Study Literature of
Model-based Testing For Test Integrity

Sanga Lawalata
Matriculation number :29446

Student Project

submitted in partial fulfillment of Master Program
in Information and Media Technologies

Supervised by
Prof. Dr. Ralf Möller

Software, Technology and System (STS)
Technical University of Hamburg Harburg
March 2006

Acknowledgments

First of all, I would like to thank Prof. Dr. Ralf Möller from Software, Technology and Systems (STS), Technical University of Hamburg-Harburg, for providing this topic of Student Project and thus, giving me chance to further discover about Media Based Testing.

I would like to thank Ralph Lembcke as my project supervisor in Airbus, for all his ideas, guidance, and encouragement through this project.

Lastly, I am grateful for all helpful suggestion and support from my friends during my work.

Table of Contents

Illustration Index

1 Introduction

System testing (hardware or software) goal is to find „the error“. It is important for a system to be tested in order to see if the system has already fulfilled the system specifications. Several testing methods are presented. They are based on the system characteristics and requirements.

In the beginning, system testers tested the system using „hands“ or manually. For software or hardware that is related with a computer system, it is possible to make automatic testing and let the computer change the role of the „human“ from “hands on” testing to writing „ testing scripts“.

While developing the system automatic testing, the system itself often starts growing and becomes more complex. Because of this the test scripts become obsolete and can't be used to test the system anymore without modification to be under taken. Maintaining test scripts is an important factor that must be considered but often maintaining test scripts, caused by system changes, consumes more time than is actually spend inside system testing.

The Airbus A380 Aircraft is the latest and the largest very-long-range, four engine subsonic commercial transport. It introduces new cabin functionalities which includes cabin automatic customization based on the cabin configuration. To test the cabin functionalities, automatic testing in the testing environment is performed.

1.1 The Motivation

To make sure full interoperability of cabin functionalities, automatic testing is performed in the testing environment. It is important to make sure before performing cabin functionalities testing that the testing environment is in a good condition so that the healthy checking script is needed. So that the system tester can have the the test result which are not effected by the testing environment. Health checking script will be needed to test the testing environment.

1.2 The Purpose

The purpose of this project is to present a literature research of the Model-based testing concept and discuss the possibility of its implementation within System Integration Test Bench Health Check. the implementation of this Model-based testing may offer automatic and dynamic system testing in the growing test environment.

1.3 Structure Of The Work

What is the idea of system testing (software and hardware) will be described briefly in chapter 2. This chapter also includes the basic overview, the ideas, how to implement, the advantages and disadvantages of Model-based testing.

Subsequently, chapter 3 will give the overview of System Integration Bench and how it works to perform cabin functionality testing.

Chapter 4 is bringing Model-based testing and SIB together, model generation, how the possibility automated automatic testing script is generated, results and the reason why it does go like it is expected before.

Chapter 5 is conclusion, derived from results in chapter 4.

2 System (software and hardware) testing

In System testing, it is important to know not only if the system does what it is supposed to do, but also if it does things it isn't supposed to do. The idea is not to find that the tested system is working without error but to find the system error or misbehavior. This is because no error found can be „there is no error“ or „the errors that haven't been found yet“.

It is important because finding the error as soon as possible will produce more mature systems and save lots of resources. There are many methods to test a system. Usually the method is chosen based on the characteristics of system under test.

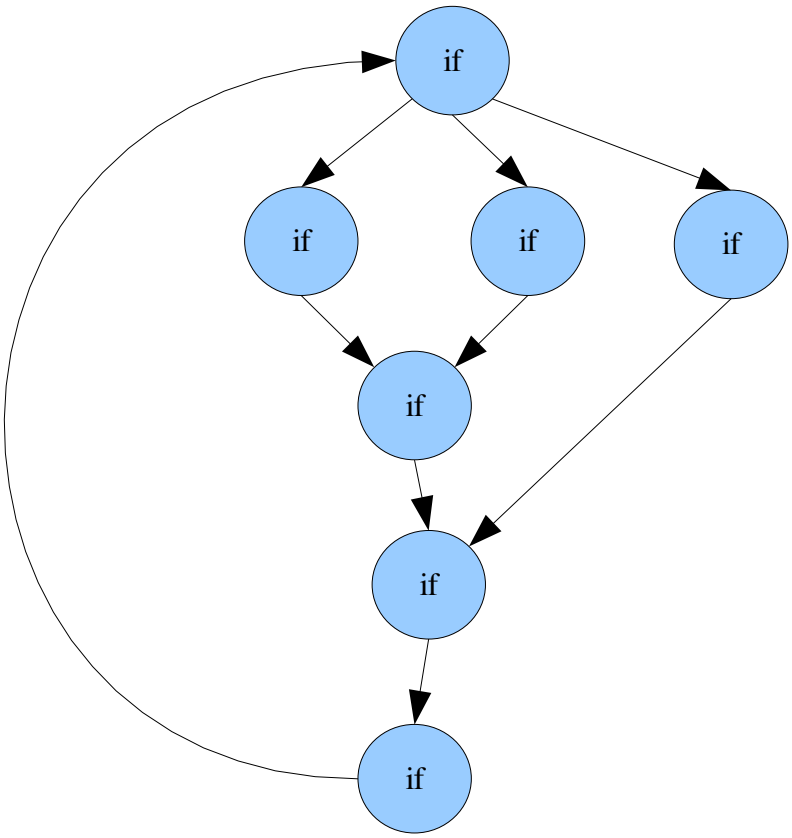
Considered a whole system as a big modules collection, there is no single testing method that can be applied completely in one system and will „answer“ all the test requirements. If it is needed, several testing methods can be combined to test „one big system“. This means that it is possible to use several testing methods to test several modules in „one big system“.

Another idea which often appears in system testing is “exhausted testing”. It means to try to put all “possible load or input ” to the system.

2.1 White Box Testing

White box testing is „also know as a glass box, structural, clear box, and open box testing“ [14]. White box testing often is called unit testing. A unit is commonly a small part of the system and is defined by a system developer. For an example in software, a unit can be a single procedure or functions or maybe the whole program can be a unit, when nested in a bigger software system. The system testers know the „internal process of the system under test“ which later is used to choose the set of input data. In the software development, programmers can take over the unit testing because they know the internal process of the unit so that they can choose a set of input data to test. This will reduce the size of the data set used for input, just focus on the specific data which will (hopefully) affect the unit. The drawback (still in the same example) is the programmer tends to prove or test that the unit or program is working which is called human psychology factor. Because the programmer, psychology, is connected with the unit that they produced. A external programmer who has at least the same knowledge but was not involved in developing the unit, should perform the unit test to find the error.

Knowing the internal process of the unit doesn't guarantee that unit tester can perform “exhausted testing” even though the system testers have defined a limited set of data instead of “whole data”. The time to test “all internal process“ such as branching, looping etc will grow exponentially.



