TESTING OF ON-BOARD MISSION-CRITICAL SOFTWARE

A. Santarelli, F. Di Pace

DATAMAT Ingegneria Dei Sistemi SpA

Via S. Martini 126 - 00143 Roma, Italy

Abstract

The paper describes the DATAMAT experience gathered in testing on-board critical applications and focuses on the approach adopted to manage the testing activities.

The experience led to the definition of a formal approach to testing, whose characteristics were derived not only from the theoretical approach, but mainly from the know-how drawn from the testing activities carried out within the Mission Critical Programs in which DATAMAT has been involved since late seventies.

The approach, aimed to the formalization of the test procedures, to their automation and to the standardization of the test documentation, and the tools developed to implement it are described. The benefits obtainable using this approach are discussed.

Then a description of the experience drawn in implementing the above approach is given, focusing on the testing of the EH101 (Anti-Submarine Warfare Anglo-Italian Helicopter) Mission Software/Mission System Integration and on the testing of the Hermes (European Spaceplane) On-Board Mission Software Mock-Up.

Introduction

One of the major problems related to the Quality Assurance in the designing and implementing critical systems is the ensuring of the quality during all the phases of the life cycle of the systems at the maximum level, keeping the overall costs, and specifically the ones related to the testing activities, within the scheduled limits.

The efforts for the verification and validation carried out during the Mission Critical Programs in which DATAMAT has been involved since late seventies (like the Italian Navy On-Board Command and Control Systems for Hydrofoils, Minehunters and Submarines; like the EH101 Mission Software/Mission System Integration and the Hermes On-Board Mission Software Mock-Up), have led to the definition of a set of strategies and methodologies that meet the above goals.

They are based on the concepts of the Test Formalization and Test Automation, whose aim is to migrate the test engineers' efforts towards the definitions of the test procedures while relying on the machine for the test execution and documentation.

The Approach to the V&V Testing

During the development of large systems, it is necessary to verify the compliance of the intermediate products with the functionalities allocated to them during the architectural decomposition (water-fall model (1.)), and, at the end of the development effort, to verify the compliance of the whole system versus the user requirements (the validation). The above V&V activities (6.) are not limited to the described phases, but they also continue in the maintenance one, when updatings and upgradings of the systems require to re-validate the new deliveries.

Traditional V&V techniques for the testing activities include the development of test drivers and stubs (mainly at the unit level testing), and environment simulators (at the system level testing).

Drivers and stubs are generally developed by designers to verify their products in the early coding phases, without following standards and rendering difficult the control on the test activities. Furthermore the requirement coverage of the tests performed using drivers and stubs is little evident, unless the designer provides the necessary documentation.

Environment simulators are vice versa used in the integration and system testing, to generate the stimuli of the environment around the system and to get the system responses.

As the environment simulation can involve the management of complex scenarios, or the amount of different data exchanged with the system is considerable, simulators can be large software products, and therefore require a separate development and maintenance life cycle.

The experience in the usage of environment simulators shows that they take the advantage of better addressing the dynamic and intuitive testing, but have the following drawbacks:

- The related problem of testing the environment simulators - "who tests the tester?" -.
- The difficulty in locating bugs when system under test and environment simulators run together - "the bug is in the system or in the simulator?"-, and consequent disagreements between the designers and test engineers.
- The complexity in tracing the requirements, unless an extensive additional documentation is produced.
- The difficulty in tracking the results of the tests, often assigned to the visual observation of the test personnel during the test executions.
- The more complex is the simulator, the more difficult is to obtain predefined and deterministic behaviors (required for validation tests).

The above considerations along with the necessity to generate a test documentation that spans all the testing phases according to predefined standards (often required by the customer), leaded to the definition of a more formalized approach towards the testing activities, in which the coverage of the requirements and, more in general, the test procedures understanding and transparency become the central aims.

The Formal Approach to Testing

The adopted formal approach to testing is based on the following main ideas:

1. Data Formalization, that means the unambiguous specification of the interfaces.
2. Test Formalization, meaning the unambiguous specification of the test procedures.
3. Linkage towards the Requirements, meaning that the requirements must be traced from the test procedures.
4. Test Report Standardization, that means the production of the test reports, according to predefined and controlled standards, in all the testing phases.

The formalization of the data exchanged through the relevant interfaces of the system is not related to the testing activities only, but it is a general requirement for the subsystem integration. We highlight here this concept because the interface specifications, as direct product of the system specification, are the basic inputs for the test procedures, that are in charge of the stimulation/verification of the system under test through the interfaces themselves.

The formalization of the test procedures aims to the production of documents increasing the comprehensive understanding of them by the quality assurance personnel and by the customer.

Moreover the clearness of the test procedures eases the activities related to the regression testing, assuring the repeatability of the validation tests each time a new delivery of the system is released (a typical situation in the life cycle of large systems).

The linkage of the test procedures towards the requirements is the straightforward way to verify which requirement is covered by which test procedure (needed for the requirements' matrix filling). This linkage also allows to know, for each requirement, if it has been successfully tested or not.

