Nationaal Lucht- en Ruimtevaartlaboratorium
National Aerospace Laboratory NLR

NLR-TP-2000-671

Unit Testing at the NLR
as applied in the MPTE project

R.H.J. Oerlemans and I.A. Woodrow

Nationaal Lucht- en Ruimtevaartlaboratorium
National Aerospace Laboratory NLR

NLR-TP-2000-671

Unit Testing at the NLR
as applied in the MPTE project

R.H.J. Oerlemans and I.A. Woodrow

This report is based on a presentation held at the 6th Nederlandse Testdag, Amsterdam, 3 November 2000.

Division:	Information and Communication Technology
Issued:	March 2001
Classification of title:	Unclassified

Summary

This paper describes how NLR has prepared and executed unit testing in the European Space Agency project '*European Robotic Arm Mission Preparation and Training Equipment*' (ERA-MPTE). The MPTE is a ground based computer system (software and hardware), supporting the preparation, simulation and validation of ERA missions. The ERA is located on the Russian part of the International Space Station.

Contents

16 Figures

(26 pages in total)

Abbreviations

AD	Architectural Design phase
ADD	Architectural Design Document
AQAP-110	NATO Quality Assurance Requirements for Design/Development and Production
BOP	Build Operations Plan
CM	Common Modules
CMM	Capability Maturity Model
COMM	COMpose Mission
CVS	Concurrent Version System
DD	Detailed Design and Production phase
DDD	Detailed Design Document
EDRS	Extract Mission Data for the Russian Segment
EMDS	Extract Mission Data for Simulation
EPC	Error report, Problem report, Change proposal
ERA	European Robotic Arm
ESA	European Space Agency
ESTEC	Research and Technology Centre for the European Space Agency
GCTC	Gagarin Cosmonaut Training Centre
GUI	Graphical User Interface
ICT	Information and Communication Technology
IDRS	Import Data from the Russian Segment
IICD	Interface Control Document
ISO	International Standards Organization
ISS	International Space Station
LIBGUI	LIBrary of GUI modules
MGI	Manage Generic Items
MPTE	Mission Preparation and Training Equipment
NATO	North Atlantic Treaty Organisation
NLR	Nationaal Lucht- en Ruimtevaartlaboratorium (National Aerospace Laboratory)
OM	Operations and Maintenance phase
PM	Private Modules
RS	Russian Segment
RSC/E-MCC	Rocket Space Corporation/Energia,-Mission Control Centre
SR	System/software Requirements definition phase
SRD	System/software Requirements document

SVVP	Software Verification and Validation Plan
TOPG	TOP level Graphical user interface
TR	TRansfer phase
UR	User Requirements definition phase
URD	User Requirements Document

1 Introduction

Because of the high-level customer requirements in the aerospace field, the NLR has a long-standing tradition with respect to product quality and reliability. Therefore the NLR division of Information and Communication Technology (ICT) continuously invests in the quality of the software development process. In 1993 the ICT division received the ISO9001/AQAP-110 certificate and the CMM level 2 predicate is expected per September 2001 followed by CMM level 3 one year later. The ICT Quality System is based mainly on the military standard MIL-STD-498 for phases and products and the ESA-PSS-05-0 standard is incorporated as an alternative.

This paper describes how NLR has prepared and executed unit testing activities according to the ESA Software engineering standard ESA-PSS-05-0 in the MPTE project. Unit testing is one of the activities in the software development process to ensure robust software with as few as possible bugs.

2 The European Robotic Arm Mission Preparation and Training Equipment

ERA-MPTE is the acronym for ‘European Robotic Arm Mission Preparation and Training Equipment’. The European Robotic Arm (ERA) is located on the Russian part of the International Space Station (ISS).

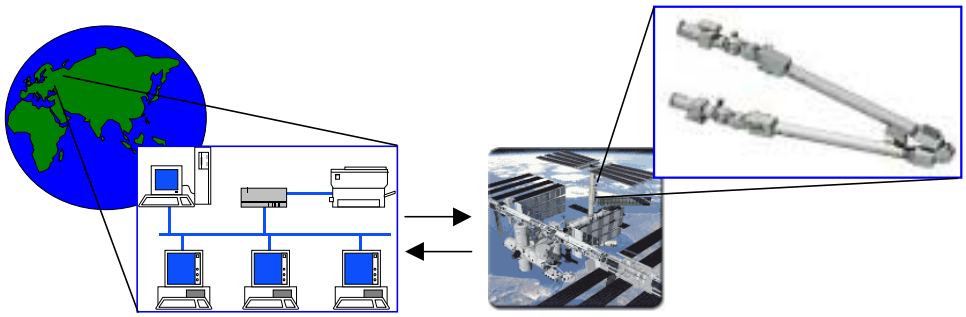


Fig 1 the European Robotic Arm Mission Preparation and Training Equipment

Performing ERA-missions, such as moving payloads, for instance moving and installing solar panels is a very complex matter because:

- In comparison to the payloads (up to 8000 kg) the arm is a very fragile construction. Moving payloads is comparable to controlling an oil tanker at sea.
- Payloads are to be moved while avoiding collisions with the ISS. Therefore a suitable path must be defined and checked beforehand.

- The extra vehicular involvement of cosmonauts (space walks) must be minimised because of the hostile conditions (extreme temperatures, bad lighting conditions, danger of impact of particles, radiation and zero gravity).

Among others these were the reasons for building the MPTE. The MPTE is a computer system (software and hardware), located on earth, supporting:

- ERA mission preparation, simulation and validation.
- Training of cosmonauts in executing ERA missions.
- On-earth monitoring of ERA missions being executed.
- ERA mission-evaluation after execution.

