Safety-Critical Software Testing in Airborne Systems
The Modified Condition/Decision Coverage Criterion

by

Arnaud Dupuy

Submitted to the Department of Aeronautics and Astronautics
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999
[June 1999]

Author .....
Department of Aeronautics and Astronautics
May 7 '999

Certified by .....
Nancy G. Leveson
Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by .....
Jaime Peraire
Chairman, Department Committee on Graduate Students

Safety-Critical Software Testing in Airborne Systems

The Modified Condition/Decision Coverage Criterion

by

Arnaud Dupuy

Submitted to the Department of Aeronautics and Astronautics
on May 7, 1999, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

In order to be certified by the FAA, airborne software must comply with the DO-178B standard. For the unit testing of safety-critical software, this standard requires the testing process to meet a strong source code structure coverage criterion, refered to as Modified Condition/Decision Coverage criterion. This part of the standard is controversial in the aviation community, in particular because the coverage criterion is apparently not related to the safety of the software. In this thesis, we follow the letter of the DO-178B standard to perform the unit testing of the Attitude Control System of the HETE-2 satellite. This allowed us to gain some insights on the DO-178B testing procedure, and to prove that in the case of the considered software, this method was well adapted.

Thesis Supervisor: Nancy G. Leveson
Title: Professor of Aeronautics and Astronautics

Acknowledgments

I would like to thank the many people who helped me to complete this project:

My advisor, Pr. Nancy Leveson, for suggesting the subject of this thesis, which allowed me to explore a very exciting field of software engineering, and at the same time do some useful work for the HETE project. Thank you also for giving me the freedom and the support to pursue my own ideas.

Dr. George Ricker, for offering me the opportunity to work on the HETE project, and Dr. Francois Martel thanks to whom I am at MIT today.

The HETE team for always being great to work with. The help of Geoff Crew, Jim Francis and Greg Huffman has been decisive for the practical part of this thesis. I am very grateful to Bob Dill for reading my thesis, making pertinent suggestions, and always being encouraging.

John Chilenski, from Boeing, for answering all my questions on MC/DC and explaining me the subtilities of the DO-178B standard.

Steve Miller and David Statezni from Rockwell for guiding my research and supporting me through the all project. Their advice and their knowledge of the software testing problem from the industry point of view were very valuable.

Shaun Davey, from IPL, for answering my questions and sharing his experience of software testing, particularly for the C language.

Finally I would like to thank IPL and Attol testware for providing me with a license for their software testing tools. The efficiency of these tools surely had an impact on the conclusion of this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Airborne Software Certification

The verification of safety-critical software for airborne system is a complex process that requires several types of testing such as system testing, functional testing and module testing. System testing and functional testing are both high-level testing; they are performed on the final, complete product. System testing uses the high-level requirements to verify that the software does what the customer wants it to do. Functional testing uses the internal specification of the design team to verify that all the tasks the software has to support are correctly implemented.

Here we are interested in module testing. Module testing is the low-level testing of the software. It refers to the verification of the different components, or units of the program. Module testing shows two different aspects, both of them being equally important. First it verifies that there is no mismatch between the module specifications (low-level requirements) and the module coding. Then, it makes sure that the module has no unwanted side effect that could compromise the integrity of the system.

In the aviation domain the integrity of the on-board software is extremely important, since it is often related to the safety of the passengers. Hence module testing is of primary concern. Guidance for carrying out safety-critical on-board software testing in order to guarantee the airworthiness of an airborne system is provided by a standard called DO-178B, "Software Considerations in Airborne Systems and Equipment Certification". This standard was published jointly by the RTCA (Radio

Technical Commission for Aeronautics) and the EUROCAE (European Organization for Civil Aviation Equipment) in 1992. The certification by the FAA of safety-critical on-board software for civil aircraft is based on this document.

This standard is very controversial in the aviation community because it requires a testing technique called MCDC (Modified Condition / Decision Coverage) for the highest level of critical software. MCDC is a type of testing that guarantees that all the structure of the program has been reviewed during the tests. This process is very time consuming and extremely costly. In some avionics programs, half of the total cost of the program can be attributed to achieving the MCDC criterion for the code. Of course it makes sense to verify extensively the software responsible for the safety of the aircraft passengers, but many companies claim that MCDC is not the optimal way of spending their money. According to some avionics companies, MCDC is not effective in finding the 'interesting' (i.e. the safety) errors of the software.

But those claims have not been backed up by data, and they come from the companies themselves, those that spend a lot of money to achieve the MCDC criterion and are directly involved in the controversy. Hence they are easily dismissed by the certification authorities.

The idea of this thesis is to apply this DO-178B standard to an independent project involving critical on-board software. The selected project is HETE (High Energy Transient Explorer), a science mini-satellite developed at MIT which is to begin its software testing process in the summer of 1999. The different types of testing will be performed on the HETE satellite (and more particularly on the attitude control system) and possibly this will lead to useful results for the MCDC debate.

For now, we will just set the framework of the thesis. The testing techniques used in the aeronautical industry for on-board software will be described as well as the evolution that lead to the MCDC method. Then we will examine the DO-178B standard and give some theoretical arguments to explain why MCDC is so controversial.

1.1 Software Testing Techniques in Airborne Systems

The module testing of a program has two objectives. The first objective is to verify that the software does what it is supposed to do: the test process must check that the module matches its specifications (low-level requirements). The second objective is to verify that the software does not do what it is not supposed to do: the module testing process must demonstrate that it is not possible for the module to create an unwanted output that would cause the system to fail.

To meet this double goal, there exist two testing strategies. The first one is called *black box testing*, which is driven by the module specifications. The second one is called *white box testing*, and it is driven by the structure of the source code. The DO-178B standard combines those two strategies to ensure the total integrity of the software. Hence we first describe in more detail black box and white box testing.

1.1.1 Black Box Testing

When using black box (or requirement driven, or input-output driven) testing strategy, the testing team is only interested in the specifications of the module and the object code of the software. It chooses to ignore the structure of the source code and derive test cases only from the specifications in order to detect anomalous behavior of the software. Then the object code is used to run the test cases.

The main difficulty of black box testing is of course the design of pertinent test cases that will find as many bugs as possible in the module [11]. This section presents a few guidelines that can help in the selection of effective test cases.

Two Categories of Test Cases

There must be two categories of test cases: normal range test cases and robustness test cases. They correspond to the double objective of module testing (compliance with the specifications and preservation of the software integrity).

The normal range test cases feed the software with normal values, ie values that are expected by the module while the system operates nominally. The output of the module is then examined to make sure that the input has been processed correctly and that there is no discrepancy with the specifications.

The robustness test cases feed the software with abnormal values or have the module operate in off-nominal conditions. Examples of robustness test cases include

- Test cases providing out-of-range arguments to a function.
- Test cases inducing out-of-range results (overflow, type mismatch) in a function.
- System initialization under abnormal conditions.
- Time delay violation for the input data.

The goal of these test cases is to demonstrate that the module can handle off-nominal conditions without inducing system failure.

Exhaustive Testing

In all the practical cases, even the most simple ones, it is impossible to test a module exhaustively using the black box technique. Firstly it would require test cases that try all the valid inputs of the module to check that those inputs are correctly processed. Generally, this already implies an infinite number of test cases. But then all the non-valid input values must also be tested in order to prove that the module is able to deal with them without provoking a system failure. This can never be achieved.

Hence the tester must choose a limited number of inputs among all the possible solutions. Two methods, equivalence partitioning and boundary-value analysis allow the tester to select the most effective subset of inputs, that is the one which is the most likely to find all the different errors of the code with a given finite number of test cases.

Equivalence partitioning consists in dividing the input domain in equivalence classes such that it can be reasonably assumed that testing one value in the class is the same as testing all the values of the class. If the test case fails, then all the

other inputs in the class would find the same error. Conversely if the test case is successful, then all the other inputs of the class would also pass it. Of course since black box testing ignores the source code, it is possible for an equivalence class to contain a particular value singularized by the source code that could lead to an error. For example if some specification imposes an input to be greater than 1 and less than 10, we can define one valid equivalence class (the input is between 1 and 10) and two invalid equivalence classes (the input is less than 1 and the input is greater than 10). But if for some reason, the code treats the case 'input=5' separately, this partitioning does not allow us to test this case.

Boundary-value analysis refers to the design of test cases that focus on the edges of the input and output specifications. Such test cases will assign to the inputs the minimum and the maximum values allowed by the specifications; they will try input values that lead to the boundaries of the output value range; they will inspect the first and the last entry of a list, the limits of an index, etc. Experience shows that these test cases have a high probability of finding an error.

Black box testing has been found very effective at revealing coding errors. Equivalence partitioning and boundary-value testing allow selecting test cases that are likely to maximize the number of bugs found in the time period allocated to the testing process. But still this method by itself does not guarantee that the software is error free.

1.1.2 White Box Testing

White box testing (or structural testing, or logic-driven testing) is a testing strategy that takes advantage of knowledge of the source code. It uses the structure of the program (branching, conditional loops,...) to design the test cases.

First we give some definitions that are useful for the description of a program's structure:

Boolean variable A Boolean variable is a variable that takes on the Boolean values true (T) or false (F).

Boolean expression A Boolean expression is an ensemble of boolean values linked by Boolean operators (and, or, xor, not, =,...).

Condition A condition is a leaf-level Boolean expression (it cannot be broken down into a simpler boolean expression).

Decision A decision is a boolean expression that controls the flow of the program, for instance when it is used in a *if* or *while* statement. The simplest decision is composed of a single condition. More complex conditions are composed of larger expressions that combine many conditions.

Several white box testing techniques are available to verify a module. They differ by the level of code structure coverage they achieve. The criteria below are described in order of increasing structural coverage [1].

Statement Coverage Criterion

> **Statement coverage:** Every statement in the program has been executed at least once.

It is the simplest idea that comes to mind when testing the structure of a module. The test cases are designed so that they will execute every statement. In the following piece of code it suffices to set the condition to true to achieve full statement coverage.

```
if CONDITION
   then STATEMENT;
endif;
ANOTHER_STATEMENT;
```

Whatever the condition is (or even if the condition does not actually test anything), whenever it evaluates to true, the corresponding test cases are successful if the statements are correct. So this does not check at all the correctness of the flow control. In fact the statement coverage criterion is so weak that it is useless most of the time.

Decision Coverage Criterion

> **Decision coverage:** Every point of entry and exit in the program has been invoked at least once, and every decision in the program has taken all possible outcomes at least once.

Another way of expressing this is that all the branches in the module must be traversed at least once. In the following piece of code the decision coverage criterion is achieved with two test cases.

```
if CONDITION_1 and CONDITION_2
   then STATEMENT;
   else ANOTHER_STATEMENT;
endif;
```

Setting (CONDITION_1=true, CONDITION_2=true) and then (CONDITION_1=false, CONDITION_2=true) will execute the two branches of the program. But again this criteria is quite weak since even for this simple example, the influence of CONDITION_2 on the flow is not tested: in both of our test cases, it was set to true.

Condition/Decision Coverage Criterion

> **Condition/decision coverage:** Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, and every decision in the program has taken all possible outcomes at least once.

Condition/decision criterion may appear to guarantee the coverage of all the conditions in the module, just like the decision criterion was covering all the decisions. This is not true because in many test cases some conditions of a decision are masked by the other conditions. This situation can happen even in a simple case like :

```
if CONDITION_1 or CONDITION_2
   then STATEMENT;
   else ANOTHER_STATEMENT;
endif;
```

To meet the condition/decision coverage criterion the following test cases are chosen:

CONDITION_1	CONDITION_2	decision
true	true	true
false	true	true
false	false	false

CONDITION_1 takes on the two possible outcomes true and false, but its influence on the decision is not demonstrated. CONDITION_2=true in conjunction with the *or* operator masks CONDITION_1 in the first two test cases. In bigger decisions, such a masking is not as obvious as in this example, and a complete coverage of the code structure is not guaranteed by the condition/decision coverage criterion.

Modified Condition/Decision Coverage (MCDC)

Logically the next step of this review should be a criterion where "all the possible combinations of the outcomes of the conditions within each decision have been taken at least once". This would tackle the problem of masking in the condition/decision coverage criterion and guarantee the thoroughness of the structural testing. Unfortunately in the domain of avionics, each decision often consists of complex boolean functions. A figure of N=20 conditions in a decision is not uncommon. Since the number of test cases grows exponentially (like 2^N) with the number of conditions per decision, this type of testing is soon impossible to apply to a typical module. This is why the modified condition/decision coverage criterion has been created.

> **Modified condition/decision coverage:** Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once,

and each condition has been shown to independently affect the decision outcome. A condition is shown to independently affect a decision's outcome by varying just that decision while holding fixed all other possible conditions.

The crucial new point is that each condition must be shown to be able to act on the decision outcome by themselves, everything else being held fixed. The MCDC criterion is then much stronger than the condition/decision coverage criterion. But surprisingly enough, the number of test cases to achieve the MCDC criterion still varies linearly with the number of conditions N in the decisions.

We recall the preceding example to illustrate the method:

```
if CONDITION_1 or CONDITION_2
   then STATEMENT;
   else ANOTHER_STATEMENT;
endif;
```

The following table shows all the structural tests that can be performed on this fragment of program.

test case	CONDITION_1	CONDITION_2	decision
1	true	true	true
2	true	false	true
3	false	true	true
4	false	false	false

Now to satisfy the MCDC criterion for CONDITION_1, test cases 2 and 4 must be performed because they are the only ones for which the decision outcome changes because of CONDITION_1 while CONDITION_2 is fixed. Similarly for CONDITION_2, test cases 3 and 4 must be performed. Hence for this piece of program with a two-condition decision, N+1 test cases (test cases number 2, 3, and 4) are necessary to meet the MCDC criterion.

The MCDC criterion always requires a minimum of N+1 test cases [2] : one test case for the starting configuration of the conditions, plus at least one test case per condition. Most of the time those N+1 test cases are sufficient to achieve the coverage.

The principal exception to this empirical rule is the case of coupled conditions [2]. Conditions are said to be coupled when it is impossible to vary one without varying the other, as in the decision ((A and B) or (notA and C)). The method to deal with this situation is to make a distinction between the different instances of a condition. For the above example, the instance of A in the first expression will be denoted by A_1 and the one in the second expression by A_2. The condition becomes ((A_1 and B) or (notA_2 and C)). Then a complete coverage, with respect to the conditions, is guaranteed if we can find some test cases that will show the effect of A_1 while the condition A_2 is masked and the other conditions remain fixed. And conversely for A_2. Of course it is not sure that such a set of test cases exists for every decision with coupled conditions, and when it exists, it may be bigger than usual. The following table shows the different possible test cases for ((A and B) or (notA and C)):

test case	A	B	C	decision
1	true	true	true	true
2	true	true	false	true
3	true	false	true	false
4	true	false	false	false
5	false	true	true	true
6	false	true	false	false
7	false	false	true	true
8	false	false	false	false

Test cases 2 and 6 can be used to prove that A_1 satisfies the MCDC criterion since

- The outcome of A_1 changes and so does the decision
- The condition A_2 is masked by C, which is set to false
- The conditions B and C remain constant.

Similarly, test cases 3 and 7 can be used for the condition A_2. We can choose test cases 2 and 4 for B, and 7 and 8 for C. Finally the MCDC criterion for the decision ((A and B) or (notA and C)) can be satisfied with test cases 2, 3, 4, 6, 7, 8. In this example N+2 cases are necessary. Note that if a decision can be tested, it will never require more than 2N test cases (two for each condition) —the linearity is always preserved.

Despite the problem of coupled variables, the MCDC criterion is a very good compromise for the white box testing technique. First it insures a much more complete coverage than decision coverage or even the condition/decision criterion. But at the same time, it is not terribly costly in terms of number of test cases as a total coverage criterion would be. So in theory, MCDC is very appealing; it means a serious coverage of the code structure with a number of test cases varying linearly with the number of conditions.

1.1.3 Additional Comments on the MC/DC Criterion

Masking MC/DC

As described above, the technique of masking is used in the standard when the MC/DC criterion is applied to a decision with coupled variables. For example in the decision (A or B) and (A and C), B=true is used to mask the first instance of A and test the second instance; C=false is used to mask the second instance of A and test the first instance.

This leads to the natural question: why couldn't we use the masking technique for all the test cases, and not only for the cases with coupled variables [6]. This would be equivalent to replacing the DO-178B criterion for achieving MC/DC coverage:

> Each condition must be shown to independently affect the decision's outcome by varying just this condition while holding fixed all other possible conditions.

with, for example, this definition:

Each condition must be shown to independently affect the decision's outcome such that only this condition can have an influence on the decision's outcome.

The first criterion, used in the standard is referred to as "unique cause MC/DC" since the tested condition is toggled while all the other conditions are fixed(when it is possible, i.e. when there is no coupling). The second criterion is referred to as "masking MC/DC" since the tested condition is toggled while the other conditions stay fixed or are masked.

The following example for the decision A and (B or C) highlight the differences between the two versions. The truth table for this decision is given below.

test case	ABC	decision's outcome
0	FFF	F
1	FFT	F
2	FTF	F
3	FTT	F
4	TFF	F
5	TFT	T
6	TTF	T
7	TTT	T

In the case of the *unique cause* MC/DC criterion, the possible pairs of tests for each condition are:

condition A (1,5) (2,6) (3,7)

condition B (4,6)

condition C (4,5)

Hence a minimum of four tests must be run to check this decision, and the possibilities are (2,4,5,6) or (1,4,5,6).

Now in the case of the *masking* MC/DC criterion we have the following possible pairs:

condition A (1,5) (2,6) and (3,7); (1,7) and (3,5): C=T masks C switching; (2,7) and (3,6): B=T masks C switching

condition B (4,6)

condition C (4,5)

The masking version of MC/DC also requires four tests to complete the check, but there is one more possibility, (3,4,5,6), in addition to (2,4,5,6) and (1,4,5,6).

Masking MC/DC being less strict than unique cause MC/DC, it requires less test cases to be run in general. But the other major advantage of the masking version is that it never requires a complete analysis of the coupling between the conditions. The coupling in a decision can be very easy to see, for example when the same variable appears several times in the expression. But sometimes it is more difficult to figure out the hidden relation between the different conditions, and to find out that certain states of the truth table cannot exist. The link between the variable can originate from anywhere in the program, and an exhaustive study of all the possible values that the decision vector can take on is a problem of exponential complexity that cannot be solved systematically. For the unique cause criterion, this problem must be addressed, since only the coupled conditions are allowed to be masked. For the masking version of MC/DC, since the masking criterion treats the coupled conditions and the regular condition in the same manner, the study of the coupling is completely avoided.

Finally, note that if all the boolean operators of the decision are the same, then the unique cause criterion and the masking criterion are equivalent, since in this case, the outcome of the decision cannot change if one of the conditions is masked.

To conclude, masking MC/DC is an interesting option to consider for the evolution of the standard on the question of white box testing. It simplifies a lot the analysis of a program (particularly, it makes the task of an automated tool much easier to implement, since the coupling analysis is skipped) and preserves the essence of the MC/DC criterion.

Short Circuiting Languages

In a language such as C, all the Boolean operators are short circuited. For instance in a decision like (A and B), if A is false, B is not evaluated and the decision is directly assigned the outcome "false". In the same manner, in a decision like (A or B), if A is true, B is not evaluated and the decision is assigned the outcome "true". In other terms, as soon as the output of the expression is not ambiguous any more, the rest of the expression is short circuited.

This feature reduces a lot the actual truth table of a decision, since in many cases, some decisions are not even evaluated. For instance, recall the example of the decision A and (B or C). For a non short circuiting language, like ADA, the truth table with the corresponding MC/DC tests is as follow:

test case	ABC	outcome	A	B	C
0	FFF	F			
1	FFT	F	5		
2	FTF	F	6		
3	FTT	F	7		
4	TFF	F		6	5
5	TFT	T	1		4
6	TTF	T	2	4	
7	TTT	T	3		

The two valid minimum sets of tests are (1,4,5,6) and (2,4,5,6). However in the case of short circuiting operators, the effective truth table is greatly reduced (an 'X' means 'not evaluated').

test case	ABC	outcome	A	B	C
R0	FXX	F	R2,R3		
R1	TFF	F		R3	R2
R2	TFT	T	R0		R1
R3	TTX	T	R0	R2	

There is only one valid set of tests for the reduced truth table which is simply (R0,R1,R2,R3). However, this set gives rise to 8 different sets when we go back to the level of the actual variables ('X' can be chosen freely as true or false). The eight valid sets of tests are (0,4,5,6), (1,4,5,6), (2,4,5,6), (3,4,5,6), (0,4,5,7), (1,4,5,7), (2,4,5,7), (3,4,5,7).

So the MC/DC criterion is much more relaxed when it is applied to a short circuiting language and if the short circuiting characteristic is taken into account in the truth table.

Moreover it is almost impossible to run the tests based on the complete truth table. This would require a deep change in the code for the testing, so the tested code and the the real code would be very different, which is not a safe situation. So the short circuiting nature of a language *must* be taken into account, and the reduced truth table is the right table to be used. All the coverage testing tools for C follow this rule.

In consequence, the efficiency of the MC/DC criterion will be very different for a short circuiting language as C and for a non short circuiting language as ADA. All the conclusions on the efficiency of the DO-178B standard applied to software written in C will only be valid for the class of the short circuiting languages.

After reviewing the current testing strategies that are relevant to the validation of on-board software, we are ready to examine and to understand the motivation of the DO-178B standard.

1.2 Airborne Systems Software Certification

1.2.1 Software Levels in Airborne Systems

The programs running in airborne systems are divided into different categories, based on the possible consequences of a software failure on the system [1]. Table 1.1 is a summary of the different software levels.

Software level	Potential consequence of a failure
A	catastrophic failure condition for the aircraft
B	hazardous/severe-major failure condition for the aircraft
C	major failure condition for the aircraft
D	minor failure condition for the aircraft
E	no effect

Table 1.1: Software classification in the DO-178B standard

Objective to be achieved	level A	level B	level C
Test coverage of low-level requirements	X	X	X
Test coverage of software structure (statement coverage)	X	X	X
Test coverage of software structure (decision coverage)	X	X	
Test coverage of software structure (MCDC)	X		

Table 1.2: Testing requirements for the different types of software

Each software category has to comply with a certain level of testing. The more critical is the software for the safety of the aircraft, the more constraining is the test plan required by the standard. No guidelines are given for a level E software, which has no implication for safety whereas a very extensive program is demanded for level A software. Between level E and level A, the thoroughness of the testing process increases step by step. As shown in the table 1.2, the major differences between the levels are found in the degree of structural coverage achieved by the white-box testing of the code.

We will now focus on Level A software.

1.2.2 DO-178B Testing Strategy

The DO-178B philosophy is to combine black box testing and white box testing to derive a reasonable but very complete set of tests.

Testing Process

Step 1 The process starts with black box testing. The first goal is to make sure that the module meets its low-level requirements. Black box testing is very well suited

for that since the test cases are directly derived from the specifications, and they are designed to verify every specification. So at the end of the first step, the software requirement coverage is complete.

On top of finding requirement-related errors, this step is also intended to find the first batch of purely coding-related errors, that is the ones made by the developer while writing the code and that are not related to a bad interpretation or a bad implementation of the requirements. Black box testing has been found very effective at revealing this type of error too, certainly because it has the developer and the tester work on the same basis —the requirements. Therefore an experienced tester can check the most common human mistakes (e.g. errors on the requirement boundaries) or detect the sensitive points in the requirements that can lead to difficulties in coding.

This property is used to find quickly as many errors as possible in the first phase of testing. Then to find the remaining problems which are more hidden to human reasoning, another method which will be less efficient but more systematic will be used.

Step 2 Before switching to another method, the conclusions of the black box testing phase must be drawn : this step is to determine what level of structural coverage has been achieved at this point. The decisions that have not been fully exercised so far, in term of coverage criterion, are identified. For level A software, all the decisions for which the previous test cases failed to achieved the MCDC criterion are marked.

Step 3 Additional testing is made on the not fully covered part of the code that was identified in step 2. More test cases are designed so that the code structure coverage is guaranteed according to the MCDC criterion. Since the structure of the code is analyzed in this phase, this is white box testing technique. White box testing test cases depend less on human judgment so this step is expected to find uncommon errors that were missed by the previous review.

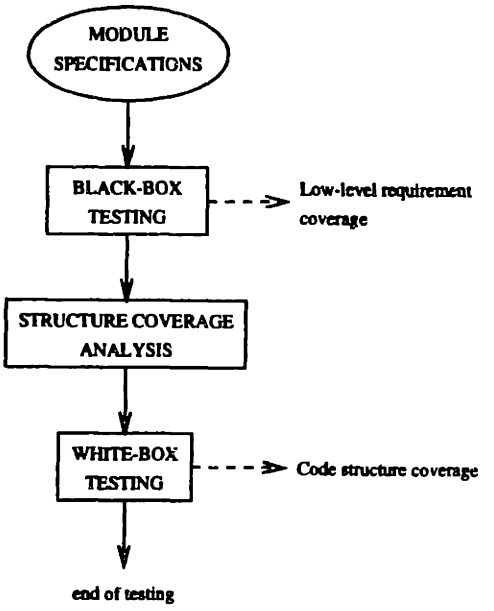

Figure 1-1: Testing process

Discussion

The two last steps involving MCDC and white box testing are the controversial parts of the standard because they are terribly costly and time consuming: moreover, the avionics companies claim that they find very few errors (unfortunately at the moment they have exhibited very little data to support this).

From the point of view of the standard, structural testing is not meant to be efficient at finding errors —this is the role of the first step of the process. Rather its goal is to make sure that no code structure was left untouched. And as has been shown in the preceding section, the MCDC criterion is the only criterion that can provide the tester with good confidence in the structural integrity of the software.

Nevertheless this does not mean that structural testing with MCDC is the best possible way to convince ourselves of the integrity of the software. In particular one of the main drawbacks of the method is that it totally loses track of the requirements. It focuses on the code structure and neglects the specifications, not to mention safety considerations. The MCDC criterion is imposed to insure that the software is safe for the system, but the idea of safety never appears in its definition. Some testing

operand combinations	possible Boolean functions						
	f_1	f_2	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$f_{2^{2^N}}$
$0000\cdots0$	0	0	$\cdots$	$\cdots$	$\cdots$	$\cdots$	1
$0100\cdots0$	0	1	$\cdots$	$\cdots$	$\cdots$	$\cdots$	1
$0010\cdots0$	0	0	$\cdots$	$\cdots$	$\cdots$	$\cdots$	1
$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\vdots$
$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$\vdots$
$1111\cdots1$	0	0	$\cdots$	$\cdots$	$\cdots$	$\cdots$	1

Table 1.3: The 2^{2^N} possible Boolean functions of N operands

techniques directly related to safety would be more appropriate, but such techniques have not been developed yet.

