Paper presented at DASIA'2015, 20.05.2015, Barcelona, Spain

Characterizing Verification Tools through Coding Error Candidates Reported in Space Flight Software

Christian R. Prause
Deutsches Zentrum für Luft- und Raumfahrt e.V. (DLR)
Bonn, Germany
e-mail: Christian.Prause@dlr.de

Ralf Gerlich, Rainer Gerlich
Dr. Rainer Gerlich BSSE System and Software Engineering
Immenstaad, Germany
e-mail: Ralf.Gerlich@bsse.biz, Rainer.Gerlich@bsse.biz

Anton Fischer
etamax space GmbH
Braunschweig, Germany
e-mail: A.Fischer@etamax.de

***Abstract*—Mastering the continuously increasing amount of software requires identification of more efficient strategies for software verification. Currently, fault coverage is only indirectly addressed, e.g. by code coverage. The idea as presented in this paper is to get a better understanding of fault coverage by a systematic classification of software fault types, derivation of footprints of verification tools regarding coverage of such fault types, and recording of required effort. A number of issues regarding fault identification and classification are discussed in this context.**

Keywords: software faults, fault identification, fault coverage, software verification, verification tools, verification efficiency, ECSS, DO178, EN 50128

I. INTRODUCTION

The objective of software verification is driven by two goals

- to demonstrate that a software product performs as intended, and
- to identify sources which could prevent the product from performing as intended.

Such sources are called defects, bugs, faults, errors or failures. Definitions of such terms are given below in Section A.

If both goals could be fully covered, only one of both would be sufficient. However, in practice it is impossible – especially for complex systems – to achieve full coverage for both. Therefore both goals must be tackled in parallel during the verification process.

For software products of high complexity or large size, tools need to be used in the verification process. Such tools are also software products and their output might be wrong or incomplete.

In this paper, we present a planned activity addressing the capabilities of verification tools regarding identification of faults – the *Fault Coverage* – and related reporting, targeting an evaluation of the efficiency of fault identification, also considering the required effort.

A. Definition of Terms

In the context of this paper the following definitions apply to the terms fault, error, failure and defect (as used in ECSS-Q-HB-80-03A[1]:

- An *error* is a bad or undesired state in a software system.
- A *fault* is the cause of an *error* having its origin in the code which may be called a *mistake*.
- A *failure* is a non-compliance regarding external behavior being recognized between expected and observed properties of the software product as a consequence of an error.
- A *defect* commonly refers to troubles with a software product, with its external behavior or its internal features (e.g., its maintainability). This includes consideration of the risk of faults by potential changes of the context.

In this terminology, faults are considered as a subset of defects. A fault may cause an undesired, observable behavior of a system. A defect, which is not a fault, will not, but it still addresses issues to be considered[1].

Note that an error can be encountered either while abstractly reasoning about the software, e.g. in the context of the virtual semantics of a programming language, or during actual execution.

From these definitions the following chain of causality results as shown in Fig.I-1.

Term	Scope
Fault	Mistake in code
⇓	⇓
Error	Bad state of a system
⇓	⇓
Failure	Unexpected observed behaviour

Fig.I-1: Causality Chain Fault, Error, Failure

The term *fault coverage* describes the degree to which defects present in the software are or were detected, or are detectable in the course of the defined verification process. It is usually represented by the ratio of the number of defects recognised and the number of defects present.

Be aware that the number of defects present is an unknown figure. Therefore, a percentage cannot be derived.

[1] As the term "fault" is widely used, e.g. in context of "fault coverage", this term is kept, although "defect coverage" would be the proper term following the terminology of this paper. Similarly, „criteria for fault identification" need to be re-considered as "criteria for defect identification", and "fault types" as "defect types".

Hence, the absolute number of identified defects is of relevance.

The term *code coverage* describes the degree to which the code of the software under test is or was executed during test. In the simple case, it is represented by the ratio of the number of code instructions executed and the total number of instructions. There are different kinds of code coverage such as statement coverage, block coverage, decision or branch coverage, or Modified Conditional/Decision Coverage (MC/DC).

B. *Verification Issues*

Due to the increasing amount of software, its increasing complexity and its criticality, verification tools shall help to decrease the effort and to increase the fault coverage.

Standards such as ECSS (space domain), DO178 (aviation domain) or EN 50128(railway domain) require the definition of a verification process, by which a software supplier shall demonstrate that the software has the required properties and – in addition – does not have undesired properties.

The desired properties are usually expressed by explicit requirements, while the undesired ones are a matter of generic requirements, valid for every line of code, in principle. This makes it difficult to address all verification issues on undesired properties.

Examples of requirements which mainly focus on the undesired properties are the so-called RAMS requirements on reliability, availability, maintainability and safety.

The current approach to demonstrate compliance between properties of a software product and requirements is to analyze the source code for non-compliances – e.g. violation of given rules – and to run tests, aiming to cover both, compliance and non-compliances. Such activities are supported by verification tools.

Bearing in mind that also verification tools can have faults and may not accurately report defects, a verification process based on just applying a tool and believing in its reports is not sufficient. Without further quantifying or qualifying the expected reduction in risk, the result of this application is nothing more than a good feeling.

E.g. a tool may fail to report a defect or may report a defect where no defect is present. There are 4 distinct cases depending on whether a defect exists or not and whether a tool reports a defect or not (Fig.I-2).

		Code	
		Defect present	*Defect NOT present*
Result	*Defect Reported*	true positive	false positive
	Defect NOT reported	false negative	true negative

Fig.I-2: Classification of Tool Reports on Defects

These terms are discussed in detail in Sect. III.A.