The choice of focusing on the formalization as guideline in the testing activities carried to the definition of a set of formal languages for the test description. In such a way it was possible to overcome the intrinsic ambiguities of the natural languages, assuring the comprehension of the test readers, and to obtain the automation of the test procedures. The following three main concepts summarize the efforts in testing formalization:

1. The Data Language, allowing the unambiguous interface specification.
2. The Test Language, providing a set of simple statements for test procedure definitions and requirement reference.
3. The Automatic Report Generation, providing an automatic and comprehensive documentation of test execution results.

Data Language

The aim of the Data Language is the formalization of the interfaces relevant for testing. This means that both the in-

terfaces of the whole system towards the environment and the internal interfaces separating its structural components can be described in the same formal way.

Generally speaking, we have that the system under test (or its components) communicates with the environment via one or more, logical or physical, communication channels(Fig. 1.). The interface description focuses on the general characteristics of these channels and on the data exchanged through them.

General characteristics of the communication channels are type and address (the address is necessary to identify channels of the same type). Supported channel types by the Data Language are those ones opened towards the 1553 bus, the generic SW link among tasks (based on the IPC facilities of the operating platform), etc.

Moreover, the system exchanges data through the channels packed into messages. For the messages it is possible to specify their addresses (to identify each message from each other), and their data structures.

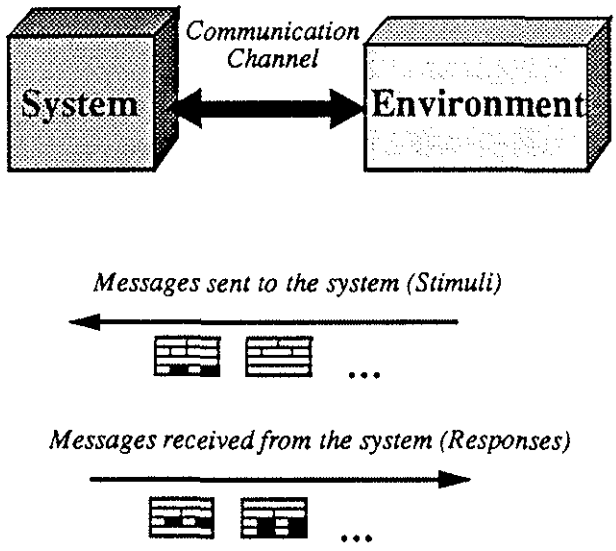

Fig. 1. System Interface with the Environment

The approach to the data description provided by the Data Language starts from the description of the physical layout of the datum (till the atomic quantum - the bit), then allows logic views (the "engineered" data). This approach guarantees an unambiguous data representation, but, on the other hand, includes also the constructs to make the data description comprehensive to different users (like the QA personnel).

The basic data types supported for the description of the physical layout (the "raw" data) range from the integer types (bits, bytes, words, etc.) to reals (single and double precision) and to texts. For each basic type, the memory size and the set of allowed operations are fixed: it is also possible to define ranges for them, such as intervals or set of allowed values. Basic data can be aggregated using commonly known constructs, like structures, unions and arrays. In such a way it is possible to describe messages as complex as needed.

Often data are stored in the messages in order to match some constraints (mainly the memory occupancy) - the "raw" data - while their usage needs a conversion to a high level representation - the "engineered" data -. The logic views of the physical layout are supplied by means of the declarations of engineered data (belonging to the set of basic data types, of course) specifying the raw ones from which they derive, and the conversion procedures applied to compute them (Fig. 2.). A set of predefined conversion criteria is supplied, and the user can add own conversion procedures.

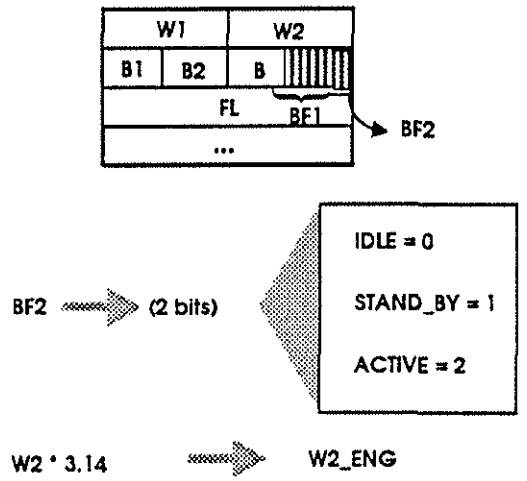

Fig. 2. The Physical-Logic Data Layout

Hence the Data Language allows to create a centralized Test Data Dictionary, containing all the information required for the interface descriptions. This information is used by the test procedures to handle operations over the interface data (like sending and check operations).

Test Language

The Test Language is a formal language, independent from the common programming languages, that supports

the test engineer in the execution and documentation of the tests.