Practically, the difficulty of achieving MCDC is not in the number of tests (in general N+1 tests are required for each condition, which is reasonable); rather the real problem is to determine what coverage has been achieved by the former part of the testing process and to find good complementary tests that will satisfy the coverage criterion.

So the question is to know if it is really worth it to search for the MCDC-related test cases and implement them, or if a number of randomly chosen test cases for each condition would be as efficient while much less difficult to implement.

Statistically speaking, there is no difference between a set of test cases chosen in order to satisfy the MCDC criterion and a random set of tests. Let for example a decision represented by a Boolean function of N operands (ie N independent conditions). There are 2^N different combinations of these operands. The Boolean function is defined by the value it takes on for each combination of the operands. Since there are two possible outcomes for the function (0 or 1) in each case, there are 2^{2^N} possible Boolean function of N operands [13].

If one runs M test cases on this decision, then the outcome of the function for N combinations of the operands is checked. $2^N - M$ combinations that can lead to a wrong outcome are left. The number of wrong functions that can be generated using those $2^N - M$ unverified combinations is $2^{2^N-M} - 1$ (the one corresponds to the

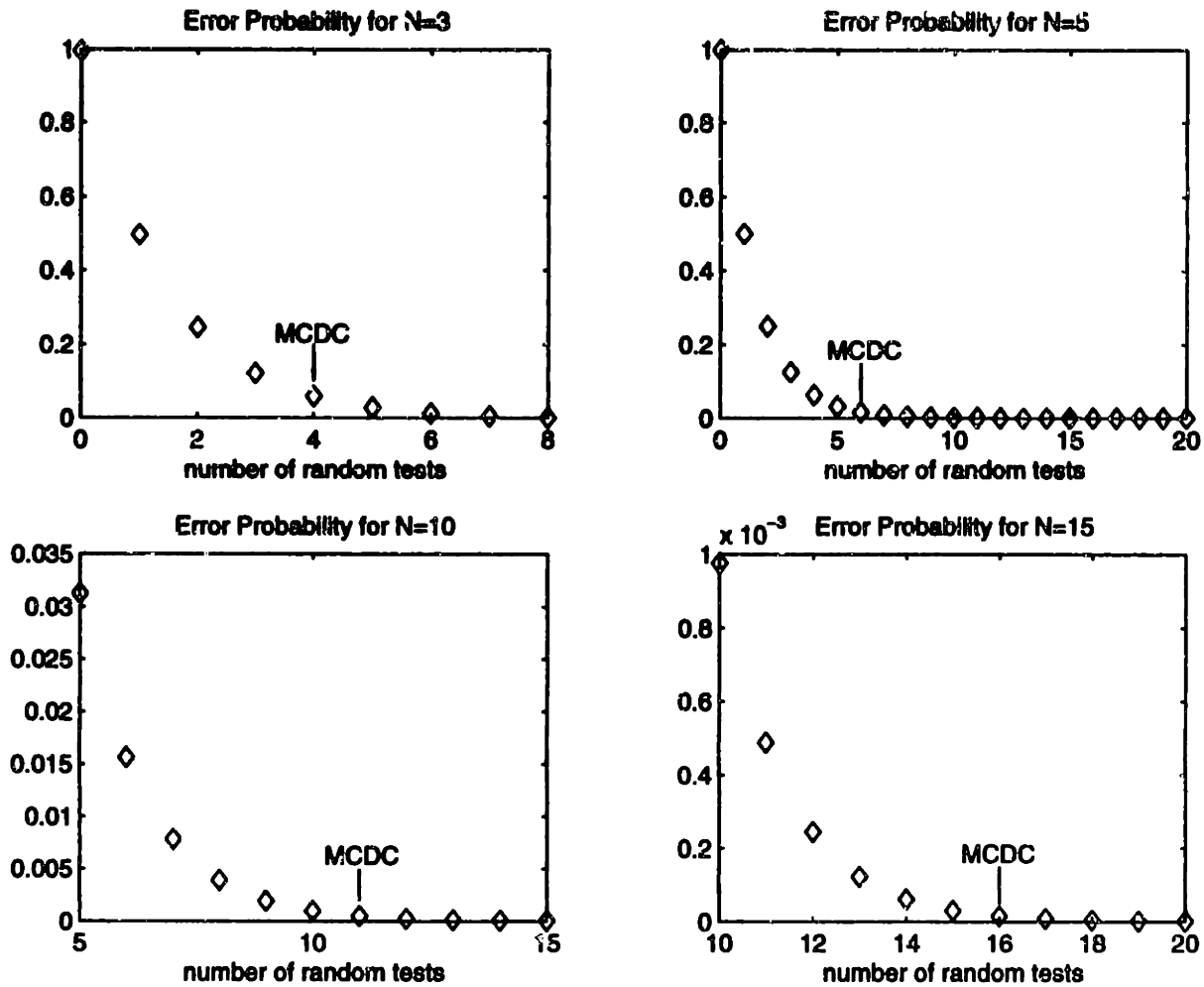

Figure 1-2: Error probability for different sizes of decision function

correct function). Therefore the probability of having a wrong function is

$$P = \frac{2^{2^N - M} - 1}{2^{2^N}}$$

and for $N \geq 3$ (ie three or more conditions in the decision) :

$$P \approx 2^{-M}$$

Of course this result does not depend on the specific test cases that are chosen since it has been considered that all the test cases are equally likely to detect an

error. Nevertheless, it turns out that the probability of finding an error decreases exponentially with the number of test cases. So if a large number of tests are run (and N+1 tests, like in the case of MCDC is already a large number of tests, considering the exponential behavior of P), we dramatically reduce the risk of error, whatever the test cases are. It seems that part of the error finding power of MCDC resides in the number of tests requested and not in the type of tests requested. This is especially for the complex decisions (N large) since the probability curve is almost invariant in N. The only parameter that changes when N increases is the point corresponding to the N+1 tests of the MCDC criterion: it moves toward the flat part of the probability curve, close to zero and the advantages provided by MCDC in this case is questionable. The careful choice of the test case has little influence compared to the safety provided by the large number of tests.

Chapter 2

The HETE Project

The topic of the thesis is to set up a testing plan for on-board, safety-critical software that will follow the letter of the DO-178B standard. Hopefully this experiment will be able to provide useful, independent data to the industry and allow a better understanding of the different issues addressed in section 1.2.2.

The project which is to be considered for this experiment is HETE-2 (High Energy Transient Explorer), a science mini-satellite developed at MIT [14], and more precisely its Attitude Control System (ACS) software. Of course a satellite is not a typical FAA project but the safety implications of the ACS software are very close to that of level A safety airborne software. In particular, a failure of the ACS can cause the loss of the satellite or at least a complete failure of the mission. The satellite ACS software, as far as the requirements, the constraints, and the safety issues are concerned is similar enough to avionics software to consider that the study of the ACS software testing process is relevant to level A airborne software.

A first HETE satellite was launched in November 1996. It was lost because of the failure of its Pegasus XL launcher. But the scientific issues addressed by the HETE mission are still relevant, so a second satellite, HETE-2 is being rebuilt. It is scheduled for launch in early 2000 and the satellite is now in the final phase of development, requiring extensive software testing. We can use this need to apply the DO-178B testing process to the ACS software.

2.1 General Presentation of the Satellite

2.1.1 Mission

The HETE mission is the study of the astrophysical events known as gamma ray bursts. Gamma ray bursts are high energy transients in the gamma range that seem to be isotropically distributed in the sky. They last from a millisecond to a few hundreds of seconds, and they involve a huge amount of energy. HETE is expected to detect the bursts, locate them and capture their spectral characteristics [14]. The satellite orbit will be circular, with an apogee of 640 km, a perigee of 600 km, and an inclination of about 2 deg.

2.1.2 Payload

The spacecraft carries several instruments that are used for the mission, namely four gamma ray telescopes, two wide field X-ray cameras, two soft X-ray cameras and two optical cameras. Two of the optical cameras provide the Attitude Control System with the drift rates during orbit nights.

2.1.3 Bus

The satellite weighs 296 lbf, it is 35.5 inches in height and 34.0 inches in diameter. Four deployable solar paddles and six batteries (for eclipse time) will supply 150 W of power. During normal operation, the instruments have an anti-solar pointing so that the solar cells, which are located on the bottom side of the solar paddles, are fully illuminated.

The communication system includes a VHF transmitter and an S-band transmitter receiver. The five S-band antennas are located at the tips of the solar paddles, and on the bottom of the satellite, so that they cover all space and no antenna pointing is required by the ACS. The VHF system is not used for telemetry; it has a low gain whip antenna located at the end of one of the solar panels.

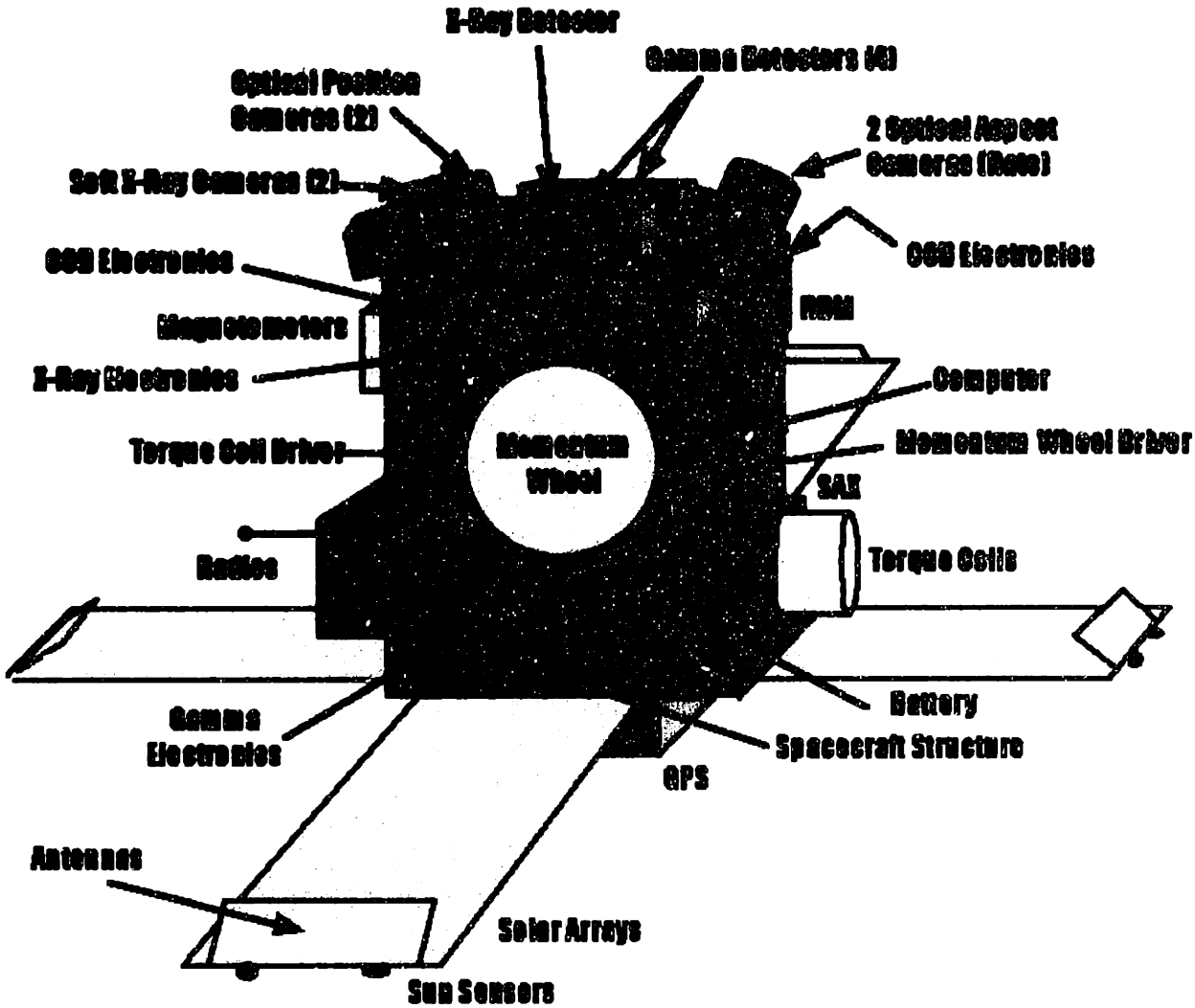

Figure 2-1: The HETE satellite

2.1.4 Computing System

The computing system of the satellite includes 12 processors organized in four different groups of boards, called the four "nodes". Each node has three processors, one INMOS T805 transputer and two Motorola DSP56001, and some memory devices (memory boards and frame buffers). The DSPs run the application codes and handle the communication protocols. The transputers control the DSPs, manage the node memory space, handle and store the data generated by the DSPs.

The different devices on the satellite communicate through the "Aux bus", which is a medium data rate serial bus. For example the Aux bus allows the processors to send commands to the ACS actuators and to read the sensors' A/D converters.

Each node operates independently and is responsible for operating one or several subsystems of the satellite. Node 0 operates the platform subsystems (Attitude Control System, RF and VHF communications) and controls the Aux bus. Node 1, 2 and 3 operate the payload instruments.

2.2 Attitude Control System Hardware

2.2.1 Sensors

The ACS uses two types of sensors to determine the attitude of the spacecraft: sun sensors and magnetometers. These devices produce analog outputs which are multiplexed, converted and buffered in the SAX (sensors to Aux) board. The SAX board is connected to the Aux Bus so the spacecraft DSP can access the data via an Aux request/read cycle.

Sun Sensors

The sun sensors allow the ACS to compute the attitude of the spacecraft with respect to the sun. Twelve Coarse Sun Sensors (CSS) are mounted on the satellite, two on the tip of each solar paddle, two on the top of the satellite and two on the bottom.

Each CSS outputs a current which is approximatively a cosine function of the

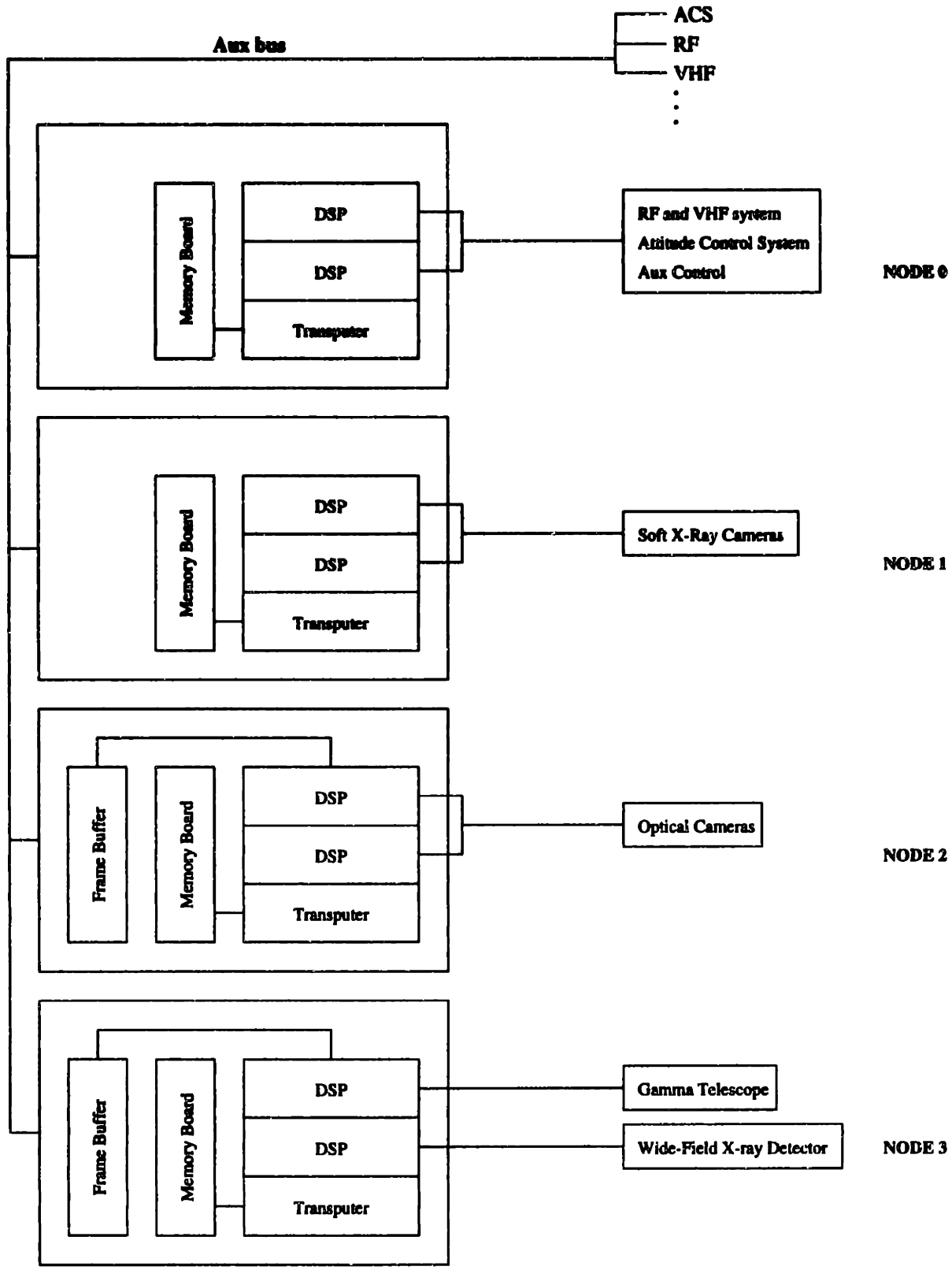

Figure 2-2: HETE computer architecture

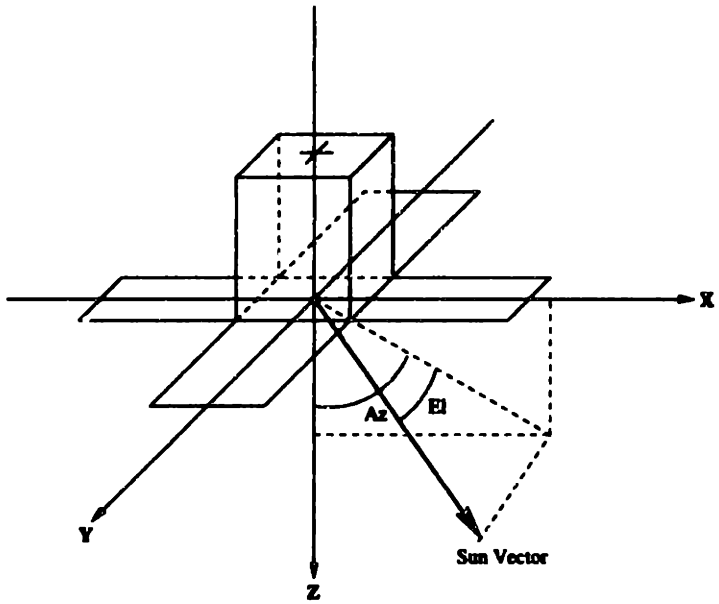

Figure 2-3: Definition of the elevation and azimuth angles

CSS#	location	axis (paddle stowed)	axis (paddle deployed)
1	bottom	+Z	+Z
2	bottom	+Z	+Z
3	top	-Z	-Z
4	top	-Z	-Z
5	+X paddle	-Z	+X
6	+X paddle	+X	+Z
7	+Y paddle	-Z	+Y
8	+Y paddle	+Y	+Z
9	-X paddle	-Z	-X
10	-X paddle	-X	+Z
11	-Y paddle	-Z	-Y
12	-Y paddle	-Y	+Z

Table 2.1: CSS pointing directions

angle between the CSS axis and the sun. A sun pointing vector is computed first by averaging the outputs of the CSS facing the same direction and then by subtracting the -X, -Y, -Z values from the +X, +Y, +Z values respectively. This sun pointing vector allows the determination of the azimuth angle and the elevation angle between the +Z direction and the sun direction in the satellite frame.

In addition to the coarse sun sensors, there is one Medium Sun Sensor (MSS) and one Fine Sun Sensor (FSS) on the bottom of the satellite facing the +Z direction. The MSS gives directly a precise value of the elevation and azimuth angles. So as soon as the data from the CSS indicates that the sun is in the field of view of the MSS (MSS FOV = 50 degrees from boresight), the elevation and the azimuth are computed from the MSS. The Fine Sun Sensor works on the same principle as the MSS but it is more precise. As soon as the the sun is found to be in the field of view of the FSS (FSS FOV = 15 degrees from boresight), the elevation and azimuth are computed from the FSS data.

Magnetometers

The magnetometers allow the ACS to determine the spin vector of the spacecraft. Two magnetometers are mounted on the spacecraft: one on the +X wall and one on the -X solar paddle. Nominally, the software uses the data from the +X magnetometer, while the other one is only for redundancy.

Each magnetometer outputs the three components of the magnetic field (X,Y,Z), and a bias. The ACS software corrects the components for the torque coils disturbance (see next paragraph on the actuators) and compensates for the bias.

The spin vector is computed from the derivative of the magnetic field, $\frac{dB}{dt}$, which is estimated from B by a digital filter. Since B is almost constant over a sample period (1 second), $\frac{dB}{dt} \simeq \omega \times B \simeq \omega_\perp$ where ω is the rotation vector of the spacecraft in the inertial frame and $\omega_\perp$ is the component of ω perpendicular to B. So in fact the magnetometer gives information only on the component of the spin vector which is perpendicular to B. But since the B vector changes with respect to the satellite as the satellite goes along its orbit, globally all the components of the spin can be

observed.

2.2.2 Actuators

The ACS uses 3 torque coils and one momentum wheel to modify the attitude of the spacecraft. Both types of devices apply torques to the spacecraft (the satellite is only capable of *attitude* control, not *orbit* control: no translation is possible), but the maximum wheel torque is about 100 times larger than the maximum coil torque.

Torque Coils

The three torque coils are aligned on the three axes of the satellite. The magnetic moment of a torque coil $M = IS\vec{n}$ (where I is the current intensity, S is the effective area of the coil and $\vec{n}$ is the normal vector to the coil) interacts with the magnetic field B to create the torque $M \times B$. The coils are fed with the bus voltage which is fixed at about 28V, hence the intensity of the current cannot be varied. Rather the ACS feeds a train of pulses into the coil. The ACS controls the frequency of the pulses, hence it can control the duty cycle, that is the fraction of time for which the coil is active (ie when it is fed with the maximum current corresponding to the 28V bus voltage). The duty cycle is computed as the desired theoretical current divided by the max current corresponding to the bus voltage. Note that only the value of the bus *voltage* is read by the Aux bus. Hence to calculate the value of the maximum current, the coil resistance must be known accurately. Since this resistance depends on the temperature, a temperature sensor is coupled with each coil to allow the ACS to pick the right value.

Momentum Wheel

The momentum wheel is mounted on the +Y wall of the satellite, with its axis aligned on the +Y axis. Thus it can generate a torque about the +Y axis.

The spacecraft DSP operates the wheel via the Aux bus and the wheel driver. It can read the A/D converted value (torque and voltages), and send different commands

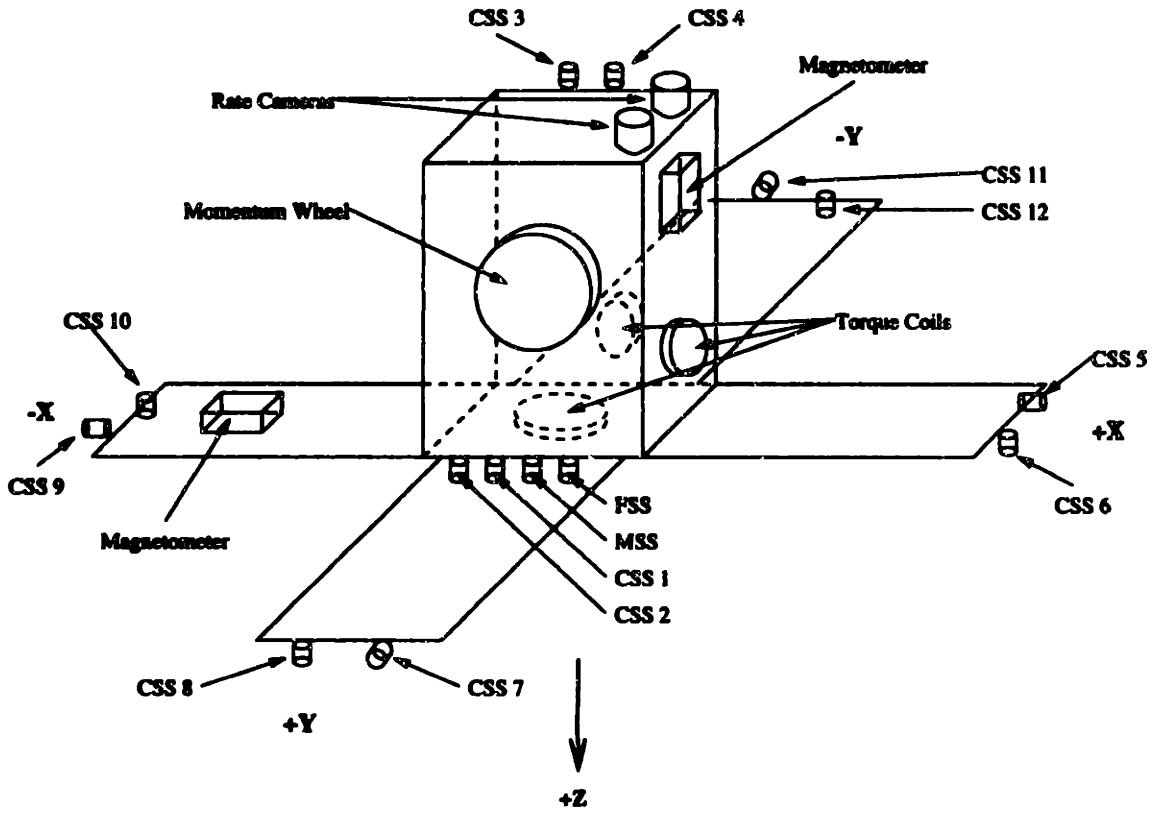

Figure 2-4: Placement of the ACS hardware

(apply torque, prepare speed). The "prepare speed" command requests a wheel speed and it differs depending on the speed of the wheel. The "all range prepare speed" command can be used for any speed up to the maximum speed (±377rad/s). The "mid range prepare speed" command can be use for speeds above 24 rad/s (it has the best resolution in the 24 rad/s range). The "high range prepare speed" command can be used for speeds above 64 rad/s (and it has the best resolution for speed above 64 rad/s). The use of these commands below their respective threshold results in a counter overflow. The "apply torque" command allows applying a torque to the satellite ranging from -0.020Nm to +0.020Nm.

