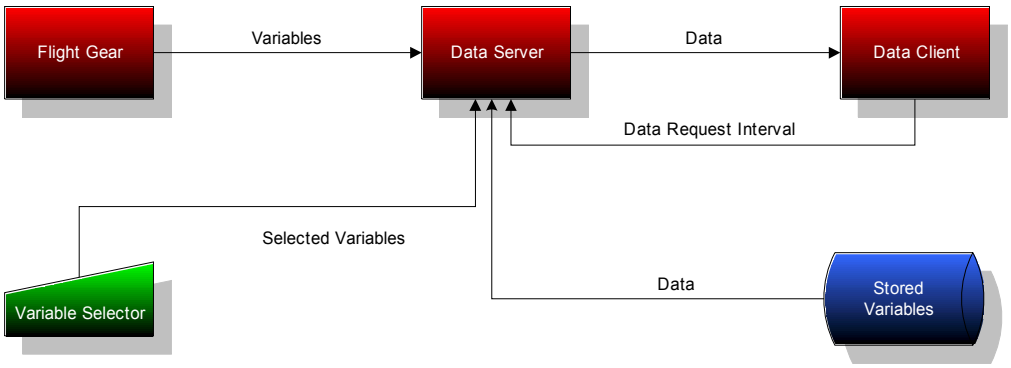


Figure 14 Flowchart of Data Server.

The Data client has a direct connection to the Data Server. The context needed for the situations has to be loaded from hard drive. The processed data can be viewed on a type of blackboard, which is a representation of variables stored in memory. The Data Client displays the current situation and the next possible situation.

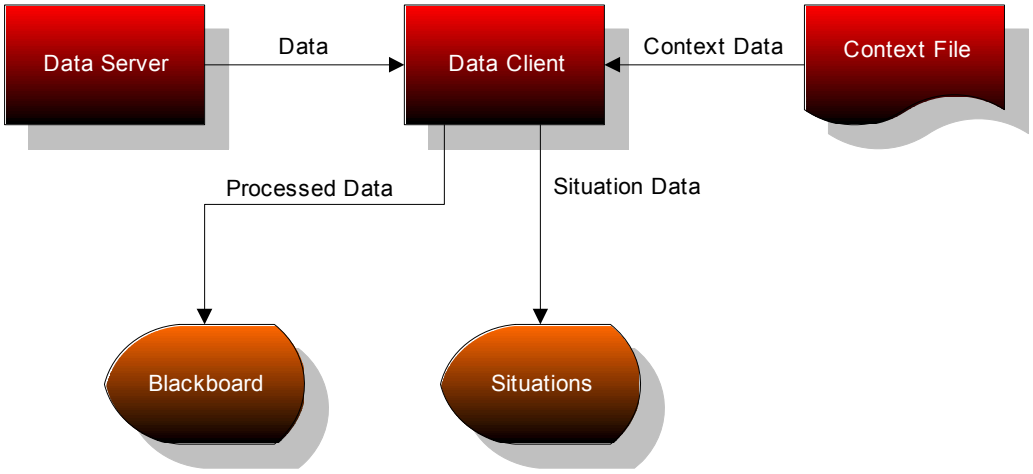


Figure 15 Flowchart of Data Client.

5.3.2.3 Yourdon method

The "Yourdon method" is a technique that has evolved over the last 20 years. Almost 200.000 people contributed to the method, making it very usable. Nowadays the "Yourdon method" consists of two things: tools and techniques. The "tools" are a variety of graphical diagrams, used to visualize and model the requirements and architecture of the system. The most commonly known of these diagrams is the Data Flow Diagram (DFD) as shown in figure 21.
Although the DFD is an excellent way to show what functions the software should carry out, it says little or nothing about the relationships between data, and time-

dependant behavior. Therefore the latest version of the "Yourdon method" also includes Entity Relationship Diagrams (ERD), State-Transition Diagrams(STD) and Structure Charts(SC). All these diagrams make it easy for future developers to further expand the system.
Although this modeling approach is quite old, and forms the basis for modern approaches, like UML, it still suffices for our project. Because of the lack of object orientated programming, and the basic simplicity of the model, the Yourdon method fulfilled our needs. This results in the following approach:

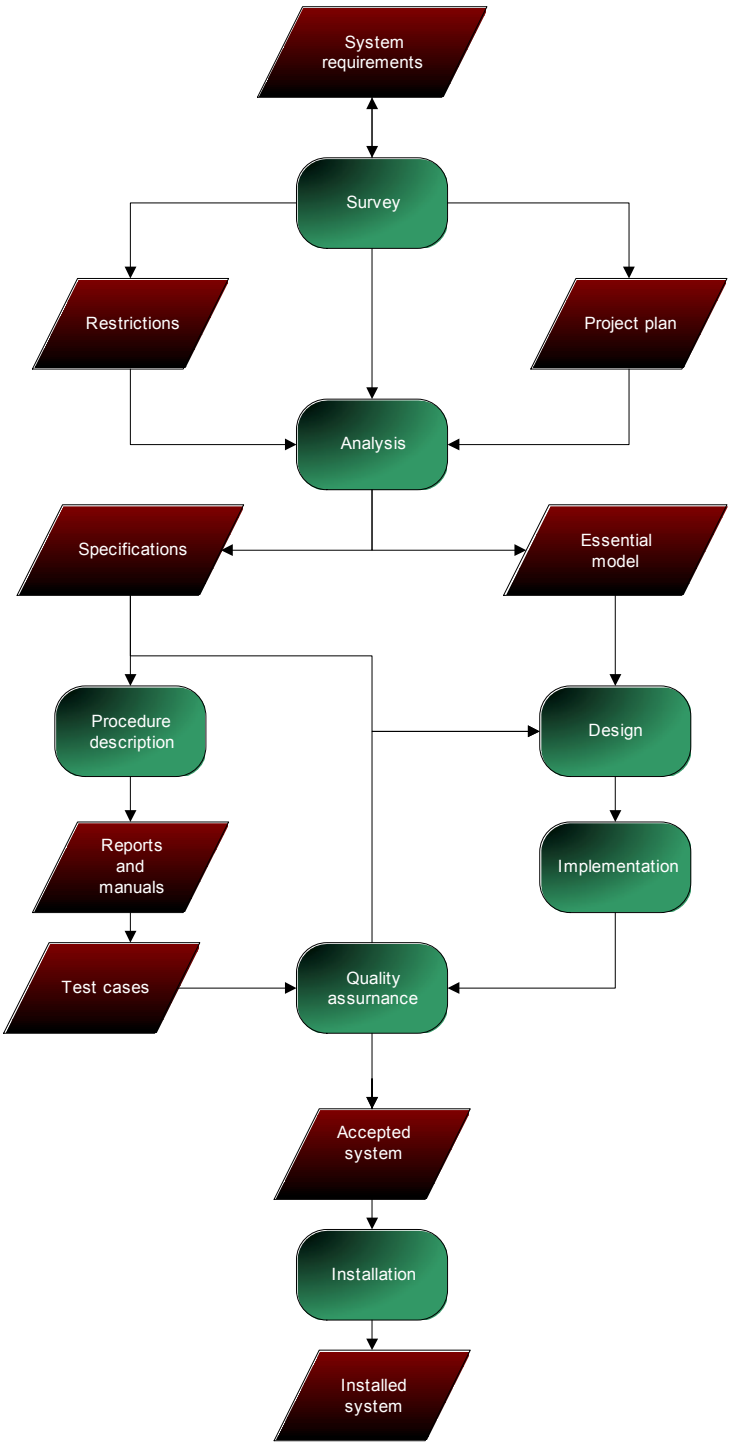


Figure 16 System approach.

5.3.3 Functional demands

5.3.3.1 Introduction

According to the Yourdon method of designing SAM has to meet up with some functional and non-functional demands. The following demands are deducted of the context diagram, found further on in this thesis.

5.3.3.2 Functional demands

SAM has to comply with the following demands. These demands limit themselves to the functionality SAM must offer to the user:

- The user must be able to get a system advice
- The user must be able to ask for a system advice
- The user must be able to input knowledge
- The user must be able to input a context
- The system must be able to obtain data from Flight Gear
- Flight Gear must be able to deliver data to SAM

5.3.3.3 Non-functional demands

SAM has to comply with the following demands. These demands limit themselves to the functionality the software must offer under certain circumstances:

- The host machine of SAM must be equipped with a JAVA virtual Machine
- The communication between SAM Data Client and SAM Data Server needs to be with the least latency as possible
- The communication between SAM Data Server and the Flight Gear simulator needs to be with the least latency as possible
- The user must have a system equipped with Flight Gear

5.3.4 Functional operation

5.3.4.1 Introduction

This section of the thesis will describe the functional operation of the program. The Graphical User Interface (GUI) is described, as well as the users known to SAM.

5.3.4.2System Functions

The Data Server

The network connection between the server and the Simulator is established by use of TELNET protocol, over TCP/IP. This means that the Data Server is making

requests that are translated to TELNET commands, these commands are sent through the outcome flow and finally the Flight Gear Simulator will convert them to internal commands.
The specified parameters are checked and the values are sent back to the server through the outcome flow of the Simulator, the equivalent of the income flow of the Data Server.

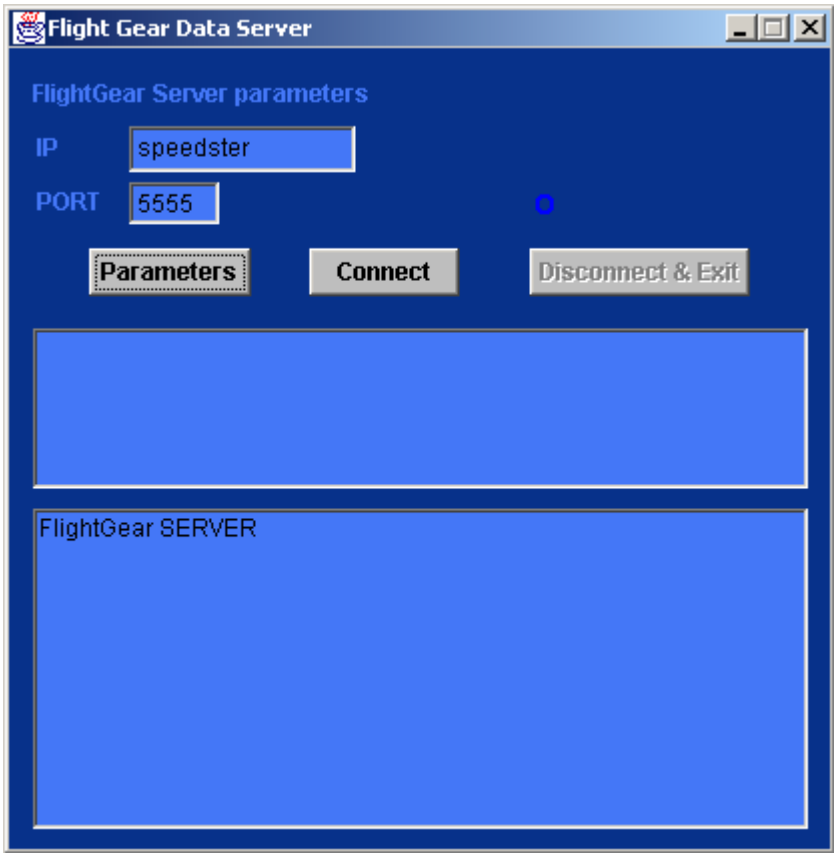


Figure 17 Data Server.

The Data Server uses TCP/IP to connect to the Flight Gear simulator by connecting to port 23 (TELNET). Specifications on how the TELNET protocol works can be found in RFC854 document: http://www.faqs.org/rfcs/rfc854.html
This Server can be started on any PC in a local area network since local area networks usually provide fast enough connection speeds.

Several options can be set once the Server is started. First, the user has to click the button named "Parameters" to get a list of parameters that are pre-programmed into the Server.

Next, the user has to select several parameters that are going to be needed. This is done because if you want to use all parameters as a default, the telnet connection with the Flight Gear Simulator will slow-down considerably as to a speed that will update all parameters every 40 seconds. The fault for this lies in the Simulator program itself. There are no limitations on the amount of data transferred between the Data Server and Client.

After the above two steps the user has to give in the ip-address or the Domain Name of the PC the simulator is running on. After that, the button "Connect" may be clicked and the Server will connect to the simulator and retrieve data from the simulator and handle the incoming connections.

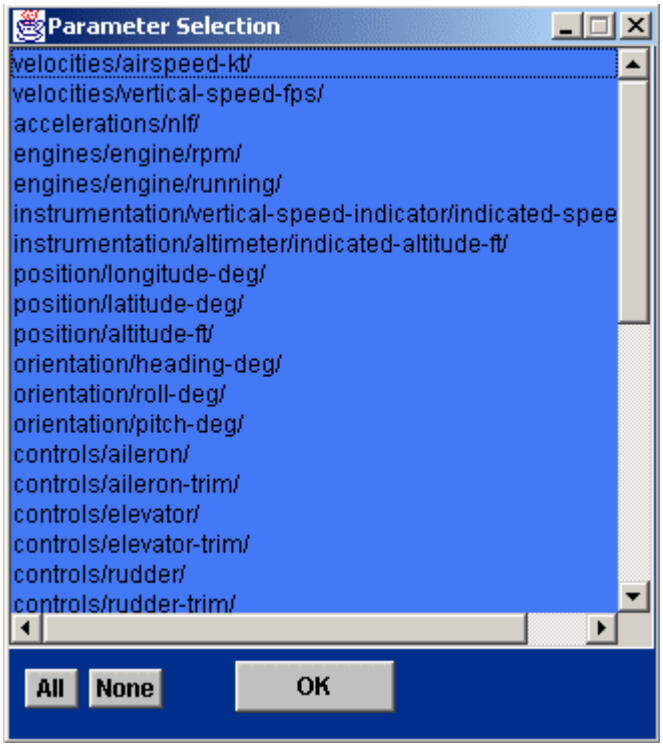


Figure 18 Parameter selection in Data server.

The Data Client

The Data Client connects to the Data Server to get the values for the parameters every few seconds or milliseconds. This depends on what great an interval was set in the frequency box.

The Client needs to load a context file written on disk by the Data Modeler. This is simply done by clicking the "Load Context" button. If a wrong context file was given or when the file is corrupt, nothing will happen.

After that and after the connection with the Data Server is made, the Client will run through all possible situations and will compare them internally with the provided parameters as they are coming in. From that a possible situation is given as to in what stage of the flight the plane is in. i.e.: The plane is taking off or is in mid-flight.

The communication between the Data Server and Client Server relies on a non-standard protocol. The communication is done by a stable TCP/IP connection. Both the Data Server and the Client Server have two kinds of data streams, namely an incoming and an outgoing stream. In this case the communication is based on package sending and receiving. So if the Data Server has some new information, it builds a package and sends it to a connected client which will use the data from there on. A package that is send (or received) contains a code that is a byte value and the useful data.

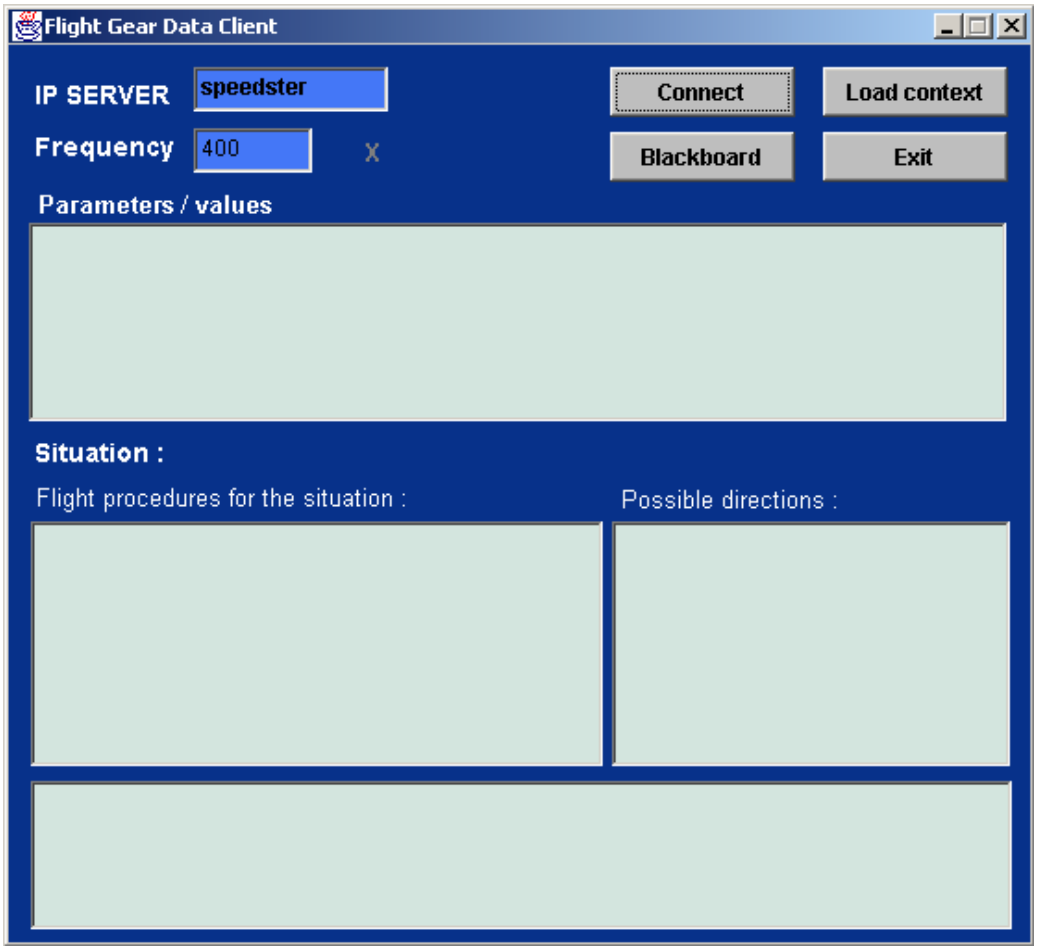


Figure 19 Data Client.