The so-called 'ERA operations plan', the result of planning, preparing, simulating and validating a mission with MPTE, is sent up to the ISS where it is executed step by step by the cosmonauts.

The MPTE will be operational at the following sites:

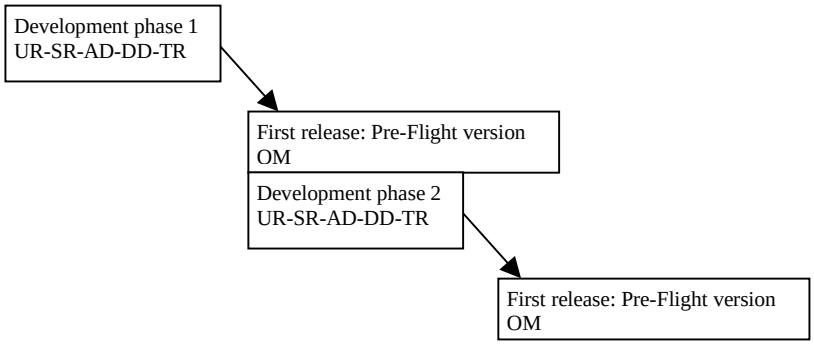
- ESA/ESTEC, Noordwijk, the Netherlands, for instructor training and software maintenance.
- The Russian Rocket Space Corporation/Energia,-Mission Control Centre Korolev (RSC/E-MCC), Moscow region, for mission preparation, control, evaluation and training.
- The Russian Gagarin Cosmonaut Training Centre (GCTC), Star City Moscow region, for cosmonaut training.

Due to the severe consequences of an error in an ERA mission it is understandable that the MPTE software development process and the resulting software must conform to severe quality requirements.

3 The MPTE software development process and practice

The MPTE software has been developed in accordance with the ESA Software Engineering Standards (Ref. 1, Ref. 2, Ref. 3). These standards are to be applied for all deliverable software implemented for the European Space Agency (ESA). They prescribe product standards, defining the overall software lifecycle, as well as procedure standards, defining the activities that are essential for managing the software life cycle, in terms of mandatory practices, recommended practices and guidelines. The tailoring of the ESA Software Engineering Standards for the MPTE project is documented in the ERA Software Standards (Ref. 4).

The Software Life Cycle of MPTE is based on the evolutionary development approach with two planned releases (Fig. 2) (Ref. 1). Most required functionality has been incorporated in the first release. This release is evaluated by ESA/ESTEC and improvements will be implemented in development phase two.



Abbreviations:
- UR: User Requirements Definition phase
- SR: System/Software Requirements Definition phase
- AD: Architectural Design phase
- DD: Detailed Design and Production phase
- TR: TRansfer phase
- OM: Operations and Maintenance phase

Fig. 2 the MPTE software life cycle

Within a development phase the MPTE software has been developed according to the so-called waterfall model or V-model (Fig. 3). This model prescribes a software development process consisting of a well-defined sequence of production and verification activities.

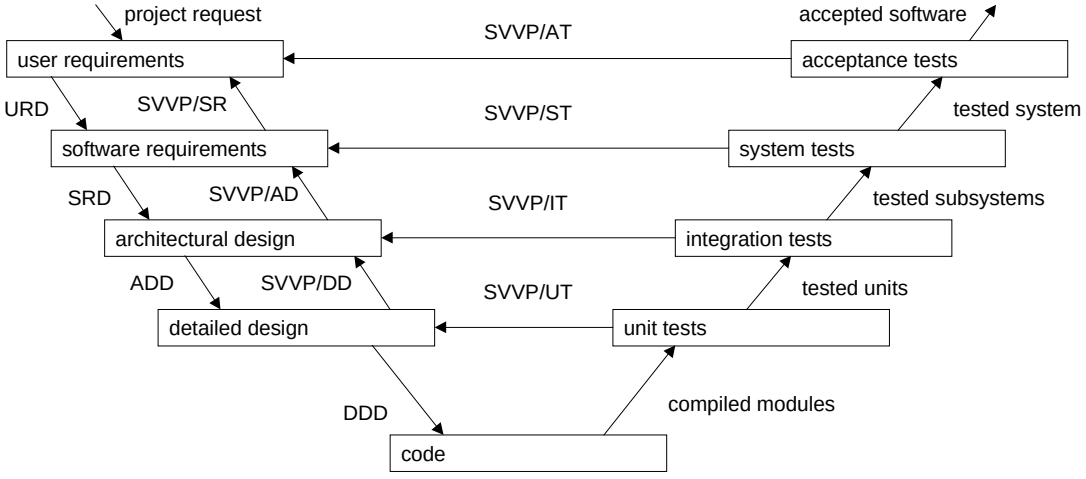


Fig. 3 Software life cycle verification and validation approach (V-model or waterfall model)

Unit Testing is the first formal test activity in a sequence of formal tests. However it is not the only way the production of high quality software is ensured on the level of detailed design and coding. Prior to and during the detailed design phase a number of actions were taken to ‘do it right the first time’:

- The detailed design and the subsequent coding phase were only started after a careful analysis of the architectural design. This resulted into an object-based approach to ensure data encapsulation as much as possible and create flexible and maintainable software.
- A library of generic basic programming modules was created:
  - Modules for run-time checking of memory allocation.
  - String manipulation modules.
  - File handling.
  - List management.
  - Error handling modules.
- The weekly build: all developers were obliged to deliver their code at the end of the week after which a successful build was required before placing updated software under configuration control. This ensured a weekly stabilisation of the software.
- A programming standard was applied for code layout and programming rules.
- Error handling was enforced using standard error handling code.
- Colleagues checked code on regular basis.
- The basic unit test requirements were made clear to the software coding team.