The language, containing a set of general statements (like data handling and control flow ones), is mainly characterized by a set of powerful statements specific for the description of sequences of actions needed to stimulate the system (or its components), and verify the related responses. It also includes a statement that allows the requirement references.

As the involved data are those exchanged via the interfaces of the system under test, the Test Language statements are fully integrated with the descriptions contained in the Test Data Dictionary.

The Test Language provides a test session organization based on a hierarchical structure. At the top level there are the Test Procedures: each Test Procedure can be related to a main component of the system under test. A Test Procedure is divided into a set of Test Sequences that focus on specific requirements (derived by the related functionalities). Then each Test Sequence is instantiated to a set of elementary Test Cases (minimal number of test steps addressing specific test conditions to be verified).

The verification of the system responses is managed with two classes of test statements.

The first class includes statements that express reception of data from the system under test. These statements, along with the statements involving data sending, allow the test engineer to describe the expected behavior of a single component (unit test) or of the whole system (validation test), according to a functional (Black Box) testing methodology (2.).

The second class includes statements that capture data exchanged among components under test. These statements allow to describe the expected data traffic among components under test (integration test).

Therefore the language covers all the testing phases, from unit test to integration test, and, finally, to the validation test of the whole system, providing a unique approach (Fig. 3.).

With regard to the usage of the first class of statements, it is clear that the test engineer can carry on it up to minimal components, increasing the inspection level to something similar to the White-Box testing.

The set of statements, specific for the testing activities, include:

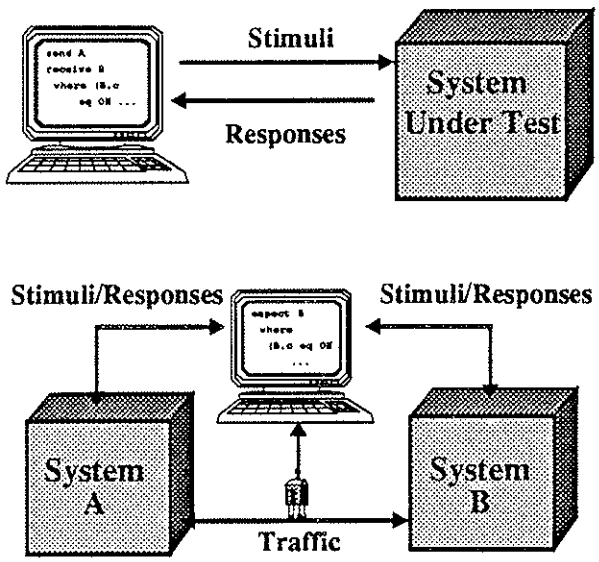

Fig. 3. Unit/System Test and Integration Test

- Declaration of Test Procedures/Sequences/Cases.
- Declaration of assertions. Assertions are predicates evaluated at the test execution time.
- Transmission and reception of data towards physical/logical devices, using two different classes of statements:
  - Data sending/receiving statements (to/from the component under test).
  - Data traffic capturing.
- Logging of data. The user can log own informational messages that will increase the readability of the test results (see the Automatic Report Generation).
- Monitoring of data exchanging.
- Definition of surveillance conditions and related actions.
- Automatic generation of test input data.
- Dialogues with the test personnel to correctly step through the test procedures (semi-automatic test: the user's responses affect the test evolution).
- Requirement references.

The test procedures written by the user are compiled by a tool developed ad hoc, the Test Compiler, that generates an object version of the source module containing low level instructions plus additional information (like the compilation time, etc.). Moreover, the Test Compiler generates ASCII files containing, for each item of the test hierarchy, the list of referenced requirements: these files are imported

by a tool producing the cross-reference between the requirements (extracted by the user data bases) and the tests.

Another tool, the Test Controller, is in charge of the actual test execution: when it is invoked to execute a supplied test procedure, it performs the settings for the dynamic evolution and the checks necessary to verify that the functional dependencies among modules (generated by the test procedure calls) are respected.

The Test Controller supplies a connection-oriented mechanism for the data exchanging with the system under test. Actually, the communications are established by a different tool, the Communication Server, that is structured in two main layers: the low level one is dependent of the type of communication channels to be managed (e.g. the 1553 bus), whilst the upper level one is the general back-end towards the Test Controller. The Communication Server can be hosted on a machine connected by means of the network with that one where the Test Controller runs: therefore it is possible to perform a distributed testing.

Once the communications with the system under test are established, the Test Controller "executes" the instructions contained in the loaded objects, sending and receiving data and performing all the specified checks.

An additional tool, the Monitor Tool, has been developed to capture the messages running among the components under test during the integration testing activities, and to record the whole messages traffic (when requested for off-line analysis). Also this tool is handled at run-time by the Test Controller, while the files containing the recorded traffic are used by a set of utilities that allow to analyze the data evolution by means of queries and statistics.

Automatic Report Generation

During the execution of the tests, the Test Controller provides a double recording of the execution into ASCII files: the Log File and the Result File.

