

8. Testarea sistemelor embedded

8.1. Notiuni generale despre testare

8.2. Tehnici primare de testare

8.3. Testarea sistematica si reproductibila

8.4. Testarea SIL

8.5. Testarea HIL



UNIUNEA EUROPEANA



MINISTERUL EDUCATIEI SI
CERCETARII



FONDUL SOCIAL EUROPEAN
2007-2013



INSTRUMENTE STRUCTURALE
2007-2013

8.1. Notiuni generale despre testare

8.1.1. Scop

Testarea – este un proces centrat in jurul scopului de a gasi defecte intr-un sistem. Fie ca e vorba de depanarea sistemului sau de verificarea unor conditii de acceptare, procesul de identificare a defectelor este o parte esentiala a procesului de testare. Testarea reprezinta procesul de imbunatatire a calitatii unui sistem.

Nu in ultimul rand, procesul de testare furnizeaza informatii organizatiei despre modul in care este produs sistemul. Testarea, in sine, nu imbunatateste in mod direct calitatea sistemului ci, mai degraba, indica slabiciunile acestuia si permite componentei de management sa aloce resurse pentru imbunatatirea calitatii.

Pentru a indeplini aceste functii, fiecare proces de testare trebuie prevazut cu un plan de testare (calendar de activitati) care sa specifice ce trebuie testat si cu un set de teste care trebuie executate. Trebuie subliniat faptul ca este imposibila producerea unui sistem fara defecte datorita resurselor limitate. Scopul testarii este acela de a indica modul in care resursele existente pot fi utilizate pentru obtinerea unei cresteri maxime a calitatii.

Elemente cheie: *lifecycle, testing process, testing unit, infrastructure.*

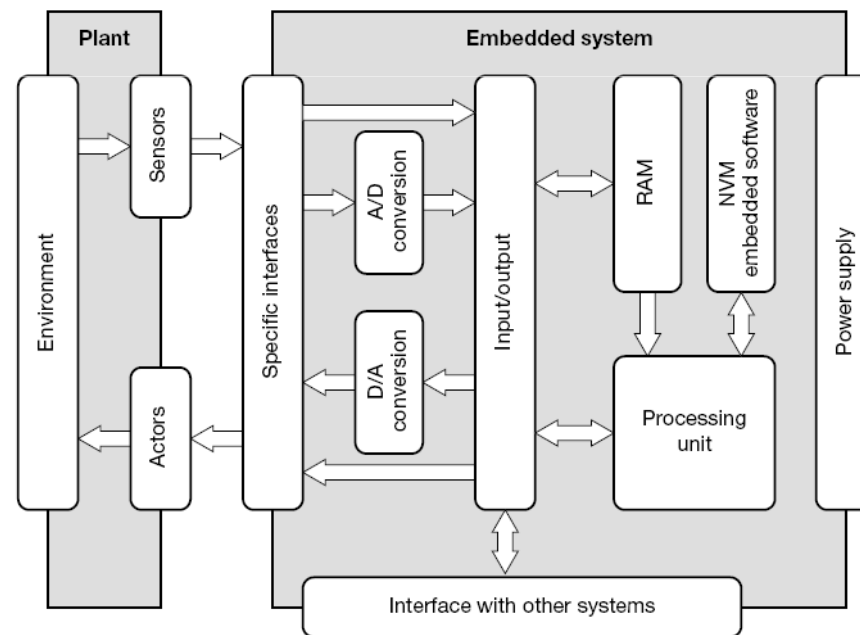


8.1. Notiuni generale despre testare

8.1.2. Testarea sistemelor embedded

Testarea sistemelor embedded:

- Proces axat pe functionalitatea sistemului embedded;
 - Proces axat pe componentele sistemului embedded;
 - Proces axat pe caraterul real-time al sistemului embedded (soft-, hard-);
 - Proces axat pe destinatia sistemului embedded (biomedical, aerospace, military, long-term, etc.);
- => Nu exista o abordare **unica** pentru orice sistem embedded.

