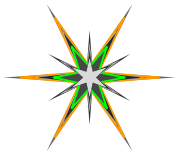


S T S A R C E S

Standards for Safety Related Complex Electronic Systems

Annex 9

Safety Validation of Complex Components

Intercomparison black box/white box tests

Final Report of WP3.2

Dr. P. Gil & J. Badiola

CNVM - INSHT

European Project STSARCES

Contract SMT 4CT97-2191

FOREWORD

STSARCES -Standards for Safety Related Complex Electronic Systems- project is funded by European Commission SMT programme. The project is divided into six research work packages and this report is part of work package 3 “Safety validation of complex components” documentation. This report introduces the results of the work package 3.2 Intercomparison black box/white box tests. We wish to underline the contribution of the Fault Tolerant Systems’ Group of the Computer Engineering Department at the Technical University of Valencia (particularly, Dr. Gil P., Mr. Gracia J. and Ms. Blanch S.) to the development of WP3.2, which has participated as a subcontractor.

SUMMARY

The goal of this work package was to investigate the inherent characteristics of black box and white box approaches to be used as testing strategies and the effectiveness of common testing criteria corresponding to each approach, to discover faults.

The first part of the report introduces the concepts of black box and white box and puts them in perspective with respect to the whole life cycle of a product.
After that the study concentrates on the field of testing and analyses black box and white box approaches among other basic criteria normally used to classify the different tests like, for instance, purpose of testing, input data generation criteria, fault or criteria based testing, etc. This set of selection criteria produces a wide range of combinations. Some of these combinations are named with generic terms that make reference to the fundamental criteria for a type of tests, and thus there are terms like, probabilistic tests, fault injection tests, or just black box and white box tests.
From this point, the study considers in depth firstly the field of tests known as black box and white box and secondly explores the use of the two approaches in fault injection testing.

TABLE OF CONTENTS

1st Part

1. Introduction

When one refers a system in any phase of the design process (specification, modelling, design, testing, etc) can adopt two main approaches. These are characterised depending on the aspect the analyst, designer or tester focus its attention, that is, in the operation of the external functioning or behaviour of the system, or in the details of internal operation or structure.

In general, it is said that a **functional approach** is used referring a system, when the system is considered as a whole, emphasising its external perceived behaviour. That is, when the system is viewed as a **black-box** that interacts with other entities or systems throw its inputs and outputs. It is the definition of a system from the user point of view.

On the other hand, under **structural approach** a system is defined as a set of components bound together in order to interact; every component is in turn another system. The recursion stops when a system is considered as being atomic: any further internal structure can not be discerned, or is not of interest and can be ignored. In this case, the system is transparent to the person handling and this is why structural viewpoint is called **white box**, though it would be more appropriate to use the term glass box.

Although at first glance they seem two clearly different approaches, in practice the boundary between function and structure is fuzzy.

If a system is seen as built in layers, the outermost layer is a layer of pure function and each layer inward will be less related to the system's function and more constrained by its structure: so, what is structure to one layer, is function to the next.

As we will see later on, there are many methods to describe or represent a system (specification in natural language, functional block diagram, detailed electrical circuit, VHDL program, control flow graph, the source code, etc). Although some of them are in principle methods better adapted to a functional or global view of the system and others to represent the details of a specific characteristic or component operation, in practice, they can be oriented in both directions depending on how comfortable the designer fells or its experience about the method.

Nowadays there is no controversy between the use of an absolute structural versus functional approach: both have limitations and both target different aspects.

The programmable systems usually are decomposed/subdivide in: HW and SW, to better deal with the specific nature of each component. The functional or structural approaches are applicable both of them.

Finally, it can be said that the meaning of functional and structural concepts has a strategic sense (a manner to consider or model a system) and are not referring to a particular kind of

technique or method. However, in the testing field, this terminology is used to group a number of SW and System testing methods which principally are characterised by these view.

This WP deals with studying the application of functional and structural strategies to test whatever of the two “components” of a system (HW and SW) during validation stage.

WP3.2 has been structured into two parts. The first one is intended to make a classificacation of black box and white box testing methos and then a selection of the most suitables. The second part will concentrate on a intercomparison study of the selected methods.

2. Validation of complex electronic systems (according to EN 954)

According with EN 954, ***validation*** of the Safety-Related Parts of the Control System (SRPCS) is the process by which one determines the level of conformity of the SRPCS to their specification, within the overall safety requirements specification of the machine.

The standard gives some general guide lines (principles) on how carrying out the validation of each class of requirement, but the current contents do not result very helpful in the case of electronic complex system validation.

It is said that, validation consists of applying analysis and if necessary, executing tests in accordance with a plan. Validation is structured in:

- safety function validation;
- category validation; and
- environmental requirement validation.

To do that, just as said above, it is firstly recommended to apply appropriate analysis techniques (deductive or inductive) and check-lists, and if it were not sufficient, complete the analysis with tests. Among the possible tests to carried out are mentioned:

- functional testing, which takes into account normal and expected abnormal conditions and exercised the system functions into a prototype or into a model, simulating, in the latest the system behaviour both in static and dynamic;
- concerning with categories it is recommended to apply fault injection into the real system or into HW and/or SW models.

Eventually, the standard EN 954-1 references the draft IEC 61508 for guidance on further validation procedures in case of PES (Programmable Electronic System).

If the purpose is to cover even the scope of programmable electronic systems, at a first glance seems that it poses a modification of the design process presented in the paragraph 4.3 “Process for the selection and design of safety measures”, in order to represent at least the

basic stages of the design of a complex electronic system and later include the corresponding requirements.

A general framework that introduces the principal validation techniques and helps us to situate the BB/WB test strategies in a global validation context, could be that in the figure 1. [ESPRIT 95].

3. Design process of a system

Figure 1 of EN 954-1 standard shows a possible model of the process for the design of a safety related control system, included programmable electronic systems. It is a simplified model, which tries to be valid for a great variety of systems and technologies.

Starting from this model and adding some key steps, it is possible to represent in a bar diagram the sequence of stages and basic activities of a generic process for the development of a programmable electronic system. This way, it will be able to define, at least roughly, the test processes for HW and SW to which we refer along this report and situate more relevant testing activities in the corresponding stages (see figure 2).

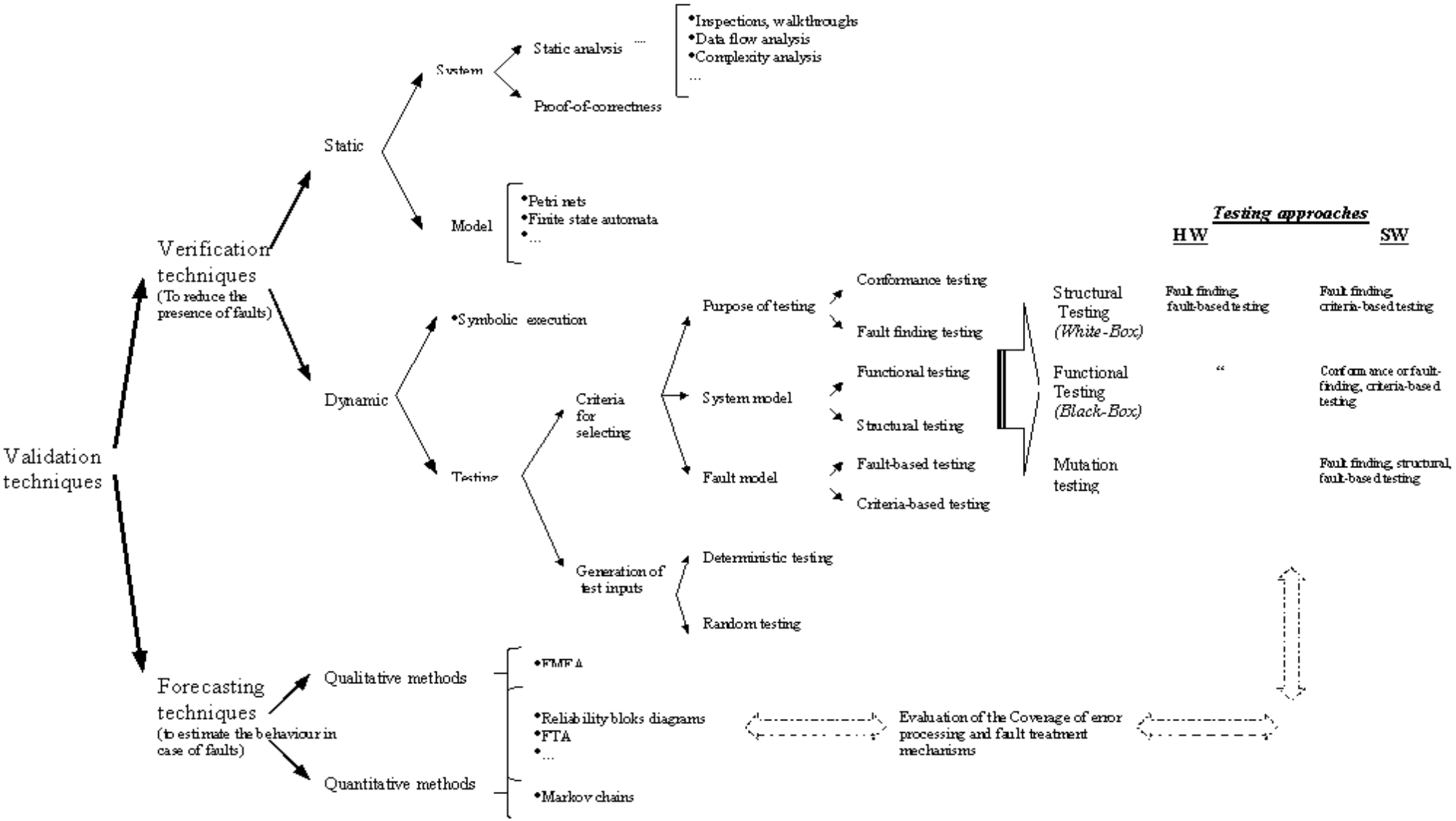
Logically behind the steps showed is hidden a process of activities (planning and managing, development, verification, quality assurance control…), and transition criteria between steps (objectives, decision criteria, etc), that will serve to define more in detail the development model, with its returns and iterations.

The total different nature of HW and SW, makes that within the general model there will be additionally a branch that leads to two parallel different processes although closely related. These processes set off from design specification phase (design, in the case of figure 2), and converge in the integration phase.

In turn, for each of them, there exist several specific models. It could be said even that each company has its own model, as a result of the necessary adaptations of a more or less known model (for instance V model for SW), to the company organisational structure and manufactured product features.

Figure 2 includes an explanation about the steps considered, which emphasises the principles that must be considered to achieve the objectives.

All of these explanations will serve to lay down a first approach on the use of structural and functional views.



NOTES:

1 - The diagram shows a global view of the main techniques structured according to their meaningful characteristics.

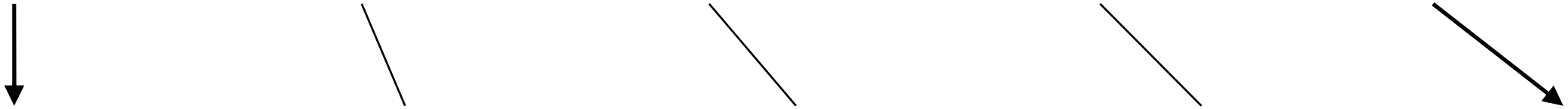
2 - Some techniques classified as dynamic may be classified as static and at the reverse, depending on whether it is apply manually on a paper or it is used an automatic tool.

3 - Fault forecasting deals with estimating the present number, future incidence, and consequences of faults.

4 – Mutation test consists of creating a big number of mutants (elemental modifications in a code line that represent a typical fault), and looking for a set of test cases which detects all the mutants. This technique is more used in the experimental comparison of other testing methods.

Figure 1 - General framework of validation techniques

Design process				Manufacturing & installation		Operation	
Concept and definition --- Analysis techniques: Semi-formal or formal methods; inspections; re views; checklists…	**Design & development** --- Analysis techniques: Inspections; reviews; semi-formal methods; checklists; static and dynamic analysis… HW component, board, subsystem testing: Function and performance simulation; functional testing; fault injection testing… SW module and integration testing: Structure-based testing; functional testing; interface testing…	**HW prototype construction Integration (HW & SW)** --- HW and SW integration testing: Structure-based testing; functional testing…	**System validation (prototype)** --- Analysis techniques: Static and failure analysis; documentation review… System testing: Functional testing; environmental feature testing; fault injection testing; performance testing…	**Manufacturing, installation and commissioning** --- Acceptance testing	**Overall validation** --- Overall system testing	**Operation and maintenance (modification and retrofit)** --- Analysis techniques: Inspections and check- lists; impact analysis… Regression testing	**Disposal (decommissioning)** --- Disposal instructions review



It is a step in which are defined the specifications of the system in terms of functional and performance requirements at the highest level (external specification) The designer (or analyst) must focus its attention in what the system must do and not how to do (i.e., separate functionality from implementation). The specification written should be complete, consistent ant accuracy, in such a way that it finally leads to a correct implementation. The size and complexity of systems usually advises the use of modelling principles (to make easier and systematise the analysis job – model simulation (animation) allows to define the information flow without function interpretation), and partition principles (to divide the operation of a system too complex as a whole in small parts easier to handle). Commonly the result, is a combined model of graphs and texts.	Design stage purpose is to define a component or system with the smallest details so that it can be realised physically (internal design specification). In a preliminary step, specifications are transformed in an architectonic and data representation. After that, it begins a process of refinement until achieving to define in detail the structure and every component. Some principles for the design are: the abstraction, the refinement and partitioning, and the modularity. The two later involve the creation of design hierarchies that consist in the definition of several levels of functional modules or sub-designs. Again, systems size leads to a partition and hierarchy of the system structure. These principles make possible to work in parallel (concurrent engineering) There exist automatic tools (EDA) able to produce a final result (e.g., the logic of a component) from a high level description, which allows the designer to focus all the attention in modelling the system with accuracy. However, design optimisation (maximum exploitation of memory resources, the HW of an ASIC), the processing speed, the control on the system, etc, during this phase of development, still in the case of EDA environments, makes structural techniques to be predominant. Mainly during the preliminary, it design should be controlled the influence that reusable subsystems and components could cause. The most used, top-down and bottom-up design methodologies, involve the creation of hierarchical designs. In the development phase will be implemented [either at logical level (logic gates diagram) or physical] and verified the components, boards, and subsystems based on design phase results.	During this stage is proceeded to the construction and the integration test of a physical prototype from all the components and subsystems developed in the prior stage. It is intended to obtain a first physical system that integer the developed HW and SW. During this stage the *engineering team*, verifies that the system behaves according to design specifications for whole HW and SW. In particular, it will be checked the system internal operation details related to interaction of HW and SW [interfaces HW – SW, partial process functions, inputs and outputs access, control functions (synchronism, etc)], and their performances (process speed, load capacity, etc).	Once there is a prototype, that apparently operates correctly, it will be submitted to a complete phase of tests to determine the degree of conformity of the prototype with its specifications written at the beginning. These tests should be executed, if possible, by objective people (i.e., people do not having participate in the design, for instance q*uality department*). The tests in this stage will be mostly functional tests at the system level, environmental tests, etc.

NOTES: *a) To describe the first part of the life cycle of a product that take place before the beginning of its manufacturing, we have used the term Design Process to be coherent with EN 954-1.*

b) In this process it has not been included the previous stages of Definition, Risk Assessment and Overall Requirement Specification and Allocation made at an application level (machinery). Thus, it uses as inputs, among other external requirements (e.g., performances, dimensions, environmental strength, etc), the safety requirements in terms of safety functions and categories.

Figure 2 - Lifecycle of a SRPCS

3.1. Domains of description and levels of abstraction in a system

In the introduction of this report it has already been presented the functional and structural approaches in a generic frame. Now, they are going to be situated/placed in a more specific context (design of programmable electronic systems), to show the design space where a designer or tester usually moves/work on.

To describe a system may be use one or several of the following description domains:

- functional,
- structural
- physical.

Moreover, within each of them, it is possible to think in a description at different levels of detail or depth, what we will call, level of abstraction. In the figure 3 has been summarised the description domains and abstraction levels characteristics of HW and SW components in a system.

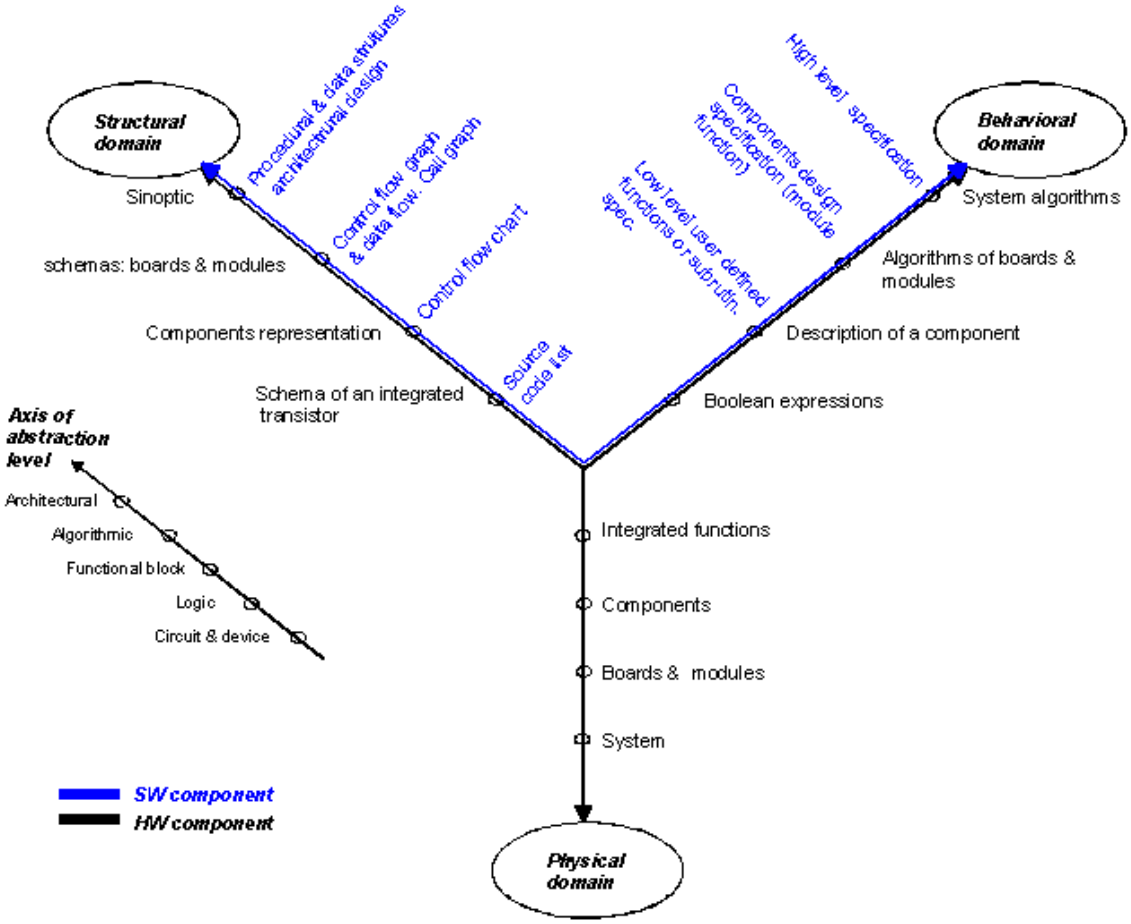


Figure 3 - Description domains and abstraction levels of a system

The use of one or another domain and the level of abstraction will come determine mostly by the stage of the design process and the specific task of the person or department involved (analyst, HW or SW engineer, tester, etc).

Referring to the functional and structural domains, which attract our interest, it can be advanced in a generic way that their degree of utilisation is that in the figure 4.

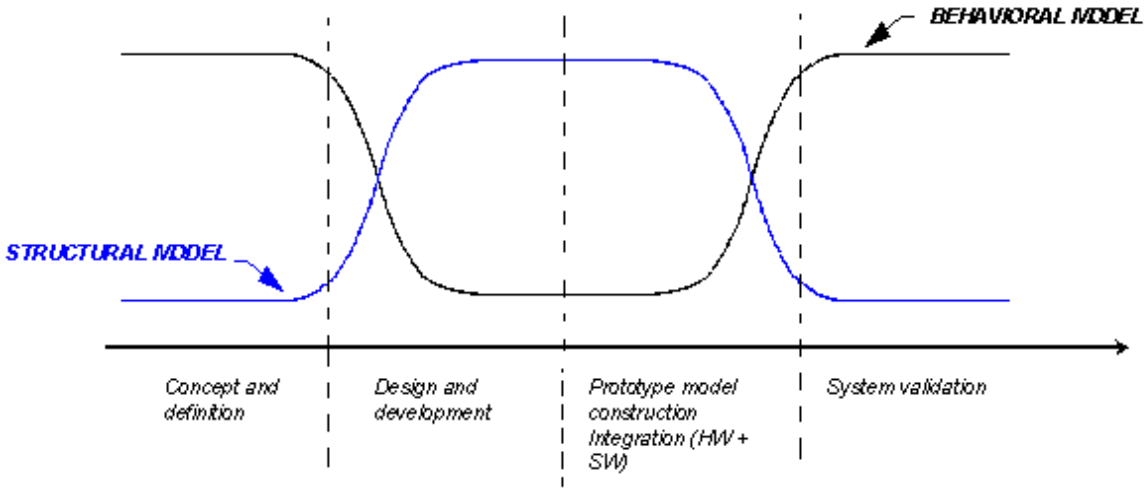


Figure 4 - Degree of utilisation of functional and structural models

3.2. Testing during design process

When testing is undertood, in a wide sense, as part of quality assurance that allow to prevent and explore possible faults introduced in the system (HW and SW), and evaluate the behaviour of the system in case of fault, it is extended along the whole design process.

Nevertheles, the term **testing** is usualy interpreted as the process of checking wether a system meets its features exercising it with some apropriate input data, being precisely this dynamic nature its distintive feature. Then, from this initial conception of the term, all the analytical and modeling methods are excluded.

As a result we find that the field of testing is restricted only to those stages in the design process where exist a real item (HW or SW), which can be exercised in an adecuate physical environment. In this report is also considered as testing, any kind of functional analysis carried out in a simulation environment which models properly all the dynamic features of the component or system.

The preventive objective of testing, responds to the potentiality of test planing and design phases to prevent faults before being implemented.

On the other hand, and despite this WP 3.2 deals with validation (i.e., what is considered as the last stage of the design process), sometimes it is not possible to conclude this final stage (to make a judgement on the degree of conformity with the safety requirements) isolately,

without taking into account other measures and verifications applied in previous stages. They are either measures to prevent faults from being introduced (fault prevention measures), or measures adopted to detect slipped faults by means of analysis or tests (fault removal measures), which could be both synthesized as fault avoidance.

This reasoning, has motivated that this WP 3.2 is not limited exclusively to validation stage, and takes into account also contributions made of other measures in other stages.

According with the design process represented in figure 2, it has been stablished three testing levels for a programmable electronic system. The objectives in each of them are different and it is spected that the set of methods used in each of them will differ. The testing levels are:

- **Components and Subsystems testing**

  A component can be considerer like the smallest item which is possible to test. HW components cover from discrete components (resistors, capacitors, transistors, sensors, etc) and ICs, to small circuits with a concrete and meaningful function. Hw components isolated are considered well proved, and then during component testing is only checked its behaviour in the circuit (its application). Within ICs, logic programmable devices need a separate treatment because they generate adittional verification and validation problems (field of WP3.3 research).

  In relation with SW, sometimes it is made a distinction between unit and component. Unit is usually the work of a programmer that consists in subprogramms or routines that at most have hundreds of code lines. However, a component is an agregate of one or more elementary units that have an entity in the architectural design of the program.

  Component testing, must demonstrate that these do not satisfy their functional specifications and/or the implemented structure do not match with that planned in the design.

  They are tests carried out at very low level and in many cases will require the creation of auxiliar circuits to feed and load the component under test with neccesary data. In Sw testing are used stub and driver modules.

  Unfortunately, many components can not be tested properly with a simple circuitry or additional software, and in those cases the complete testing is (use to be) postponed to the next testing stage (integration testing).

  There exist techniques and criterions that make easier component testing. For instance, it is called/talked of/about design for testability. SW tests are simplified when components are designed with a high degree of cohesion and the lowest coupling (i.e., a modular design with functional independency).

  In all the test process, component testing is the stage where more is used simulation technique.

A subsystem is considered from a group of circuits or SW components to/of a certain degree/order, to the final agregate (HW or SW structure). Subsystem testing in the case of SW is called integration test.

Subsystem or integration testing is carried out to demonstrate that the combination of components is not correct or consistent, in spite of components having demonstrated a satisfactory individual behaviour

These tests revel interface and interaction faults between components (e.g., the hold time of a data at the input of a component is not enough to be read; or an incorrect handling of data objects in a program).

To built and test the structure, in the development stage, one can adopt two strategies mainly: non incremental integration or big-bang, and the incremental. And into/within the incremental the approaches top-down and bottom-up.

From all the above, it is possible to deduce that component and subsistem or integration testing, require a detailed knowledge of the internal functions, performances and structure of the components, wath leads to the fact that they are usualy realised by the designer themselves.

- **HW and SW integration testing**

Till now it has been developed and tested the HW and SW separately. In this stage it is proceeded to merge the SW with the HW prototype built, and to the integration of external components (sensors, actuators, other modules, etc).

As indicated in figure 2, the engineering team verify that the system as a whole behaves according with the design specifications, and specially with some internal functional details refering to the interaction between HW and SW. If these tests detect some/any fault, the proposed changes must be first analysed to consider their extent in the system, as well as, to foresee design documentation updating and proper develops.

Finally, there will have to initiate all the necessary re-verification activities.

- **System testing**

It is intended to determine the degree of conformity of the final prototype (which is judged as a system that meets all the requirements by the engineering department) with its initial or external specifications, whenever they represent the original objectives without any error.

System tests are not restricted to the developed system, but they consider also the surrounding environment: other related systems, terminals, operators, etc, as though all the tools used to create and test it.

Tests carried out in this stage range from functional tests at the highest level, workload tests, stress tests, performance tests, storage tests, behaviour in case of fault (fault

tolerance), recovering tests, EMC inmmunity tests, functional tests under limit environment conditions, etc, to the test or revision of user manuals.

In this stage is therefore predominat a view of the system as a whole, because it is concerned of all the issues and behaviours that only can be explored testing the whole system.