The Data Modeler

The Data Modeler, as noted in a previous chapter, parses a XML file in order to create some rules that will be written to hard drive as a file.

These rules or definitions if you will, are supposed to resemble as closely as possible the reality. The main advantage of such a system is that there will have to be made almost no modifications to the source code of the program.

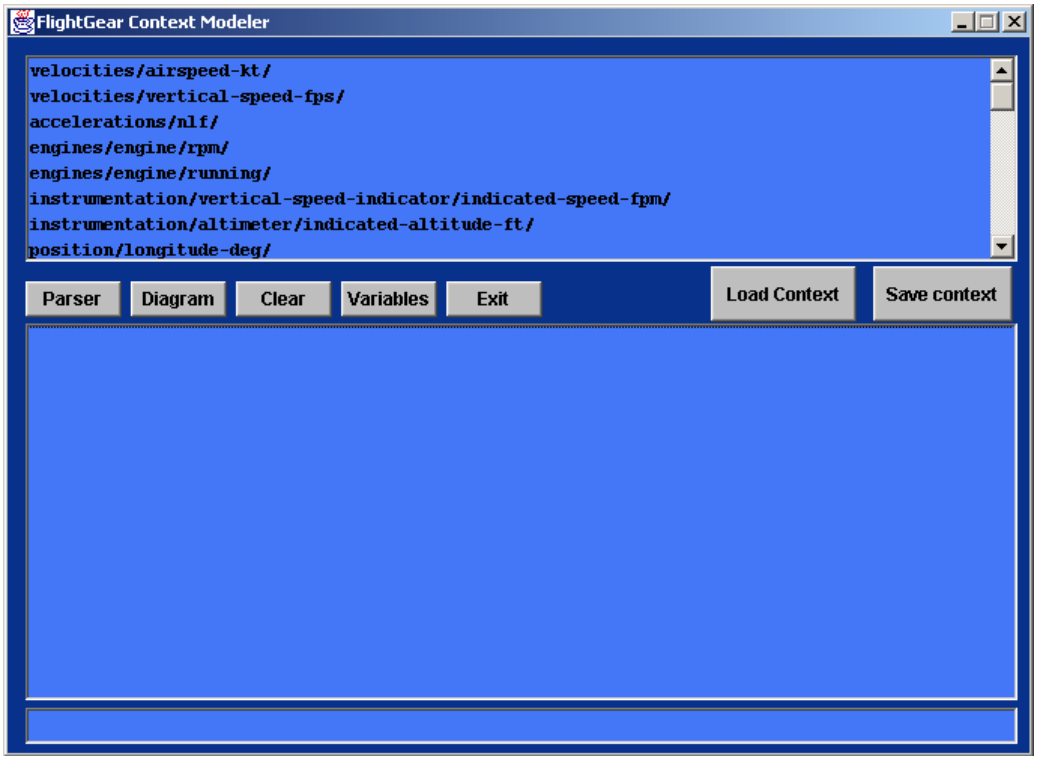


Figure 20 Data modeler.

For future advances it would be nice to make a Data Modeler which not only parses a given XML file to rules, but make a XML file based on rules given in by the user.

An addition that is already implemented in the Data Modeler is the creation of a context diagram, which makes it easier for a user to understand the data that is processed from the given XML file.

Figure 21 Context Diagram within Data modeler.

5.4 System analysis

5.4.1 Introduction

This section will show the various diagrams involved in the development of SAM.

5.4.2 Context diagram

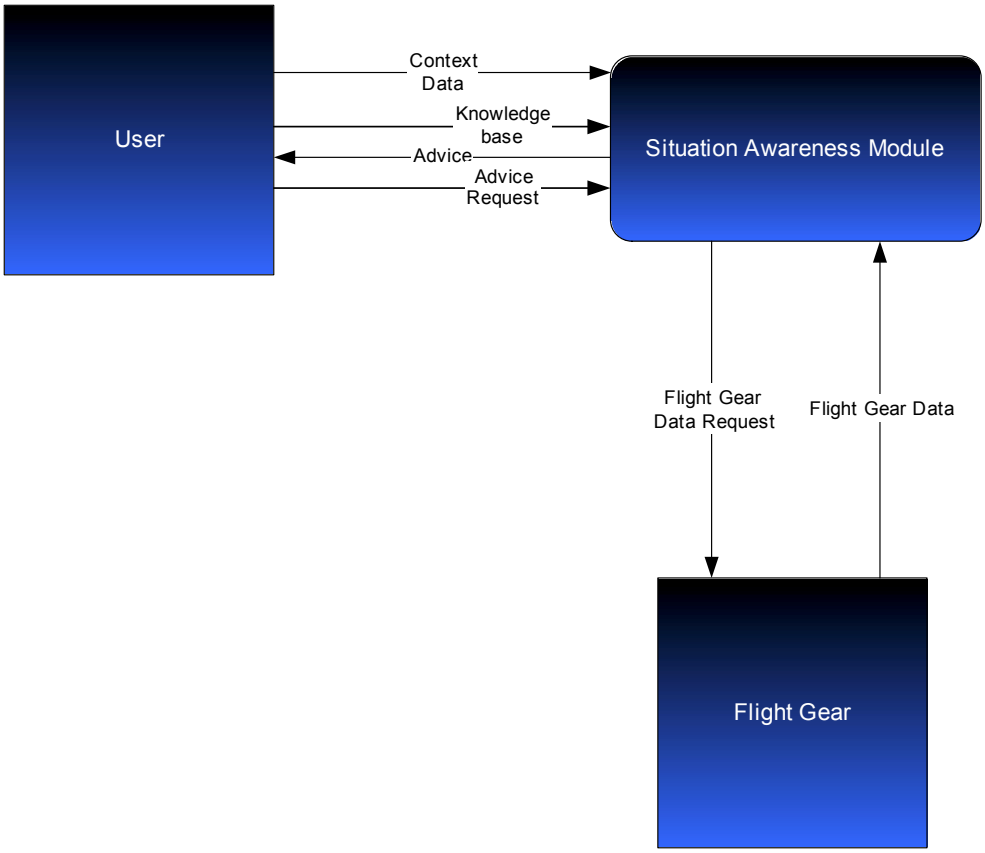


Figure 22 Context diagram of SAM.

5.4.3 Data flow diagrams

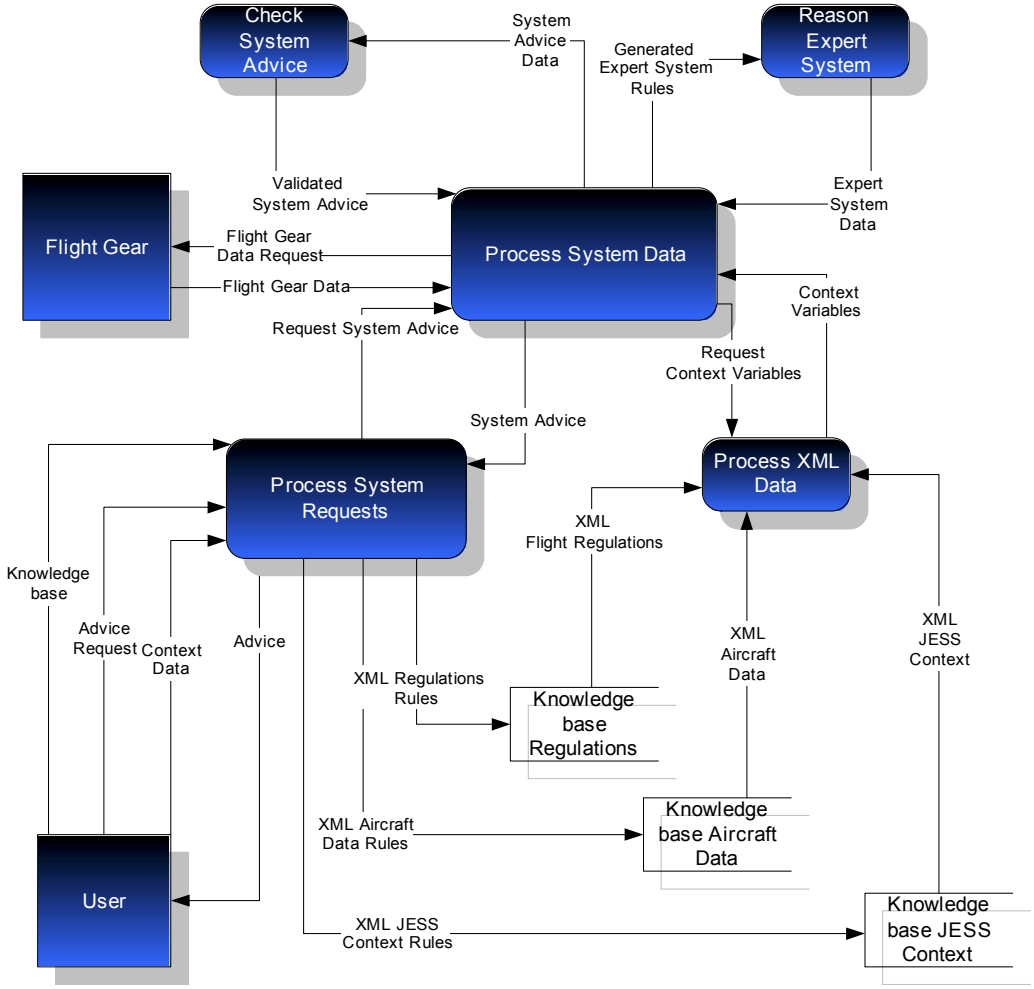


Figure 23 SAM DFD layer 1.

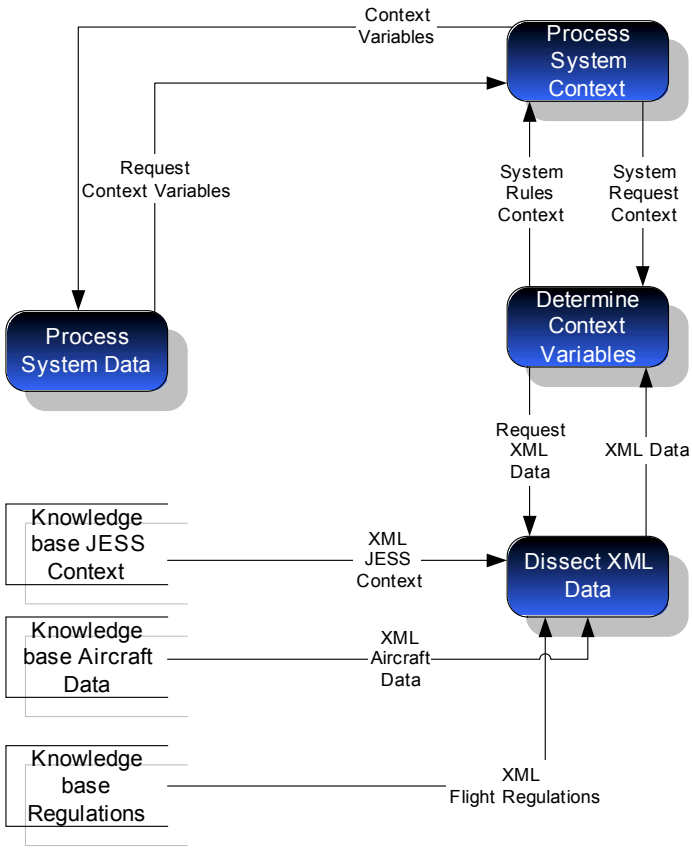


Figure 24 SAM DFD layer 2 XML data.

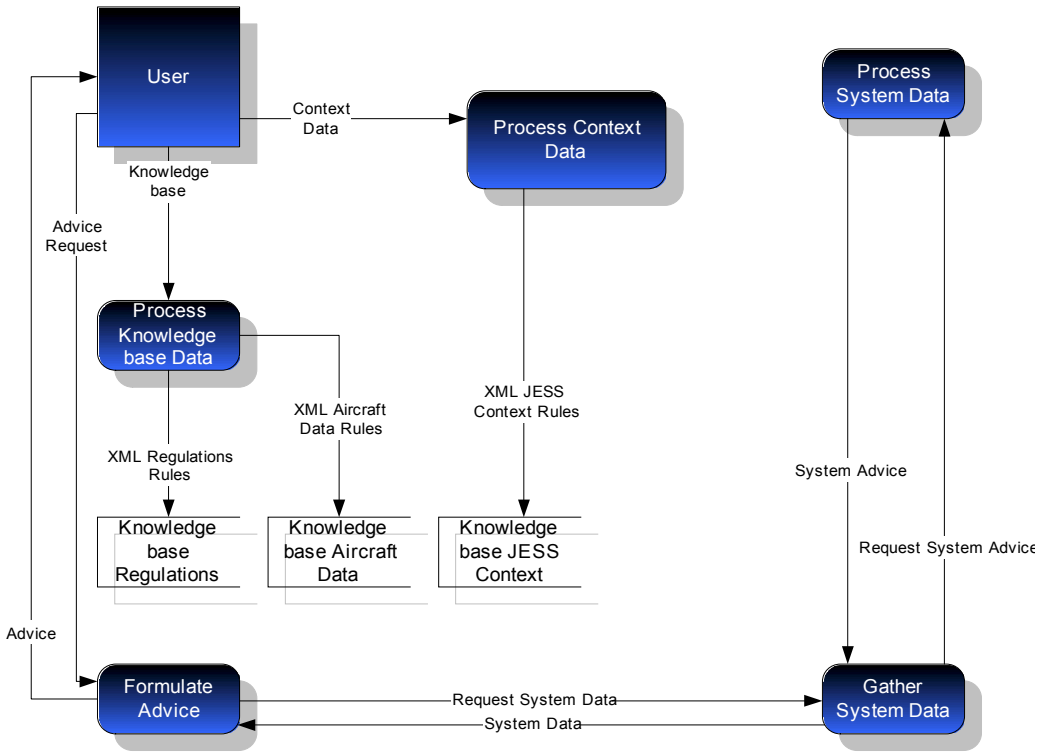


Figure 25 SAM DFD layer 2 process system requests.

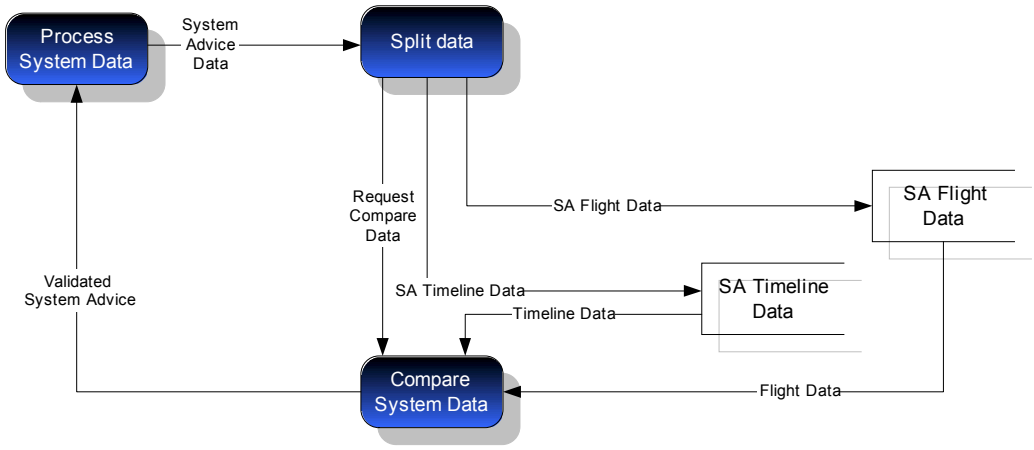


Figure 26 SAM DFD layer 2 check system advice.