4 Introduction to unit testing

Unit testing refers to the process of testing modules against the detailed design while applying a systematic assembly sequence for constructing the units, the major components of the architectural design (Ref. 3, section 2.6.1). Studies have shown that unit testing is the most effective type of testing for removing bugs because less software is involved when the test is performed, and so bugs are easier isolated (Ref. 3 section 2.6.1).

According to ESA PSS-05-0 (Ref. 1) a **unit** is a major component of the architectural design and is composed of one or more **modules** (also called routine, procedure or function). The process of unit testing is described in the corresponding **unit test designs**. The outputs of unit testing are the successfully tested architectural design components.

The remainder of this chapter discusses the following unit test aspects

- Assembly sequence of the modules
- Module test techniques: white and black box testing
- Regression testing

4.1 Assembly sequence

Testing small amounts of code, modules for instance, at a time is most effective in terms of isolating bugs (Ref. 3, section 2.6.1), but it is not enough. It is quite possible that the combination of tested modules results in errors. It is therefore necessary to assemble tested modules into larger pieces of software, and to submit these assemblies to tests until finally the unit as a whole has been assembled and tested. There are two different approaches for incremental assembly:

- Top-down.
- Bottom-up.

A simple example explains the bottom-up and the top-down approach. Suppose there is a unit U1 composed of the modules M1, M2 and M3:

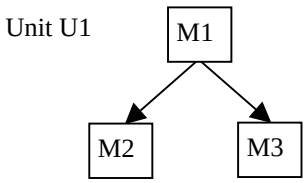


Figure 4: Unit U1 consisting of modules

For the bottom-up approach, two test drivers D2 and D3 (test programs) are required to simulate M1. First modules M2 and M3 are tested by the drivers D2 and D3, after that module M1 is tested based on the tested M2 and M3.

Figure 5: Bottom up assembly sequence

For the top down approach two stub modules (S2 and S3) are required to simulate the modules M2 and M3:

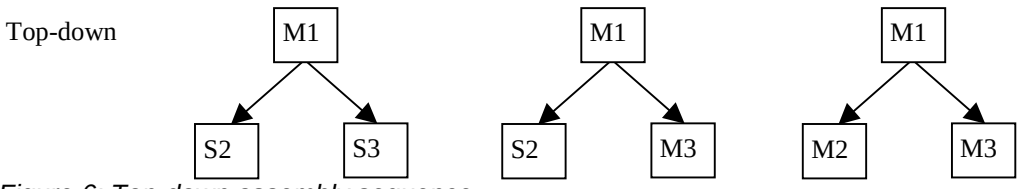


Figure 6: Top-down assembly sequence

Incremental assembly not only is effective with regard to bug isolation, it also minimises the total test effort involved in module testing. In the previous example testing M1, M2 and M3 in complete isolation before assembly would require four drivers and stubs, whereas the bottom-up incremental assembly requires only two drivers and the top-down incremental assembly requires only two stubs.

4.2 Module test techniques

The white box and black box test techniques have to be applied to the individual modules. This section describes these test techniques.

The objective of **white-box** testing is to check the internal logic of the code (Ref. 3, chapter 2.6.1.2.1). In MPTE the **structured testing method** (Ref. 3, chapter 3.6) is applied which implies:

- Full **branch coverage**: each decision outcome is executed at least once;
- Full **statement coverage**: every statement is executed at least once. This is implicitly achieved by achieving full branch coverage.

Reference 3 prescribes the structured testing method because it improves the testability by limiting complexity during the detailed design phase and it guides the definition of test cases during unit testing.

The structured testing method provides a technique, called the '**baseline method**' for defining test cases in order to achieve full branch coverage. First the **cyclomatic complexity** according to McCabe (Ref. 5) is determined which measures the minimum number of independent paths to be tested in a given module. Fig. 7 shows McCabe's equation to determine the cyclomatic complexity. Another method to determine the cyclomatic complexity is to count the number of decisions and add 1. Next a sufficient set of **independent** paths is identified. In a nutshell this means that first the so-called baseline path is identified (the path usually representing the basic logic of the module) and subsequently each decision point is switched once to identify other paths until the number of found paths equals the cyclomatic complexity of the module. Fig. 7 shows the application of the baseline method for a module with cyclomatic complexity of 3. Only 3 of the 4 possible paths have to be tested.

The number of paths found this way could be significantly less than the total number of possible paths. If for instance a module contains 10 decision points, the total number of paths equals 2^{10} which amounts to 1024 paths. The structured testing method selects only 10 of these paths as a sufficient set.

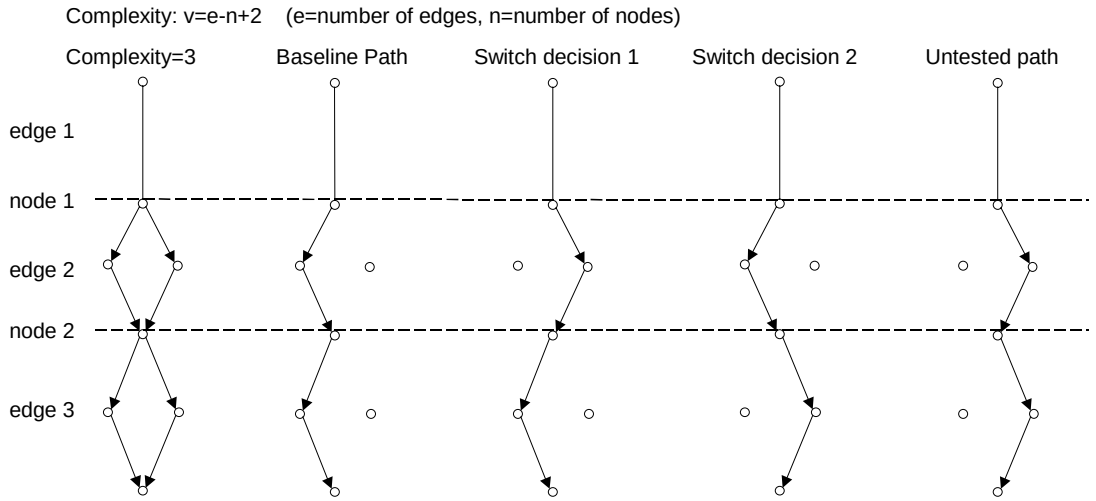


Fig. 7 the baseline method used in structured testing

The objective of **black box** testing is to verify the functionality of the software (Ref. 3, chapter 2.6.1.2.2) by checking whether or not the software does no more and no less than what has been specified in the detailed design for the whole range of possible inputs.