2.3 Attitude Control System Software

The ACS software is written in C and is compiled using the GCC GNU compiler under a Sun/Solaris environment. Counting lines of code is often problematic, but to give an idea of the size of the software, we shall say it has about 6000 lines of source code.

Ultimately, the software will run on the spacecraft's transputer on board the satellite, and the object code for the this processor will be generated by a cross compiler. No simulation facility is supported on the transputer, so the module testing had to be done in the UNIX environment, and only the high level testing (functional testing and system testing) will take place in a realistic software environment.

The DO-178B document requires the module testing to be performed in the target environment, so the testing does not comply with the standard on this point. As a matter of fact, the testing on UNIX does not guarantee us against cross compiler bugs.

But in the case of HETE, it was the only feasible way to conduct the module testing process. I think this form of testing, even if it did not involve the definitive source code, was useful to the commisioning of the ACS software; and anyway, this problem does not interfere with the study of the efficiency of the DO-178B testing procedure.

2.4 Attitude Control System Software Requirements

The ACS controls the deployment sequence from the moment the satellite is released by the rocket (solar paddles stowed, tumbling attitude) until the moment it reaches its final orbit configuration (paddles deployed, axis stabilized, anti solar instrument pointing). When the operations phase begins, the ACS is still a key system for the success of the mission: it is crucial for the correct operation of the instruments, for the power balance, and for the thermal balance. During the operations phase the ACS

	attitude control
orbit-day	(+Z axis, sun vector) $\leq 5^o$
orbit-night	drift $\leq$ 30 arcsec

Table 2.2: ACS requirements

and the instruments are closely coupled, which leads to some precise requirements for the ACS. These requirements are divided into orbit-day requirements and orbit-night (eclipse time) requirements.

Orbit-day The attitude is determined from the sun sensor data. Anti-sun pointing of the instruments: the angle between the +Z axis and the sun vector must be less or equal to 5 degrees.

Orbit-night The optical cameras provide the drift rate to the ACS.

- Less than 20 arc seconds per second drift rate on all three axes upon entering optical tracking mode.
- Less than 30 arcsec drift from initial position (beginning of camera tracking).

Initial conditions When it is released from the rocket, the satellite has nominal rotation rates of zero about all three axes. The maximum rotation rate at release time about any axis is 5 deg/s. The ACS must be able to go through all its deployment sequence in this worst case.

2.5 Attitude Control System Software Specifications

In order to perform efficiently all the different operations it is responsible for, the Attitude Control System is broken down into ten different modes. The system goes through mode 0 to 6 during the deployment sequence. During the operations phase,

modes 7 and 8 are activated alternatively. Mode 9 is the backup ground command mode.

2.5.1 Mode Description

The following is a qualitative description of the different modes. The goal is to present the general characteristics of the systems that are necessary for understanding the code.

Mode 0

Mode 0 is a wait mode. No torques are commanded, the ACS just collects and processes the data from the sensors.

Mode 1

Detumble mode. The goal of mode 1 is to minimize the spin component about the all three axes. The commanded magnetic moments for the torque coils have the form $M = -k_1 \frac{dB}{dt}$. Hence the generated torque $\tau = m \times B$ is opposite to the spin vector or more precisely it is opposite to the spin vector component which is perpendicular to B (see the paragraph on the magnetometers). Due to the orbital motion of the satellite, the magnetic field direction changes and globally mode 1 generate torques that oppose the rotation in all three axes.

Mode 2

The goal of mode 2 is to spin up the satellite about the Y axis until the Y component of the momentum vector reaches a certain threshold. This provides some stiffness to the Y axis so that the elevation error is well-behaved and will be easier to minimize in the following steps. The target momentum vector is computed so that the wheel, when it speeds up during its deployment (mode 4), will absorb all the momentum created by the Y axis rotation, leaving the satellite at rest in the inertial frame when it reaches its nominal speed. The commanded magnetic moments for the torque coils

have the form $M = -k_1 \frac{dB}{dt}$ for the Y axis (we still want to minimize the elevation error), and $M = k_2 B \times H_{needed}$ for the X and Z axes. H_{needed} is the difference between the target momentum error and the measured momentum vector. The plus sign in the equation causes the X and Z torque coils to magnify the rotation about the direction of H_{needed}.

Mode 3

Mode 3 is designed to set the position of the momentum vector in the inertial frame. When the ACS enters mode 3, the satellite is supposed to be spinning about the Y axis, and the ACS will aim at bringing the momentum vector perpendicular to the sun, minimizing the elevation angle error. The control law for the magnetic moments is:

$$M = k_3 B \times H_{needed}$$

where

$$H_{needed}(x) = sin(el)sin(az)$$

$$H_{needed}(y) = H_{desired}(y) - H_{measured}(y)$$

$$H_{needed}(z) = sin(el)cos(az)$$

$H_{needed}(x)$ and $H_{needed}(z)$ are such that they are going to zero whatever the instantaneous azimuth angle is. For example when the azimuth angle is close to 180 degrees because of the spinning movement (that is the +Z axis is opposite to the sun), $H_{needed}(x)$ and $H_{needed}(z)$ will still contribute to reduce the elevation error. The second equation gives the commands to maintain the value of the spinup rotaion about the Y axis. The combination of these three corrections will allow a proper deployment of the momentum wheel (the momentum wheel can only control the Y component of the momentum vector) in the following step.

Mode 4

Deploy wheel. In this mode the wheel spins up to its nominal speed (141.37 rad/s). The wheel speed is controlled by a first order system with feedback.

Mode 5

Mode 5 is the last mode of the deployment sequence. At the end of mode 5, the spacecraft is operational and it switches to the two modes that perform the attitude control during the operations (mode 7 and 8).

Mode 5 has two different controllers: the elevation error controller and the azimuth error controller. Since the wheel rotation provides some stiffness to the Y axis in the inertial frame, the elevation error is relatively easy to control, and the magnetic moments applied by the Y torque have almost the same form as in the preceding cases: $M_y = -G_{nut}\ el\ B_x - G_{nut\,dot}\frac{dB_y}{dt}$. The first term corresponds to nutation dampening and the second term corresponds to nutation rate dampening (opposition to a rotation about the Y axis).

The azimuth controller is more complicated: the satellite does not have any rotational stiffness about the X and Z axes, so it must be controlled more carefully in azimuth. The heart of the azimuth controller is a Kalman filter with the following entries:

- A 4-component state vector: azimuth, azimuth rate (rotation about Y), wheel rate and wheel drag.
- A 2-component measure vector: sun sensor (MSS or FSS), wheel tachometer.
- A 2-component command vector: wheel torque, body torque Y (steering law).
- A 3-component noise vector: magnetometer bias and quantization, suns sensor quantization, tachometer quantization.
- The following disturbances: environmental torques (aerodynamic, magnetic,...), wheel drag and wheel torque noise.

Mode 6

This mode is responsible for the deployment of the solar paddles. No torques are commanded, the wheel speed is just maintained steady by the wheel controller (cf mode 4). The hot wax actuators that perform the deployment can be damaged if they are activated too long. The ACS exits mode 6 either if the paddles are deployed (which is detected by 4 switches) or if a timer has expired.

Mode 7 and 8

Mode 7 and 8 are the nominal ACS modes during the operations. Mode 7 is for orbit-day and mode 8 is for orbit-night (eclipse time). The principle is the same as in mode 5 except that the nutation damping gains (G_{nut} and G_{nutdot}) are smaller. Indeed during the operations phase the satellite is close to its nominal state (elevation=0 and azimuth=0) so the corrections are smaller than in mode 5. Smaller gains are sufficient to correct the attitude, and they make the correction smoother.

The difference between modes 7 and 8 resides in the Kalman filter measurement vector. The mode 5 and mode 7 measurement vectors are similar, whereas the mode 8 state vector does not include the azimuth angle given by the suns sensors since it operates during orbit-night. The sun sensor azimuth angle is replaced by the azimuth drift measurement given by the optical cameras. In order to account for the delay needed by the optical cameras to deliver azimuth drift, the delay estimate for optical camera azimuth drift measurement is added to the state vector.

Mode 9

Mode 9 is the back up manual mode. It is entered only via a ground command. All the actuators are set to zero and wait for a ground command.

2.5.2 Mode Switching Logic

The previous paragraph discussed the tasks of the ACS software within each mode. This paragraph is concerned with the links that relate the different modes.

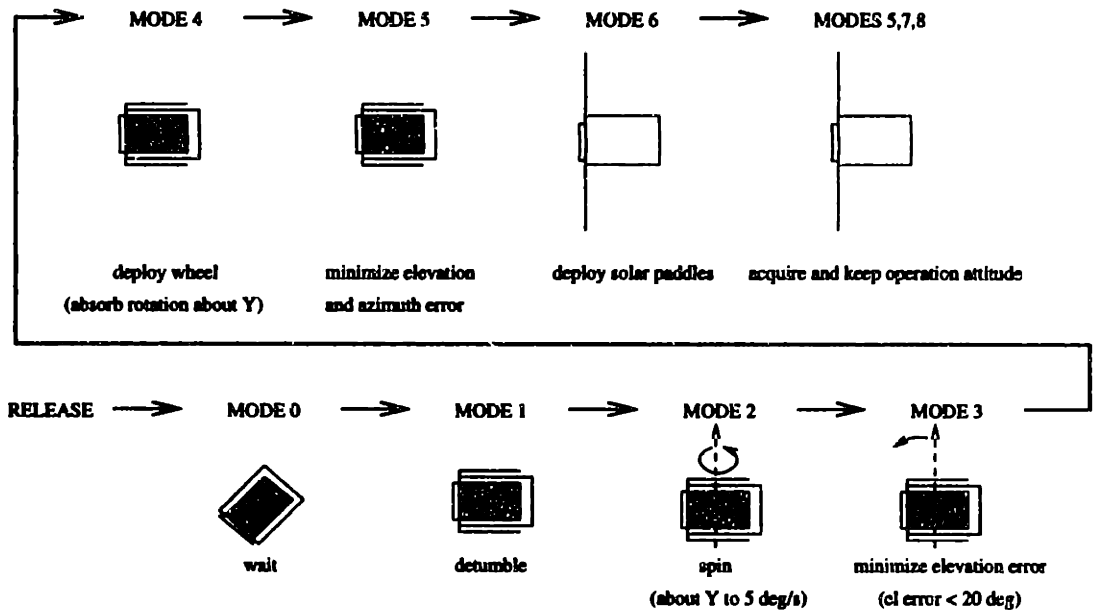

Figure 2-5: ACS modes

Figure 2-5 shows the nominal sequence from the release of the satellite by the rocket to the orbit night/day cycle. The first mode to be called is mode 0 (wait), so that the sensors have time to acquire some data and the filters can initialize. The ACS exits mode 0 simply when a certain delay has elapsed. When leaving mode 0, the satellite is still tumbling since no actuators have been activated since the release. The ACS switches from mode 0 to mode 1 (detumble). Mode 1 will cancel most of the rotation about all three axes. The ACS exits mode 1 when the rotation has been dampened enough, that is when the x and z components of the momentum vector drop below a certain threshold. Then the plan is to give the spacecraft some rotational stiffness about the Y axis so that the elevation angle can be controlled easily. This is the role of mode 2, which comes after mode 1. The mode 2 exit criteria also concerns the momentum vector: when the satellite momentum vector is close enough to the target momentum vector the spacecraft is ready to have its elevation angle error corrected and it switches to mode 3. Mode 3 precesses the momentum vector; once the elevation error is stabilized under a certain threshold, the ACS enters mode 4 to cancel the rotation about Y. Mode 4 deploys the momentum wheel: the wheel spins up to a speed such that it completely dampens the rotation about Y.

The wheel needs a certain delay to reach its speed, when this delay is elapsed, the ACS exits mode 4. Then the system may have to return to mode 3 if the wheel deployment has induced a disturbance strong enough for the elevation error to grow past the threshold. When the elevation is stabilized again, the ACS goes to mode 6 to have the satellite deploy its solar paddles. As mentioned in the previous section, the solar paddle release system must not be activated for too long, so the ACS leaves mode 6 either after the paddles are released or after the delay is elapsed. At this point the satellite is in its final configuration and it is then going to reach its final attitude by correcting the azimuth error as well as the elevation error. Modes 5, 7 and 8 all accomplish this, but they are used in different situations. Mode 5 has the higher gains, so it used only at the end of the acquisition sequence, when the azimuth error can be big, or when the spacecraft has been submitted to important disturbances that cause the azimuth error to go over its threshold. Mode 7 is the default mode during the mission at orbit-day. Mode 8 is the default mode when the optical cameras operate, that is during orbit-night.

Figure 2-6 shows a state diagram that describes the complete switching logic of the ACS. The acquisition sequence from mode 0 to modes 7/8 appears clearly, as well as some other paths that correspond to anomalous situations. In particular the ACS switches from mode 7 to mode 3 when mode 7 is not able to maintain a small elevation error while dealing with the azimuth error at the same time. It switches from mode 7 to mode 2 when it is orbit night and no tracking data is provided (problem with the optical cameras). In this case the ACS cannot use the sun sensors and no drift rate is available, so the best compromise is to go back to mode 2 (detumble) where only the magnetometers are used to keep the satellite roughly stable. When the sun rises, the ACS will go again through mode 3, 5 and 7 to resume normal operations. The same scheme occurs when the satellite is in mode 8 (orbit-night) and the data from the cameras drops out; or when the rotation of the satellite as determined by the magnetometers is too big for the ACS to cope with while in the tracking mode (in mode 8, the rotation rate must be small enough for the cameras to not blur). As in the previous case, the satellite will then enter mode 2, and wait for the sun to rise.

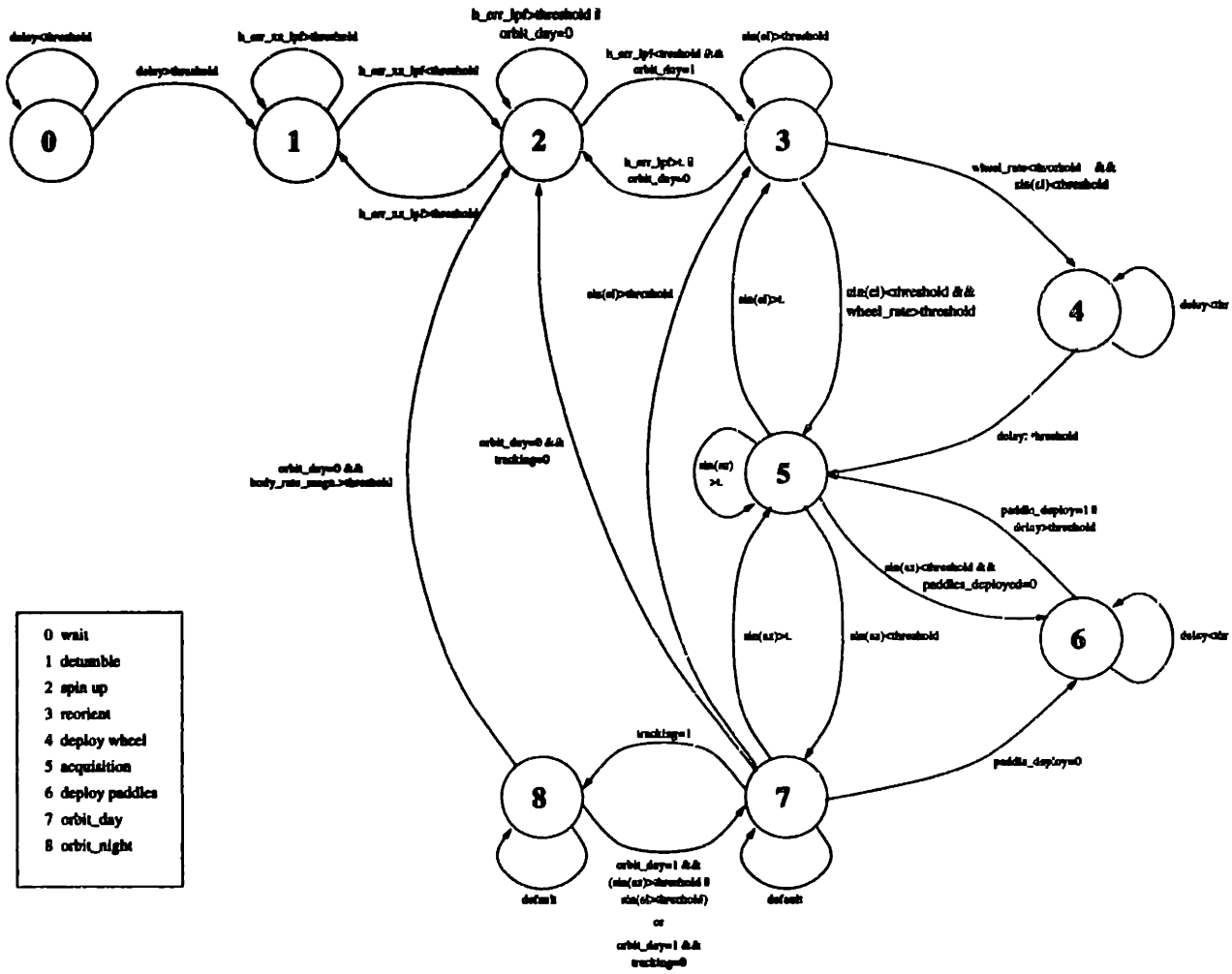

Figure 2-6: ACS switching logic

2.6 Conclusion

This section presented all the features that must be coded in the ACS software. It will serve as a basis for the design of the requirement-based test cases.

To summarize, these test cases will have to check the following points:

- The switching logic is correctly implemented (the switching between the modes occurs when the spacecraft is in the expected configuration).
- The behavior of each mode is correct (each mode performs the task it is designed for and does not corrupt any other parameter).
- The spacecraft meets the ACS requirements while it is on station (after it has gone through all the acquisition sequence, the spacecraft maintains a correct attitude so that the other subsystems can perform normally).

Chapter 3

Testing

This chapter is concerned with the actual testing of the HETE-2 Attitude Control System software.

The testing of ACS systems has always been a problem in the space industry since it is virtually impossible to control all the parameters that affect the system, such as the orientation of the sun, the components of the magnetic field, the gravity, the small perturbations that affect the spacecraft in orbit, etc. To exercise the full system in real on-orbit conditions, one would essentially need a 0G vacuum chamber with full control of the lighting and the magnetic field! For this reason the testing of the ACS hardware and the testing of the ACS software are decoupled. Each hardware item is verified on its own, and a simulation environment is created to provide the software with the information it expects, and to collect the commands it outputs.

For HETE-2, a complete simulation environment is available for the testing of the ACS.

- It is able to feed the ACS software with all the environment parameters corresponding to the position of the satellite (sun direction or orbit night, magnetic field, disturbance torques,...).
- It can simulate the dynamics of the spacecraft: given initial conditions, actuators commands and environment torques, the state of the satellite (rotation rates, pointing direction,...) are continuously updated.

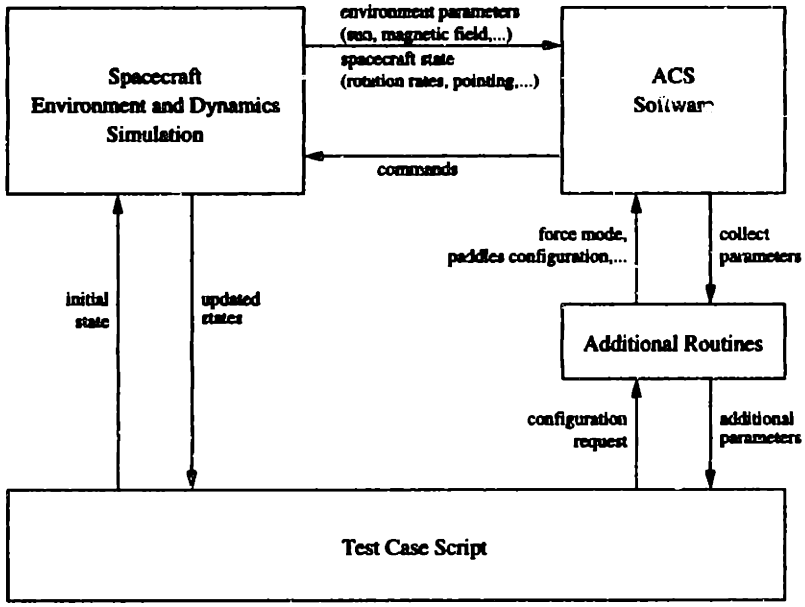

Figure 3-1: ACS testing environment

- It takes into account the commands generated by the software to update the state of the spacecraft's actuators.

Each test case is run via a script that sets the initial conditions of the system, calls the simulation program, launches the ACS software, and finally collects and displays the results. To allow better control of the software, some additional routines have been written. For example, these routines allow one to start the ACS in a certain mode (after staying in WAIT_MODE for a moment in order to initialize the filters), they collect directly the parameters that are of interest for the test, and they provide a complete control of the paddles deployment sequence, the AUX errors, the time, etc.

Using this set up, the test cases that are designed can be implemented quite easily, and they can be repeated as desired since all the data that is necessary for the initialization, and all the information-extracting process is stored in the script.

3.1 Black Box Testing

3.1.1 ACS Software Parameters

The following is a summary of all the parameters that appear in the specifications of the ACS software, and therefore are necessary for understanding the black box testing process.

Momentum Parameters

These parameters are directly related to the angular momentum of the spacecraft.

h_err_lpf Norm of the difference between the desired angular momentum and the actual angular momentum. For example in mode 2 (spin up mode) the satellite is spun up about the Y axis until it reaches a rate of about 6 deg/s. The momentum error is the difference between an angular momentum of 1 Nms about the Y axis (which corresponds to a rotation of 6 deg/s about Y), and the actual momentum of the spacecraft. "lpf" stands for "low pass filter": the angular momentum components cannot change very fast because of the inertia of the satellite, so they are filtered to minimize the effect of noise and possible misreadings and to prevent the ACS from over reacting after a temporary glitch in the sensor readings.

h_err_xz_lpf is the projection of h_err_lpf on the XZ plane of the satellite. It is used when it is necessary to control the rotation only about the X and Z axes, as in mode 1 (in this mode, the rotation about the Y axis is not to be considered since eventually the satellite will be spun up about this axis).

body_rate_magn: Magnitude of the rotation vector of the satellite as determined from the magnetometers.

The thresholds for these parameters are as follow:

H_ERR_SPINUP Threshold on h_err_lpf to move on from spinup mode (mode 2) to reorient mode (mode 3). It is set to 0.2 Nms.

H_ERR_XZ_LPF Threshold on h_err_xz_lpf to move on from detumble mode (mode 1) to spinup mode (mode 2). It is also set to 0.2 Nms.

OMEGA_COAST_THRESHOLD Threshold on body_rate_magnetometer to stay in orbit night mode. If the rotation is too high during orbit night, the ACS cameras will not be able to track, and the ACS must move on to another mode.

Sun Parameters

These parameters are directly related to the attitude of the satellite with respect to the sun.

sin_sun_el Sine of the elevation (angle between the direction of the sun and the YZ plane of the satellite).

sin_sun_az Sine of the azimuth (angle between the direction of the sun and the XZ plane of the satellite).

orbit_day Equal to 0 during eclipse time and to to 1 when the sun is in view. When orbit_day=0, no sun parameters are available, and sin_sun_el and sin_sun_az are set to zero.

The threshold for these parameters depends on the mode. The sun pointing gets more precise as the ACS gets closer to the on-station modes.

SIN_SUN_EL_STAY Threshold on sin_sun_el to quit the reorient mode (mode 3). It is set to 0.342, which corresponds to an elevation angle of 20 deg.

SIN_SUN_AZ_STAY Threshold on sin_sun_az to quit the acquisition mode. It is set to 0.1736, which corresponds to an azimuth angle of 10 deg.

SIN_SUN_EL_STAY_NIGHT, SIN_SUN_AZ_STAY_NIGHT Thresholds on sin_sun_el and sin_sun_az to stay in orbit night mode (mode 8 with orbit_day=1). They are set to 0.087156, which corresponds to an angle of 5 deg. This is directly related to the specification that requires a sun pointing accuracy of 5 degrees while the satellite is on station and it is orbit day.

Configuration Parameters

The mode switching logic also depends on the configuration of the satellite. The states of the paddles, ACS cameras, and wheel are stored in the following parameters.

paddles_deployed Equal to 1 if all the paddles are deployed, 0 otherwise.

tracking Equal to 1 if the ACS cameras are tracking stars and are able to pass on the azimuth drift to the ACS, 0 otherwise (Recall that the ACS cameras are the sensors that determine the attitude of the satellite for the ACS during orbit night).

wheel_rate Rotation speed of the wheel in rad/s.

The wheel produces a momentum of 1.0 Nms at its regulated speed (so reg_speed = $H_{desired}/I_{wheel}$ = 1.0/0.0106 = 94.3 rad/s). At the end of mode 3, if the wheel rate is not high enough, the ACS enters mode 4 to spin up the wheel to the regulated speed. The threshold for entering mode 4 is WHEEL_RATE_THRESHOLD=0.9 reg_speed=85 rad/s.

Also the wheel has a maximum speed WHEEL_SPEED_MAX=300 rad/s which must not be exceeded or the hardware could be damaged.