Drawing 1: Sequential If Branching Illustration

For example, look at the picture above, it reflects a sequential process in one software. Each circle reflects branching or „if state“. In this example testing all “if state” is “the testing requirement”. A testing will run 20 times, total time to test each branching or circle is 5 minutes. The total test time is $\Delta t = 5^{20} + 5^{19} + 5^{18} + \ldots + 5^{0}$ if the decision (if state) is independent [10]. At the end the white box testing can't guarantee that all the „possible path is tested“. Time to perform testing will grow exponentially based on the testing requirements.

2.2 Gray Box Testing

Gray box testing combines white box techniques with black box input testing [7]. Because system testers don't know all internal process of system under test, they use the partial known knowledge to test the system. In general, many system under tests are a „gray box environment“. Usually complex system, which consist of lots of smaller black box modules, can form a gray box system.

2.3 Black Box Testing

Black box testing is „also known as *functional testing*. A software testing technique whereby the internal workings of the item being tested are not known by the tester“ [3]. It can be called verification test. System testers know the expected output and will evaluated the system output to decide if the system meets the requirements. The system testers just supply input to the system and evaluated the output of the system. They don't know the internal process of the system (completely hidden). At the end the system testers will evaluate if the system has met its requirement by comparing the measured result and the system specification.

One of the advantages is that the system tester doesn't need to know the internal „processes“ of the black box system. He/she can design the test case as soon as specifications are completed because the system specification defines the set of input data and which output data they will produce.

Black box testing has a few drawbacks. Because „Testing is the process of executing a program with the intent of finding errors“ [10], there is an idea to use „exhaustive input testing“ which means to use all possible inputs to find the error not only using a defined input out of the specification. Instead of to verify that the system provides “correct behavior” based on defined input, the system must be verified also that it doesn't do anything unwanted for sets of “undefined input”.

The first drawback using the „exhaustive input testing“ in black box testing is the system testers should input not only „the valid inputs“ but also „the wrong inputs“. This results in testers must supply „all possible input“ which will soon result in „infinite number of input“ which is impossible to execute. The second drawback is the system testing environment must be considered as a factor that will effect the result of back box testing. There is no system that is actually independent from its environment. The system tester must distinguish if there is an error, it is caused by the system under test or

testing environment. It is important when performing black box testing, system tester must be sure that the testing environment is ready to test the system under test and doesn't affect the system under test.

2.4 The Model-Based Testing

„Model-Based Testing is the automatic generation of efficient test procedures/vectors using models of system requirements and specified functionality.“ [6].
The idea behind Model-based Testing is automatic system testing based on a model. What is a model is explained in section 2.4.1, how to build the model and how to generate automatically the test script will be explained in section 2.4.2 .

2.4.1 Model

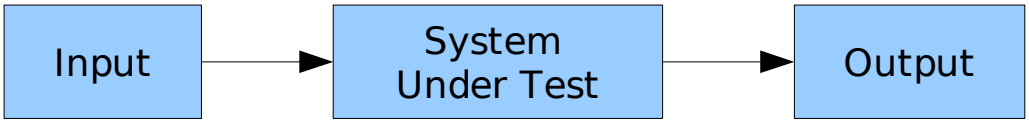
A system model is a „copy“ or a mimic of the systems behavior. „Behavior can be described in terms of the input sequence accepted by the system, the action, conditions, and output logic, or the flow of data through the application's modules and routines“ [6]. A model can hide the system complexity, internal processes and later can be used to analyze the system structure.
Hiding the system complexity means either system tester and automatic script generator see the system as a collection of models. They test and manipulate the model, but they don't know the internal process of the model. Taken a lamp switch for example, from system testers point a view, they know and see only a switch that will turn on or off if they push the button. But they don't know internally how the switch works, e.g. how it looks in the on or off state. The switch hides the internal specific process. Hiding the system complexity and internal specifics how it works, replaced by a model makes its possible to create the what is called a “framework” or a standard. Two systems with the same behavior can share the model, even though the internal process details may be different. Different switches have their on internal process, but they share the same action, *turnOn()* (defined as a method to show an action) and *turnOff()*. The implementation of the *turnOn()* and *turnOff()* method will be specific dependent on the switch company. Each switch company can defined their own specific internal process to turn on and turn off switch.
There is no model that can present entirely all one system because a big system is often to complex to be described by one model only. Complex system can be broken down into smaller model that are connected each other. Smaller model is much more easy to

build and tested. Based on the system characteristic and testing requirements, the model (or collection of models) is chosen.

2.4.2 Building the model

To define a model of the system under test, the steps are :

1. Based on the accepted input sequence, the action and output logic, The System Under Test (SUT) can be defined by a set of states and state transitions which are triggered by inputs.



Drawing 2: General Overview Of System Under Test

Expected SUT behavior can be described by a transition table, consisting of starting state (previous state), action (input), and ending state (current state).

Starting State (Previous State)	Action (Input)	Ending State (Current State)
State1	input1	State2
State2	input2	State3
State1	input3	State3
State3	input1	State3

It is possible to make a model of „an action“ or a SUT behavior in form of a transition function. Within a transition function definition „When this action is possible“ is a “*starting state*”, „action“ is “input”, and „What the outcome of executed action“ is a “*ending state*”.

input -> Transition Function -> output

$F_T : (s_{0,} i_0) \rightarrow (s_1)$

$s_{0,} s_1 : states$

$s_{0,} s_1 \in S$

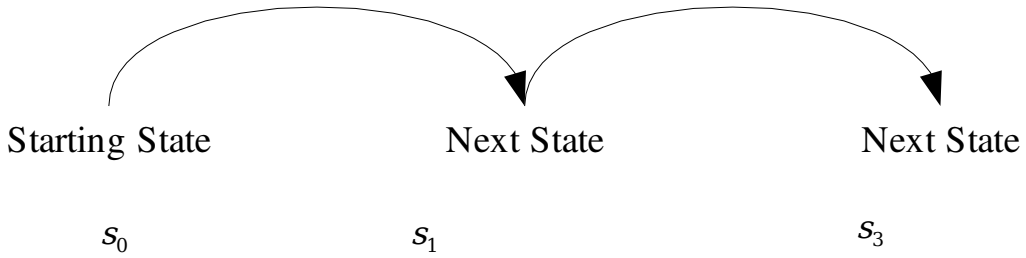
$i_n : Input$

$i_n \in I_n$

$S = set\ of\ all\ states$

$I = set\ of\ all\ input$

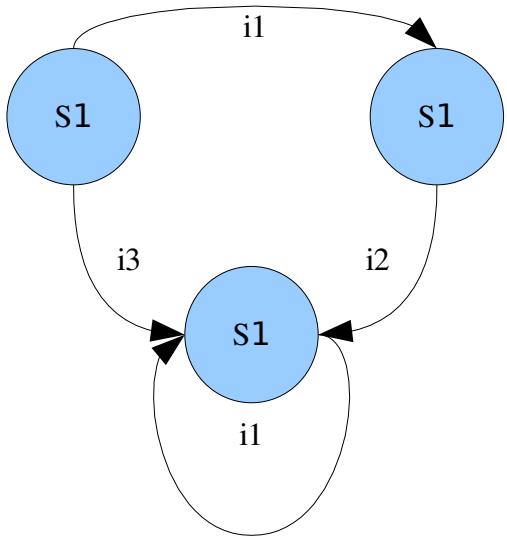
A set of actions with sets of starting states and next state can be illustrated as a direct graph :