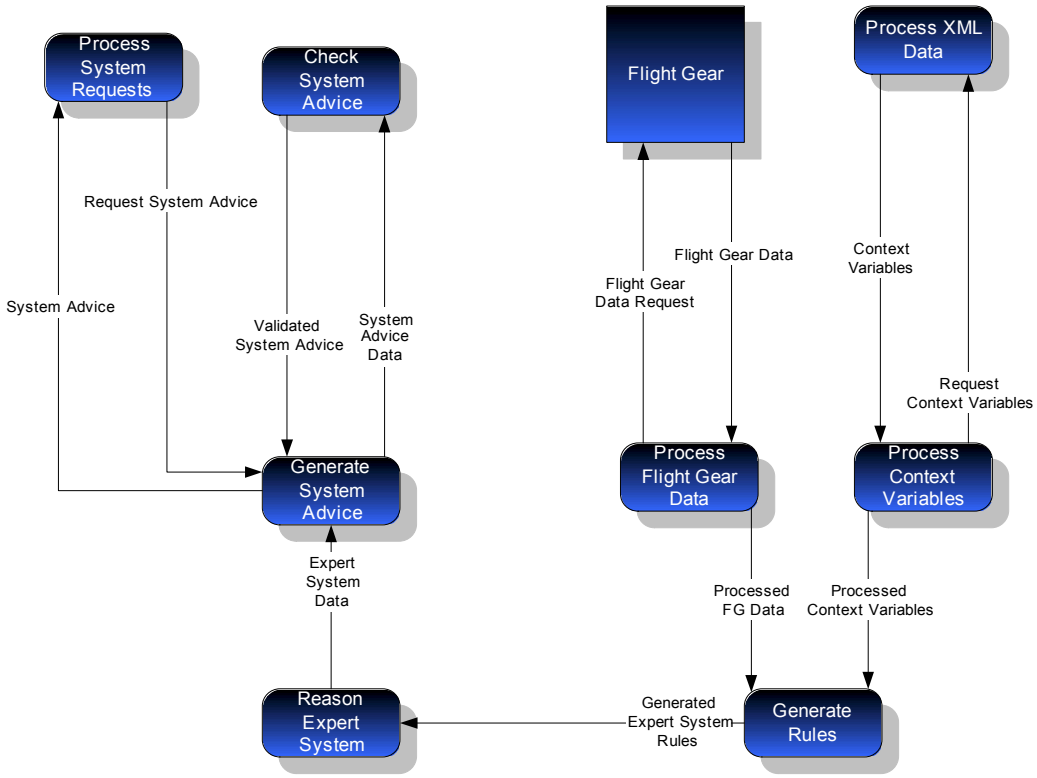


Figure 27 SAM DFD layer 2 process system data.

5.4.4 Entity-relationship diagram

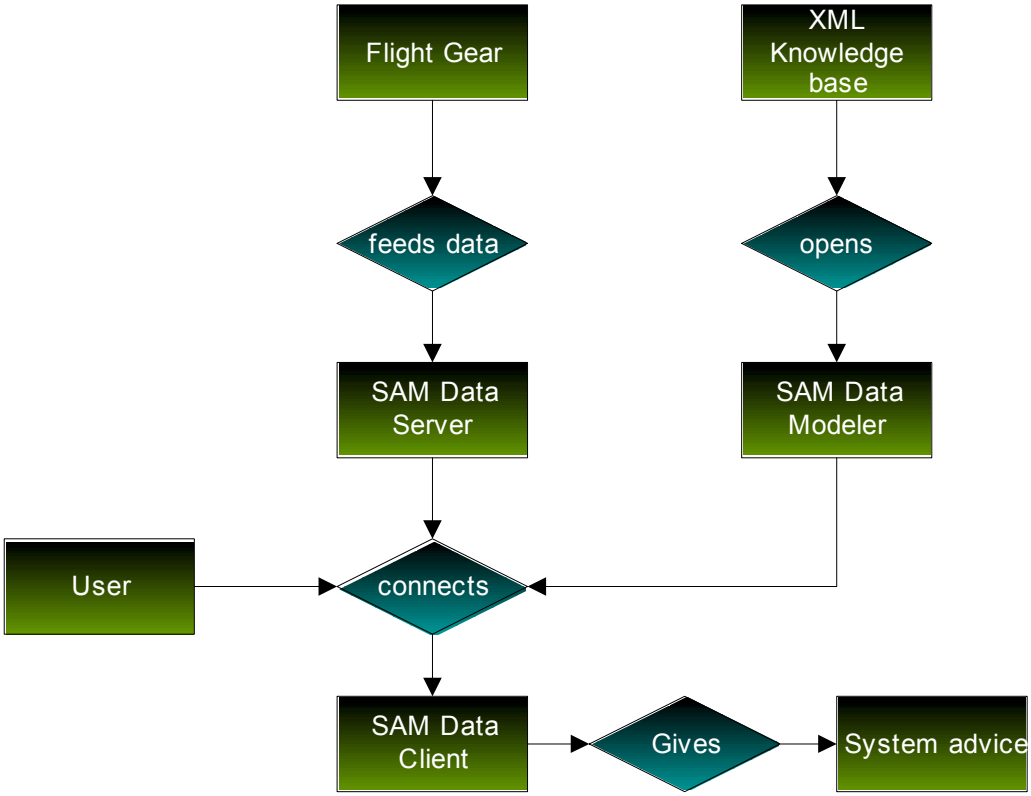


Figure 28 ERD diagram of SAM.

5.4.5 State transition diagrams

5.4.5.1 Data Modeler

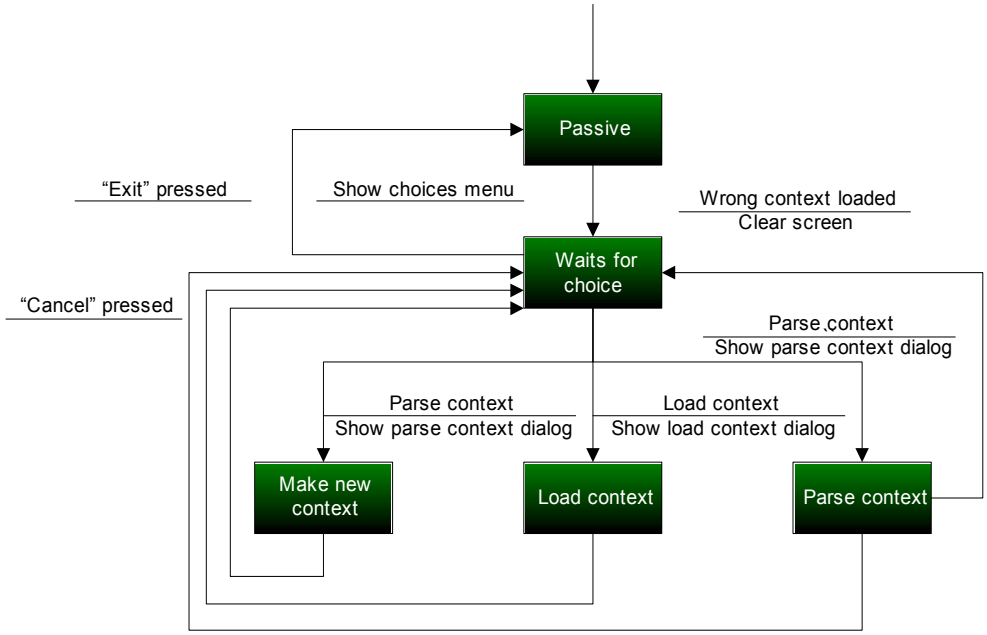


Figure 29 The main Data Modeler STD.

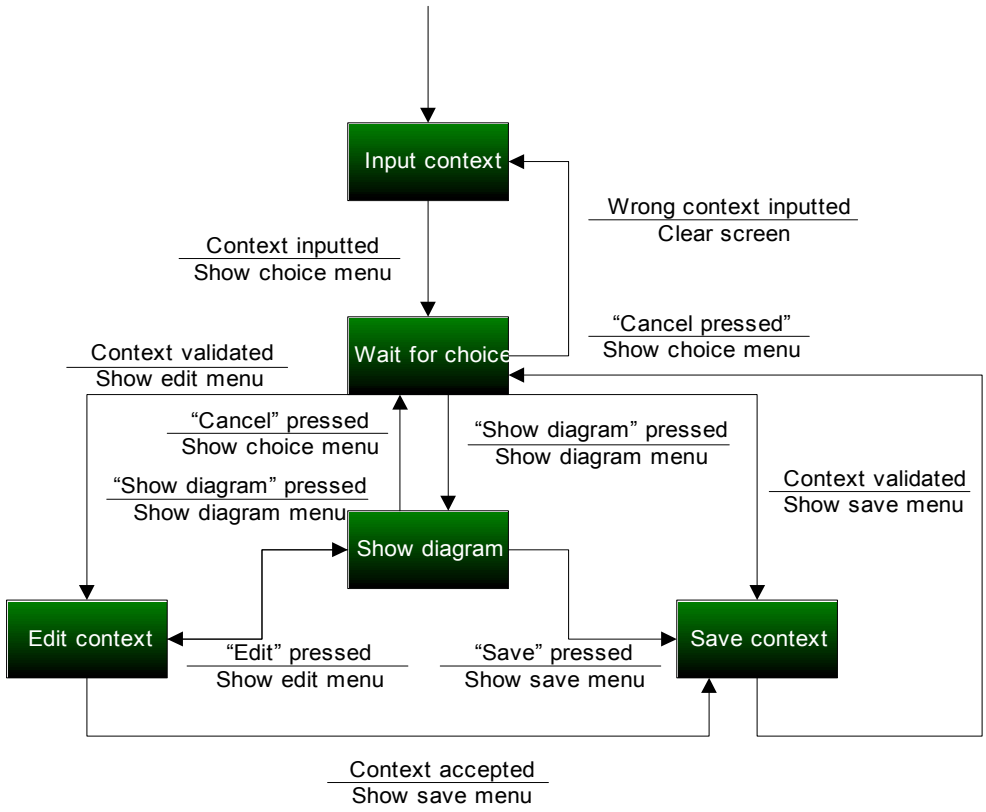


Figure 30 Make context STD.

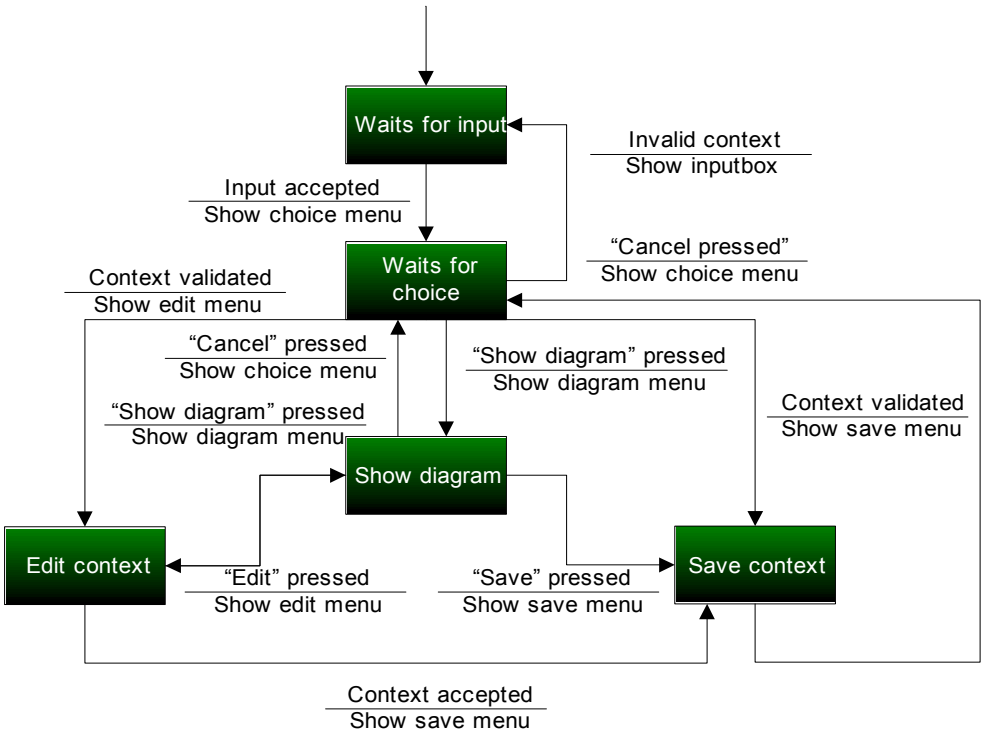


Figure 31 Load context STD / Parse context STD.

5.4.5.2 Data Server

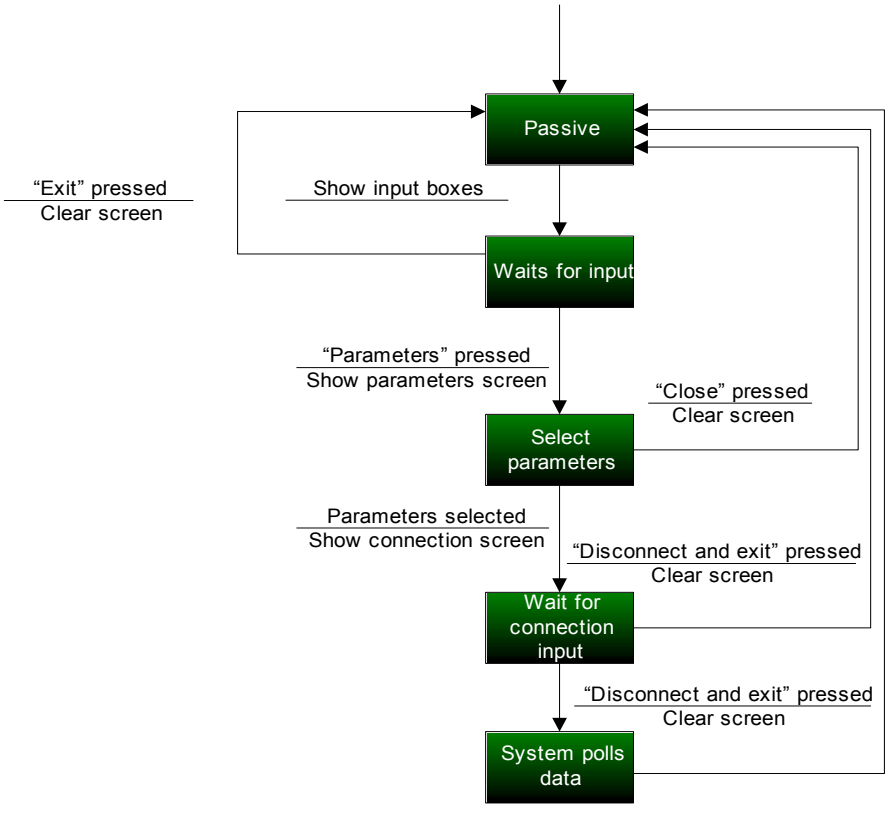


Figure 32 Data Server STD.

5.4.5.3 Data Client

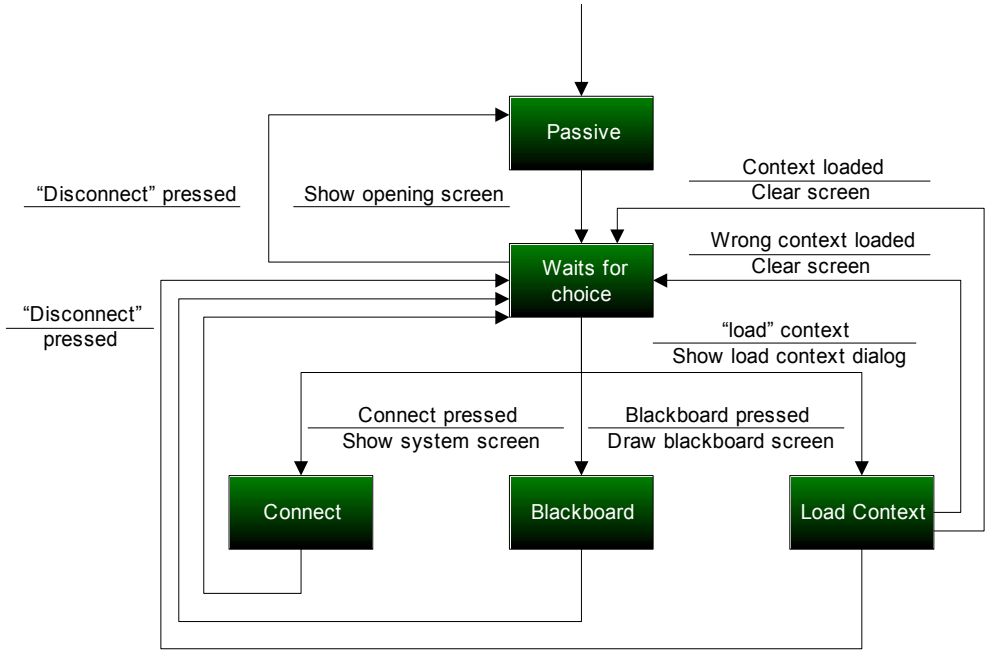


Figure 33 Data Client STD.

(This page has been left blank intentionally)

Chapter 6: Artificial Intelligence

6.1 Description

In this chapter the artificial intelligence and the knowledge base will be described. The first section (6.2) will be about the use and background of the knowledge base. Section (6.3) will be about expert systems and the use of tools thereof.

6.2 Knowledge base

6.2.1 Description

Knowledge bases are used as a means to quickly and at an interpretable level show what information is available for, in this instance, an airplane.
The knowledge about how to fly an F-16 has been gathered and stored in a knowledge base in a preliminary research project by Quint Mouthaan. This inspired us to adapt this knowledge to a Cessna type airplane and use it in our modular program. The advantage of such a knowledge base is that one does not have to re-program and re-code the entire program in order to adapt it to his or her specific plane to fly in a flight simulator. One simply adapts the XML file in which the knowledge is stored and run it through the parser in the Data Modeler.