The need of dealing with large size items makes infeasible, or al least rather difficult, in practise to consider structural details, except very particular cases (e.g., safety critical components, etc).

It is recommended that these tests be practiced by objective people, free of prejudice or bias, created as consequence of the design knowledge and direct participation in it.

In search of discrepancies between the built system and their objectives, it is paid an special attention to the possible translation errors made in the process of writing the specifications.

Some principles of testing valid for all the seen testing levels are shown on table 1.

Some other relevant aspects of testing that are worth being treated shortly before going on, are the **test coverage** and **test completeness criteria**.

The notion of coverage changes according with the field where it is applied. For instance, in the field of SW testing, it is a concept simply related to the extent of the test. Thus, when it is said that an statement test carried out on a program has reached a coverage of 80 %, means that that test, in total, has executed the 80 % of the statements present in the program.

When we refer to the coverage during a test campaign addressed to evaluate fault tolerancy, this term is used as an indicative of the ability of a specific mechanism or even the whole system, to detect faults.

If, in a higher level, instead of refering only to a particular test we refer to the set of tests realised in each field (e.g., SW testing, injection testing, etc), the term coverage indicates the measure or extent that has been considered during testing the possible faults in this field (e.g. errors of requirements, features/functionality, structural, implementation/coding, integration, etc., in the case of SW). In this case, it is talked about the coverage of the complete SW testing, asociating the term to the ability of test methods used, to test “all”potential kind of faults of a program. Similarly, by the coverage of a injection test we will understand the degree in which the testing techniques used (physical fault injection, software implemented fault injection, etc) cover the possible type of HW faults and transient disturbances produced mainly by electromagnetic interferences.

Table 1: Principles of testing

	Principles	Remarks
1	Plan tests long before it is time to test	
2	A designer or engineering department should avoid attempting to test its own system (some level of independence is recommended specially for system validation).	Most designers can not effectively test their own systems because they can not bring themselves to form the necessary mental attitude (wanting to expose errors).
3	The objective of testing is to find faults. Thus, a good test case is one that has a high probability of detecting an as-yet undiscovered fault, and it will be successful if it detects the fault.	This objective of testing is higher productive than that which tries to demonstrate that a system has no faults (i.e., it performs its intended functions correctly), because of human psychology is highly goal oriented.
4	Due to the fact that an exhaustive input test of a system is impossible, in addition to the principle above, when the tests are carried out one wants to select a small subset with the highest probability of finding the most errors, for all the possible inputs. Thus, a well-selected test case should have these other properties: • It reduces the number of other test case needed to achieve the predefined goal of testing. It covers a large set of other possible test cases.	The first property implies that each test case should invoke as many different input conditions as possible in order to minimize the total number of test cases necessary. The second one says something about the presence or absence of faults over or above this specific set of input values (partitioning of input domain into a finite number of equivalence classes).
5	Testing exposes presence of faults.	It can not be used to verify the absence of faults. It can increase tester confidence that a system is correct, but it can not prove correctness.
6	A test case must consist of two components: a description of the input data and a precise description of the correct output or expected result.	If the expected result is omitted, there is no way to determine wheter the system succeeded or failed. Furthermore, a tester may assess an incorrect result as correct because there is always a subconscious desire to see a correct result.
7	The results of each test should be thoroughly inspected and when a failure is detected, the causes shall be analyzed.	Many times the tester fails to detect certain faults or errors even when symptoms of those were clearly observable on the output listing. A detected fault analysis will contribute by preventing them for occurring the next time.

NOTE: Part of these principles are dicussed in paragraph 4 "Black box versus White box testing" of this report.

Finally, we could say that the coverage refers to the measure of representativity of the situations to which the system is submitted during its validation, compared to the actual situations it will be confronted with during its operational life.

The simple application of a combination of tests (test criterias or test techniques), designed considering all the possible types or classes of faults (according with the accumulated experience), seems to be a reasonable strategy (enough coverage), but it is not accepted for some authors. They suggest to adopt some other type of reference on the test completeness and stoping criteria (i.e., to judge when an item is tested enough, and a phase of testing is finished), instead of simply executing the test cases designed using those test criteria without detecting any fault.

There exist several ways of limiting the period of testing:

- stop when the scheduled time for testing expires;
- stop when all test cases designed from a general test strategy are executed without fault detection;
- stop when a set of tests representative enough of the possible faults in the item under test is applied with satisfactory coverages and without fault detection (i.e, an specific strategy);
- stop when a predefined number of errors are detected (e.g., when three errors are detected);
- stop when the number of faults found in a period of time becomes equal to an established low value;
- etc.

Some standards offer a structured classification of test techniques in such a way that they allow, in each particular case, to define the testing strategy in terms of safety requirements. It could be said that the stoping criteria used is the application of all test cases designed from a specific testing strategy. This kind of criteria has the objection of not defining a target for the test, as it can be the number of detected errors per time unit.

Another term used to define the tests, sometimes confused with the coverage, is the effectiveness. In this report, effectiveness is used to define the degree of attainment of the objectives in the shortest time. Then, we say that a test is very effective if it has a high probability of revealing faults with a small number of cases. This term can also be applied to the implemented fault control measures.

4. Black box versus White box testing

A test can be designed from a functional or a structural point of view. In **functional testing** the system is treated as a Black-box, that is, the tester is completely unconcerned about the internal behaviour and structure of the system. Rather, the tester is only interested in finding circumstances in which the system does not behave according to its specifications.

Test data are derived solely from the specifications (i.e. without taking advantage of knowledge of the internal structure of the system).

If one wishes using this approach to find all the faults in a system, the criterion is exhaustive input testing (i.e. the use of every possible input condition as a test case). However, this would imply that one should test not only valid inputs (i.e. to check for what should be done), but all possible inputs, included invalid inputs (i.e. to check for what should not do). And to single or individual data would have to add all combinations of input data or dependent sequences needed to test the respective functions. Hence, to test a system, one would have to produce a huge number of test cases.

This reflection shows that in general, exhaustive input testing is impossible in big systems. And from this statement are deduced two fundamental consequences:

a) the test of a complex system can not guarantee that it is fault-free; and
b) a fundamental consideration in complex system testing is one of economics.

Then, since exhausted testing is out of the question, the objective should be to maximise the yield on the testing investment (i.e. maximise the number of faults found by a finite number of test cases). Doing so will involve, among other things, being able to peer inside the system (specifications) and making certain reasonable, but not air-tight, assumptions about the system. This will form part of the test case design strategy.

Structural testing does look at the implementation details. In using this strategy, the tester derives test data from an examination of the system internal logic (often at the neglect of the specification).

Structural tests are inherently finites, one examines for example, specific internal unit functions or performances of HW, or specific characteristics of SW such us module interfacing, critical paths in the program, etc. Unlike the functional testing, it is assumed that structural testing can not make a complete test of the system. If one try to do an exhaustive structural test (to achieve a complete testing) to a complex system will find right away that the number of unique test cases tend to infinite. Moreover, a structural test in no way guarantees that a system matches its specification (due to misconceptions, missing parts, etc).

If one analysis these strategies from the involved persons point of view, in particular the designer and tester, will find that designer when act as a tester are by nature biased toward structural considerations while independent tester due to his ignorance of structure (no preconceptions) are bias-free and can better deal with functional tests.

It is noteworthy to mention the psychological problem associated with designers when testing. Most designer can not effectively test their own systems because they can not bring themselves to form the necessary mental attitude (a destructive frame of mind) after having had a constructive perspective during the design

NOTE: Beizer calls this a constructive schizophrenic attitude and makes an analogy with Dr. Jeckill and Mister Hide personality.

Another significant problem related to the designer, it is the fact that, the system may contain design faults due to misunderstandings of the specification and when testing it is likely that the designer will have the same misunderstanding.

In conclusion, neither functional nor structural tests prove to be absolutely useful strategies: both have limitations and both target different faults. What is proposed, it is to combine elements of both BB and WB testing to derive a reasonable, but not air-tight, testing strategy. The art of testing, in part, is in how the tester chooses between structural and functional tests.

5. Classification of test methods

Being our wish to give this first phase of the project a general character, the study has being extended beyond the traditional field of WB and BB strategies, as they are, software and some system testing. Thus, it is decided to include also the system behaviour testing in case of faults (injection testing), given its relevancy in safety related systems validation.

5.1. Fault injection

5.1.1. Introduction

EN 954 standard includes the fault injection technique as a method to validate complex electronic systems.

NOTE: ***Fault injection*** *is defined in [Arlat 90] as the dependability validation technique that is based on the realisation of controlled experiments where the observation of the system behaviour in presence of faults, is explicitly induced by the deliberate introduction (injection) of faults into the system.*

In the d**esign process of a system** (Refer report, fig. 1), the injection technique is mainly used for **fault removal** and **fault forecasting** [Laprie 92] [Gil 96] and, as it will be shown later, this technique can be applied either as a white box or a black box approach.

Fault removal involves a system verification to reduce the outcomes produced by possible faults introduced in the design, development and prototype construction stages and also to identify the proper actions in order to improve the design. Fault injection tries to determine whether the response of the system matches with its specifications, in presence of a defined range of faults. Normally, faults are injected in perfectly chosen system states and points, previously determined by an initial system analysis. Tester knows the design in depth and so it designs the test cases (type of faults, test points, injection time and state, etc.) based on a structural criteria and usually in a deterministic way. In this case, fault injection is being used as a **white box or structural strategy.**

Fault forecasting estimates the influence of the occurrence, presence and consequences of faults in safety systems **during its operational phase**. Then, a forecast based on the massive and random fault injection may be **useful to determine the designed system category**.

Moreover, we will be able to achieve results such as the system fulfils all the safety functions on a percentage basis of injected faults. Fault injection tests described before use a **black box strategy**, since the required knowledge of the inner parts of a system is minimum.

In practise, frequently fault removal and fault forecasting are not used separately, but one is followed by the other. For instance, after rejecting a system by fault forecasting testing, several fault removal tests should be applied. These new tests provide actions that will help the designer to improve the system. Then, it will be applied another fault forecasting test, and so on.

The methods introduced here below are not subdivided a priori in WB and BB, because in this field (injection testing) testing methods or techniques have not a defined borderline. For instance, it is possible to inject all combination of short-circuits in pairs systematically at the pins of all ICs or only to a group randomly selected in a board (BB approach). Or in the contrary, just to apply only a few specific short-circuits to some selected components in specific operational states and times (WB approach).

5.1.2. Fault injection techniques

There exist several techniques that allow to inject faults at different abstraction levels of the system, offering then the opportunity to verify the system behaviour in presence of faults from the very early stages of the design process.

In [Arlat 90], [Gil 92], [Jenn 94], [Iyer 95], [Clark 95], [Pradhan 96], [Vigneron 97] are shown different states of the art on fault injection. From these references we can deduce that, depending on the abstraction level of the system to be validated, fault injection techniques can be grouped as follows:

- Injection in a simulation system model (fault injection based on simulation).
- Injection either in a system prototype or in the final system. In turn, this group can be divided in:
    - Fault injection at physical level (physical fault injection).
    - Software implemented fault injection.

Fault injection based on simulation techniques is applied into simulation system models. With this technique it is not necessary to build a physical system and it can be applied at the very early development stages, that has a great importance in the premature design decisions about the system structure.

On **physical fault injection techniques**, faults are induced inside the system prototype, over their own hardware (in a transistor, a logic gate, a bit of a register, etc.), disturbing their physical or electrical properties.

Software implemented fault injection techniques (SWIFI) perform a hardware faults emulation (both internal faults and external faults). For example, the program can be corrupted or the internal registers modified through a particular fault injection routine.

Figure 5 shows a summary of fault injection techniques, which will be explained in depth in the following paragraphs.

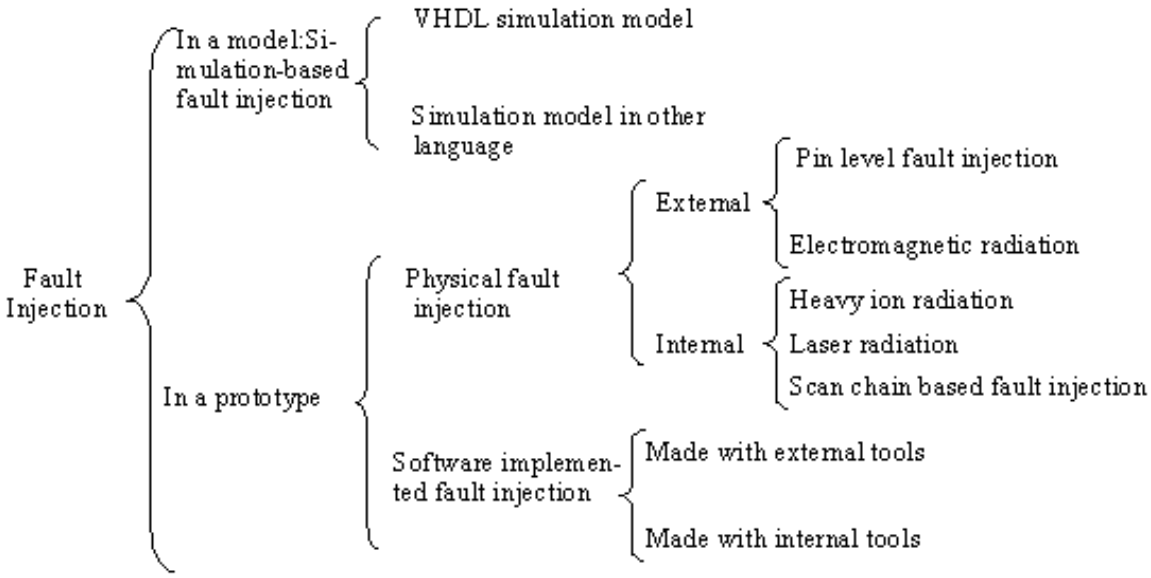


Figure 5 - Summary of the different fault injection techniques

5.1.2.1.Fault injection based on simulation

Objectives

The goal of fault injection based on simulation is to detect, mostly in an early phase of the design process, whether the behaviour of a system under development in presence of faults matches with its design specifications. The system can be an integrated circuit, a functional unit or subsystem or the whole system. The first requirement to achieve the goal is to perform a system model (based normally on VHDL, a hardware description language). After that, it will be possible to simulate its behaviour in presence of injected faults. The simulation is based normally on EDA tools, and the fault injection on special tools.

Moreover, this technique is currently also applied in fault forecasting. That is possible due to the existence of special tools for the massive and random fault injection in simulation models, as described below.

Description of the technique

Fault injection based on simulation can be used at different abstraction levels in complex electronic systems:

- integrated circuits level,
- processor level,
- full computer level,
- top level of a distributed computer system.

Typically, fault simulation has been applied in integrated circuits manufacturing. However, this kind of fault simulation is not related to fault injection, but to generation of test patterns, in order to detect faults in the production of VLSI circuits. In this case, fault simulation used for validating test patterns, works with logic simulation to verify design functions.

Table 2 [Vigneron 97] shows a summary of the commercial logical simulators associated with EDA (Electronic Design Automation) tools. Some of these logical simulators have a fault simulator associated. These simulators can be used in a manual or semiautomatic way, to verify several safety system mechanisms. Nevertheless, they are always oriented to systematic faults testing in integrated circuits or in full printed circuit boards. Currently, due to the wide diffusion of the VHDL hardware description language (that allows the system simulation in all the abstraction levels), most of fault injection systems based on simulation use the VHDL language and their associated simulators.

A desirable goal in the design process of safety systems is to tightly couple both design and fault injection-based verification tasks. Then, it will be possible to implement incremental steps during the design process. That derives in the optimisation of the design choices and the corrective actions. The development of integrated and coherent design methodology for safety systems will be reachable if we take into account the emerging hardware description languages. In this context, the VHDL language has been recognised as a very useful, since it presents the following interesting features:

- possibility of describing either the structure (white box view) or the system behaviour (black box view) in only one syntactic element;
- wide diffusion in the current digital design;
- inherent capability to perform hierarchical descriptions at different abstraction levels [Dewey 92] [Aylor 92];
- good performance in the modelling of digital systems at high level.

Table 2: Commercial fault simulators

Distributor	**Product**	**Simulator**	**Abstraction Level**	**Language**
Cadence	Verilog XL	Verilog XL	Behavioural, gates	Verilog
Ikos	Voyager	Voyager VS Voyager CS Voyager CSX	Behavioural Behavioural, gates Behavioural, gates with accelerator	VHDL
Mentor	Idea Station	Quicksim	Gates	BLM

Graphics	Entry Station Quick HDL Pro	Quick HDL Quick HDL Quicksim	Behavioural, gates Behavioural, gates Gates	VHDL, Verilog VHDL, Verilog BLM
Simucad (Intsys)	Silos	Silos	Behavioural, gates, transistors	Verilog, Verilog - A
Summit (Backstreet Intsys*)	Visual HDL	Visual HDL Visual Verilog	System, behavioural, gates System, behavioural, gates	VHDL Verilog
Veda	Vulcan	Vulcan	Behavioural, gates	VHDL
Viewlogic	Fusion HDL	Speedwave VCS Viewsim	Behavioural Behavioural, gates Gates	VHDL Verilog
Zycad	Paradigm	Paradigm VIP	Gates	VHDL, Verilog
Microsim (ALS Design)	PSPICE	PSPICE	Gates, transistors	ABM

* INTSYS distributes the fault simulator

There are two main fault injection techniques based on VHDL [Arlat 93] depending whether the code is or not modified:

a) Modifying the code:

- Injection by means of additional components called **saboteurs.** A saboteur element is a VHDL component that changes the value, or the temporary features, of one or more signals when it is activated. It remains inactive during the normal system operation and is activated only to inject a fault. A **series saboteur** breaks the connection between a driver (output) and its corresponding receiver (input) and modifies the value of the receiver. It could also modify a set of drivers and their corresponding set of receivers. A **parallel saboteur** is implemented easily adding an additional driver to a set of drivers connected in parallel. The resolution function of the VHDL, that permits the selection of one signal between several parallel signals, is useful to implement this type of injection.

- Injection by means of special components called **mutants** that have a function in the circuit. A mutant is a component that replaces another component. When it is inactive, it performs the function that the original component does. However, when it is activated, its behaviour is an imitation of the modelled faulty component. VHDL configuration mechanism is useful for this type of mutation since it permits the assignment to an entity of an architecture among several possibilities (there will have one fault free and several faulty architectures).

  There are several forms of performing the mutation:

  - modifying structural descriptions by means of the replacement of sub-components. For example, a NAND gate may be replaced by a NOR gate;

- modifying manually behavioural descriptions to obtain full and detailed fault models;
- modifying automatically instructions in behavioural models. For example, generating wrong operators or changing identifiers of some variables. This approximation is similar to the mutation techniques used in software validation.

a) Without modifying the VHDL code. Then, the injection is performed using simulator commands allocated with the VHDL compiler.

As an example of VHDL-based fault injection environment, figure 6 shows the block diagram of the tool developed by the research group GSTF of the DISCA Department (Technical University of Valencia, Spain) [Gil 97][Gil 98].Three main blocks can be distinguished in it:

- Experiments' set up block. With the help of a program, it is written a configuration file containing all the faults and points of injection for the subsequent injection campaign.
- Simulation block. An injection macro generator writes a file, using the configuration file, with all the macros that will carry out the injection.
- Readouts block. A data analysis program determines the system behaviour in presence of faults. This task is carried out comparing the results of each injection with the respective results without faults.

Other examples of injectors based in VHDL are the MEFISTO-L [Boue 98] of the LAAS, in Toulouse (France) and the MEFISTO-C, [Folkesson 98] of the Chalmers University of Technology, in Göteborg (Sweden).

Generally, the kind of injected faults are stuck at, open line, delay, bit flip, short circuit and bridge between connections. Respect to the time parameter, the faults can be permanent, transient or intermittent.

Advantages

- Possibility of injecting faults before the prototype has been built,
- arbitrary reachability, just depending of the detail level into the model,
- arbitrary controllability, depending of the model level detail,
- arbitrary observability, depending of the model level detail,
- the faults are easily reproducible,
- evidently, no component can be destroyed with the fault injection process, or due to disturbances caused by the injector in the system under test,
- easy application, since it is only needed a computer with the specific compiler language, the simulator and the injection tool,

- low cost of the necessary infrastructure (if there is a disposable computer and simulation tools),
- the latency times in the error detection or recuperation can be easily measured,

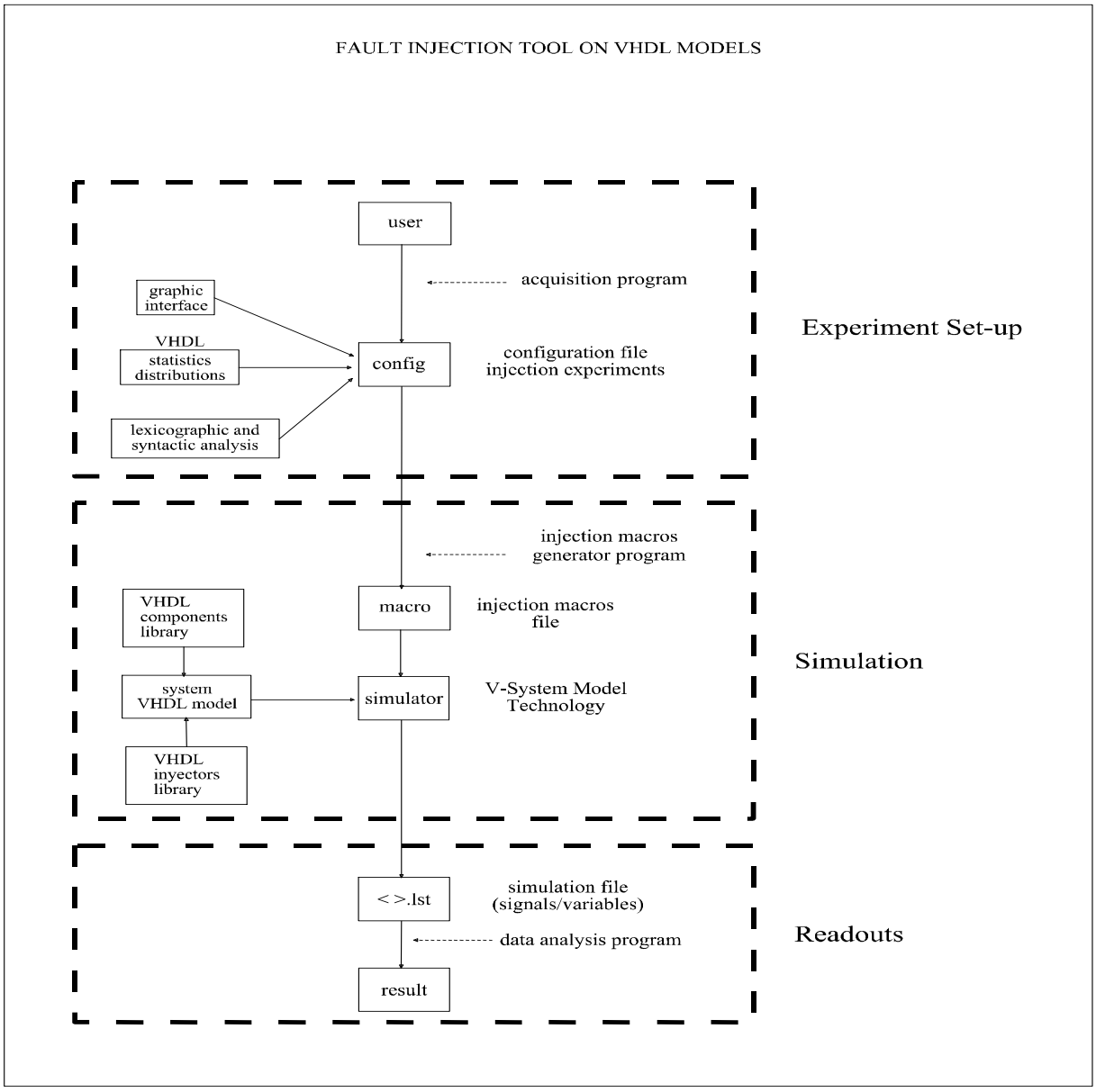


Figure 6 - Block diagram of a fault injector for VHDL models

Disadvantages

High simulation times, causing that a fault injection process spends a lot of time with a medium speed computer (in [Folkesson 98] the injection of 1000 faults spends about 6 days for a UNIX workstation with a clock frequency of 70 MHz)

- Accuracy of the results depends on the goodness of the model used. Obviously, the greater accuracy (high level of detail of the structural model), the greater simulation time.

- No real time faults injection possibility in a prototype.

This technique is implemented in some research laboratories (see table 3) and there are not any commercial tool.

5.1.2.2.Physical fault injection

Objectives

As any other fault injection technique, it tries to detect any difference between the specified and observed behaviour in presence of faults In this case, the injection is carried out on a real system (prototype). The advantage is that the results are closer to the reality than in fault injection based on simulation. All physical fault injection techniques described below perform a physical fault injection, either at integrated circuits terminals, or signal lines between components or even inside components.

The physical injection technique is used either during partial verifications of the design, in order to eliminate faults, or on the final process validation, to determine the conformity of the prototype with the category specifications. The last one is referred to the behaviour in presence of random faults (fault forecasting).

Description of the technique

There are two injection techniques involved at this level:

- the **external physical injection**,
- the **internal physical injection**.

With external physical injection, faults are injected outside the system to validate. For example, at the pins of an integrated circuit.

With internal physical injection, faults are injected inside the system to validate. For example, by means of a laser beam, heavy ion radiation, or with special mechanisms integrated inside the hardware of the system (fault injection based on scan chains).

5.1.2.2.1.External physical fault injection

External physical fault injection is performed mainly at pin level on integrated circuits (pin level fault injection) [Arlat 90], [Gil 92], or at circuit level by electromagnetic disturbances [Damm 88], [Karlsson 95]:

a) **Physical fault injection at pin level** uses a special fault injection tool to modify logical values at the pins of integrated circuits inside the system to validate. Two fault injection techniques are included at this level:

- **Forcing** technique: The fault is injected directly into an integrated circuit terminal, connector, etc, without any part disconnection. The fault injector probe forces a low or high logical level at the selected points.
- **Insertion** technique: A special device replaces a part of the circuit previously removed from its support (socket for an integrated circuit, connector for a bus, etc.). That device injects the faults. The connection between two circuits is cut off before injecting the fault. Thus, the injection is performed on the side that remains at high impedance (in other words, this side is an input). Because of there is not any signal forcing, there is not any danger of damage in the injected component.

Figure 7, shows an example with both fault injection techniques. FFIM is a forcing fault injection module and IFIM an insertion fault injection module.

