A MODULE-LEVEL TESTING ENVIRONMENT FOR SAFETY-CRITICAL SOFTWARE SYSTEMS

A.Silva, L.Marcocci, M.Didoné

Agusta S.p.A. Business Unit Systems & Space
Tradate Unit , Tradate (VA) Italy

Abstract

Full coverage of Software Testing, from both the Functional and Structural viewpoints, is a key aspect in the assurance of Safety-Critical Systems, and a major portion of the Development effort. A strategy has been developed to achieve most of coverage during module testing in isolation. A Testing Environment allowing to describe the test cases in an understandable and formal language, and to execute them on the Target machine has been developed. It produces automatically a detailed set of Test Reports, covering the Module's functionality as well as the Module's structure and execution threads down to the machine elementary instructions It has been integrated with SD-SCICON's Perspective Development Environment and targeted for the Motorola M680X0 Microprocessors. The Testing Phase of the Software Development Life Cycle has been formalized in much the same way as the Application Software Development, introducing a standard approach, a set of rules and Configuration Management of the Module Test Sets, along with a substantial advantage in terms of efficiency and usage of human and machine resources. The product is currently in operation and has been extensively used on the EH101 Autopilot Safety-Critical Software.

Background

The Unitary Testing Phase is always on the critical path in the development of Safety-Critical Software Systems. This is due to the central position of this phase, downstream of actual code production and preliminary to integration testing. The criticality of this phase is further increased by the very stringent requirements that must be complied with in order to achieve certification. The effort for this phase often is so high that the phase can be considered the only real project bottleneck, for both manpower and computing resources.

Agusta Approach

The AGUSTA approach to Testing for Safety-Critical Avionics has been worked out within the EH-101 Anglo-Italian Helicopter Programme, and particularly for the Automatic Flight Control System (AFCS), developed in cooperation with Smiths Industries PLC for Westland Helicopters LTD. The approach has been to follow the rules given in RTCA-DO-178A for Level 1 criticality classification. The requirements call for full identification and coverage of functional capabilities at the S/W module level, as well as structural coverage of the produced code.

This approach is very well suited for control systems, where the design is such that most of the functionality is built combining a set of basic building blocks, with function complexity increasing with successive aggregations in the S/W hierarchical structure. Testing proceeds therefore in a bottom-up fashion, proceeding to the next higher level only when the current level is test-cleared. It became immediately clear, during the early stages of Unitary Testing, that the workload was large, comparable to the sum of the other development phases, also taking into account the need for non-regression testing following changes. The need to reduce workload, as well as to orderly maintain Test cases and results for multiple simultaneous baselines forced the decision to automate the process.

This Paper is being simultaneously presented at MILCOMP Europe 91

Automation had to satisfy two main requirements:

- non-intrusive checking
- integration with the EH-101 Software Development Environment, based on the Perspective PASCAL / Assembler development environment (SD-SCICON), targeted in this case for the Motorola 68000 microprocessor. The Environment is installed on VAX/VMS Host.

The concept was to provide rules and tools to the development team with facilities for:

- writing the Test Cases in a language easy to understand and maintain
- automatic generation of code for giving stimuli to the Module Under Test (MUT) and for retrieval and check of MUT responses
- automatic generation of command sequences for building executable Test Cases
- automatic run of a single Test Case or a full set of Test Cases on emulators or standard Boards and generation of Test Reports
- automatic run of a full set of Test Cases with trace and generation of Test Coverage Reports
- easy rebuild of a test suite for any Module revision

Test Environment Components

The Test Environment has been developed in three phases, clearly separating the requirement and strategies definition, the Functional Test Environment implementation and the Structural Coverage Analyzer implementation.

Test Strategy and Language Definition

A Test Strategy has been defined, dictating the requirements for the Test Environment. The Strategy has been oriented purely to Unitary Testing of Modules, based on the following definitions and considerations:

- a basic Module needs only external data
- a compound Module needs both external data and external services
- external services are provided by either basic or compound Modules
- a capability is defined as an observable functional topic, i.e., a suitably small software computation that processes observable (and in most cases alterable) inputs to yield observable outputs
- basic Module procedures are fully tested across their set of capabilities, including accuracy and precision performances
- compound Modules are tested covering the capabilities they perform internally, i.e., intermediate computations and calls to external services
- external services called are substituted by instantiations of a Generalized Stub Module, that implements a linear input-to-output transfer or constant (presettable) output. The Stub enables recording of call sequences and invocation parameters for their retrieval and check

This strategy allows to focus the attention on software-only issues, decoupled from system-level functionality, limiting the use of simulation data to the bare minimum. The check of system-level functional performances is deferred to higher levels of Integration and System Test.