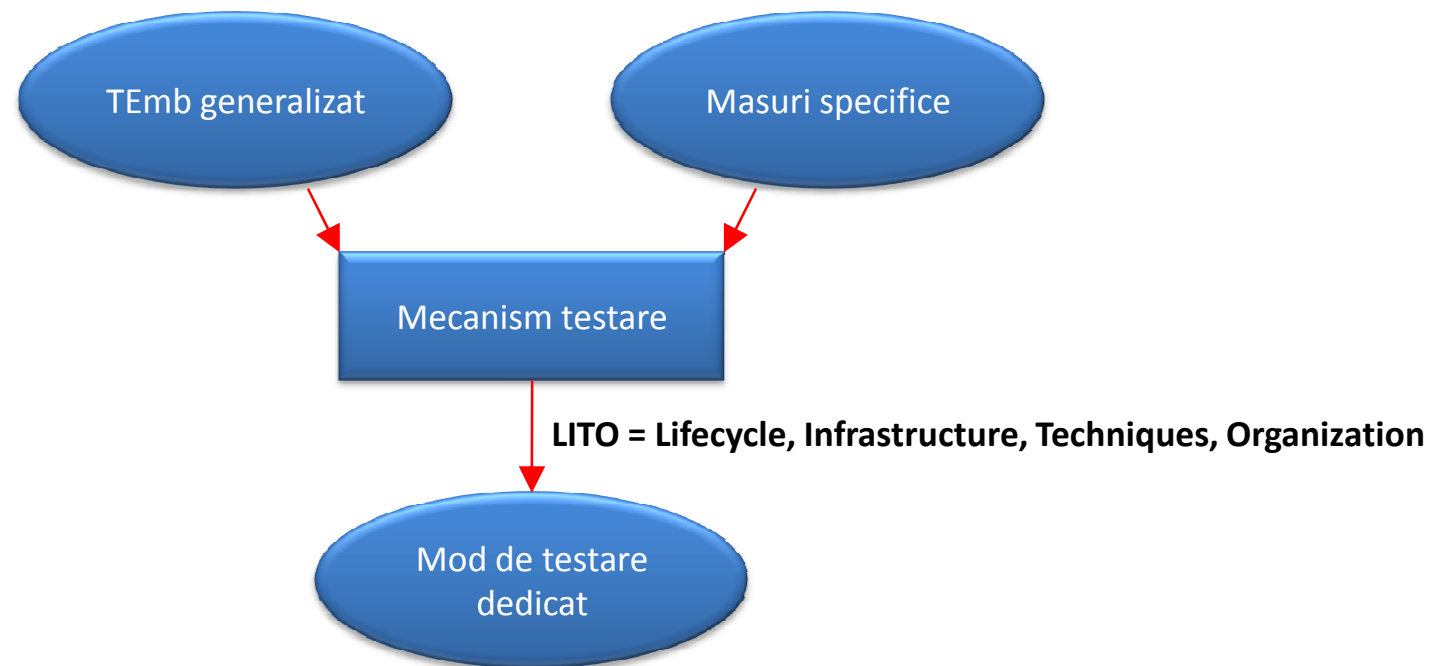


8.1. Notiuni generale despre testare

8.1.3. Metoda TEmb

Temb:

- Este e o metoda care permite abordarea unui mod de testare potrivit pentru un sistem embedded particular;
- Propune un mecanism de asblare a unor elemente generice aplicabil oricarui proces de test;
- Propune un set de masuri specifice caracteristicilor relevante ale sistemului embedded;



8.1. Notiuni generale despre testare

8.1.3. Metoda TEmb

Mecanismul care se afla la baza generarii procesului dedicat de testare va analiza urmatoorii factori:

Riscul: masurile de testare trebuie sa acopere riscul implicat de calitatea necorespunzatoare a produsului.

Caracteristicile sistemului: masurile de testare trebuie sa ia in calcul problemele ridicate de caracteristicile tehnice ale sistemului analizat: algoritmi, semnale AD, safety-critical, etc.