The contents of the knowledge base will be used to determine which situation is most likely to occur at a certain moment in time. The knowledge base has been stored as an XML file, which is displayed in Appendix B.

6.2.2 Layout

The knowledge base is divided in a number of situations that we want to be able to recognize, for example, taking off and landing the plane. These situations are described in section 6.2.6. For every situation there is a set of rules that state the probability of the situation dependant on the state of the aircraft or the events that are happening.
An event can have three sources:

Pilot: Pilot events are actions the pilot is taking, for example a button that he presses or the yoke that is adjusted.

Aircraft: Aircraft events are changes in the aircraft's state. For example a change in altitude or speed.

Environment: An event from the environment can be another aircraft that is on a collision course towards the pilot's plane or at a low altitude, a tree. For this source it is necessary to monitor the plane's outer range for a set distance.

Pilot and environment events are usually related to variables that have discrete values whose changes will likely be important. Therefore these values should be monitored constantly so that changes will be detected immediately. Aircraft events are usually related to variables with continuous values that change often and gradually. These continuous values should be sampled at regular intervals.

Next to the events there is another source of information that can be used to determine the current situation. This source is the flight plan. The flight plan contains information about the steering points the pilot will fly to during the flight, just like a normal flight plan, but it also contains information about the situations that will occur at those steering points. This enables the program to predict what situation should occur at what time.

6.2.3 Rules

As stated before, rules are grouped according to the situation they relate to. Every rule will have a value that states the probability of a situation when that particular rule is fired. More on the subject of probability will be discussed in section 6.2.4. For every situation there are several kinds of rules:

Action rules:
An action rule is a rule that states that a pilot has to or might perform a certain action during that situation. An action from a pilot always has effect on a control or instrument of the airplane. For example, moving the stick to the front will result in the elevator position becoming negative. An action rule is therefore a rule that states with how much probability a particular situation is taking place if at some point in the situation a control or instrument is set to a specific value. Values are handled in section 6.2.5.
An action rule also has a priority value. This priority value has nothing to do with probabilities, but with the importance of an action for the situation it belongs to. The priority values are values between 0 and 1. A priority value below 0.5 means that the action might never be performed during the situation. A priority value of 0.5 or above means that the action is mandatory.
A priority value of 1 means that the action is vital for the successful completion of the maneuver. If the action is mandatory but the control or instrument is never set to the given value during the situation, the pilot might have forgotten to perform the action and he might have to be informed about that.

Some situations can be split up into a number of phases which all have a certain set of actions that have to be performed during that phase. The actions that have to be performed in a phase are all time-dependant in the sense that they must be performed in a certain chronological order. There are also time-independent actions, which are actions that might be performed at any time during the situation. Therefore the set of action rules is split up into two parts, one for every phase and one for every time-independent part. If an action in the time-dependant part is performed "out of turn", which means it is performed while not all actions that should have been performed earlier have been performed yet, this might result in two things:

The probability value of the rule might become smaller or the pilot might have to be informed that he has forgotten to do something.

Visual check rules:
A visual check rule states that the pilot should check a certain instrument during the situation. These rules can be used when a gaze tracking device is used to observe the pilot during the flight. When the system detects that a certain situation is occurring and the pilot forgets to check an instrument, the system could give the pilot a hint that he should check the instrument.

Conditional rules:
The conditional rules can be used to determine if a situation has started or if a situation has finished. These rules apply to the same controls or instruments as the action rules. For each control or instrument there are at most two conditions. One condition defining the value the control or instrument should have at the start of the situation and one condition that states what value the control or instrument will have at the end of the situation. The set of start- and end-conditions actually forms one rule that states that if the given controls and instruments have the given values the probability that the situation has started or ended has a certain value. These probability values are predetermined. The end rule can only fire if at some time in the past the system has detected the start of the situation.

Additional rules:
Additional rules are rules that do not fall under one of the above categories, e.g. rules that specify maximum or minimum values a control or instrument can have during a situation. They might also say something about actions that should not be performed during a situation and how much the probability will decrease if such an action is performed.

6.2.4 Probability

A probability calculator will combine all probabilities that are the result of the rules that fire and calculate a new probability for the situation. Thus, depending on the rules that fire, the probability the Cessna is in a certain situation will increase or decrease.
The probabilities which are stored in the knowledge base are fuzzy values from the following fuzzy set: VBP (Very Big Positive), BP (Big Positive), MP (Medium Positive), SP (Small Positive), VSP (Very Small Positive). The probability calculator will combine all the fuzzy values from the rules and produce one fuzzy value which represents the probability of the situation it belongs to. Once the probability calculators for every situation have produced a probability, an overall controller will evaluate all those probabilities and determine if it can say with enough certainty which situation is taking place.

How the fuzzy values will be implemented and which combination algorithm should be used is not part of this document. For that information we refer to the work done by Quint Mouthaan in his document called "*Towards an intelligent cockpit environment: a probabilistic approach to situation recognition in an F-16*".

6.2.5 Values

In the above sections we mentioned values. The values of the controls and instruments in the action and conditional rules might be a range of values. This is indicated with the mathematical symbols < (smaller than) and > (bigger than). Furthermore it is possible that a control or instrument can have more than one value or it might be that a control or instrument should not have one or more values. This is indicated with the logical operators '|' (OR), '&' (AND) and '!' (NOT).
The operator '|' is used for controls or instruments that may have more than one value, '!' is used for controls or instruments that are not supposed to have a certain value and '&' is used together with the '!' operator if there is more than one value that a control or instrument is not supposed to have.

6.2.6 Situations

In this section all the situations that we will want to recognize are described. In describing the situations a lot of terms and abbreviations are used that may be unknown to most people.
The following situations will be described by the knowledge base:

- Startup
- Taxiing to runway
- Taking off
- Aborting take off
- Normal flight
- Landing
- Aborting landing
- Flame out landing
- Taxiing to ramp
- Shutting down

Startup

Startup is the phase in which the plane is made ready to taxi to the runway. All systems are switched on, to make starting of the engine possible. The navigation beacons and strobes are switched on, to make it visible for the environment that the plane is in the startup state.

Taxiing to runway

During this phase the pilot drives the aircraft to the runway, which he will take off from. This is considered a different situation than taxiing from the runway after the pilot has landed, because the start and end states are switched. The state of the airplane at the beginning of this phase should be equal to the state of the airplane after starting engine Taxiing is assumed to be finished when the aircraft comes to a

halt. It is assumed the pilot will not halt the airplane between the ramp and the start of the runway. This is a reasonable assumption because the pilot usually has only a limited amount of time for takeoff. The flight controllers generally want to get the airplanes in the air as fast as possible. Therefore it is also assumed that if the pilot arrives at the runway and is cleared for takeoff by the tower he is not required to hold short and can enter the runway immediately. However once on the runway he will have to halt to make a final check of his systems. This is why taxiing can still be considered to be finished when the aircraft comes to a halt.

Take off

This phase starts with an airspeed of zero, because the pilot will stop at the start of the runway to make a final check of the systems. The program will have a very good indication that the take off has started when the ground speed exceeds the maximum taxi speed. If the plane reaches the minimum altitude for flight then the takeoff phase can be considered to be ended once all actions in the action list have been performed. Aborting a takeoff can only be done when taking off has already started and before it has finished. Therefore the aborting takeoff situation is a nested situation. When the system detects that the takeoff is being aborted it will consider the taking off situation to be finished.

Normal flight

In the situation normal flight the pilot is flying towards a steering point. This is the situation in which the pilot will find himself most of the time. Because this is the standard situation we will assume that if no other situation is detected then there is a big chance that the aircraft is in normal flight. This is implemented by setting the start probability of the normal flight to VBP and the end probability to VSP, although there is only a small amount of start and end conditions. Examples of the conditions are that the altitude must be bigger than the minimum altitude. There are also no real characteristic actions or visual checks the pilot should perform in this situation. When the start probability of another situation becomes high enough, the normal flight situation will be considered finished.

Landing

When the pilot gets below the minimum altitude, the landing phase starts. The start is also marked with actions as extending the flaps, and lowering the speed. When the pilot touches down, meaning when the altitude above ground level is zero, he or she will have to brake. This marks that the situation is almost ended, as the speed drops to taxi speed.

Aborting a landing

Aborting a landing occurs when for some reason the landing cannot continue. In order to abort a landing the pilot usually increases throttle and pitches the nose of the plane up. Since these two variables are the exact same for a flame out landing one other variable is used, namely the altitude.

Flame out landing

As stated above, a flame out landing is almost the same as aborting a landing. The flame out is used to get the nose up a little more just before touchdown. In order to achieve this, the plane has to pick up a little more speed and the nose has to have some help from the elevators. What the pilot does at this point, is throttle up the engine and pulls on the yoke. This phase will end when the wheels have made contact with the ground, i.e. when the altitude above ground level is zero.

Taxiing to ramp

When the pilot has finally landed, he will taxi from the runway. This phase starts when flame out landing has ended and the ground speed is below the maximum taxi speed. The situation ends when the ground speed is zero and parking brakes have been set.

Shutting down

This is the phase in which the pilot shuts down all systems. The most important information of this phase is the end state of the airplane, so that the system may be able to check if the pilot has not forgotten to shut down a system.

6.3 Expert Systems

6.3.1 Description

Conventional programming languages, such as C and Java, are designed and optimized for line-by-line reasoning. Humans, however, often solve complex problems using (dependant on the person) very abstract approaches. Although abstract information can be modeled in these languages, considerable programming effort is required to transform the information to a format that's usable and takes a reasonable amount of time to run through.
Research in the area of artificial intelligence incorporated the development of techniques that allow a higher level of abstraction while modeling information. Tools have been developed that allow programmers to develop and maintain at an easier level the abstract problems many customers or supervisors provide them with. These tools are called expert systems.

6.3.2 Boolean logic

Boolean logic is the most commonly used logic used by programmers. Boolean logic is also the oldest logic around in the programming scene, developed by an English mathematician and computer pioneer, George Boole who lived from 1815 until 1864. Boolean logic is a very simplified means to determine the outcome of a given problem, because it uses only two statements, i.e. a statement is either TRUE or FALSE.
A way to integrate these values in a program is to make use of if-then statements that check if a variable has reached a certain value (if it has become 'TRUE') and

then execute a certain rule provided by the programmer. Subsequently 'if' rules can be stretched by entering three different operators, namely AND, OR and NOT. 'AND' operators are used when two or more variables are considered to be correct for a rule, 'OR' operators are used when either of the two or more variables are considered to be correct and 'NOT' operators are used when a variable should not have a certain value.

```
if($Flaps == 0 and $FuelFlow > 1500 and $Airspeed > 70) {
  print"\t\t\tFlight: Flying\n";
  }
if($Flaps == 0 or $FuelFlow > 1500) {
  print"\t\t\tFlight: Flying\n";
  }
if($Flaps == 0 and !$FuelFlow > 1500) {
  print"\t\t\tFlight: Flying\n";
  }
```

Figure 34 Different 'if' statements.

The first statement in the above figure shows that when the Flaps are in resting position and the fuel flow is greater than 1500 and the airspeed is greater than 70 knots, the plane is considered to be in normal flight. The second statement shows that when the flaps are in resting position or the fuel flow is greater than 1500, the plane is considered to be in normal flight. The last statements shows that when the flaps are in resting position and the fuel flow is not greater than 1500, the plane is considered to be in normal flight.
The problem with this type of programming is that it's time consuming, it slows the program down considerably and it's not very accurate. By the latter it is meant that if there are two separate situations that are almost alike, such as 'Taxiing from runway' and 'Taxiing to runway', the program will have a hard time determining what situation it is actually in. To counteract that the first prototype was built with references to upcoming situations, but this was quickly dismissed, because of lock-ups in situations. Sometimes the plane was in Normal flight, but the program still thought it was Taking off. One solution was to look into the basics of Fuzzy logic.

6.3.3 Fuzzy logic

As an alternative to Boolean logic, where one variable is either true or false, Fuzzy Logic was introduced by Lotfi Zadeh, a professor at the University of California at Berkley, as a way of processing data by allowing partial values rather than crisp set values. This approach was not applied to control systems until the 70's due to insufficient small-computer capability. The reasoning behind this, was that normal humans do not require precise information. This is illustrated in the next example. If a computer is told a certain person is "young", the computer cannot do anything with that information, whereas a person regards "young" as an age of 1 to 21, or even 30, depending on the person's own age.

Another example is the Fuzzy Logic used in present cars. Due to environmental laws, a car has to produce as little as possible pollution. In order to achieve this, computers have been installed in all new cars as a means to regulate the exhaust fumes of cars.

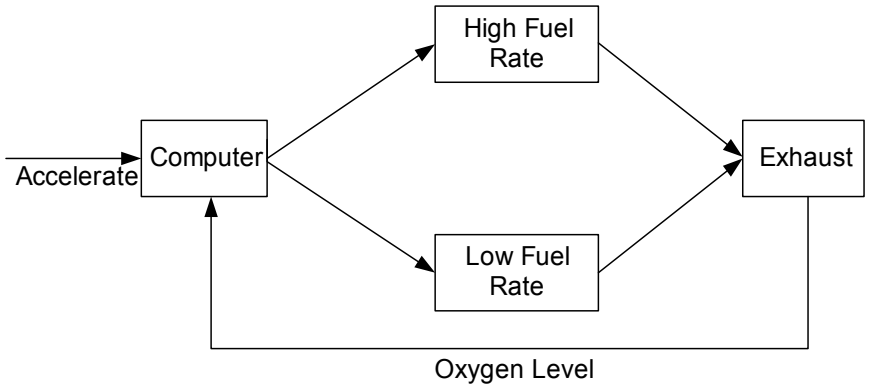


Figure 35 Exhaust regulation with Fuzzy Logic in a car.

As can be seen in the above illustration, the computer get a signal that the driver wants to accelerate. Now with the environmental laws the amount of CO_2 should be as low as possible and thus the O_2 (Oxygen) as high as possible. This is done through a sensor, called the Lambda-Sensor, which is situated in the exhaust pipe. This sensor sends a signal to the computer, which will adjust the amount of fuel injected into the engine in order to achieve a high level of oxygen in the exhaust fumes.

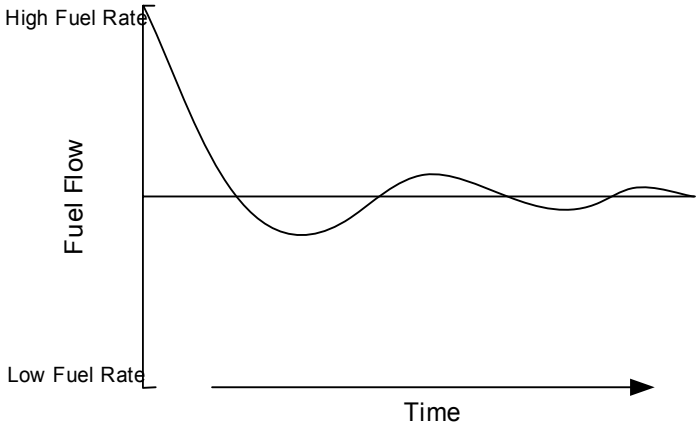


Figure 36 Response of injection computer to oxygen sensor.

This example can be fitted onto any type of situation. If we were to apply this model to an aircraft, you could picture the Fuel flow rate to the altitude of a plane where the central line would be the optimum altitude, or given altitude by the flight plan, and the curve the current altitude of the aircraft. If an auto-pilot were to be switched on, the computer would try to get to the optimum altitude in the shortest amount of time and with a minimum amount of discomfort.

6.3.4 JESS

The Java Expert System Shell (JESS for short) was developed by Ernest Friedman-Hill at Sandia National Laboratories as part of an internal research project. The first version of JESS was written in late 1995, when JAVA was very, very new.
JESS is a tool for building a type of intelligent software called Expert Systems. An Expert System is a set of *rules* that can be repeatedly applied to a collection of *facts* about the world. Rules that apply are *fired,* or executed. Jess uses a special algorithm called *Rete* to match the rules to the facts. Rete makes Jess much faster than a simple set of cascading *if- then* statements in a loop. Jess was originally conceived as a JAVA clone of CLIPS, but nowadays has many features that differentiate it from its parent.
As was probably clear already, JESS is written in JAVA. It is compatible with CLIPS (See next section) and most JESS scripts are valid CLIPS scripts. JESS has some extra functionality over CLIPS. One of these extra functions is that JESS makes it possible to communicate with JAVA objects during execution of a JESS program. JAVA is renowned for its slowness in comparison to C or C++ programs, so real-time applications might not seem as something you want written in JAVA. This is where JESS steps in, because JESS takes over the reasoning part of the awareness system, which of course it is designed for. Combined with the XML parser, which is in a separate program and not in the Data Client, the real-time analyses of data is possible.
As stated before, JESS has been around for quite a while and it is constantly undergoing updates and fixes. We originally wanted to use version 6.0 first, but since that time version 6.1p1 has been released. Unfortunately time was limited to us and thus JESS was not implemented in the prototypes.

6.4 Temporal Reasoning

6.4.1 Introduction

As previously described in chapter 3, one of the main problems discovered in the prototypes, was the accuracy of the analyzed situations. In normal flight conditions the accuracy was quite good, but at the point of an emergency or unexpected situation, the program fails to give an accurate advice. One of the solutions to this problem is temporal reasoning. The operation of this system is described in the following part of this thesis.