Fault coverage can now be expressed as the ratio of *true positives* and the sum of *true positives* and *false negatives*. In information theory synonyms for *fault coverage* are *sensitivity* or *recall*. Similarly, the *precision* by which a tool does detect *true positives* in presence of *false positives* can be expressed as the ratio of *true positives* and the sum of *true positives* and *false positives (*Fig. I-3*)*.

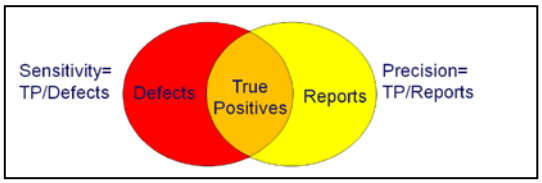

Fig. I-3: Sensitivity and Precision

Even if a certain defect was already successfully identified by a tool, this does not imply that all similar defects have been reported, apart from the fact that other defects might not be addressed at all. Consequently, an unknown, but possibly non-negligible risk still remains.

This suggests that more knowledge about identification capabilities of verification tools is required.

Therefore the Space Administration of the German Aerospace Center (DLR) has initiated an activity for a systematic investigation of fault/defect identification capabilities of verification tools, aiming to derive footprints of tools.

The focus is put on tools which do not require manual intervention for error identification, apart from configuration of the tool.

A number of criteria for fault/defect identification are known, e.g., violation of layout rules, aiming to obtain good readability in context of reviews and code inspection, or out-of-range conditions, addressing actual run-time errors.

However, when selecting criteria and tools, it is not exactly known today which fault/defect coverage can really be achieved at the end for the following reasons:

- the defects in the software are not known at all,
- even if a tool claims to support defect identification for certain defects, it is left open whether all such defects will be reported and if a user will recognize and be able to handle all such reports at the end,
- even if reports are provided and recognized it is left open whether a user can remove all real defects due to limited effort and time and possibly at presence of irrelevant reports(false-positives, for details see Sect. III.A)

In the past, a number of tool evaluations [1][10][11][12]indicate discrepancies between expectations and achieved results: identification support might not be as advertised, or practical difficulties may keep reported defects from being recognised by a user. These evaluations can be considered as a first step towards an analysis of tool characteristics.

However, they neither do cover tools as used in space and/or safety critical domains nor do they fully and/or systematically characterise a tool. This situation should be improved by the following measures:

Firstly, defects need to be characterised and classified into defect types.

Secondly, it should be known which defect types a tool does reliably cover and in which context a defect is reported and can well be recognised by a user.

Thirdly, an effort figure should be derived, allowing to optimise the use of tools and to define a sequence on the order they shall be applied, aiming to reduce the overall effort while increasing the risk arising from defects not being addressed.

In consequence, a footprint of a verification tool should be available if it is considered as a candidate for use in a project. Footprints shall enable planning which defects be tackled regarding a project's criticality issues and to ensure or at least improve the chance that they will be found.

In Ch. II we discuss verification issues in context of standards from space, aviation and railway domains and identify drivers for fault/defect identification.

In Ch. III we consider classification of defect types, criticality issues and their relevance to tools. A major issue regarding assessment of fault identification capabilities is the classification of defects according to Fig.I-2. As we will see, the classification for true and false positives depends on the verification issues. Therefore we discuss such issues in detail to get a solid base for tool assessment.

In Ch. IV the contents of the planned activity is described aiming to derive tool footprints.

Finally, conclusions are drawn in Ch. V together with an outlook on the next steps planned.

II. CURRENT PRACTICES OF SOFTWARE VERIFICATION

Common to standards in several domains is the definition of a software verification plan. The verification plan shall describe how compliance can be demonstrated and how defects will be identified. However, there is no requirement to state which defects will be covered and to which degree. It is sufficient to define a process as such. This is unsatisfying from a principal point of view.

In this chapter we will discuss a number of issues related to improvement of the current verification process and consider the position expressed in the standards regarding such issues.

Issues arising from insufficient knowledge on tool footprints are addressed in Section A. In Section B issues on verification in context of current practices are discussed.

In SectionC.1)the verification approaches of the standards ECSS, DO178 and EN 50128 are considered regarding issues on capabilities of fault identification of tools.

A. Issues on the Software Verification Plan

The Software Verification Plan (SVP) defines the verification procedures to satisfy the software verification process objectives. Such objectives depend on the application domain and the standards applicable for a certain domain. Therefore we consider standards from three domains.

In general, the standards demand that a contractor describes in the SVP how the objectives of the project can be reached and the risks can be minimised. The content of a SVP may be subject to negotiation between customer and contractor like it is for ECSS or not as in case of EN 50128.

In an SVP, procedures are defined and tools are selected which shall allow to achieve a sufficient level of confidence in the software product. The selection of tools is driven by what is available on the market, in use in projects and what is considered as adequate to manage the expected risks.

Instead of fault coverage, code coverage is put in the focus, hoping that sufficient code coverage will result in sufficient fault coverage.

But still a number of questions are left open. The use of a static analyser and execution of tests – with appropriate code coverage – are considered as sufficient, while it remains unclear whether they actually are sufficient due to insufficient knowledge

1. regarding defect identification
   a. To which degree is identification of the defects, for which identification is expected, supported, and can such defects really be found?
   b. For which defect types are tools complementary?
      In this case the fault coverage could be improved by benefitting from different tool capabilities resulting in better fault coverage.
   c. For which defect types are tools equivalent?
      If equivalent, false negatives related to a certain tool in the chosen set of verification tools, could be compensated by true positives by another, equivalent tool in the set.
   d. Do the remaining risks, which result from incomplete fault coverage, comply with the expectations?
2. regarding effort
   a. Which tool is the most efficient for identification of certain defect types?
   b. In case of several complementary or overlapping tools, which is the best order of sequence to achieve minimum overall effort?