Referring the time parameters, faults can be permanent, transient or intermittent.

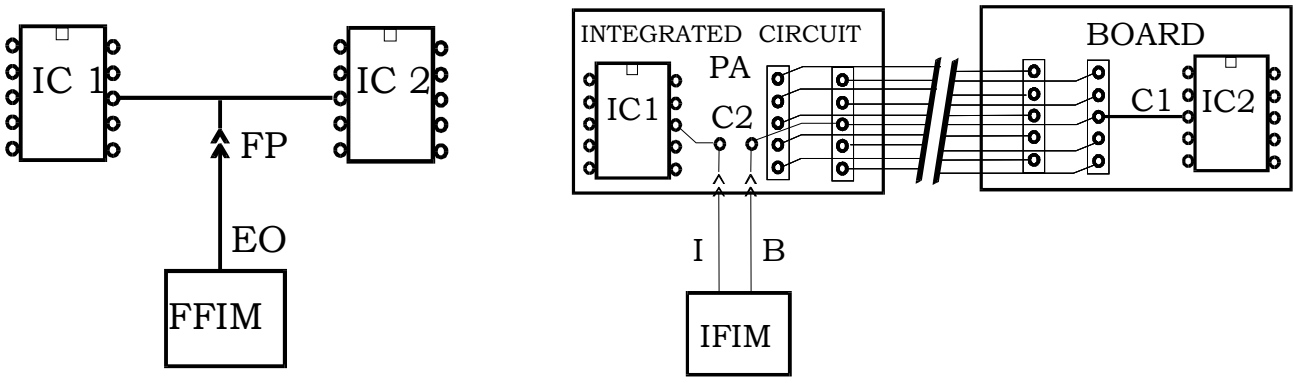


Figure 7 - Physical injectors of forcing (FFIM) and insertion (IFIM)

Currently, there are several pin level injection tools; among them it is worth to mention the following:

- The MESSALINE injector from LAAS [Arlat 90], [Arlat 90a], in Toulouse (France), which can inject multiple faults, at any time, allowing also the automation of the injection experiments. This fault injector implements both forcing and insertion injection techniques.
- The RIFLE injector [Madeira 94], from the University of Coimbra (Portugal), includes a trace memory circuit which allows to speed up of the injection experiments. This injector uses only the insertion technique.
- FAC (DEFOR) [Vigneron 97]. Forcing injector with 40 pins probe, composed by power MOS transistors that have a short-circuit impedance of some ohms, reducing the possibility of circuit damage.
- The injector from the Technical University of Valencia (Spain), that has two versions. The first version [Gil 93] allows both forcing and insertion techniques. The second one [Gil 97], named AFIT (Advanced Fault Injection Tool) allows high-speed

fault injection (the minimum length of time of the injected fault is 25 ns). This fault injector only implements the forcing technique because nowadays is very difficult to implement the insertion technique due to the high frequencies and packages (surface mount) of the integrated circuits

Figure 8 shows the block diagram of the AFIT fault injector. It is divided in the following blocks:

- PC bus interface block. The injector is plugged into a PC through its ISA bus interface.
- Timing block, which consists of several programmable counters that generate the different clock signals for the other modules.
- Synchronisation and triggering, which starts injection when a predefined system state matches a triggering word and a subsequent programmable delay.
- FTS system activation, which is used to initiate the system to be validated with a set of convenient inputs. This task is carried out before injecting the faults.
- High speed forcing injectors, which can inject faults of 25 ns resolution.
- Events reading, which consists of a transitional logical analyser that is able to read from the experiment both fast and slow events. When an injection sequence finishes, collected data is recorded in the PC disc.

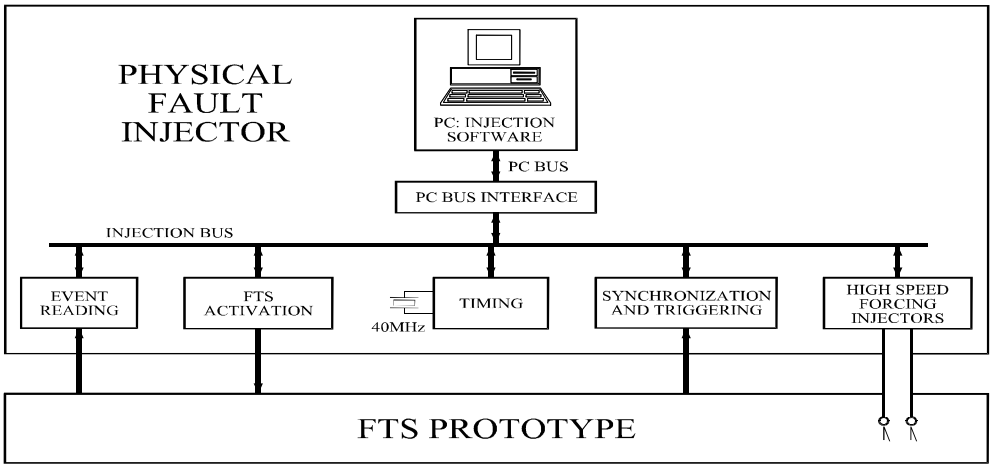


Figure 8 - Block diagram of the AFIT fault injector

These are some of the existing European injectors. For more information, especially about American injectors, [Iyer 95], [Pradhan 96] and [Vigneron 97] can be consulted.

Concerning the types of faults injected, the following classification can be done:

- "stuck at" faults at 0 or 1 logical value. This is the fault model most frequently used, because it is the easiest one to implement and it covers a great number of faults at transistor level.
- Open circuit faults, short circuit and bridge between connections, that are used to cover the faults at transistor level that are not covered by the "stuck at" model.
- Single (single pin alteration) or multiple faults (alteration of some pins simultaneously).

Advantages of physical fault injection at pin level

- Possibility of injecting faults in the physical system, so the results are real.
- Real time injection.
- Testing time is not very long whenever an automatic tool is used (injection control, outputs registration and analysis of the results).
- The faults are easily reproducible.
- In the case of pin level forcing technique, easy connection to the prototype.
- Some tools make an analysis of the pins logic activity to discard the test cases that are not effective and thus reduce testing time.
- Good space controllability (faults are injected in perfectly specified places). Less time controllability (injection time can be more difficult to synchronise with the prototype in high frequency systems).
- It can easily measure latency times in the error detection or recuperation.

Disadvantages of physical fault injection at pin level

- Restricted reachability, especially in the new generations of VLSI integrated circuits, which have a very high complexity/number of pins ratio.
- Low observability. It depends on the integrated circuits complexity and if its design is oriented to testability.
- There is a low probability of a component to be damaged in the process of fault injection using the forcing technique. Using the insertion technique it is almost impossible that a component is damaged.
- Difficulties when preparing the injection experiment, by the difficult wiring between injector and prototype, particularly when we use surface mount components (SMD). With components in these packages, the insertion technique is practically impossible to carry out.
- High cost of the necessary infrastructure (it is necessary a special equipment for the fault injection).
- Disturbances of the injector on the prototype, especially in systems with high clock frequencies. These disturbances can impede the correct operation of the prototype.

It is a widely used and experimented technique, which in many cases is still applied manually. Existing tools are privates and in most cases belong to universities, companies and test houses (see table 3). Currently, there are no commercial tools available.

a) **Physical fault injection with electromagnetic interference (EMI** [Damm 88], [Karlsson 95]. It uses a burst generator to interfere the component or system under test. The test equipment generates pulses according with the standard EN 61000-4-4.. The disturbances are injected on the printed circuit boards of the system, in two different ways (see figure 9):

- With the printed circuit board located between two conducting plates connected to the burst generator (left part of the figure).
- With the aid of a special tool that permits the exposition of a smaller area within the printed circuit to the electromagnetic interference (right part of the figure).

The injected type of faults try to model those perturbations produced by some external electromagnetic interferences.

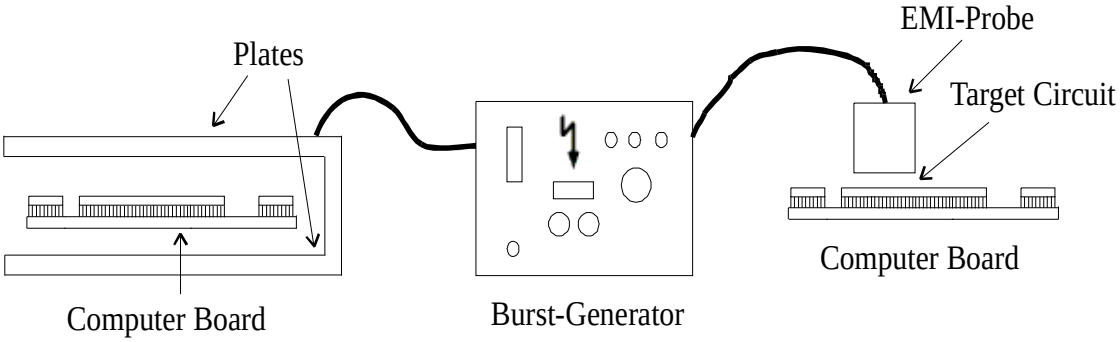


Figure 9 - Experiment of fault injection based on EMI radiation

Advantages of physical fault injection with electromagnetic interference

- Possibility of injecting faults in the real system, so the results are real.
- Real time injection.
- The duration of an injection campaign will not be very high whenever an automatic tool is used. Nevertheless, if it is necesary to know whether or not there have been errors after a burst injection, it would have to compare the behaviour of the system in presence of faults with other system without faults ("golden unit"), what slows down the process.
- Easy to apply on the prototype, because there is no physical contact between the injector and the system.
- Low probability of damaging the prototype.

Disadvantages of physical fault injection with electromagnetic interference

- Restricted reachability, since disturbances act at pin level, as the previous case.
- Low space and time controllability.
- Low observability. It depends on the integrated circuits complexity and if the design is oriented to testability. It must be taken into account that error syndromes will be detected with the comparison of both systems, with and without faults.
- The disturbances can provoke multiple faults, for example, induced through power and ground lines. The activation of these faults is very difficult to detect.
- The experiments are very difficult to reproduce.
- High cost of infrastructure (it is necessary a special equipment for the fault injection).
- Latency times difficult to measure. It could be done comparing the execution of a prototype with faults with other without faults ("golden unit"), but it is difficult that the radiation does not affect the "golden unit" too.

This technique has been implemented in some research laboratories (see table 4).

5.1.2.2.2.Internal physical fault injection

Objective

As always, this technique tries to summit the system under test to a number of specific kind of faults to verify if the behaviour of the system meets the specifications. In this case, faults are injected into the components at transistor level or internal parts.

Internal physical fault injection is currently accomplished in three ways:

- by means of heavy ion radiation,
- laser radiation, or
- through special mechanisms integrated inside the hardware of the system, connected to its periphery through a test access port (TAP), like the JTAG port of the boundary scan. This last technique is named scan chain based fault injection.

Description of the techniques

a) **Heavy ion radiation injection** technique [Gunneflo 89] and [Karlsson 95]. It involves the generation of transient bit flip errors inside the irradiated integrated circuits. It uses a Californium 252 source. These faults simulate transient faults in the transistors originated by external sources, as electrical, electromagnetic or radioactive interferences. The irradiation must be accomplished within a vacuum chamber because molecules contained on the air and other materials can stop the heavy ions. It is necessary to remove also the package from the circuit (or circuits) to be irradiated.

This technique has been used to evaluate the internal error detection mechanisms of a MC6809E microprocessor [Gunneflo 89], as well as for validating the MARS architecture [Karlsson 95].

The foremost feature in this kind of injection is that the faults can be injected, within the VLSI circuits, in places impossible to reach by other injection techniques, such as pin level fault injection or software implemented fault injection.

Advantages of the heavy ion radiation injection

- Possibility of injecting faults on the real system, so the results are real.
- Real time injection.
- The duration of the injection campaign will not be very high whenever an automatic tool is used. Nevertheless, if it is necessary to know whether or not there have been errors after the ion radiation, it would have to compare the system behaviour in presence of faults with other system without faults ("golden unit"), what slows down the process.
- Good reachability, since the faults can be injected inside the VLSI chips at great depth.
- Low interference in the prototype chips that are not ion irradiated.

Disadvantages of the heavy ion radiation injection

- Difficulty in preparing the experiments. The integrated circuits package must be removed, a vacuum chamber is needed, etc.
- Components are destroyed during the fault injection process (the package must be removed).
- Experiments are so difficult to reproduce because the fault (or faults) is injected in unknown positions.
- Low space controllability (the place of injection is unknown) and no time controllability (the injection time can not be controlled since the generation of heavy ions is a random process).
- Low observability. It depends on the integrated circuit complexity. It must be taken into account that error syndromes will be detected with the comparison of both systems, with and without faults (“golden unit”).

This technique has been implemented in one research laboratory (see table 4). There are no commercial tools available.

b) Fault injection based on laser radiation technique [Sampson 98]. It is a better alternative than heavy ion radiation since it can inject soft faults (transient) in a very controlled points inside of an integrated circuit without package. The injection is carried out pointing a laser

beam over the chip, which is more accurate and easy to implement than heavy ion radiation. The physical principle to generate a fault, using laser radiation, is based on that the laser energy generates electron-hole pairs in the semiconductor material, just as the SEU (Single Event Upset) pairs generated by high energy particles.

As in heavy ion radiation, the fault can be injected with the desired depth, but this method has the advantage that the fault is better reproducible, since both the direction of the laser beam is very accurate and the chip site, where the fault is injected, is also delimited. Figure 10 shows an experiment presented in [Sampson 98]. The experiment uses a fixed laser beam and a mobile 6 degrees freedom table (X, Y and Z-axis, table turn and tilt axis in X and Y). The laser beam accuracy is 0.1 μm. According to this article, fault injection could be possible over a full printed circuit board, although the entire integrated circuit packages must be removed.

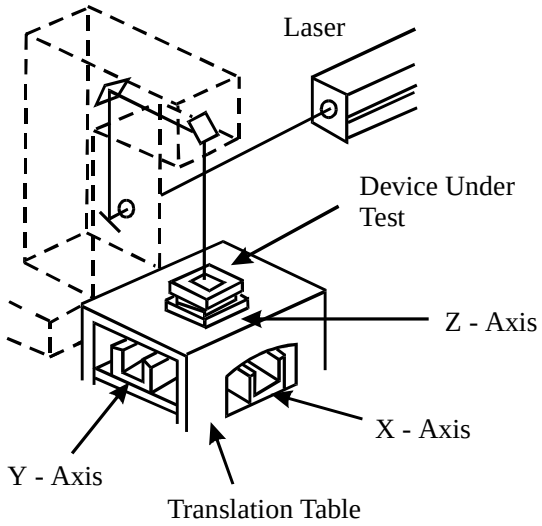


Figure 10 - Experiment of fault injection based on laser radiation

Advantages of the fault injection based on laser radiation

- Possibility of injecting faults in the real system, so the results are real.
- Real time injection.
- The duration of the injection campaign is not very high because an automatic tool is used. As in the previous case, if it is necesary to know whether or not there have been errors after the laser radiation, it would have to compare the behaviour of both systems with and without faults ("golden unit"), what slows down the process.
- Good reachability, since the faults can be injected inside the VLSI chips in depth.
- Better reproducibility than the case of heavy ion radiation, since the faults are injected in perfectly delimited points, due to the high precision of the laser beam.
- High space controllability (the injection point can be determined with precision).
- Low interference in the chips of prototype that are not injected.

Disadvantages of fault injection based on laser radiation

- Low observability. It depends on the integrated circuit complexity and if its design is oriented for testability. It must be taken into account that, as in the previous case, error syndromes will be detected with the comparison of both systems, with and without faults (“golden unit”).
- Low time controllability (if the injection tool is not synchronised with the prototype).
- Difficulties in preparing the experiments. The integrated circuits package must be removed, also we need a special table to move the prototype, etc.

- Components are destroyed during fault injection process (the package must be removed).
- Not multiple injection allowed.

This technique has been implemented in some research laboratories (see table 4). There are no commercial tools available.

c) **Scan chain based fault injection technique.** It involves the use of additional circuitry inside the VLSI chips to ease the fault injection. The best way to insert test patterns within the integrated circuits is using shift register chains of the test access port (TAP), like the JTAG of the standard boundary scan, or may be with other special TAP. This port is also useful to read the injection results (error syndromes), and other internal signals from the integrated circuit. Injected faults model transient and permanent faults in the internal flip flops due to physical faults (external and internal).

A first approach for this technique could be the experiments such as those described in [Vigneron 97]. There, faults are injected in ASICS over functional test terminals (used for fault injection purposes), or over special terminals added to ease the fault injection.

A meaningful example is shown in [Folkesson 97], with FIMBUL (Fault Injection and Monitoring using Built in Logic), a tool from Chalmers University of Technology, of Göteborg (Sweden). In this paper, jointly with MEFISTO, FIMBUL provides a validation of Thor, a microprocessor specially designed for fault injection.

FIMBUL generates an injection control file. It contains both the microprocessor load test, and break points in the program where the faults will be injected. When a break point is detected, FIMBUL halts the process and injects the faults in the internal registers of the microprocessor by changing predetermined bits of them (fault model of bit flip). This step is accomplished first by reading the values of the internal register chain (using the TAP) and then, changing some of their bits. The modified values are loaded again in the microprocessor via the TAP, and the process runs again (now in presence of faults). The process continues until an error occurs or until a timer stops. In both cases, the values of the internal register chain will be read again using the TAP in order to determine the error syndrome.

This technique allows fault injection inside a chip in any depth. Just one condition is required, that the injection site belongs to the scan chain. In [Folkesson 98] can be seen a comparative study between this technique and VHDL-based fault injection, using in both cases the Thor microprocessor as example.

Advantages of chain based fault injection

- Possibility of injecting faults in the real system, so the results are real.
- Good reachability since faults can be injected in depth inside the VLSI chips (depends on the number of registers connected with the scan chain).
- Good observability, since the error syndromes will be read through the scan chain.

- High space controllability (precision in determining the injection place) and time controllability (precision in determining injection time).
- Good reproducibility of the experiments.
- No possible damage into the prototype.
- There are no interferences in the prototype.

Disadvantages of chain based fault injection

- No real-time injection.
- Experiments need a long time to conclude.
- Complexity in application. A special design of the integrated circuits is needed, with internal scan chains.

This technique has been implemented in some research laboratories (see table 3). There are not any commercial tools.

5.1.2.3.Software implemented fault injection

Objective

As above, software implemented fault injection tries to verify the behaviour in presence of faults of a physical prototype. In this case the technique applied consist of modifying the program according with some algorithms or criteria in such a way that the modifications model internal or external faults (HW faults, errors induced by external interferences, or even program design and implementation errors [Iyer 95]. That is, it is assumed that the consequences of faults often translate into changes of program or data memory contents, erroneous addresses of memory, etc.

Unlike other techniques described before, software implemented fault injection is only applicable on advanced prototypes.

Description of the technique

This technique is based on several practical methods of injection, such as the data modification in memory, or the mutation of the application software. A first interesting feature of this technique is that it can be applied to physical models as well as to information models.

There are two types of injectors. Those which use an **external tool** for the corruption of the programs (such an emulator), and those which are integrated inside the own system (**internal tools**). These are implemented both as agents that inject faults (injector processes) and agents that observe the response of the system (observer processes).

In general, the injection techniques that emulate hardware faults by software produce a modification of memory contents. Within these kind of techniques we can distinguish two

groups: those which consider the program memory as a table of independent data, and those which interpret the contents and takes into account the semantics.

In [Chillarege 89] is described an example of the first category, where randomly selected memory pages are filled with specific values.

The second category involves the following techniques:

- The technique used in **DEFI** [Gérardin 86] and **DITA** fault injection tools, is based on the modification of memory values, replacing the original contents by other erroneous, in order to emulate the system behaviour in presence of physical faults. This modification is accomplished with an external tool independent of the system to be validated.

- The **FERRARI** tool [Kanawati 92] is the first example of an injection tool integrated within the system to be validated. This tool offers a set of mechanisms that allows the logic emulation of transient and permanent physical faults into memory, external buses of the processor, decoding logic of the instruction code, control unit, internal registers and the arithmetic and logic unit. At a higher level, these hardware faults produce errors in the address and data buses. The objective of FERRARI is the injection of these errors via software. The technique used is based on the deviation of the control flow by means of a logical interruption. The interruption routine proceeds to modify the executed instruction into the main program, the value of the instruction pointer, the condition flags, the value of a memory data, etc, according to the hardware fault to be emulated.

- The **SFI** tool [Rosemberg 93] allows the permanent and transient faults emulation at memory level of certain processor functional units (add and product units) and certain communication network in a distributed real time systems (HARTS). Every fault calls to an integrated specific mechanism. This mechanism is called while runs the compilation in case of fault injection into memory, in the communication protocol in case of fault injection into a communication network, and in the assembly code in case of the fault injection into processor functional units.

- **FIAT** tool [Segall 88] [Barton 90], allows fault injection either into the code or the application data, and either into the sequence of tasks (delay and abnormal completion of them), or the messages exchanged between them (corruption of its content, loss and delay).

- The approaches of Echtle [Echtle 91] and Avresky [Avresky 92] use fault injection by logic emulation technique to test fault tolerant protocols. They consider in this case test vectors that are obtained with deterministic methods (using an heuristic method in the first case). These faults correspond to the message disturbances exchanged with the needs of the protocol.

These kind of fault injectors contrast with that adopted by IBM in the validation of IBM 3090 systems [Merenda 92] where certain processor parts such as the general purpose register can be disturbed with instruction sequences interpreted by the service processor.

Advantages of software implemented fault injection

- Possibility of injecting faults in the real system, so the results are real.
- Reachability depending on the accessible variables from the injector, but in any case better than pin level physical injection.
- Good observability, since modern processors have internal exception-based debugging that gives the possibility to detect error syndromes in depth.
- High space controllability (the injection place can be determined with precision) and time controllability (the injection time can be selected with precision).
- Good reproducibility of experiments, since the faults are injected at perfectly delimited points.
- There is no probability of damaging the prototype.
- There is no interference over the prototype.
- Easy to apply on the prototype, although if the injection is made with internal tools, the software has to be modified.

Disadvantages of software implemented fault injection

- Injection is not possible at real time: injection processes (agents) interfere with the normal system processes.
- Each experiment spends a long time to conclude.
- Model representativity is questioned.
- Difficult to apply.

This technique is implemented in some research laboratories (see table 4). There is no commercial tool available.

5.1.3. Summary of the main features of fault injection techniques

Table 3 summarises the main features of all injection techniques presented in this clause.

Table 3: Summary of the main features of fault injection techniques

TECHNIQUE	Type of faults injected	Testability and accessibility	Applicability	Cost – effectiveness	White – Box Test	Black – Box Test
Based on simulation models.	Model of physical faults Any level of abstraction (it depends on the model) Very high reproducibility	High accessibility High observability High space controllability High time controllability	Low difficulty of application No damage to the system No disturbances Impossible the real time injection	Low infrastructure cost Very high time cost	Yes	Yes
Pin level.	Real physical faults Pin level injection in prototype High reproducibility	Low/media accessibility Low/media observability High space controllability Media time controllability	Media difficulty of application Low damage to the system High disturbances Real time injection is possible	High infrastructure cost Low timecost	Yes	Yes
EMI radiation.	External temporary physical faults Pin level injection in prototype Low reproducibility	Low/media accessibility Low/media observability Low space controllability Low time controllability	Low/media difficulty of application Low damage to the system High disturbances Real time injection is possible	High infrastructure cost Medium time cost	No	Yes
Heavy ion radiation	Temporary physical faults (bit flips) Internal level injection in prototype Low reproducibility	High accessibility Low/media observability Low space controllability No time controllability	High difficulty of application High damage to the system Low disturbances Real time injection is possible	Very high infrastructure cost Medium time cost	No	Yes
Laser radiation.	Temporary physical faults Internal level injection in prototype Medium reproducibility	High accessibility Low/media observability High space controllability Low time controllability	High difficulty of application High damage to the system Low disturbances Real time injection is possible	Very high infrastructure cost Medium time cost	No	Yes
Based on scan chains.	Physical faults Internal level injection in prototype High reproducibility	High accessibility High observability High space controllability High time controllability	High difficulty of application No damage to the system No disturbances Impossible the real time injection	High infrastructure cost High time cost	Yes	Yes
SW implemented.	Physical faults Internal level injection in prototype High reproducibility	Media/High accessibility High observability High space controllability High time controllability	Low difficulty of application. No damage to the system No disturbances Impossible the real time injection	Low infrastructure cost High time cost	Yes	Yes

NOTES: ***Accessibility*** *(physical reachability): ability to reach possible fault locations (nodes) in a system.*

Controllability*: it can be considered with respect to both the space and time domain. The space domain corresponds to controlling where faults are injected, while the time domain corresponds to controlling when faults are injected.*

Observability*: ability to observe a circuit or system (internal logic) from its primary outputs. This attribute is of great relevance in order to observe the syndrome produced by the errors due to injected faults.*

Reproducibility*: ability to reproduce results statistically for a given set-up and / or repeat individual fault injection exactly. Statistical reproducibility of results is an absolute requirement to ensure the credibility of fault injection experiments. The possibility to repeat experiments exactly, or at least with a very high degree of accuracy, is highly desirable, particularly when the aim of the experiments is to remove potential design/implementation faults in the fault tolerance mechanisms.*

5.1.4. Main fault injection tools summary

Table 4 shows some of the existing injection tools (the ones commented in this report and some others too). As far as we know most of them have not been commercialised. DEFI and DEFOR were commercialised for a time.

Table 4: Fault injection tools

TECHNIQUE	Tool	Developed by
Based on simulation models.	MEFISTO-L	LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes, at Toulouse (France)
	MEFISTO-C	Technical University of Göteborg; (Sweden)
	Without name	GSTF research group, DISCA, Technical University of Valencia (Spain)
Pin level	MESSALINE	LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes, en Toulouse (France)
	RIFLE	University of Coimbra (Portugal)
	AFIT	GSTF research group, DISCA, Technical University of Valencia (Spain)
	FAC	INRS / ESS (France)
	IDEE	INRS / MDP (France)
EMI radiation	Without name	Technical University of Vienna (Austria)
Heavy ion radiation	Without name	Technical University of Göteborg (Sweden)
Laser radiation	Without name	Florida University (USA)
Based on scan chains.	FIMBUL	Technical University of Vienna (Austria)
Software implemented	DEFI	INRS (France)
	DIAL	INRS (France)
	DITA	Technicatome (France)
	SOFI	GSTF research group, DISCA, Technical University of Valencia (Spain)
	FERRARI	Texas University (USA)
	FIAT	Carnegie Mellon University (USA)
	Without name	University of Essen (Germany)

5.1.5. Conclusions

After this review on injection tools and techniques, the following general conclusions can be extracted:

- The simulation fault technique seems to be very interesting because it allows the safety validation from the beginning, so it is possible to make corrections in the early stages of development. This technique also allows injecting great variety/different of types of fault models, with a good accessibility, observability, reproducibility and it is ease to use. Nevertheless, the difficulties to accomplish simulation models adjusted

to reality and, besides, the long delay simulation times, make this technique very time costly. Despite all this drawbacks, the fault simulation based on the standard language VHDL seems to have a future. In fact, there are a lot of fault injection tools based on VHDL simulation that are being developed at present.