6.4.2 Problem setting

The main problem in analyzing a situation is the way an human react to situations, and the way a computer program react. The main difference is, that a computer program can only do what it is trained for, and does not know how to react in an undocumented situation. There are numerous ways to circumvent this problem, but the technologies available at this moment in time are limited. The best solution to the problem would be a neural network. But as previously mentioned, the

technologies available at this point in time, are too limited. This results into the following problem setting:

"How can a situation be accurately detected with technologies and knowledge currently available"

6.4.3 Philosophy

To come to a solution to the problem, we fist took a look on how a human react to unexpected situations. Although this is a problem on it's own, and certainly needs some further studying, the basis of this can be extracted by common knowledge and own experiences.

The first thing that came to light is the human ability to improvise, an ability a computer program lacks. A human person always knows, or improvises with the information available at the moment, how to react to an unexpected situation. This ability provides a solution to the problem that has occurred. Because a computer program lacks this ability it can only react to the situation in the way it is trained to do that. This normally results in errors and solutions to the problem that is not logical.

6.4.4 Solution

Taking the previously mentioned philosophy in consideration, we have thought of the following solution. Humans think in a three, maybe even four dimensional way. When a situation occurs he or she knows how to react, based on experience and knowledge gained in life. This can be regarded as an alternate timeline running parallel to the current timeline. When an alternate situation occurs which is not as expected, the person will "enter" the second timeline and react to the situation. If the problem is solved as expected, the person will go back to the normal timeline. When this is not the case, the person will "enter" the so called third timeline, also known as the emergency timeline.

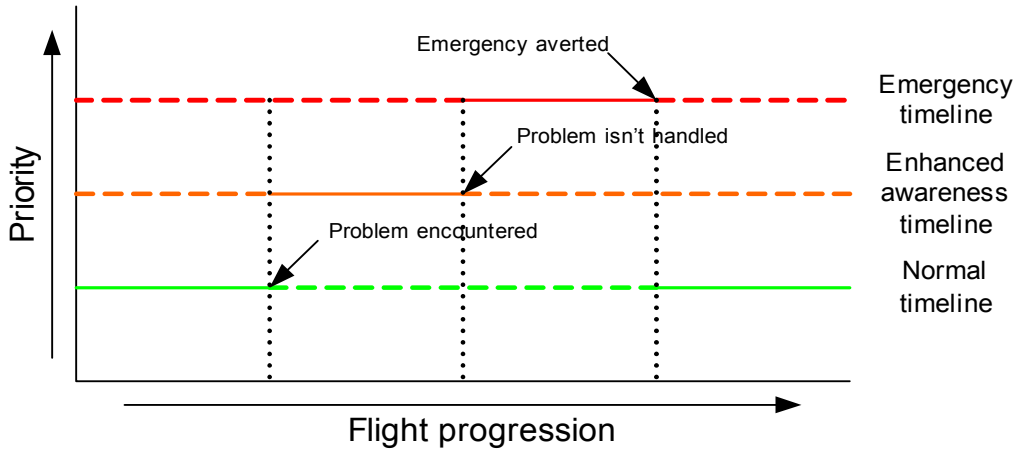


Figure 37 Different timelines displayed in a diagram.

When a situation occurs which involves the third timeline, a high level of stress can be expected. The person will probably discard the pervious timelines, and will do all in its power to return to a normal situation. This also characterizes the third timeline. Looking as this fact in a more technical way, rather then a psychological way, it distinct itself as a different timeline, as the way that the second timeline distincts itself from the first.

Taking all of the above into consideration, as described above would be the use of timelines. This enables the program to simulate a part of the creative thinking that distinct it from its human counterpart. Implementing such a system into the program could lead to an improvement of the accuracy. This system would improve the accuracy in abnormal situations, and will also improve the accuracy of normal situation, as timelines are used, and this result in a relation between situations. It enables the program to predict the next possible situations, based on what situations that have occurred in the past.

6.4.5 Functional Implementation

The implementation of this idea is done by an extension of the knowledge base. The original knowledge base structure by Quint Mouthaan already had some room for a following situation, but lacked the structure for possible following situations.

The Data Client itself will make distinction between the timelines. For now the program will know three timelines:

- Normal timeline
- Enhanced awareness timeline
- Emergency timeline

The implementation in the knowledge base is done by adding a <time constraint> tag within the situation tag. Within this tag the possible following situations are listed, as well as the situation before the current. The listed situations also get a probability value, according to the probability they will happen. These probabilities have a direct connection to the timelines. The lesser the probability is of a following situation, it will be in a higher timeline. Meaning that the highest probabilities will be contained in the normal timeline, as the lowest will be in the emergency timeline. As a future implementation, one can choose for to let the program override the pilot, if a situation is detected which does not fit in the first timeline.

(This page has been left blank intentionally)

Chapter 7: Project results

7.1 Description

This chapter will describe what has happened and what has been achieved during the period that was set for this part of the ICE project. This chapter will discuss what project results have been achieved and how they correspond to the project goals, which were set in Chapter 1 and 2.

7.2 Literature study

To get ourselves familiar with the ICE project we first studied the journals and reports that were available to us through the network of the TU Delft. After the conclusion of the preliminary research we made extensive studies into the field of expert systems, neural networks and JAVA.

7.3 Coginitive and system model

7.3.1 Cognitive model

As was described in the project goal, one of the main sub goals was the creation of a cognitive model, which is able to do real-time assessment of situations, using a rule-based system in combination with adaptive logic.

7.3.1.1 Data acquisition

The first stage of creating a cognitive model is the data acquisition. This is done by the *Data Server* and the *Data Modeler*. The Data Server is responsible for the throughput of data. It feeds all variables available for the reasoning process to the *Data Client*.

The Data Modeler provides the user in a graphical way the ability to create their own context, or load one in using the XML parser. Because of the way the data is presented, it's easy to understand the context of the data, and how it will be interpreted by the system. Because of the use of the XML parser, it is also possible to create own contexts with specific characteristics, which can be altered with the Data Modeler by the user.

7.3.1.2 Rule based reasoning

Trough use of the Data Modeler the user provides the system with a rule based context model. The rules inputted into the system are converted to rules interpretable by the program. The program itself does not rely on any native code to do the reasoning, so all the situations are assessed by the data provided from the modeler. This also makes the system highly adaptable. Using this approach makes it

possible for the user, to do the same analyzing process, for different types of planes.

7.3.1.3 Adaptive Logic

The program provides the user with the means to use adaptive logic. The current version of the SAM program only uses internal reasoning systems, but in future releases it can be equipped with external expert systems.

7.3.2 System model

The three layered system model we chose, namely the Modeler, Server and Client has been implemented as the model we described in Chapter 5.
The Data Server and Data Modeler work like we intended them to work. The Data Modeler is implemented with hard-coded if-then statements, but is guided by the context file the Data Modeler provides. The fuzzy logic expert system is not yet implemented in the Data Client; this is intended to be done by another team as a graduation thesis.

7.4 Artificial Intelligence techniques research

As stated in section 7.1 extensive studies have been made into different kinds of Artificial Intelligence techniques and the tools one can use to implement them in self-created applications. We also looked into the possibilities of Boolean logic, which is used in the Data Client. The intend was to implement JESS as a Fuzzy Logic Expert System, but as stated in section 7.3.2, this goal was not reached.

7.5 Demonstrator

The Data Server and Data Modeler can be considered as finished. The Data Client however, still has several bugs in it.
This is not to say that the entire Data Client does not function, it is only limited to a few working functions.
The functions of the Data Client that do work are listed below:

- Connect and maintain connection to the Data Server.
- Change frequency of data gathering.
- Draw conclusions based on manually inputted context run through Boolean logic.
- Display the blackboard and its variables.

7.6 Evaluation, tests and validation

During the implementation phase of the project, all software developed went trough extensive testing and validation. As a result of these tests, we could incorporate the results into future versions of the program, in order to avoid recurring defects in the reasoning functions of the program. This approach to the programming resulted into to a better understanding of how the program reacted to problems given to it.

(This page has been left blank intentionally)

Chapter 8: Evaluation

8.1 Description

In this chapter we will make an evaluation on the work that has been done and needs to be done in the future.

8.2 Prototypes

In the tradition of the HR&O we made a jump-start in the programming of our very first prototype. This prototype was written in PERL and made use of the now infamous if-then statements. The first thing we noticed was not the incorrectness of situation recognition, but rather the slowness of the simulator to provide variables over the network. This was not because of slow network connections, but was totally due to simulator itself and the way it is built.

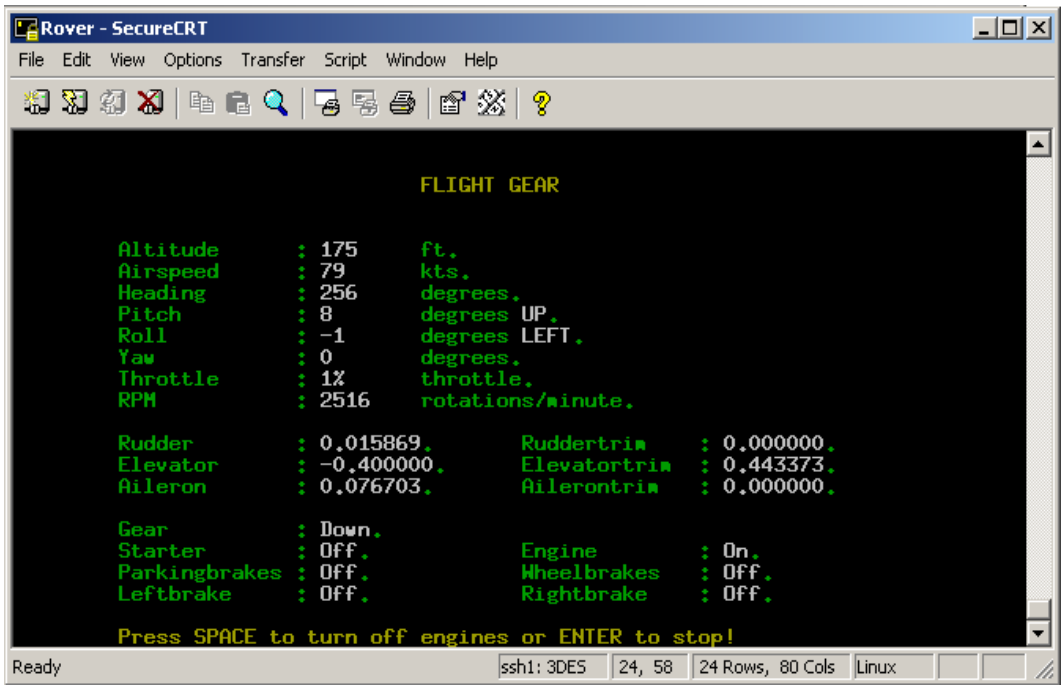


Figure 38 The first prototype running on a Linux operating system.

In order to circumvent these slow-downs, we decided to use a Data Server, which could extract the variables and their values, store them into memory and let the Data Server provide the values and variables to the prototype.

The next notice was the inability of the prototype to make dissensions between Taxiing to runway and Taxiing from runway. In order to solve this, we introduced "pointers" to the next possible situations, but this resulted in extensive coding and that in turn made the program very slow. Also, on an execution level of the program, if a situation had ended without a certain variable was passed on to the Data Client, the situation would be reported perpetually and thus would the Client

be of no further use. The thought of using an expert system came into mind, but with the way the program was built, it meant making it ready for an expert system. This is when the real problems started, because as it turned out, there were several bugs in the program and the source code was poorly documented
The most of the time available to us went into figuring out what specific functions were doing, restyling of the source code and bug tracing, which means that in the end the expert system was never implemented.
We accomplished the task of restyling the source code and making it more readable even tough the documentation provided with the source code was very poorly arranged, if not non-existent. Also, some bugs were dissolved (mainly because of the restyling), but to our frustration, some of the bugs were so deeply engraved in the code, it turned out to be an undo-able task in the time that was given us.
One of those bugs loads the context into the Data Client as provided by the Data Modeler's XML parser in a wrong way, which results in the Data Client stalling when it passes its first situation test.

8.3 Proof of Concept

Of course, everybody thinks his or her ideas are the best. What makes the difference between a good idea and a good working concept is the way it is thought out. If an idea is good, but the background information was already faulty or the reasoning process the person used is shaky, the idea may very well never work. Now, why do we think the idea is a good one? Basically, because we looked at several points of angle to the problem. Ours was more of a "trial and error" approach. As stated before we started out with a prototype that made use of a lot of if-then statements, with which later on we realized that in order to make it work this way, a tremendous amount of programming and testing would be needed. Looking at the time we had this meant that at least a year would be needed in order to come to a satisfying end.

It is therefore our belief that if-then statements are not the correct approach of this problem. The logical next step is the use of "fuzzy" expert systems.
In the first stage of our project, the program was "dumb", but with the use of expert systems, the program gets a certain level of intelligence.

The introduction of a XML file, is solely done in order to achieve a level portability. This portability is not only intended on a operating system-basis, but also on an aircraft-basis. An aircraft's systems and specifications, as well as air traffic regulations can be stored in a XML file, which will then be parsed by, in this case, the Data Modeler. It is our belief that this is the simplest solution for the portability issue.

Different timelines are a way to keep track of special situations, such as collisions and emergency landings. Basically these timelines come in effect when an out-of-the-ordinary event is imminent. For instance, when an aircraft is on a collision course with another aircraft, the normal timeline is set aside; the program gives a warning and gives possible solutions in order to evade the upcoming aircraft.

8.4 Recommendations

The expert system's intelligence may be enough to reach a satisfactory level of situation awareness for Flight Simulators, but in "real life" it is still unacceptable. To increase the level of intelligence, it is almost inevitable to look at the use of neural networks. Neural networks work with the same concept as the human brain. The human brain "trains" itself in order to come up with solutions to everyday and specific problems. Neural networks are also trained, but at this point in time, the training process of neural networks has not yet been optimized. This means that in order to get a self-sufficient and accurate neural network implementation, a lot of training is required.
The way the training is done, is totally up to the people continuing this project, but we would like to suggest those people make use of a "real-life" pilot; this to come to an as accurately as possible model. It is all fine and well to let a person of the project fly an aircraft in a simulator, but it is most certain that that person will "neglect" to fulfill some of the steps in a flight plan that are important in real-life.

Neural networks alone are not the total answer to the problem of situation awareness in an aircraft. Humans always have an alternate situation in their minds. We call this the "just in case" state. When a pilot is flying, he or she is constantly aware that something out of the ordinary could happen. A computer system simply does not have that ability unless it is specifically implemented in its programming. Therefore, to make an as accurate as possible situation awareness program, we recommend the implementation of the three timelines. Maybe this could be a separate project?

(This page has been left blank intentionally)

Appendix A: The knowledge base in XML

A.1 Description

As said before, the knowledge base will be stored in a XML file. XML is a tag-based language for structured documents or data. It is not a way to represent data like HTML, although it could be used that way. XML is described on the official XML website of the World Wide Web Consortium [1] in ten points:

1. XML is for structuring data.

2. XML looks a bit like HTML.

3. XML is text, but is not meant to be read.

4. XML is verbose by design.

5. XML is a family of technologies. XML has been extended by a lot of people with all kinds of modules, which makes XML very powerful.

6. XML is new, but not that new.

7. XML leads HTML to XHTML.

8. XML is modular.

9. XML is the basis for RDF (Resource Description Framework) and the semantic web. RDF is an XML text format that supports resource description and metadata applications.

10. XML is license-free, platform-independent and well-supported.

After all data that should go into the knowledge base had been selected it had to be put into a form that could be used by a program to reason with the knowledge. We considered languages like CLIPS and JESS, but eventually chose for XML because of the following reasons:

- XML is a widely accepted standard.
- XML is easier to read than a list of rules.
- XML is very well supported.
- If the knowledge base is written in XML it can easily be extended.
- With XML it is possible to define a DTD (Data Type Definition) or a schema that defines the structure the XML file should have so that future knowledge bases for other aircrafts will have the same generic form.
- XML data can easily be translated to another desired format like CLISP or JESS.
- It is easy to write a program to translate the XML data to if-then rules. This way we will be able to generate a lot of rules from just a few lines of XML code.

The structure of the XML file will have to be very well defined in order to be able to describe flying with other aircrafts in the same way as we have done for the Cessna. The ideal situation would be that one program could be written that detects the current situation in a flight and that that program can be used for different aircrafts if a different XML file is used as the knowledge base. We have tried to achieve this by specifying the structure the XML file should have in an XML schema.

A.2 The schema for a flightplan

As explained in section A.1, a XML schema has been created for defining a flight plan. This flight plan can then be used to help determine the current situation in a flight. The flight plan schema has been made independent of the aircraft with which the flight is flown, just like the XML flight schema. A XML file of a flight plan will have the following structure:

- flightplan
    - steerpoint
        - heading
        - altitude
        - TOS (Time Over Steerpoint)
        - action
    - steerpoint
        .
        .

The information that is stored about a steer point is the heading that should be flown towards the steer point, the altitude at which the pilot should fly over the steer point and the time at which the pilot should reach the steer point (TOS). This is all standard information that is also present in real flight plans, however what can also be added in this flight plan is the actions a pilot will perform at the steer point.