Delays

In addition to the logic that has already been described, each mode has a hysteresis delay that prevents the ACS from switching to another mode too soon. A certain amount of time must be spent in the current mode, regardless of the values of the other parameters before a switch to another mode is possible. This delay is used to avoid a situation where the ACS would toggle rapidly between two modes (for example because a parameter is very close to a limit), neither mode having enough time to significantly modify this parameter. Table 3.1 lists the delays for the different modes.

mode 0	WAIT_MODE_DELAY	10 s.
mode 1	DETUMBLE_MODE_DELAY	100 s.
mode 2	SPINUP_MODE_DELAY	100 s.
mode 3	REORIENT_MODE_DELAY	6000 s.
mode 4	DEPLOY_WHEEL_MODE_DELAY	180 s.
mode 5	ACQ_MODE_DELAY	10 s.
mode 6	DEPLOY_PADDLES_MODE_DELAY	120 s.
mode 7	ORBIT_DAY_MODE_DELAY	10 s.
mode 8	ORBIT_NIGHT_MODE_DELAY	10 s.
mode 9	GROUND_COMMAND_MODE_DELAY	10 s.

Table 3.1: Mode delays

3.1.2 Black Box Testing Organization

The ACS specifications being divided into 9 distinct modes, it is possible to give a fixed structure to the requirement-based testing. The same types of tests must be run for every mode, so the testing process will always follow the same scheme with the three following steps:

Switching logic testing In this step the goal is to verify that the ACS enters and exits the modes when the parameters take on the expected value and when the mode delay is elapsed. This is in fact equivalent to verifying that the switching logic diagram (figure 2-6) is correctly implemented.

Parameter testing The ACS software sees the satellite through a set of parameters, which are then exploited to decide to send some commands to the actuators or to do a mode switch. For the ACS to perform correctly, the integrity of these parameters is crucial, hence it must be verified that they really reflect the physical state of the satellite. For example in detumble mode (mode 1), if the rotation about the X or Y axis is increased, h_err_xz_lpf should increase accordingly. Then the ACS software will be able to determine the value of the current in the X coil and be able to switch to mode 2 at the right time.

Functional testing Finally it is necessary to make sure that each mode accomplishes its fundamental task correctly. Basically, mode 1 is called to reduce the

test case	time_in_mode	outcome
1	8	mode 0
2	9	mode 1
3	10	mode 1
4	0	mode 0
5	-1	mode 0
6	-2	mode 1
7	2147483648	mode 1, negative time_in_mode

Table 3.2: Mode 0 switching logic test cases

rotation of the satellite about the X and Z axes, mode 2 is called to spin up the satellite about the Y axis, mode 3 is called reduce the elevation error, mode 5 is called to reduce the azimuth error, etc. We want to check that the ACS interprets the parameters correctly, and issues the right commands that have the expected impact on the spacecraft's attitude.

These three different types of tests will cover efficiently the specifications of every mode.

The following paragraphs describe, mode by mode, the most significant test cases of the black box testing process. The plots of interest for parameter testing and functional testing are shown below, as well as tables summing up the switching logic tests.

3.1.3 Mode 0

Switching Logic Testing

Table 3.2 summarizes the switching tests for mode 0.

Test cases 1, 2, 3 and 4 show that in a nominal situation, the ACS stays in mode 0 as long as time_in_mode is less than 10 seconds (time_in_mode starts at 0); it switches to mode 1 as soon as the delay is elapsed; and if for some reason it is still in mode 0 and time_in_mode is greater than the delay, it switches immediately to mode 1. This is the expected behavior.

Test cases 5 and 6 are robustness test cases, in that sense that the variable time_in_mode is forced to take on a negative value, which should never happen in a normal situation. If time_in_mode happens to be negative and less than -2, the ACS proceeds directly to the following mode which is rather unexpected. In case of a bad initialization for this variable when the mode initiates, there would be an unwanted mode switching.

In fact time_in_mode is declared as a long unsigned integer, so it should never be able to take on negative values, as in test cases 5 and 6. But somewhere in the code it gets converted to a long signed integer (cf test case 6: a value bigger than 2147483647, which is the maximum range of long signed int in ANSI C, results in a negative value). So every negative time_in_mode smaller the -2, every time_in_mode greater than 2147483647 will result in a mode switch.

The upper boundary of 2147483647 s. is again not a concern in nominal cases, since it exceeds by far even the life of the satellite.

This set of test cases can be repeated in all the modes, and always leads to the same results. That is, in every mode, an out of range value of time_in_mode will bring the ACS into the next mode if the other parameters allow it even if the mode delay is not over.

There is no parameter testing nor functional testing for this mode since its only task is to wait for the delay to elapse.

3.1.4 Mode 1

Switching Logic Testing

The switching logic that handles the transition from mode 1 to mode 2 has been verified with the test cases listed in table 3.3.

The first test checks that if h_err_xz_lpf is smaller than the threshold (H_ERR_XZ_LPF=0.2Nms) at the beginning of the mode, the ACS will switch to mode 2 as soon as the delay is complete. The delay for mode 1 is 100 s., as expected.

When the spacecraft has a non-zero initial rotation, h_err_xz_lpf is not yet in a

test case	h_err_xz_lpf	outcome
1	<threshold	mode 2
2	<thres. at delay, >thres. at delay+1	mode 2
3	<thres. at delay-1, >thres. at delay	mode 1
4	>threshold	mode 1

Table 3.3: Mode 1 switching logic test cases

steady state at the beginning of mode 1: it keeps increasing because the low pass filter is not completely initialized. So h_err_xz_lpf can grow past the threshold once the ACS has switched to mode 2, even if it remains under the thresholds while in mode 1. Tests 2 and 3 explore this limit case, and it is found that in the worst case, the ACS can enter mode 2 with a rate of 8.5 deg/s. This corresponds to the situation where h_err_xz_lpf hits the threshold on the first sample of mode 2 (the ACS will stay in this later mode at least until the delay is elapsed).

After the delay is complete, the ACS will switch to the following mode as soon as h_err_xz_lpf drops under the threshold (which has the right value, 0.2 Nms), as shown in test 4.

Parameter Testing

The goal of this paragraph is to show that the rotation vector of the satellite and h_err_xz_lpf are related to each other in a sensible way so that the algorithms of mode 5 work on a good basis.

Figure 3-2 shows the influence of OMEGA_X, OMEGA_Y and OMEGA_Z (the rotation rates about X, Y and Z) on h_err_xz_lpf.

OMEGA_X When OMEGA_Y and OMEGA_Z are fixed, h_err_xz_lpf is supposed to be a linear function of OMEGA_X. For positive rotation rates, this is true for OMEGA_X smaller than 30 deg/s. Beyond this value, h_err_xz_lpf decreases when the rate increases, certainly because the magnetometers cannot keep up with such high rates. This is not a problem because the maximal rotation rate is well below 30

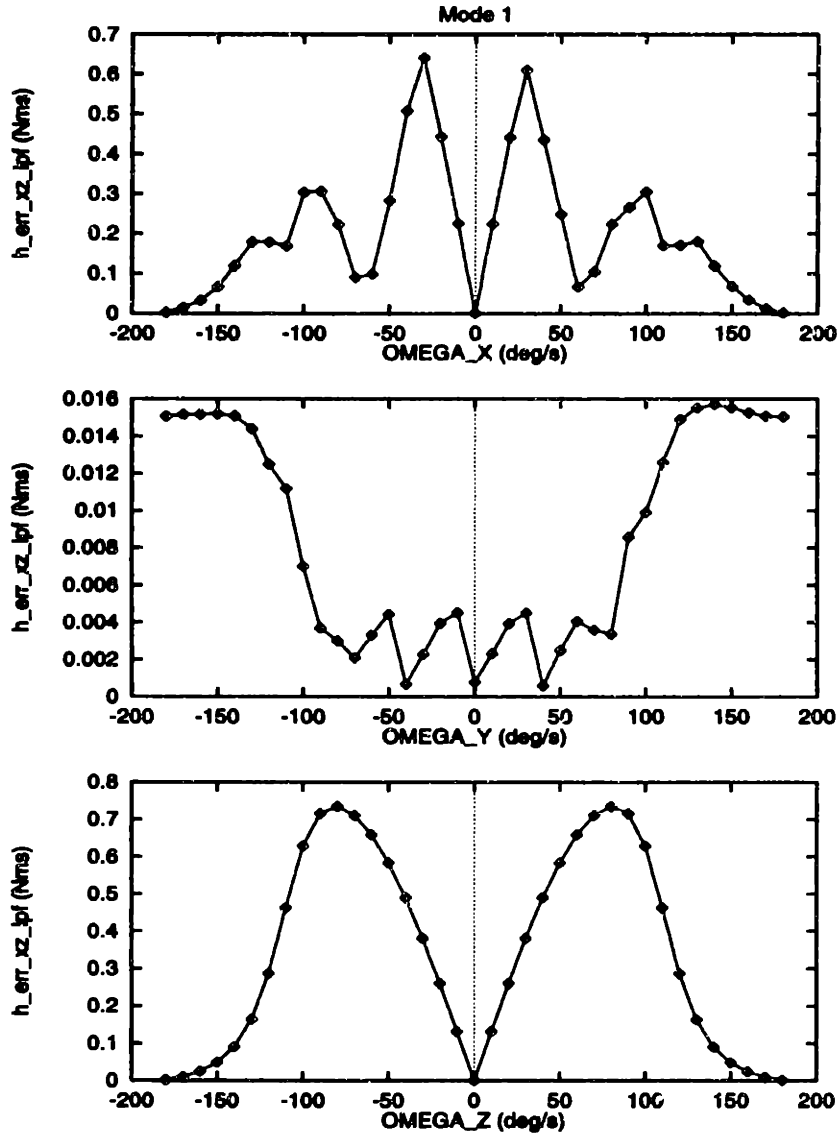

Figure 3-2: Mode 1 parameter testing

deg/s (recall that the maximal rate when the satellite is released from the rocket is 5 deg/s); moreover, if the magnetometer outputs briefly an erroneous value of, say, 70 deg/s, this will not lead to a sudden drop of h_err_xz_lpf since the data is low-pass filtered. So this situation would not result in a sudden switch to mode 2.

The negative part of the plot has the same shape as the positive part of the plot, in particular note that h_err_xz_lpf is positive (the threshold is a positive value, so in fact it refers to the absolute value of the error).

OMEGA_Y h_err_xz_lpf is the projection of the momentum error on the XZ plane, so it should not be sensitive to a rotation about Y. The second plot shows that indeed a rotation about Y has little influence on h_err_xz_lpf. Even the low-inertia axis, Z, has more impact on the momentum error for a given rate.

OMEGA_Z The plot for OMEGA_Z is very similar to the plot of OMEGA_X so the same comments about the drop of h_err_xz_lpf after OMEGA_Z=90 deg/s can be made.

The main difference is that the Z axis has far less inertia than the X axis, so the same rate generates a smaller momentum error. A rotation rate of about 25 deg/s would be needed to make h_err_xz_lpf grow beyond the threshold at the end of mode 1 (the data which is plotted is taken after the ACS has spent 100 s. in mode 1). This will never be achieved in practice, so the conclusion is that the switching between mode 1 and mode 2 is made almost regardless of the rotation about Z.

This is not an issue if any reasonable rotation about Z is damped before mode 1 delay is elapsed. The functional testing will prove this to be true.

Functional Testing

In this paragraph we test the ability of the mode 2 algorithm to detumble the satellite, i.e. to decrease the momentum error h_err_xz_lpf.

Figure 3-3 plots the momentum error versus time for an initial rotation rate of 5 deg/s about X. According to the specification, this is the worst rotation rate that

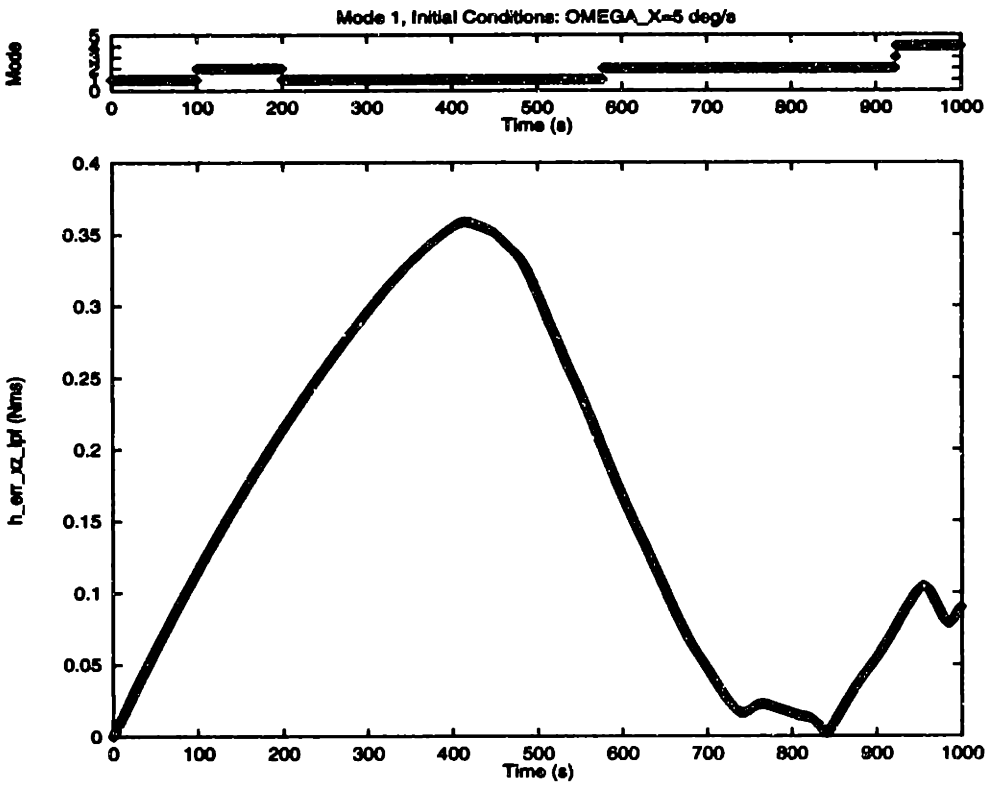

Figure 3-3: Mode 1 functional testing

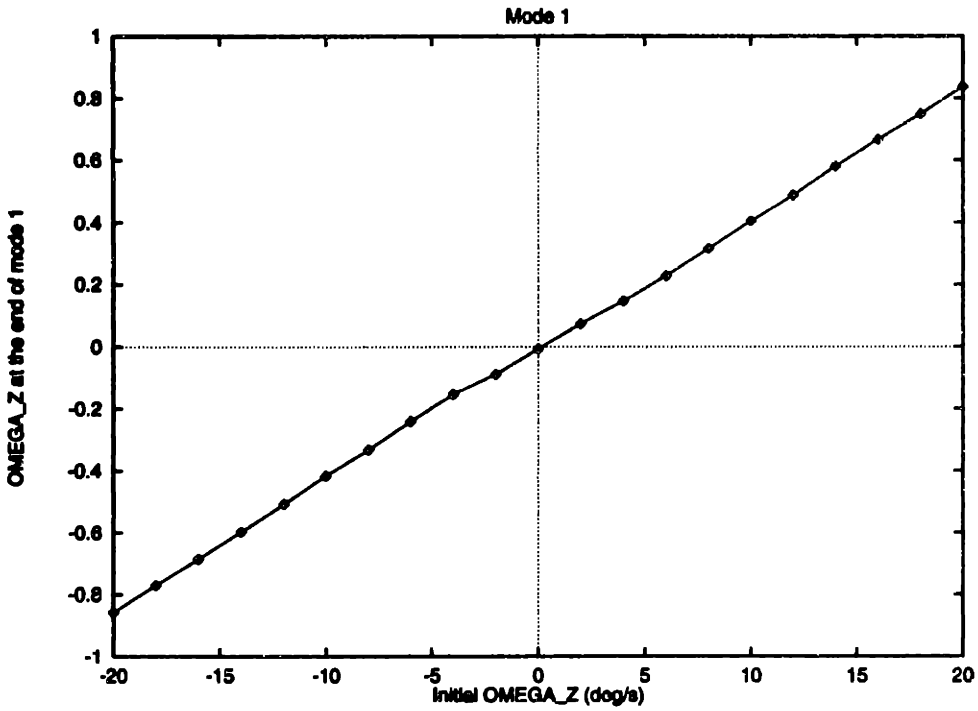

Figure 3-4: Mode 1 functional testing (continued)

the satellite can have when it is released form the satellite. At the end of the mode 1 delay, the filter is not fully initialized, so h_err_xz_lpf is underestimated and the ACS switches to mode 2 while the momentum error is still growing. When the mode 2 delay is over, the ACS switches to mode 1 and this time mode 1 has a chance to correctly accomplish its task. h_err_xz_lpf drops quickly and when the ACS switches to mode 2 for the second time, the spacecraft is not tumbling any more.

The very same kind of plot is obtained for negative OMEGA_Xs, which proves that the system controls the coils correctly for both clockwise and counterclockwise rotations.

Likewise, for the limit case described in the switching testing (OMEGA_X=8.59 deg/s), the satellite is detumbled after only one mode 1 - mode 2 oscillation.

Figure 3-4 addresses the issue of the Z axis which, as explained above, does not have enough inertia to drive the momentum error. The plot shows that whatever the initial rotation rate about Z is, mode 1 delay is long enough so that this rotation is almost completely damped. As a comparison, a threshold of 0.2 Nms for the X axis

corresponds to a rotation rate of OMEGA_X = h_err_xz_lpf/I_{xx} = 0.2/8.82 = 1.30 deg/s. Therefore the low influence of OMEGA_Z on h_err_xz_lpf is not a problem.

Note that as was mentioned before, the magnetometer body rate estimation is only sensitive to the component orthogonal to the magnetic field. So the test cases involving this estimation have been repeated for different initial orientations to make sure this problem does not corrupt the algorithm.

3.1.5 Mode 2

Switching Logic Testing

Two parameters drive the outcome of mode 2 switching logic, h_err_xz_lpf and h_err_lpf. orbit_day also plays a role, as the ACS is not allowed to move on to a further mode when orbit_day=0 (eclipse time). Let's start with orbit_day=1.

In the first two test cases (see table 3.4), h_err_lpf is greater than the threshold (H_ERROR_SPINUP=0.2Nms), since OMEGA_Y is set to zero. The ACS either stays in mode 2 if h_err_xz_lpf is under its threshold, or it switches to mode 1 after the correct delay (100 s.).

In the two following test cases, h_err_lpf and h_err_xz_lpf are low, and the switching to mode 3 is made when mode 2 delay is over.

It is not possible to find a test case with h_err_lpf<threshold and h_err_xz_lpf>threshold at the same time since h_err_xz_lpf is the projection of h_err_lpf on the XZ plane, and the thresholds for both momentum errors are equal. This statement was verified with a routine that tried all the possible combinations of OMEGA_X and OMEGA_Y, and displayed the resulting momentum errors.

The same set of tests has been run for orbit_day=0 leaving the switching to mode 1 unaltered, and disabling the switching to mode 3 (mode 3 uses sun sensor information for its control law, hence it does not make sense to go into this mode during the night).

test case	orbit_day	h_err_xz_lpf	h_err_xz_lpf	outcome
1	0	<threshold	>threshold	mode 2
2	0	>threshold	>threshold	mode 1
3	0	<threshold	>threshold	mode 2
4	0	<threshold	>threshold	mode 2

Table 3.4: Mode 2 switching logic test cases

Parameter Testing

In the same fashion as in mode 1, the goal of this paragraph is to check the relationship between the rotation rates and the momentum errors. This should insure that the control laws are fed with sensible parameters.

The target angular momentum for mode 2 is 1.0 Nms about the Y axis. This corresponds to a spin about Y of OMEGA_Y = $h_{desired}/I_{yy}$ = 6.16 deg/s. Therefore if the spacecraft is stable (OMEGAs=0), the momentum error should be 1.0 Nms, and if the spacecraft is spinning about Y at 6.16 deg/s, the momentum error should be zero.

Figure 3-5 is a plot of the momentum error h_err_lpf versus the rotation rate about Y, after the ACS has spent 100 seconds in mode 2 (minimum initializing time for the filters, ACS free to switch to another mode). The initial rotation rates about X and Z are zero.

As expected the momentum error is a linear function of OMEGA_Y, and it has a value of about 1.0 at the origin. But the curve is not very steep, and the momentum error cancels only for a very high OMEGA_Y (almost 50 deg/s). This is due to the initialization of the filter. After only 100 seconds, the filter is not yet in a steady state, and h_err_lpf is under estimated. As the filter stabilized, h_err_lpf really reflects the rotation rates and the minimum of the plot approaches 6.16 deg/s. For example, it takes 830 seconds to have h_err_lpf drop below the threshold for a perfect initial state (OMEGA_X=OMEGA_Z=0, OMEGA_Y=6.16 deg/s). And after the regular delay of 100 seconds, the switching to mode 3 occurs for a Y rate as high as 35 deg/s.

This behavior has no consequence on the integrity of the ACS if it enters mode

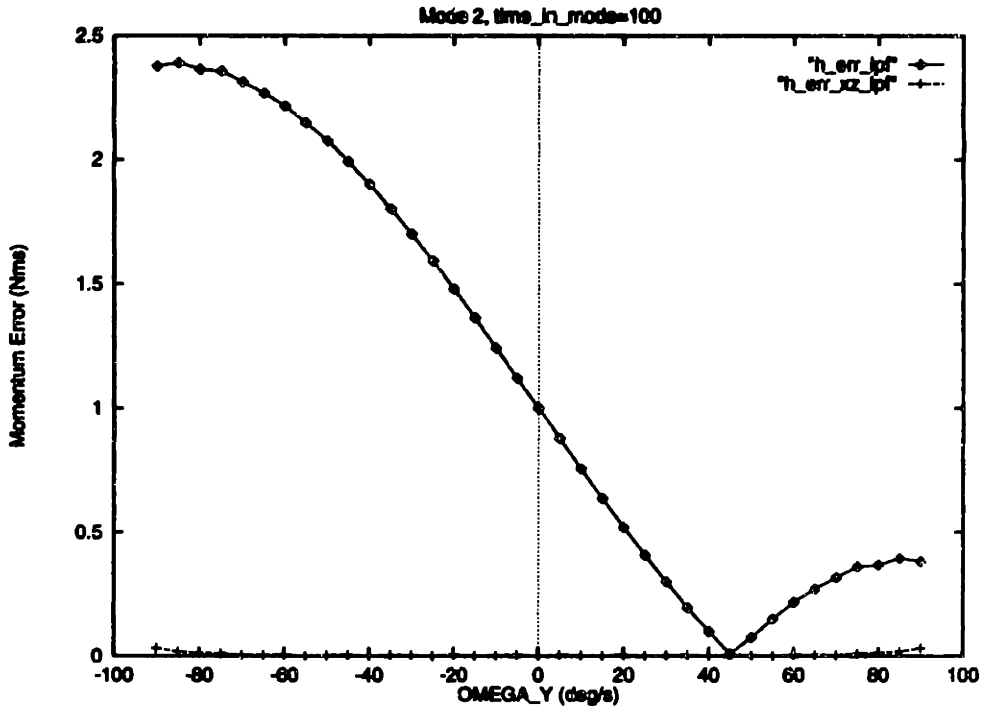

Figure 3-5: Mode 2 parameter testing

2 with a rotation rate about Y which is less or equal to 6.16 deg/s. The momentum error will be abnormally high, but this will only result in a longer stay in mode 2, to give time to the filter and to h_err_lpf to settle (see also next paragraph, the extra time in mode 2 does not cause the ACS to over spin the satellite noticeably). So if the spacecraft state is within the specifications when it is released by the rocket (rotation rate about the all three axes less than 5 deg/s), this behavior is not a problem.

On the other hand, if for some reason the ACS enters mode 2 with an abnormally high Y rate, mode 2 will not have enough time to damp all the extra spin, as mode 3 will take over sooner than expected. This has not caused any problem for reasonable rotation rates. As shown below, the following modes can cope with the extra spinning. So finally this situation is safe.

As expected, h_err_lpf keeps increasing when OMEGA_Y gets negative: the sign of the rotation matters, the spin up must be positive so that it will be canceled by the deployment of the wheel.

h_err_xz_lpf shows the same characteristics as in mode 1.

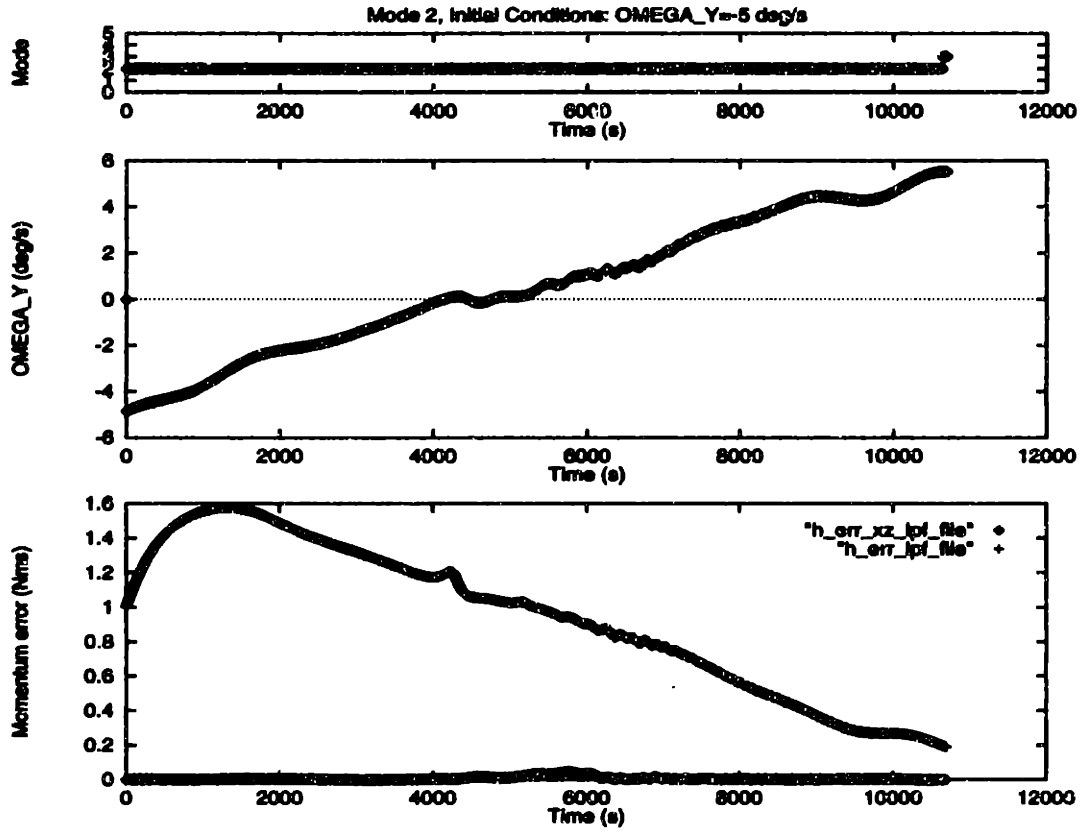

Figure 3-6: Mode 2 functional testing

Functional Testing

The fundamental function of mode 2 is to spin the satellite up to a target angular momentum of 1.0 Nms about the Y axis.