- The pin level physical fault injection technique has been very well developed and systematised, with the advantage that it injects real faults. Furthermore, its reproducibility is good, the injection is at real time, and the required time on an injection campaign is not so high. This technique can be adequate in the safety validation of complex electronic systems that have already been built, specially if we don't have much information of them. Nevertheless, this technique has several problems, over all, the difficult application in current electronic systems, with great complexity/number pins ratio. This causes that accessibility and observability will be less every time. Other problems are the high clock frequencies of integrated circuits and its connection with the prototype to validate, since the packages of these circuits are surface mount. These packages are getting smaller and smaller, with the consequent difficulty to access at its pins.

- The physical external fault injection technique with electromagnetic interference may be useful too when we must inject faults into prototypes, specially if we don't have much information of them. It is because it can be directly applied to printed circuit boards. It allows, as the previous technique, the fault injection in real time systems. Nevertheless, poor reproducibility of the experiments causes that it seems less advisable that the previous techniques for the validation of the safety of complex electronic systems.

- The internal physical fault injection techniques based on heavy ion radiation and in laser fault injection would be a good practice in specialised research laboratories. But in other organisations, like certification laboratories, validation could be almost impossible. It is because there are so many requirements to carry out the experiments, such as the set-up, the removing of the integrated circuits package previously to the tests, and the arrangement of specific equipment.

- The technique of internal physical injection based on scan chains is now in experimental phase. It is an alternative to injection at pin level, since nowadays the complexity of the integrated circuits is very high and requires the design oriented to ease the test and the fault injection. This eases the accessibility, controllability and observability. The use of injection based on scan chains is not possible to validate real-time systems.

- Finally, all fault injection techniques implemented by software have the advantage of being useful to validate current high-speed systems. Thus, most laboratories, that were accomplishing physical injection at pin level in the past, have begun to implement this technique. Comparing with external physical fault injection, this technique is considered easier to implement. However, the addition into the system of fault injection processes.

5.2. Software Testing

5.2.1. Introduction

Software testing tries to detect systematic faults introduced during the design process.

When speaking about testing we refer to the test case design (the expectations or previsions, the detailed test procedures, and the results of the concrete tests), their execution and the analysis of results. The result of a successful test will be a series of anomalous or strange symptoms that will not correspond with the expected correct behaviour, and will show that the program has errors.

After observing that the program fails it is needed to find the error or misconception that produces it (i.e., its exact nature and its location) and then design and implement changes to correct it.

The process following testing, just described, is called debugging. Debugging differs from testing in their objectives (goals), methods and more important, in the psychological aspect.

In this report, debugging process has not been developed. For more information on this matter [Myers 79] is a good reference for beginners.

Unlike fault injection technique classification in software testing, one of the more relevant attributes for classifying the testing methods or criteria, and doing a subsequent selection of a test strategy, is the approach applied to the test: functional and structural. That is way, it is relatively easy to establish a subdivision of test methods under these attributes.

There are some publications or references in the literature concerning with the classification of SW faults and their distribution. That classification must be the base that supports any test strategy. For instance in [Beizer 90] it is presented a chapter “The Taxonomy of Bugs” which includes the statistical data of table 5 taken from many different sources.

The absolute frequency of faults, in programmes written by good programmers, is approximately 2.4 faults per thousand source statements. This rate includes mostly integration testing and system testing, carried out by an independent tester after thorough component testing by the programmers. The rate becomes 1 % – 3 % when considered all the faults, included those discovered by the programmer during self-testing and inspection. The author warns that absolute frequency should not be taken seriously as a reference given the variation in the sources. However, he recommends strongly the relative frequency of various fault types as a guide to selecting effective testing strategies.

In short, it is necessary to adopt a fault classification similar to above, personalised (well understood by the programmer or tester) and adapted to the particular characteristics of the product, and keep it updated.

Table 5: Fault statistics in programmes

Fault statistics Size of sample – 6.877.000 statements (comments included) Faults per 1000 statements – 2.4	
Type or category of fault	**Relative frequency (%)**
Requirements	8.1
Features & functionality	16.2
Structural	25.2
Data	22.4
Implementation & coding	9.9
Integration	9.0
System, SW architecture	1.7
Test definition & execution	2.8
Other, unspecified	4.7

5.2.2. White Box Tests

White Box Tests are actually a set of different criteria which use a structural model of the program for the selection of test inputs.

The adoption of a program model focused on a detailed internal view makes this strategy be preferably applicable to small size components, increasing rapidly complexity with size.

To represent the different structural features of a program usually are employed the following notations:

- Control flow graphs;
- Data flow graphs, often it is used a control flow graph adapted to represent the operations on data or variables, and;
- Call graphs.

Under these notations are grouped the main families of existing test case design criteria, related to structural modelling. That is to say, the criteria based only on the control flow graph, those which take into account control and data flow, and those criteria more adapted to integration testing.

Using structural criteria, it is intended to design test cases that exercise at least once “all” structural elements of a program (if not all, at least the most fault prone) and so ensure selectively its validity. For instance, execution of all paths related to functions, logic decisions, invoking both the true and false outcomes, loops, internal data structures, etc.

Structural testing often is also called path testing because when structural test techniques are applied, different program paths are executed with the objective of finding determined classes of structural errors.

Some reasons that justify the structural tests are:

- The certainty, that the probability of a determined path being executed is inversely proportional to the amount of logic errors and incorrect suppositions (the normal processing tends to be more understandable, while processing “special cases” tends to be chaotic). During black box testing it will be difficult to find some errors because surely we will not exercise the erroneous parts of the code.
- The erroneous assumptions about which are the normal or more frequent paths of the program (the program's logic flow sometimes is not intuitive).
- Typographical errors are introduced randomly (It has the same probability that there exist a typographical error in a dark path as in a principal path).
- The assumption of, complete functional testing is unfeasible (black box test), and it needs to be complemented with some structural tests in order to achieve an acceptable test strategy; etc.

According to [Beizer 90], faults lurk in corners and congregate at boundaries, being easier to discover them with white box tests.

In short, structural tests will be mostly applied, when developing new programs, by programmers during the unit and integration testing and will require a detailed knowledge of program structure. Rarely they will be used during system testing. Moreover, they turn out more useful in decision based algorithms than data centred programs.

Most of the structural tests, as said before, basically execute paths through the program’s control structure. So in order to classify them it is used the corresponding structural criteria and, the name path testing, is reserved to designate the exhaustive path testing. In this report, besides exhaustive path testing, path testing includes a simplified test called basic path testing.

5.2.2.1.Path Testing

A **path** through a program is a sequence of instructions or sentences that starts at an entry, junction, or decision and ends at another, or possible the same, junction, decision, or exit. A path may go through several junctions, processes, or decisions, one or more times. Paths consist of segments. In practise, the word path is used in the more restricted sense of a path that starts at the routine’s entrance and ends at its exit.

5.2.2.1.1.Exhaustive path testing

Objectives

It is considered an analogous strategy to exhaustive inputs testing within the functional or black box testing, and consequently it tries to detect all the errors existing in a program. The criterion used to achieve the objective of a complete testing is the execution of all possible paths that can be traced through the program's control flow.

Description of the test

Exhaustive path testing is the strongest criteria among structural tests. However, it is impossible to put in practice because of the huge number of possible paths that exist in normal routines.

It is a test applied basically by the programmer during the unit testing.

In path testing it is assumed that the specifications are correct and reachable, that do not exist more processing errors than those which affect to control flow and that data have been defined and are accessed correctly.

The following test case design and execution technique is valid for all structural tests that use the control flow graph as a program model, with the exception of the corresponding coverage criterion.

Although there are a lot of practical limitations, if we want to apply this method, the technique will be:

- The tester builds a control flow graph based on the module design specifications, the flow graph, or the source code, and traces all the possible paths on it.
- Some general rules for path selection useful whatever it is the coverage criteria:
  - Pick the simplest, functionally sensible entry/exit path
  - Pick additional paths as small variations from previous paths (favour short paths over long paths, simple paths over complicated paths, and paths that make sense over paths that don't).
  - Pick additional paths that have no obvious functional meaning only if it is necessary to provide coverage. But ask yourself first why such paths exist at all. Why was not coverage achieved with functionally sensible paths?

Once the testing paths have been traced, input data and previous states/conditions of the system have to be defined, in order to exercise each path. Moreover, it must be defined the expected outputs to be compared with test results.

Specified tests are executed. During test case execution could be that the results were coincident with the expected without having covered all branches of the foreseen path. This

phenomenon is known as **coincidental correctness** and it is solved with the **path instrumentation** technique. This technique consist of adding probes in concrete points of the structure to know the real path exercised during the test and to confirm the result. A problem of this technique is that adding probes causes that the program losses its real behaviour. Instrumentation can be another source of errors.

It is necessary to indicate that the coverage degree achieved at control flow graph level, or source code level, is not the same to that at object or machine language level.

For example, it has demonstrated that the criterion of covering all the instructions applied at source code level on a modern routine written in a high level language and which employs an intense logic, could achieve a result of 75% at code object level. For this reason it is recommended to analyse the coverage at object code level.

Advantages

It is the strongest structural criterion, though the loop existence and other structural elements cause that this method can not be applied.

Disadvantages

Some common weaknesses of the strategies that use the control flow graph as a model of the program are:

- Specification errors could be undetected;
- Path testing does not distinguish those paths impossible to achieve in practice;
- Planning to cover does not mean it will be covered. Path testing may not cover if there is faults;
- Path testing may not reveal totally wrong or missing functions;
- Interface errors, particularly at the interface with other routines, may not be caught by unit-level path testing;
- Database and data-flow errors may not be caught;
- Initialisation errors may not be caught by path testing.

Most of the errors explored by structural tests, that use the control flow as program model, are avoided when a structured programming language is used. Consequently, the effectiveness of those tests for programs written in such languages is reduced. However, those tests are indispensable for code written in Assembler, Basic, Cobol and Fortran due to the high error proportion that present in the control flow.

5.2.2.1.2.Basic Path Testing

Objective

This test tries to detect, the same as exhaustive path testing, the possible errors related to the program's control flow, applying a less exigent criteria that consists of executing at least once each independent paths.

Description of the test

In this case the tester, once control flow graph is obtained, must trace a set of lineally independent paths so that all the structure is covered at least once. The total number of independent paths can be obtained calculating the Cyclomatic Complexity. This metric actually provides a quantitative measure of the program's logic complexity. A complexity greater than 40 in a module without CASE sentences invites to redesign it. The theory of flow graphs, as well as the calculation of Cyclomatic Complexity are beyond this report. For any reference see [Pressman 97] and [Beizer 90].

Both design of the flow graph and calculation of the Cyclomatic Complexity can be accomplished in a manual or automatic way. There exist commercial tools that assist in the design of the path set and the coverage analysis, called coverage monitors.

Advantages

It is a test that includes statement and decisions tests. It shows a higher effectiveness in case of no structured languages.

Disadvantages

It is effectiveness reduces quickly when the size of code increases (the high number of paths to test increases test duration, complicates test cases definition and does not distinguish impossible paths). Some independent paths could not be tested separately.

5.2.2.2.Statement testing

Objective

Similar to Path Test, this strategy intends to detect anomalies in the logic of the program, mainly relative to the control flow, applying the simple criterion of executing at least once each line of code of the program under some test.

The analysis of the test coverage will inform us of the degree of execution of program's instructions. The coverage of this test is designated as C1. In the case of safety related programs it should be required coverage of 100%, during unit testing. The statement coverage at system level hardly it is over 85%.

Description of the test

It is a test applied basically by the programmer during unit testing. The tester based on the modular design specifications, or even the source code listing will define a set of test cases that exercise the different functional features until achieving the wished code coverage. The lower is the coverage analysis the more exhaustive is the test. It is the minimum criterion that programmer should apply during the unit testing.

There exist commercial tools that allow analyse the statement coverage at object code level automatically, called coverage monitors. These monitors are often implemented inside performance profilers included in debuggers or emulators.

This test can be carried out also in manual mode, using for instance the trace function of debuggers. Nevertheless, due to the large amount of information given by each trace, it is difficult to analyse and record the executed and not executed parts of code.

Advantages

It detects parts of unreachable code and parts not tested or executed yet (basically due to control, sequencing and processing faults).

Another favourable point it is that faults are uniformly distributed through code (supposing that exists a constant relationship between branches and number of statements). Consequently the percentage of instructions covered reflects the percentage of errors found (if we execute 70% of instructions and we found 3 errors, we can predict that remains approximately 1.3 errors undetected).

Disadvantages

It is considered a necessary but not sufficient criterion. The main disadvantage of this criterion is that it is not sensitive to some structures. That is, we can achieve a total coverage without covering some parts of the program (e.g., in the case of IF THEN structures it is not necessary to execute the false condition). It is not sensitive also to logic operator (AND, OR).

Usually test cases are more related to decisions than to instructions, and sometimes this leads to extreme measure results (e.g., two paths from a decision with a proportion of 1/99 instructions).

Statement coverage is more affected by calculus or processing than by decisions.

It is the weakest criterion of the structural test family.

5.2.2.3.Branch/Decision testing

Objective

It is intended to detect anomalies in the program's logic (control flow), but this time focussing the attention in decision elements. The criterion consists of executing at least once the alternative branches of each decision element. The coverage of this test is designated as C2. Decision coverage usually satisfies statement coverage (not in the contrary).

Description of the test

The same as statement coverage, decision coverage is considered as a basic criteria to be applied by the programmer during Unit Testing.

The tester will identify on the control flow graph or even the source code listing all existing logic of decisions. After that, he will select a set of test cases that exercise both true and false outcomes of each boolean expression (all branches in the case of multi-branch decisions), regardless of logical operators (and, or). Additionally, this test will consider exceptions and interruptions handling.

Some examples of instructions, which represent decisions element are:

a) In assembler:

- Conditional instructions that employ boolean variables: JZ, JNZ, JB, etc.
- Conditional instructions that employ variable of byte type: CJNE, DJNZ, etc.

b) In C language:

- Decision statements: IF-THEN-ELSE, SWITCH (multi-branch).
- Loop structures: FOR, WHILE, DO-WHILE.

There exist commercial coverage monitors that allow measuring the decision or branch coverage automatically.

Advantages

It is a very simple method and it has not the problems of statement coverage. When a structured language is used the achievement of this criterion (C2) implies the fulfilment of C1.

Disadvantages

It is still rather weak criterion. One of its problems, it is that if applied at source code level it ignores branches, due to logical operators, inside boolean expressions.

5.2.2.4.Condition testing

Objective

It is intended to detect anomalies in the program's logic (control flow), focussing the attention in the logic conditions inside decision elements. The criterion consists of writing enough test cases such that each individual condition in a decision takes on all possible outcomes at least once.

Description of the test

This test explores the true and false outcomes of each boolean sub-expression (condition), inside each decision element which are separated by logical operators (and, or). Sub-expressions are tested independently each other.

The tester firstly, will identify all the decision elements in the control flow graph like in the decision coverage, and then looking into each decision, will define the individual conditions inside. Finally, he will design the test cases that force the execution of both possible outcomes for each identified condition.

There exist commercials coverage monitors that allow measuring the conditions coverage automatically.

Advantages

It has usually a better sensitivity for control flow than decision coverage.

Disadvantages

Although the condition coverage criterion appears, at first glance, to satisfy the decision coverage criterion, it does not always do so. Then what is recommended, it is a mixed test condition/decision such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once.

5.2.2.5.Multiple condition testing

Objective

As in the previous strategy, multiple condition testing is intended to detect anomalies in the program's logic (control flow), focussing its attention in the logic conditions inside the decision elements, but in this case, even exploring all possible conditions combinations.

Description of the test

This criterion ensures the test of all "possible" combinations of conditions in a decision point. We say possible because it can happen that some combinations could not be created (e.g., in the case of (A>2)&(A<10), the combination false-false is impossible).

The way to proceed is like in the previous test, with the exception that here the tester must consider all possible combinations of conditions and define consequently a set of test cases that force their execution.

The required test cases for a complete testing can be obtained from the truth table of the logical operator.

The set of test cases that satisfies this criterion also satisfies decisions, conditions and decisions/conditions coverage criteria.

Typical faults detected by this testing are: errors in logical operator, (wrong logical operators, omitted or more than expected), errors in logic variable, errors in logical bracket, errors in relational operator, errors in arithmetic expression, etc.

There exist commercial coverage monitors that allow measuring multiple condition coverage automatically.

Advantages

It is considered the most complete criterion for the treatment of routines that contain decisions with multiple conditions. This criterion includes decision, condition and decision/condition criteria.

Disadvantages

The calculation of the minimum number of tests for covering all possible cases may be quite complex especially in case of complex boolean expressions. Furthermore, each logical operator added doubles the number of test cases.

The number of test cases required can vary substantially between condition expressions with similar complexity.

5.2.2.6.Linear code-sequence and jump testing (LCSAJ)

Objective

It is a variant of path testing that offers a different coverage criterion from those described before. Its objective will be then to detect errors related to the control flow of the program.

Description of the test

LCSAJ testing only considers sub-paths that can be easily represented in the program source code, without being necessary the control flow graph.

An LCSAJ is a sequence of sentences from the source code that start in an entry point or after a jump and are executed sequentially until other jump or an exit point.
It is defined with three values that conventionally are identified by the code listing numbers: starting line number, number of the last line, and number of line taking the control.

This “linear” sequence can contain decisions, as long as, the control-flow continues from one line to the next at run-time. The sub-paths are built concatenating several LCSAJ.

The tester, from the source code listing, can write a list of branches noting the necessary conditions to satisfy them. From this list, he will find the starting points of each possible LCSAJ. After that, he will seek the jumps that break the sequences, and finally, he will define all the LCSAJ.

Advantages

The advantages of this testing are that it is more complete than the decision testing and furthermore, it avoids the exponential complexity of path testing (number of paths is a function of the number of decision elements).

Disadvantages

The main disadvantages of this method are that, it does not avoid impossible paths, and the LCSAJs are not easily identifiable from documentation. Moreover, they are identified after the code has been written and still are complicated to deduce, and finally small changes in a module or routine may imply a great impact in the LCSAJs and test cases.

5.2.2.7.Loop Test

Objective

It is a test that only checks the validity of loop constructions. Its objective is to detect any fault related to loops, and especially, those in the control logic because they are the most likely to happen.

Description of the test

As the preceding methods, loop testing is a method that must be systematically applied by the programmer during unit testing if he wants to examine some errors specific of loop structures. Errors, that may not be detected by the previous methods.

It is said that there are four types of loops: simples, nested, concatenated and not structured.

A simple loop can be covered, in the sense of statement and decision coverage, with two cases: entering the loop "looping" and bypassing the loop "not looping". However, the experience shows that many of loop-related faults are not discovered by C1 + C2. We have already said that faults lurk in corners and congregate at boundaries. Thus, in the case of loops, faults are usually at or around the minimum and maximum number of times the loops can be iterated.

The technique proposed as effective to test a simple loop (minimum value of iteration is zero and without excluded values between zero and maximum value) consists of two steps. Firstly, identifying in the source code the existent loops, and secondly, designing the test cases applying next steps:

- Try bypassing the loop (zero iteration).
- Check whether the loop-control variable could be negative.
- One pass through the loop
- Two passes through the loop (Theorem by Huang).
- Execute m iterations, (m < n), being m a typical number of iterations and n the maximum number.
- Execute n-1, n and n+1 iterations.

NOTE: In assembler language, in addition, it should pay special attention to multiples of 2.

When the minimum value is not zero and/or have been excluded intermediate values for the loop-control variable, the procedure must be complemented with cases that explore the new extreme values. For instance, in case of a minimum value different from zero add the following cases: min-1, min, min+1, and in case of excluded values define two sets of tests as explained above for each possible set of values of the variable. Finally, in all cases include some test with an excluded value.

Unreasonably long test execution time could indicate faults in the SW or the specification.

In the case of nested loops, it is recommended to apply some strategies to reduce the large number of test cases produced if the previous procedure is applied directly (iteration values are multiplicatives).

Concatenated loops are considered as simple loops when are independents and as nested when the control variables are related each other.

In case of not structured loops (horrible) common in assembler, it is recommended to redesign.

Advantages

It is considered a complete and plausible criterion for the treatment of routines that have loops.

Disadvantages

The main problem of this method is the need of a detailed knowledge of all program's loops. It is also notable the increase of total testing time.

5.2.2.8.Data flow testing

Objective

This test tries to explore the anomalous things related to the program data, applying the strategy of selecting paths on a model of the data flow. There is a family of test criteria for path selection, which offer different degrees of coverage over a model.

This testing technique is earning relevancy, due to the current trend of the programs to migrate from code to data.

Description of the test

The model often used with all **structural data flow strategies** is the notation of control flow graph as in path testing, but this time each link is annotated with the operations accomplished on the data object of interest.

The data flow anomalies may be denoted by a two-character sequence of actions based on a nomenclature defined on the uses that can be done with a data object (create, define, initialise, kill, use in a calculation, use in a condition). What is an anomaly may depend on the application. Some combinations are clearly errors, other are suspicious combinations and the remainders are normal situations (e.g., a sequence of kill - use, is an error).

In addition to the above two-letter situations there are six single-letter situations. It can be used a leading dash to mean that nothing of interest occurs prior to the action noted along the entry-exit path of interest and a trailing dash to mean that nothing happens after the point of interest to the exit. The single-letter situations do not lead to clear data-flow anomalies but only the possibility thereof. Also, whether or not a single-letter situation is anomalous is an integration testing issue rather than a component testing issue because the integration of two or more components is involved.

In contrast to path testing strategies, data flow strategies take into account what happen to data objects on the links in addition to the raw connectivity of the graph.

The tester first creates the model form the source code in which annotates all data operations. Then, he selects test path segments that satisfy some characteristic (e.g., all sub-paths that have a *d* operation).

The strategies for selecting path segments differ on the extent to which predicate uses and/or computational uses of variables are included in the test set. The strategies also differ as to whether or not all paths of a given type are required or only one path of that type (e.g., all predicate uses versus at least one predicate use).

The main strategies of data flow testing are:

- All du (**ADUP**).
- All uses (**AU**).
- All p uses/some c uses (**APU + C**), all c uses/some p uses (**ACU + P**).
- All d (**AD**).
- All p uses (**APU**), all c uses (**ACU**).

It has been accomplished many comparative studies among data flow strategies and in relation with control path flow strategies on their relative strength. Figure 6 shows an order of the strategies from the strongest to the more weakness, as a result of the studies. The right-hand side of the graph, along the path from "all path" to "all statements" seems to be the more interesting hierarchy for practical applications.

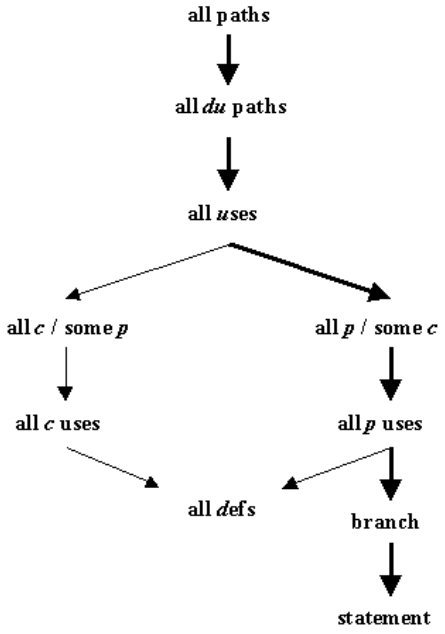


Figure 11 - Relative strength of data-flow strategies

In data flow testing is assumed that control flow is correct and faults are in the use of data objects. Despite of this, it is expected that some control flow problems will produce also symptoms that are detected by data flow analysis.

Currently, an important part of data flow anomalies are detected automatically by compilers.

The simplest or easiest data flow anomalies are usually found during unit testing. However, the most frequent and unfortunately subtler or more difficult to detect anomalies tend to

involve several components and they require integration testing. In the future it is expected an intense development of commercial data flow testing tools.

Advantages

It has been found that data flow testing is a cost-effective testing strategy. Furthermore, the test cases designed and executed have direct relationship with the way the program handles data. Another advantage is that the number of tests cases it is not so large as it seems, because usually one test case covers others.

Disadvantages

Finding data flow covering test sets is not more nor less difficult than finding branch covering tests, merely more tedious. There is more bookkeeping because it is necessary to keep track of which variables are covered and where.

5.2.2.9.Summary of the main features of White Box tests

Table 6 summarises the main features of White Box tests presented in this clause.

5.2.3. Black Box Tests

Black Box tests focused on verifying whether the actual behaviour of the program or system matches the described in their specifications. They are not an alternative to WB tests, but they will be a complement with the aim of finding out different types of errors.

BB tests try to find the following categories of faults:

- Incorrect or missing functions.
- Interface errors.
- Errors in data structures or in external database access.
- Performance errors.
- Initialisation and termination errors.

BB tests design generates some modelling problems which do not exist in structural testing. WB analysis is always performed in an homogeneous frame, that is, a control or data flow graph deduced from the source code, whereas in BB testing there is not a standard model for describing the expected behaviour.