If there are more that one actions to be performed at a steer point then the actions specified at the steer points should be in chronological order, so the first action in the XML file should be performed first, then the second and so on.

A.3 The XML flight scheme

There are several ways in which we can define the structure to which the XML files for different aircrafts must conform. We can use either a DTD or an XML schema. The DTD is older than the XML schemas and therefore has less functionality and is less flexible. On the other hand, DTD is much easier to understand and implement. Although we could use the DTD because the form in which we will store the data in the knowledge base is not too complex, it will make the knowledge base much more flexible and easier to expand if an XML schema is used to define the structure. Therefore we have chosen to use an XML schema. The complete XML schema is given in appendix B.

A.3.1 XML specific considerations

When creating a XML file, it is often not clear when to use an element to denote information and when to use an attribute. For example the following pieces of XML all contain the same information:

```
<tag name="test">
  <child name="child1">10</child>
  <child name="child2">1</child>
</tag>

<test>
  <child name="child1" value="10" />
  <child name="child2" value="1" />
</test>

<test>
  <child1>10</child1>
  <child2>1</child2>
</test>
```

Figure A.1 A XML code example showing the same function.

In the first example `tag` and `child` are the elements and `name` and value are attributes. In the second example the `tag` element has been replaced by a `test` element. These two elements contain the same information. Finally in the third example the `child` element has been split into two different child elements. There are no clearly defined rules that say when to use an attribute and when to use an element; it often depends on the kind of information and the intention of the author of the XML file. Our main consideration in deciding whether to use element or attributes is the generality of our XML schema. To make our XML schema usable for other aircrafts our tags can not be too specific and we will have to put a lot of information in the attributes.

A.3.2 The hierarchy

The hierarchical structure of the XML file as it is defined in the XML schema is the following:

- flight
  - situation
    - actions
      - phase
        - action
          .
          .
          .
          .
        - visual checks
          - instrument
            .
            .
        - constraints

- constraint
  .
  .
- time constraints
  - time constraint
    .
    .
- situation
  .
  .

Every element in this hierarchy will be a tag that can be put in the XML file. The tags all have some metadata and a certain meaning.

flight

The flight tag is the root tag of the XML document. It contains the name of the aircraft for which the knowledge base has been created.

situation

For every situation there is a separate tag with the name of the situation in it. It also has an attribute that contains the time window of the situation.

actions

This tag is the parent tag of the set of action rules for the situation.

phase

This tag is the parent of all actions that are part of a phase in the situation. It has an attribute containing the name of the phase. The actions that are the children of this phase tag should usually be performed in the order in which they occur in the XML file, so the first action of the first phase should be performed first, then the second action of the first phase, etc., until all actions of the first phase have been performed after which the actions of the second phase should be performed and so on. The exceptions are action tags that are children of the phase tag with the name "time independent". These actions may occur at any time during the situation in random order.

action

This tag defines an action rule for the situation. It contains the name of the control or instrument, the priority value and the fuzzy probability value as attributes. The value of the element is the value the control or instrument will get when the action is performed.

visual checks

This tag is the parent tag for all the instruments that the pilot should visually check during the situation.

instrument

This tag is a tag for an instrument that the pilot should check during the situation. For now this tag only contains the name of the instrument, but in the future this might be extended, in order to contain information for a gaze.

constraints

The constraints tag is the parent tag for all start- and end-constraints on controls and instruments for a situation. It has two attributes with fuzzy probability values. These are the probability values for the start and end rules.

constraint

This tag contains information about the value a control or instrument will have at the start and/or end of the situation. It contains a name attribute with the name of the control or instrument, a start attribute with the value the control or instrument should have at the start of the situation and an end attribute with the value the control or instrument should have at the end of the situation.

time constraints

The time constraints tag is the parent tag for all possible start- and end-constraints on controls and instruments for a situation. It has two attributes with fuzzy probability values. These are the probability values for the alternate situations.

time constraint

This tag contains information about the value a next situation will have at the start and/or end of the situation. It contains a name attribute with the name of the situation and an attribute with the possibility value the situation should have at the end of the situation containing the chances that that would be the following situation.

A.3.3 The values

A lot of the values that will be stored in the knowledge base represent positions of switches and buttons in the cockpit. The easiest way to represent these values for a computer program is with numbers (e.g. the ON position will get the value 1 and the OFF position will get the value 0). These numbers are not easy to interpret for human readers however and since we want the XML file to be readable, such values will be represented by variables. In a DTD it is possible to define variables the can be a reference for the XML file. The ON and OFF positions could for example be defined as:

```
<!ENTITY ON "1">
<!ENTITY OFF "0">
```

Figure A.2 XML values described.

They could then be referenced in the XML file by putting the name of the variable between a ”&” and a ”;” (e.g. &ON;). When a parser reads the XML file all variables will be replaced by their values. The variables will be defined in a separate file so that they can easily be changed. This file will then be specified in the XML file as the DTD for the XML file. The DTD is displayed in Appendix D.

Appendix B: The knowledge base described in tables

B.1 Prestartup

The Actions

Name	Value	Priority	Probability
Prestartup			
Parking brakes	ON	1	Best Probability
Throttle	IDLE	1	Best Probability
Ignition switch	OFF	1	Best Probability
Avionics power switch	OFF	1	Best Probability
Master switch	ON	1	Best Probability
Pitot heat	ON	1	Best Probability
Avionics master switch	OFF	1	Best Probability
Static pressure alternate source valve	OFF	1	Best Probability
Fuel selector valve	BOTH	1	Best Probability
Flaps	FULL	1	Best Probability
Pitot heat	OFF	1	Best Probability
Master switch	OFF	1	Best Probability
Fuel shutoff valve	ON	1	Best Probability
Taxi and landing lights	OFF	1	Best Probability
Beacon	OFF	1	Best Probability
Strobes	OFF	1	Best Probability
Navigation lights	OFF	1	Best Probability
Trim	SET	1	Best Probability

The visual checks

Instrument
Fuel Quantity Indicators
Annunicator panel

The conditions

Name	Start	End
Parking brakes		ON
Ignition switch		OFF
Pitot heat		Checked
Navigation light		Checked
Beacon		Checked
Strobes		Checked
Taxi & landing lights		Checked
Avionic master switch		ON
Fuel selector	OFF	BOTH
Fuel shutoff valve		ON
Master switch		OFF
Avionics power switch		ON
Flaps	RETRACT	FULL
Elevator trim		0
Rudder trim		0
Ground speed	0	0
Altitude	0	0

B.2 Startup

The Actions

Name	Value	Priority	Probability
Startup			
Parking brakes	ON	1	Best Probability
Brakes	ON	1	Best Probability
Avionics power switch	OFF	1	Best Probability
Avionics master switch	OFF	1	Best Probability
Circuit barkers	In	1	Best Probability
Fuel selector valve	BOTH	1	Best Probability
Fuel shutoff valve	ON	1	Best Probability

The visual checks

Instrument
Circuit breakers
Brakes
Autopilot

The conditions

Name	Start	End
Parking brakes	ON	ON
Brakes	OFF	ON
Circuit breakers	OFF	ON
Autopilot		OFF
Avionics power switch	OFF	OFF
Fuel selector valve	BOTH	BOTH
Fuel shutoff valve	ON	ON
Elevator trim	0	0
Rudder trim	0	0
Ground speed	0	0
Altitude	0	0

B.3 Starting engine

The Actions

Name	Value	Priority	Probability
Starting Engine			
Fuel shutoff valve	OFF	1	Best Probability
Throttle	IDLE	1	Best Probability
Mixture	LEAN	1	Best Probability
Master switch	ON	1	Best Probability
Auxiliary fuel pump	ON	1	Best Probability
Ignition switch	Start	1	Best Probability
Ignition switch	OFF	1	Best Probability
Mixture	RICH	1	Best Probability
Auxiliary fuel pump	OFF	1	Best Probability
Navigation lights	ON	1	Best Probability
Beacon	ON	1	Best Probability
Radios	ON	1	Best Probability
Flaps	RETRACT	1	Best Probability

The visual checks

Instrument
Oil pressure
Engine instruments
Radios
Avionics power

The conditions

Name	Start	End
Parking brakes	ON	ON
Brakes	OFF	ON
Avionics master switch	OFF	OFF
Fuel selector	BOTH	BOTH
Fuel shutoff valve	ON	OFF
Avionics power switch	OFF	OFF
Beacon	OFF	ON
Strobes	OFF	ON
Navigation lights	OFF	ON
Elevator trim	0	0
Rudder trim	0	0
Ground speed	0	0
Altitude	0	0

B.4 Taxiing to runway

The Actions

Name	Value	Priority	Probability
Taxing to runway			
Throttle	!0	1	Best Probability
Parking brake	OFF	1	Best Probability
Mixture	RICH	1	Best Probability
Flaps	RETRACT	1	Best Probability
Speed	< 20 KIAS	1	Best Probability

The visual checks

Instrument
Brakes
Directional gyro
Turn indicator
Artificial horizon

The conditions

Name	Start	End
Speed brakes		>0
Ground speed	0	<20
Throttle	IDLE	!IDLE

B.5 Before takeof

The Actions

Name	Value	Priority	Probability
Before takeoff			
Parking brake	SET	1	Best Probability
Mixture	RICH	1	Best Probability
Fuel selector valve	BOTH	1	Best Probability
Elevator trim	SET	1	Best Probability
Throttle	1800 RPM	1	Best Probability
Throttle	< 1000 RPM	1	Best Probability
Strobe lights	ON	1	Best Probability
Radios	SET	1	Best Probability
Autopilot	OFF	1	Best Probability
Flaps	SET 0 – 10	1	Best Probability
Brakes	OFF	1	Best Probability

The visual checks

Instrument
Fuel quantity
Magnetos
Suction Gauge
Engine instruments
Caution panel
Fuel flow
Wheel brakes

The conditions

Name	Start	End
Parking brake	ON	OFF
Brakes	ON	OFF
Throttle	IDLE	<1000 RPM
Flaps	0	SET 0-10
Speed	0	0
Altitude	0	0

B.6 Takeoff

The Actions

Name	Value	Priority	Probability
takeoff			
Flaps	SET 0-10	1	Best Probability
Throttle	FULL	1	Best Probability
Mixture	RICH	1	Best Probability
V1 decision	Speed > 55	1	Best Probability
Elevator	0-10	1	Best Probability
V2 Rotate	Speed > 65	1	Best Probability

The visual checks

Instrument
Airspeed

The conditions

Name	Start	End
Airspeed	0	>70
Throttle	IDLE	FULL
Altitude	0	>300
Pitch	0	>0
Climbing rate	0	>0

B.7 Enroute climb

The Actions

Name	Value	Priority	Probability
Climb out			
Throttle	FULL	1	Best Probability
Mixture	RICH	1	Best Probability
Mixture	LEAN	1	Best Probability

The visual checks

Instrument
AirSpeed
Climbrate

The conditions

Name	Start	End
Throttle	FULL	FULL
Mixture	RICH	LEAN
Climbrate	>0	>0
Airspeed	>70	<85
Altitude	>300	>3000

B.8 Cruise

The Actions

Name	Value	Priority	Probability
Cruising			
Throttle	2000/2400 RPM	1	Best Probability
Elevator	0	1	Best Probability
Elevator trim	SET	1	Best Probability
Mixture	LEAN	1	Best Probability

The visual checks

Instrument
Engine instruments
Flight instruments

The conditions

Name	Start	End
Airspeed	>85	!>150

B.9 Descend

The Actions

Name	Value	Priority	Probability
Descend			
Throttle	<2000 RPM	1	Best Probability
Mixture	RICH	1	Best Probability
Elevator	<0	1	Best Probability
Fuel selector	BOTH	1	Best Probability

The visual checks

Instrument
Climbrate

The conditions

Name	Start	End
Mixture	LEAN	RICH
Pitch	0	<0
Climbing rate	0	<0

B.10 Before landing

The Actions

Name	Value	Priority	Probability
Before landing			
Fuel selector	BOTH	1	Best Probability
Mixture	RICH	1	Best Probability
Landing lights	ON	1	Best Probability

The visual checks

Instrument
Engine instruments
Flight instruments

The conditions

Name	Start	End
Fuel selector	LEFT/RIGHT	BOTH
Landing lights	OFF	ON

B.11 Landing

The Actions

Name	Value	Priority	Probability
Landing			
Airspeed	110	1	Best Probability
Flaps	SET 0-10	1	Best Probability
Airspeed	85	1	Best Probability
Flaps	SET 10-30	1	Best Probability
Airspeed	60-70	1	Best Probability
Flaps	FULL	1	Best Probability
Brakes	ON	1	Best probability

The visual checks

Instrument
AirSpeed

The conditions

Name	Start	End
Brakes	OFF	ON
Airspeed	>110	< 20 KIAS
Flaps	0	FULL

B.12 Taxiing to ramp

The Actions

Name	Value	Priority	Probability
Taxiing to ramp			
Throttle	< 1500 RPM	1	Best Probability
Mixture	RICH	1	Best Probability
Landing lights	OFF	1	Best Probability
Flaps	RETRACT	1	Best Probability
Speed	< 20 KIAS	1	Best Probability

The visual checks

Instrument
Engine instruments

The conditions

Name	Start	End
Throttle	IDLE	!IDLE
Mixture	RICH	RICH
Landing lights	ON	OFF
Flaps	FULL	RETRACT
Speed	>0 KIAS	<20 KIAS

B.13 Shutdown

The Actions

Name	Value	Priority	Probability
Taxing to ramp			
Parking brake	SET	1	Best Probability
Mixture	CUT OFF	1	Best Probability
Avionics switch	OFF	1	Best Probability
Ignition switch	OFF	1	Best Probability
Master switch	OFF	1	Best Probability
Fuel selector valve	LEFT/RIGHT	1	Best Probability

The visual checks

Instrument
Engine instruments
Flight instruments

The conditions

Name	Start	End
Parking brake	OFF	SET
Mixture	RICH	CUT OFF
Avionics switch	ON	OFF
Ignition switch	ON	OFF
Master switch	ON	OFF
Fuel selector valve	BOTH	LEFT/RIGHT

(This page has been left blank intentionally)

Appendix C: The XML files

Flightplan.xsd

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="flightplan"
   targetNamespace="flightplan"
   xml:lang="en">

<xsd:annotation>
 <xsd:documentation>
  This is a schema for a flightplan.
 </xsd:documentation>
</xsd:annotation>

<xsd:annotation>
 <xsd:documentation>
 The root tag containing all steerpoint tags.
 </xsd:documentation>
</xsd:annotation>

<xsd:element name="flightplan">
   <xsd:sequence>
            <xsd:element ref="steerpoint" maxOccurs="unbounded" />
   </xsd:sequence>
</xsd:element>

<xsd:annotation>
 <xsd:documentation>
 The steerpoint tag contains information about the steerpoint,
 like heading to fly to the steerpoint, the altitude at which
 the pilot should fly over the steerpoint and the TOS (Time
 Over Steerpoint) which is the time at which the pilot should
 be over the steerpoint. It also contains two attributes, one
 that says what type of steerpoint it is and one that contains
 the number of the steerpoint.
 </xsd:documentation>
</xsd:annotation>

<xsd:element name="steerpoint">
   <xsd:sequence>
            <xsd:element name="heading" type="xsd:integer"/>
            <xsd:element name="altitude" type="xsd:integer"/>
            <xsd:element name="TOS">
                     <xsd:restriction base="time">
                              <xsd:pattern value="hh:mm:ss"/>
                     </xsd:restriction>
            </xsd:element>
            <xsd:element name="action" minOccurs="0" maxOccurs="unbounded"
type="xsd:string"/>
   </xsd:sequence>

<xsd:attribute name="type" type="steerpointType"/>
   <xsd:attribute name="number" type="xsd:integer"/>
</xsd:element>

<xsd:annotation>
 <xsd:documentation>
```

```
 A steerpoint can be one of the following types:
   - STPT: This is a normal steerpoint.
  </xsd:documentation>
</xsd:annotation>

<xsd:simpleType name="steerpointType">
   <xsd:restriction base="xsd:string">
            <xsd:enumeration value="STPT"/>
   </xsd:restriction>
</xsd:simpleType>

</xsd:schema>
```

Flightplan.xml

```
<?xml version="1.0"?>

<!-- This is an example of a flight plan for a Cessna 172-R.
   This flight plan is NOT based on a real flight plan! -->


<flightplan xmlns="flightplan">

   <!-- Departure airfield -->
   <steerpoint type="STPT" number="1">
            <heading>120</heading>
            <altitude>100</altitude>
            <TOS>13:10:00</TOS>
            <action>Pre Startup</action>
            <action>Startup</action>
            <action>Start engine</action>
            <action>Taxiing to runway</action>
            <action>Before takeoff</action>
            <action>Taking off</action>
            <action>Enroute climb</action>
   </steerpoint>
   <!-- Navigational steerpoint -->
   <steerpoint type="STPT" number="2">
            <heading>160</heading>
            <altitude>5000</altitude>
            <TOS>13:34:30</TOS>
            <action>Cruise</action>
   </steerpoint>
   <!-- Navigational steerpoint -->
   <steerpoint type="STPT" number="4">

            <heading>260</heading>
            <altitude>5000</altitude>
            <TOS>14:28:00</TOS>
            <action>Cruise</action>
   </steerpoint>
   <!-- Navigational steerpoint -->
   <steerpoint type="STPT" number="7">
            <heading>270</heading>
            <altitude>5000</altitude>
            <TOS>14:45:00</TOS>
            <action>Cruise</action>
   </steerpoint>
   <!-- Destination airfield -->
   <steerpoint type="STPT" number="8">
            <heading>135</heading>
            <altitude>50</altitude>
            <TOS>15:00:00</TOS>
            <action>Descent</action>
            <action>Before landing</action>
            <action>Landing</action>
            <action>Taxiing to ramp</action>
            <action>Securing Airplane</action>
   </steerpoint>
</flightplan>
```

KB_Cessna.xsd

```
<xsd:documentation>
A schema for an XML file that describes a knowledge base for flying
with a Cessna 172-R. The XML file will have to conform to the following
hierarchy:
- flight
 - situation
        - actions
                - phase
                        - action
                                .
                                .
                        .
                        .
        - visual checks
                - instrument
                        .
                        .
        - constraints
                - constraint
                        .
                        .
        - situation
                .
                .