Whether it has a positive or a negative initial rate about Y, the satellite must end up spinning counterclockwise (positive rotation). In figure 3-6, the initial rate is negative, at the limit of the release specifications for Y (OMEGA_Y=-5 deg/s). So the control law has to first cancel the negative rotation, then spin the satellite up the other way. This is the worst case for nominal conditions.

The ACS behaves as expected, slowly bringing the Y rotation to the correct target rate (the initial rate is less than the target rate, so the early switching described above does not take place). The process takes quite a long time (2 hours 45 minutes for

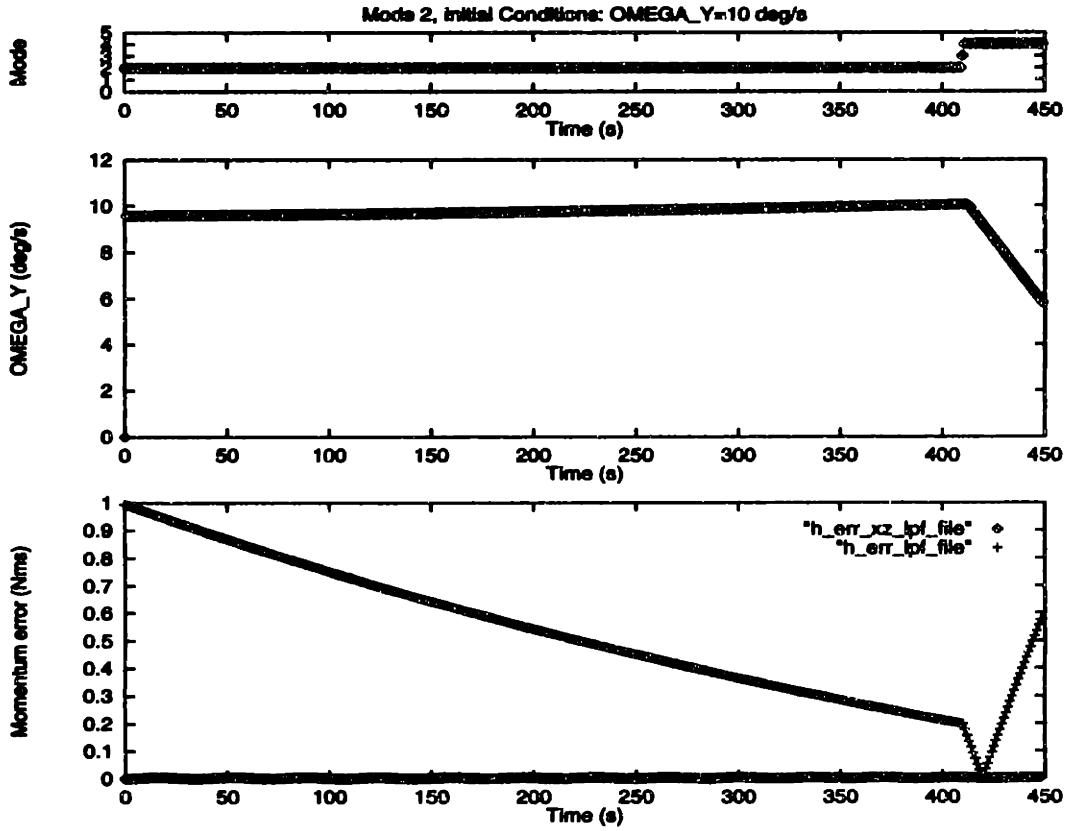

Figure 3-7: Mode 2 functional testing (continued)

OMEGA_Y=-5 deg/s, about 50 minutes when the satellite is initially at rest).

The same plot for an initial OMEGA_Y equal to 10 deg/s (greater than the target rate, then inducing an early switching) is shown in figure 3-7. h_err_lpf drops under the threshold well before the Y rotation speed reaches its target, causing the ACS to switch to mode 3 while the rotation rate is still well above 6.16 deg/s. The dynamics of the spin up are quite slow, and when the switching occurs, OMEGA_Y is not noticeably lower than 10 deg/s.

As a consequence, there will be a residual positive rotation about Y after the deployment of the wheel, which could cause mode 5 to have some difficulties to acquire the sun azimuth. We will see later that mode 5 is able to damp the extra rotation, so the problem does not persist after this step.

3.1.6 Mode 3

Switching Logic Testing

The outcome of mode 3 switching logic is driven by the following parameters:

- h_err_lpf (switch back to mode 2 if the spin up error is too big).
- time_in_mode (don't switch to any mode before mode 3 delay is elapsed).
- orbit_day (switch back to mode 2 at night).
- sin_sun_el (proceed to further mode if the elevation is small enough).
- wheel_rate (spin up the wheel before going to mode 5 if it is not spinning fast enough).

The 34 test cases listed in table 3.5 were used to check this logic.

Test cases 1 to 4: For these tests the momentum error h_err_lpf is above the threshold, so the ACS should switch back to mode 2 as soon as the delay is elapsed, whatever the other parameters are. This is not at all what happens, rather the software ignores the delay and the momentum error, and switches to the following mode (4 or 5) as soon as sin_sun_el drops under the threshold.

Test cases 5 to 8: The only difference with the above tests is that now time_in_mode is greater than mode 3 delay. In this case the ACS has a normal behavior and it always switches back to mode 2 due to the momentum error.

The conclusion is that there is an error in the switching logic implementation for mode 3. The delay *is not* taken into account for a switching induced by sin_sun_el; it *is* taken into account for a switching induced by h_err_lpf. Then during the first 6000 seconds of mode 3, if sin_sun_el drops below the threshold the ACS will proceed to mode 5 regardless of the value of h_err_lpf. This can have some serious consequences since the satellite can go into mode 4 or 5 with arbitrary large momentum errors.

test	h_err_lpf	orbit_day	time_in_mode	wheel_rate	sin(el)	outcome
1	>thres.	1	<delay	<thres.	<thres.	mode 4, no delay
2	>thres.	1	<delay	<thres.	>thres.	mode 3
3	>thres.	1	<delay	>thres.	<thres.	mode 5, no delay
4	>thres.	1	<delay	>thres.	>thres.	mode 3
5	>thres.	1	>delay	<thres.	<thres.	mode 2
6	>thres.	1	>delay	<thres.	>thres.	mode 2
7	>thres.	1	>delay	>thres.	<thres.	mode 2
8	>thres.	1	>delay	>thres.	>thres.	mode 2
9	>thres.	0	<delay	<thres.	<thres.	mode 2, no delay
10	>thres.	0	<delay	<thres.	>thres.	mode 2, no delay
11	>thres.	0	<delay	>thres.	<thres.	mode 2, no delay
12	>thres.	0	<delay	>thres.	>thres.	mode 2, no delay
13	>thres.	0	>delay	<thres.	<thres.	mode 2, no delay
14	>thres.	0	>delay	<thres.	>thres.	mode 2, no delay
15	>thres.	0	>delay	>thres.	<thres.	mode 2, no delay
16	>thres.	0	>delay	>thres.	>thres.	mode 2, no delay
17	<thres.	1	<delay	<thres.	<thres.	mode 4, no delay
18	<thres.	1	<delay	<thres.	>thres.	mode 3
19	<thres.	1	<delay	>thres.	<thres.	mode 5, no delay
20	<thres.	1	<delay	>thres.	>thres.	mode 3
21	<thres.	1	>delay	<thres.	<thres.	mode 4, no delay
22	<thres.	1	>delay	<thres.	>thres.	mode 3
23	<thres.	1	>delay	>thres.	<thres.	mode 5
24	<thres.	1	>delay	>thres.	>thres.	mode 5
25	<thres.	0	<delay	<thres.	<thres.	mode 2, no delay
26	<thres.	0	<delay	<thres.	>thres.	mode 2, no delay
27	<thres.	0	<delay	>thres.	<thres.	mode 2, no delay
28	<thres.	0	<delay	>thres.	>thres.	mode 2, no delay
29	<thres.	0	>delay	<thres.	<thres.	mode 2, no delay
30	<thres.	0	>delay	<thres.	>thres.	mode 2, no delay
31	<thres.	0	>delay	>thres.	<thres.	mode 2, no delay
32	<thres.	0	>delay	>thres.	>thres.	mode 2, no delay
33	<thres.	0	>delay	<thres.	<thres., <0	mode 4, no delay
34	<thres.	0	>delay	<thres.	>thres., <0	mode 4, no delay

Table 3.5: Mode 3 switching logic test cases

Test cases 9 to 16: Same parameters as above, except orbit_day=0. The ACS switches to mode 2, and it does not wait for any delay to do so. It is not very clear from the specifications, but this is the expected behavior for all the modes concerned with orbit_day.

Test cases 17 to 24: Now the momentum error h_err_lpf is smaller than the threshold and when the elevation error is under the 20 deg. threshold, the ACS goes to mode 5 if the wheel speed is high enough, to mode 5 otherwise. This is the correct behavior, but the delay is not respected. The switching occurs whether time_in_mode is smaller or greater than the delay. This error is directly linked to the one described above.

Test cases 25 to 32: Same parameters as above, except for orbit_night=0. As required, orbit_day=0 masks all the other parameters and has the ACS switch to mode 2 without waiting for the end of the delay.

Test cases 33 and 34: Tests for the switching logic related to sin_sun_el when the elevation is negative. h_err_lpf is under the threshold, so a switch to mode 4 (mode 5 when the wheel spins fast enough) is expected when the elevation angle is smaller than 20 deg. In test case 33 the elevation is about -10 deg, and as expected the ACS switches to mode 4. In test case 34 the elevation is about -30 deg and the ACS switches again to mode 4. This is not supposed to happen! In fact whenever the sine of the elevation is negative (ie whenever the elevation is negative, since the elevation ranges from -179 to +180 deg) the ACS considers that the elevation is under the threshold and proceeds to the next mode. This is obviously wrong, since this can cause the ACS to leave mode 3 with an elevation of 179 deg.

The origins of the errors described above were easy to localize in the source code. For sin_sun_el taking control during the delay, it was a confusion in the if-then-else branching logic that handles mode 3 switching. For the negative elevation angle problem, it was a missing abs() in front of sin_sun_el.

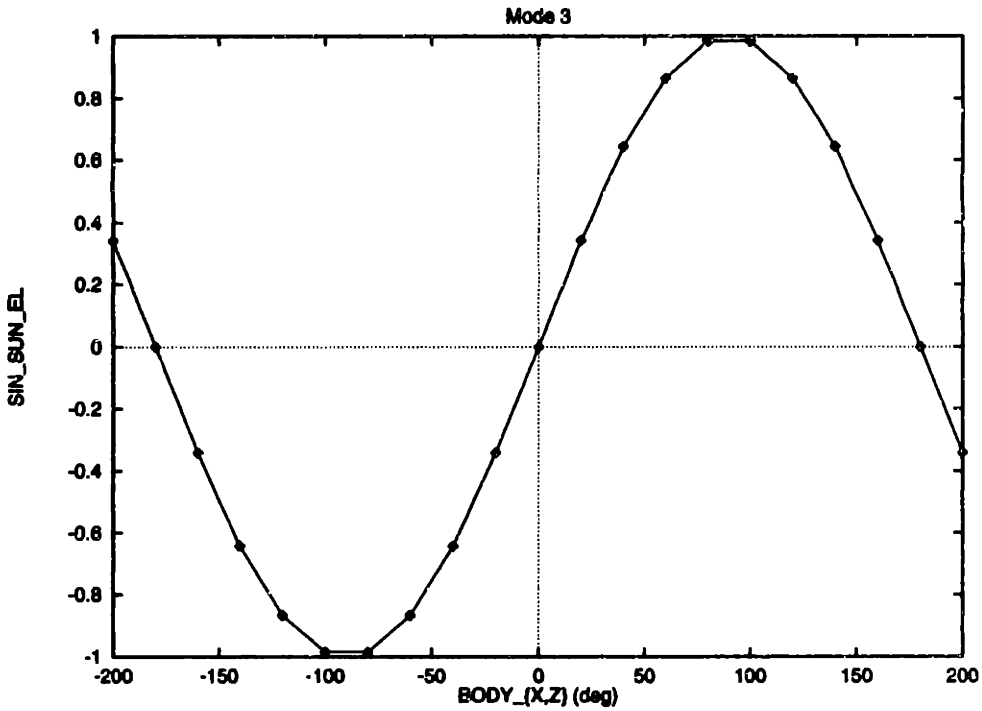

Figure 3-8: Mode 3 parameter testing

Parameter Testing

The goal of this paragraph is to check the relationship between the parameter sin_sun_el and the position of the satellite with respect to the sun.

The ACS simulation environment allows the initial position of the satellite to be set via three parameters, BODY_X, BODY_Y, and BODY_Z. These parameters represent the angles between the satellite axis system and the ECI coordinate system (+X pointing toward vernal equinox, +Z pointing out the celestial pole, and +Y defined by right hand rule).

So fixing BODY_Y, and varying BODY_X and BODY_Z directly translates into a variation in elevation. This is pictured in figure 3-8: as the satellite moves around its Y axis, sin_sun_el is a sine curve corresponding to the elevation going from -180 to 180 deg.

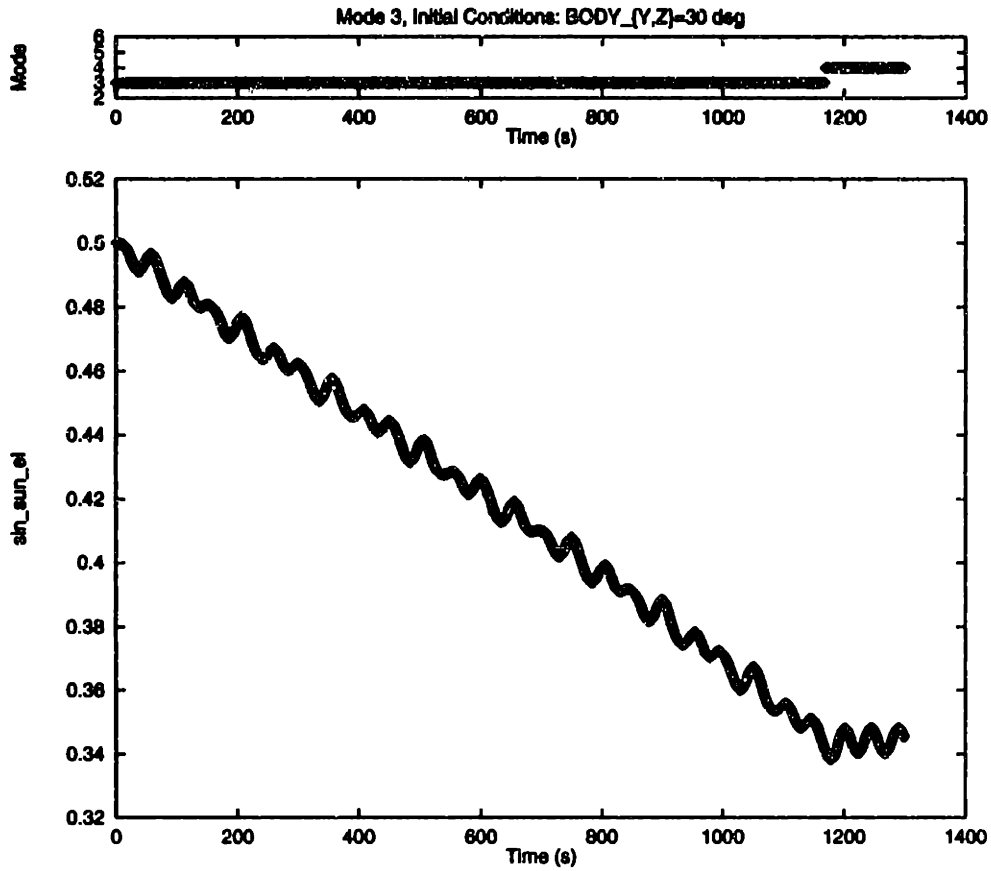

Figure 3-9: Mode 3 functional testing

Functional Testing

The task of this mode is to reduce the elevation down to a value between -20 and +20 deg. As shown in the switching logic paragraph, there are some situations where this will not happen: whenever the elevation is negative, the ACS moves on to mode 4 or 5, and the mode 3 control law does not even have a chance to be applied. Since this part of the testing is concerned with the control law, all the simulations were run for positive elevations.

Figure 3-9 plots the sin_sun_el versus time for an initial elevation angle of 30 deg, for a nominal spin rate (the tests have also shown that the algorithm works for any spin rate, it is just slower for higher spin rates, and faster for smaller spin rates). Mode 3 has sin_sun_el decrease all the way down to its threshold; because of the

rotational stiffness provided by the spinning about Y, this is a rather long process. The worst case is when the initial elevation is 90 deg: it takes several orbits to reorient the satellite.

3.1.7 Mode 4

Switching Logic Testing

The wheel deploy mode switching logic is only based on the time the ACS has spent in this mode, as for mode 0. So the same kind of test cases were run, namely:

- Test case 1: Checks that the ACS does not leave mode 5 before the delay is complete.
- Test case 2: Checks that the ACS goes to mode 5 as soon as the delay is complete.
- Test case 3: If time_in_mode>delay and the ACS has not, for some reason, already switched to mode 5, it does so immediately.

Functional Testing

During mode 4, the satellite must spin the momentum wheel up to the regulated speed (95 rad/s) to transfer the angular momentum of the spinning satellite body to the wheel.

In the ideal case, where the initial wheel rate is zero and the initial body rate about Y is exactly the mode 3 target rate (6.16 deg/s), the wheel speed ramps up to 95 rad/s and the satellite rotation about Y is completely canceled (cf figure 3-10).

If the wheel has a non zero initial speed, and/or if the initial OMEGA_Y is not exactly 6.16deg/s, the wheel speed still goes to 95 rad/s, but there will be some remaining momentum at the end of mode 4. This is the way the system was designed: mode 5 wheel controller will damp the extra angular momentum of the satellite and regulate the wheel speed.

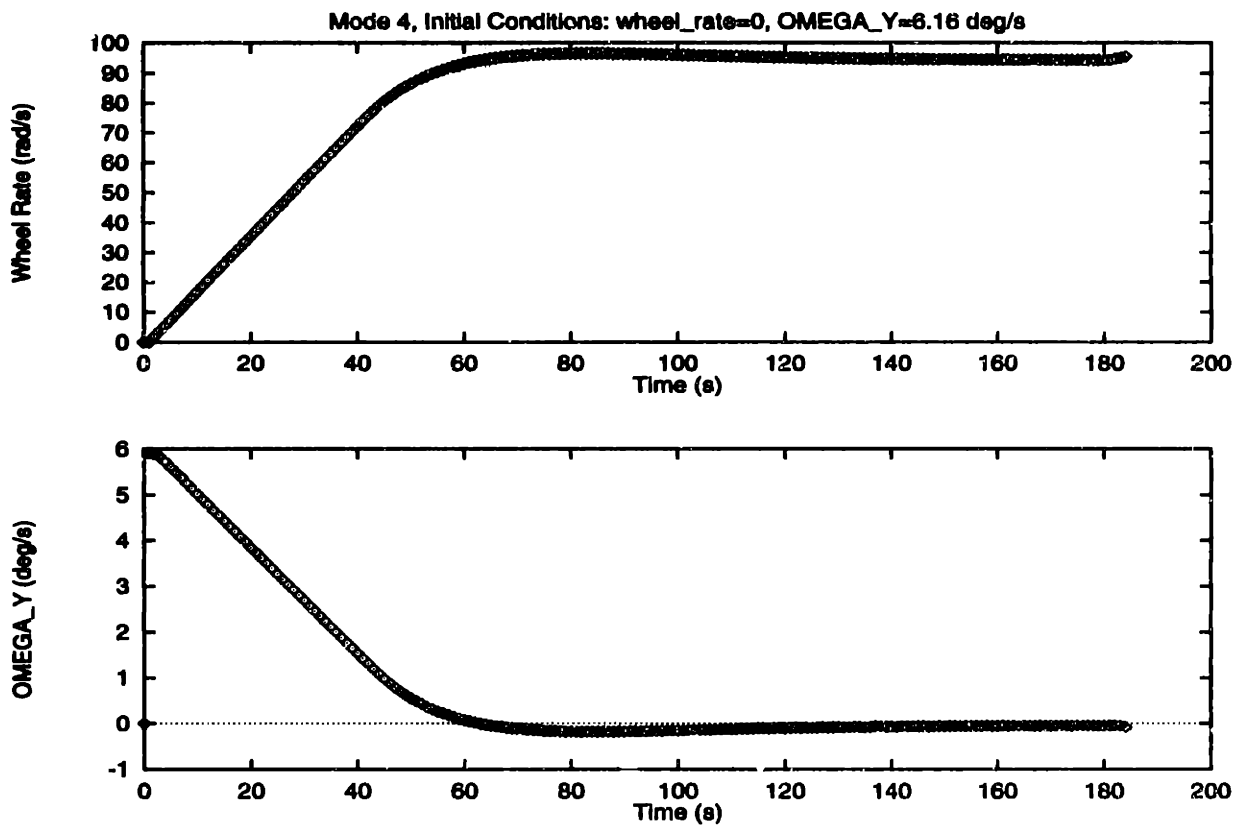

Figure 3-10: Mode 4 functional testing

test case	paddles_deployed	orbit_day	sin_sun_el	sin_sun_az	outcome
1	0	1	<threshold	<threshold	mode 6
2	0	1	>threshold	>threshold	mode 3
3	0	1	<threshold	>threshold	mode 5
4	0	1	>threshold	<threshold	mode 3
5	0	0	N/A	N/A	mode 2
6	0	1	>threshold, <0	<threshold	mode 6
7	0	1	<threshold	>threshold, <0	mode 6
8	1	1	<threshold	<threshold	mode 7
9	1	1	>threshold	>threshold	mode 3
10	1	1	<threshold	>threshold	mode 5
11	1	1	>threshold	<threshold	mode 3
12	1	0	N/A	N/A	mode 2
13	1	1	>threshold, <0	<threshold	mode 7
14	1	1	<threshold	>threshold, <0	mode 7

Table 3.6: Mode 5 switching logic test cases

3.1.8 Mode 5

Switching Logic Testing

Three parameters must be taken into account for testing mode 5 switching logic: sin_sun_el, sin_sun_az, orbit_day, and the paddles configuration (stowed/deployed). The threshold for the elevation is the same as in mode 5 (20 deg, sin_sun_el=0.342). The threshold for the azimuth is one half (10 deg, sin_sun_az=0.1736). If orbit_day=0, the ACS has to leave mode 5, as the attitude is computed with respect to the sun in this mode. Paddles_deployed decides if the ACS must go directly to mode 7 (orbit day) when the acquisition is done, or if it must first go to mode 6 (deploy paddles).

The first half of the test cases are for a "paddles stowed" configuration (cf table 3.6.

Test case 1 sin_sun_el and sin_sun_az are under their respective thresholds, so mode 5 criteria are already met, and the ACS switches to mode 6 (deploy paddles) as soon as the delay (10 s.) is over.

Test case 2 sin_sun_az and sin_sun_el are both above the threshold, so the ACS must go back to mode 3 to reorient the XZ plane first. Note that when the ACS follows the path mode 2→mode 3, the satellite enters mode 3 with OMEGA_Y $\simeq 6$ deg/s, whereas when the ACS follows the path mode 5→mode 3, the satellite enters mode 3 with OMEGA_Y $\simeq 0$. But in the later case, the wheel is spinning, so the total angular momentum is the same. Simulations were run to confirm that the behavior of mode 3 is the same in the two cases.

Test case 3 This is the nominal state for entering mode 5. The ACS stays in mode 5 until sin_sun_az drops under its threshold. This test also checks that the threshold for the azimuth angle is 10 deg.

Test case 4 Even if sin_sun_az is under its threshold, if sin_sun_el is too big, the ACS goes back to mode 3 (at the end of the delay).

Test case 5 Orbit night. As in mode 3, as soon as orbit_day=0, the ACS switches to mode 2, which is the orbit night safe mode (this transition is not shown on the switching diagram).

There is no required delay here, which makes sense because there is no reason to stay in mode 5 when there is no sun. On the other hand, if the sun sensors have a misreading regarding orbit night/orbit day (due to a power drop, for example) the ACS will immediately go back to mode 2, and there is no second chance. This results in a waste of time, but it is a safe behavior.

Test cases 6 and 7 In the same fashion as in mode 3, the program is wrong for negative elevation and azimuth. No matter how big (in absolute value) the elevation and the azimuth are, if they are negative, the software will proceed to mode 7. In consequence the satellite can enter mode 7 with arbitrarily high elevation or azimuth angles.

In the source code, it is the same problem of missing abs()'s in the function that handles the mode switching.

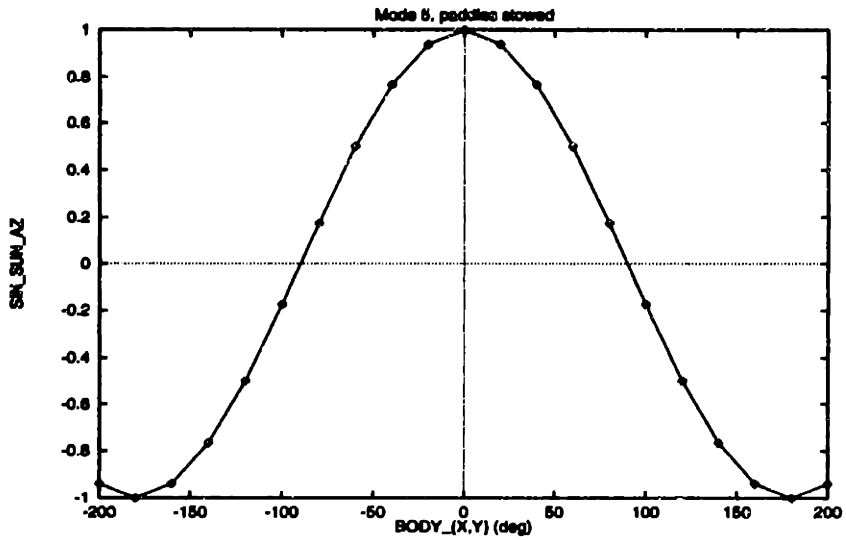

Figure 3-11: Mode 5 parameter testing

Test cases 8 to 14 The last test cases are for a "paddles deployed" configuration. There is no difference between the "paddles deployed" and the "paddles stowed" results, except that instead of switching to mode 6 to deploy the paddles, the ACS directly switches to mode 7.