While the verification process is clearly defined, the resulting quality of the software product itself in terms of still hidden defects remains unclear, because it is left open which of the – known – defect types are actually covered.

More detailed information on the fault identification capabilities of tools shall allow closing this gap and tailoring the verification process regarding minimization of risks and effort.

B. Verification and Risk Issues

Manifestation of a fault as an error is subject to fault activation conditions (see Section III.A).In the course of a development and verification process, such activation conditions may differ at different stages of integration.

Subject to discussion is, for example, whether a is of relevance when its activation condition is raised during module/unit testing, but cannot be raised during integration or system-level testing.

Then we may ask: Is such an identified fault a *true positive* or a *false positive*? And the project manager may ask: Does module testing impose an overhead on the project?

We are addressing this issue because it is of relevance for tool evaluation regarding fault identification capabilities.

A simple, abstract example for such a case is given in Sect. III.C. In general, what is addressed here is:

- a function receives a valid input vector during module testing,
- in turn it provides a wrong output vector or shows other unintended behavior like an abort,
- it is confirmed that this input vector cannot occur in the system context.

In this case the activation condition for the fault does never occur in the system context.

It may even happen that a function provides the expected output vector and no unintended behaviour for all (valid) input vectors occurring in system context. However, there may be further input vectors in the context of module testing, for which it provides a wrong output vector or shows unintended behaviour, so that an error occurs.

In both of the above cases, the system is not affected at all, although in the context of module testing, a defect is identified. Shall such defects be fixed? Or should they even not be reported?

It is important to be aware of that the missing activation of a fault in system context just represents a snapshot. In case of reuse or maintenance the conditions may change – possibly silently – and the fault may be activated.

In case of such a latent fault, verification of the system must be repeated after every change, which may be costly. The safest and probably cheapest approach is to identify such latent faults and to fix them.

Whether a fault should be removed or not is a matter of risk assessment.

Risk assessment has to estimate the probability that a software product may not perform as intended when being operated under conditions defined as valid, and the costs of such an occurrence. Classification of such a risk regarding its criticality requires knowledge about the conditions under which the risk may be realized. Consequently, a hidden fault implies an unknown risk.

It may happen that the risk is considered as low or negligible once the fault has been identified. Therefore, from the view of risk reduction, it is highly desirable to identify as many faults as possible to allow for their assessment.

The lack of information about the actual valid input domain may lead to tools flagging possible faults in the code at unit level, although in an integrated setting, the users of the unit properly take care that no invalid values are passed.

Unfortunately, this scenario is often unavoidable, as most relevant programming languages allow the definition of the input domain only as an orthogonal set, i.e. as a set of tuples where each parameter may take any value from its type independently of the other parameters. However, the set describing the valid input domain often is non-orthogonal, imposing interdependencies on the parameters and thereby limiting the combinations.

Given enough information, a tool may theoretically avoid flagging such possible faults and instead focus on the faults affecting the handling of valid inputs. In practice, consideration of extensive context information may be beyond the capacities of the computing hardware of the verification tool, therefore making verification at integration or system level impractical.

However, assessment of robustness is also a valid goal. If only valid inputs are provided, aspects of robustness are not addressed. Although invalid values might not be expected, they may occur due to fault propagation, also in the system under operation. From this point of robustness, reports on faults which are expected to be dormant should be of relevance, too.

C. Position of Standards

1) ECSS

Software verification is intended to confirm that the product is built right. The verification process therefore aims at the correctness and consistency of outputs of software-related activities with respect to their inputs. It occurs in parallel to the actual development, and consists of (i) a process implementation that creates and deploys a verification plan, and (ii) the actual verification activities according to this plan (ECSS-E-ST-40C[1], ECSS-E-HB-40A[3], ECSS-Q-ST-80C[4], ESA ISVV Guide[5]).

ECSS-E-ST-40C does not make universally valid prescriptions about software verification efforts, the identification of risks and the degree of independence of coders and testers. It recognizes, however, that efforts can differ and that there are varying degrees of separation between developing and verifying organizations, ranging from no separation (same person) to Independent Software Verification and Validation (a person in a different organization, ISVV)[5]. The standard therefore demands that a determination shall be made regarding verification effort, the identification of risks and the degree of independence to be applied in different verification activities (ECSS-E-ST-40C). Another example is test coverage: While 100% code coverage is required for higher criticality levels, lower coverage values can be negotiated for less critical software.

The verification activities are therefore a point of negotiations between software suppliers and customer. The verification approach must be defined in the Software Verification Plan.

For example, independent verification has considerably higher costs but increases confidence in development results. A better understanding of the characteristics of different software verification tools – regarding overlapping/ equivalent and complementary features – with respect to flight software is therefore important for customers and suppliers alike: the more is known about fault identification capabilities of tools, the more efficient verification is, and the higher is the confidence.

Verification activities typically address the areas of the requirements baseline, technical specification, architectural design, detailed design, unit testing plan and test results, coding and the verification of the software validation, and related documentation

In particular, the verification of code may cover, for example, correctness of data and control flow, error handling, controlling of the effects of run-time errors, memory leaks, numerical protection mechanisms, code coverage of tests, and the verification of source code robustness (e.g. resource sharing, division by zero, pointers).

The use of static analysis tools is explicitly recommended by the standard ECSS-E-ST-40C.