Characteristics	Lifecycle	Infrastructure	Techniques	Organization
Safety critical	Master test MOD-00-56, safety test, load/stress level	Coverage analysis	FMEA = Failure Mode Effect Analysis FTA = Fault Tree Analysis Rare-event test	Safety manager Safety engineer
Technical- scientific algorithms	Algorithm validation	Coverage analyzers, Threat detectors	Evolutionary algorithms, Threat detection	Math expertise

8.1. Notiuni generale despre testare

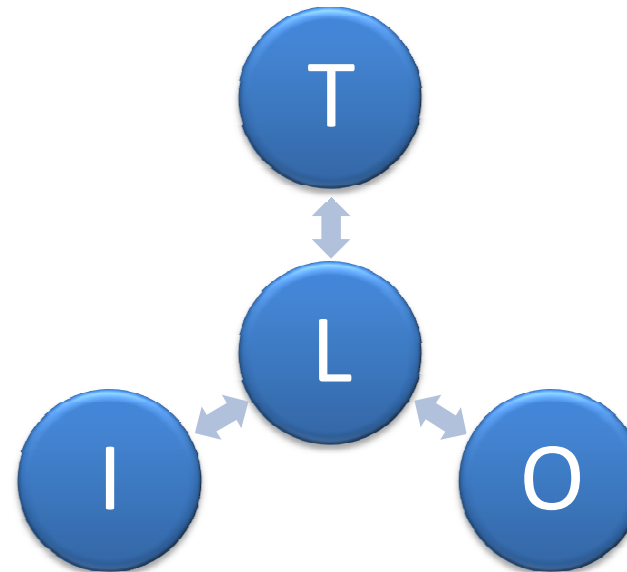
8.1.3. Metoda TEmb

Lifecycle: ciclul de viata al produsului – defineste ce activitati trebuie efectuate si in ce ordine. Permite factorilor de management si de testare sa aiba control asupra intregului proces;

Infrastructure: infrastructura defineste ce este necesar in mediul de testare pentru a putea indeplini activitatile planificate;

Techniques: set de tehnici standardizate pentru efectuarea testelor;

Organization: defineste rolurile si expertiza necesara pentru personalul de testare precum si modul de interactiune inter- si intra-compartiment;

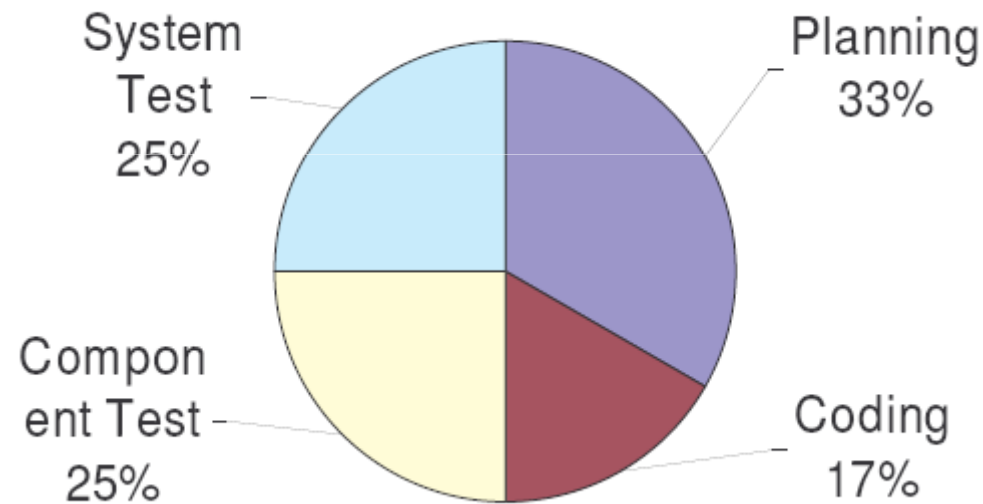


8.1. Notiuni generale despre testare

8.1.4. Ciclul de viata al sistemului

Lifecycle: ciclul de viata al produsului – defineste ce activitati trebuie efectuate si in ce ordine. Permite factorilor de management si de testare sa aiba control asupra intregului proces;