The Log File is intended to be an auxiliary support in the trouble-shooting analysis. In fact it contains the informational messages logged explicitly by the user (with the log statement), plus messages logged by the Test Controller, when entering and exiting from the items of the test hierarchy, and messages raised by failures in testing actions (like data out of range, time-out expired when expecting messages, etc.).

The Result File summarizes the results of the executed test session. More in detail, it contains a set of parameters related to the test execution, like the identifier of the test session, the identifier of the test engineer and additional information, plus, above all, the completion stati of the executed levels of the test hierarchy (Procedures/Sequences/Cases). All the information is saved in ASCII transport format: therefore an off-line report generator imports it in order to produce reports according to predefined documentation standards.

As consequence of the described characteristics, the Result File must be considered the relevant output automatically generated by the proposed test environment. The documents generated from this file can be used as official documents to certificate the testing activities on the delivered system.

Advantages and Drawbacks of the Approach

The choice of formalizing and automating the test activities by means of the above described approach, leads to the following advantages:

- The test unambiguity, since the languages, being understandable by the machine, do not admit ambiguities.
- The test readability, because the description of the test procedures is made by a language containing powerful statements for the specification of the test actions.
- The test repeatability, as each test procedure can be executed as many times as necessary. This is useful not only for the regression testing, but just to locate the reasons of test failures (when they are not immediately clear).
- The test exhaustiveness, because large quantities of input data for testing can be automatically generated with the provided statements of the languages.
- The test is incremental, as the availability of a formal language allows to easily add new test cases during the life cycle of the project.
- The test traceability, since the possibility of referring the requirements inside the test procedures and the automatic recording of the test results, allow to keep track of the current status of each requirement (covered by at least one test, not covered by any test, not tested, successfully tested, not successfully tested).
- The test controllability, because the test procedures, being described inside ASCII files, can be put under the configuration control: therefore the revision of the test procedures can be always linked to that of the related system. Moreover the tool facilitates the control operations, as it inserts in the test results (the Result File) all the information necessary to identify the version of the executed tests.

- The automated test documentation, as the documents containing the cross-reference of the tests against the requirements and the test reports are automatically generated.
- The test reusability, as the test procedures written for a prototype of the system can be reused for real system, except for the changes regarding the connection with the system under test. Moreover, also the test procedures written for the unit test can be reused in many cases with some changes for the integration test.

Vice versa the approach has the following drawbacks:

- The dynamic test in which the timing constraints must be strictly respected is better fulfilled with environment simulators.
- A larger initial effort is required for the formalization of the test interfaces and test procedures.

Anyway, the dynamic testing can be addressed also with the proposed approach writing simple simulators (containing only the code governing the dynamic evolution) controlled by the test procedures via the statements allowing data exchange among logical devices.

With regard to the efforts in test formalization, it has been already underlined how these efforts leads to the standardization of all the test phases (unit test/ integration test/ system test) that, along with the automation of test execution and documentation, provide an overall reductions of the costs.

The EH101 Mission SW and Integration Testing

This section of the paper describes the experience gathered using the formal approach in the testing activities related to the EH101 program located at the DATAMAT premises.

The Avionic System of the EH101 is based on a dual redundant MIL-STD-1553B buses. Each bus connects a complex system controlled by a dual redundant computer unit. The first system is dedicated to the monitoring and control of the helicopter flight. The second one, named in the following Mission System, controls and monitors the operational mission of the helicopter.

DATAMAT is in charge of the development of the software for the mission computer unit (Mission Software) controlling the mission system, and of the preliminary ground integration of the mission system on the whole. The developing activities kicked off in the mid 80s are foreseen to end in 92.