The objectives of software verification depend on criticality levels (A-D, with A highest) based on the severity of the consequences of system failures as defined in ECSS-Q-ST-80. Knowing tool characteristics can help to choose the right tools or combinations of tools for the respective criticality levels.

Tailoring of the standard is possible. As with all ECSS standards, tailoring is explicitly recommended on a case-to-case base or as a matter of negotiations. The standard even comprises an appendix with a table that contains a proposed tailoring according to software criticality.

2) DO178C

DO-178C (and its nearly identical European counterpart ED-12C) distinguishes between validation and verification in much the same way as the ECSS does.

The purpose of the verification process is to detect and report faults (called "unintended functionality" there, which indicates that faults and not defects are in the focus) that might have slipped into the software during development. While testing takes a major role in the verification chapter, verification is seen as more than simply testing. The reason is that testing, in general, cannot show the absence of errors. Consequently, combinations of reviews, analyses and tests are applied to verify the software. The standards note that removal of errors is part of the activities in the software development processes.

Interesting about the verification approach in DO-178C is its discrimination between high- and low-level requirements, and furthermore that diverse kinds of coverage values are the drivers of the process: Test Coverage Analysis, Requirements-Based Test Coverage Analysis, Structural Coverage Analysis, and Structural Coverage Analysis Resolution.

DO178 recommends addressing certain fault types, of which examples like invalid data are provided, in a non-exhaustive list.

A verification tool is subject to qualification by which correct behavior must be demonstrated. The procedures of qualification are defined in a separate supplement (DO330)[7].

The verification approach shall be document in the *Plan for Software Aspects of Certification* (PSAC).

The objectives of software verification depend on criticality levels (A-E, with A highest) based on potential failure conditions.

3) EN 50128

To get a broader view on the position of standards we also consider a standard outside the aerospace domain: EN 50128 which is applicable in the railway domain.

There are clear rules regarding the verification activities. Tailoring common objectives of this standard is not possible while it is for ECSS. Verification characteristics and the respective objectives of processes and documentation depend on the Safety Integrity Levels (SIL, 0-4 with 4 as highest), based on (numerical) probability-to-failure figures) Similarly, to ECSS and DO178 the verification approach must be documented in a plan (Software Verification Plan, SVP) early in the requirements engineering process. The SVP needs to provide all information regarding criteria, methods, techniques and also tools planned to be used within the different phases of safety critical railway software verification.

EN50128 is applicable for software engineering tools (integrated development tools, verification and validation tools, simulation tools, etc.). An "adequate" set of tools shall be used for software engineering and a high degree of automated testing and verification shall be reached. Compared to other standards, EN 50128 states in a relatively soft manner that software tools need to be available and known "as soon as possible" within the project and that these tools shall be suitable for the specific usage within the software engineering project. Their suitability could be proven by independent tool validation, appraisal and permission for usage (comparable to DO-330). Currently, tools with qualification and tools without qualification are often used in parallel. EN50128 does not explicitly focus on failure identification methods and techniques of verification tools during tool qualification processes.

EN50128 focuses explicitly on the important step of software-software integration test activities. Here special emphasis shall be put on software module integration having used different implementation and verification tools during modules implementation, e.g. at different supplier premises (refer to 11.4.5 in EN50128).

Taking all this information into account, verification tool diversification as proposed within this paper could therefore be an objective of the EN 50128 based software qualification plan (see intro to appendix A of En 50128, in combination with EN 50126). According to EN 50128 a third party certification authority is able to choose additional approaches for adequate verification. Especially during system integration with multiple customer-supplier-relations in the value chain this shall be mandatory to cover different failure detection approaches of verification tools in order to detect most of the common failure types and potential failure conditions.

EN 50128 also focuses on the phase of software maintenance (chapter 16). Tools used in maintenance and evolution phases shall fit to the ones used for implementation. In practice, special attention shall be given to substitution actions (changes in tool chains) during operational software usage.

D. Summary on Standards

All standards require definition of a verification process which shall be proposed by the software supplier and shall be agreed by the customer. This process shall sufficiently identify the measures to demonstrate that the software will behave as expected, including measures for fault identification.

A major driver for detailed testing on module level is the so-called RAMS requirements (Reliability, Availability, Maintainability, Safety). The knowledge on imperfectness of

verification, especially in a complex system context, suggests a bottom-up approach applying the outmost degree of verification on every level, starting with units or modules – the functions.

If tools for fault identification are applied in this context, qualification of such a tool is required demonstrating that expected properties, e.g. fault or code coverage, will be achieved based on a representative scenario. So far, every project performs tool qualification on its own, tailored to the project's specific conditions.

Use of tools with overlapping and complementary features to increase confidence in the verification process is not addressed, probably because there is lack of information on tool characteristics, so far.

This situation should improve when footprints are available.

III. Issues on Fault/Defect Identification

Recently, a paper on the software verification approach of the Mars rover Curiositystated: "Before we can do so [to reduce the likelihood of faults], though, we have to know what types of mistakes occur"[13].

A first step towards such a strategy is to identify sets of similar defects, called defect types.

In this sense, defects are instances of defect types. An out-of-range-condition may occur at a number of locations in a software system, but typically all of these can be found by a small set of common detection mechanisms. For example, out-of-range-conditions can be found by index checking or detection of corrupted memory. While index checking targets the source of an error, memory corruption targets its further consequences.

In general, it is possible to detect faults by generic and non-specific detection mechanisms, such as watching for run-time-exceptions or memory corruption.

Sometimes finding such defects may depend on chance. For example, using an invalid index to access an array may lead to memory corruption by accessing a memory area that is otherwise unused and thus trigger an exception or memory corruption detection mechanisms.