Parameter Testing

The same kind of tests as in mode 3 are run, except that this time both sin_sun_el and sin_sun_az must be checked. Recall that in the coordinate system used for the simulation a variation in BODY_X and BODY_Z, while holding BODY_Y fixed, directly translates into a variation in the elevation. In the same manner, a variation in BODY_X and BODY_Y, while holding BODY_Z fixed, directly translates into a variation in the azimuth.

Also the satellite can be in mode 5 with its paddles deployed or stowed. This makes a difference for the determination of the sun pointing parameters, since some of the sun sensors are located on the paddles and do not have the same pointing direction if the paddles are stowed or if they are deployed (see section 2.2). So the software must take the paddles configuration into account to correctly evaluate sin_sun_el and sin_sun_az.

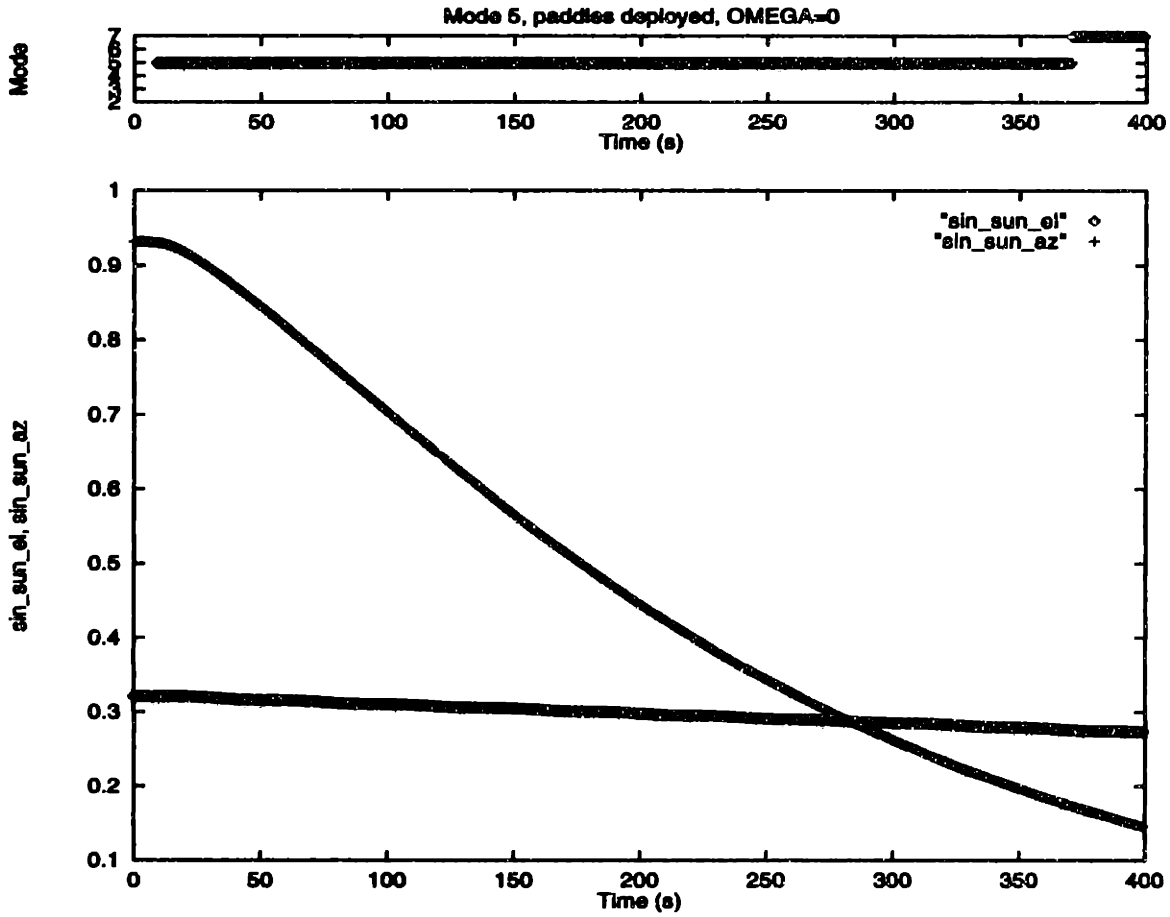

Figure 3-12: Mode 5 functional testing

The same results as in mode 3 are obtained for sin_sun_el, in the two different paddle configurations. The plot for sin_sun_az is shown in figure 3-11, in the paddle stowed configuration but it is identical in the paddles deployed configuration. The conclusion is that the sun pointing parameters are correctly evaluated in mode 5, and that the routine which is responsible for this evaluation accounts correctly for the paddle configuration.

Functional Testing

This mode is in charge of acquiring the sun pointing attitude, it must bring the sun azimuth under 10 deg, and maintain the elevation under 20 deg.

Figure 3-12 shows a basic check, where the spacecraft enters mode 5 in a nominal configuration, that is, a regulated speed of 95 rad/s for the wheel and no parasite

rotation[1]. At the beginning of the simulation, the azimuth is about 70 deg, and the elevation is near the threshold (10 deg). As expected, the spacecraft rotates to decrease the azimuth and the elevation. It takes much longer to modify the elevation than the azimuth because of the stiffness provided by the rotation of the wheel about the Y axis. Also, the torques used to control the elevation angle are generated by the magnetic coils, whereas the torques used to control the azimuth are generated by the wheel, which is much more powerful.

The plot obtained for the same initial conditions, but with the paddles stowed is completely similar, which shows that the CSS positions and the inertia matrix are correctly updated.

The introduction of perturbations like residual OMEGA_X or OMEGA_Z causes a nutation of the rotation vector around the Y axis related to the gyroscopic effect of the wheel: an oscillating componant at about twice the rotation speed of the wheel appears in sin_sun_el and sin_sun_az. If the perturbations are reasonable. they will be damped by mode 5 and 7 (on a much longer time scale than for sin_sun_el and sin_sun_az, though). If the perturbations get too big, the induced nutation causes important errors in sin_sun_el and sin_sun_az, causing the ACS to go back the preceding modes to cancel the residual rotation.

A residual rotation about the Y axis causes another sort of problem. As shown in figure 3-13, at the beginning of mode 5 the rotation is not canceled instantaneously, so if it has the right sign, it can cause the azimuth to decrease "artificially" (ie not because of the control law but because of the residual rotation) and trigger an unwanted switch to mode 7. The simulations have shown that this is not a problem because mode 7 can damp the extra rotation (but it takes longer than in mode 5), or switch back to mode 5.

sin_sun_az is zero for two azimuth angles: 0 and 180 deg. We must make sure that azimuth=180 deg is not a stable position, since in this attitude, the satellite is pointing directly *away* from the sun. For the two following situations, the initial conditions are again nominal for mode 5 (stable satellite, regulated wheel speed, zero

[1]All the tests begin with a 20-second stay in mode 0 to allow the filters to initialize

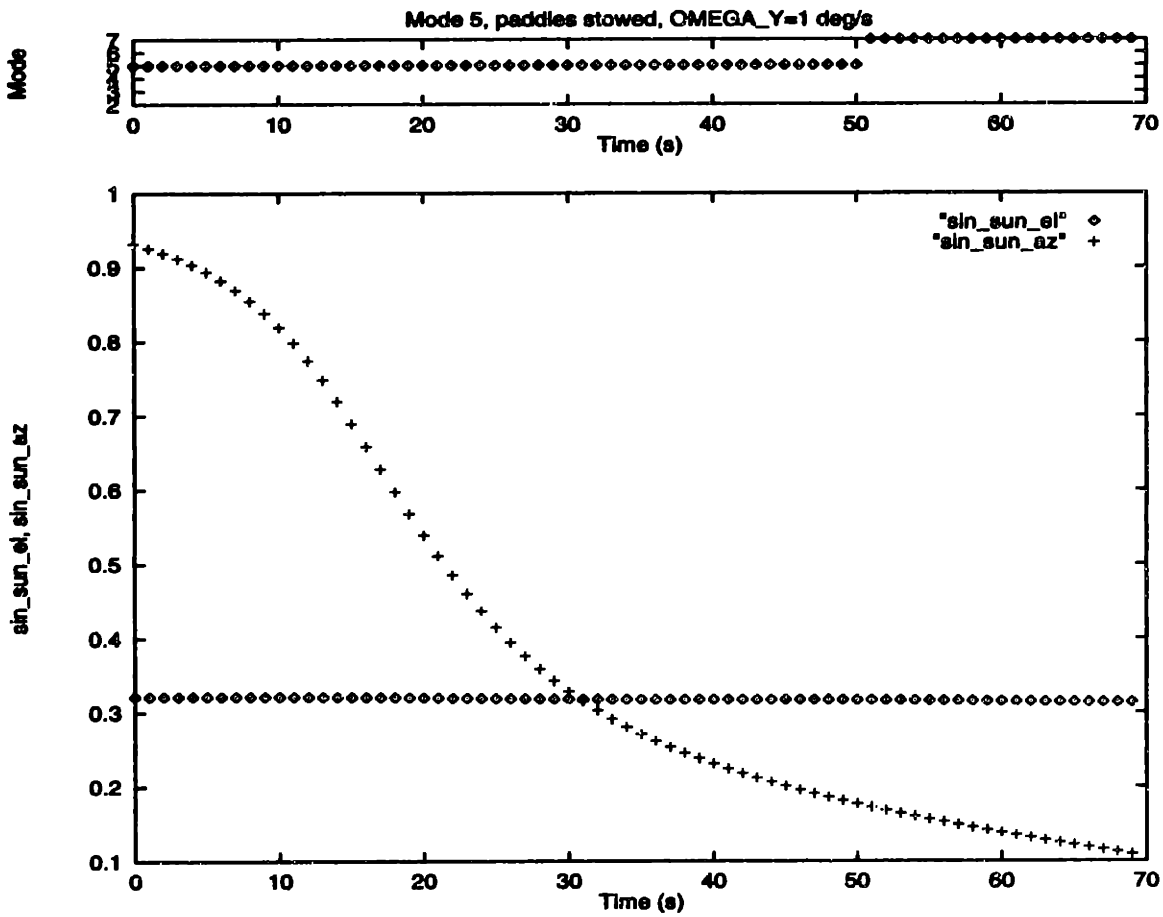

Figure 3-13: Mode 5 functional testing (continued)

elevation).

In the first plot of figure 3-14, the initial azimuth is about 170 deg, and the situation is correctly handled by the ACS: sin_sun_az grows until it reaches 1, and then tends to zero. This corresponds to the satellite drifting away from a near 180 deg. azimuth angle (low sin_sun_az), passing through the position azimuth=90 deg (sin_sun_az=1), then going to the zero azimuth sun pointing attitude.

But we get a very different scheme for initial azimuth angles ranging from 174 deg to 180 deg (figure 3-14). In this situation mode 5 tries to bring the azimuth to zero, which is what is expected, but when the delay is over, sin_sun_az is still under the threshold (we start from azimuth=180 degrees, hence a very low sin_sun_az). So the ACS switches to mode 6 and deploys the paddles. Meanwhile sin_sun_az has grown past the threshold, so the ACS switches back to mode 5. At this point the software seems to be very confused: sin_sun_az keeps on growing and it stabilizes around 0.9. It should normally go to zero.

Note that this situation (paddles deployed and satellite pointing away from the sun) is not very healthy from a power and heat budget point of view. The spacecraft finally recovers when the night comes and the ACS switches to mode 2. Then the satellite is spun up again, and it is very unlikely that the azimuth will be between 174 and 180 deg again.

The azimuth range 174-180 deg corresponds to the cases where sin_sun_az does not have time to grow above the threshold before mode 5 delay is over. So the paddles are deployed in a high azimuth attitude and this seems to generate some problems.

When the paddles are already deployed, everything is normal.

An easy fix for this problem is to make mode 5 delay big enough so that the configuration that leads to this situation never occurs. But the core of the problem is not here, and the acquisition controller has been modified in the next version of the code to eliminate this behavior.

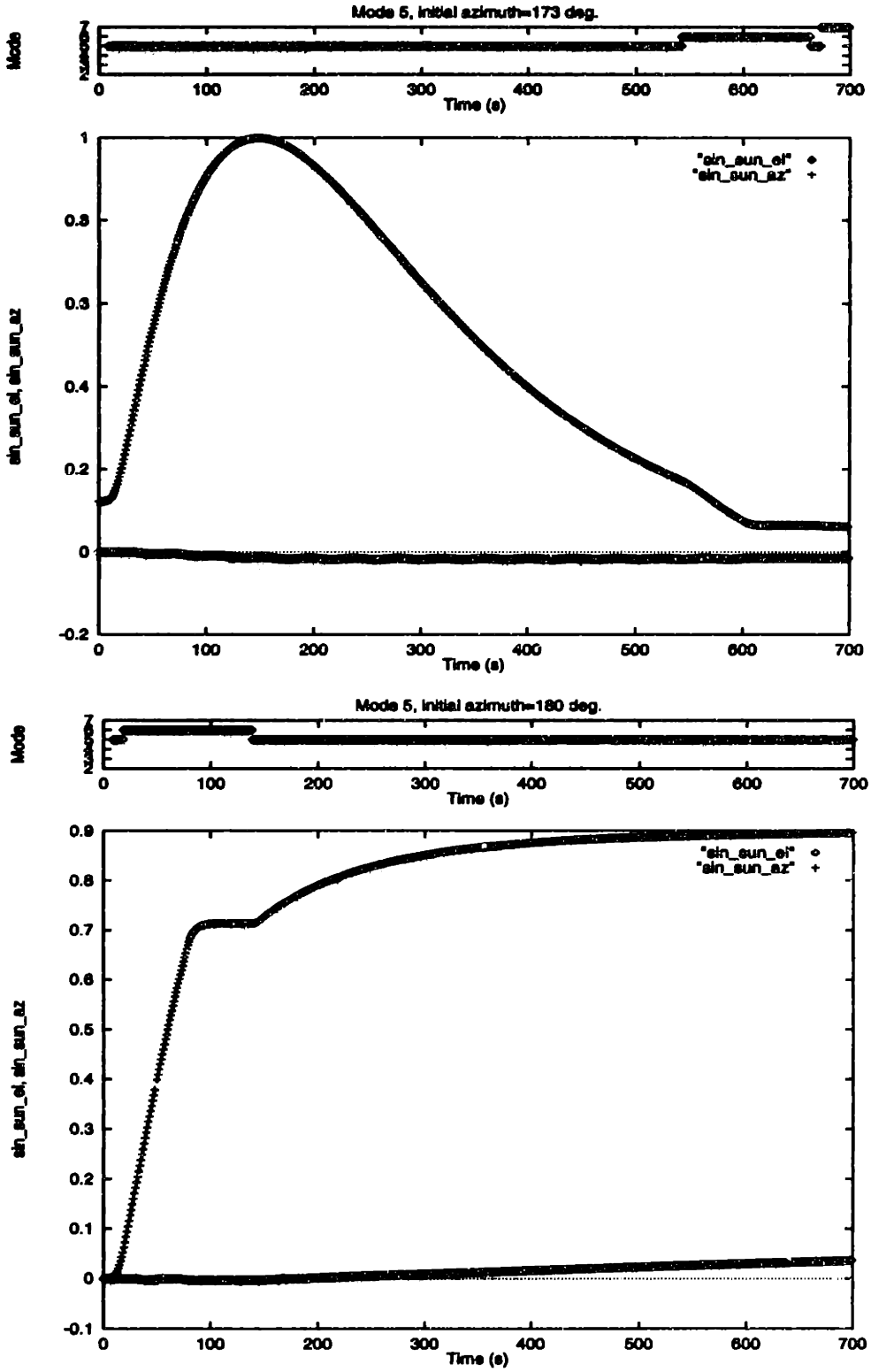

Figure 3-14: Mode 5 functional testing (continued)

3.1.9 Mode 6

Mode 6 is for the deployment of the solar paddles. A command triggering the four paddle actuators initiates the deployment.

The software has access to the paddle configuration via four sensors, and this data is stored in four bits of an ACS sensor variable (one bit per paddles, 0=stowed, 1=deployed).

The ACS delay is 120 seconds, and the switching tests show that according to the sensors, the deployment begins after 30 seconds in the mode and is complete when the delay is elapsed. The correct delay is implemented, and the following mode is always mode 5, as specified in the transition diagram.

The two following test cases were for abnormal deployment sequences, for example when one paddle fails to deploy. The code that deals with this situation was added after this testing process began, and this version of the code does not have the right behavior: the tests show that if at least one of the paddles is deployed, the ACS proceeds to mode 5 and so on as if the deployment had been completely successful.

3.1.10 Mode 7

Switching Logic Testing

The switching logic of mode 7 is very similar to mode 5, except that the azimuth must remain under the threshold in order not to go back to mode 5, and as soon as the attitude data is available from the cameras, mode 8 takes over.

The same type of tests as in mode 5 were made, and the results for the "paddles deployed" configuration are listed in table 3.7. The new logic related to the camera tracking information shows no default: the ACS goes into mode 8 if mode 7 meets its requirements and as soon as the cameras deliver data. Otherwise the same comment as in mode 5 can be made, in particular, the problem with the negative elevation or azimuth is also present in this mode.

The same results are obtained when the paddles are stowed, except that when the sun pointing is correct or when it is night and the cameras are tracking, mode 6 is

test case	tracking	orbit_day	sin_sun_el	sin_sun_az	outcome
1	0	1	<threshold	<threshold	mode 2
2	0	1	>threshold	>threshold	mode 3
3	0	1	<threshold	>threshold	mode 5
4	0	1	>threshold	<threshold	mode 3
5	0	0	N/A	N/A	mode 2
6	0	1	>threshold, <0	<threshold	mode 7
7	0	1	<threshold	>threshold, <0	mode 7
8	1	1	<threshold	<threshold	mode 8
9	1	1	>threshold	>threshold	mode 3
10	1	1	<threshold	>threshold	mode 5
11	1	1	>threshold	<threshold	mode 3
12	1	0	N/A	N/A	mode 8
13	1	1	>threshold, <0	<threshold	mode 8
14	1	1	<threshold	>threshold, <0	mode 8

Table 3.7: Mode 7 switching logic test cases, paddles deployed

activated to deploy the paddles.

Parameter Testing

Mode 7 uses the same parameters as mode 5, so the same tests were run, and they resulted in the same correct results (correct computation of sin_sun_el and sin_sun_az whether the paddles are deployed or not).

Functional Testing

The functional testing of mode 7 is divided into three main points. First, we must check that mode 7 can cope with the imperfections of the preceding modes and will recover from these non-nominal situations. Second, the "steady state" regime (i.e. when the ACS is engaged in the orbit day/orbit night loop) must be proven to meet the high level requirements: mode 7 is the on-station day mode, so the sun pointing must have a ± 5 degrees precision. Finally, the ACS must comply with the requirements for entering orbit night, so the drift rate must stay under 20 arcmin/s.

As seen before, there are some conditions for which the satellite leaves the modes before the on-station modes in a non-ideal configuration (residual rotation or non

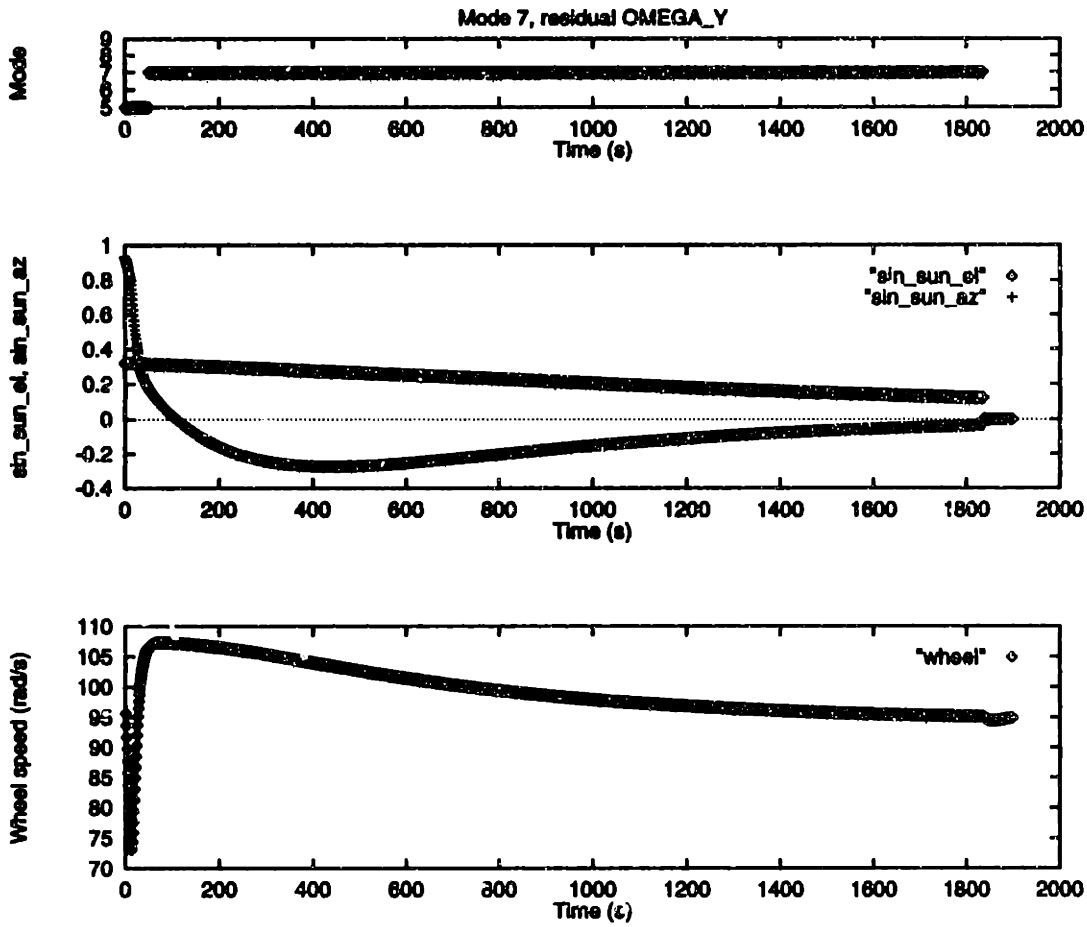

Figure 3-15: Mode 7 functional testing

nominal sun pointing error). In all the simulations the ACS has recovered from this kind of situation.

In particular , as seen in section 3.1.8, some values of OMEGA_Y can cause the ACS to bypass mode 5 almost completely and then go to mode 7 with a residual rotation about Y and an abnormal azimuth error. For managable errors, like a rotation rate of a few deg/s on one axis, mode 7 is robust enough to cancel the error itself. For instance, figure 3-15 shows that mode 7 can handle the situation generated by the test case cited above. When the error gets unreasonably big, the ACS switches back to the crude but more efficient preceding modes.