2. When „sequencing“ the „current state“ through the different states by applying defined inputs, the inputs used define the input sequence ($i_1, i_2, i_3, i_4, \ldots, i_n$).
3. each state S_n is assigned a set of valid input symbols as dictated by SUT specification. Input symbols that are allowed to apply when in the simulation state n is i_n , for example $I_{s_1} = i_1, i_3$, $I_{s_2} = i_2$
4. Let the computer search the most possible path through the direct graph. Possible means to reach all states which are connected with a set of inputs and no connected states aren't visited. Time, cost, computer times are things that must be considered when selecting the path. Random walk is the easiest way (and most often poorest) to search the possible path. But it is seldom leaving few states untouched (without being covered) or maybe facing „looping“ that makes the total of time for testing longer. For „a bigger or complex model“ with lots of states and possible input combinations, random walk is not efficient anymore.
   One of the efficient algorithm that is currently used is „the Chinese postman walk“. “ ... how to find a shortest closed walk of the graph in which each edge is traversed at least once, rather than exactly once. In graph theory, an Euler cycle in a connected, weighted graph is called the Chinese Postman problem.” [4]. With this algorithm, the computer will search all possible paths with all the nodes being visited at least once.

5. Later there is „an efficient walk“, based on „the Chinese Postman Walk“ which eliminates the actions that don't change state.

Previous State (Starting State)	Input (or Action)	Current State (Ending State)
State1	input1	State2
State2	input2	State3
State1	input3	State3
(State3)	*(input1)*	*(State3)*



Drawing 3: Transaction State II

6. Input, in general point of view, is something that can trigger a change of state. „Input implementations“ means to make a „method“ which can be called by the test generator. Define the „method“ to generate input. Later the test script generator puts this method inside the generated test script to generate input.
   No error doesn't mean that the system is free from error. There are two possibilities. One is there is error free, the other is no error has been found yet.

System tester can be „more subtle where there is no error", which means :

1. During the testing, automatically generated test scripts can report to the tester that one functionality is not found yet or not yet implemented by the system under test.
2. Knowing the last state value, tester can create the „test routine" to check if the value is either expected or unexpected.

3 CIDS-SIB

3.1 CIDS-SIB Overview

The Cabin Intercommunication Data System(CIDS)-System Integration Bench (SIB) is a platform that supports the testing of the CIDS, which is the cabin core system for illumination control, public address and many other functions.

The CIDS-SIB is divided into 2 sub systems, which are :

1. Simulation of the aircraft electronic bay related systems.
2. Simulation of the cabin-related systems.

Besides these two main systems, there are some auxiliary units like power supplies. To run all aircraft parts of CIDS, there are power supplies that supply 28 V DC and 115V AC. It has capabilities to simulate over-, under-voltages and power interruptions. All these simulation systems serve operating the CIDS, which is the System Under Test.

3.2 CIDS Overview

CIDS consists of :

- Three redundant main computer (Directors)
- One or more Flight Attendant Panels (FAP)
- The Data Buses (two types – Top and Middle Lines)
- The Decoder Encoder Units (DEU-A and -B)

AFDX, Ethernet, and CAN are protocols used for message transfer between the Directors and other aircraft systems or the simulation system respectively.

All the aircraft devices, which are installed on an aircraft inside the cabin, connect to CIDS through several Decoder-Encoder Units. The simulation system that substitute these devices on the SIB is called DEESi, which is an abbreviation for DEU Electrical Environment Simulation (DEESi). The DEESi system simulates all original cabin parts and also monitors the data exchange between the Decoder-Encoder Units and the original equipment parts if installed.

Based on the Airbus documentations, several tools to monitor CIDS operation are available to the tester

- ZOC : is a Debug and Maintenance interface on the Director. As explained later, this port will be used to trigger „some actions“ at the DEUs independent from the „cabin configuration“.

- DEESi system : can be used as a monitor to read or to set the current state or value of specific simulators. These states relate to the state of the outputs and the DEUs and the history of the state's sequence.
- fdXplorer : to monitor the AFDX message traffic. The monitored messages are in „raw message in a duration time“, to be analyzed during CIDS testing.
- CIDS-Bus-Simulator : to monitor internal traffic message between the directors and the DEUs.
- CANalyzer : to monitor and analyze CAN bus systems.

3.2.1 Director

The Director is „a special computer system“ that deals with all cabin functionalities. The Director is connected to the original equipments through DEUs. Inside the 3 redundant directors, there is „a cabin configuration function“ with controls the PA distribution, illumination control according to the cabin layout, i.e. number of set rows, classes, class boundaries etc.

3.2.2 Decoder Encoder Units (DEUs)

Decoder Encoder Units (DEUs) are interfaces between the CIDS Buses and the cabin devices. they decode and format data from the director to the cabin devices and vise versa encode data received from cabin devices and send it to directors.

There are 2 type of DEUs servicing the following sets of equipment :

- DEU-A
  - Up to 8 Illumination Ballast Units (IBU)
  - Up to 8 Passenger Interface and Service Adapter (PISA)
- DEU-B
  - Up to two Handsets
  - Up to two Attendant Indication Panels (AIP)
  - Up to two Area Call Panels (ACP)
  - Up to two Additional Attendant Panels (AAP)
  - an Emergency Power Supply Unit (EPSU)

- an Ice Protection Control Unit (IPCU)
- several smoke sensor (interconnected via a CAN Bus).