Testing for the whole range of possible inputs often is impracticable. In such cases the following techniques should be applied to determine a limited set of test cases with maximum effect:

- **Equivalence partitioning**: partition the range of possible inputs into 'equivalence classes' for the range of inputs (Ref. 3 section 2.6.1.2.2).
- **Boundary-value selection**: selecting suitable input values for the equivalence classes (Ref. 3 section 2.6.1.2.2).

By applying the black box techniques as mentioned above, automatically a large number of the required paths for white box testing are covered, so it is advisable to combine white box and black box testing.

4.3 Regression Testing

For any software product sooner or later the requirements and related code will change, requiring re-testing of parts of the code. Therefore unit testing should be organised in such a way that re-testing updated code, or **regression testing**, is a simple, efficient and secure activity. This can be achieved by automating certain activities such as test execution and the comparison of current and previous test results. This of course requires the configuration control of test programmes, test tools, test inputs and outputs etc.

5 MPTE Mission preparation software architecture

In this section a concise survey of the software architecture of a major part of the MPTE, the Mission Preparation component, is presented. A more detailed description of the architectural design of MPTE can be found in (Ref. 6).

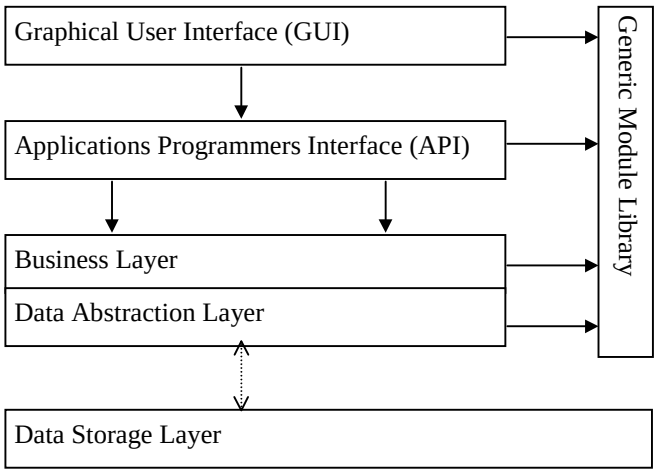


Fig. 8 Software layer view

Fig. 8 shows the software layer architecture view consisting of the following layers:

- Graphical User Interface (GUI): all the high level user interaction functionality as was foreseen in the architectural design.
- Application Programmers Interface (API): The API layer consists off modules that can be traced back to a certain architectural design unit. If this unit requires user interactions they are implemented in the GUI-layer. An API module uses modules from the Business layer and library to perform its tasks.
- Business layer: The Business layer contains modules that provide access to various data stores. These modules have an object-based approach and they are used by the modules in the API layer for data handling and manipulation. Every data storage item has a module (file with related data manipulation modules) assigned to it. The data may only be accessed through the modules offered by that module. Almost every module defines a data structure, which acts as an object. This object has only private members. Modules are provided to get and set elements from this data structure. How and where the data is physically stored is decided in the data abstraction layer.
- Data abstraction layer: This is the lowest software level, which hides the physical data storage from the Business layer modules. This means that the modules in the Business layer

are not aware of the physical source from where data is read and the physical destination (file or database) to which the data is written.

- Generic Module Library : This is a vertical layer accessible to all other layers. It contains general/generic programming modules for file, memory and string manipulation, not specific for MPTE project. The error handler module is also part of the generic module library. This error handling module is used by all layers to maintain consistent error handling. The generic module library also contains GUI functionality not specific for the MPTE project.
- The physical data storage: Data is stored either in files or in a data base. Access to the physical data is not provided directly but through the use of special modules from the data abstraction layer. In this way changes in the data storage hardly affects modules in other layers. The dotted line in figure 8 indicates data transport rather than module calls.

The following figure shows the high-level process architecture view. The circles represent processes (units) and rectangles represents data stores

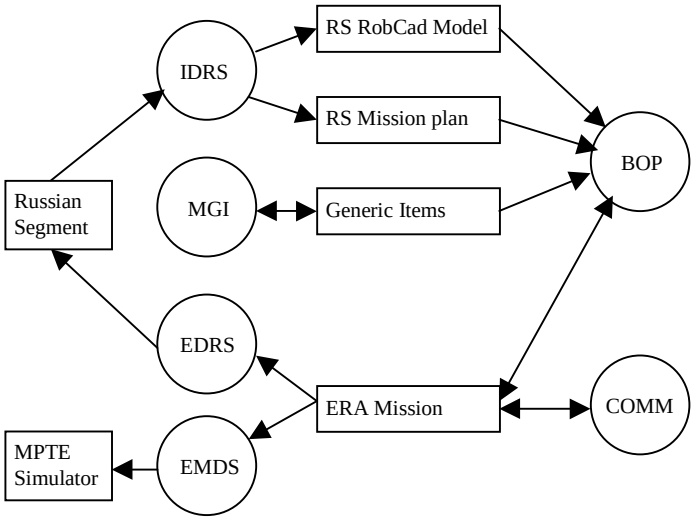


Fig. 9 High level process architecture view

This view shows the following units:

- IDRS (Import Data from Russian Segment (RS)): before a mission can be built with MPTE first a mission plan and the ISS geometry data are imported from the Russian Segment and stored in the stores RS_RobCad_model and RS_Mission_Plan.
- MGI (Manage Generic Items): maintenance of building blocks from which ERA missions are constructed. All generic item data is stored in the store Generic_Items.