Table 6: Summary of the main features of White Box Tests

TEST	Inputs	Fault assumption	Coverage criteria	Applicability	Cost-effectiveness
Path testing	Module design documentation (control-flow graph, flow chart, etc) or directly source code listing.	Most faults can result in control-flow errors and therefore misbehaviour could be caught by control-flow testing. Specifications are correct and achievable, and data are properly defined and accessed. (mainly control and sequence faults)	Exhaustive path testing: exercise all possible paths through program control flow. Basic path testing: exercise a set (it is not unique) of independent paths. It assures C1 + C2 coverage criteria.	The number of possible paths in a module becomes soon impractical. Not distinguish unachievable paths. Sometimes it is complicated the definition of test cases for the selected paths. Instrumentation methods to verify paths can generate time distortion.	Exhaustive path testing is unachievable. Basic path testing is effective and easy to implement. Efficiency reduces quickly when the size of code increases (time cost). Particularly recommended for unstructured languages.
Statement testing	Module design documentation. Source code listing and object code listing.	Untested pieces of code leaves a residue of faults in proportion to their size and probability of the faults. And low-probability paths can be not exercised during testing (mainly control flow faults and unreachable code).	Execute all statements in the program at least once under some test.	Easy to implement.	The weakest criterion. Insensitive to some control structures and logic operators. C1 should be achieve at object code level.
Decision or branch testing	Module design documentation (control-flow graph, flow chart, etc) or directly source code listing.	Faults directly affect control-flow decision elements.	Execute enough tests to assure that every decision has a true and false outcome at least once (or each branch direction is traversed at least once) under some test.	Easy to implement.	When applied to a structured language it includes C1 criterion, but still rather weak. Enough for routines containing only one condition per decision. It ignores the conditions in a decision.
Condition testing	Module design documentation (control-flow graph, flow chart, etc) or directly source code listing.	Faults directly affect control-flow predicates (individual conditions in decision elements).	Execute enough tests to assure that each condition in a decision takes on all possible outcomes at least once.	Requires more design effort than decision coverage.	Sometimes stronger than decision coverage. Not always satisfy decision criteria.
Multiple condition testing	Module design documentation (control-flow graph, flow chart, etc) or directly source code listing.	Faults directly affect control-flow predicates (combination of conditions in decision elements).	Execute enough tests to assure that all possible combinations of condition outcomes in each decision are invoked at least once.	Calculation of the minimum number of test cases needed to cover all possible combinations of condition outcomes in each decision can be complicated, specially in the case of complex boolean expressions.	More thorough test criterion than condition coverage. More complex and time consuming. Includes decision, condition and decision/condition coverage criteria.
LCSAJ	Source code listing	Faults affect control transfer and segments processing.	Exercise all linear sequences of statements and jumps in a routine.	Selection of LCSAJ is rather complicated. Small changes in the code can produce a great impact on LCSAJs and test cases. Does not distinguish unfeasible LCSAJs.	Stronger than decision coverage, but less than multiple condition criteria. This criterion avoid the exponential difficulty of the criterion above.
Loop testing	Module design documentation (control-flow graph, flow chart, etc) or directly source code listing.	Faults affect the loops in the program (initial or terminal value or condition, increment value, iteration variable processing, etc).	Execute enough tests to assure that all loops in the program are exercised thoroughly.	Testing defined procedure for loops. Testing a single loop requires to run 7 or 8 tests. Test can take a relative long time.	Complement other coverage criteria (C1 and C2) when the routine has loops.
Data-flow testing	Module design documentation in case of module testing and system design documentation for SW integration test. Control flow graph annotated with data objects created from the source code listing, call trees, etc.	Control flow is correct but the program could contain data faults (e.g., initial and default values, overloading, wrong type, closing before opening a file, etc).	Execute enough tests to assure that data characteristics (state and usage) for all data objects are exercised. There exist several coverage criteria: from the weaker *All Definitions* to the stronger *All Def. and Uses*.	Not more difficult, but usually more tedious.	AU criteria has probably the best payoff for the money. Recommended to low-level testing for programs with a considerable data design, and essential for almost any program in integration testing.

NOTES: - In the input column it has not been included information about test specification (test planning documents) and other relevant information as the inspection &reviews results.
- All these test criteria assume that program specifications are correct and achievable.
- The stronger strategies, typically the higher cost.
- Sometimes the test tools (Coverage monitors) do not specify the coverage measured. Then, the tester will not know whether it is monitoring statement coverage, branch, both, etc, for source code, for object, or for memory words, unless it asks. Modern tools have coverage monitors built-in, and independent stand-alone coverage monitors has become the exception.

The main representation techniques used for modelling a program or system at a functional level are:

- Graphs (control-flow, data flow, finite state machine, etc.).
- Decisions table.
- Equivalence classes.

BB tests are usually predominant in higher order testing, like function testing, system testing and acceptance testing.

5.2.3.1.Behavioural or high level control-flow path test

Objective

This technique tries to detect those program faults that produce an anomalous behaviour in the program control flow visible for the user. The criterion used is the execution of all the paths functionally sensible of the flow graph built from the specifications, considering opaque the program or system.

Description of the test

This test begins to be effective during SW components integration testing and SW and HW integration testing, but really where it turns useful is in system testing. When SW components have a considerable size, this technique is even advisable during unit testing.

The tester or the programmer will create a system control flow model from the specifications (high-level specifications of the system, or specifications of the program design). To build the model is used the same graphical notation as the control flow in structural testing, though with a different semantic (nodes = objects, and links = relations).

Based on the model, tester traced enough paths to assure 100 percent link cover. After that it sensitises the selected paths and executes the corresponding tests to check whether the states and events specified exist or not in the implementation, and outputs generated are the expected.

The graph can also be represented in form of tables or lists of objects and relations.

This technique assumes that most faults can result in control flow errors and therefore misbehaviour could be caught by control flow testing. However, the primary assumption about the faults targeted by this technique is that they directly affect control-flow decisions or predicates or that the control flow itself is wrong. Gross control flow faults are not common in programs written with structured programming languages.

Although this technique may detect some computational faults that do not affect the control flow, obviously is not the best to use.

The test design and execution consists of the following steps [Beizer 95]:

1. Rewrite the specification as a sequence of short sentences, identifying conditions and breaking up compound conditions to equivalent sequences of simple conditions.
2. Number sentences (they will be the nodes).
3. Build the model.
4. Select test paths.
5. Select input values that would cause the software to do the equivalent of traversing the selected paths if there were no errors.
6. Run the tests.
7. Confirm the outcomes and paths.

Advantages

Behavioural control flow testing applies to almost all software and is effective for most software. It is a fundamental technique. Its applicability is mostly to relative small programs or segments of larger programs. To test an entire large program would be too difficult because the models would be very big and, as a consequence, path selection and sensitization would be much too complicated to justify the effort.

Disadvantages

Usually, specifications errors and omissions are not discovered. This technique is unlikely to find spurious or gratuitous features that were included in the software but were not in the requirements. If this technique was already applied during unit test, it will not discover many new errors. It is unlikely to find missing paths and features if the program and the model on which the tests are based are done by the same person.
In case of using an oracle, the technique does not result effective (supposing both oracle and tests has been developed by the same person). This technique will not discover process faults (calculation faults) that they do not affect the control flow.

5.2.3.2.Logic-based testing (decision tables)

Objective

This technique explores specification and implementation errors of logic-intensive programs using decision tables as program or system logical model. This type of representation allows to define all possible input conditions and to establish the relationships that lead to the outputs or actions.

The use of an algebraic base makes this technique to detect completeness, consistency and redundancy errors in the specifications.

Description of the test

In this report, logic based testing technique is presented as a functional test method because it is applied to the specifications, and therefore recommended for validation testing. This technique may also be applied in the same way to the program structure (i.e., to the implementation control graph) but in this case it would be considered as a structural test and recommended for other stages of the test process.

This strategy consists of generating a set of combination of input conditions (rules) that will cover a large part of possible input circumstances according to the program specification. To do that, the tester builds a logical model from the specifications, so that this model offers a concise description of the system in the form of logic equations. Then, applying the Boolean algebra rules, he tries to simplify all input conditions that lead to each action or output until achieving the minimum expression.

The coverage criteria can range from only using the conditions of the minimum expressions of an action, to the generation of all conditions of the expanded expression.

These logical modelling techniques are applied especially in programs or portions of programs (segments) with combinational logic.

The test design and execution consists of the following steps [Beizer 95]:

1. Identify conditions and actions in the system specifications, and establish their relationships.
2. Fill up the decision table (model).
3. Expand the table for its analysis (completeness - every input conditions combination is described by at least a rule; and consistency - every input conditions combination leads always to the same set of actions).
4. According with the coverage criteria used, select the input conditions combinations that cause the wished action
5. Generate the input values for those condition.
6. Execute the tests.
7. Confirming the outputs or actions and the paths go through.

Advantages

In case of logic-intensive systems with specifications written in natural language, this technique allows to analyse the specifications in an organised way, previously to the test. By means of the Boolean algebra rules it is possible to detect parts of unreachable code, infinite loops, etc.

Disadvantages

It is required a knowledge of Boolean algebra. The conversion of specifications into the decision table is not trivial (identification of input conditions, decomposition of compound conditions, combined actions, etc.). It is useful for combinatorial systems only.

5.2.3.3.Behavioural or high level data-flow test

Objectives

In the above presentation of the data flow test techniques (chapter 5.2.2 White Box Tests), we said that these techniques are more powerful than control flow techniques. More powerful in the sense they detect more errors and let us to create all tests that we would be able to create using control flow testing techniques.

In this case data flow testing techniques use the data flow graph, obtained from the specifications, as a model of the system or program. This technique tries to detect anomalies in the definition and processing of data, the same as structural testing. For this reason, it is avoided introducing unnecessary control aspects in the model (that is, aspects not specified in the requirements and that are produced by the inherent characteristics of the programming languages and system hardware), assuming that the control flow does not contain simple errors.

Description of the test

This test is based on the creation of a data flow graph with an appropriate detail all the time in order to describe data processes and then to select test “paths” (data flow slices) from the model according to a specific coverage criteria.

In principle it is a recommended technique for high level testing on system with a certain data contents and not many control requirements.

The data flow graph will represent the information flow and it will not provide explicit indication of the processing sequence (conditions, loops, etc), unless it is necessary for data processing.

Some common kind of sequencing information that may be present in a data flow graph are:

- Convenient, but not essential.
- Essential.
- Synchronisation (concurrent processes).
- Iteration (loops).

Usually, the loops deduced from the specifications are tested applying the same heuristic techniques as in structural test on the behavioural control flow model. Nevertheless, if the loops are not complicate (nested, etc), it is possible to consider their key data elements in the data-flow graph and thus, to include this test within the data-flow test.

Some strategies used for the selection of test cases, ordered from the highest to the lowest coverage are [Beizer 95]:

- All - Uses + Loops.
- All - Uses. It covers all links in the data flow graph.

- All Nodes.
- Partial Node Cover, called also All Definitions.
- Input/Output Cover + All Predicates.
- Input/Output Cover

The design and execution of the test cases is similar to the behavioural control flow test, with some minor differences which result from the different nature of the models.

The test cases are designed as following:

1. Identify input variables, especially constants.
2. Rewrite the specification as one sentence per function to be calculated.
3. Elaborate different function lists, starting with those that depend only on input variables, and going on with the next level of functions, until expressing the output variables as function relationships.
4. Examine intermediate functions to check if the sequencing is essential or merely convenient. Try to simplify the model (adding intermediate nodes, etc.).
5. Finish the model naming the links with functions variables (nodes) and verify it.
6. Select test paths (data flow slices) according to precedent criteria.
7. Select inputs values for the selected cases (sensitise slices), and predict the expected outputs.
8. Run the tests and confirm the outputs (values at intermediate nodes).

Advantages

Firstly, it is appropriate to underline the theoretical power of this technique, which includes all the cases explored by the control flow. But, as always, the development of its capabilities supposes more work in the design of test cases. On the other hand, to profit from that capability it will have to take into account some control aspects which complicate the data flow model.

It is likely to find characteristics or extra functions that have been included in the program without being in the specifications.

Disadvantages

This test does not usually discover errors or omissions in specifications. It might lose effectiveness when SW and test design are done by the same person, but less so than for behavioural control flow testing. Tests are not better than oracles (supposing both, oracle and tests design has been developed by the same person). It will not provide the tester much if he does not find ways to verify those intermediate nodes.

5.2.3.4.Equivalence partitioning

Objective

The objective of this technique is to discover any type of error in the individual treatment of inputs that could have in the program or system, by designing test cases from the specifications using the partition of the input domain in classes of equivalence.

Description of the test

It is a technique widely used during the system testing.

This method is focused on the input domain defined in the specifications. It tries to cover this domain applying the criterion of partitioning it in a finite number of equivalence classes from which it is enough to choose one representative value per class, reasonably assuming (but, of course, not absolutely sure) that a test with this value element is equivalent to a test with any other value of its class.

The design of test cases is developed in two stages [Pressman 97]:

1. Identify the equivalence classes. The equivalence classes are identified by taking each input condition (usually a sentence or phrase in the specification), and partitioning it in two or more groups using a largely heuristic process. Noticed that two types of equivalence classes are identified:÷ valid equivalence classes (they represent the inputs specified for the program) and invalid (rest of possible inputs).
2. Define test cases. It will be assigned a unique number to each equivalence class identified and defined test cases until covering all the equivalence classes. The valid classes will be covered with the minimum number of test cases while invalid classes will be covered individually (to prevent that certain erroneous-input checks mask or supersede other erroneous input checks).

Usually, it is applied to an input variable or a simple combination of two variables.

Advantages

It is considered an effective technique because its direct application and limited number of test cases. It is vastly superior to a random selection of test cases

Disadvantages

It overlooks certain types of high-yield test cases, for instance, input boundary values, combinations of input values, etc.

5.2.3.5.Boundary-value analysis

Objective

It is a technique that complement equivalence partitioning. In this case, it is intended to discover program or system errors associated with limit conditions that define the input and output domains in the specifications: situations directly on, above, and beneath the edges of input equivalence classes or output equivalence classes.

Description of the test

It is a technique mostly used during the system testing.. However, it can be based on implementation information and be applied as a structural technique in earlier steps than system testing.

This method is based on the experience has showed that test cases that explore boundary conditions have a higher payoff than test cases that do not. It differs from equivalent partitioning in that instead of selecting any element in an equivalence class as being representative it analyses each edge of each equivalence class and summits them to test with one or more cases. Furthermore, this technique explores also the equivalence classes of the output domain, through the corresponding input values (boundary conditions of the input domains do not represent always the same set of circumstances as the boundaries of output ranges).

In case of multidimensional domains, it is recommended to check also combinations of boundary values.

It is assumed that the processing is accomplished correctly and errors have been produced in the domain definition or implementation.

Although it apparently turns out a simple technique, in practice its correct application (identification of all the boundaries) supposes a considerable intellectual effort.

Within this method it is also convenient to test internal data structures boundaries (for example buffers, tables, arrays, etc.).

Advantages

It is considered a very effective technique.
A random testing will have a very low probability to verify the domain limits.

Disadvantages

Its correct application requires a certain mental effort.
It must be complemented with other test techniques that verify the processing.

5.2.3.6.Error Guessing

Objectives

The aim of this method is the design of special test cases that explore possible errors not covered by the rest of testing methods.

Description of the test

It is a technique that apparently does not apply any methodology for the design of test cases. The tester uses his intuition and experience to predict certain types of probable errors in a given program, and develops test cases to explore them.

It does not exit a defined procedure for the design of test cases since it is largely intuitive and ad hoc process. Some basic ideas are: to define possible errors and error-prone situations list, to identify possible assumptions that the programmer might have made when reading the specifications, etc.

When a tester chooses a test technique he is assuming a certain class of errors since each technique is specialised in certain classes of errors. Error guessing is not actually a technique, it is more a compendium of all testing techniques. The basic objective is to demonstrate that there are not common (usual) errors in the program. To do that, it is based on the program error statistics, taking into account that these data should not be used directly as a guide to build the test cases, but as orientation.

In this way, using the error guessing does not imply a priori to assume a type of error, unlike to the rest of test methods.

Advantages

This test, when it is carried out by an experienced tester, is probably the most effective simple method of test design. It is based on the experience, being very quick detecting errors.

Disadvantages

The main problem is the total lack of methodology. Used by inexperienced testers is a complete waste of time.

5.2.3.7.Syntax Testing

Objective

As stated by its name, the objective of this test is to validate the input-data syntax. In other words, to explore what happen when data is introduced using the syntax defined in the specifications. This test ought to evaluate the program ability to accept valid data and reject invalid data.

Syntax testing is a powerful technique for testing applications using command languages, for example: programs that receive commands (command-driven software); menu-driven software; software packages for PC that use *macros* to automate repetitive operations (macro languages); format messages language in communication systems; etc.

Description of the test

This technique is related to the hostile character of the external world which inexorably summits sooner or later to the programme to anomalous data that may produce an immediate failure or a chink in the system's armour allowing other bad data to go in the system and corrupt it.

This problematic can also exist in the internal environment of big systems, usually subdivided into loosely coupled subsystems and consequently with many interfaces. These interfaces present new opportunities for data corruption and may require explicit internal validation. Furthermore, HW can fail in bizarre ways that will cause to pump streams of bad data into memory, across channels, and so on. And then there are always alpha particles.

Syntax testing is applied usually by independent tester during system testing.

The method consist of the following steps [Beizer 95]:

1. Identify the language o format. Sometimes it is difficult to identify a language because it is hidden, that is, it is used one not recognised programming language. Some examples are: user or operator commands in applications used interactively, task control languages in the case of batch processes, convention used for the communication between processes at system level, etc.
2. Define a formal model of the language syntax, using for instance a Backus-Naur form. In most cases the syntax will be no documented and before to begin testing there will be to know what is tested. Some useful sources that can help to define the syntax are: co-operation between designer and tester, user manuals, help screens, data dictionary, etc.
3. Test and debug the syntax to ensure that it is complete and consistent and it satisfies the expected semantics.
4. Test all normal conditions (Clean Testing), that is, the set of input strings that cover all options, included critical loop values. The most difficult part of test of normal cases is to predict the outputs and verify the correctness of the process. That is, it is turned into an ordinary functional test (semantics test). Covering the syntax graph (go through all links/branches) it is assured the test of all options. This is the minimum obligatory requisite; similar to branch testing in the control flow graph.
5. Add cases that can not be produced with the graph –dirty cases- (Dirty Testing). It is recommended to generate them methodically with a top-down process, starting by changing an element of the graph each time.

The proposed strategy for designing test cases is the creation of an error at each time, being the rest of the syntax correct. When single error testing has been completed, combination of errors will be tested.. Some reasonably judgements will have to be done in order to restrict the number of test cases. However, this is almost impossible without a knowledge of the implementation details.

The errors assumed in this test are syntax errors, and not errors in the field values that will be tested with equivalent partition and boundary value testing. Typical syntax errors are: errors in the syntax specification (incomplete, inconsistent), errors in the syntax of the data validation routine (it is rejected a valid data, it is accepted a not valid data), etc.

Advantages

It is easy to apply and automate.

Disadvantages

There are two main disadvantages. The belief that random test cases are enough and turns out more profitable than automatic generated cases. And the great faith in the effectiveness of what is called monkey testing.

5.2.3.8.Other Tests

- **State Transition Test**

There exist programs whose behaviour fits well a finite states machine, and consequently they can be modelled or described using state graphs and tables.

State transition testing is appropriate for that type of programs and uses state graphs to design test cases. The strategy for the state testing in principle is analogous to path testing in the flow graph, that is, to cover all possible paths of the state graph. But, the same as for path testing, this strategy is impracticable also in this case.

Then it is turned to the coverage notion and it is talked about states-transitions coverage as the minimum mandatory requirement. From this criterion might be established other more demanding, generally based on the execution of longer sequences (number of transitions). It may be interesting to include cases to test that the unspecified transitions can not be induced.

These models are mainly applied by independent tester especially during system testing–for functional tests design,.

This test could be useful in any process in which output depends on one or more event sequences (for example, detection of specific input sequences, sequential formats validation, etc), most of the protocols, menu-driven software, whenever a characteristic is directly implemented as a state transition table, etc.

A disadvantage of state graphs is that they do not consider the time, they only represent sequences. Nevertheless, some models of finite state machines can be adapted to consider the notion of time (i.e. temporized Petri nets).

Some tests sometimes considered as Black Box Test, but exclusive of systems test are:

- **Volume testing**

This type of system testing is subjecting the program to heavy volumes of data. For instance, an operating systems job queue would be filled to their maximum capacity. The purpose of these tests are to show that program can not handle the volume of data specified in its objectives/specifications. It is expensive, but at least a few volume tests must be carried out.

- **Stress testing**

This type of test involves subjecting the program to heavy loads or stresses (peak volume of data over a short span of time). It is applicable to programs that operate under varying loads, or interactive or real time, and process-control programs. For instance, a process-control program might be stress-tested by causing all of the monitored processes to generate signals simultaneously.

- **Performance testing**

Many programs have specific performance or efficiency objectives, stating such properties as response time and throughput rates under certain workload and configuration conditions. These tests must be devised to show that the program does not satisfy its performance objectives.

- **Usability testing**

Another category of system test cases is an attempt to find human-factor, or usability, problems. Some kind of considerations that might be tested are: degree of adaptation of the user interface to the intelligence, educational background and environmental pressures of the end user; outputs of the program meaningful, non-abusive; straightforward error diagnostics (error messages); return of acknowledgement to the inputs; etc.

Some aspects of usability might be considered under syntax testing (e.g., syntax consistency and uniformity, conventions, semantics, format, etc).

- **Configuration testing**

Some programs support a variety of HW configurations (e.g., types and number of I/O devices and communications lines, different memory sizes). Such programs should be tested with each type of hardware device and all possible configurations. If the program itself can be configured, each program configuration should also be tested.

- **Recovery testing**

Those programs which have recovery requirements, stating how the system is to recover from programming errors, HW failures, and data errors, should be tested to show that these recovery functions do not work correctly.

Some characteristics considered in these system tests are studied again in clause 5.2.4 “real time system test”.

5.2.3.9.Summary of the main features of Black Box tests

Table 7 summarises the main features of Black Box tests presented in this clause.

5.2.4. Real Time System Test

Real time systems (RTS) are systems that interact with the real world in a base of time. The RTSs generate an action in response to an external event in a predefined time. Therefore, in addition to the functional requirements, these kind of systems will have time constraints.

Industrial automation and specifically, machinery control is one of the typical RTS application area. So, the systems object of the project STSARCES are systems of this type which will realise safety related control and monitoring.
The software and hardware configurations of these systems will vary depending among others on:

- functional requirements (one or more tasks, programmability, etc),
- safety requirements (control category),
- performance requirements (response time, multitasking, hierarchical interruption services, etc), etc.

This way, it is possible to find RTSs ranging from a simple embedded systems (in which the function is codified as an endless loop, i.e. one or some few task(s) that are executed continuously), more complex embedded systems that include multitask and interruptions (might have limited programming capabilities), to PLCs or PCs totally programmable by the user.

Some RTS characteristic attributes are:

- Interrupts handling and context change.
- Concurrency, multitasking and multiprocessing.
- Communication and synchronisation between tasks.
- Response time (time constraints).
- Data transference rate and time.

Table 7. Summary of the main features of Black Box Tests.

TEST	Inputs	Fault assumption	Coverage criteria	Applicability	Cost-effectiveness
Behavioural or high level control-flow testing	Functional requirements and specification documents from which it is created the behavioural control-flow model (graphical or list notation).	Most faults can result in control-flow errors and therefore misbehaviours could be caught by control-flow testing. Requirements and/or specifications can be incorrect and their analysis could be included in this test.	Executing enough test paths on the model to assure 100 % link cover. (to verify that system logic meets requirements).	Applicable to relatively small programs or segments of larger programs. To verify behavioural control-flow paths sometimes it is needed programmers` co-operation to install assertions for intermediate calculations.	The better the unit testing, the less likely it will find new faults with this method in high level testing. Effective for most software.
Logic-based testing (decision table)	Functional requirements and specification documents, which can be represented by decision tables.	Logic-intensive SW designed instinctively is almost never right, hard to test and maintain (specifications contains combination of input conditions). Requirements and/or specifications can be incorrect (incomplete, inconsistent, etc).	Exercising enough test cases to ensure all combinations of conditions (predicates) that lead to the actions. Coverage criteria range from just test the conditions present in the minimum expressions of an action to test all the conditions after expand the table.	Conversion of specification into decision table is not easy. Require a strong boolean logic development to simplify the expressions of the actions.	Very effective when specifications are given as a decision table or they can be easily converted into it. Adequate only for combinational systems or segments, and loop-free. Beneficial side effect in pointing out errors in the specifications.
Behavioural or high level data-flow testing	Requirements and specification documents. Data-flow graph.	Control flow is correct but the program still could contain data faults.	Execute enough tests to assure that characteristics (state and usage) for all input/intermediate/output variables are exercised. There exist several coverage criteria: from the weaker *Input/Output cover*, to the stronger *All Uses + Loops*.	Test design is harder than behavioural control-flow and is done manually. Sometimes it is needed programmers` co-operation to install assertions to verify intermediate nodes.	More powerful than behavioural control testing whenever control dependencies of data are considered.
Equivalence partitioning	Requirements and specification documents.	An input domain can be partitioned into a finite number of equivalent classes or segments such that can be reasonably assume that a test using any value of a segment is equivalent to other test using another value of the same segment (that is, if the first test detects an error, the will be expected to find it as well).	Cover as many uncover classes as possible with one test case for valid classes; and to test one class each time for invalids.	Seems to be simple, but the specification of domains, their closures, extreme points, etc. sometimes is rather complicate.	Efficient due to its direct application and reduced number of tests. Vastly superior to a random selection of test cases. Need to be complemented with other methods (e.g., boundary value, etc).
Boundary-value analysis	Requirements and specification documents.	Experience has shown that inputs and outputs domains boundaries are fault-prone.	Execute enough test cases to assure that all domains boundary/extreme values and their combinations in case of multidimensional domains are tested.	Seems to be simple, but the specification of domains, their closures, extreme points, etc. sometimes is rather complicate.	Very effective method. Testing all combinations of extreme points is ineffective [generate many tests (4^n with $n=n^o$ dimensions), most of which meaningless and/or misleading]. Strategies as 1x1 and other stronger, increase the effectiveness of the test.
Error guessing	Requirements and specification documents.	Particular error-prone input or output situations (overbooked when the program was designed), faulty assumptions that the programmer might have made when reading the specifications, etc.	Does not follow any method or criteria. Based on intuition and experience, for each particular program certain probable type of errors supposed and tested.	Easy to implement.	Effective and more simple, if done by an experienced tester (a few tests concentrated on error causes).
Syntax testing	Requirements and specification documents, help screens, data dictionary and user manuals. Backus-Naur Form graph (BNF).	Inputs commands (and data) are not processed properly, and as a consequence their interpretation can lead to crashes and data corruption.	Can be divided into two parts: clean testing intended to cover the syntax graph, and dirty testing intended to exercise syntactic errors in all commands in an attempt to break the software and to force every diagnostic message to be executed.	Easy design automation of test cases and execution (by means of a capture/replay system or a driver). The number of multiple errors in a command should be limited to a reasonable, because this number increases exponentially with the multiplicity factor.	Effective for applications with command languages (e.g., command-driven SW, menu-driven, communications, etc.), whenever it is automated.