 </xsd:documentation>
</xsd:annotation>
<xsd:annotation>
 <xsd:documentation>
 The flight tag that contains an attribute with the name of the
 airplane that is described by the knowledge base and has situation
 tags as children.
 </xsd:documentation>
</xsd:annotation>
<xsd:element name="flight">
  <xsd:sequence>
        <xsd:element ref="situation" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="airplane" type="xsd:string" use="required" />
</xsd:element>

<xsd:annotation>
 <xsd:documentation>
 The situation tag with an attribute that contains the name of the
 situation and child elements containing the list of time dependent
 and time independent actions, the list of visual checks and the
 list of constraints.
 </xsd:documentation>
</xsd:annotation>
<xsd:element name="situation">
  <xsd:sequence>
        <xsd:element ref="constraints" minOccurs="0" maxOccurs="2"/>
        <xsd:element ref="actions" minOccurs="0" maxOccurs="2"/>
        <xsd:element ref="visual checks" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string" use="required" />
  <xsd:attribute name="timewindow" type="xsd:integer" use="required" />
```

```
</xsd:element>

<xsd:annotation>
 <xsd:documentation>
 The actions tag is the parent of a number of phase tags.
 The actions in the phases can be time dependent actions
 (which means that they have to be performed in the order
 in which they occur in the table) or time independent actions
 (they may be performed in any order). Time independent actions
 are grouped in a phase called "time independent".
 </xsd:documentation>
</xsd:annotation>
<xsd:element name="actions">
  <xsd:sequence>
        <xsd:element ref="phase" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:element>

<xsd:annotation>
 <xsd:documentation>
 The phase tag is the parent of a number of action tags.
 </xsd:documentation>
</xsd:annotation>

<xsd:element name="phase">
  <xsd:sequence>
        <xsd:element ref="action" maxOccurs="unbounded" />
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:string"/>
</xsd:element>

<xsd:annotation>
 <xsd:documentation>
 The action tag does not have any child tags, but does have three
 attributes. One containing the name of the control or instrument
 that action has an effect on. Another containing the priority
 value of the action. This value is a value between 0 and 1.
 And finally an attribute containing the fuzzy probability value
 of the action.
 </xsd:documentation>

</xsd:annotation>
<xsd:element name="action" type="xsd:string">
  <xsd:attribute name="name" type="xsd:string" use="required" />
  <xsd:attribute name="priority" type="priorityValue" use="required" />
  <xsd:attribute name="probability" type="fuzzyValue" use="required" />
</xsd:element>

<xsd:annotation>
 <xsd:documentation>
 The PriorityValue type is a float between 0 and 1.
 </xsd:documentation>
</xsd:annotation>
<xsd:simpleType name="priorityValue">
  <xsd:restriction base="xsd:float">
        <xsd:minInclusive value="0"/>
        <xsd:maxInclusive value="1"/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<xsd:annotation>
  <xsd:documentation>
The FuzzyValue must be one of the following values:
VBP, BP, MP, SP, VSP, VSN, SN, MN, BN, VBN.
  </xsd:documentation>
</xsd:annotation>

<xsd:simpleType name="fuzzyValue">
   <xsd:restriction base="xsd:string">
            <xsd:enumeration value="VBP"/>
            <xsd:enumeration value="BP"/>
            <xsd:enumeration value="MP"/>
            <xsd:enumeration value="SP"/>
            <xsd:enumeration value="VSP"/>
            <xsd:enumeration value="VSN"/>
            <xsd:enumeration value="SN"/>
            <xsd:enumeration value="MN"/>
            <xsd:enumeration value="BN"/>
            <xsd:enumeration value="VBN"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:annotation>
  <xsd:documentation>
  The visualChecks tag is the root tag for all instrument tags.
  </xsd:documentation>
</xsd:annotation>

<xsd:element name="visualChecks">
   <xsd:sequence>
            <xsd:element ref="instrument" maxOccurs="unbounded" />
   </xsd:sequence>

</xsd:element>

<xsd:annotation>
  <xsd:documentation>
  The instrument tag is a tag describing an instrument that the
  pilot should check during a situation. For now it only has one
  attribute containing the name of the instrument.
  </xsd:documentation>
</xsd:annotation>
<xsd:element name="instrument">
   <xsd:attribute name="name" type="xsd:string" use="required" />
</xsd:element>

<xsd:annotation>
  <xsd:documentation>
  The constraints tag is the parent tag for a set of constraint tags.
  It contains two attributes, the first contains the start
  probability, the second contains the end probability value.
  </xsd:documentation>
</xsd:annotation>

<xsd:element name="constraints">
   <xsd:sequence>
            <xsd:element ref="constraint" maxOccurs="unbounded" />
   </xsd:sequence>
   <xsd:attribute name="endProbability" type="fuzzyValue" use="required"/>
   <xsd:attribute name="startProbability" type="fuzzyValue" use="required"/>
</xsd:element>
```

```
<xsd:annotation>
  <xsd:documentation>
  The constraint tag sets a condition on the value of a control or
  instrument for the start and/or the end of the situation. It
  contains one attribute with the name of the control or instrument
  and two optional attributes for the start and end value of the
  control or instrument.
  </xsd:documentation>
</xsd:annotation>
<xsd:element name="constraint" type="xsd:string">
    <xsd:attribute name="name" type="xsd:string" use="required" />
    <xsd:attribute name="start" type="xsd:string"/>
    <xsd:attribute name="end" type="xsd:string"/>
</xsd:element>

</xsd:schema>
```

Variables.dtd

```
<?xml version="1.0"?>

<!-- Generic variables, used for multiple controls and instruments -->
<!ENTITY OFF "OFF">
<!ENTITY ON "ON">
<!ENTITY CLOSED "CLOSED">
<!ENTITY OPEN "OPEN">
<!ENTITY CHECKED "CHECKED">

<!-- Throttle variables -->
<!ENTITY IDLE "IDLE">
<!ENTITY  FULL "FULL">

<!-- Flaps variables -->
<!ENTITY RETRACT "RETRACT">
<!ENTITY FULL "FULL">
<!ENTITY SET_TAKEOFF "SET_TAKEOFF">
<!ENTITY FLAPS_FINAL "FLAPS_FINAL">
<!ENTITY FLAPS_LANDING "FLAPS_LANDING">

<!-- Engine variables -->
<!ENTITY LEFT "LEFT">
<!ENTITY RIGHT "RIGHT">
<!ENTITY BOTH "BOTH">
<!ENTITY START "START">
<!ENTITY RICH "RICH">
<!ENTITY LEAN "LEAN">
<!ENTITY CUT_OFF "CUT_OFF">

<!-- Airplane variables -->
<!ENTITY MAXTAXISPEED "MAXTAXISPEED">
<!ENTITY MIN_ALT "MIN_ALT">
<!ENTITY CRUISE_ALT "CRUISE_ALT">
<!ENTITY MAX_APPROACH_SPEED "MAX_APPROACH_SPEED">
<!ENTITY MAX_LANDING_SPEED "MAX_LANDING_SPEED">
<!ENTITY LANDING_SPEED "LANDING_SPEED">
```

KB_Cessna.xml

```
<?xml version="1.0"?>

<!DOCTYPE variables SYSTEM "variables.dtd">

<flight aircraft="Cessna 172-R" xmlns="./KB">
<situation name="Pre startup" timewindow="20">
 <actions>
  <phase name="prestartup">
   <action name="parking brakes" priority="1" probability="BP">&ON;</action>
   <action name="throttle" priority="1" probability="BP">&IDLE;</action>
   <action name="ignition switch" priority="1" probability="BP">&OFF;</action>
   <action name="avionics power switch" priority="1" probability="BP">&OFF;</action>
   <action name="master switch" priority="1" probability="BP">&ON;</action>
   <action name="pitot heat" priority="1" probability="BP">&ON;</action>
   <action name="avionics master switch" priority="1" probability="BP">&OFF;</action>
   <action name="static press alt source valve" priority="1" probability="BP">&OFF;</action>
   <action name="fuel selector valve" priority="1" probability="BP">&BOTH;</action>
   <action name="flaps" priority="1" probability="BP">&FULL;</action>
   <action name="pitot heat" priority="1" probability="BP">&OFF;</action>
   <action name="master switch" priority="1" probability="BP">&OFF;</action>
   <action name="fuel shutoff valve" priority="1" probability="BP">&ON;</action>
   <action name="taxi & landing lights" priority="1" probability="BP">&OFF;</action>
   <action name="beacon" priority="1" probability="BP">&OFF;</action>
   <action name="strobes" priority="1" probability="BP">&OFF;</action>
   <action name="navigation lights" priority="1" probability="BP">&OFF;</action>
   <action name="trim" priority="1" probability="BP">&SET;</action>
  </phase>
 </actions>
 <visualChecks>
  <instrument name="fuel quantity indicators"/>
  <instrument name="annunciator panel"/>
 </visualChecks>
 <constraints startProbability="MP" endProbability="BP">
   <constraint name="parking brakes" end="&ON;"/>
   <constraint name="ignition switch" end="&OFF;"/>
   <constraint name="pitot heat" end="&CHECKED;"/>
   <constraint name="navigation light" end="&CHECKED;"/>
   <constraint name="beacon" end="&CHECKED;"/>
   <constraint name="strobes" end="&CHECKED;"/>
   <constraint name="taxi & landinglights" end="&CHECKED;"/>
   <constraint name="avionics master switch" end="&CHECKED;"/>
   <constraint name="fuel selector" start="&OFF;" end="&BOTH;"/>
   <constraint name="fuel shutoff valve" end="ON;"/>
   <constraint name="master switch" end="&OFF;"/>
   <constraint name="avionics power switch" end="&ON;"/>
   <constraint name="flaps" start="&RETRACT;" end="&FULL;"/>
   <constraint name="elevator trim" end="0"/>
   <constraint name="rudder trim" end="0"/>
   <constraint name="ground speed" start="0" end="0"/>
   <constraint name="altitude" start="0" end="0"/>
 </constraints>
</situation>
```

```
<situation name="Startup" timewindow="20">
 <actions>
  <phase name="startup">
   <action name="parking brakes" priority="1" probability="BP">&ON;</action>
   <action name="brakes" priority="1" probability="BP">&ON;</action>
   <action name="avionics power switch" priority="1" probability="BP">&OFF;</action>
   <action name="avionics master switch" priority="1" probability="BP">&OFF;</action>
   <action name="circuit breakers" priority="1" probability="BP">&IN;</action>
   <action name="fuel selector valve" priority="1" probability="BP">&BOTH;</action>
   <action name="fuel shutoff valve" priority="1" probability="BP">&ON;</action>
  </phase>
 </actions>
 <visualChecks>
  <instrument name="circuit breakers"/>
  <instrument name="brakes"/>
  <instrument name="autopilot"/>
 </visualChecks>
 <constraints startProbability="MP" endProbability="BP">
  <constraint name="parking brakes" start="&ON;" end="&ON;"/>
  <constraint name="brakes" start="&OFF;" end="&ON;"/>
  <constraint name="avionics master switch" start="&OFF;" end="&OFF;"/>
  <constraint name="fuel selector" start="&BOTH;" end="&BOTH;"/>
  <constraint name="fuel shutoff valve" start="&ON;" end="ON;"/>
  <constraint name="avionics power switch" end="&OFF;"/>
  <constraint name="circuit brakers" end="&ON:"/>
  <constraint name="elevator trim" end="0"/>
  <constraint name="rudder trim" end="0"/>
  <constraint name="ground speed" start="0" end="0"/>
  <constraint name="altitude" start="0" end="0"/>
 </constraints>
</situation>

<situation name="Start engine" timewindow="20">
 <actions>
  <phase name="starting-engine">
   <action name="fuel shutoff valve" priority="1" probability="BP">&OFF;</action>
   <action name="throttle" priority="1" probability="BP">&IDLE;</action>
   <action name="mixture" priority="1" probability="BP">&LEAN;</action>
   <action name="master switch" priority="1" probability="BP">&ON;</action>
   <action name="auxiliary fuel pomp" priority="1" probability="BP">&ON;</action>
   <action name="ignition switch" priority="1" probability="BP">&START;</action>
   <action name="ignition switch" priority="1" probability="BP">&OFF;</action>
   <action name="mixture" priority="1" probability="BP">&RICH;</action>
   <action name="auxiliary fuel pomp" priority="1" probability="BP">&OFF;</action>
   <action name="navigation lights" priority="1" probability="BP">&ON;</action>
   <action name="avionics master switch" priority="1" probability="BP">&ON;</action>
   <action name="beacon" priority="1" probability="BP">&ON;</action>
   <action name="radios" priority="1" probability="BP">&ON;</action>
   <action name="flaps" priority="1" probability="BP">&RETRACT;</action>
  </phase>
 </actions>
 <visualChecks>
  <instrument name="oil pressure"/>
  <instrument name="engine instruments"/>
  <instrument name="radios"/>
 </visualChecks>
 <constraints startProbability="MP" endProbability="BP">
  <constraint name="parking brakes" start="&ON;" end="&ON;"/>
  <constraint name="brakes" start="&OFF;" end="&ON;"/>
  <constraint name="avionics master switch" start="&OFF;" end="&OFF;"/>
  <constraint name="fuel selector" start="&BOTH;" end="&BOTH;"/>
```

```
    <constraint name="fuel shutoff valve" start="&ON;" end="&OFF;"/>
    <constraint name="avionics power switch" end="&OFF;"/>
    <constraint name="beacon" start="&OFF;" end="&ON;"/>
    <constraint name="strobes" start="&OFF;" end="&ON;"/>
    <constraint name="navigation lights" start="&OFF;" end="&ON;"/>
    <constraint name="elevator trim" end="0"/>
    <constraint name="rudder trim" end="0"/>
    <constraint name="ground speed" start="0" end="0"/>
    <constraint name="altitude" start="0" end="0"/>
   </constraints>
  </situation>

  <situation name="Taxiing to runway" timewindow="10">
   <actions>
    <phase name="taxiing">
     <action name="throttle" priority="1" probability="MP">!&0;</action>
     <action name="parking brake" priority="1" probability="MP">&OFF;</action>
     <action name="mixture" priority="1" probability="MP">&RICH;</action>
     <action name="flaps" priority="1" probability="MP">&RETRACT;</action>
     <action name="groundspeed" priority="1"
  probability="MP">&MAXTAXISPEED;</action>
    </phase>
   </actions>
   <visualChecks>
    <instrument name="brakes"/>
    <instrument name="directional gyro"/>
    <instrument name="turn coodinator"/>
    <instrument name="artifival horizon"/>
   </visualChecks>
   <constraints startProbability="MP" endProbability="BP">
    <constraint name="speed brakes" end="&>0"/>
    <constraint name="ground speed" start="0" end="&<20"/>
    <constraint name="throttle" start="&IDLE;" end="!&IDLE;"/>
   </constraints>
  </situation>

  <situation name="Before takeoff" timewindow="10">
   <actions>
    <phase name="before takeoff">
     <action name="parking brake" priority="1" probability="BP">&ON;</action>
     <action name="mixture" priority="1" probability="BP">&RICH;</action>
     <action name="fuel selector" priority="1" probability="BP">&BOTH;</action>
     <action name="elevator trim" priority="1" probability="BP">&SET;</action>
     <action name="throttle" priority="1" probability="BP">&1800;</action>
     <action name="throttle" priority="1" probability="BP">&1000;</action>
     <action name="strobe lights" priority="1" probability="BP">&ON;</action>
     <action name="landing lights" priority="1" probability="BP">&ON;</action>
     <action name="radios" priority="1" probability="BP">&SET;</action>
     <action name="autopilot" priority="1" probability="BP">&OFF;</action>
     <action name="flaps" priority="1" probability="BP">&TAKEOFF_SET;</action>
     <action name="parking brake" priority="1" probability="BP">&OFF;</action>
    </phase>
   </actions>
   <visualChecks>
    <instrument name="fuel quantity"/>
    <instrument name="magnetos"/>
    <instrument name="suction gage"/>
    <instrument name="engine instruments"/>
    <instrument name="caution panel"/>
    <instrument name="fuel flow"/>
```