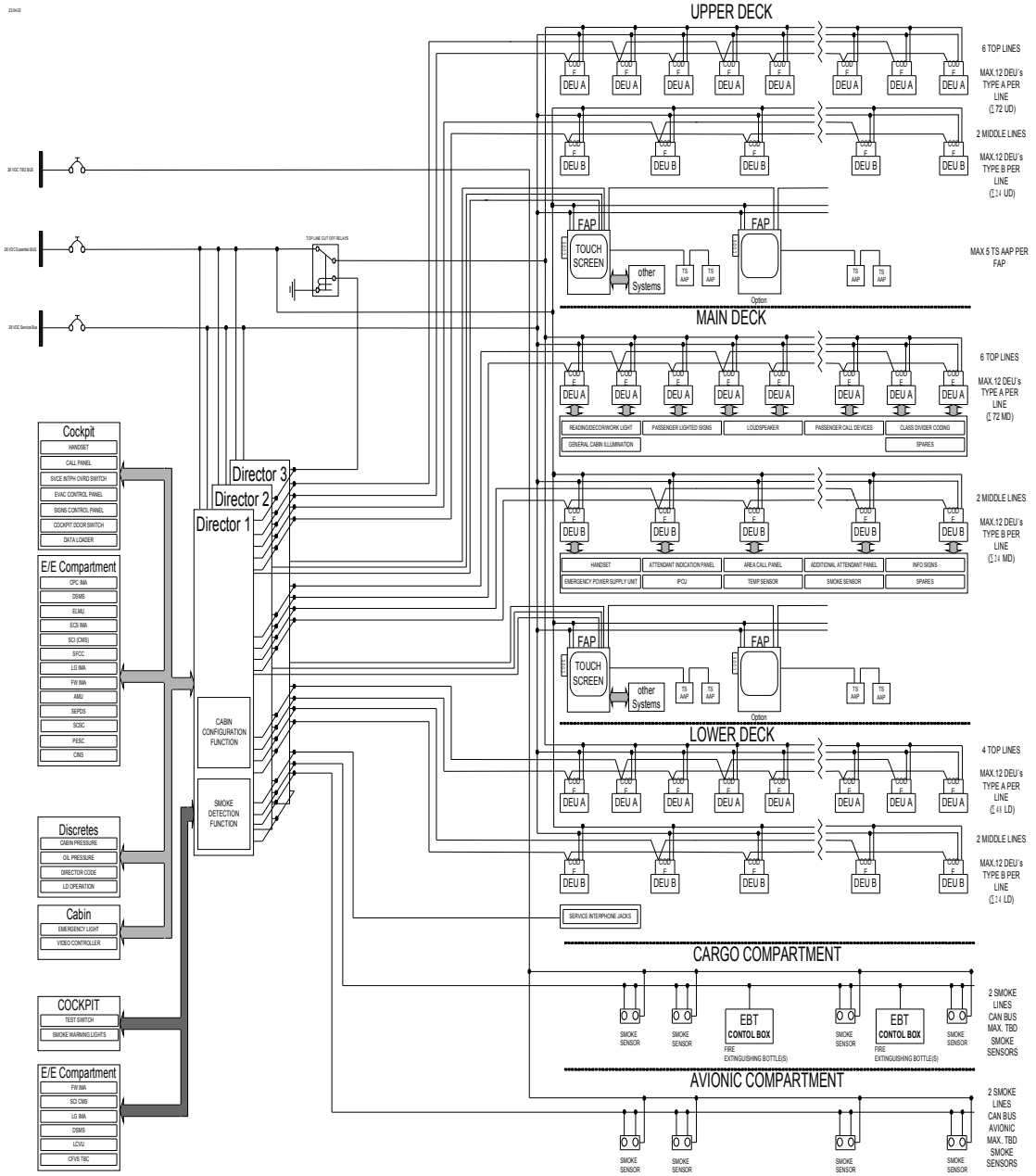
DEUs need 28 V normal and 28 V essential power supply.

3.2.3 Passenger Interface and Service Adapter (PISA)

Connected to each PISA are :

- a loudspeaker
- Signs (Fasten Seat Belt – FSB, No Smoking – NS, Return To Seat – RTS)
- Up to four reading lights, also used as decor, spot and attendant work lights
- Up to two passenger call lights/switches (PAX call)

A single PISA needs 115 V and additional 28V sourced by the DEU.



Drawing 4: CIDS-SIB Testing Environment

3.3 Cabin Functionality Testing Overview.

Before running any test, the tester will load a DEESi configuration to set the initial values of all simulations. Based on the test case, the tester will „change the state of simulators“ through display and command programs to create stimulus to the SUT CIDS. One of these programs is the ASCIIBanz terminal. The director will then „react“ and change the CIDS system state, based on the „cabin configuration“ which will trigger subsequent state change on the simulators. Different „cabin configurations“ will result in different „current states of CIDS and simulators respectively“. The current state of CIDS and simulators can be read using i.e the ASCIIBanz again. Later after testing, all the „current states of the simulators“ will be analyzed based on the „system specification“ to judge whether the test case is failed or passed.

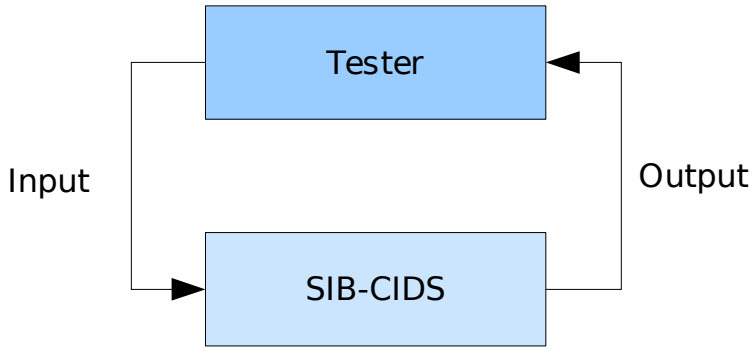
Using the ASCIIBanz, the tester can run „automatic testing“. The tester will make and run scripts which will produce a log file for later analysis.

4 Analysis and Design

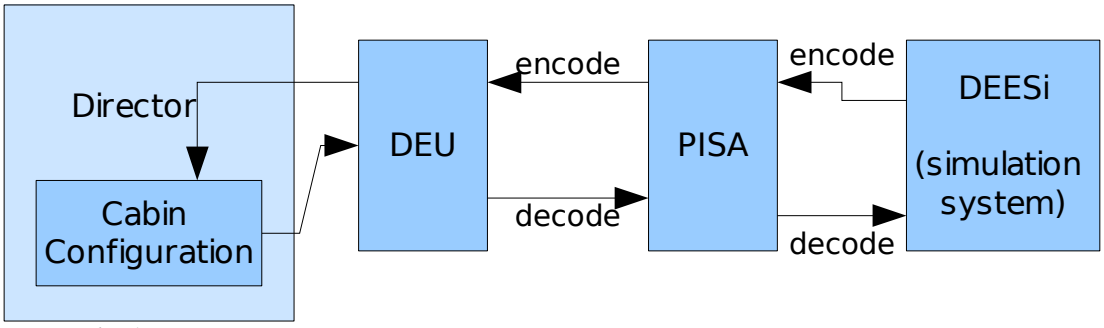
4.1 What is System Health Check

Before running any test, the tester needs to be sure that the CIDS-SIB is in „healthy condition“. „Healthy condition“ means at least there are no hardware defects that later would affect CIDS testing result. For example : a tester, based on a test case, wants to turn on a FSB LED. FSB can be set i.e. by setting the directors FSB-Cockpit switch to „on“ position. If the FSB LED isn't turned on, there are two possible reasons : that the LED is broken or the LED isn't assigned in „the cabin configuration“.