Distributia timpului in proiecte software: accentuarea planificarii si testarii [Brooks]



The sooner you start coding, the longer it will take to finish the program

8.2. Tehnici primare de testare

8.2. 1. Filozofia testarii

Premise eronate ale testarii:

- Un program poate fi testat complet;
- Testand complet un program se asigura functionarea lui corecta;
- Misiunea testarii este sa asigure corectitudinea programului prin testarea lui completa;

Premise in discutie:

- Care este scopul testarii?
- Ce este testarea corecta/eronata?
- Care este efortul necesar de testare?

8.2. Tehnici primare de testare

8.2.1. Filozofia testarii

Testarea completa este imposibila:

- Exista prea multe intrari posibile:
 - Intrari valide;
 - Intrari invalide;
 - Timing diferit de intrare;
- Exista prea multe cai posibile de parcurgere a programelor:
 - Conditii, bucle, switch-uri, intreruperi,..
 - Explozia combinationala;
 - Fiecare bug gasit inseamna re-testare;
- Anumite erori de proiectare nu pot fi gasite prin testare:
 - Specificatii incomplete;
- Nu se poate demonstra logic ca un program functioneaza corect:
 - Daca un program indeplineste specificatiile, specificatiile pot fi eronate in continuare;
- Interfatarea cu utilizatorul este un proces complex;



8.2. Tehnici primare de testare

8.2.1. Filozofia testarii

Testarea NU ESTE procesul prin care se verifica daca un program functioneaza corect:

- Nu se poate verifica functionarea corecta;
- Programele nu functioneaza corect in TOATE cazurile posibile si nu vor functiona:
 - Un bun programator genereaza cam 1-3 bug-uri la fiecare 100 linii de cod;
- Testarea nu trebuie sa dovedeasca functionarea programului:
 - Daca se asteapta ca programul sa functioneze, erorile vor fi omise;
 - Fiinta umana este sugestionabila;
- Scopul testarii este de a gasi probleme:
 - Trebuie gasite cat mai multe probleme;
- Scopul gasirii problemelor este repararea erorilor:
 - Repararea celor mai importante probleme – toate problemele = timp infinit;
 - Principiul Pareto: the vital few, the trivial many;



UNIONE EUROPEANA



MINISTERUL EDUCATIEI SI
CERCETARII



FONDUL SOCIAL EUROPEAN
2007-2013



INSTRUMENTE STRUCTURALE
2007-2013

8.2. Tehnici primare de testare

8.2.2. Planificarea testarii

1. Planning

- System goals: what it will do and why
- Requirements: what must be done
- Functional definition: list of features and functionality
- *Testing during Planning: do these make sense?*

2. Design

- External design: user's view of the system
 - User interface inputs and outputs; System behavior given inputs
- Internal design: how the system will be implemented
 - Structural design: how work is divided among pieces of code
 - Data design: what data the code will work with (data structures)
 - Logic design: how the code will work (algorithms)
- *Testing during Design*
 - *Does the design meet requirements?*
 - *Is the design complete? Does it specify how data is passed between modules, what to do in exceptional circumstances, and what starting states should be?*
 - *How well does the design support error handling? Are all remotely plausible errors handled? Are errors handled at the appropriate level in the design?*

8.2. Tehnici primare de testare

8.2.2. Planificarea testarii

3. Coding and Documentation

- Good practices interleave documentation and testing with coding
 - Document the function as you write it, or once you finish it
 - Test the function as you build it. More on this later

4. Black Box Testing and Fixing

- After coding is “finished” the testing group beats on the code, sends bug reports to developers. Repeat.