A wrong index, however, may also lead to an access to an unintended, although valid address, e.g. if the accessed address is part of another object in memory. In these cases neither an exception nor the memory-corruption detection mechanism would be triggered directly. The corruption of the other object may lead to another error elsewhere in the software, however, neither is occurrence nor visibility of this follow-on error guaranteed.

These generic detection mechanisms may sometimes miss specific errors. Once such an error has been found, it may be possible to find a more specific mechanism that may have a higher chance of detecting these kinds of errors. In the example, the introduction of explicit index checking would be a more specific mechanism.

In this way, the capabilities of a verification tool can be improved iteratively.

Having defined defect types, the verification tools can be characterized accordingly.

A further consequence of such a classification is that it shall be possible to identify for which defect types tools are complementary and/or equivalent. This will allow to define a set of tools covering a well-defined / well-known and possibly broader set of defect types.

A. *Considerations on Fault/Defect Identification*

An approach for fault identification needs to consider the imperfection of verification. This imperfection increases the risks remaining after the verification process. The question is: Can such risks be reduced at affordable effort and costs?

Improving the knowledge about such imperfection is a necessary pre-condition to achieve this goal.

Therefore in this section we further consider the mechanisms of fault activation and fault identification by tools.

Regarding the software or the source code respectively, identification of faults may happen in two different ways by

1. analysis of the source code
   called "Static Analysis", and/or
2. execution of the source code,
   called "Dynamic Analysis" or testing.

It is well-known that testing cannot prove absence of defects, only the presence of defects. Regarding static analysis it is broadly believed that tools would identify defects which they ought to detect. However, this is not true in general. Static analysers may also miss violation of rules.

In general, to detect a defect, certain conditions must be fulfilled.

In case of static analysis the principal conditions are:

A1. The location of the defect in the source code must be subject of analysis by the tool.

A2. Identification of the defect type must be supported by the tool.

A3. The defect must be identified in its actual context and reported by the tool.

A4. The defect report must be recognized by the user.

A1considers that the faulty statement may not be analysed at all due to limited resources like time or memory. Regarding A1 and A2 a validated footprint of the chosen tool is a pre-condition to success. If unknown, one should not expect to get a message on such a defect. A3 states that identification may depend on the context in the source code. E.g. a fault in the tool itself or a missing consideration of that context may prevent identification and reporting. A4 addresses visibility / usability issues: a relevant message may not be recognised amongst thousands of messages.

In case of dynamic analysis / testing the following conditions must be fulfilled:

T1. The statement must be executed.

T2. The fault's activation condition must occur.

T3. The fault must manifest as an (observable) error
e.g., by observation of an exception or an out-of-range condition, or comparing expected with observed output.

T4. The defect report must be recognized by the user.

T1addresses coverage issues and is a trivial, but necessary condition. T2 reminds us that covering a statement

once only may not be sufficient to detect a defect. This explains why code and fault coverage are not identical. T3 requires that a tool must be sensitive for that defect type (must communicate an observed defect). Finally, T4 is equivalent toA4.

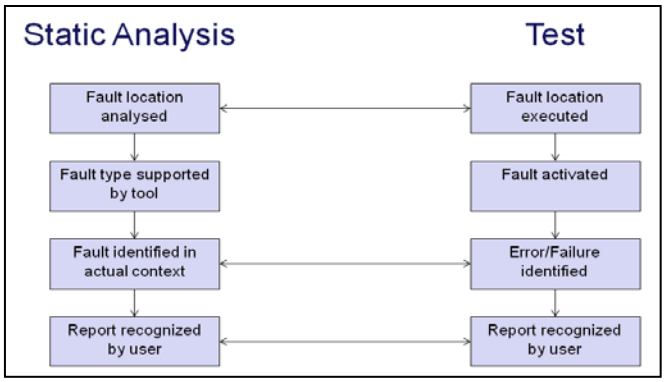

Fig. III-1: Conditions for Fault Identification

These considerations demonstrate that fault identification depends on 4 principal conditions (

Fig. III-1). In consequence, just to know that a tool aims to support identification of a certain defect type is not sufficient.

Moreover, we have to face the possibility that a tool wrongly reports a defect. The following two cases may occur:

- a report is provided,
  called the *positive* case because something is provided, or
- no report is provided,
  called the *negative* case, because nothing is provided.

Each such case may be correct or wrong, i.e.,*true* or *false*.

According to Fig.I-2 there are 4 combinations to be considered:

- *true positive*
  a tool flags a defect and it is an actual defect
- *true negative*
  a tool flags no defect, and there is no defect
- *false positive*
  a tool flags a defect, but it is not a defect
- *false negative*
  a tool misses an actual defect

The cases with *true* are desired, the others are undesired. *False positives* increase the effort required for analysis of the reports without providing payback in the form of detection and possibly eradication of a defect. *False negatives* mean that a possibly critical fault remains undetected.

A final decision on *true positive* or *false positive e*is usually only possible by manual inspection of the source code (which is also required to fix the defect). To decide whether *true negative* and *false negative* did occur, reports from at least two tools are required (or another additional information sources).

True negatives are trivial in principle, but become a positive property of the tool when another tool reports a defect. Conclusions on *false negatives* can only be drawn when information on a confirmed defect is available.

B. Relevance to Tool Assessment and Selection

Classification of defect types, understanding of fault identification conditions (A1-A4 and T1-T4) and classification of defect reports as shown above and in Fig.I-2 are pre-conditions to deriving footprints on verification tools, i.e., to characterize a tool regarding its fault identification properties.