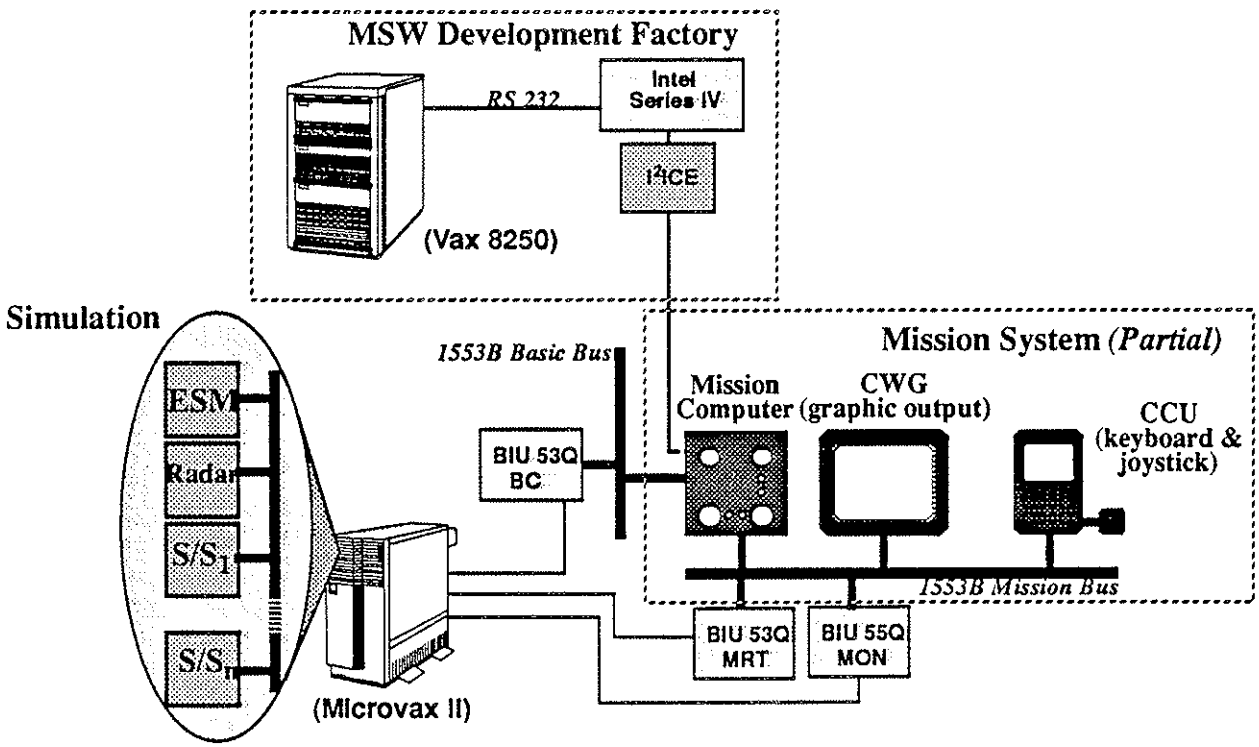

Fig. 4. The EH101 Mission Software Test Bed

A detailed discussion about the development of the Mission Software (MSW) can be found in (5.).

The complexity and criticality of this program imposed a controlled life cycle, in which all the intermediate products (both documents and software) are subjected to a formal validation by the customer. Consequently, the proposed formal approach for the testing activities has been fully applied.

The Testing Support Environment

The testing activities related to the EH101 DATAMAT program are:

1. The support to the Validation of the Mission Software
2. The support to the Integration of the Mission System.

In order to meet the above needs, it was thought to develop a unique testing system, named in the following Mission Integration Rig (MIR), supplying the means for both the test of the Mission Software and the integration of the Mission System.

As all the subsystems of the Mission System, involving the mission equipments and the operator interfaces (like the CCUs - Common Control Units - multifunction terminals), are connected via the 1553 mission bus, it is clear that the interface through which stimulate and verify the tested object is the Bus itself. Therefore the testing has been carried out accessing to the data traffic on the bus. The MIR, hosted on a commercial computer, is connected to the 1553 bus via non-intrusive standard interface cards that allow the message delivering and capturing.

The Fig. 4. shows the MIR architecture for the testing of the Mission Software. The architecture for the Mission System Integration (Fig. 5.) is quite the same with the subsystems simulators replied by the real equipments, or their emulators, connected to the 1553 bus. The emulators, provided by the developers of the real subsystems, are generally implemented using the part of the real equipment interfacing the 1553 bus, and simulating the remainder part interfacing the external environment: typically the emulator behavior is controlled, via serial lines, from this side.

The main software components supporting the approach to the Mission System Integration are shown Fig. 6.: the Test Manager, a simplified version of the Test Controller, executing the test procedures, the SW 1553 Drivers, providing the low level connection with the bus 1553, and the CCU Output Formatter. This task has been developed to format the output of the CCUs (decoding the messages sent to them) in order to be easily managed by the test procedures