5. Post-Release Maintenance and Enhancement

- 42% of total software development budget spent on userrequested enhancements
- 25% adapting program to work with new hardware or other programs
- 20% fixing errors
- 6% fixing documentation
- 4% improving performance



UNIUNEA EUROPEANA



MINISTERUL ÎNVIȘĂRII ȘI
CERCETĂRII ȘTIINȚIFICE



FONDUL SOCIAL EUROPEAN
2007-2013



INSTRUMENTE STRUCTURALE
2007-2013

8.2. Tehnici primare de testare

8.2.3. Testarea incrementala vs. testarea big-bang

Incremental Testing

- Code a function and then test it (*module/unit/element testing*)
- Then test a few working functions together (*integration testing*)
 - Continue enlarging the scope of tests as you write new functions
- Incremental testing requires extra code for the *test harness*
 - A *driver function* calls the function to be tested
 - A *stub function* might be needed to simulate a function called by the function under test, and which returns or modifies data.
 - The test harness can *automate the testing of individual functions* to detect later bugs

Big Bang Testing

- Code up all of the functions to create the system
- Test the complete system
 - Plug and pray



8.2. Tehnici primare de testare

8.2.3. Testarea incrementală vs. testarea big-bang

Finding out what failed is much easier

- With BB, since no function has been thoroughly tested, most probably have bugs
- Now the question is “Which bug in which module causes the failure I see?”
- Errors in one module can make it difficult to test another module
 - If the round-robin scheduler ISR doesn't always run tasks when it should, it will be hard to debug your tasks!

Less finger pointing = happier team

- It's clear who made the mistake, and it's clear who needs to fix it

Better automation

- Drivers and stubs initially require time to develop, but save time for future testing



8.2. Tehnici primare de testare

8.2.4. Bug report

Goal: provide information to get bug fixed

- Explain how to reproduce the problem
- Analyze the error so it can be described in as few steps as possible
- Write report which is complete, easy to understand, and non-antagonistic

Sections

- Program version number
- Date of bug discovery
- Bug number
- Type: coding error, design issue, suggestion, documentation conflict, hardware problem, query
- Severity of bug: minor, serious, fatal
- Can you reproduce the bug?
- If so, describe how to reproduce it
- Optional suggested fix
- Problem summary (one or two lines)



8.2. Tehnici primare de testare

8.2.5. Testarea Clear box (White box)

How?

- Exercise code based on *knowledge of how program is written*
- Performed during Coding stage

Subcategories

- Condition Testing
 - Test a variation of each condition in a function
 - True/False condition requires two tests
 - Comparison condition requires three tests
 - » $A > B$? $A < B$, $A == B$, $A > B$
 - Compound conditions
 - E.g. $(n > 3) \ \&\& \ (n \neq 343)$
- Loop Testing
 - Ensure code works regardless of number of loop iterations
 - Design test cases so loop executes 0, 1 or maximum number of times
 - Loop nesting or dependence requires more cases

8.2. Tehnici primare de testare

8.2.6. Testarea Black box

Complement to white box testing

Goal is to find

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavioral or performance errors
- Initialization and termination errors

Want each test to

- Reduce the number of additional tests needed for reasonable testing
- Tell us about presence or absence of a class of errors

8.2. Tehnici primare de testare

8.2.7. Testarea White box vs. Black box

Clear box

- We know what is inside the box, so we test to find internal components misbehaving
- Large number of possible paths through program makes it impossible to test every path
- Easiest to do *during development*

Black box, behavioral testing

- We know what output the box should provide based on given inputs, so we test for these outputs
- Performed *later in test process*

8.2. Tehnici primare de testare

8.2.8. Test plan

A test plan is a general document describing the general test philosophy and procedure of testing. It will include:

- Hardware/software dependencies
- Test environments
- Description of test phases and functionality tested each phase
- List of test cases to be executed
- Test success/failure criteria of the test phase
- Personnel
- Regression activities

8.2. Tehnici primare de testare

8.2.9. Test case

A test case is a specific procedure of testing a particular requirement. It will include:

- Identification of specific requirement tested
- Test case success/failure criteria
- Specific steps to execute test

Test Case L04-007:

Objective: Tested Lab 4 requirement 007.