- BOP (Build Operations Plan): construction of the ordered list of ERA commands required executing an ERA mission. All data is stored in the store ERA_Mission.
- COMM (COMpose Mission): converts mission data, as constructed with BOP, into a format suitable for up linking from the RS to the ISS.
- EMDS (Extract Mission Data for Simulation): makes an ERA mission available to the MPTE simulation facility.
- EDRS (Extract Data for RS): places the results of COMM in a location accessible by the RS and ready for transport to the RS.

Fig. 9 does not show the following units because they contain general functionality that is not directly related to a single high level process:

- TOPG (TOPlevel Graphical user interface): provides a graphical user interface and user-access control facilities.
- LIBGUI (LIBrary of Graphical User Interface modules): library of generic user interface modules.
- CM_PM (Common Modules and Private Modules): libraries of generic MPTE functionality, for instance data-access modules.
- Support: a library of application independent basic functionality such as modules for file manipulation, memory manipulation, error handling, string manipulation, etc.

In Fig. 10 the software layer view and the process view are combined. The data abstraction layer and the business layer are taken together because both have been implemented in the units CM and PM.

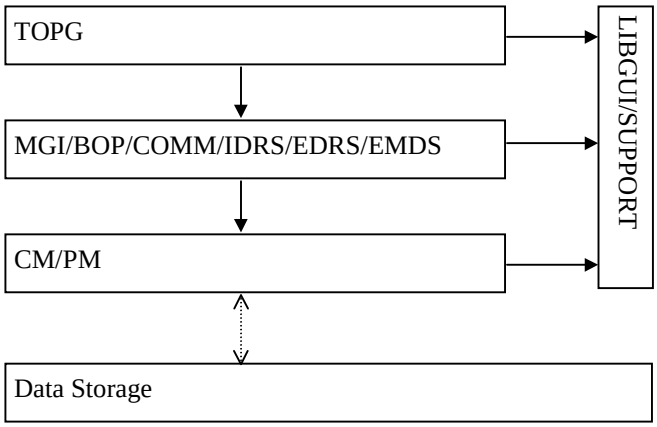


Fig. 10 Combined software layer and process view

The non-GUI code has been developed mainly in C and the GUI code mainly in Tcl/Tk.

The C-code in MPTE has been developed according to the object based approach (abstraction, modularity, encapsulation). Code, data and data access code has been grouped into files based on object logic. Code in these files is only partially accessible by code outside the files (encapsulation) and data can only be accessed through its accompanying access modules (the so-called getters and setters).

6 MPTE Software metrics

Software metrics give an indication of the amount of test work and they show which parts of the code require extra attention because of many lines of code or a large cyclomatic complexity.

- Number of modules per unit.
- Lines of code per unit and number of comment lines per unit.
- Cyclomatic complexity per unit.

As is shown in *Fig. 11* and *Fig. 12* the bulk of the code can be found in the unit CM_PM.