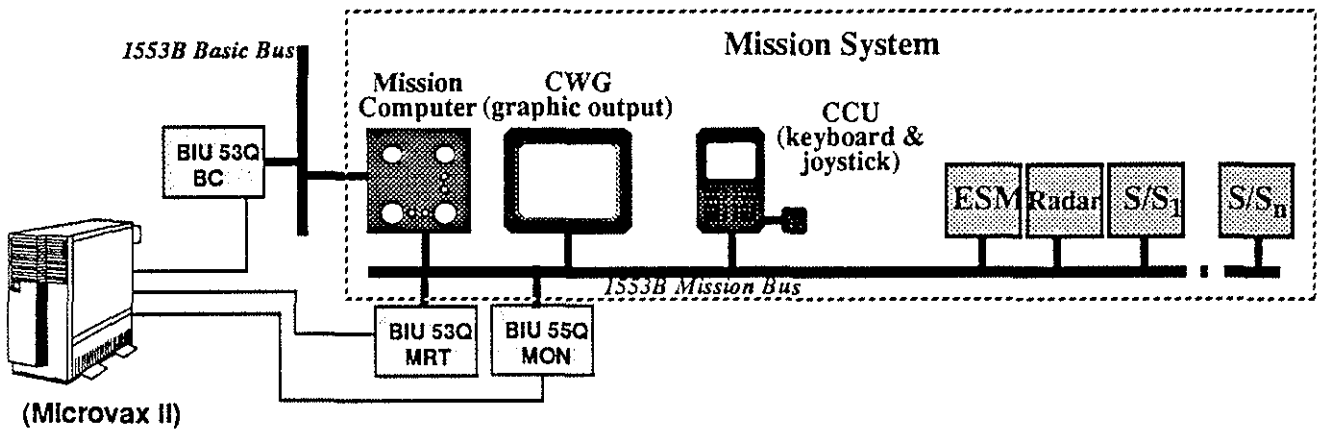

Fig. 5. The EH101 Mission System Integration Test Bed

and to be displayed by an alphanumeric terminal. A last block contains a set of simple simulators supporting the development and informal testing of the Mission Software.

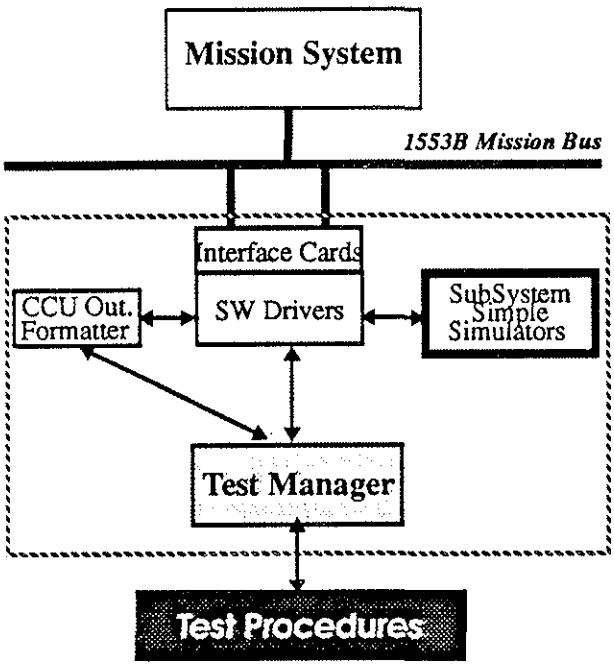

Fig. 6. Mission System Integration Approach

Test Procedures

The following three examples are drawn from two main testing phases of the EH101 program: the Mission Software Unit and Validation test, and the Mission System Integration test.

A Unit is an architectural component of the Mission Software implementing a particular function (like the management of an equipment): it is composed by one or more tasks running on the Mission Computer. A typical sequence of events during the test of a Unit is the following (indicated in the Fig. 7.):

1. Sending of a request (e.g. the presetting of an equipment, sent by another Unit), simulated by the MIR (according to the specifications contained in the Test Procedures).
2. Action of the Unit (e.g. command towards the equipment), received and verified by the MIR.
3. Response of the equipment, built and sent by the MIR.
4. Answer of the Unit (e.g. the function call result), received and verified by the MIR.

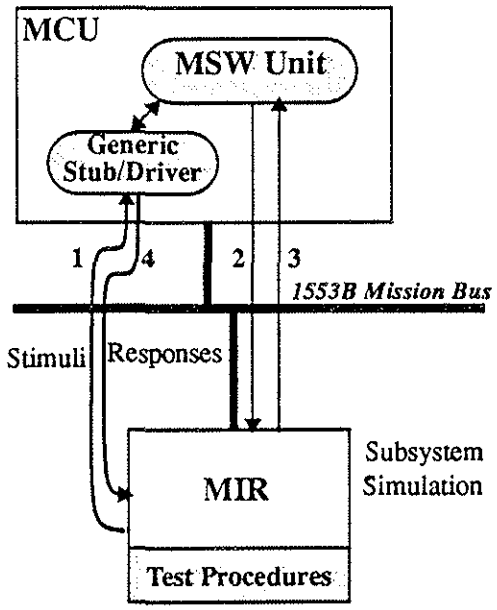

Fig. 7. Mission Software Unit Test

The listed steps can be easily translated by the test engineer in the appropriate sequence of statements of the Test Language (using the sending and receiving constructs).

The test procedure containing the above steps can be widely reused in the Mission Software Validation test. In fact, the steps 1 and 4 are replaced by actions involving the Mission Software on the whole instead of the single Unit (Fig. 8.):

1. Sending of the Operator request (e.g. the presetting of an equipment), simulated by the MIR.
2. Action of the MCU (e.g. command towards the equipment), received and verified by the MIR.
3. Response of the equipment, built and sent by the MIR.
4. Answer of the MCU towards the Operator (e.g. the actual presetting status of the equipment), received and verified by the MIR (via the CCU Output Formatter).

The test procedure can be also reused in the Mission System Integration test. In fact, in this case the steps 2 and 3 need to be changed because the actions simulating the real equipments must be substituted with actions monitoring the equipment behavior (Fig. 9.):

1. Sending of the Operator request (e.g. the presetting of an equipment), simulated by the MIR.
2. Action of the MCU (e.g. command towards the equipment), captured and verified by the MIR.