Once the ACS settles in the orbit night/orbit day cycle, the satellite is considered to be on station and the sun pointing accuracy must be better than 5 degrees (sine(el) less than 0.087). Figure 3-16 gives an idea of what the on-station "steady state"

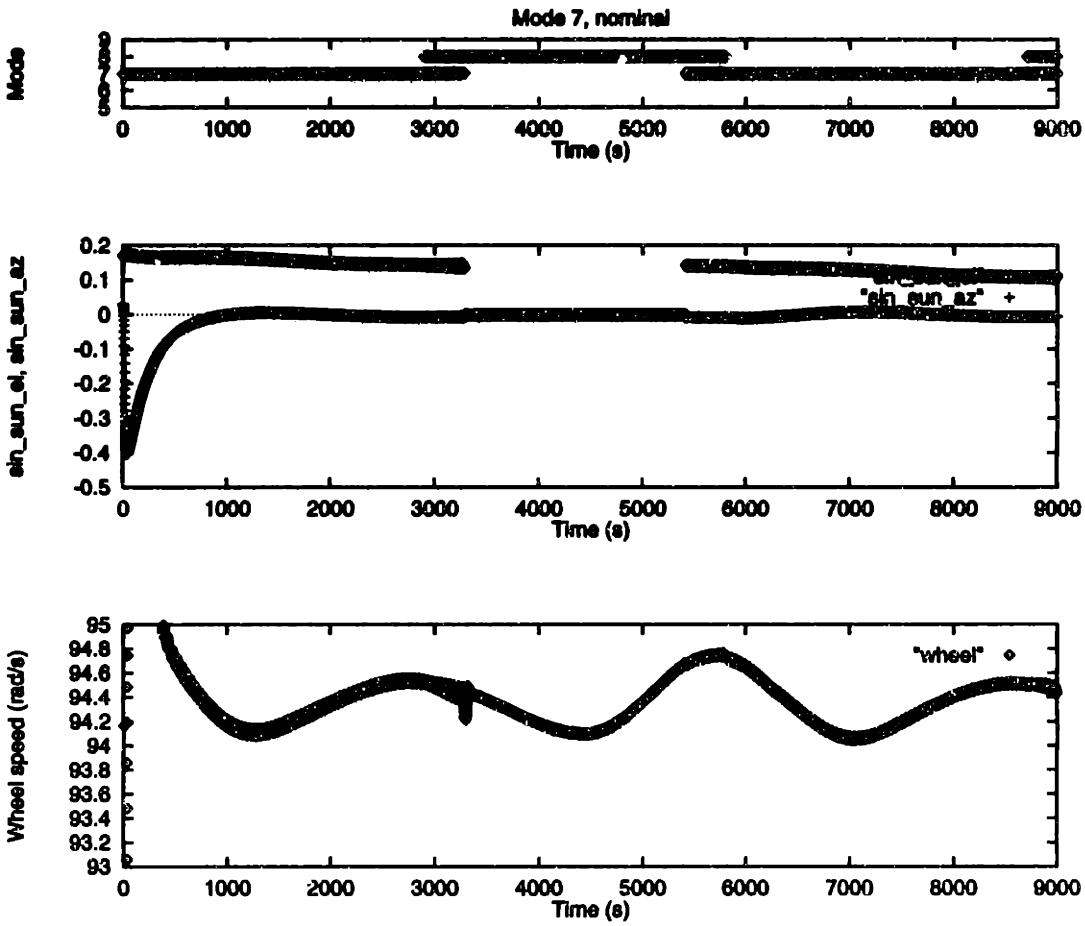

Figure 3-16: Mode 7 functional testing (continued)

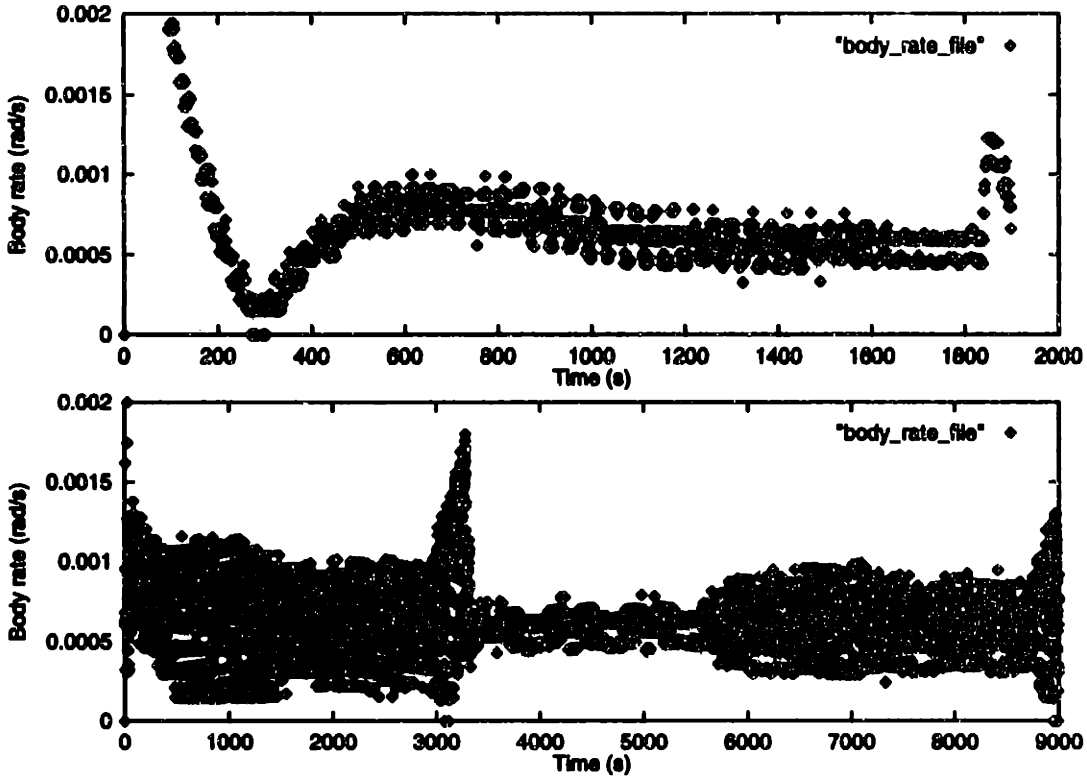

Figure 3-17: Mode 7 functional testing, corresponding body rates

simulations look like. At the very beginning, the azimuth is positive, above the threshold. It is corrected and this causes an overshoot before the angle drops and remains close to zero. The dynamics for the elevation angle is slower, and the elevation errors cancellation takes several orbits. But the gyroscopic effect of the wheel also guarantees the stability of the elevation angle, and once the error is damped, the elevation will stay close to zero. Also note that the ACS manages the wheel speed correctly: the speed never goes too far away from the regulated speed, and the wheel controller unloads the wheel without inducing perturbations. No simulation showing a non respect of the sun pointing requirements could be found.

For all the simulations , we kept track of the satellite drifting rate (body_rate_magnetometer), as it is a very important parameter for the switching from mode 7 to mode 8: in order for the ACS cameras to lock correctly at the beginning of orbit night, the drift rate must be smaller than 20 arcmin/s (5.82e-3 rad/s) during the transition from day to night. Mode 7 could never be proven to let a drift rate bigger

than this threshold develop.

3.1.11 Mode 8

A first batch of test cases showed that when it was in mode 8, during orbit night and with the cameras tracking, the ACS would not switch to mode 2 even if a very big rotation error was introduced. This was due to a mistake in the definition of OMEGA_COAST_THRESHOLD, the limit on the spacecraft tumbling in mode 8. It should have been 1.7453e-3 rad/s (6 arcmin/s), but instead it was set to 1.7453.

In order to do some useful testing on this mode, the definition of OMEGA_COAST_THRESHOLD was corrected, and all the following test cases were run with the proper value.

Switching Logic Testing

The ACS is designed to stay in mode 8 as long as possible, that is as long as the cameras are able to track. When the cameras stop tracking, if it is orbit day, the ACS goes to mode 7 (nominal situation); if it is orbit night, the ACS goes to mode 2 (night safe mode). The other abnormal situations are when there is too much tumbling while the cameras are tracking (more than 6 arcmin/s=1.7453e-3 rad/s drift), in which case mode 2 is called; and when the cameras are still tracking while it is orbit day, but the sun pointing error is too big: mode 7 is called to decrease the sun pointing error. Note that the sun pointing error tolerance drops to 5 deg in mode 8 (the value of the sine for the threshold is 8.71e-2) to comply with the on-station sun pointing requirements.

There is a problem with the delays in this mode: all the test cases ignore the mode 8 delay, the period of time during which no switching should be allowed. This condition seems to be completely missing in this part of the code. Mode 8 delay is not very long (10 seconds), but it can be useful to smooth the transition between mode 7 and mode 8, for example, when the cameras just begin to be able to track and might miss some samples.

test case	orbit_day	tracking	sun pointing error	body_rate	outcome
1	1	1	<threshold	<threshold	mode 8
2	1	1	>threshold	<threshold	mode 7
3	1	0	<threshold	<threshold	mode 7
4	1	0	>threshold	<threshold	mode 7
5	0	1	N/A	<threshold	mode 8
6	0	0	N/A	<threshold	mode 8
7	1	1	<threshold	>threshold	mode 8
8	1	1	>threshold	>threshold	mode 7
9	1	0	<threshold	>threshold	mode 7
10	1	0	>threshold	>threshold	mode 7
11	0	1	N/A	>threshold	mode 2
12	0	0	N/A	>threshold	mode 2
13	1	1	>threshold, <0	<threshold	mode 8

Table 3.8: Mode 8 switching logic test cases, paddles deployed

As shown in table 3.8, all the possible combinations of the parameters orbit_day, tracking, sun pointing error, and tumbling rate were tested.

The goal of the tests 3, 4, 9 and 10 is to verify that in any condition, if it is orbit day and the cameras are not tracking, the ACS will switch to mode 7. In the same manner, if it is orbit day and the sun pointing error is greater than 5 degrees, the ACS will always switch to mode 7, whether the cameras are still tracking or not (tests 2, 4, 8, 10). Finally test 1 and 7 represent the two cases when the ACS is allowed to stay in mode 8 while it is orbit day: the sun pointing error is small , and the tumbling rate is not taken into account by the software in this case (but anyway if the tumbling gets too big, the sun pointing error will grow out of its limits).

For orbit night, in the nominal situation (orbit night with cameras tracking, small drift), the ACS remains in mode 8 (test 5). Also when the body rate is too big, the ACS switches to mode 2 (tests 11 and 12). The other non-nominal situation is orbit night and the cameras are not tracking, as in test 6. The ACS stays in mode 8 and the satellite coasts, issuing no command and waiting for the camera data to come back. This is certainly the best that can be done, as no sun pointing information is available for a daylight mode, and it would be unfortunate to waste a good on-station

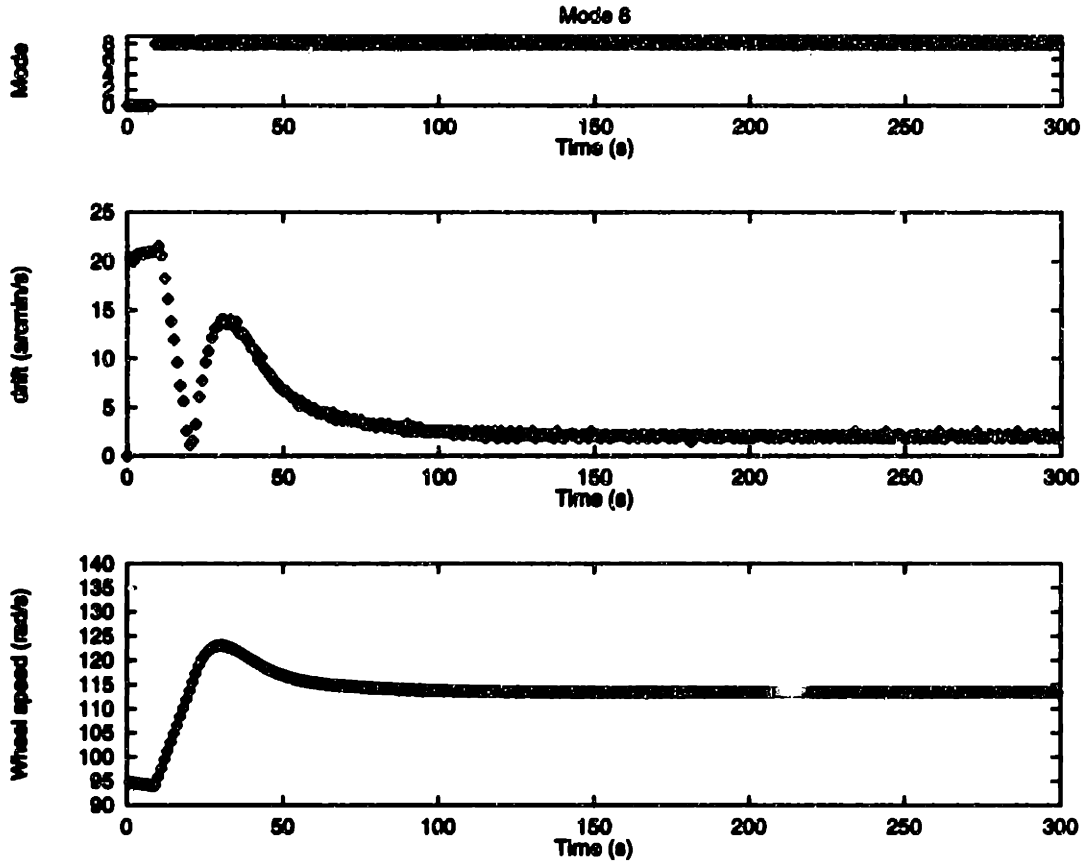

Figure 3-18: Mode 8 functional testing

attitude by going back to mode 2 and spinning the satellite up.

Finally, the absolute values for sin_sun_el and sin_sun_az are also missing in this mode, as shown in test 13.

Parameter Testing

In this mode the attitude is determined by the ACS cameras and the code that processes the raw data from the CCD and sends it to the ACS is part of the science software. So the validity of the drift information used by the night controller cannot be tested here.

Functional Testing

The ACS cameras can track as soon as the drift rate is less than 20 arcmin/s, so mode 8 control laws must be able to handle this same drift rate. To have the ACS

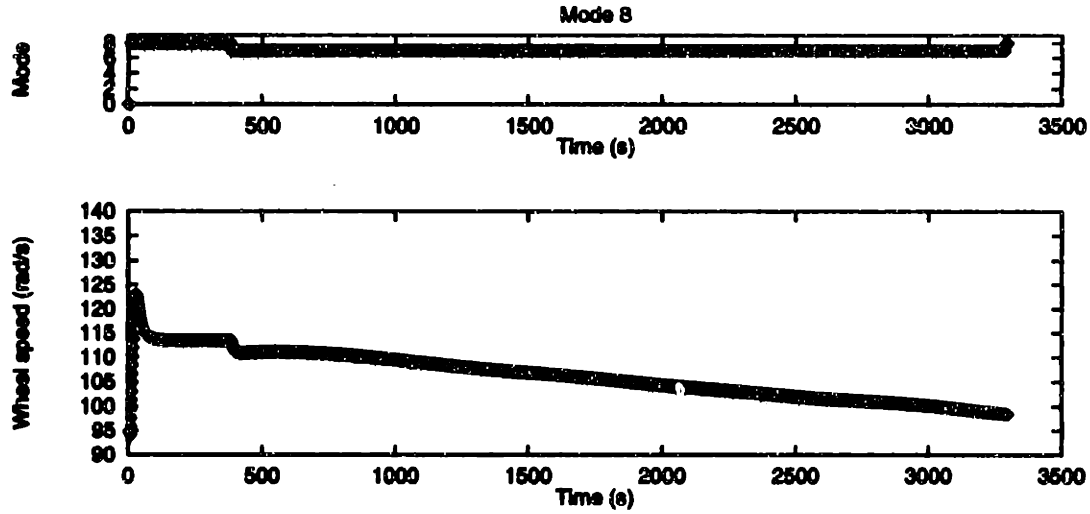

Figure 3-19: Wheel loading (mode 8) and unloading (mode 7)

enter in mode 8 with some controlled rotation speed, mode 0 is first activated for a certain time, and then mode 8 takes over. While it is in mode 0, the wheel is coasting and starts generating a drift about Y. The longer the time spent in mode 0, the bigger the drift.

This procedure was used to generate figure 3-18, for which the drift rate upon entering mode 8 is just over 20 arcmin/s (the configuration is orbit_day=1, tracking, and nominal wheel speed). The plot shows that mode 8 can deal with this rate, and the drift drops quickly. The ACS uses the wheel to correct the attitude, but it is not able to bring the wheel speed all the way back to the regulated speed once the drift has been cancelled. In fact there are some power limitations during orbit night, so mode 8 often loads the wheel. The wheel speed is corrected later, as soon as the ACS switches to mode 7 (cf figure 3-19).

In orbit day, the mode 8 controller can cope with drift rates higher than the one used above (as high as 120 arcmin/s), but past about 50 arcmin/s, the sun pointing hits its threshold before the drift can be sufficiently reduced.

Mode 8 in orbit night has the same behavior as described above, except that the body rate threshold prevents drift rates greater then 6 arcmin/s.

In order to verify that the wheel speed is limited to 300 rad/s, a test where the the satellite enters mode 8 during orbit day with a very high rotation rate about

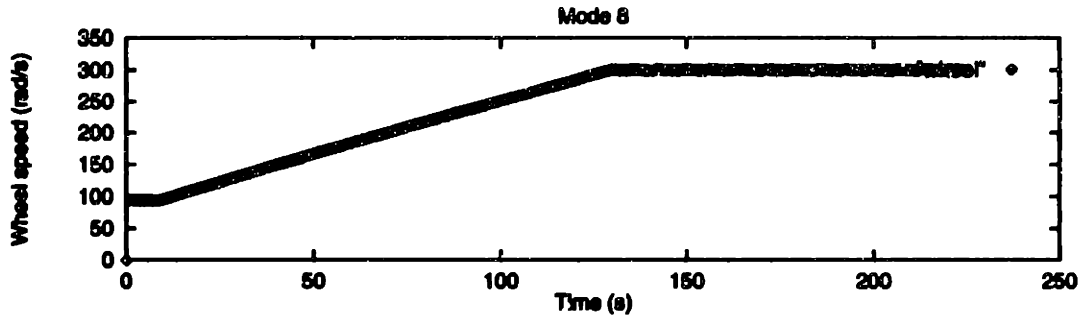

Figure 3-20: Mode 8 functional testing (continued)

Y was run. As the ACS switches between mode 8, 7 and 5 to cancel the rotation, the wheel speed increases a lot until the controller strictly limits the speed at 300 rad/s (figure 3-20). The same test was made with a negative rotation about Y, and it showed that the wheel speed was also limited to 300 rad/s for a negative rotation.

3.1.12 Mode 9

Mode 9 is the ground command mode. When the ACS enters this mode, it just stops issuing commands and waits for a command sent from the ground for 10 seconds. Then it goes to mode 1.

The timing in this mode has been checked in the same manner as in mode 0. Also while the ACS is in mode 9, we verified that no command is issued (no coil torque are applied, and the wheel speed decays under the effect of the drag).

3.1.13 Conclusion of Black Box Testing

This is the end of the black box testing process. All the requirements and the specifications that were available have been checked, and the test cases that were designed can be run systematically to facilitate the coverage estimation.

Note that this first step in the testing process is not necessary per se to achieve MC/DC coverage. An alternate method would be to inspect the source code directly at the beginning of the testing, to list and analyze all the branches, decisions, and function calls, and then deduce the minimal set of tests that would comply with

the coverage criterion. This method is known as *structural testing*, as opposed to *structural coverage*.

To sum up, for structural testing the test case design is based on the structure of the source code (so the coverage is controlled directly) whereas for structural coverage, the test cases design is based on the specifications; the coverage only is subsequently estimated from the source code's structure.

Structural coverage has been used for the testing of the ACS software because even if both procedures achieve the required coverage level, the procedure described in the standard really corresponds to structural coverage.

Even if both procedures achieve the required coverage level, structural coverage has been used for the testing of the ACS software first because it is the form of testing which is described in the DO-178B standard. Also it is the only procedure that guarantees the software meets its requirements. And finally, without any requirement-based testing, the testing process has no link with the reality: it uses only abstract features like conditions, decisions and calls, but the features that are of interest to the software's outside world, like the system state or the commands issued are ignored and that makes the testing less complete.

3.2 White Box Testing

The white box testing process is divided into two different steps: coverage estimation and design of additional test cases.

The goal of coverage estimation is to reveal the portions of the code that have been left unexplored by black box testing and therefore could contain more errors. Since the total numbers of errors in the code is not known, the thoroughness of the testing can only be approximated, and this is done via one of the coverage criterions that have been described earlier (statement coverage criterion, decision criterion, MC/DC criterion, etc.). The testing process is considered to be complete when the coverage criterion is met.

The design of additional test cases is necessary if the black box tests do not satisfy

the coverage criterion. Considering the parts of the source code that are not covered, new test cases are designed to fill the gaps. The testing is over when the set of all the test cases achieve the desired coverage level.

3.2.1 Coverage Evaluation

The DO-178B standard requires MC/DC coverage for level A-critical software. So the coverage for the testing of the HETE ACS software has been evaluated with this criterion.

Three different tools were used to help asses the level of coverage achieved by the test cases:

- Attol Test Coverage, from Attol Testware. This tool is compliant with level A of the DO-178B standard (every point of entry and exit in the program is checked, the MC/DC criterion is applied).

- Cantata from IPL. It is not fully compliant with level A of the standard because the C version of the tool uses the masking version of MC/DC, not the unique cause version.

- GCT (Generic Coverage Tool, Free Software Foundation). This tool only supports decision coverage or the multiple condition coverage, not MC/DC, and it was not designed specifically for the DO-178B standard. However, it is the coverage estimation tool which is used for the regression tests of the HETE software, so it has been also included in the coverage estimation process.

All three tools have been tried first on the examples of chapter one, and the results were consistent with their characteristics. Also during the coverage estimation for the ACS software, the differences between the tools' results corresponded precisely to the differences between the MC/DC, masking MC/DC, and decision coverage criterions.

Each uncovered portion is explained in detail below.

Controller File

The controller file *acs_cont.c* contains the routines that compute the raw commands from the processed sensor data for modes 1, 2, and 3 (detumble controller, spinup controller and reorient controller). Three points were left uncovered by black box testing in this file.

First the `switch` operator which selects the right controller depending on the mode has a default that handles erroneous modes (i.e. modes outside [0-9]). This default case has never been exercised since all the previous simulations used correct modes.

Also for redundancy reasons, there are two magnetometers on the spacecraft. The detumble controller, the spinup controller, and the reorient controller choose to use the magnetic field calculated from magnetometer A or magnetometer B via an `if-else` instruction. Since the black box tests never swapped magnetometers, this decision is not covered.

Finally, when the satellite is not on station (i.e. for mode 0, 1, 2, 3, 4, and 6), the wheel torque is limited by the function *control_wheel()* to a maximum of 0.02 Nms (in absolute value). The black box tests mainly simulate positive rotations about the Y axis for mode 0 to 4, and the negative torque limit was never reached. So the decision `if (*wheel_torque < -rom->wheel_torque_max)` in the *control_wheel()* routine was left uncovered.

On-Station Controller File

Acs_onst.c contains the routines that compute the raw commands from the processed sensor data while the satellite is on station: mode 7-8, and mode 5, (the only differences between mode 5 and mode 7 controllers are the gains).

The on-station controllers also compute the magnetic field from the selected magnetometer, so like in *acs_cont.c*, the code that processes the second magnetometer is never exercised. But for this part of the code, the selection of the magnetic field is not a simple `if-else` instruction, rather a structure like:

```
if (selected_mag == 1)
   use magnetic field from mag B
else if (selected_mag == 0)
   use magnetic field from mag A
else
   send debug message
```

So the case where mag B is selected, but also the case where a wrong magnetometer number is selected must be run by the on_station controller and by the night_controller.

One of the functions of this file, *limit_mag_moment()* is never called. Indeed nothing limits the value of the processed torques in the on-station controllers. The physical limit for the coil torques is given by $V/R \times A_{eff}$ (where V is the bus voltage, R is the coil resistance and A_{eff} is the effective area of the coil). In the simulations, the torques returned by the on-station controllers sometimes go well above this limit. However they are later limited by the function *process_commands()* (cf acs_proc.c), which processes the raw commands and sends them to the actuators. So the magnetic moments of the coils are actually limited, and the function *limit_mag_moment()* is not needed.

The inertia matrix of the satellite is different if the paddles are deployed or not. The controller selects the right inertia matrix with the following decision:

```
if (rom->paddles deployed == 1)
   use I_deployed
else
   use I_stowed
```

The condition `rom->paddles deployed == 1` is a holdover from an old version of the code: initially the state of the paddles was a binary variable (0 for paddles stowed and 1 for paddles deployed). It was then decided that the paddles' deployment should be monitored individually for each paddle, and the state of the paddles was stored in four bits (0x0 for all four paddles stowed, 0xF for all four paddles deployed). In

the version of the code that was tested, the paddles' state is supposed to be coded on four bits, but the two notations are mixed and in several places in the code, as here for the on-station controller, the single bit notation is still used.

This results in a bad selection of the inertia matrix for the on-station controller when the paddles are actually deployed (rom->paddles_deployed $\neq 1$ when the paddles are deployed). The two inertia matrices are not very different, so the bad selection was not noticed in the black box simulations, and the error was revealed only by the white box tests.

This version skew does not exist anymore in the new code, all the paddle variables use the four-bit notation now.

Two portions of the code could not be covered in this file, one in the on-station controller and the other in the night controller. In the on_station_controller, the second decision of the following structure is always true:

```
if (state->mode == ACQ_MODE) {
   L = &(rom->L_acq);
   K = &(rom->K_acq);
}
else if (state->mode == ORBIT_DAY_MODE) {
   L = &(rom->L_day);
   K = &(rom->K_day);
}
else {
   /* FIX: */
   debug_value("!onstation mode = \%d", state->mode);
   exit(1);
}
```

It is not possible to have the `else if` take on a false value without deeply modifying the code, since the on_station_controller is never called from another mode than ACQ_MODE or ORBIT_DAY_MODE. So the error message and the abnormal exit

are never executed, and the logic can be replaced by a simple `if-else` in the definitive code.

Similarly, in the night controller, the first condition of the decision

```
if ( (state->previous_mode != ACQ_MODE) &&
     (state->previous_mode != ORBIT_DAY_MODE) )
```

can never take on a false value because there is no path between mode 8, the only place where the night_controller is called, and the acquisition mode (mode 5). The purpose of this decision is to decide whether the state for the Kalman filter must be initialized (it must be initialized if the ACS comes from a mode which is not a on-station mode). So the first decision, even if it is in principle useless, makes sense.

Sensor and Command Processing File

The routines of the file *acs_cont.c* process the raw data coming from the sensors, and translates the controllers' outputs into commands for the actuators. The software checks for errors in the sensor data, but this capability was not exercised during black box testing, so part of the code in this file was not covered

The software keeps track of the errors occurring on the AUX bus devices via two variables, `mss_css_err` and `fss_whl_mag_err`. These variables can be masked to indicate the possible errors on mag A, on mag B, on the different sun sensors, or on the wheel tachometer. If an error occurs, the data of the corresponding device is not updated, and the old values are used. No AUX errors were simulated during black box testing, so this needs to be done to go through the branches of the codes that handle this situation.

The ACS software also watches for time roll overs, i.e. when the present time is smaller than the time of the previous sample. Time roll overs can happen when a time register reaches its maximal value or after a reboot of the processor (this should never happen on HETE-2 because a 64-bit digital watch permanently keeps track of the time). They cause a problem in the ACS software, particularly for the calculations of the time derivatives and the wheel speed which use the time difference between

two samples. If a time roll over happens, the software is supposed to not compute the magnetic field time derivative or the tachometer wheel speed, and use the old values.

If the sensor data happens to be too old by the time the commands to be sent to the actuators are computed, the software sets all the commands to zero. This insures that no out-of-date commands are executed, after a processor lock out, for example.

These time checks of the ACS software were never exercised during black box testing, so they require additional tests.

Other sanity checks in *acs_proc.c* include a verification of the bus voltage before the computation of the commands for the torque coils drivers. If the bus voltage reading is too low (less than 1 volt) or too high (more than 100 volts), then the reading is certainly erroneous and the nominal bus voltage (28 V) is used instead. A simulation of bad bus voltage reading must be included in the additional test cases.

As explained before, the number of coil pulses computed by the controllers to generate magnetic torques are limited in *acs_proc.c*. At least, this capability is coded, but it was never tried during the first part of the testing, it is another test which must be added.

On the other hand, for the momentum wheel torques, each controller has some instructions to limit the value of the wheel torque it outputs, and in *acs_proc.c*, there is again some logic to limit the value of the wheel torque required by the controllers. This a dead part of code, it will never be used and it can not be covered.