- Special requisites to handle errors and failure recuperation.
- Resources assignation and priority handling.
- Asynchronous processing.
- Unavoidable coupling between SW-SO-HW-other external elements.

In consequence, RTS software will be closely coupled with the real world.
Time dependencies of tasks and asynchronous events add a new element to testing, the **time.**

Now, tester not only has to consider the design of test cases from the WB and BB techniques seen before, but also he will have to add new cases to test the specific characteristics of these systems listed above.

A general test strategy could be [Pressman-97]:

Tasks Testing: The first step is testing each task independently, designing WB and BB tests to discover logical and functional errors. These tests do not explore temporisation or behaviour errors.
Behaviour Testing: Using system models simulate the system behaviour and examine the behaviour as a consequence of external events. In this step behaviour faults are detected (the system model do not have the behaviour specified in case of events).
Inter-Tasks Testing: Asynchronous tasks that communicate with other tasks are tested, with different data rates and different process workloads. They are tested also the tasks that intercommunicate through queues or data buffers. In this step time related errors and data storage errors are tested to detect size errors in the memory areas involved.
System Testing: After integrating the SW and HW use to be applied a complete series of tests to discover faults in the interface SW-HW, etc.

Interruptions handling are another essential tests. These test will verified the programmed priorities on the system, the different interrupt processing (interrupt service), management time, processing of several interruptions close in time, global data area used by interruptions, etc.

There is another characteristic type of test, called race coverage test. This type of test checks whether multiple threads execute the same code at the same time. It helps in detecting synchronisation faults in the access to resources. It is very used when testing programs with multiple threads (“multi-threads”), as operative systems or multiprocessor systems.

Exceptions handling in RTS have a special consideration due to the high reliability and time requirements these systems usually have. Exceptions are produced by errors that provoke a malfunctioning in the program operation or a system failure. These errors can be originated by faults in the software or in the hardware of the system or even in external components.

Testing methods for RTS are not so developed as the methods for other type of programs (consumer, general business, etc).

5.2.5. Conclusions

In clause 4 “Black box versus White box testing”, it has reflected on two of the existing approaches to demonstrate the correction of a program: structured-based testing and test based on the function. And it has concluded that no one demonstrate to be absolute (do not assure test completeness), furthermore, given that they are complementary, any test strategy should use both.

Discarded the complete testing, the position must be develop a reasonable strategy that provide enough testing to ensure that the probability of failure due to residual faults is low enough to accept. “Enough” implies judgement. And what is sufficient for an application is insufficient to other, due to the consequences that a failure can have.

In our case, the standard EN 954-1 defines requirements for the SRPCSs in functional and category terms. It is just the translation of those requirements in others more in agreement with the parameters normally used to measure SW quality and testing (fault types and distribution, testing types and levels, coverage to achieve, metrics, reliability models, etc.) on of this project’s challenges. In WP 3.2, it is intended to evaluate the contribution that accomplishes the use of a functional or a structural approach in testing.

It does not exist conflict between behavioural, structural and hybrid (combination of the two previous) strategies. And a priori it can not be said that one of these testing strategies is superior to the other for testing a program. This is because the relative usefulness of a strategy depends on several factors, such as: safety requirements of the program (functions and category), program size, procedural/fault prevention measures applied in the design process, program’s features (structure, logic and data), nature of possible faults (according to programming language, stage in the testing process, etc.), state of our knowledge, application cost, etc..

This dependency shows that the future definition of a test strategy will be rather complex, and it should be flexible enough to take in account and adapt to several factors.

Nevertheless, and despite of the phase we are in the project, the study accomplished allows us to advance a series of rules or guidelines which help us selecting the most interesting techniques. These guidelines may also be useful as a starting point for a subsequent design of an effective testing strategy (selection algorithm): more suitable test technique, testing approach to use, coverage definitions for the different testing levels, etc. in order to meet the requirements of each category.

Some researches have given theoretical results on the relative structural techniques effectiveness considering only inclusion relations. However, in practice, the relative effectiveness will depend on the number of test cases necessary to satisfy the different criteria (testing cost) and the probability that there exist a type of fault detected by a test criteria and not by others.

Supposing a coverage of 100% for all test criteria (i.e., each test is complete, that is all criteria specific structural elements are exercised), it is possible to define a relationship among the test criteria of the type "test A includes test B". This inclusion relation means that a set of test cases that satisfies A criterion also satisfies B criterion. The inclusion relation it is not enough to establish an effectiveness relationship, and that is why it provides only a partial indication of the relative effectiveness of test techniques.

The inclusion relation is not valid to order all structural coverage criteria. And it does not serve to establish a relationship between structural and functional criteria.

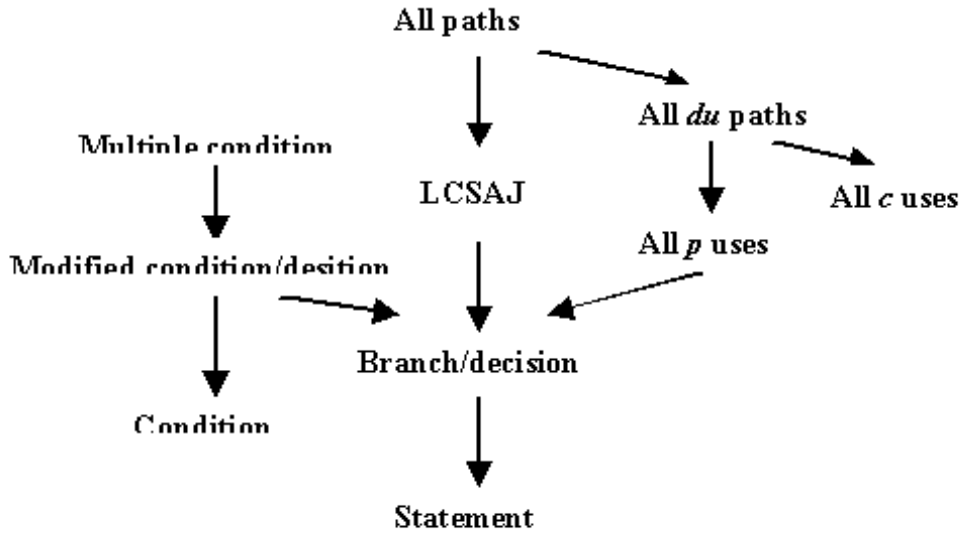


Figure 12 - Partial ordering of Structural Test Coverage criteria

100% statement and decision coverage in unit testing are accepted today as the minimum mandatory testing requirement. Statement coverage is established as a minimum testing requirement in the IEEE unit testing standard (ANSI87B).

However, that goal results unreachable or prohibitive in terms of test productivity when we try to test larger size items. This is mainly because, as said before: it is difficult to test low level details when handling large size objects, and high level testing tries to detect other type of faults.

Some authors say that there exists 15% of coverage loss in each testing level, for most of structural coverage criteria.

In general, at system level, it can be achieve a maximum C1 coverage of 85/90%, having applied all possible test methods.

As we can see, there are many reasons that advise to assign higher structural coverage values at lower testing levels (e.g., unit testing), and reduce the requirements as we go up at testing levels.

Respect to functional strategy, it will be predominant during validation stage. It can be applied in unit and integration testing (where the term function has to be situated in the context of routine's specification). But where actually it results almost exclusive it is in subsequent phases in which it is necessary to handle large objects and abstract from the details, in order to verify high level functions.

During validation, independent and objective testers are recommended.

Sometimes hybrid strategies are used. These strategies apply functional criteria for the design of test cases and use structural indicators to measure the degree of execution of the program. This way, it is achieved a behavioural verification of the program complemented with information on the degree of execution of the code, in structural terms.

This is the strategy usually applied by SW testing tools. Testing process consist of applying first a series of functional tests to detect the behaviours that do not match the specifications, and then changing the strategy, execute new test cases to exercise those paths and code parts that have not been still executed, until achieving the proposed coverage rates. The tools inform tester which parts of the code (according with the coverage criteria) have not been exercised yet, and sometimes, even offer the necessary conditions for exercising them.

Functional tests, unlike structural tests, have a heuristic basis (they use criteria based on faults made traditionally), though some authors give them a more analytical or formal treatment.

The functional strategy could be completed with some structural tests carried out on specific parts of the program revealed as critical by a prior analysis.

Functional tests, derived from structural techniques (e.g., Behavioural control-flow test), applied during system testing result low effective if previously during unit testing the fundamental structural techniques have been used.

Some functional tests have a beneficial side effect of examining the requirements (for operational satisfactory, completeness and self-consistency) and verifying that specifications correctly reflect the requirements.

For safety related electronic control systems to be applied in the machinery field, it seems that deterministic functional tests provide enough testing.

Note: A deterministic test uses a reasonable number of tests cases selected according specific criterions. Statistical testing uses a large number of test cases (which is a function of the probability of fault detection and the level of confidence applied) selected basing on a probability distribution.

Functional tests, similar to structural tests, can be executed with a weak or strong coverage criteria. Obviously, the power of a criterion is not without cost.

About SW testing tools (Computer Aided Software Testing - CAST), the market survey carried out has verified that there exist a great number of tools. Nevertheless, emulators come out as the best solutions for testing dynamic behaviours of systems.

Finally, we must recognised that the effectiveness of a give strategy degrades with time because of the types of fault change. Yesterday’s elegant, revealing, effective test suite wears out because programmers and designers, given feedback on their faults, do modify their programming habits and style in an attempt to reduce the incidence of faults they know about.

6. Results of the sounding on testing methods

At the beginning of this WP 3.2, we thought to carry out a sounding between project partners mainly, in order to find out:

- if BB and WB testing strategies were known,
- how they were applied (using automatic tools for the design, execution and analysis of test cases, or manually), and
- what was the opinion about them of the partners that had had some experience.

This information would be intended to guide WP 3.2 work and define a baseline from which investigate.

The attempt has not been very successful since participation has been little (just five answers from eleven sent out), and therefore the results must be treated carefully. Despite everything, collected information has been useful.

Table 8 shows the results of collected answers in a condensed form.

Analysing the results it is noticed that:

- BB and WB approaches and different test criteria seem to be more used by test houses than manufacturers.
- Given the increasing complexity of designs, the test house testing strategy is turning unavoidably more functional or BB, and it is complemented with a few structural tests designed from an analytical study. On the contrary, designers' strategy is basically structural, focused on testing exhaustively the different components of the system.
- All the answers share the same opinion on the necessity of applying fault injection tests.
- With regard to the testing tools, it seems that the most useful are the emulators and other equipment as logic analysers, etc. In some cases, SW testing tools (static analysers) and private injectors are also used.

Table 8: Sounding results

	MF	TH1	TH2	TH3	TH4
SOFTWARE TESTING					
BB or WB strategy	Don't use these terms. V development model.	Both. BB at SW and system level. WB at specific SW and HW	Yes, they are one of the element among others in validation.	Mainly WB.	Just a few BB at system level. Rarely WB tests.
Techniques to design test cases	-	Statement coverage, function coverage. Boundary-value analysis, equivalence classes.	Critical paths testing (revealed by the analysis) Coverage are not calculated. Boundary-value analysis, syntax testing, error guessing	A tester does not have to test exhaustively a program, that is the task of the programmer. Tester should have the knowledge to evaluate programmer tests and has only to apply (redo) some tests.	Analysis (FMEA) at system level is the common practice, and WB tests on SW or HW are done only if it detects some problematic failures. Simple functional tests at system level are always done.
Automation	Manual design of test cases and execution by means of specific emulators of the µc family. Assembly level.	Manual design of cases and use of some emulator utilities (coverage analysis, trace, and macro programming) for execution. Assembly and high level language.	Static analyser support the manual test cases design. Logic analysers and emulators for execution. Assembly and high level language.	Static analyser support the manual test cases design.	Manually.
Experience	-	Combination of BB tests on system level and WB tests in SW and HW has been found efficient.	Tests are apply at unit and system level. No universal test generators are known for real-time applications. Test cases normally are defined during specification and analysis of design.	The tools help in the analysis and if necessary designing test cases.	TH4 deals more with the overall validation (applications of a system) than system validation (as a prototype).
SYSTEM BEHAVIOUR AT FAULT (fault injection)					
BB or WB strategy	BB for the product and WB to some components (e.g., ASIC)	To study the system behaviour at fault and effectiveness of fault detection mechanisms(as a complement to the theoretical FMEA).	Always.	WB approach, and not systematically.	-
Techniques to design test cases	Pin level fault injection (as BB) and internal program corruption (as WB).	Pin level fault injection (as BB) and program corruption by means of an external tool (as WB)	Pin level fault injection and simulation based fault injection at pin and function level (all as BB and WB).	Pin level fault injection (as WB), simulation based fault injection (as BB & WB) and program corruption by means of an external tool (as BB).	-
Automation	Manually.	Manual manipulation of HW components, and patching the code memory or modifying the source code with a emulator.	Both manual and automated fault injection in the HW and at system level in the peripheral components. Usually manufacturers prepare test cases specified by the TH.	All the techniques need a prior analysis. Sometimes simulation is made in an automatic mode.	-
Experience	-	Useful for embedded real-time systems. The analysis of the SW has to be supported by tests.	The private automatic injection tool is useful when the analysis is critical. These tests should be based in a previous analysis (engineering judgement).	Testing strategies, tools, etc should be based in a prior analysis. The main purpose is fault forecasting (testing the efficiency of fault detection mechanisms, etc it is estimated the future incidence and consequences of faults). Adaptation of the tool to the EUT usually is a problem.	-

NOTES: MF – manufacturer TH – test house

-

7. Conclusions of the first part

In this first phase it was intended to make a study about fundamental characteristics of functional and structural strategies, test criteria and techniques more commonly used in the fields of SW testing and fault injection, and lastly, classify these methods according with functional and structural approaches.

It was also expected that, from the test method classifications and knowledge acquired on both strategies, would be followed the arguments to select the method(s) on which to centre the second phase of studying in depth and experimentation.

It is not our intention to reproduce here the conclusions of each classification, but we do consider interesting to point out the most relevant ideas.

The first idea to underline is that a classification of BB and WB tests can be as wide as one wishes, since that the concept of BB and WB are applicable to any testing method.

In this report, it has gone farther than just the field of SW testing, traditional field of structural and functional tests, and it has considered also fault injection techniques under that perspective.

Functional strategy although theoretically results enough to achieve a complete testing, in practise it is infeasible given the large number of test cases needed (valid, invalid, combinations and sequences of data). And this is without taking into account the additional number of cases due to the special requirements of safety systems on the behaviour in case of fault.

On the other hand, **structural strategy** is inherently finite, but if it is used to reach a high degree of coverage, it shows quickly the same problem of volume of test cases as functional strategy.

Therefore, and given their complementary character, it is adopted an intermediate solution combining BB and WB tests, to derive a reasonable test strategy. Reasonable, in the sense of the relation between the number of test cases and sought guaranties are acceptable. The proportions in which structural and functional approaches must be combined depend essentially on the stage of test process (component and subsystem or integration test, HW & SW integration test, system test).

In validation stage (system test) should predominate functional approach without concerning too much on the implementation. This is a consequence of validation nature (i.e., verification of fulfilling initial specification or requirements – high level or external requirements). This idea is reaffirmed by the increasing complexity of systems which makes infeasible a structural approach that covers the whole system at this stage of the design process (at least manually).

We have said a predominantly functional strategy because in practise it is demonstrated that it is necessary to turn to a structural approach. That is due to the huge number of test cases it would have to apply, to satisfy functional tests criteria and achieve a reasonable testing, is strongly restricted by economic (cost of the tests) and time conditions (deliveries time). The most logical seems then, to limit functional tests and add a complementary set of structural tests. The latest will restrict to explore certain key features or performances revealed by the structural analysis (of HW and SW) carried out before testing. This way, it will get to keep the total number of test cases (functional + structural tests) within a reasonable order and will obtain adequate guaranties.

The increasing complexity of new systems (integrated function, processing speed, components and assembling technology miniaturisation, etc, increase) is complicating largely the execution of tests late in design (e.g., in validation). Structural tests will not get to go in depth and will be applied more and more to external layers, what supposes a process of functionalization or change to more functional testing. These obstacle, are forcing to apply design techniques that make easier further testing of circuits and programs (so called, testability technique), and direct testing from a physical to a simulation domain.

Fault injection techniques classification, shows clearly a fact that everybody knows: the almost non-existence of commercial injection tools, simulation programs excepted. Some test houses have developed their own automatic injection tools that implement usually pin level fault forcing technique or fault injection through the program. These tools allow black box tests to be applied at different levels (component, subsystem or board connector, program memory, etc), and also white box tests in specific points of the circuit or program, using the tool either in a manual or semiautomatic mode.

From these tools the more accepted and extended are those which use fault injection at pin level, since it is considered that generated type of faults simulate faithfully the real faults. This it is not the case of almost all of the rest injection techniques that are used by test houses, which even have faced with acknowledgement troubles.

The rest of techniques mentioned in the report have been developed mostly by universities and used into research programs. Most of cases, testing equipment cost makes them not to be a reasonable option for manufacturers and even for test houses.

Despite of the existence of some tools, manual physical injection at pin level, according with the sounding results, still remains the most used technique between manufacturers and test houses in this industry field, for the time being.

Obviously, it turns out difficult to plan a definition of a common strategy for testing the system behaviour in of fault. However, it would be, interesting to give some guidance on testing (type of techniques and approaches to be used, coverage requirements to achieve, places or points in the structure where to inject, operation phases to consider, etc).

A comparative study of three physical injection techniques (heavy ion radiation, pin level fault injection and electromagnetic radiation), carried out in the frame of a ESPRIT project on Computer Systems Dependability, comes to the conclusion that the three techniques are rather complementary.

It seems that pin level fault injection exercises more effectively the HW of error detection mechanisms located out of the CPU, while heavy ions and electromagnetic radiation are more appropriate to exercise the error mechanisms implemented in the system SW and at application level.

Related to **SW test methods classification,** it can see that there exists a large collection of structural and functional testing criteria. In the case of **structural tests** it has presented a partial ordering of coverage criteria in relation with their effectiveness, and has also added some guidelines or recommendations on how they use (coverage to achieve, stage in the design process, recommended applications, etc).

For **functional tests**, it is not possible to advance a classification on the relative effectiveness of criterions. However, among the considered methods could provisionally form two groups: one basic

containing the criterions that turn out effective in the majority of programmes (e.g., behavioural control flow, equivalence partitioning, boundary values, error guessing), and other group more adapted to programmes with specific characteristics (e.g., logic based testing). The same as structural tests, it is possible to talk about the coverage of program functional models, and therefore, about weak and strong tests.

Obviously, in the second part of WP 3.2 we can not keep on covering such a broad scope, and therefore, it will be necessary to restrict the research field to one of the classifications and within it to one or a few of interesting test methods.

On the work to carry out in the second part, we think that the most suitable is to centre our research in the field of fault injection. This opinion is due to the fact that, SW testing can be enough developed taking into account the research of WP 1 and our contribution in this first part. And on the other hand, because we consider interesting for the project to take advantage of the possibilities that our subcontractor (UPV) offer to us, given its background in the field of fault injection.

From the classification of injection techniques, we can deduce that the most interesting techniques with regard to its practical utility and availability, for the present time, are:

- Fault injection based on simulation technique;
- Physical fault injection at pin level, and;
- SW implemented fault injection.

This selection is also in line with sounding results, which point to that group of techniques as the most used.

So in the second part, we think of experimenting with a set of tools developed by the UPV, which implement the selected group of techniques. In the case of injection based on simulation technique there is an additional problem concerned with the need of a system description using VHDL language.

Unless some of the manufacturer involved in the project provides us with a representative sample of safety system, we have thought to practice on a prototype of distributed node fault tolerant system called DICOS “Distributed control system”, which is available at the UPV. In this case, it would not be possible to experiment with the injection based simulation technique because there is no a VHDL model of the system.

2nd Part

8. Introduction

This second part of the project is intended to do a comparative study of the techniques selected in the first part, to evaluate their contributions and weaknesses face their application in the validation process of safety related systems.

The study has to analyse in detail the different aspects of the fault injection testing (objectives of testing, parameters taking part, test result interpretation, etc.) and recommend how to handle them according with the safety requirements.

In the paragraph 5.1 “Fault Injection” was introduced that Fault Injection contributes in the validation process in two ways: helping to discover and correct design and implementation faults (fault removal) and helping to estimate the future behaviour of the system in presence of faults (fault forecasting).

The aim of Fault Injection when used for fault removal is to verify by means of an analysis essentially qualitative the adequacy of the fault tolerance mechanisms and verification procedures to the considered fault hypothesis. However, when Fault Injection is used to predict the future system’s behaviour in presence of faults it deals with evaluating the efficacy of the fault tolerance mechanisms (coverage factor, temporal characteristics of these mechanisms, etc.) and of the verification procedures.

This report is going to concentrate on the use of Fault Injection for validating the fault tolerance mechanisms.

Usually the level of abstraction used to represent or describe a system during fault injection tests goes together with the design evolution itself, this is, the more abstract models are used in the earlier stages and at the end faults are injected on the real system or prototype. In spite of this progression in the modelling of the system, from the more abstract to the more detailed, in the final validation is possible to reverse the tend and develop models with certain level of abstraction.

The possibility of using the selected techniques through the design process is depicted in the figure 13.

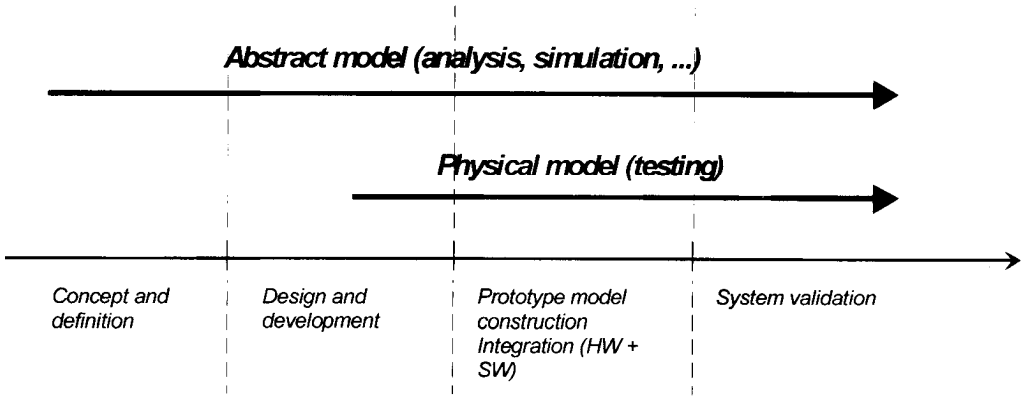


Figure 13 - Validation methods during the design process

According with the figure, analytical and simulation methods do not precise a physical model of the system and therefore they are applicable all along the design process. Opposite to this, testing methods start to be used when there are developed parts of the system.

It has to be noticed that Fault injection based on simulation (VHDL) technique finally has been omitted form the study given the lack of samples to experiment.

9. Fault injection attributes

The fault injection set of attributes to be programmed in any test, can be denominated FARM, that is, the Faults to be injected (F), the set of system Activations (A), Readouts obtained (R), and the Measures derived from these readouts (M).

The specific values o parameters given to these attributes in a test depends on several factors (injection technique, level of injection, specific characteristics of the system under test, ...), but among them partularly on the validation objective or purpose.

Fault injection attributes and validation objectives are related as fallow:
Faults set (F): If validation is performed for fault removal, faults $f \in F$ to be injected are a reduced number of the total, selected in such a way that allow to check the system fault tolerance mechanisms. The injection is deterministic. Because we study the system behaviour in presence of faults, the experiments must be reproducible. Besides, faults should be injected in a synchronised way with the system evolution, otherwise, the conclusions obtained about the functioning of the safety mechanisms will not be valid. Summarising, the main problems are, in this case, both the election of a reduced set of faults representative of the mechanism or mechanisms to evaluate, as well as the perfect synchronisation with the system of its injection.

In the case of fault forecasting, we are interested in injecting the biggest number of possible faults, and verifying the system functioning with this set of faults. So, in this case, the most important thing is the election of the set of faults to be injected (f), that must be a representative random sample from the set F. Another characteristic that should be taken into account is the capacity to realise numerous experiments in an automatic way, obtaining results that allow later to calculate the system dependability.

Activations set (A):If fault removal is made, set A must be appropriate to the safety mechanism or mechanisms that we want to verify. In this case *stress* workloads that exercise the system mechanisms could be advisable in order to check their functioning according to their specifications. On the other hand, the injection activation inputs must be oriented to make easy the injection in specific hardware or software parts of the system (*stress* inputs).

If fault forecasting is made, the A set is equal to the real workload, or the closest possible. If it is not possible to inject faults inside the application process in which the system is involved, i.e. real time systems in process control, as well as another applications that do not allow the injection outside the laboratory, the system inputs must be simulated, so a simulator of the real environment for the developed system must be made.

Readings set (R): With fault removal, readings include the system outputs that reveal the success or not in the verification of the considered fault tolerance mechanisms. On the other hand, they must include the system states reached and their outputs in case of error and/or failure in order to make a posterior diagnosis that allows the improvement of the considered safety mechanism so it can tolerate the injected fault or faults. For this reason, a fault injector that has this purpose, must imply both a logical state and timing analysis that allow to study the error propagation trajectory, as well as their consequences, from the fault activation until the failure detection.

With fault forecasting, the set of readings will be the number of detected and/or tolerated faults, and on the other hand, the error detection and/or recuperation latency time.

Measures set (M): The Measures set obtained from the readings will be, in case of fault removal, oriented to extract conclusions about the mechanisms evaluated. In case of fault forecasting, the set M would be constituted by several statistical measures that allow evaluating the dependability of the system.

10. Criteria for analysis of results and test completeness

10.1. Analysis of results

10.1.1.Coverage factor

The probabilistic value

As we said before, the actions carried out by a system when a fault occurs involve error detection, failed component identification, isolation of this component and the system reconfiguration. From each of those actions we can extract a different coverage. Then, the value of the coverage depend on the predicate (or action) to be evaluated.

Let t be the maximum time in which the predicate has to be asserted as true:

$$C(t) = \text{prob. } \{ \text{ the instant of assertion of a predicate p} \leq t\}$$

It is worth noting that C(t) is usually defective since all the faults cannot be properly covered.

$$Lim_{t\to\infty} C(t) \leq 1$$

Some limits of Fault Tolerance should be taking into account when rating the coverage factor:

1. If it is not possible to find out faults that affect the fault tolerance mechanism with respect to the fault assumptions stated during design is *a lack of error and fault handling coverage*.
2. If fault assumptions differ from the faults really occurring in the operational phase is *a lack of fault assumption coverage*, which can be in turn due to either:
   - Failed components not behaving as assumed, that *is a lack of failure mode coverage*.
   - Correlated failures, that is *a lack of failure independence coverage*.