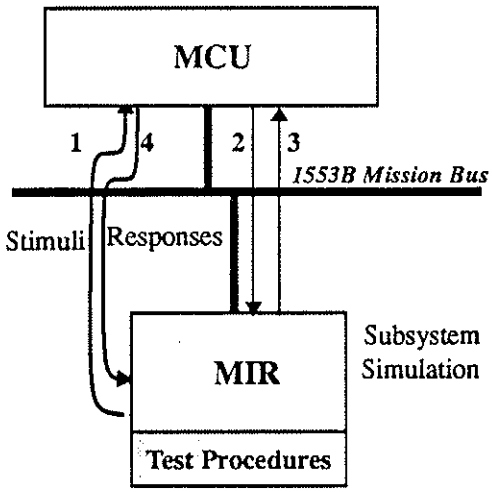

Fig. 8. Mission Software Validation Test

3. Response of the equipment, captured and verified by the MIR.
4. Answer of the MCU towards the Operator (e.g. the actual presetting status of the equipment), received and verified by the MIR (via the CCU Output Formatter).

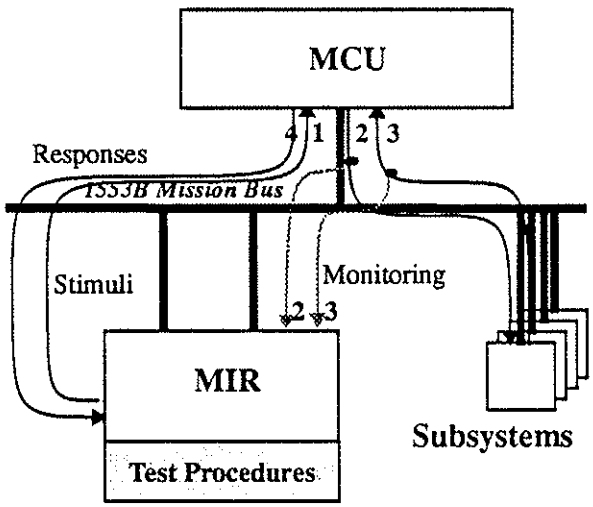

Fig. 9. Mission System Integration Test

The Hermes On-Board Mission Mock-Up Testing

This application is a prototype written in Ada language of the Hermes On-Board Mission Software, whose purpose was to gather information for the requirement definition phase of the Hermes Avionic System program. The project, in which DATAMAT was involved in the period 1989-1990, was also a test bench for the adopted design methodologies and language (Ada) and for the corresponding automated tools, and it also aimed to the set-up of standards for the product quality assurance (7.). For these reasons, it was chosen to apply the formal approach to the testing activities related to this project.

The formal approach to the testing leaded also in this case to a standard strategy in test developing and documentation, spanning from the Unit to the Validation test of the Mission Mock-Up, up to the Integration test of some Mock-Ups.

The Testing Support Environment

For the designed Hermes Avionic System architecture, and for the requirements formulated for the Mission Software, the Mock-Up can be considered a very reduced scale of the EH101 Mission Software, implemented using Ada on a UNIX operating system (in two steps: first UNIX commercial host, then UNIX embedded configuration).

The Hermes Mission Software is characterized by two main interfaces towards the environment.

1. The 1553 bus (to be simulated in this application) connecting the on-board subsystems (external to the mission computer).
2. The line for downloading the Operations Plan. This plan contains the activities to be performed during the mission exploitation; the downloading had to be provided by the test environment, too.

In this case we had not to develop drivers for specific hardware devices (the real 1553 bus), but only the basic communications between the task implementing the Mission Software and the test environment. These means were provided developing two simple Ada packages based upon a standard communication library associated to the test tools. An additional benefit provided by this library, is that the testing environment and the tested tasks can be hosted on different machines (networked over TCP/IP). The Fig. 10. shows the test bed for the integration of Hermes Mock-Ups.

Once available the Mission Mock-Up Test Environment (MMTE), the test engineers activities focused on the development of the Drivers and Stubs for the Mission Mock-Up Unit test (simple packages with standard templates supplying the data exchange between the task under test and the MMTE), and on the test procedures, of course.

For the analogies with the requirements of the EH101 Mission Software, the strategy followed in designing the test procedures was the same as in the former case, therefore no example of test activity is provided.

An important benefit gained from the usage of the formal languages for test description, was the automation of regression testing activities, that were frequently exercised. In fact the Hermes Mission Mock-Up passed through four different releases, from Ada to Ada different compilers, and from host to target configurations. The automatic test re-execution strongly reduced the time spent in the porting phases, putting quickly in evidence the related problems, such as the data codification (an important issue in interfacing products developed with different languages and/or compilers).