A Test Language has been defined, that allows writing of a Test Case typically reflecting the test cycle: stimulus data preset, invocation of the Module Under Test, check of outputs against expected values. Each Test Case can contain several test cycles for the MUT.

A Test Case is structured in a standard fashion, dictated by the Language syntax and dependencies; a typical Test Case skeleton, omitting the details of the language syntax, is described in the following:

Test Specification Section: This section declares the Test Spec and the MUT characteristics within the Software Factory.

Header Section: Textual description of the Test Case. For Critical Software, it contains administrative data, like author, revision, date and revision history.

Declaration Section:

- Mode Specification, declaring Language or initialization mode
- Procedure (Function) Name Specification, identifying the entry to be tested and its parameters

For M68000 assembly modules, where parameters are passed in registers, the parameter list reflects registers identifiers.

- Module and Interface Specification, identifying the MUT interface characteristics
- External Modules and Interfaces Specification, identifying external services and data areas
- Data Type Definition Specification (used when the MUT requires complex structured data defined externally).
- Variable Data Access, declaring access mode for all set and check points

Input Section

- Input Constant Specification (integer, Hex or Scaled decimal), loading setpoints with data
- Input Sequence Specification for repeated setpoint loads and invocations (LOOP)

Invoke Section: Invoke the MUT procedure/function a specified number of times (default: once)

Output Section

- Single Output Specification to read a checkpoint, defining expected value/range
- Output Sequence Specification with expected check value/range sequence (to be used in conjunction with Input Sequence specifications and LOOP structures)

End Section: This section instructs the Compiler to stop parsing of the Test Specification Source.

The Language has been designed to run tests non-intrusively, and therefore no keywords are provided to instrument the code. This is essential to ensure that the tested Modules are the actual code that will be incorporated in the embedded application.

Compiler Development

A Compiler has been developed for the Test Language. It produces Perspective Pascal code, including a Pascal process providing stimuli to the MUT, collecting output values, checking them against expected output and producing a report file on the Host.

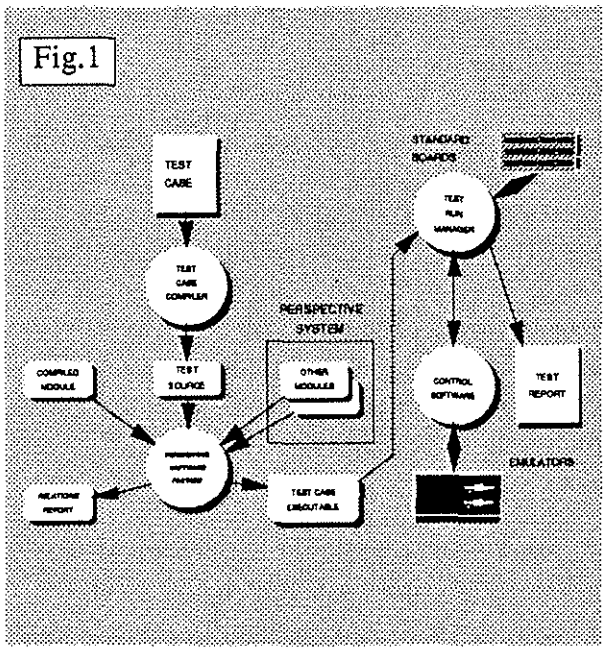
The Compiler provides also source code for the assembly language interface routines necessary for register setup and checking, as well as command streams for the complete build of the Test Case Executable image within the PSP Software Factory. Different variants of build are generated for Emulation and Host-Target run using standard Boards

The output file concatenates all necessary items, separated by appropriate tags for automatic separation and run. The Compiler is lodged under Configuration Control and stamps the output file with its revision.

Functional Test Flow Diagram

Figure 1 shows the structure of the Functional Test toolset and its interaction with the PSP Software Factory.