The effect of a fault depends on both the injected fault and the system activity. In order to validate the fault tolerance mechanisms, it must be considered both the F set and the A set. Thus, in [Cukier99] the coverage factor of a fault tolerance mechanism is formally defined in terms of the complete input space defined as the Cartesian product: $G = F \times A$.

Some authors consider this descriptive association $F \times A$ as a multidimensional variable of the event space G. It must be taken into account that $f \in F$ and $a \in A$ are also multidimensional variables in function of their descriptive parameters.

If the predicate is that the system covers correctly an error, let H be a variable characterising the handling of a particular fault is a discrete random variable that can take the values 0 or 1 for each element of the fault/activity space G.

$$C = Pr\{ H = 1 \mid g \in G\}$$

Let *h(g)* denote the value of H for a given point and *p(g)* be the relative probability of occurrence of *g*, the coverage factor is defined as:

$$C = \sum h(g)\, p(g) \quad \forall g \in G$$

Estimation of the coverage function

Analysing a determined predicate; let $\varepsilon_i(t)$ denote the random variable defined by:

$\varepsilon_i(t) = 1$ if the predicate is observed in [0,t]; 0 in other case.

Let N(t) be:

$$N(t) = \sum_{i=1}^{n} \varepsilon_i(t)$$

Being n the number of effective injected faults.

Then, the basic estimation of coverage results as [1]:

$$\hat{C}(t) = \frac{N(t)}{n}$$

The most accurate way to determine the coverage would be to submit the system to all $g \in G$ and to observe all outcomes. However, such exhaustive testing is only possible under very restrictive hypotheses. For this reason, coverage evaluation is in practice carried out by submitting the system to a subset of $\{f \times a\}$ occurrences $G' \in G$ obtained by random sampling in the space *G* and then using statistics to estimate the coverage.

Considering a pair $\{f \times a\}_i$, let be $\pi(\{f \times a\}_i)$ its occurrence probability p(g) on the system; let be $\tau(\{f \times a\}_i)$ its selection probability inside our sample; let be n de number of experiments to be considered in our sample; let be $h(\{f \times a\}_i)$ the discrete value [0,1] of this pair depending on the accomplishment of the predicate.

Then, we can consider the unbiased point estimator is given by [Powell 93] [2]:

$$\hat{C} = \frac{1}{n} \sum_{i=1}^{n} h(\{f \times a\}_i)\ \pi(\{f \times a\}_i)/\tau(\{f \times a\}_i)$$

With representative sampling, we have $\pi(\{f \times a\}_i) = \tau(\{f \times a\}_i)$, then [3]:

$$\hat{C} = \frac{1}{n} \sum_{i=1}^{n} h(\{f \times a\}_i)$$

10.1.2.Latency times

Many systems that analyse the system behaviour in presence of errors uses the coverage factor in order to specify the system probability for detecting an error. Thus, the designer shall carry out procedures to recover the system. Actions taken by the system involve from error detection to system recovery. Each one of those actions have to be treated as fast as possible, before other error could appear into the system. Thus, most designers include latency detection mechanisms in order to evaluate the time needed by the system to perform an action.

As we saw before, each action could be considered as a predicate. The latency for a specific predicate, usually is calculated by the mean (basic estimation) of those latencies derived from each individual pair $\{f \times a\}_i$.

One of the most representative distributions, for the latencies included into a sample, is the Normal Distribution.

Then, it should be considered the sample asymmetry, because, in case of being a positive or negative asymmetry sample, it could be more appropriate the use of the median as the basic estimator instead of the mean, due to possible values long away from the mean.

There exist more specific latency estimators in the literature.

10.2. Test completeness

10.2.1.Confidence interval

As in most of statistic measures, the confidence interval of the sample must be calculated in order to define the completeness of the experiment and the accuracy of the obtained value.
It is recommend to obtain a confidence percentage of 95% or higher.

10.2.2.Stratification

To complex systems, it could be difficult to obtain a balanced sample. Then, the stratification can be a method to reduce the number of inputs while offers the possibility to be focused into specific means within validation.

Stratification means that a sampling space *G* is considered as partitioned into *M classes* or *strata*:

$$G = \bigcup_{i=1}^{M} G_i \quad \forall i, j, i \neq j, G\,i \cap G\,j = \varnothing$$

In a stratified fault-injection campaign, a fixed number of experiments are carried out in each class. Then, the coverage factor with M classes will be:

$$C = \sum_{i=1}^{M} \sum_{\forall g \in Gi} h(g)\, p(g)$$

And the coverage estimator with representative sampling (see [3]):

$$\hat{C} = \sum_{i=1}^{M} \frac{1}{n_i} \sum_{j=1}^{ni} h(\{f \times a\}_j)$$

11. Description of selected fault injection techniques and tools

In this section we describe in detail the techniques and tools selected in the first part of the project, which now are going to be used to perform a practical case study.

The two injection tools (AFIT and SOFI) used in the experiments has been developed by the Fault Tolerant Systems' Group (GSTF) in the Department of Computer Engineering of the Technical University of Valencia.

AFIT is a new high-speed hardware implemented fault injector at pin level, a prototype of which with many of its internal building blocks has already been built. Considering the difficulties involved when injecting faults in modern processors, which are very fast, a high-speed physical injector has been developed (currently up to 40 MHz). As injection will be applied on already built systems, pin level injection with the forcing technique has been chosen. SOFI is a software implemented fault injector able to inject faults into the code and data segments.

During prototype phase, appropriate techniques for experimental evaluation of fault tolerance are Hardware implemented fault injection (HWIFI), which is a physical fault injection at pin level, and Software implemented Fault Injection (SWIFI), that means, the deliberate insertion of faults into an operational system to determine its response. At the *system level*, a logic fault is manifested as an error in the program being executed by the system.

Two types of error may be distinguished. A word error occurs when a computer word (data word or instruction) is altered by a fault. A description of the alteration is called the damage pattern of the fault. A logic error occurs when an individual logic variable, which is not part of a structured word, is altered by a fault. Examples of such variables are the various control signals in a computer or the corruption of the CPU's internal registers. Thus, a logic error alters the algorithm being executed in some undesirable manner.

A logic fault may be seen *at hardware level* or *at software level*. At hardware level means the physical implementation of a fault, but at software level the fault is related to the functional inconsistencies that it produces.

The most common hardware an software fault models are summarized in table 9.

Table 9: Types of fault model

Hardware fault model	***Software fault model***
Open line Bridging	Storage data corruption (memory, register or disk)
Spurious current Power surge	Communication data corruption (bus, communication network)
Bit flip Stuck at	Manifestation of software defects (machine level or higher levels)

Either SWIFI and HWIFI are able to generate determined types of word and logic errors. Those errors derive from faults injected with a hardware fault model. When this fault is co-ordinated with an activation input forming a pair {f × a}, then we obtain a software fault model.

11.1. Physical fault injection at pin level

11.1.1.Attributes

Either with active probes and socket insertion belong to the Physical fault injection at pin level.

Those techniques are characterised by the FARM attributes:

Faults set (F): Fault injection is reached by introducing faults into the system hardware during an injection campaign. These faults must be representative faults to those occurred during the operational phase of the system.

The set has to be defined either by the injection goal in terms of fault removal or fault forecasting, and the system characteristics in terms of reproducibility and observability.

The parameters that characterised each fault are Location, Type and Temporal characteristics.

> *Location*: Refers to the system pines where an alteration (fault) of the logical value or physical real value will be produced. These injection pines can belong to any system component, although according to the target of the campaign, they will be restricted to a determined subset in case of fault removal or they will randomly extended in case of fault forecasting.
>
> *Type*: It refers to the fault model. Most common until now have been: stuck-at zero, stuck-at one, logical or physical bridging, bit flip and open line at hardware level. Using physical fault injection at pin level with active probes, the most used is the stuck-at zero and one.
> At software level, PFIAPL at pin level is directly implied with Communication data corruption, but indirectly Storage data corruption is also evaluated by means of corrupting instructions sending through the busses or the communication network. These instructions came into or go out of specific storage targets (memory, registers or disk).
>
> *Temporal characteristics*: Refers to the temporal model. We can distinguish three types: permanent, transient and intermittent faults.
> The first one affects indefinitely from the fault activation time. The transient one affects from the fault activation time throughout a predetermined duration. With this injection technique, transient faults are totally independent of the system states from the fault activation time. They affect in the same location, a continuous time not having into account the following system states.
> The intermittent faults are defined as a transient fault series that appear regularly in the same location from the fault activation time until the end of the experiment.

Activation set (A): Defining as activation input ($a \in A$) the state that the system has reached when injection is activated for each experiment.

An experiment consists of a Cartesian pair of $F \times A$ resulting a reading ($r \in R$).

The trigger that actives the injection may be temporal or spatial. A temporal trigger actives an injection based on temporal parameters of the workload running. A spatial trigger actives the injection depending on a state or condition of the system.

The advantage of using a temporal trigger is that the more frequent is a workload state in execution time, more probable is an injection in this state. That is, the most frequent states are the more injected.

The advantage of using a spatial trigger is the possibility of injecting previously considered states, independently of their occurrence frequency during execution time, being possible to easily evaluate critical states not very frequents.

Readings set (R): When a fault is injected in the system, it must be guaranteed if the fault is effective or not. Depending on the fault model, an injection can be non-effective. This is the case of the stuck-at zero and one, logical and physical bridging faults, i.e. if we try to inject a stuck at zero fault in a pin which logic value is just zero.

The cases of non-effective injection should be recognised and removed from experiments.

If the injection tool lacks of appropriate mechanisms for the identification of the fault effectiveness, it is possible to appeal to a *Golden-run*, that is, a previous execution without faults that is useful to compare with the executions with faults.

Then, the readings derived from the experiment have to refer as well to the fault effectiveness (in order to reject readings derived from a non-effective injection), as the system behaviour in presence of this fault.

From the first ones, unsuccessful injections are rejected, while from the second ones, readings for fault forecasting and system mechanisms verification are obtained.

Measures set (M): Most of fault tolerant systems designers use the known parameter called **coverage** to specify the probability of systems to detect an error. This will allow executing the appropriate procedures to recover. The actions to take by a system go from error detection until system reconfiguration, passing through the affected component identification and its isolation.

Different coverages are extracted from different actions. Also, each one of the mentioned actions must be treated as fast as possible before another error can overload the error treatment mechanisms. For this reason, a lot of models incorporate **latency times** distributions, that is, necessary times for detecting, identifying, isolating and reconfiguring the system. Besides, small variations in coverage and latency can greatly affect dependability. Thus, these parameters should be estimated based on data from the real system rather than approximated.

11.1.2.AFIT (Advanced Fault Injection Tool)

11.1.2.1.Architecture of the tool

In figure 14 can be seen the injector with its modules. These modules are:

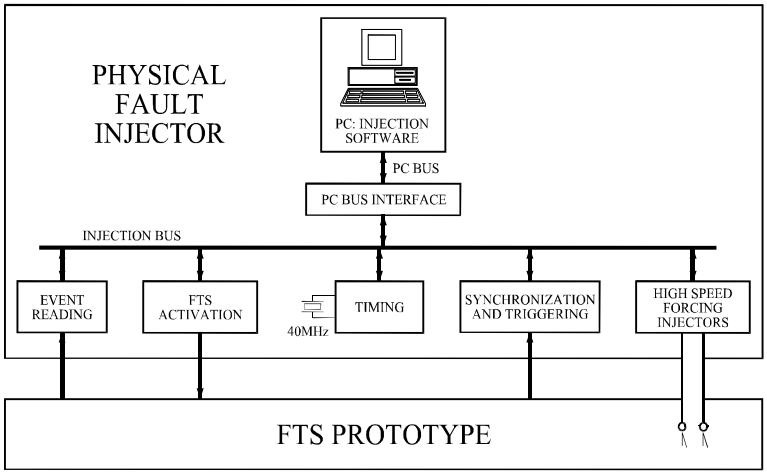


Figure 14 - AFIT injection tool modules.

- PC interface. The injector is connected to a PC through an ISA bus interface.
- Timing: This module generates the clock frequencies used by the injector. These frequencies are programmable up to a maximum, currently 40 MHz. Furthermore, it provides the Activation of the Injection signals (AI) that activate the High Speed Forcing Injectors. The shapes of the AI signal depend on the number of faults to be injected, on each fault whether it is transient, intermittent, or permanent.
- Synchronisation and Triggering: This module determines the beginning of the injection experiment. It is formed of two parts, the Triggering Word and the Programmable Delay Generator. After a Triggering Word has been programmed, and the injection has been enabled, this module samples some of the FTS signals (address, data or control signals) in order to determine its internal state. When this state matches the programmed triggering word, the Injection Triggering Signal is generated and activates the Timing module, which controls the fault injection.
- FTS Activation: The main function of this module is to prepare the FTS for each injection experiment. Reset signals have been added to the injector in order to restart the system at the beginning of each injection.
- High Speed Forcing Injectors: They physically inject the faults into the FTS. When the injector receives the AI signals, the injector activation logic deviates the signal to one of the transistors implied into forcing depending on the injection type: stuck at one or zero.
- Event Reading: This module determines the response of the FTS after the fault has been injected. It has been implemented using a logic analyser, and captures the results derived from each injection.

The fault injection attributes implemented in or by AFIT are:

Faults set (F): Location is referred to the system pins. We can use 2-multiplicity inside one integrated component adding two more anywhere. The type of fault is stuck at one or zero at hardware level, and as temporal characteristics we can inject permanent, intermittent or transient faults (from 25ns to 3s) and the fault activation time depends on the selected trigger.

Activation set (A): Usually, we are using a logic analyser to sample the system signals. Then, activation can be carried out by temporal (programming a random delay from execution start) or spatial trigger (programming a triggering word).

The workload depends on the system under test and we decide the activation inputs with a spatial trigger, according to the purpose of fault removal (deterministic or statistical testing). With fault forecasting, we use temporal trigger.

Readings set (R): Readings derived from one injection are both the errors detected by the system (hardware and software tolerance mechanisms) and latency detection and/or recovery times.

Measures set (M): Extracted measures from an injection campaign identify the failure modes, the activation percentage of each tolerance mechanism, the coverage and the latency times.

11.1.2.2.Test sequence

A fault injection experiment is divided in the following steps:

Configuration: In this first stage the injector and the system under study have to be configured. The appropriate connections between the injector and the system have to be made so that faults can be injected at the desired places (specifying the Location of the F attribute). From the inputs and the synchronisation signals in the system (A attribute), the trigger word (the desired time at which to inject), the initialisation inputs and the signals that have to be analysed (R attribute) have to be determined.

Injection: Before each experiment, it will be necessary to take the system to a known initial state. Once the injector is configured with the FAR attributes, this stage begins with the injection enabling, what activates the synchronisation and triggering module. When the triggering word becomes true the timing module is activated. This module generates the activation signals for the active probes. From this moment, the Event Reading begins, monitoring the signals specified at the R attribute, until the fault detection or recovery is activated in the system. When not detected or not tolerated faults are injected in the system or the fault is not effective, the expected activation in those signals can never happen, so there is a timeout mechanism to stop the injector monitoring the system.

Results logging: When the injection stage finishes, the information files with obtained states and generated times for each experiment have to be written. These files contain the events happened and the times for each of them from the time the injection is activated to the recovery time of the system (or the timeout in case it does not recover).

11.2. Software implemented fault injection

11.2.1.Software implemented fault injection: attributes

Software implemented fault injection injects faults by means of changing the contents of memory or registers. It is limited to those locations reached by software. This technique is used to both emulate the occurrence of hardware faults and emulate the occurrence of software faults.

According to the FARM attributes, we can describe this injection technique as follow:

Faults set (F): The parameters that characterised each fault are Location, Type and Temporal characteristics.

> *Location*: A great number of the functional units that compose the system can be accessed via software (memory, processor, buses, clocks, etc.). Nevertheless, there are points that are not reachable (parity codes in the memory and buses, check mechanisms, etc.)
> With this technique, we can too refer to experiment multiplicity as the number of faults that we inject in the system at a time. By software fault injection, injecting more than one fault and having them controlled is very easy. Nevertheless, it is usual to inject only a fault to study their consequences.
>
> *Type*: The fault model at hardware level is the bit flip (sometimes stuck at is also used). At software level, it is directly implied with Storage data corruption and Manifestation of Software defects.
>
> *Temporal characteristics*: The injected faults can be permanent, transient and intermittent. In most cases, real faults are transient, so the main goal is to produce this type of faults.
> The injection duration has relation too with the selected abstraction level: from machine cycles, to the low level, instructions, and microseconds, until bigger units.

Activations set (A): There exist two cases: Firstly, *off-line* injection, where faults are injected before the system loads the application. Usually, to realise this type of injection, the executable code is modified, substituting one or more instructions for other corrupted ones.

Secondly, runtime injection. The runtime injection types are a) time-out: using an internal timer where to program a random time delay from the application's running start. When the time expires, the injection is activated by means of the interrupt handler of the timer; b) exception/trap: whenever certain event or conditions occur, the injection will be activated. When the trap executes, an interrupt is generated that transfer control to an interrupt handler; c) code insertion: instructions are added to the target program that allows fault injection to occur before particular application's instructions.

Readings set (R): Refers to the set of readings derived from each experiment. With this injection technique it is also possible a previous *Golden-run* as a comparative model with posterior experiments.

In experiments using this technique, it is possible to obtain latency times concerning to fault detection by the system. These latencies, usually measured using resources of the own system, must be debugged because of the overload introduced by the same software injector, specially in case of performing a runtime injection where an internal injector is required.

Measures set (M): It must be applied the same parameters that in subclause 11.1.1. Physical fault injection at pin level.

11.2.2.SOFI (Software Fault Injector)

11.2.2.1.Architecture of the tool

The tool is composed by three blocks: the injection agent, the activation agent and the supervisor (see figure 15).

The activation agent is in charge of determining both, the fault activation time and the fault characteristics. The injection agent is aimed to inject the faults and finally, the supervisor has to perform the functions of monitoring the system reactions in presence of faults and latency times to each injection.

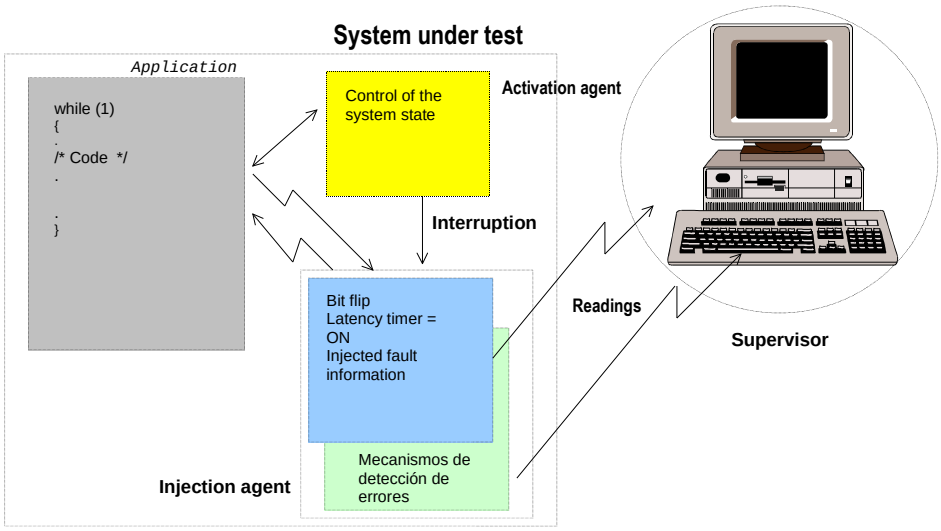


Figure 15 - SOFI Architecture

The fault injection attributes implemented in or by SOFI are:

Faults set (F): SOFI include logic emulation of physical faults, because of both internal and external influences, i.e., corrupting the executable program or changing the contents of the registers by a special fault injecting routine.

Referring location, it must be determined either the injection is in the code or in the data segment. In the first case, injection simulates the corruption of a memory cell in the code space of the next executable instruction, the fault effectiveness is guaranteed. In case of the data segment, it is the injection agent who decides what bit flip would be applied inside the data limits used by the application. Inside these limits the user stack can also be injected. With the used microcontroller, apart from the system stack the application has its own stack, to store local variables too large to fit in registers and the ones needed to implement recursively. Fault effectiveness in this segment is not guaranteed, i.e. if the injected variable is overwritten before being read from the beginning of the injection.

As we said, the type of faults is the bit flip due to the increasing in injection effectiveness. While temporal characteristics, the faults may be transients with variable duration or permanents.

Activations set (A): SOFI implements a runtime injection. The activation inputs are selected at random using a temporal trigger.

Due to the activation agent that maintains same control over the principal microcontroller states, it is possible to perform a filtering over the activation inputs.

Readings set (R): Reading obtained from the prototype (DICOS) are those sent by the internal hardware detection mechanisms, and from the watchdog communication control. Adding to those reading concerned with the evolution of the systems after an injection, the system is prepared to obtain latency detection times.

Measures set (M): Extracted measures from an injection campaign identify the failure modes, the activation percentage of each tolerant mechanism, the coverage and the latency times.

11.2.2.2.Test sequence

As every SWIFI tool, SOFI has to be integrated into the system under test. It is an internal tool.

In this case the system under test is a prototype of a Distributed Control System (DICOS) developed by the GSTF of the Technical University of Valencia.

A node of DICOS has been chosen to be tested either with SOFI as with AFIT.

The DICOS node may have got different architectures, among them we have duplex and dual architectures. In the analysed experiments a node is composed of two microcontrollers. One of them (Siemens C167CR) is dedicated to the execution of the control algorithms. The second microcontroller (Intel 251) is meant to be in charge of communications with the rest of the system, and to work as the main controller's watchdog processor. Communication between both controllers is established through a double port memory (DPM). Both received and transmitted messages as well as the signature from the application to the watchdog controller for the flow checking are written through this memory.

As it can be appreciated in figure 16, a personal computer is in charged of monitoring the generated systems results from each injection (the supervisor).

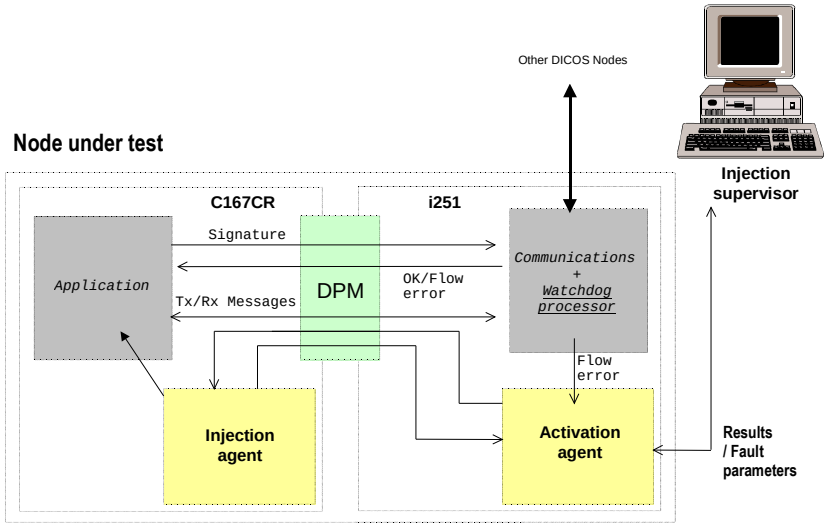


Figure 16 - Implementation of SOFI into DICOS

The activation agent is integrated inside the watchdog processor. The watchdog processor keeps information about the system state, it knows when the control application is in a safe state, when it is a working state and when it is a recovery state. So this information is very useful for deciding when to activate the injection. Even with a temporal trigger can be used to decide whether or not to activate the injection.

The activation trigger interrupts the application controller. This interrupt starts the injection agent. The agent's function is to generate the memory bit-flip and to activate the mechanisms that obtain the detection latency time. It has also to deactivate transient faults when the time expires.

An important matter of such injection agent is to be in charge of preparing the system after each experiment. This way, before an experiment begins, the system has to start from a known state. For that, the activation agent sends a restart command to the injection agent. From that command, the injection agent has to reload the application under test. This process is the following: When the executable version of the program is generated, it is developed in such a way that the original copy of the application is kept at the lower memory addresses, which is only readable. When the system starts, the injection agent starts working, and, before running the application copies it addresses to the memory where it will inject, in this case RAM. When the experiment starts and the agent injects, these addresses are modified. When the activation agent sends the restart, a software reset is forced to reload the application from the original copy, so that the following injection starts from a known state.

If either internal or external mechanisms are activated, or if a non safe failure is detected, the information related to the system response in presence of a fault is sent through the watchdog processor to the monitor, which stores each injection results. These results include detection latency.

We should have said that the injection agent has to be mounted with the application under test. This agent needs memory for about 3KB.

As a last information about the injector, it is important to remark that it is able to inject more than 1000 failures per hour, what is a good throughput compared with other injectors.

12. Case study

12.1. Description of the prototype (DICOS)

The study performed with DICOS by both software and hardware implemented fault injectors has as objective the evaluation of the system behaviour in presence of faults. Then, fault forecasting is the goal of the injection campaign.

A Node of DICOS has the architecture shown in figure 17.

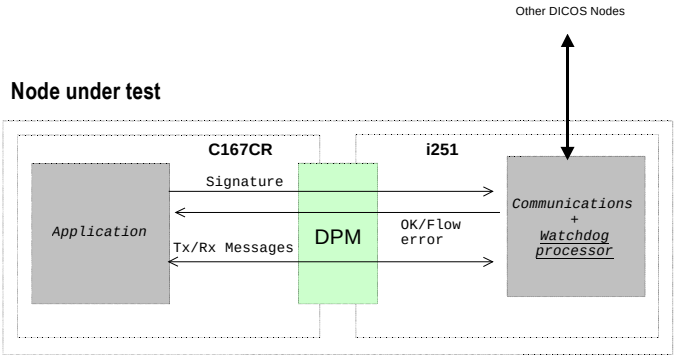


Figure 17 - DICOS architecture

As we said in the previous section, the DICOS node may have got different architectures, among them we have duplex and dual architectures. In the analysed experiments a node is composed of two microcontrollers. One of them (Siemens C167CR) is dedicated to the execution of the control algorithms. The second microcontroller (Intel 251) is meant to be in charge of communications with the rest of the system, and to work as the main controller's watchdog processor. Communication between both controllers is established through a double port memory (DPM). Both received and transmitted messages as well as the signature from the application to the watchdog controller for the flow checking are written through this memory.