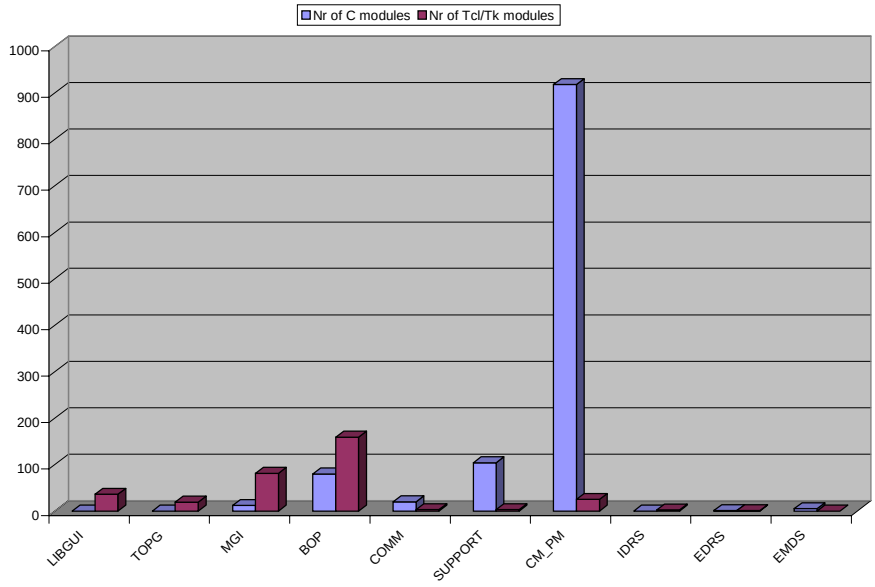


Fig. 11 Number of modules per unit

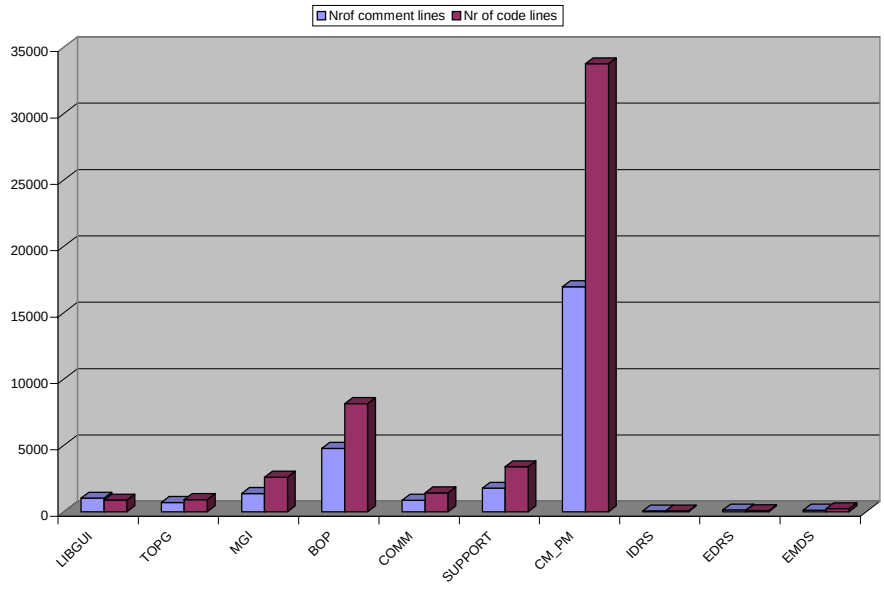


Fig. 12 Lines of code per unit

Another important software metric is the cyclomatic complexity distribution for the C and the Tcl/Tk code in the most relevant units.

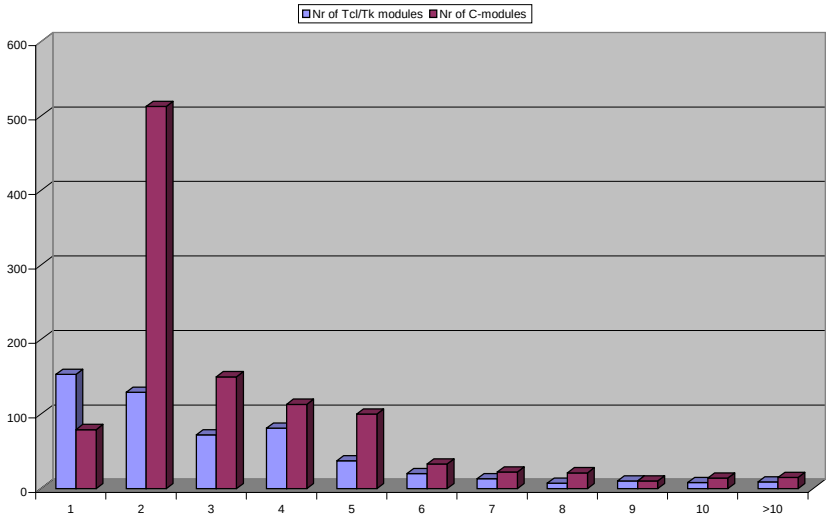


Fig. 13 Cyclomatic complexity distribution

The graph in *Fig. 13* clearly shows that most modules have a cyclomatic complexity of 2 and that the number of modules with a cyclomatic complexity of 6 or larger is quite small. This is quite important because during the development of the unit tests it turned out that the effort

involved in identifying the set of testable paths according to the baseline method for modules with a cyclomatic complexity of 5 or larger was very time consuming.

7 The unit test approach used for the MPTE-component Mission Preparation

This chapter describes the unit test approach, which involves unit test design, the unit test procedures and the selected test tools and the test team.

7.1 Unit test design

A unit test design describes how the architectural design units are assembled from the modules, how individual modules are tested and how regression testing is supported.

7.1.1 Assembly sequence

For this part of MPTE a pragmatic approach was chosen with respect to the assembly sequence, based on the analysis of the software design while ensuring that as little as possible untested software is used. The choice was made for a combined assembly sequence (top-down and bottom-up). For example: the lower level units SUPPORT and CM/PM are white-box tested and then used during unit testing of higher level units such as BOP, COMM and MGI, a clear case of bottom-up assembly. Black box testing of the lower level units is done implicitly during the testing of the higher level units, which is top-down assembly.

The reasons for this choice are:

- At the time unit testing started most code had already been developed without a clear unit test approach.
- The detailed design of the low-level code was not sufficient for defining isolated black-box tests.

7.1.2 Module testing

Per module white-box and black-box tests are conducted. These types of module testing are not considered as independent activities but are executed simultaneously.

In the ADD the specifications of the high-level units TOPG, BOP, COMM, MGI, EMDS, EDRS and IDRS are described in sufficient detail for black box testing. The low-level units CM, PM, LIBGUI and SUPPORT are not described in detail in the DDD and therefore lack documented functional specifications suitable for black box testing. On the other hand, all modules from low-level units are executed directly by modules from high-level units. They are

never executed directly by the application. A successful black box test of all high level units therefore implies the successful black box test of the low-level units.

With respect to white box testing a breadth-first approach was considered more efficient for revealing as much errors as possible in an early stage of the unit testing than a depth-first approach. In literature an initial 80% path coverage of all code is recommended (Ref. 12). After that the remaining 20% percent of paths to be covered should be looked at carefully, possibly leading to the conclusion that only testing by inspection is possible.

As has already been explained, modules belonging to an object have been grouped into a single file. A consequence is that modules in such a file cannot be tested separately, they have to be tested as a whole. Therefore in these cases the files are considered to be the smallest elements of module testing.

7.1.3 Regression testing

For each C module test programmes are written in C. Running these test programmes requires no user interaction. The test programmes, their input and their output are saved in a separate test directory and placed under version control. By doing this tests can be repeated with a limited effort.

For the GUI s/w test procedures have been developed and a capture/playback tool (Ref. 9) is used to record user actions to be used for replay during regression testing.