Fig.1



Functional Test Cases and Reports

Figures 2 and 3 show respectively a sample of a Test Case and the corresponding Test Report.

```
INPUTC SYS I9LLEE 9LLRR4 20 W 9M 7 := 60.;
{9LL.ICC2_AHRS1_Roll_rate_Euler_Lto}
INPUTC SYS I9LLEE 9LLRR6 24 W 9M 7 := 60.;

LOOP 2
   INPUT PAS INSERT := 1.1;      { insert_pointer }
   INPUT O2  := 2.2;             { select Roll }
   INPUT ENV 9VLRR4   B := 1.0; {9VL.ICC1_AHRS1_Roll_rate_validity}

   INVOKE
   OUTPUT PAS OUMPS[2].O_R[2] 9M 7 CHECK 50.,60.;   {AHRS_tmo}
ENDLOOP
```

Fig.2 Sample of Test Case

```
---      ITERATION          1  ---

INSERT                      :=                  1
SVLRR4                      :=                  1

02                          :=                  2
-- INVOKED -- COR                    --

VARIABLE NAME       COMPUTED VALUE          RESULT     EXPECTED VALUE
DUMPS(2).O_R(2)           12800             PASSED        12800


---      ITERATION          2  ---

INSERT                      :=                  1
SVLRR4                      :=                  0

02                          :=                  2
-- INVOKED -- COR                    --

VARIABLE NAME       COMPUTED VALUE          RESULT     EXPECTED VALUE
DUMPS(2).O_R(2)           15360             PASSED        15360
```

Fig.3 Same Section of Report

Structural Coverage Toolset Development

The Structural Coverage Toolset has been developed combining the RTCA-DO-178A requirements and the concept of assessing the coverage of the actual code instead of an interpretation of the design.

Since full functional coverage is mandatory for critical applications, the Structural Coverage has been designed from the start with the aim of maximum reuse of functional Test Cases. This approach is particularly promising when most of the application software is written in assembly language for performance reasons, and does not include unusual code constructs generated by a High-Level Language compiler.

The Structural Coverage requirements have thus been interpreted as follows:

- the structure of the actual MUT machine code must be precisely identified, i.e., all conditional branch points, flow junctions and the sequential code segments between them must be recognized and listed.
- the MUT must be exercised with a number of runs and stimuli conditions sufficient to execute all code instructions, giving evidence that no code section has been neglected. Hardware or Software constraints preventing complete coverage must be clearly identified and justified.
- all conditional paths based on machine-level (binary) decisions must give evidence of the decision effects in both cases. Structures implying no black-box difference in the executed statements in the two cases (e.g.,REPEAT..UNTIL loops) must be identified and covered at the functional level.

The check for complete coverage, according with the criteria stated above, can also be considered a way of highlighting deficiencies in the functional tests, although not implying complete functional test when successful.

The development has been split in two major areas: a Code Analyzer and a Coverage Checker.

The Code Analyzer

The Code Analyzer processes directly the assembly source code, parses the code, identifying branch points, junction points and loopback branches, as well as the sequential code fragments between them.

The outputs of the Code Analyzer are:

- a marked listing, where special mark labels are added to the source code, highlighting branch points, junctions and loopback branches.
- a list of sequential code fragments, each identified by the pair of mark labels it interconnects. This list will be used by the Coverage Checker as a column of labels in Route Table matrices (for single Test Cases or for the whole Test Set [Global Route Table]), where other columns (one for each MUT invocation) will show a mark in each row corresponding to an executed fragment.

The advantages of this approach are:

- no interpretation mistakes may occur
- the process is automated and repeatable
- evidence of the structure is computer-based, mapped onto the actual source code and bound by naming conventions to the MUT's controlled revisions.
- the process' outputs can be used by subsequent automated procedures.

The tool relieves the two major drawbacks of techniques of the past, when paper-and-pencil techniques have been used, involving structure diagrams and decision tables to show the shape of the Module or Procedure:

- drawing diagrams takes time, since diagrams must be based on the source code, rather than a higher-level description (e.g.,PDL), and the diagrams must be revised whenever the source code changes.
- drawing diagrams and filling decision tables is a process prone to human error, and extensive verification effort is required. Decision tables turn out to be trivial for assembly code, where all decisions are binary, but their size easily grows to impractical magnitudes.

The Coverage Checker

Given a set of Test Cases, their executable image is run on In-Circuit-emulators, activating a trace window over the MUT's program segment.