Knowing a tool's footprint, including the effort required to identify the source of a defect, will help to improve the verification strategy: weaknesses of one tool may be compensated by another one, and overall effort can be minimized by choosing the most efficient tool or set of tools being applied in an optimized sequence.

Such an optimization strategy shall consider:

1. Start with tools which allow efficient identification of defects for the/a major part of defect types, though they may not cover all defect types.
2. Then continue with tools covering the remaining defect type. Theymay require a higher (manual) evaluation effort per defect, but it is likely that only a few defects occur for such defect types.

C. Conflicting Verification Issues

A major issue of verification arises regarding the benefits of identifying a defect and the related effort and costs. Different steps of verification – as driven by the standards – address different verification issues. This may imply that verification objectives e.g., on the level of module testing must be fulfilled, which do not contribute to the reliability of the system at all.

From the system's point of view – and from the project's costs – full tests seem to imply an overhead. However, as far as the environment does not accurately specify what is required, the goals of module testing – as understood today – must be kept.

The following exaggerated example shall illustrate this conflict:

```
// Module Level:
float sin(float x){ return 0.0; }
```

```
// System Level:
y=sin(float(n)*π); // n an integer value
```

Fig.III-2: Wrong Implementation, No System Failure

As the module *sin* is called for multiples of π in system context, only, it returns the expected values: although the code is faulty, no error manifests and no failure will be observed on higher levels.

This example is more relevant than it may seem at first look: in practice (see Sect. III.F), we have observed such cases in all analysed software packages, at higher complexity, of course. The implemented algorithm was

wrong, but for the parameters passed during operation, the output was correct.

Do such defects matter at all, when they do not cause any failure? Does the implementation violate the module's specification? If yes, does it matter? Does the (successful) identification of a defect imply a cost overhead, which should be avoided from the perspective of a project manager? Is such a defect a *false positive*?

The essential point is that the conditions (usually) are not known for which a module is used in context of the system. Consequently, this requires the test of the full functionality in context of module testing. If the conditions would be known, the module tests could be limited, even the implementation could be simpler.

In practice, such information is not available, today. But is it desirable at all to provide it?

Firstly, the effort and costs would increase due to the required documentation. Secondly, risks would increase regarding reuse and maintenance because a module with limited functionality – less than expected – may wrongly be used by ignoring the limitations and the documentation and passing the full spectrum of inputs.

D. Fault Identification vs. Fault Activation

Above discussion leads to the following principal classification on fault activation conditions:

F1. inherently risky
F2. temporarily disabled
F3. permanently disabled.

Inherently risky means that an activation condition may occur any time when the system is being operated, even if the probability may be low.

Temporarily disabled means that an activation condition may not occur currently when the system is being operated, but it may occur when the operational context is changed.

Permanently disabled means that actually no fault is present and even a change of context will not introduce one.

When the context of fault activation changes, e.g. due to maintenance, the type of activation may change from *inherently risky* to *temporarily disabled* and vice versa. In the latter case, knowledge about such a (potential) fault is important. The only way to avoid a silent conversion from *disabled* to *risky* is to fix the fault early enough.

The critical point is that the risk is not known as long as the fault is not identified. In consequence, for matters of risk reduction, as many faults as possible should be identified, even when after analysis they turn out to be *temporarily disabled*.

From this point of view, it is desirable to identify *temporarily disabled* faults, because these are candidates for silent activation.

A project may need to make a trade-off between risk reduction and effort imposed by evaluation of such faults. But as long as the fault report is not evaluated or is not available at all, no conclusion about the relevance of such a fault is possible.

In consequence, two types of *false positives* exist:

- those which may become *true positives*, and
- those which cannot become *true positives*, at all.

Only latter ones shall be considered as a fault of a tool.

E. Issues on Defect Reporting and Report Evaluation

Efficiency of fault identification depends to a major part on the correct tool reports as discussed above and shown in Fig.I-2. In previous projects major discussions were raised on *false positives*.

The number of *false positives* significantly impacts the efficiency of a tool and its application. Therefore, a detailed risk assessment is required when deciding whether a defect is considered a *false positive* or not.

Minimization of risks requires minimization of *false negatives.* While (real) *false positives* cause an overhead, *false negatives* increase the risks. Mastering of risks requires good knowledge on *false negatives*.

1) Fault Potential of False Positives

A defect report may be considered as *false positive* either because

- the tool reports a defect and cannot preclude that the report is inherently risky (*true positive*) or temporarily disabled due to missing information, or
- a tool erroneously reports a defect.

Temporarily disabled fault activation is mostly a matter of consistency. Experience shows that there may be hidden dependencies in the software, which may cause an inconsistent context, e.g., due to incomplete maintenance as a matter of unknown dependencies.

Such an inconsistency can already occur by use of a constant number at different locations in the source code: being used in the declaration of an array's size and in a corresponding for-loop for initialisation of array elements, an inconsistency occurs when the number is changed only at one location, either the upper limit of the loop or the size of the array.

This example may seem manageable, but in previous activities we found many cases where the dependency was more complex and could have caused major impacts on system operations after incomplete maintenance. An example is provided in Sect. III.F.

In case the for-loop causes an out-of-range condition (upper limit of the loop exceeds the number of elements), a tool may detect this defect. In case the for-loop does not cover initialisation of all array elements (upper limit of the loop is less than the number of elements), it is more complex. Detection of partial initialisation is currently not sufficiently covered by available tools.

The challenging question is whether detection of such fault potential is considered as a *true positive* or as a *false positive* in case of a *temporarily disabled* activation condition. Clearly, in a consistent context, the fault cannot be activated at all and it may be argued that it was useless to spend effort on the analysis of the defect report indicating an inconsistency.