7.2 Standard test procedure for C-software

In this section the standard test procedure that is applied to all C-software under test is described. It consists of the following steps:

Step 1: Identify the requirements of the code under test (for black box testing).

Step 2: Instrument the code for path coverage analysis (for white box testing).
In order to keep track of the paths covered during testing, print statements are inserted into the code under test

```
before instrumentation
if a<b
      x=a-b
else
      x=a+b
if b>c
      y=2*b
```

```
after instrumentation
if a<b
      print_to_trace_file(hit in edge 1)
      x=a-b
else
      print _to_trace_file (hit in edge 2)
      x=a+b
if b>c
      print _to_trace_file (hit in edge 3)
      y=2*b
else
      print _to_trace_file (hit in edge 4)
```

Fig. 14 Illustration of code instrumentation

At the left-hand side the original non-instrumented code is shown and at the right hand side the resulting instrumented code is shown. Note that an extra *else* is added to check the outcome of bÒc. During testing the print statements write to a trace file thus registering which edges have been hit during the execution of the test program. Later this trace output is used to determine the path coverage per tested module (step 6).

Step 3: Identify the paths according to the baseline method.
McCabe's cyclomatic complexity number determines the number of paths to be tested. In the MPTE project the cyclomatic complexity ought to be < 10 so a note is to be taken if it is Ó10. Next the baseline method is applied to identify the paths to be tested (see section 4). These baseline paths then are recorded in so-called path reference files and represent 100% coverage. These files are used to determine the path coverage during analyses of the test results.

Step 4: Generate a test programme and input test data.
Generating a test programme involves: writing a test programme, creating test input, generating an executable. The test programme should do all the required white and black box tests.

The comment section of test programmes should describe the following:

- Purpose.
- Input (input files, database).
- Output.
- Requirements (refer also to ADD, DDD, IICD).
- Test description.
- Test procedure (especially deviations from the standard procedure).
- All the necessary information to run the test.

Various techniques can be applied to execute all the required baseline paths: manipulation of input parameters, input files or data in the database or manipulation of the error handler.

To monitor the test progress, relevant data/results/messages should be printed by the test programme. If the tested programme generates output in files, that effect should be monitored as well as a black box test aspect.

Fig. 15 shows steps 1 to 4. In the context of unit testing the source code file `*.c` is regarded as the module under test and can be a file that is composed of several smaller C modules (see also section 7.1.2).

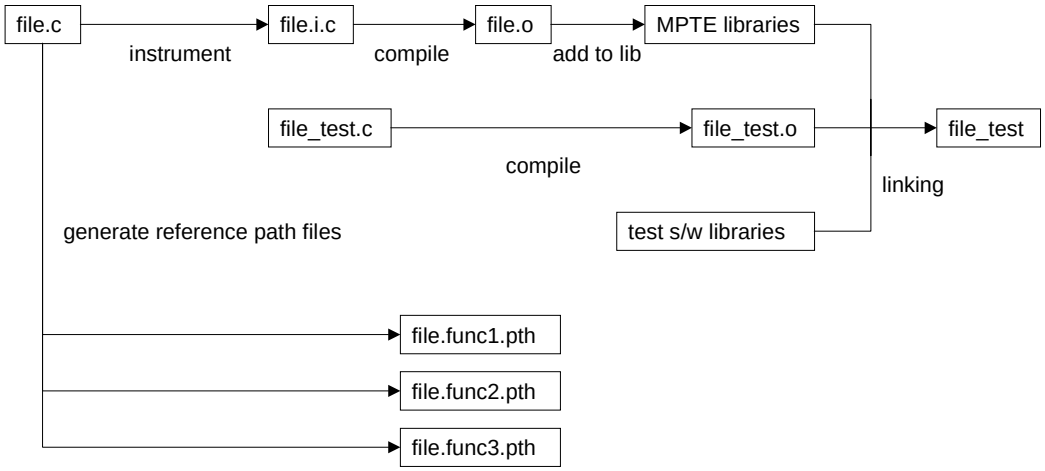


Fig. 15 Instrumentation of code and generation of a test programme and reference path files

Step 5: Execute the test program.

Step 6: Analyse and verify the test results with respect to the requirements.
Compare the test results with respect to the black box and white box requirements.

Steps 5 and 6 are presented schematically in Fig. 16

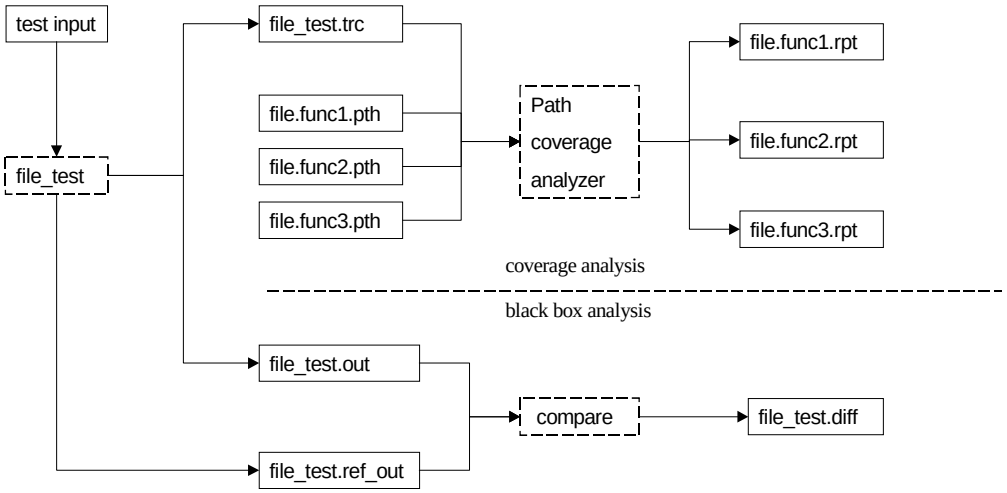


Fig. 16 Running the test programme and analysing the unit test results

Step 7: Clean up instrumented code.
After testing the instrumented code is deleted, it is not placed under configuration control.