A similar situation is encountered for the selection of the correct tachometer to compute the wheel speed. If the wheel spins too fast, the wheel speed is calculated from the high range tachometer:

```
if( (sensor->processed_sensor.wheel_rate_body >= rom->period_tach_threshold) ||
(sensor->processed_sensor.wheel_rate_body <= -rom->period_tach_threshold)) {
   compute the wheel rate accordingly
}
```

The second condition is never true, and the reason is that `wheel_rate_body` is always a positive number at this point in the code: in the previous statements, it is assigned

the value of `wheel_rate_body_freq_tach` which is the absolute value of the rotation rate. The sign of the rotation is determined only in the following statements, and at this time only the wheel_rate_body variable can become negative. The second decision is therefore useless.

Finally, the magnetometer selection logic must be tested more carefully. One of the magnetometers is selected in the ROM and is to be used preferentially. When this magnetometer has no error, it is selected by the software, and all the calculations in the following acs functions are based on its measurements. If an error occurs on the rom-selected magnetometer, the other magnetometer is used.

So in order to achieve a complete coverage, we must try both magnetometers as ROM-selected magnetometers, and for each case simulate no error, an error on mag A, an error on mag B, and an error on both magnetometers (in this last case the software is supposed to not reprocess the field and use the old value). Also a test where no valid magnetometer is selected by the software must be run because of the magnetometer selection instructions described above in *acs_onst.c*.

Mode Selection File

The file *acs_mode.c* handles the mode selection logic of the acs software.

The coverage estimation shows that all the switching logic of the ACS has been verified by the black box testing cases except the default case for the mode selection (corresponding to a non-valid mode) and the situation where the satellite enters mode 6 with the paddles already deployed.

The problem with the `paddles_deployed` variable appears again in this file: in one of the decisions, the condition used to test the deployment of the paddles is `paddles_deployed==1` instead of `paddles_deployed==0x000F`. Because of this error, one branch of the switching diagram, which goes from mode 2 to mode 8 when the cameras are tracking and the paddles are deployed, was never taken. This branch was not included in the specifications, so the problem was not noticed during black box testing.

Driver File

Acs_drvr.c contains the driver for the ACS software. The driver keeps calling sequentially the routines that process the sensor data, select the mode, compute the commands and process the commands. This code was completely covered by black box testing.

Initialization Files

Acs_init.c and *acs_irom.c* initialize the ACS variables when the program starts. This part of the code was also completely covered by black box testing except for a decision in acs_init.c which does not uses the `paddles_deployed` variable correctly (same problem as in *acs_mode.c*, *acs_onst.c* and *acs_proc.c*).

Mathematical Functions File

Mtx_math.c gathers all the custom mathematical functions used in the ACS software. Only twelve functions of this file, out of thirty two are actually called (mxm, make_unit, vec_add4, cross, vec_mag, scalxvect, vecsub2, vecsub4, det_prod, make_mtx, make_vector, and mxv). The following test cases are meant to complete the black box testing and achieve a complete coverage.

3.2.2 Additional Test Cases

The coverage evaluation has revealed some parts of the code which were not fully covered (according to the MC/DC criterion) by black box testing. The additional test cases are meant to fill these gaps.

Illegal Mode

In order to cover the default case for the mode selection switches, a test that calls invalid modes was designed. It proved that the software detects illegal modes and goes to mode 0 instead, to restart the acquisition process.

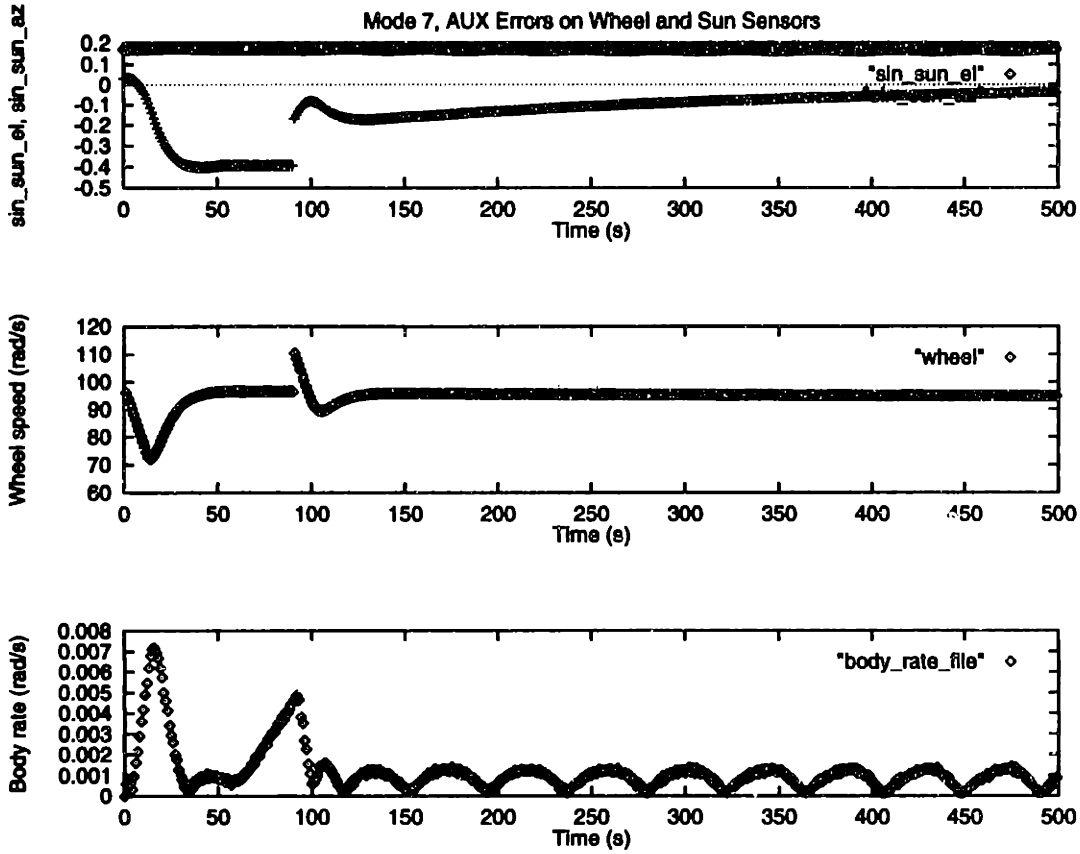

Figure 3-21: Simulation of AUX errors

Mode 6

To complete the black box testing of the switching diagram for mode 6, a test was designed where the satellite enters mode 6 with the paddles already deployed. The simulation shows that in this case, the ACS switches immediately back to mode 5 and the on-station cycle continues normally.

AUX Errors

In this paragraph we verify how the ACS software handles the AUX errors that can corrupt the data coming from the different sun sensors and from the wheel tachometer.

In principle when an error occurs on an AUX device, the ACS software does not do the calculation to update the informations linked to this AUX device, and it uses the old data.

This is the behavior that have been seen in all the modes . For example figure 3-21 shows a situation where the satellite is on-station during orbit day (mode 7), and errors in the AUX data coming from the sun sensors and from the wheel are detected between t=60s and t=110s. The plots show that the sun pointing parameters sin_sun_el and sin_sun_az are not updated anymore, and neither is the wheel speed. The satellite begins to drift, since the controller does not have good data to compute the torques. As soon as the AUX bus recovers, the controller gets fresh measurements, it can compute accurate commands and the satellite is stabilized.

In the above example the Fine Sun Sensor is selected, since the satellite is on station and it is orbit day. The same behavior is observed when the Coarse Sun Sensors or the Medium Sun Sensors are used. When the AUX data is missing only for a few samples, the drop out is not even noticed in the simulation. On the contrary, when the AUX data is missing for a very long time, the satellite begins to drift a lot, and the ACS has to back up several modes to acquire the attitude again.

Note that when an AUX error is detected on only one of the sun sensors (i.e. one of the twelve CSS, the MSS or the FSS), the software does not try to use the other sensors to compute the sun pointing parameters, it discards all the sun sensor information. This is optimized for short AUX bus black outs, for which it is not worthwhile to lose some time to go through a complex selection logic to pick up the good information since the data will be available again a few samples later. This is not adapted to the case of a sun sensor hardware failure, for example, which produces a permanent error. In this case, in order to have the software operate correctly, the mask that corresponds to the failed sensor must be set to zero manually so that the error is ignored, and the code must be modified so that it does not use the failed sensor.

Magnetometer Selection Logic

There are three levels in the magnetometer selection logic. First, the best magnetometer (i.e. the one which is the least influenced by the magnetic fields created by the satellite) is selected in the ROM; it will be used preferentially. The second level is

test	mode	ROM-selected mag	magA	magB	software-selected mag
1	1	A	AUX error	OK	B
2	2	A	AUX error	OK	B
3	3	A	AUX error	OK	B
4	7	A	AUX error	OK	B
5	8	A	AUX error	OK	B
6	1	A	AUX error	AUX error	B
7	1	B	OK	OK	B
8	1	B	AUX error	OK	B
9	1	B	OK	AUX error	A
10	1	none	OK	OK	B
11	7	A	OK	OK	none
12	8	A	OK	OK	none

Table 3.9: Magnetometer selection logic tests

a software decision: based on the ROM-selected magnetometer and on the detected AUX errors, for each cycle the software chooses the magnetometer which is to be used by the controllers to compute the control torques. Finally the detection of the magnetometer AUX errors is directly used in *acs_proc.c* to determine whether the measured field of a particular magnetometer can be updated or not (if AUX errors are detected, the magnetic field is not updated, and the variables are set to zero). The tests listed in table 3.9 were run to verify this logic.

Tests 1 through 5 were designed to verify that when an AUX error occurs on the ROM-selected magnetometer, the other magnetometer is used by all the controllers (detumble controller in mode1, spinup controller in mode2, reorient controller in mode 3, on_station controller in mode 5 and 7, and night controller in mode 8). The simulations show that the software actually switches to magnetometer B, and the data form this magnetometer is correctly processed: the exact same plots as in figures 3-2, 3-5 were obtained for all the controllers.

In test 6 an AUX error was simulated on both magnetometers. In this case the magnetic field variables are set to zero for both magnetometers in *acs_proc.c*. This is shown in figure 3-22 for the XZ momentum error computation in mode 1. Since the magnetometer data is low-pass filtered, when the AUX errors hit both magnetometers

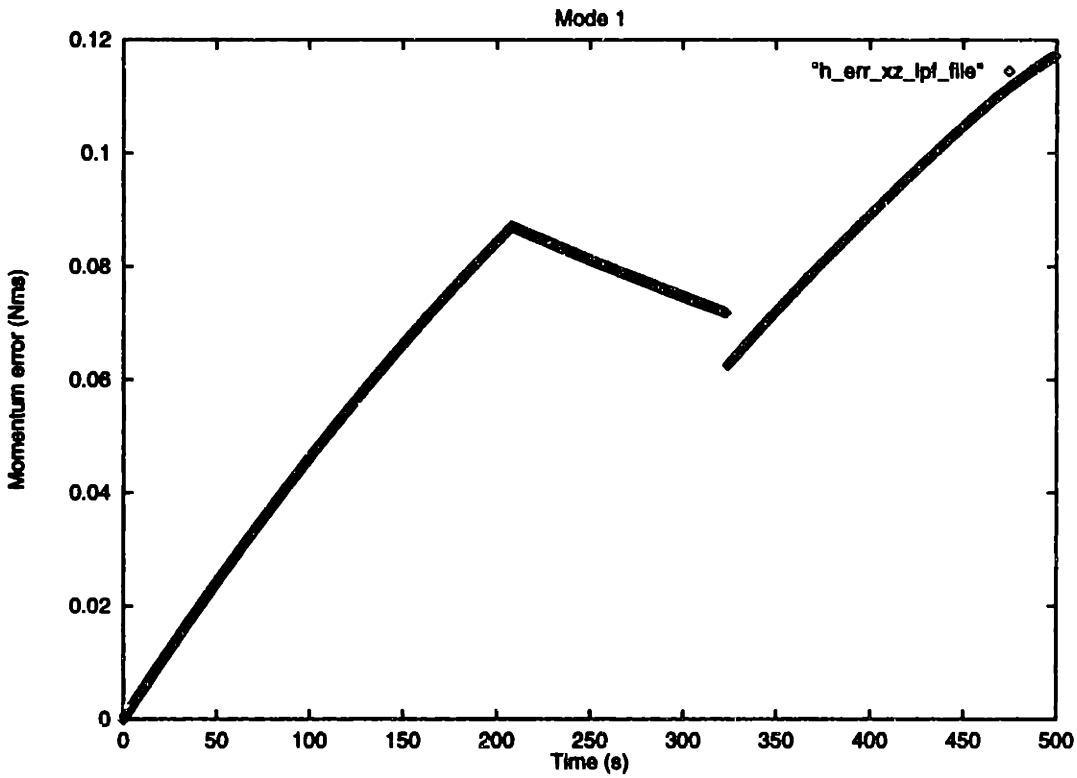

Figure 3-22: AUX errors on both magnetometers

between t=220s and t=320s the momentum error does not go instantaneously to zero, rather it starts to decrease.

In tests 7, 8 and 9, magnetometer B is select in the ROM instead of A. As expected, the software chooses mag B for the controllers when no AUX errors occur on this magnetometer (it goes back to mag A otherwise), and it makes no difference in the simulations. In test 10 a non-valid magnetometer value is selected in the ROM. In this case mag B is used by default as the ROM-selected magnetometer, and the software performs as usual.

Finally to completly cover the instruction in the on-station controller and in the night controller (cf paragraph 3.2.1), we forced the software to select a wrong magnetometer value in test 12.

```
if (selected_mag == 1)
   mag_field = &(sensor->processed_sensor.magB_field_body);
else if (selected_mag == 0)
   mag_field = &(sensor->processed_sensor.magA_field_body);
else
   send debug message
```

This makes the program crash. If the `if` and the `else if` are both false, the pointer `mag_field` is not referenced, and when it is used later in the program, it causes a segmentation fault.

In all the previous tests, the software never failed to assign a good magnetometer to the state variable, so the default case (last `else` of the `if-else if-else` structure) can be suppressed safely. The new code would be a simple `if-else` structure. With this new logic, in the case of a bad magnetometer selection in the state variable, the software would by default assign the magnetic field of mag A to the variable `mag_field`, so that the pointer is initialized and will not cause a problem later.

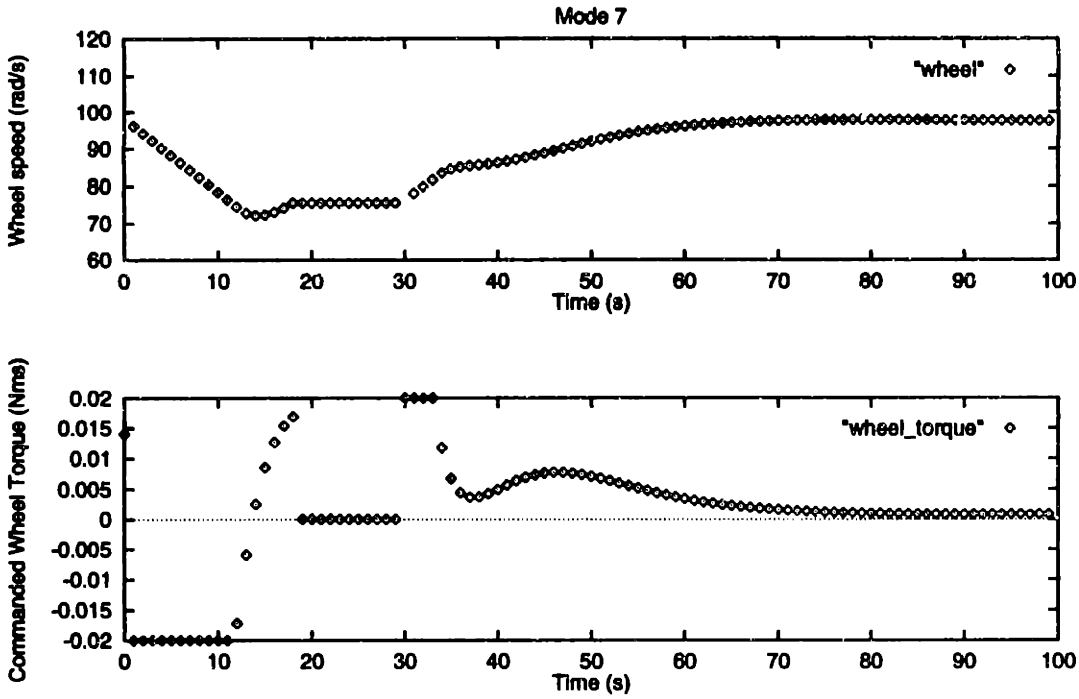

Figure 3-23: Effect of time rollover and old sensor data

Time Roll Overs and and Old Sensor Data

Some tests were designed to take control of the ACS software time and create situations where a time roll over occurs or situations where the time elapsed between the acquisition of the sensor data and the computation of the commands to be sent to the actuators is very long.

The first plot of figure 3-23 is an example of sensor data when multiple time roll overs are created (between t=19s and t=29s). As expected, the sensor data is not updated and the software continues on with the previous values. The wheel sensor data is shown on the plot, but the same results are obtained for the sun pointing data and for the magnetometer data.

For the second set of tests the time was fast forwarded between the sensor data acquisition and the command computation, so that the time elapsed between these two events exceeds the maximum delay allowed by the software. All the commands (coil torques and wheel torque) are set to zero, as shown in the second plot for the

case of the wheel torque coil (timing problem simulated between t=19s and t=29s).

Torque Coil Controller

Several test cases have been designed to fully cover the part of the code that generates the torque coil commands.

The first set of tests explores the behavior of the routine that process the torque command when the bus voltage varies. As deduced from the source code, the simulation shows that if the bus voltage reading gets above 100V or under 1V, the computation of the torque coil driver pulses is not made using the sensed bus voltage, but using the nominal value of the bus voltage (28V).

For the second set of tests, some "torque coil intensive" simulations were done in order to hit the torque coil pulses threshold, as this was never achieved during black box testing. As expected, in every mode, the counts commanded to the torque coil driver were in the [-2046 to 2046] range.

Wheel Torque Limit

Finally we must check that the wheel torque threshold is respected while the satellite is in mode 1, 2, 3, 4 or 6. The wheel torque is limited by a special controller when the satellite is not in an on-station mode, and the tests proved that this controller was working as specified, keeping the commanded wheel torques between -0.02 and 0.02 Nms.

3.2.3 Conclusion of White Box Testing

In order to be able to fully test the code, the decisions using the bad convention for the **paddles_deployed** variable were fixed. Also the three conditions in *acs_proc.c* and *acs_onst.c* that could not be toggled (at least without deeply modifying the structure of the code) were removed. As explained above, these conditions were useless because their value was constant in this part of the code.

Then all the test cases were rerun. The simulations results were what we expected,

with only one difference: fixing the `paddles_deployed` variable enabled a new path in the switching diagram; when the paddles are deployed and the cameras are tracking, the ACS can switch from mode 2 (spinup mode) to mode 8 (orbit night mode). So now when the spacecraft is in mode 8, if the camera tracking data becomes unavailable for a short moment, causing the ACS to switch back to mode 2, the spacecraft has a chance to come back directly to mode 8 if it is not already spinning too fast for the camera to be able to lock.

After these modifications, the code structure is 100% covered according to the MC/DC criterion, and the testing process is over.

3.3 Conclusion

The example of this testing process using the DO-178B procedure allows to address some issues related to the MC/DC controversy, at least for the case of the HETE-2 ACS software.

Relation between MC/DC criterion and software safety The main question which must be answered is whether MC/DC coverage improves the safety of the software. In other words, do the additional tests required by the MC/DC criterion make sense, or do they just consist of playing with some variables to artificially toggle some conditions, a process which is by no mean related to safety or even to the software practical environment?

In the case of the HETE-2 ACS software, all the additional tests required to satisfy the MC/DC criterion were directly linked to an important feature of the software. More precisely, the need for additional tests corresponded to four kinds of caveats of the preceding testing process:

- Something was forgotten during black box testing. Examples of this for the HETE-2 ACS software include the torque coil driver pulses and the wheel torque limits, and the testing of the case where the satellite enters mode 6 with the solar paddles already deployed.

- The software features a complex logic mechanism which requires in-depth understanding and a precise, customized testing. This was the case of the magnetometer selection logic for the ACS software.

- Some feature of the software was not included in the specifications and therefore could not give rise to a test case in a black box testing context. This situation happened for the AUX error checks and the time checks, and for the bus voltage verification before the coil torque computations. The fact that the white box testing process serves as a verification of the completeness of the specifications has been very useful.

- Finally the white box tests can lead directly to an error which effects are too small to be detected by black box testing. In the case of the `paddles_deployed` variable, we found that a bad value was assigned to this variable because the conditions in which it was involved could not be toggled. The consequences of this error at a higher level had gone undetected before., because the difference in output was so small.

So the justification for each white box test has its root at a high level. The tests required by the MC/DC criterion were not relevant only to the structure of the code, they made us explore some important points of the code, and in that sense they contributed to insure the safety of the software.

Complexity of the MC/DC criterion In this paragraph, we try to determine whether the coverage required by the MC/DC criterion is too excessive. That is, if we recognize that white box testing is a good method to make a software safe, would it be possible to use a less constraining coverage criterion and still insure the same level of safety?

From the experience gained form the HETE-2 ACS software, the answer to this question is clearly no. For example, in `acs_proc.c`, the problem with the expression

```
if(  (sensor->processed_sensor.wheel_rate_body >= rom->period_tach_threshold)
  || (sensor->processed_sensor.wheel_rate_body <= -rom->period_tach_threshold))
```

was detected only because the second condition was preventing the decision from passing the MC/DC criterion. The tests could exercise the cases TX=T and FF=F, so the decision was passing the decision coverage criterion and also the condition/decision coverage criterion. But because the second decision was not appropriate in this decision, the last test case FT=F could not be done, and this was detected by MC/DC only.

The problem we refer to above was not a crucial problem, it would not have caused any damage, had it remained undetected. However, the decisions that spotted the error concerning the `paddles_deployed` variable were the same kind of boolean function, except that the AND operator was replaced by an OR. The fact that this important problem could also be detected by the decision coverage criterion relies only on this little difference.

Difficulty of achieving MC/DC Let's first consider the time required for the white box testing with respect to the total testing time. In the case of the HETE-2 ACS software, the coverage estimation and the design of the additional test cases represented about 40% of the total testing time, the rest of the time being devoted to black box testing. Note that powerful tools were used to help determine the coverage, so the coverage estimation process was quite fast and easily repeatable. Another factor, this time inherent to the nature of the code itself, facilitated white box testing: part of the black box testing activity consisted in checking the switching diagram of the ACS (figure 2-6), i.e. verifying that the different branches of the diagram were connected under the correct conditions. These switching specifications are in fact very close to the structure of the source code itself, so in fact part of black box testing was equivalent to testing the source code structure. Therefore the number of the subsequent white box tests was certainly reduced, and the proportion white box testing time / black box testing time biased in favor of white box testing. White box testing has been useful to find errors in the code but it indeed represents a time-consuming step in the complete testing process.

Secondly it is interesting to compare the difficulty of achieving MC/DC coverage

versus achieving a simpler form of coverage as decision coverage. In fact in the case of HETE-2 ACS software, MC/DC was not much more difficult to achieve than decision coverage. This is due to the programming style of this code: only 11% of the decisions were composed of boolean functions, all the other decisions were single-condition decisions. In this later case decision coverage and MC/DC are the same.

Hence the fact that the decisions were kept simple contributed a lot to making this software well adapted to MC/DC testing. But keeping the decisions simple in the source code also has its limits, because in general simple decisions induce more complex logical structure. The principle is to replace multiple conditions linked by some boolean operators (AND, OR, etc.) with nested `if-else if` instructions. This kind of logic is very prone to errors, as shown for example in the switching logic testing of mode 3 (section 3.1.6). So even if this style of programming facilitates MC/DC testing, a good balance between the complexity of the boolean conditions and the complexity of the logic structure must be preserved to insure a maximum safety.

Finally the testing process of the HETE-2 ACS software could be implemented as it was described in the DO-178B standard. The required testing method —requirement-based testing, coverage estimation, additional test cases did not arise unexpected problems and the proportions of the different phases were what we expected. In our particular case, the DO-178B testing strategy was well adapted, and it provided us with the guidelines to conduct an efficient testing.

Bibliography

[1] RTCA. *Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[2] John J. Chilenski and Steven P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 1994.

[3] Nancy G. Leveson. *Safeware: System Safety and Computers.* Addison-Wesley, 1995.

[4] Robert Jasper, Mike Brennan, Keith Wiliamson, Bill Currier, and David Zimmerman. Test data generation and feasible path analysis. *SIGSOFT Software Engineering Notes*, 1994.

[5] IPL software testing papers. *IPL web site, http://www.iplbath.com/p80.htm*, 1996.

[6] John J. Chilenski and L.A. Richey. Definition for a masking form of modified condition decision coverage. *Boeing web site, http://www.boeing.com/nosearch/mcdc/*, 1997.

[7] Elaine Weyuker, Tarak Goradia, and Ashutosh Singh. Automatically generating test data from a boolean specification.

[8] A. Simonetti, A. Coupier, and C. Secher. Experience gained from recent avionics software certification. *Bulletin technique du Bureau Veritas*, 1992.

[9] Robert Foch. Software quality characteristics and operating procedure for their assesment. *Bulletin technique du Bureau Veritas*, 1992.

[10] Frederique Copigneaux. Principal approaches to software quality measurement. *Bulletin technique du Bureau Veritas*, 1992.

[11] Glenford J. Myers. *The Art of Software Testing.* John Wiley, 1946.

[12] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice Hall, 1988.

[13] Z. Kohavi. *Switching Theory and Finite Automata Theory.* McGraw-Hill Book Company, 1978.

[14] HETE web site. http://space.mit.edu/HETE/, 1996.

[15] MIT Center for Space Research. *Hete Documentation: ACS, Electronics Box, CDR.*

[16] Attol Testware (http://www.attol-testware.com/coverage.htm). *Attol Coverage Documentation.*

[17] IPL (http://www.iplbath.com/p13.htm). *Cantata for C Documentation.*

[18] UIUC Department of Computer Science, ftp://cs.uiuc.edu/pub/testing/gct.file. *GCT documentation.*