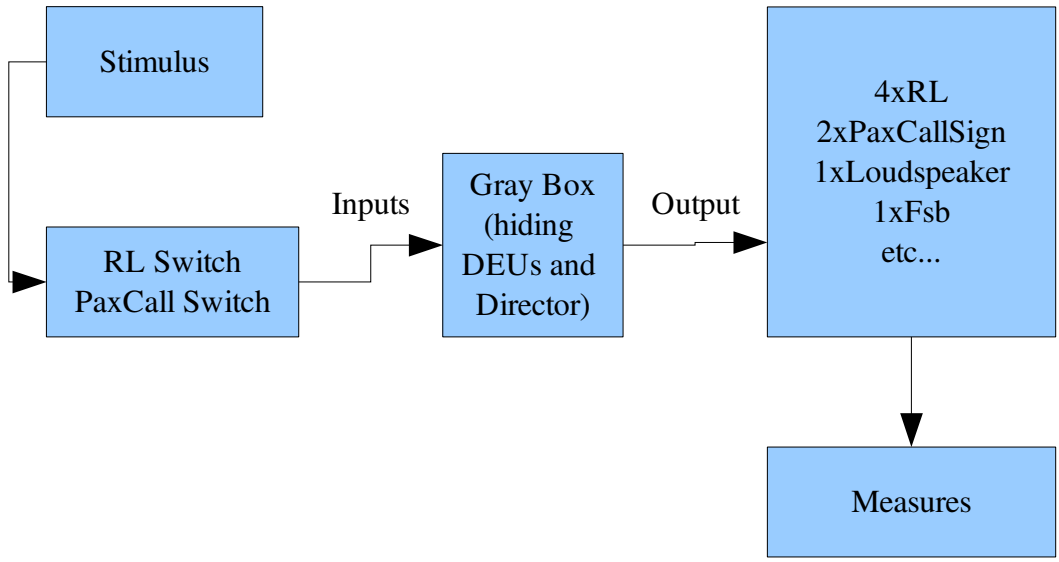
4.2 System Testing Overview



Drawing 5: General Overview of SIB-CIDS



Drawing 6: The CIDS-SIB Environment

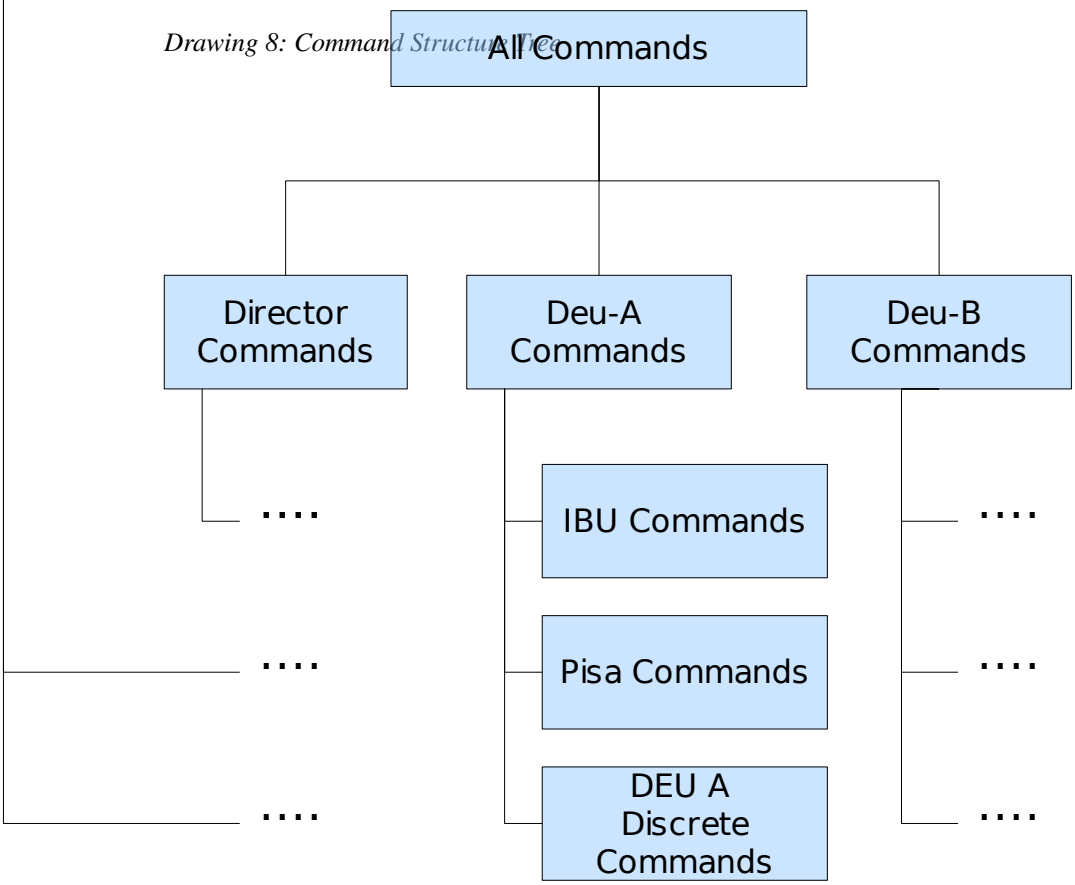


Drawing 7: DEESi Environment

4.3 Director

The director is considered as a „gray box system“. It has a cabin configuration system inside. The Director, as explained before, is “a device” that deals with all cabin functionalities.

As explained before that director has a ZOC, a debug and maintenance interface. In the picture 8, it defines the structure of all commands to control all devices connected to CIDS. There are commands for Director, DEU-A and DEU-B. Because the PISAs are attached in the DEU-A devices, their commands becomes part of the DEU-A commands.



Drawing 8: Command Structure Tree

The following table shows all commands r to control all connected equipment through a PISA based on documentation internal AIRBUS Documentation[8].

NO	Commands	Devices
1.	pcl1	paxCall light led button 1
2.	pcl2	paxCall light led button 2
3.	fsb	fasten seatbelt led
4.	fsba	external fasten seatbelt led
5.	ns	non smoking led
6.	nsa	external non smoking led
7.	rls	reading light sign
8.	rldim	reading light dim
9.	rlm	reading light mode
10.	audon	audio on

NO	Commands	Devices
11.	audvol	audio vol
12.	audch	audio channel
13.	28vEss	power supply

This table shows what DEUs need to be operable

NO	Power Requirements
1.	28vNorm power supply
2.	28vEss power supply

This table shows what PISAs need to be operable.