```
    <instrument name="wheel brakes"/>
   </visualChecks>
   <constraints startProbability="MP" endProbability="BP">
    <constraint name="parking brakes" start="&ON;" end="&OFF;"/>
    <constraint name="wheel brakes" start="&ON;" end="&OFF;"/>
    <constraint name="ground speed" start="0" end="0"/>
    <constraint name="throttle" start="&IDLE;" end="<1000;"/>
    <constraint name="flaps" start="&RETRACT;" end="&SET_TAKEOFF;"/>
    <constraint name="altitude" start="0" end="0"/>
   </constraints>
  </situation>

  <situation name="Taking off" timewindow="5">
   <actions>
    <phase name="takeoff">
     <action name="flaps" priority="1" probability="BP">&SET;</action>
     <action name="throttle" priority="1" probability="BP">&FULL;</action>
     <action name="mixture" priority="1" probability="BP">&RICH;</action>
     <action name="airspeed" priority="1" probability="BP">&TAKEOFF_ROT_SPD;</action>
     <action name="elevator" priority="1" probability="BP">&TAKEOFF;</action>
     <action name="airspeed" priority="1" probability="BP">&TAKEOFF_CL_SPD;</action>
    </phase>
   </actions>
   <visualChecks>
    <instrument name="airspeed"/>
   </visualChecks>
   <constraints startProbability="SP" endProbability="VBP">
    <constraint name="airspeed" start="0" end=">70"/>
    <constraint name="throttle" start="&IDLE;" end="&FULL"/>
    <constraint name="altitude" start="0" end=">&MIN_ALT;"/>
    <constraint name="pitch" start="0" end=">&0" />
    <constraint name="climbing rate" start="0" end=">&0" />
   </constraints>
  </situation>

  <situation name="Enroute Climb" timewindow="5">
   <actions>
    <phase name="ascent">
     <action name="elevator" priority="1" probability="BP">&CLIMB;</action>
     <action name="throttle" priority="1" probability="BP">&FULL;</action>
     <action name="mixture" priority="1" probability="BP">&RICH;</action>
     <action name="mixture" priority="1" probability="BP">&LEAN;</action>
    </phase>
   </actions>
   <visualChecks>
    <instrument name="climb rate"/>
   </visualChecks>
   <constraints startProbability="SP" endProbability="VBP">
    <constraint name="air speed" start="&>70" end="&<85"/>
    <constraint name="Mixture" start="&RICH" end="&LEAN"/>
    <constraint name="throttle" start="&FULL;" end="&FULL"/>
    <constraint name="altitude" start=">&MIN_ALT;" end="&CRUISE_ALT;"/>
    <constraint name="pitch" start=">&0" end=">&0" />
    <constraint name="climbing rate" start=">&0" end=">&0" />
   </constraints>
  </situation>
```

```
<situation name="Cruise" timewindow="5">
 <actions>
  <phase name="cruise">
   <action name="elevator" priority="0" probability="BP">&0;</action>
   <action name="throttle" priority="1" probability="BP">&CRUISE;</action>
   <action name="elevator trim" priority="1" probability="BP">&ADJUST;</action>
   <action name="mixture" priority="1" probability="BP">&LEAN;</action>
  </phase>
 </actions>
 <visualChecks>
  <instrument name="flight instruments"/>
  <instrument name="engine instruments"/>
 </visualChecks>
 <constraints>
  <constraint name="air speed" start="&>85" end="!&>150"/>
 </constraints>

<situation name="Descent" timewindow="5">
 <actions>
  <phase name="descent">
   <action name="elevator" priority="1" probability="BP">&<0;</action>
   <action name="throttle" priority="1" probability="BP">&<2000;</action>
   <action name="mixture" priority="1" probability="BP">&RICH;</action>
   <action name="fuel selector valve" priority="1" probability="BP">&BOTH;</action>
  </phase>
 </actions>
 <visualChecks>
  <instrument name="climb rate"/>
 </visualChecks>
 <constraints startProbability="SP" endProbability="VBP">
  <constraint name="fuel selector valve" end="&BOTH;"/>
  <constraint name="mixture" start="&LEAN" end="&RICH"/>
  <constraint name="pitch" start="0" end="<&0" />
  <constraint name="climbing rate" start="0" end="<&0" />
 </constraints>
</situation>

<situation name="Before landing" timewindow="5">
 <actions>
  <phase name="approach">
   <action name="fuel selector" priority="0" probability="SP">&BOTH;</action>
   <action name="mixture" priority="1" probability="VSP">&RICH;</action>
   <action name="landinglights" priority="1" probability="VSP">&ON;</action>
  </phase>
 </actions>
 <visualChecks>
  <instrument name="flight instruments"/>
  <instrument name="engine instruments"/>
 </visualChecks>
 <constraints>
 <constraint name="landing lights" start="&OFF" end="&ON" />
 </constraints>
</situation>
```

```
<situation name="Landing" timewindow="15">
 <actions>
  <phase name="landing">
   <action name="airspeed" priority="0" probability="SP">&MAX_APPR_SPEED;</action>
   <action name="flaps" priority="1" probability="BP">&FLAPS_FINAL;</action>
   <action name="airspeed" priority="0" probability="SP">&MAX_LAND_SPEED;</action>
   <action name="flaps" priority="1" probability="VSP">&FLAPS_LANDING;</action>
   <action name="airspeed" priority="0" probability="SP">&LANDING_SPEED;</action>
   <action name="flaps" priority="1" probability="VSP">&FULL;</action>
   <action name="brakes" priority="1" probability="VSP">&ON;</action>
  </phase>
 </actions>
 <visualChecks>
  <instrument name="airspeed"/>
 </visualChecks>
 <constraints>
  <constraint name="air speed" start="&>110" end="&<20"/>
  <constraint name="flaps" start="&0" end="&FULL"/>
  <constraint name="brakes" start="&OFF" end="&ON"/>
 </constraints>
</situation>

<situation name="Taxiing to ramp" timewindow="10">
 <actions>
  <phase name="taxiing">
   <action name="throttle" priority="1" probability="BP">!&0;</action>
   <action name="mixture" priority="1" probability="BP">&RICH;</action>
   <action name="flaps" priority="1" probability="BP">&RETRACT;</action>
   <action name="groundspeed" priority="1" probability="BP">&MAXTAXISPEED;</action>
   <action name="landinglights" priority="1" probability="BP">&OFF;</action>
  </phase>
 </actions>
 <visualChecks>
  <instrument name="engine instruments"/>
 </visualChecks>
 <constraints startProbability="MP" endProbability="BP">
  <constraint name="ground speed" start="&>0" end="&<20"/>
  <constraint name="throttle" start="&IDLE;" end="!&IDLE;"/>
  <constraint name="flaps" start="&FULL" end="&0"/>
  <constraint name="landing lights" start="&ON" end="&OFF" />
 </constraints>
</situation>

<situation name="Securing Airplane" timewindow="5">
 <actions>
  <phase name="shutdown">
   <action name="parking brakes" priority="1" probability="BP">&ON;</action>
   <action name="avionics power switch" priority="1" probability="BP">&OFF;</action>
   <action name="mixture" priority="1" probability="BP">&CUT_OFF;</action>
   <action name="ignition switch" priority="1" probability="BP">&OFF;</action>
   <action name="master switch" priority="1" probability="BP">&OFF;</action>
   <action name="fuel selector" priority="1" probability="BP">&LEFT; || &RIGHT;</action>
  </phase>
 </actions>
 <visualChecks>
  <instrument name="flight instruments"/>
  <instrument name="engine instruments"/>
 </visualChecks>
 <constraints>
   <constraint name="parking brakes" start="&OFF;" end="&ON;"/>
   <constraint name="avionics master switch" start="&ON;" end="&OFF;"/>
```

```
    <constraint name="mixture" start="&ON;" end="&OFF;"/>
    <constraint name="ignition switch" start="&ON;" end="&OFF;"/>
    <constraint name="master switch" start="&ON;" end="&OFF;"/>
    <constraint name="fuel selector" start="&BOTH;" end="&LEFT;" || "&RIGHT;"/>
  </constraints>
</situation>
</flight>
```

(This page has been left blank intentionally)

Appendix D: Example of a backpropagation neural network

```
<?php
/*

NOTE: DO NOT RUN THIS THROUGH A WEB INTERFACE!

If you have no previous experience with Neural networks, I suggest you read some
basic descriptions and tutorials. Here is the resource I used to compose this script -
Please remember that I was a complete beginner when I started writing this script.
There may be errors and I still need to add momentum, memory banks, fuzzy logic,
and jittering.

Here are some definitions:

* -------- The Neuron --------
* Takes a number of inputs
* Multiplies each one by a 'weight'
* Sums all inputs x weights
* Applies an activation function to give an output.

* ----- A Neural Network -----
* Recognizes patterns and after training should be
* able to give reasonable predictions as to what the output should be.

* - A Backwards Propogation NN -
* Working out how far wrong the output is in its current
* state (the 'error'), and calculating a change in weights
* backwards through the network to correct this error
*     Output -> Hidden -> Input
*/

set_time_limit(0);
//This stops PHP from timing out on us

echo "\r\n";
define("LEARNING_RATE",0.5);
//The learning rate is a measure of how much the weights are changed in each
//training cycle.

class neuron {
  //These are all different factors of each neuron
  //Read up on Neural nets to find out what they mean
  var $bias;
  var $weights;
  var $output;
  var $delta;
```

```
function neuron() {
        $bias = 0;
  $weights[1] = 0;
  $weights[2] = 0;
  $output = 0;
  $delta = 0;
 }
}

class nn {
  var $hl; //Two neurons in the hidden layer
  var $ol; //One output neuron

  /*
  We end up with the following structure:
  A three layer backpropagation network

  Input --- Hidden
        X       >  Output
  Input --- Hidden
  */

  function nn() {
        //Initializing the net
        $this->hl[1] = new neuron;
        $this->hl[2] = new neuron;
        $this->ol = new neuron;

        for($i=1; $i <= 2; $i++) {
                $this->hl[1]->weights[$i] = 0;
                $this->hl[2]->weights[$i] = 0;
                $this->ol->weights[$i] = 0;
        }
  }
  function train($input1, $input2, $target) {
   for($i=1; $i <= 2; $i++) {
    $this->hl[$i]->output = $this->activation($this->hl[$i]->bias +
    ($input1 * $this->hl[$i]->weights[1]) +
    ($input2 * $this->hl[$i]->weights[2]));
   }
   //Find the current output for the Hidden Layer Neurons:
   //Output = Activation(Bias + Input[n] * Weight[n])

   $this->ol->output = $this->activation($this->ol->bias +
   ($this->hl[1]->output * $this->ol->weights[1]) +
   ($this->hl[2]->output * $this->ol->weights[2]));
   $this->ol->delta = $this->ol->output * (1 - $this->ol->output) *
   ($target - $this->ol->output);
```

```
  //The output neuron takes as its input the output from the two hidden
  //layer Neurons.
  //So for the output neuron weight(1) is the weight from HiddenNeuron(1),
  //and weight(2) is the weight for HiddenNeuron(2)

 /*
    Once we have the delta, it allows us to make an alteration to the
    weights in the network. The bigger the Delta, the larger the error
    in the network, and so the larger we want to alter the weights.
    This enables the network to become better after every training

    The above calculation of OutputNeuron->Delta first multiplies the
    output by (1- output). This has the effect of providing a larger
    figure when the output is at 0.5, and a minimum figure when the out
    put is at either 1 or 0 (do the math to confirm this). I.E. The Delta
    will be bigger, and so we're going to adjust the weight MORE when
    the current output is in the middle of the range (i.e. near 0.5). If
    the output is at either end of the range (i.e. at 1 or 0) then the
    Delta will come out smaller, and so we want to adjust the weight LESS.
    This simply has the effect of moving the weights more quickly if
    the current output from the Neuron is around 0.5 - the weight will
    be moved less if the neuron output is near 0 or near 1. (Bear in
    mind usually you'll want to get a more definite answer from
    a neural network - you want it to say 'Yesor 'No(i.e. 1 or 0)
    0.5 corresponds to 'Maybe', which is not a very useful answer.

    This figure is then multiplied by (Target - OutputNeuron->Output)
    This has the effect of making the delta LARGER if the error of the
    Neuron is larger.

    So overall this math says 'The Delta will be larger the nearer the
    Neuron output is to 1 or 0, and it will be larger the more wrong
    the Neuron is'.
 */


  for($i=1; $i <= 2; $i++) {
   $this->hl[$i]->delta = $this->hl[$i]->output * (1 - $this->hl[$i]->output) *
   ($this->ol->weights[$i] * $this->ol->delta);
   $this->hl[$i]->bias = $this->hl[$i]->bias +
   (LEARNING_RATE * $this->hl[$i]->delta);
   $this->hl[$i]->weights[1] = $this->hl[$i]->weights[1] +
   (LEARNING_RATE * $input1 * $this->hl[$i]->delta);
   $this->hl[$i]->weights[2] = $this->hl[$i]->weights[2] +
   (LEARNING_RATE * $input2 * $this->hl[$i]->delta);
  }

 /*
    These deltas are the ones for the Hidden Layer. The math is similar
    here except for the last factor. Remember the Delta for each Neuron
    is how much we want to correct it by, but for the hidden layer, we
```

```
    does not have a specific figure of precisely what we want the output
    to be, so the Delta has to be calculated by how wrong the Output
    Neuron was (which is its Delta) and the current weight from the
    Hidden Neuron to the Output one. As far as I can see, the current
    weight is included as a factor here to reflect how 'important that
    current weight is - the more important it is - i.e. the more its
    going to affect the Output Neuron, the more it should be altered.

    So now we have the delta for each Neuron - how much we want to change
    each Neuron's weights. So we'll use them to update the weights.

    See above how the Weight is altered by the Delta multiplied by the
    Learning rate - the larger the delta, and the larger the learning
    rate (which is a constant) - the more we're going to change each
    weight. But - the important part here is that we alter the weight
    of the Neuron also in terms of the INPUT. The larger the input was
    the more important this weight is to alter and so the more we're
    going to alter it by. - Bear this in mind when you look at how
    the weights for two neurons can start moving in the same direction
    initially and then change to moving in opposite directions - this
    is because of the Delta mainly being applied to a weight when
    there is a high input on that weight.
        */

  $this->ol->bias = $this->ol->bias + (LEARNING_RATE * $this->ol->delta);
  $this->ol->weights[1] = $this->ol->weights[1] +
  (LEARNING_RATE * $this->hl[1]->output * $this->ol->delta);
  $this->ol->weights[2] = $this->ol->weights[2] +
  (LEARNING_RATE * $this->hl[2]->output * $this->ol->delta);
  //And the same for the output neuron
}

function activation($value) {
        //The activation function is used to give us a value between 0 and 1
        return (1 / (1 + exp($value * -1)));
}


function runnetwork($input1, $input2) {
  //This takes the activation function of the sum of all
  //the inputs multiplied by their respective weights.

  for($i=1; $i <= 2; $i++) {
      $this->hl[$i]->output = $this->activation($this->hl[$i]->bias +
      ($this->hl[$i]->weights[1] * $input1) +
      ($this->hl[$i]->weights[2] * $input2));
    }
  $this->ol->output = $this->activation($this->ol->bias +
  ($this->ol->weights[1] * $this->hl[1]->output) +
  ($this->ol->weights[2] * $this->hl[2]->output));
  return $this->ol->output;
```

```
  }
}

//This creates an instance of our neural network class and trains it to output 1
//when it receives 0 and 0
//This shows how you can create your own logic gates that operate using artificial
//intelligence

$neural = new nn;
for($i=1; $i <= 6000; $i++)
  $neural->train(0, 0, 1);
print "Trained 6000 times to return 1 on 0, 0 (typical XOR logic)\r\n";
print "The system recalls " . $neural->runnetwork(0,0) . " from memory!";
?>
```

(This page has been left blank intentionally

Appendix E: Bibliography

[] CLIPS: A tool for building expert systems
http://www.ghg.net/clips/CLIPS.html

[] Cessna Aircraft Company (1996) *Pilot's operating handbook for a Cessna 172R*, 172RPHUS-02 Cessna Aircraft Company

[] Ehlert, P.M. and Rothkrantz, L. J. M. *Intelligent agents in an Adaptive Cockpit Environment,* Internal Report, Data and Knowledge Systems group, Faculty of Information Technology and Systems, Delft University of Technology

[] Flight Gear Flight Simulator
http://www.flightgear.org

[] JESS: The Expert System for the JAVA platform
http://herzberg.ca.sandia.gov/jess/

[] Mouthaan, Q.M. (2003) *Flying an F16: A knowledge base describing the situations an F16 pilot may encounter*, Technical Report DKS03-03 / ICE 03, Data and Knowledge Systems group, Faculty of Information Technology and Systems, Delft University of Technology
http://www.kbs.twi.tudelft.nl/Publications/Report/2003-Mouthaan-DKS03-03.html

[] Yourdon, E. (1991) *Modern Structured Analysis*, Prentice-Hall/Academic Service

[] Unknown. *Introduction to Backpropagation Neural Networks,* Cortex
http://cortex.snowseed.com/neural_networks.htm

[] World Wide Web Consortium
http://www.w3c.org/