Indeed, it is wasted effort in the consistent context, but it is not, if the context cannot be controlled such that change into a critical state cannot be prevented.

Another aspect is that too many of such *false positives* decrease the visibility of more critical faults, e.g., when it is not possible to separate *inherently risky* faults from *temporarily disabled* ones – automatically

However, in case of *false positives* of type *temporarily disabled* a software supplier can reduce the number of such reports by inherently preventing fault activation when creating the source code. In consequence, the supplier and probably the project management, too, must decide at stage of development: more checks in the code and less risks, or less checks and more risks and more evaluation effort.

Real false positives are clearly an issue of the tool or method.

2) Mastering Risks of False Negatives

A risk is to believe faults are covered while they are not. This risk results from misleading or misunderstood descriptions of tool support. This is different from the case when it is known in advance that certain defect types are not supported by a tool.

If known in advance, another tool may be used instead or in addition to achieve the required coverage of defect types (Fig. III-3). Then risks can be mastered sufficiently.

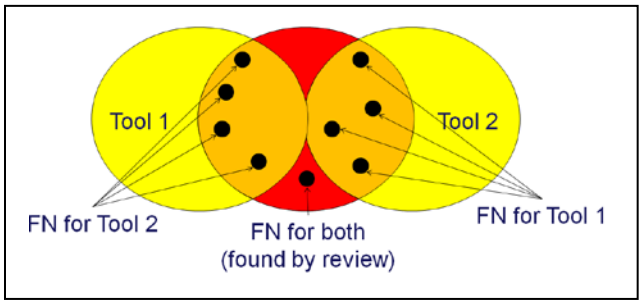

Fig. III-3: Identifying False Negatives by A Set of Tools

Hidden risks remain in case a defect report is expected, but when the tool does not issue a report. The detection of this case is only possible by comparing the results of two or more tools.

In consequence, availability of tool footprints implying the possibility to compare tool characteristics supports reduction of false negatives and reduces remaining risks.

3) The Risk of too many False Positives

The incapability of a tool to detect specific faults is one reason for the occurrence of false negatives. Another may be found in too many false positives.

As already discussed, false positives imply analysis overhead without providing value. Of course, this only applies to actual false positives, i.e. such positives that do not merely refer to temporarily disabled faults.

If the number of reports gained from the set of tools applied is larger than what can be handled within the planned time frame of a project, there are two options available.

The first option is to extend the time frame of the project, taking into account additional cost and time, thereby possibly missing deadlines and postponing finalisation. This may be necessary in some cases, most specifically in case of highly safety critical software.

The second option is to prioritize the reports for analysis, effectively analysing only a subset of reports. It is not known in advance which of the reports to be analysed are true and which of them are false positives. Therefore, there is a non-negligible chance that some of the reports not analysed are actually true positives. These effectively become false negatives, because, although they are reported, they are not considered.

In both cases, false positives are more than a mere nuisance to the user, and in both cases, they have considerable impact on the project. In the second case they may even have critical impact on the software quality.

4) Fault Coverage vs. Code Coverage

All standards refer to code coverage as driver for fault identification, but not to fault coverage. It is believed that sufficient confidence can be achieved when sufficient code coverage is reached, possibly complemented by additional robustness tests.

ECSS-E-ST-40C [2] defines concrete figures for code coverage only for the highest safety categories while in the other cases the figures are subject to negotiation between supplier and customer.

DO178 defines figures on code coverage depending on safety categories. 100% coverage is required for requirements. It suggests to add requirements when the demanded code coverage cannot be achieved with the given set of requirements (the specification). Robustness testing is demanded. Similarly, EN 50128 also requires achievement of given code coverage figures.

A percentage on fault coverage cannot be derived at all, because the number of defects in a software package is unknown. However, it is possible to measure fault coverage in terms of defect types. This is an issue of the planned activities.

5) Standards vs. Tools

DO178 and EN 50128 require tool qualification (e.g., following DO330), i.e., the proof that the tool does provide the expected output regarding a certain issue for a representative scenario agreed on by all parties. Identification of *false negatives* is not a direct issue. Proving of the output correctness for the given scenario should imply that *false negatives* should not occur.

ECSS-Q-ST-80C [4] requires that the customer must agree in the use of a tool selected by the supplier. Tool qualification is not an explicit issue of the standard.

Nothing can be found in these standards on increasing hit rate of defects and reducing risks by tool diversification. Most probably, this is a matter that today characterization of tools is not available, which is a pre-condition for tool diversification.

However, tool diversity also is not disallowed because the verification strategy can be defined within a certain scope and is agreed between supplier and customer at the end.

So the work being done within the study shall also give additional information to tool qualification approaches in safety critical domains. It is useful knowing that a specific tool works properly in the planned project scenario, but it is

also worth knowing which kind of defects can, will or may not be found.

Knowing the characteristics and footprints of certain tools means a *concerted* planning of different tools' usage within the verification processes in close harmonization with certification authorities.

F. Practical Examples on False / True Positives

The following examples provide code which is inherently risky at the level of module testing, but the fault may be temporarily disabled at system level. However, these proofs had to be derived manually by inspection of all the relevant code.

These examples are derived from real code. The intention is to show the problem and the fault potential. They are representative for the original code, but not identical and not complete in the sense that they are compilable.