NO	Power Requirements
1.	28vNorm power supply
2.	115v power supply

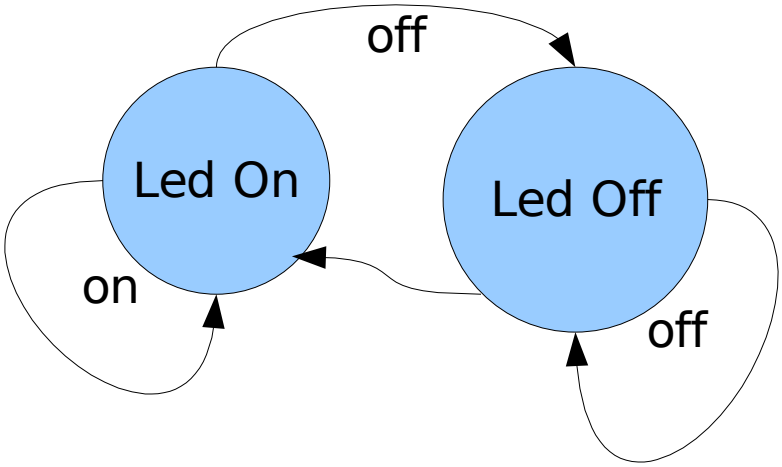
4.4 Making the model

4.4.1 Model the system behavior

The Directors, DEUs, PISAs, and simulator equipments are considered as one big gray box system. All the Director inputs are considered as an action that will change its states.

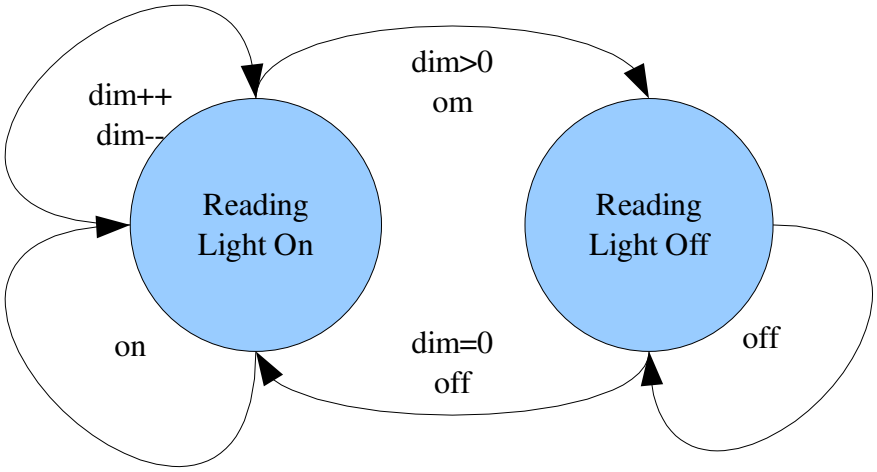
System tester need to know that all LEDs are working. So that in this circumstance, “LED working” is the behavior system which is to be modeled.

Let's say the first action (system behavior) related with Passenger Call Button Led is defined as *PaxCallLedWorking*. Inside *PaxCallLedWorking*, there are 2 different actions which are *turnOnPaxCallLed* and *turnOffPaxCallLed*.



Drawing 9: PaxCall Led State Diagram

The second action is *ReadingLightWorking*. Inside *ReadingLightWorking*, there are 2 different actions which are *turnOnReadingLightLed* and *turnOffReadingLightLed*. *ReadingLightLed* has another value which is dim value. The reading light can be turned on but it isn't visual on because the dim value is zero or in small numbers.



Drawing 10: Reading Light State Diagram

Additionaly *addDimRlVal* and *subtrDimRlVal* actions are defined to add and substract the reading light dim value and *readDimRlVal* to get the current value of dimming. Until now these are the list of modeled actions :

- *PaxCallWorking()*, consists of *turnOnPaxCallLed()* and *turnOffPaxCallLed()*.

Starting State	Action	Ending State
PaxCall_1_Off	turnOnPc1	PaxCall_1_On
PaxCall_1_On	turnOnPc1	PaxCall_1_On
PaxCall_1_Off	turnOffPc1	PaxCall_1_Off
PaxCall_1_On	turnOffPc1	PaxCall_1_Off

ReadingLightWorking(), consist of *turnOnReadingLight()*, *turnOffReadingLight()*, *addDimRlVal()*, *subtrDimRlVal()* and *readDimRlVal()*.

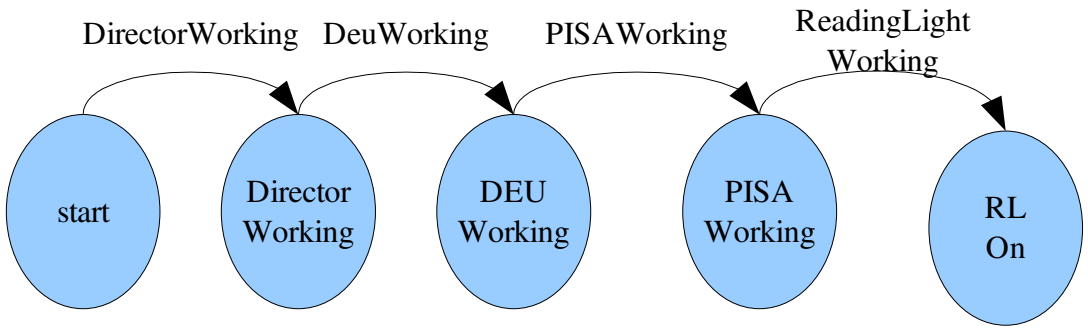
Starting State	Action	Ending State
RLS_On	turnOnRls	RLS_On
RLS_Off	turnOnRls	RLS_On
RLS_On	turnOffRls	RLS_Off
RLS_Off	turnOffRls	RLS_Off
RLS_DIM	addDimRlVal	RLS_DIM+1
RLS_DIM	subtrDimRlVal	subtrDimRlVal-1

- *AudioWorking()*, consist of *turnOnAudio()*, *turnOffAudio()*, *incVol()*, *decVol()*, *getVol()*, *setAudioChannel()* and *getAudioChannel()*.

Starting State	*Action*	*Ending State*
Audio_Off	turnOnAudio()	Audio_On
Audio_On	turnOnAudio()	Audio_On
Audio_Off	turnOffAudio()	Audio_Off
Audio_On	turnOffAudio()	Audio_Off
Audio_Vol	incVol()	Audio_Vol+1
Audio_Vol	decVol()	Audio_Vol-1

There is one additional action which is PisaWorking() which consist of turnOn28VEss, turnOff28VEss, turnOn115V and turnOff115V because the PISAs need 115v supply as well, and it can be controlled through DEESi.