Passing Criteria: All characters typed are displayed on LCD and HyperTerminal window.

Materials needed: Standard Lab 4 setup (see test plan).

1. Attach RS-232c cable between the SKP board and a PC.
2. Start HyperTerminal on PC at 300 baud, 8 data bits, 2 stop bits, even parity.
3. Type "a" key on PC. Ensure it is displayed on SKP board LCD, and in the PC HyperTerminal window.
4. Test the following characters: CR, A, a, Z, z, !, \, 0, 9



8.2. Tehnici primare de testare

8.2.9. Test case – designing a good test case

Has a high probability of finding an error

- Tester must have mental model of how software might fail
- Should test classes of failure

Is not redundant

- Testing time and resources are limited
- Each test should have a different purpose

Should be “best of breed”

- Within a set of possible tests, the test with the highest likelihood of finding a class of errors should be used

Should be neither too simple nor too complex

- Reduces possibility of one error masking another

Should test rarely used as well as common code

- Code which is not executed often is more likely to have bugs
- Tests for the common cases (e.g. everything normal) do not exercise error-handling code
- We want to ensure we test rare cases as well



UNIONE EUROPEANA



MINISTERUL ÎNVEȚĂMÎNȚII ȘI
CERCETĂRII ȘTIINȚIFICE



FONDUL SOCIAL EUROPEAN
2007-2013



INSTRUMENTE STRUCTURALE
2007-2013

8.2. Tehnici primare de testare

8.2.10. Partitionarea echivalenta

Divide input domain into data classes

Derive test cases from each class

Guidelines for class formation based on input condition

– Range: define one valid and two invalid equivalence classes

• `if ((a>7) && (a<30))..`

• Valid Equivalence Class: $7 < x < 30$

• Invalid Equivalence Class 1: $x \leq 7$

• Invalid Equivalence Class 2: $x \geq 30$

– Specific value: one valid and two invalid equivalence classes

• `if (a==20)..`

• Valid Equivalence Class: $x == 20$

• Invalid Equivalence Class 1: $x < 20$

• Invalid Equivalence Class 2: $x > 20$

– Member of a set: one valid and one invalid equivalence classes

– Boolean: one valid and one invalid equivalence classes



8.2. Tehnici primare de testare

8.2.10. Regression tests

A set of tests which the program has failed in the past

When we fix a bug, sometimes we'll fix it wrong or break something else

- Regression testing makes sure the rest of the program still works

Test sources

- Preplanned (e.g. equivalence class) tests
- Tests which revealed bugs
- Customer-reported bugs
- Lots of randomly generated data

8.3. Testarea sistematica si reproductibila

8.3.1. Caracteristicile testarii

How we design the software affects testability

- **Operability** – *The better it works, the more efficiently it can be tested.*
 - Bugs add overhead of analysis and reporting to testing.
 - No bugs block the execution of the tests.
 - The product evolves in functional stages (allowing concurrent testing)
- **Observability** – *What you see is what you test.*
 - A distinct output is generated for each input
 - System state and variables should be visible or queryable during execution (past states and variables too)
 - Incorrect output is easily identified
 - Internal errors are detected through self-testing, and are automatically reported
 - Source code is accessible



8.3. Testarea sistematica si reproductibila

8.3.1. Caracteristicile testarii

- **Controllability** – *The better we can control the software, the more testing can be automated and optimized.*
 - All possible outputs can be generated through some combination of inputs
 - All code is executable through some combination of input
 - Software and hardware states can be controlled directly by the test engineer
 - Input and output formats are consistent and structured
 - Tests can be conveniently specified, automated and reproduced
- **Decomposability** – *By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting*
 - Software is built from independent modules
 - Modules can be tested independently
- **Simplicity** – *The less there is to test, the more quickly we can test it.*
 - Functional simplicity – no extra features beyond requirements
 - Structural simplicity – partition architecture to minimize the propagation of faults
 - Code simplicity