The execution output is a Trace File for each Test Case, each of which may contain multiple invocations of the MUT. Each Trace File is used for three purposes:

- as it is, to map the Test Case coverage onto the marked listing
- merged with Trace Files from other Test Cases to produce a map of the whole Test Cases set onto the marked listing
- split by individual MUT invocations, for three purposes:
  - fill the single Test Case Route Table with a column for each invocation run
  - fill the Global Route Table, invocation by invocation, for the part pertaining to the Test Case
  - map the single invocation run onto the marked listing

Mapping is shown on the marked listing by strings highlighting the executed path, making verification against the Test Specification easy. The different types of coverage listings allow immediate detection of neglected segments and give a clear picture of the executed path for each invocation.

After building a Global Route Table, it is checked with the following criteria:

- all identified paths should have been executed at least once
- all conditional paths, with the exception of paths starting at a loopback branch point, should result skipped at least once
- all conditional branch points, with the same exception, should be reached at least twice, with execution continuing once on the left and once on the right following fragment

The outputs of the Coverage Checker are:

By default:

- Coverage Listings (for each Test Case and Global)
- Route Tables (for each Test Case and Global)
- Coverage Analysis Report

On Request:

- Single run Coverage Listings

Structural Test Flow Diagram and Route Tables

Figures 4 and 5 show the phases of a typical Structural Test session and a filled Route Table.

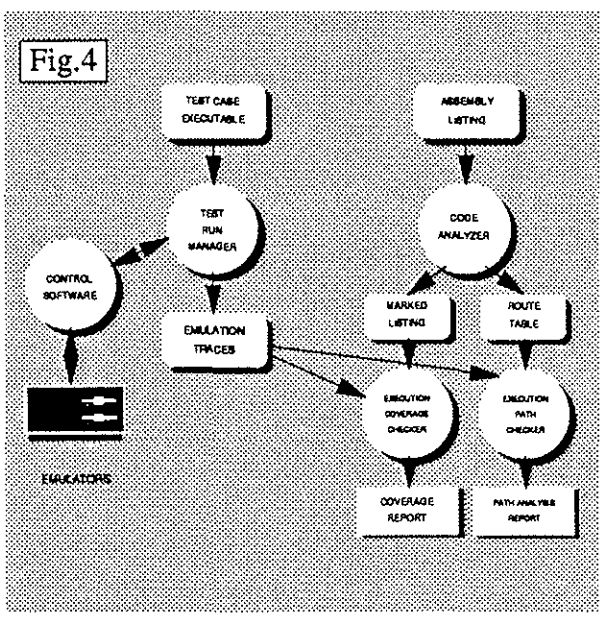


Fig.5
Filled Global
Route Table

LINK_ID				
T01_B01	*	****		
B01_T02	*	* *		
T02_B02	*	****		
T03_B03	*		****	
B03_T04	*		* *	
T04_B04	*		****	
T05_B05	*			****
B05_T06	*			* *
T06_B06	*			****
T07_B07	**	****	****	****
B07_T08	*	****		
T08_B08	**	****	****	****
B08_T09	*		****	
T09_B09	**	****	****	****
B09_T10	*			****
T10_B10	**	****	****	****
T11_B11	*	*	*	*

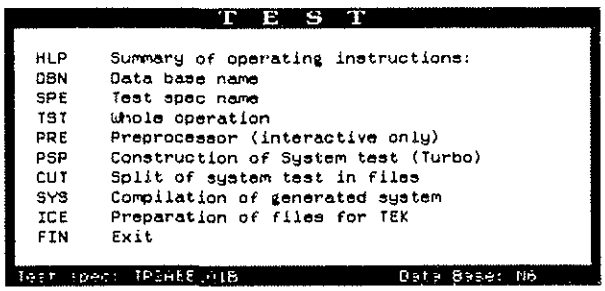


Fig.6 Functional Test Main Menu

User Interface

The Test Environment has been designed to allow interactive and batch testing of Modules, with the possibility to run single Test Cases or a full suite of Test Cases (particularly useful for non-regression testing).

A User Interface has been developed, that allows the User to select the mode of operation and the steps in the Test Process, as well as to preset parameters for batch-mode operations for several MUTs in a single session, including commands and parameters caching.