Step 8: Record test findings in a unit test report.

7.3 Unit test tools

In several steps of the standard test procedure tools can be used to support the tester by automating repetitive tasks in the test procedure.

Path coverage tests can benefit from automation of code instrumentation and coverage analysis. For this several tools are available. Some tools require a certain approach with respect to software development, others take the code as it is. For path coverage testing of the C-code the following tools were evaluated:

- Cantata (Ref. 7):
    1. Is designed for isolated testing of units by replacing sub-routine calls by stubs and creating test programs per unit under test. Code thus must be ‘detached’ from it’s usual environment and run in a Cantata test environment which is a time consuming activity.
    2. The reporting facilities are less suitable for path coverage testing because not the tested independent paths are reported but the number of hits of individual edges. It therefore isn’t possible to test multiple paths in a single test case/run: for each path a separate test case must be implemented and run.
    3. Instrumentation of code is a time consuming manual activity.
    4. Determining what paths to test is a manual activity.

- TCAT-PATH (Ref. 8):
  1. The code of interest can be tested in its usual environment, without any code changes. The only requirement is the linking of a TCAT library.
  2. The reporting facilities support multiple paths testing in a single test. Individual tested paths are displayed afterwards.
  3. The code of interest can be instrumented automatically. No manual action is required.
  4. A TCAT tool supports determining what paths to test as long as the cyclomatic complexity does not exceed 4 or 5. The higher the complexity, the less useful the output of the tool.

TCAT-PATH provided the highest level of automation and moreover TCAT-PATH was the only tool, which could instrument available code for path coverage. So the logical choice was for TCAT-PATH.

For path coverage of the GUI-code (Tcl/Tk) an in-house tool had to be developed because no commercial tool was available.

For the support of the regression testing of the GUI code the tool CAPBAK/X (Ref. 9) was selected. This tool records mouse and keyboard actions and which can be played back.

For the configuration control of the test software CVS (Ref. 10) was used and for EPC handling Keystone (Ref. 11) was used.

7.4 Unit test team

Unit testing started at the moment most code had been developed. The test team had to deal with thousands of lines of untested code. In order to arrange effective testing, a test team was created, comprising:

- Unit test manager.
- C-code testers.
- Tcl/Tk code tester.
- Software developers to be consulted by the rest of the team.

The code testers had little or no knowledge of the software under test. Therefore software developers had to be consulted regularly. This is not the most efficient way of testing. It would have been better if a test manager had been appointed right at the start of the system design so that a unit test approach could have been developed simultaneously. This approach should have

been communicated to the software developers before the start of coding. They should have developed and implemented the test cases while coding and the test manager should have monitored this process.

8 Conclusions and lessons learnt

Unit testing in the MPTE project was started at a time that most code had been developed. This is comparable to picking out the failures at the end of the assembly line, which obviously is an inefficient way of attaining high quality end products. The most efficient way to prevent mistakes in the end product is applying good basic craftsmanship, good practices and appropriate tests and checks from the very start of coding.

In this project good basic craftsmanship and practices have been applied and have indeed resulted in a very small number of coding mistakes and in robust non error prone code. It did however require the development of extra code and a general increase of code complexity. This extra code also had to adhere to the strict unit test requirements. A less sophisticated way of code development would have resulted in less code and would have been rewarded by less unit test effort which is an unsatisfying situation because it can restrain developers from writing more but better code. Looking back, unit testing revealed only a few errors, so the good development practices that were applied could have been compensated by mitigated test requirements.

If path coverage is applied during unit testing, the cyclomatic complexity of modules should be as low as possible (preferable Ò5) as the required test effort increases exponentially with an increasing cyclomatic complexity. Testing a single path in a module with a cyclomatic complexity of for instance 10 takes significantly more time than testing a single path in a module with a cyclomatic complexity of 2 or 3.

Unit testing should be the shared responsibility of the project manager, the software architect, the detailed designers, the software developers and the test team. It is most efficient and effective if the test-design and testing is done parallel to system design. A project’s test manager should therefore be appointed at the start of the system design to coordinate and plan the test activities.

A lesson learnt once more is that unit testing, based on such strict requirements, should not be under-estimated. It is a very time and budget consuming activity and requires experienced staff and professional management.

9 References

1. *ESA Software Engineering Standards*, ESA-PSS-05-0, issue 2
2. *Guide to the software detailed design and production phase*, ESA-PSS-05-05, issue 1.
3. *Guide to software verification and validation*, ESA-PSS-05-10, issue 1.
4. *ERA Software Standards*, HS-SG-ER-005-FSS.
5. *Structured testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*, T.J.McCabe, National Bureau of Standards Special Publication 500-99, 1982.
6. *ERA MPTE Architectural Design Document*, NLR-CR-97217-L, P. de Pagter, 1999.
7. *Cantata*, IPL, UK, http://www.iplbath.com.
8. *TCAT-PATH Path Test Coverage Analyzer*, Version 8.2, Software Research, Inc., USA, http://www.soft.com.
9. *CAPBAK/X*, Software Research, Inc., USA, http://www.soft.com.
10. *Version management with CVS*, Per Cederqvist et al, 1993, Signum Support AB.
11. *Keystone*, Stonekeep Consulting, USA, http://www.stonekeep.com.
12. *Code Coverage Analysis*, Steve Cornett, Bullseye Testing Technology, http://www.bullseye.com/coverage.html.