Starting State	Action	Ending State
28VEss_Off	turnOn28VEss()	28VEss_On
28VEss_On	turnOn28VEss()	28VEss_On
28VEss_Off	turnOff28VEss()	28VEss_Off
28VEss_On	turnOff28VEss()	28VEss_Off
115V_Off	turnOn115V()	115V_On
115V_On	turnOn115V()	115V_On
115V_Off	turnOff115()	115V_Off
115V_On	turnOff115()	115V_Off



Drawing 11: The System Health Checking Sequence

Based on the drawing 6, if the SIB-CIDS is regarded as a gray box system, it consists of many „black box“ parts like the DEUs, and the Directors themselves. For both devices, they need to be „ready“ or „checked“ if they are internally working, because faulty Directors will affect DEU functionally which in turn affects other sub systems of the CIDS-SIB. So the tester must check that DEUs and Directors are working. This behaviors must be modeled and later the test script generator can perform test.

Based on drawing 11, there are 4 system behaviors that must be modeled as actions, which are PisaWorking (is already defined), DeuWorking, and DirectorWorking. The same steps are applied like with modeling the PisaWorking.

The first step is to model the system behavior which is DeuWorking as an action. In this case, because the limitation of the documentation, it is assumed that DeuWorking is a condition when DEU is powered up so that it is in operational mode.

- *DeuWorking()*, consisting of *turnOnDeu28VEss()*, *turnOffDeu28VEss()*, *turnOnDeu28VNorm()*, and *turnOffDeu28VNorm()*.

Starting State	Action	Ending State
Deu_28V_Ess_Off	turnOnDeu28VEss()	Deu_28V_Ess_On
Deu_28V_Ess_On	turnOnDeu28VEss()	Deu_28V_Ess_On
Deu_28V_Ess_Off	turnOffDeu28VEss()	Deu_28V_Ess_Off
Deu_28V_Ess_On	turnOffDeu28VEss()	Deu_28V_Ess_Off
Deu_28V_Norm_Off	turnOnDeu28VNorm()	Deu_28V_Norm_On
Deu_28V_Norm_On	turnOnDeu28VNorm()	Deu_28V_Norm_On
Deu_28V_Norm_Off	turnOffDeu28VNorm()	Deu_28V_Norm_Off
Deu_28V_Norm_On	turnOffDeu28VNorm()	Deu_28V_Norm_Off

- DirectorWorking(), consisting of *turnOnDir28VEss()*, *turnOffDir28VEss()*, *turnOnDir28VNorm()*, and *turnOffDir28VNorm()*

Starting State	Action	Ending State
Dir_28V_Ess_Off	turnOnDir28VEss()	Dir_28V_Ess_On
Dir_28V_Ess_On	turnOnDir28VEss()	Dir_28V_Ess_On
Dir_28V_Ess_Off	turnOffDir28VEss()	Dir_28V_Ess_Off
Dir_28V_Ess_On	turnOffDir28VEss()	Dir_28V_Ess_Off
Dir_28V_Norm_Off	turnOnDir28VNorm()	Dir_28V_Norm_On
Dir_28V_Norm_On	turnOnDir28VNorm()	Dir_28V_Norm_On
Dir_28V_Norm_Off	turnOffDir28VNorm()	Dir_28V_Norm_Off
Dir_28V_Norm_On	turnOffDir28VNorm()	Dir_28V_Norm_Off

Starting State	Action	Ending State
Director_Not_Ready	DirectorWorking()	Director_Ready
Director_Ready	DeuWorking()	Deu_Ready
Deu_Ready	PisaWorking()	Pisa_Ready
Pisa_Ready	AudioWorking()	Audio_Working
Pisa_Ready	ReadingLightWorking()	RL_Working
Pisa_Ready	PaxCallLedWorking()	PaxCall_Working

And the tables above is the summarize of the modeled behavior based on the drawing 11.

4.4.2 Script generator

The script testing generator will automatically generate the test script based on modeled CIDS-SIB behaviors. All the tables above define the CIDS-SIB behavior that should be tested before performing cabin functionalities testing. Based on that list, the script generator will generate a test script to test all the components inside the CIDS-SIB. Based on drawing 11, if the test script generator wants to a generate test script, it should put in order from *DirectorWorking()*, *DeuWorking()*, *PisaWorking()*, and later *AudioWorking()*, *ReadingLightWorking()*, and *PaxCallLedWorking()*.

In this case, the test script generator doesn't have to find the shortest path because there is just one path available because the devices such as Directors, DEUs, PISAs are tightly coupled. DEUs needs the directors to be ready before it can be in „*readyCondition*“. It looks like the boolean state, if all condition are true (Director,DEUs,PISAs) then *AudioWorking()*, *ReadingLightWorking()*, *PaxCallLedWorking()* can be performed.

All the actions, listed above, are an abstract idea of actions. Its implementation can be implemented in any programming code.

The test script generator will call the action implementation as a method. It is up to testing strategic how to read back all the data from the devices. It can be put in the method directly or it can be done by other system. For example the method *turnOnReadingLight()* can be designed like follows :

```
turnOnReadingLight(){
.  // turn on the readingLight
.  // read the reading light value from measurement
.  // print „Do you see the light on ? “
}
```

The test script will be generated by the test script generator. The test script will turn on all the led and the sound. it reads all the data through measurements [drawing 7]. And the test script will compare the expected result with the actual result. In some occasion human is still needed to watch the actual result. For example, turning on Led, it must a human to watch if the Led works. The sound can be measured by measurement and in the same time, listened by human.

But the audio testing has different point of view compared with PaxCall button led testing. It has not only true or false value but also volume and audio channel. The script generator at this point can generate test script much precisely when testing PaxCall button led. It not only can turn on the audio function but also can read the current value of volume. It is possible that audio is working but the tester don't hear the sound because the volume is 0. The test script can help the system tester better understanding of the current system condition. It can help the tester to distinguish that the sound is not coming because it is broken or because the volume is 0.

4.4.3 Running the test script

The test script will contain these methods :

- DirectorWorking()
- DeuWorking()
- PisaWorking()
- ReadingLightWorking()
- turnOnReadingLight()
- turnOffReadingLight()

- readDimRlVal().
- PaxCallLedWorking()
- turnOnPaxCallLed()
- turnOffPaxCallLed()

In the reading light test, the test script is able to know the “precise” state of the reading light, not only just turning on of off reading light.

4.4.4 Analysis of the Implementation

In this case, the Model-based testing can be applied well. Characteristics of the system, which are mostly „hardware“, can't be modeled easily. It needs lots of information how the „hardware works“ so that it can be modeled more precisely.
Lack of information limits modeling the system behaviors. For example :
„*DeuWorking*“ is described by turning on DEU normal and essential power supply. It is supposed to be other influencing factors to be described in *DeuWorking*. These factors are later described as states which could be dependent one to another. Dependent actions are connected through current states (before the action) and future states (after the action). The connected actions will create “path”. The test generator will create test script that try to test all possible path.
The CIDS-SIB testing environment stability is an important factor that must be considered. The generated health checking script can produce different results because the system instability. In this case the health check script is able to show there is something wrong and the current several simulator values (such as dim values) but can't conclude either a testing environment instability or hardware failures. At this point, human is still needed to evaluate the test result.
health checking script run times must be as efficient as possible. because the main focus of the CIDS-SIB testing environment is to perform cabin functionalities testing. It means the health checking can't check all testing environment equipments because it will consume time. The critical testing requirements for CIDS-SIB health checking script must be define.