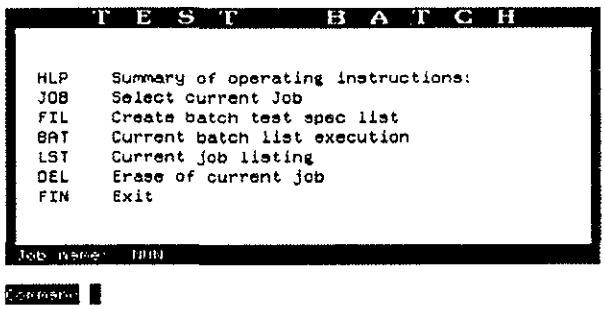
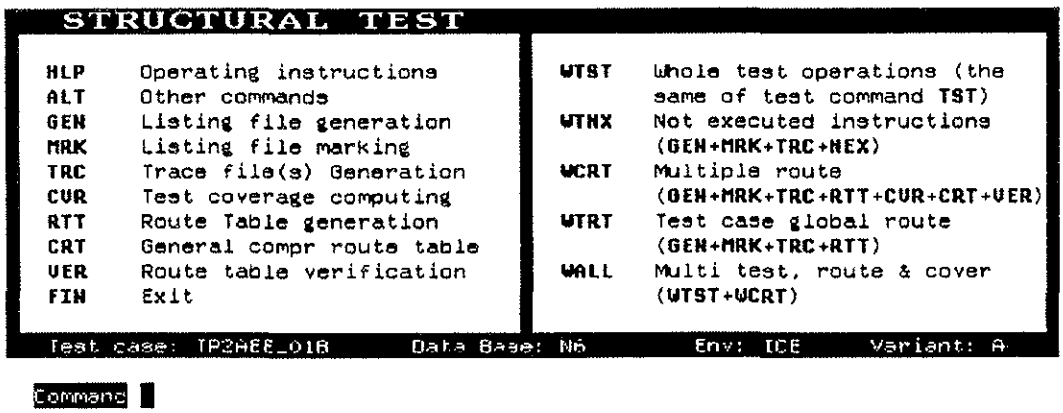
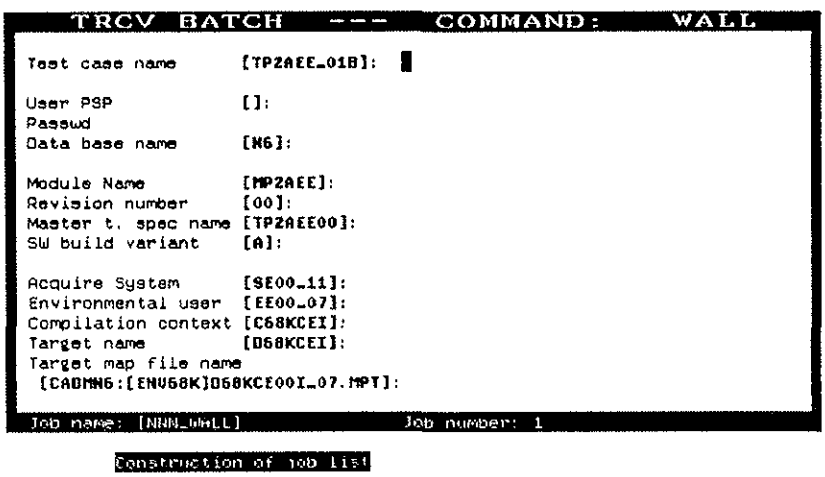


Fig.7 Batch Functional Test Main Menu



Figures 6 to 9 show actual samples of the User Interface forms, for Functional (6 & 7) and Structural (8 & 9) Test, in Interactive and Batch test modes.

Fig.9
Batch Global
Test Parameters



Conclusions

A computer-aided Test approach has been developed, focusing on the software aspects rather than system aspects in the conduct of formal Unitary Testing. Usage of the Test Environment in the EH-101 AFCS program has proved invaluable in Test effectiveness, limiting the effort to test design only. The traditional Unitary Testing bottleneck has been alleviated by usage of Hardware resources 24-hours a day in batch mode, ease of re-test of changed Modules and test data Configuration Management.

The introduction of a simple, yet powerful Test Language has allowed to transfer good Software Engineering practices to the Testing Phase, with substantial benefits in Test maintainability across several baselines and numerous revisions of the application components.

The automation of Coverage Checking, based on actual code constructs, has reduced effort, human errors and verification needs, introducing full repeatability in this Testing step.