The fault tolerance mechanisms of one Node are:

- Internal hardware mechanisms of the C167CR. These mechanisms detect undefined operation codes, illegal word operand access, illegal instruction accesses, stack underflows, stack overflows, etc.
- Control flow mechanisms. They are implemented within the watchdog processor.
    a) Signal control. To each specific application running into the main processor, it has been determined a serial of signatures to be sent to the watchdog processor. If one of those signatures is received with a wrong signature number, the signal control mechanisms will detect an error into the node.
    b) Temporal control. In the same way, if a maximum pre-established time expires before a signature is received into the watchdog processor, a temporal control mechanism will detect a error into the node.

Due to DICOS has been developed as a Distributed system, a fail-silence mode (the system deals with a failure by silencing their outputs avoiding to show to other nodes a behaviour known as the "babbling idiot" syndrome) is the most appropriated behaviour for each node.

Referring to the workload used in our experiments, we use a test workload that performs a LU algorithm. It has a strong computational grade.

The control signature passing messages have been introduced into the application.

In fault forecasting we differ between ordinal evaluation and statistical evaluation.

12.2. Ordinal evaluation

It is aimed at identifying, classing and ordering the detected errors by the system.

From each experiment, dealing with a pair {f × a}, depending on the system behaviour, we will obtain a reading classed into:

- Internal error: detected by the internal hardware mechanisms inside of the principal microcontroller.
- Control flow errors: detected by means of the watchdog processor in two ways: by a signature control and by a temporal control.
- Non-detected errors: those errors non detected by either internal or communication control mechanisms, and those that deviate the system from fulfilling its correct function.
- Non-affected errors: those errors that due to the intrinsic hardware and software redundancy of the system does not affect its functionality.

These readings are obtained in the same way either with SOFI that with AFIT.

Descriptive attributes of our experiments:

Location: Injections have been carried out over the code segment with software implemented fault injection and over the data bus with hardware implemented fault injection.

When detecting the trigger, SOFI injects a fault over the next instruction to be fetched after the routine service of the injection agent. While AFIT injects in the data bus during a reading access to code memory.

Both injections cause the same effect: decode an injected instruction by the microcontroller.

Type: At hardware level, bit flip model is usually used with Software implemented fault injection. Stuck at model is typically used with Hardware implemented fault injection.

At software level, we have performed Storage data corruption. The corrupted data referred with memory.

Temporal characteristics: We have injected transient faults during only one clock cycle. Then, there are simple and simplex faults. The fault activation time depends on the random time programmed to each injection in order to active the temporal trigger.

Activations: It has been used a temporal trigger with both SOFI and AFIT, that means, it has been selected a random time from the application start. Finishing this time, the fault is injected. Thus, activation inputs are randomly selected.

Activations corresponding to both tools have the following injection frequency inside each application routines (see figures 18 and 19):

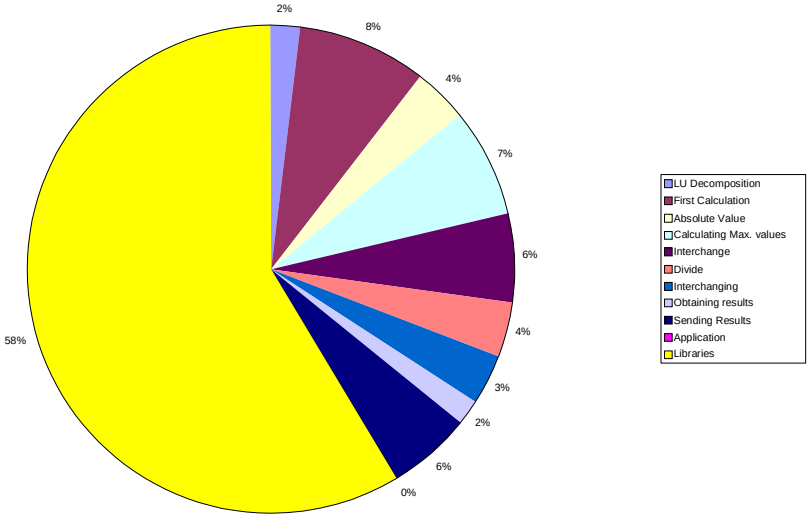


Figure 18 - AFIT injection frequencies

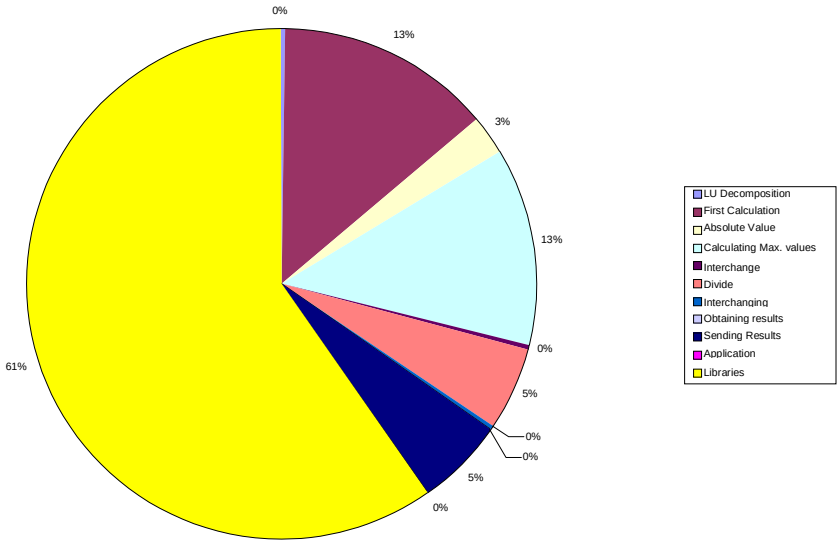


Figure 19 - SOFI injection frequencies

Readings: The monitor saves one reading from each injection. The reading correspond with the detected error by the system among all possibilities classed before.

Measures: After the injection campaign, readings have been translated into a special spreadsheet. We have extracted de percentage of activation of the different **kind of errors (see table 10):**

Table 10: Percentage of activation of the different type of errors

%	*Software Implemented*			*Hardware Implemented*		
Non-affected errors	65.8331666			64.3297697		
Signature control	0.960192			2.5493421		
Temporal control	3.9207842			4.1324013		
Internal detection	8.1416283	illegal instruction access	39,31%	8.7787829	illegal instruction access	50,82%
		undefined operation code	48,40%		undefined operation code	34,89%
		illegal word operand access	9,83%		illegal word operand access	13,58%
		Protected Instruction	2,46%		Protected Instruction	0,47%
		Stack underflow	0,00%		Stack underflow	0,23%
		Stack overflow	0,00%		Stack overflow	0,00%
		Non-defined access	0,00%		Non-defined access	0,00%
Non-detected errors	20.9641928			20.2097039		

A sample of about 5000 effective-injections has been generated. SOFI samples 5000 injections (effectiveness of 100%). AFIT has performed 5000 double experiments. Each experiment has been reproduced twice with the same pair {f × a} excepting the fault model: first it is used the stuck at one model, secondly, it is used the stuck at zero model.

On this way, with double experiments AFIT has an effectiveness of 99.8%.

From the obtained measures we can see on one hand, a comparative between both tools. AFIT and SOFI are able to emulate a similar set of pairs {f × a} over DICOS when our objective is data corruption on memory.

A similar set of pairs should derive in a similar set of measures. As we can see on the results, the tolerance mechanisms are exercised similarly. Thus, when two or more tools can emulate a similar set of pairs, the designer can perform the injection campaign with the tool, which will allow more simplicity from any point of view.

But, an injection campaign must be defined depending on the system's validation requirements. Designer should decide the most appropriate set of pairs {f × a} and use that tool which will be able to generate them.

On the other hand, the percentage of non covered errors is not as low as could be desirable for a high dependability system.

12.3. Probabilistic evaluation

It is aimed at evaluating in terms of probabilities the coverage.

Using the coverage factor equation depicted before, we have translated it in terms of the number of covered errors by our system.

The occurrence probabilities are implicit in the results due to the use of a temporal trigger (the most frequent activation inputs, the more injected).

Then, with representative sampling, we have $\pi(\{f \times a\}_i)$ (the occurrence probability) = $\tau(\{f \times a\}_i)$ (the selection probability). We have use the coverage estimator:

$$\hat{C} = \frac{1}{n} \sum_{i=1}^{M} h(\{f \times a\}_i)$$

n = Total of effective-injections,
$h(\{f \times a\}) = 1$ if the predicate *Covered error* is assertive for one pair $\{f \times a\}$; 0 in other case:

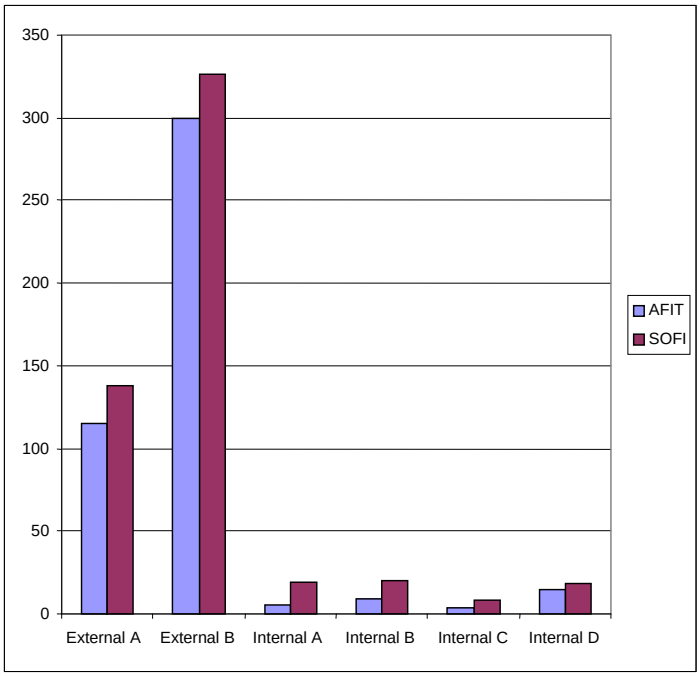
Coverage estimated value = Number of Covered errors / Total of effective-injections.

%	*Software Implemented*	*Hardware Implemented*
Coverage	78.8557712 ± 1.1279882	79.7902961 ± 1.1440602

As we can see, the confidence interval has been extracted from the sample with a confidence percentage of 95%.

The percentage of errors to be covered will depend on the safety category allocated o the system.

Finally, we have extracted the median latency for each sample (AFIT and SOFI injection campaigns). The sample corresponds with an asymmetric normal distribution, and the estimated values are shown in figure 20.



External A = Signature control flow detection.
External B = Temporal control flow detection.
Internal A = Illegal word operand access.
Internal B = Protected instruction access.
Internal C = Undefined operation code.
Internal D = Illegal instruction access.

Figure 20 - Detection latencies with SOFI and AFIT

From the results obtained, we can deduce the necessity of studding the overload introduced by a runtime software implemented fault injection tool.

13. Conclusions of the second part

This part of the project has been reduced to the experimentation with a single prototype, developed at the university of Valencia, because of the cooperation initiated with one of the partner manufacturer finally did not succed.

For this reason, the results achieved do not allow us to come to definite conclusions on the efectiveness and yield of the selected fault injection techniques and, in general on the degree of recommendation of fault injection testing as a validation method for safety related systems.

In spite of the limitations expressed, the studies made on the two fault injection techniques (Physical fault injection at pin level and Software implemented fault injection) and the testing campaign carried out using two injectors tools which implement those injection techniques (AFIT and SOFI) let us to do some statements.

To begin with, every fault injection technique (tool) is able to emulate a specific subset of faults f, and therefore it has a restricted usefulness concerning the coverage of all potential faults, i.e the set F.

Physical fault injection technique at pin level, due to its capacity to inject faults at the component pins is more suitable for emulating faults on the control and comunication lines, either internal, i.e. at system's buses, or external, i.e. at peripherals. Because of this technique is implemented by an external tool, it does not overload the application under test and although its depth is confined to the component pins it allows injecting faults without halting or interrupting the application. This supposes an important advantage. Furthermore, this technique has a good controllability and allows to reproduce experiments and extract from them precise measures about latency times.

Software implemented fault injection involves directly treatment with application structural data. Then, it is more suitable to emulate faults in storage units (registers, memory or disks) and in the internal functional units of the CPU, meaning a higher injection depth (more reachability) and, consequently, it is more recommended in systems with high integration density. However, being a technique implemented inside the workload it involves an overload over the application under test, but it is easier to be developed in systems with high clock frequency. It allows a good controllability, a reproducibility of experiments and offers a good portability as well.

Thus, both of them may be considered complementary techniques. It will be a designer decision to value their advantages and disadvantages when they have to be used with a specific system, because those advantages and disadvantages are totally dependent on the functional characteristics of this system.

The experimentation confirms the validity of fault injection as an interesting testing method for electronic complex systems' validation without being possible to confirm or refute the statement of IEC 61508-2 which classifies fault injection as a recommended (R) method requiring different efectiveness levels depending on the safety requirements.

Referring to the injection campaign planning, the main specification is to define wether the injection concerns one or several mechanisms, or it is focused on the whole system evaluation. Thus, in order to aim this objective, the faults set (f) to be injected is the first requirement of the test and this set must be complete enough to ensure the designer the accomplishment of this objective. When f is defined using both fault injection techniques, they are able to measure coverage (and latencies) of the highest coverage fault detection mechanisms, helping for the assesment of system category.

As we said before the selection of the tool will depend on the faults set to be covered. The own characteristics of each particular tool made it more or less suitable for a designed campaign. Although, as we have seen in our study, there exist at least a sub-set of faults (data corruption on memory program) covered by both of them. Then, the designer is free to take the decision of using the tool which fits closer to the system necessities.

14. Conclusions

No single testing approach is enough. Black box and white box approaches are two different ways of exploring a system, which complement each other.

BB test strategies are based on requirements. Tester is only interested in finding circumstances in which the system does not behave according to its specifications, for this reason they are also called behavioural or functional test strategies. In contrast, WB test strategies are derived from the structure of the tested object, and are also called structural test strategies. Hybrid strategies combine behavioural and structural strategies.

From the definition one can easily deduce that a pure structural strategy in no way guarantees that a system matches its specification due to possible misconceptions and omissions errors produced during the design and development. Whereas a pure behavioural strategy can in principle achieve a complete testing, but not in practice given the large number of test cases needed.

Validation deals with determining the conformity of the prototype or final system with its specifications (testing big components and system), so it is clear following the definition given above that behavioural strategy will be dominant. However, there might be reasons that justify the application of structural tests to specific parts, for instance, a little intervention by the validation team during the design verifications.

All test techniques or criteria have built-in assumptions about the nature of faults. However, when designers respond to testing by reducing such faults and using new design languages and tools, it follows that the new products improve, and then effectiveness of previous tests get reduced. So testing techniques will need to be updated to catch the new faults.

The study does not come to order the existing test criteria in relation with the different safety performance levels, since the degree of recommendation of a testing criteria depend mainly on the nature of the product under test. However, all criteria based on exploring anomalies in the control flow and timing seem essential given the strong control flow and time dependency of machinery control systems.

In fault injection testing is also possible to apply functional and structural approaches during the definition of the set of faults to be injected and the set of activation data to be apply during the injection, in order to test the measures to control failures. Moreover, the study has considered the relation between the objectives of fault injection (fault removal or fault forecasting) and the use of functional or structural approaches.

Among the existing injection techniques the study has revealed that fault injection at pin level and fault injection implemented by SW are the most interesting techniques taking into account an effectiveness/availability relation.

Each injection technique is able to emulate a set of specific faults, and therefore it has a restricted usefulness concerning the coverage of all potential faults.

The results of experiments reported in the literature and our own experiences show that fault injection at pin level is a good option to emulate permanent faults in control and communication lines, both internal (system buses) and external (periphery). This technique does not overload the system because is applied with an external tool (injector), and performs an injection process without halting or interrupting the system (dynamic injection). In addition this technique has a good controllability and allows reproducing experiments as well as obtaining measures about latency times.
On the other hand, SW implemented fault injection involves a direct processing with application structural data. It is more suitable to emulate faults in the CPU storage and functional units, though it is assumed that some injected fault models are able even to emulate faults beyond the CPU, that is, in the peripheral circuitry. The ability to inject faults inside the CPU means that SW implemented fault injection has a higher injection depth (more reachability) than previous technique, so it will be specially recommended in systems with high integration density.

However, being a technique implemented inside the workload it involves an overload over the application program but is easier to implement in high clock frequency systems. It allows a good controllability and reproducibility of experiments, and offers a good portability as well.

REFERENCES

[Arlat 90]: J. Arlat, “Validation de la Sûrete de Fonctionnement par Injection de Fautes. Méthode - Mise en Oeuvre - Application”. Thése présentée à L’Institut National Polytechnique de Toulouse. Rapport de Recherche LAAS Nº 90-399, Décembre 1990.

[Arlat 90a]: J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, “Fault injection for dependability validation: A methodology and some applications”, IEEE Trans. Software Eng., vol. 16, pp. 166-182, Feb.1990.

[Arlat 93]: E. Jenn, J. Arlat, M. Rimén, J. Ohlsson, J Karlsson. “Fault Injection into VHDL Models: The MEFISTO Tool”. LAAS Research Report Nº 93 460. December, 1993.

[Avresky 92]: D. Avresky, J. Arlat, J.-C. Laprie, and Y. Crouzet, “Fault injection for the formal testing of fault tolerance”, in Proc. 22nd. Int Symp. on Fault Tolerant Computing (FTCS-22), (Boston, MA, USA), pp.345-354, IEEE, 1992.

[Aylor 92]: J. H. Aylor, R. Waxman, B. W. Johnson, and R. D. Williams, “The Integration of Performance and Functional Modeling in VHDL”, ch. 2, pp. 22-145. In Schoen [79], 1992.

[Barton 90]: J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, “Fault injection experiments using FIAT”, IEEE Transactions on Computers, vol. 39, pp. 575-582, Abril 1990.

[Beizer 90]: B. Beizer, “Software Testing Techniques”, Van Nostrand Reinhold, 1990.

[Beizer 95]: B. Beizer, “Black-Box Testing. Techniques for Functional Testing of Software and Systems”, John Wiley and Sons, 1995.

[Boue 97]: J. Boue, P. Pétillon, Y. Crouzet, “Mefisto-L: A VHDL-based fault injection tool for the experimental assesment of fault tolerance”, in Proc. 28nd. Int Symp. on Fault Tolerant Computing (FTCS-28), (Munich, Germany), pp.168-173, IEEE, Junio 1998

[Chillarege 89]: R. Chillarege and N. Bowen, “Understanding large system failures - a fault injection experiment”, in Proc. 19th. Int. Symp. on Fault Tolerant Computing (FTCS-19), (Chicago, MI, USA) pp. 356-363, IEEE, 1989.

[Clark 95]: J. A. Clark and D. K. Pradhan. “Fault injection: A method for validating computer-system dependability”, IEEE Computer, junio 1995, pp. 47-56.

[Cukier 99]: M. Cukier, D. Powell, J. Arlat, “Coverage Estimation Methods for Stratified Fault-Injection”, IEEE Transactions on Computers, no. 7, vol. 48, pp. 707-723, July 1999.

[Damm 88]: A. Damm, “Experimental evaluation of erro detection and self checking coverage of components of a distributed real time system”, Doctoral Thesis, Technical University of Viena, Austria, ocubre 1988.

[Dewey 92]: A. Dewey, A. J. D. Geus. “VHDL: Toward a Unified View of Design”. IEEE Design and Test of Computers, pp. 8-17, 1992.

[Echtle 91]: K. Echtle and Y. Chen, “Evaluation of deterministic fault injection for fault-tolerant protocol testing”, in Proc. 21st. Int. Symp. on Fault Tolerant Computing (FCTS-21), (Montréal, Québec, Canada), pp. 418-425, IEEE, 1991.

[ESPRIT 95]: B. Randell, J.C. Laprie, H. Kopetz, “Predictably Dependable Computing, Basic Research Series”, Springer, 1995

[Folkesson 97]: P. Folkesson, S. Svensson, J. Karlsson, “Evaluation of the Thot microprocessor using scan-chain-based and simulation based fault injection”. (8th. European Workshop of Dependable Computing (EWDC-8): Experimental validation of dependable systems. Chalmers University of Technology, Göteborg, Sweden, April 1997.

[Folkesson 98]: P. Folkesson, S. Svensson, J. Karlsson, “A Comparison of simulation-based and scan chain implemented fault injection”, in Proc. 28nd. Int Symp. on Fault Tolerant Computing (FTCS-28), (Munich, Germany), pp.284-293, IEEE, June 1998

[Gérardin 86]: J.-P. Gérardin, “Aide à la conception dáppareils fiables et sûrs: le DEFI”, Electronique industrielle, no. 116/15-11-1986.

[Gil 92]: P. Gil, “Sistema Tolerante a Fallos con Procesador de Guardia: Validación mediante Inyección Física de Fallos”. Tesis Doctoral. Departamento de Ingeniería de Sistemas, Computadores y Automática. Univ. Politécnica de Valencia, Septiembre de 1992.

[Gil 93]: P. Gil, J. J. Serrano, G. Benet, R. Ors, V. Santonja, “A hardware fault injection tool for dependability validation of fault tolerant systems”. IEEE International Workshop on fault and error injection for dependability validation of computer systems. Chalmers University of Technology, Göteborg, Sweden, June 1993.

[Gil 96]: P. Gil, “Garantía de funcionamiento: Conceptos básicos y terminología” . DISCA. U.P. Valencia. 1996. Informe interno del GSTF (Grupo de Sistemas Tolerantes a Fallos).

[Gil 97]: P. Gil, J. C. Baraza, D. Gil, J. J. Serrano, “High speed fault injector for safety validation of industrial machinery”. (8th. European Workshop of Dependable Computing (EWDC-8): Experimental validation of dependable systems. Chalmers University of Technology, Göteborg, Sweden, April 1997.

[Gil 97b]: D. Gil, J. C. Baraza, J. V. Busquets, P. J. Gil, “Fault injection with simulation in VHDL models and its application to a simple microcomputer system. 6th IEEE International conference on advanced computing (ADCOMP 97), Madras (India), December 1997, pp. 466-474.

[Gil 98]: D. Gil, J. V. Busquets, J. C. Baraza, P. J. Gil, “A fault injection tool for VHDL models”, in Fast Abstracts. 28nd. Int Symp. on Fault Tolerant Computing (FTCS-28), (Munich, Germany), pp.72-73, IEEE, June 1998

[Gunneflo 89]: U. Gunneflo, J. Karlsson, and J. Torin, “evaluation of error detection schemes using fault injection by heavy-ion radiation”, in Proc. 19th. Innt. Symp. on Fault Tolerant Computing (FTCS-19), (Chicago, MI, USA) pp. 340-347, IEEE, June 1989.

[Iyer 95]: R. K. Iyer, “Experimental Evaluation”, 25th International Fault Tolerant Computing Symposium (FTCS 25) Siver Jubilee (special issue), Pasadena, CA, June 1995, pp. 115-132.

[Jenn 94]: E. Jenn, “Sur la validation des sistèmes tolérant les fautes: Injection de fautes dans des modèles de simulation VHDL”. Thèse. Laboratoire d’Alnalyse et d’Architecture des systèmes du CNRS. Doctorat de l’Institut National Politechnique de Toulouse. 1994.

[Kanawati 92]: G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, “FERRARI: a tool for the validation of system dependability properties”, in Proc. 22nd, Int. Symp. on Fault Tolerant Computing (FTCS-22), (Boston, MA, USA), pp. 336-344, IEEE, 1992.

[Karlsson 95]: J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, J. Reisinger, “Application of three phisical fault injection techniques to the experimental asessment of the MARS architecture”, in Proc. 5th Int. Working Conference on Dependable Computing for Critical Applications (DCCA-5) , Urbana, Il, USA, September 1995.

[Laprie 92]: J. C. Laprie. “Dependability: Basic Concepts and Terminology. Dependable Computing and Fault-Tolerant Systems”. 1992. Traducción al castellano por P. Gil ver referencia [Gil 96].

[Madeira 94]: H. Madeira, M. Rela, F. Moreira, J. G. Silva, “RIFLE: A general purpose pin-level fault injector”. First European Dependable Computing Conference (EDCC-1), Berlin, Germany, October 1994.

[Maxion 93]: R. A. Maxion and R. T. Olszewski, “Detection and discrimination of injected network faults”, 1993.

[Merenda 92]: A. C. Merenda and E. Merenda, “Recovery/serviceability system test improvement for the IBM ES/9000 520 based models”, 1992.

[Miczo 90]: Miczo A. “VHDL as a Modeling-for-Testability Tool”. En Proceedings COMPCON’90, pp. 403-409, 1990.

[Myers 79]: G. J. Myers, “The Art of Software Testing”, John Wiley and Sons, 1979.

[Powell 93]: D. Powell, E. Martins, J. Arlat, Y. Crouzet, “Estimators for Fault Tolerance Coverage Evaluation”, 23rd. International Symposium on Fault Tolerance Computing (FTCS-23), June 1993.

[Pradhan 96]: D. K. Pradhan: “Fault-tolerant computer system design”, capítulo 5: “ Experimental analysis of computer system dependability”. Ed. Prentice Hall, 1996, pp. 282-393.

[Pressman 97]: R. S. Pressman, “Software Enginering. A practitioner’s Approach”, McGraw Hill, 1997.

[Rosemberg 93]: H. A. Rosenberg and K. G. Shin, “Software fault injection and its application in distributed systems”, in Proc. 3rd. Int. Symp. on Fault Tolerant Computing (FTCS-23), (Toulouse, France), pp. 208-217, IEEE, 1993.

[Sampson 98]: J. R. Sampson, W. Moreno, F. Falquez, “A technique for automatic validation of fault tolerant designs using laser fault injection”, in Proc. 28nd. Int Symp. on Fault Tolerant Computing (FTCS-28), (Munich, Germany), pp.162-167, IEEE, June 1998

[Segall 88] : Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin, “FIAT - fault injection based automated testing environment”, in Proc.

18th. Int. Symp. on Fault Tolerant Computing (FTCS-18), (Tokyo, Japan), pp. 102-107, IEEE, 1988.

[Vigneron 97]: C. Vigneron, “Línjection de fautes. Méthodes et outils existants pour la validation des dispositifs électroniques à vocation sécuritaire”, Informe interno INRS ND 2067-169 -97, Nancy, France, pp. 609-619, 1997.