Conclusions

The experience in using the adopted formal approach in the two programs (EH101 and Hermes Mission SW Mock-Up) has shown that the initial effort to produce formalized test descriptions has been largely paid-back by a reduction of the overall testing work, due mainly to the automation of the testing activities (execution and report generation), and by the compliance with the customer needs in terms of quality and deadlines fulfillment.

Moreover, even if the physical context of the EH101 Avionics Testing and the Hermes Mission Mock-Up Testing (the former based on a real 1553 bus, the latter on a software simulation of it) are different, the test procedures structure of the two applications is similar. This is an important result of the formal approach, that allows the standardization of the testing activities.

The most meaningful evolution trends in this matter are towards the automation of the test design itself. This objective might be achieved with efforts in deepening the test theory associated with the systems topology, and studying the formalization methods for the systems requirements and functions. Techniques related to the Artificial Intelli-

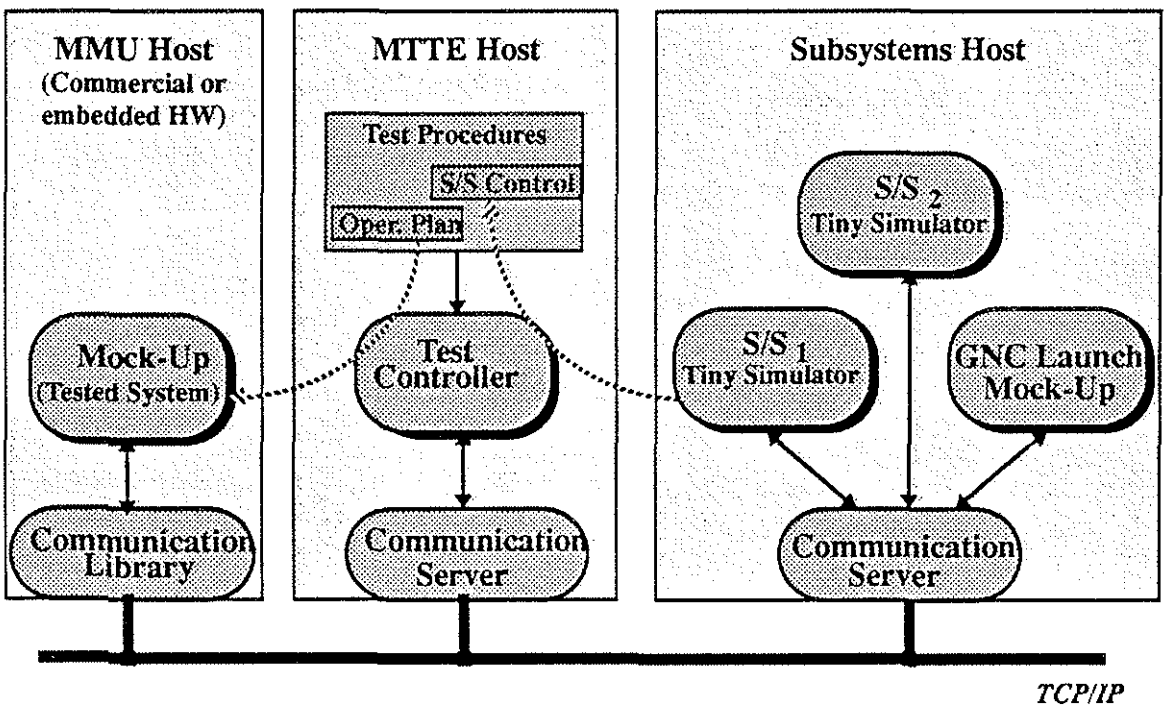

Fig. 10. Hermes Mock-Ups Integration Test Bed

gence, and specifically to the Expert Systems, may probably contribute to this research.

Another outstanding issue to be investigated is the impact of the testing and maintenance over the structures of the systems themselves. The more testable and maintainable will be the systems, the more valid and economic will be their testing and maintenance activities.

References

1. Boehm, B.W., "Software Engineering", *IEEE Transactions on Computer* (C25), pp. 1226-1241, Dec. 1976.

2. Myers, G. J., *The Art of Software Testing*, John Wiley and Sons, Inc., New-York, 1979, pp. 8-9.

3. Beizer, B., *Software Testing Techniques*, Van Nostrand Reinhold Company, Inc., New York, 1986.

4. DeMillo, R.A., McCracken, W.M., Martin, R.J., Passafiume, J.F., *Software Testing and Evaluation*, The Benjamin/Cummings Publishing Company, Inc., 1987.

5. Cambise, E., Gazzillo, S., "An Integrated Approach to Airborne Software Development", *Forum Proceedings of the 13th European Rotorcraft*, Paper n° 24, Arles, France, Sep. 1987.

6. Wallace, R.W., Fujii, R.U., "Software Verification and Validation: An Overview", *IEEE Software*, May 1989.

7. Barcellona, A., Santarelli, A., "Prototyping in Ada: Experiences in Developing the Hermes On-Board Mission Software Mock-Up", *Ada in Aerospace*, Eurospace Symposium, Barcelona, Dec. 1990.