5 Conclusion

System testing is important to find the error before the system is used. Finding an error or malfunction earlier will assist in producing mature product and save the effort to fix it in the future. Meanwhile, the system under test itself continuously grows and becomes more complex, so automatic testing is desired.

Model-based testing concept tries to answer to how to make automatic testing can be made in the growing systems. Automatic testing shifts most of the work to be done by computer.

System testers need to be sure that Cabin Intercommunication Data System-System Integration Bench is ready to perform tests when the CIDS-SIB itself is complex, has lots of features and still growing.

Model-based testing cant be applied well enough on CIDS-SIB because of the difficulty in modeling the system behaviors. Deeper understanding of the system testing environment is needed to more precisely model, and this understanding can be obtained from documentation and the system tester.

Model-based testing is more effective to test system behavior that shares the same preconditions and dependent from one behavior to other behaviors because it is the most "difficult part" if it is done by human. In Model-based testing, it will done by computer.

Modeling system behavior simplifies the way system tester observe the system. They don't have to know the details of system under test; they just have to know the behavior that they want to test. If there is additional feature or behavior that need to be tested, it will be added to the model or a new model can be created without altering the previous ones.

The amount of time to perform the test must be considered. In this case, testing the test environment is not the main goal of the CIDS-SIB. Instead of testing all the possibilities but “limited defined possibilities” are given to ensure the environment can perform well in the testing.

For the future, more comprehensive documentations are needed to get better understanding of the CIDS-SIB. The documentation may also help in modeling the system behaviors. The model of the system testing environment can be used later to represent other testing environments with the same basic behaviors. They use the same model and applied their own specific devices without changing the model.

List of Abbreviations Acronym Description

AAP	Additional Attendant Panel
A/C	Aircraft
ACP	Area Call Panel
ADS2	Avionics Development System, 2nd Generation
AFDX	Avionics Full Duplex Switched Ethernet
AIP	Attendant Indication Panel
AMU	Audio Management Unit
ATA Air	Transport Association
BCEVI1	Electric Systems Integration & Test Cabin Management Systems
BITE	Build In Test Equipment
CAM	Cabin Assignment Module
CAN	Controller Area Network
CATEGA II	Computer Aided Test Generation Assistant (Version 2)
CIDS	Cabin Intercommunication Data System
CITR	Cabin Integration Test Rig
CMS	Central Maintenance System
CVT	Current Value Table
DEU	Decoder/Encoder Unit
DEESi	DEU Electrical Environment Simulation
DIR	Director
ECAM	Electronic Centralized A/C Monitoring
EPSU	Emergency Power Supply Unit
FAP	Flight Attendant Panel
FEDC	Fire Extinguishing Data Controller
FM	Failure Message
FSB	Fasten Seat Belts
FWS	Flight Warning System
GUI	Graphical User Interface
IBU	Integrated Ballast Unit (Cabin Light)
IDEFIX	Interface of Data Exchange in Test Facilities between IP and AFDX
IPCU	Ice Protection Control Unit

AAP	Additional Attendant Panel
LDCC	Lower Deck Cargo Compartment
MMC	Maintenance Message Control
MPB	Multipurpose Bus
NS	No Smoking
OBRM	On Board Replaceable Module
OMS	On Board Maintenance System
OE	Original Equipment
PA	Passenger Address
PISA	Passenger Interface and Supply Adapter
PRAM	Pre-Recorded Announcement & Boarding Music
PTS	Purchaser Technical Specification
PTT	Push-To-Talk
S/D	Smoke Detector
SDF	Smoke Detection Function
SIB	System Integration Bench
TDS	Test Data Sheet
TIP	Test Input
VL	Virtual Link
V&V	Verification and Validation

References

(1) Apfelbaum, Larry. John Doyle. Model Based Testing. Software Quality Week Conference. May 1997. March 2006 <"http://www.geocities.com/ model_based_testing/sqw97.pdf">

(2) Blackburn, et all. Defect Indentification With Mode-Based Test Automation. Software Productivity Consortium, NFP, T-VEC Technologies, Inc. March 2006, <"http://www.software.org/pub/taf/downloads/SAE_2003.pdf">

(3) "Black Box Testing**".** WebOPedia. February 2006. <"http://www.webopedia.com/ TERM/B/ Black_Box_Testing.html">

(4) "Chinese Postman Problem". EIE507 (Network Design: Theory and Practice). February 2006. <"http://eie507.eie.polyu.edu.hk/ss-submission/B7a/">

(5) Dalal, S. R., et all. Model-Based Testing In Practice. The 21^{st} International Conference On Software Engineering. May 1999. March 2006 <"http://www.geocities.com /model_based_testing /sqw97.pdf">

(6) El-Far, Ibrahim K., James A. Whittaker. Model-based Software Testing. Software Engineering Encyclopedia. Wiley,2001. March 2006, <"http://www.geocities.com/ model_based_testing/ ModelBasedSoftwareTesting.pdf">

(7) Hoglund, Greg & McGraw, Gary. Exploiting Software: How to Break Code. Boston:Addison-Wesley Professional, 2004.

(8) Lembcke, Ralph. Requirements for CIDS DEU – Electrical Environment Simulation System. Airbus Deutschland, 2004.

(9) "Model-based Testing", GoldPractice, March 2006,

<”http://www.goldpractices.com/practices/mbt/index.php”>

(10) Myers, Glenford J. The Art of Software Testing. New York:Wiley-Interscience Publication, 1979.

(11) Robinson, Harry. Model Based Testing . Software Testing & Quality Engineering magazine. March 2006. <”http://www.webopedia.com/TERM/W/White_Box_Testing.html”>.

(12) Robinson, Harry. Finite State Model-Based Testing On A Shoestring. Star West, 1999. March 2006. <”http://www.geocities.com/model_based_testing/shoestring.htm”>.

(13) Struss, Peter., “Model Abstraction for Testing of Physical Systems”. 8^{th} International Workshop on Qualitative Reasoning, QR-94, Nara, Japan. March 2006. <” http://www.qrg.northwestern.edu/papers/Files/qr-workshops/QR94/Struss_1994_Model_Abstraction_Testing_Physical_Systems.pdf“>.

(14) “White Box Testing”. WebOPedia . March 2006 <”http://www.webopedia.com/ TERM/ W/White_Box_Testing.html”>.