Code	Comment

`#define NUM_ELEMS 5`
`typedefenum {FALSE ,TRUE } TyBool ;`
`typedefenum {NOSUCC,SUCCESS} TyStatus ;`
`TyBoolelemList[NUM_ELEMS]={FALSE, FALSE, FALSE, FALSE, FALSE};`
`void myFunc(){`
`intelemId=0, freeElem = -1;`
`TyBoolfree_elem_found = FALSE;`
`TyStatus status;`
`  while ((free_elem_found == FALSE) && (elemId< NUM_ELEMS)){`
`    if (FALSE == elemList[elemId]){`
`freeElem = elemId;`
`elemList[elemId] = TRUE;`
`free_elem_found = TRUE;`
`    }`
`elemId++;`
`  }`
`  if (TRUE == free_elem_found){`
`    status = OSfunc(paraList);`
`    if (status!= SUCCESS){ /* fault handling */`
`elemList[elemId] = FALSE;`
`    }`
`  }`
`  return;`
`}`
	Here the fault manifests as an error when `OSfunc` returns an error code `!=SUCCESS`.
	The intention of `myFunc` is to search for a free entry in table `elemList` which records e.g. running tasks (the assignment of a tasked to `elemList` is not shown / considered here).
	When the task was not started by `OSfunc` then the entry found has to be released.
	However, when this happens `elemId` was already incremented in the loop above.
	Therefore either a wrong entry is released or an out-of-range condition occurs resulting in memory corruption.
	It is unclear whether static analysers can detect this fault potential.
	During testing this fault can be detected automatically when the test environment ensures that the last element is free only, an error code `!=SUCCESS`is enforced and an out-of-range check is performed.

Fig.III-4: Fault in Error Handling Part

Code	Comment

`intgetSize(intidx){`
`  if (idx>=0 &&idx<MAX_IDX)`
`    return (10+idx);`
`  else`
`    return -1;`
`}`
	The function indicates an error by returning -1
	when `idx` is out-of-range

`unsigned intmyMax=100;`
`unsigned intexpr,len;`
`#define MIN_MACRO(x,y) \`
`    ((x) < (y) ? (x) : (y) )`
`void myFunc(intidx, char *src, char *dest){`
`len=MIN_MACRO((int)myMax,getSize(idx));`
`memcpy(dest,src,len);`
`}`
	Mix of signed and unsigned.
	If `getSize` returns -1,
	`len`gets the value 2^{32}-1 and `memcpy` writes to 4 GB.
	Do static analysers flag the fault potential of the assignment?
	During testing a crash will occur when enforcing an out-of-range condition.

Fig.III-5: Potential for Memory Corruption by Mix of Signed / Unsigned

Code	Comment

`void myFunc(int limit, int`***`incr`***`) {`
`  for (int ii=0;ii<limit;ii+=`***`incr`***`)`
`;`
`}`
	Endless-loop when `incr` == 0
	Here the challenge is to prove in system context that never 0 is passed for `incr`.
	Do static analysers flag this potential fault?
	During testing an endless-loop will occur, when enforcing 0.

Fig.III-6: Potential for Endless-Loop

Code	Comment

`intgetErrMsg(char *errMsg){`
`interrCode;`
`errCode=getErrCode();`
`if (errCode!=0){`
`if (errMsg ==NULL)`
`errMsg =malloc(1024);`
`sprint(errMsg,"ERROR %d \n",errCode);`
`    }`
`  }`
`return errCode;`
`}`
	`getErrMsg`expects that `malloc` always returns a valid pointer.
	Therefore no check on NULL is performed.
	If `malloc` returns NULL, then an error already occurs here.
	Do static analysers flag the potential change of the parameter errMsg as pointer?

`void myFunc() {`
`char *str=NULL;`
`interrCode;`
`errCode= getErrMsg (str);`
`if (errCode!=0)`
`printf("%s \n",str);`
`  ...`
	This function expects that `getErrMsg`ensures valid memory for the error message.
	However, the mistake is:
	`str`is passed by value, and the new ptrallocated in `getErrMsg`is not returned.
	The fault is activated when `getErrMsg`returns !=0, i.e. when an error occurred. This may happen very sporadically.
	When enforcing `getErrCode` to return !=0 for sure during module testing, the fault potential will be detected.

Fig.III-7: Sporadic Inherent Risk due to Temporarily Disabled Fault

Code	Comment

`intmyFunc(short *buffer, intval,`
`unsigned short ind, unsigned short bitWidth) {`
`intindW=ind / 16;`
`intindM=ind % 16;`
`int shifts=16-indM-bitWidth;`
`int mask=((1 <<bitWidth) -1) << shifts;`
`val=val<< shifts;`
`  ...`
`}`
	Even if `bitWidth` is limited to 8 or 16 `shifts` may take a negative value.
	Shift operations with arguments <0 or >31 are undefined.
	The gcc masks the operand with 0x1f and performs the operation. So the result is not what is expected if the operand is out-of –range.
	However, in this case no exception is raised and silent fault propagation may occur.
	`mask` and `val`will both be 0 after the shift. A data stream of 0 may be recognized during testing.
	Do static analysers flag this fault potential?

Fig.III-8: Potential Risk for Wrong Result

Code	Comment

file.c:
`char buffer[1024];`
`int  offset=0;`
task1.c:
`extern char buffer[];`
`extern int offset;`
`intmyWrite(char *buf, char *msg, intlen) {`
`memcpy(buf+offset,msg,len);`
`  offset+=len;`
`  return offset;`
`}`
`int taskBody1(char *msg, intlen) {`
`myWrite(buffer,msg,len);`
`  return 0;`
`}`
task2.c:
`extern char buffer[]:`
`extern int  offset;`
`intmyRead(char *buf, char *msg) {`
`intlen;`
`len=msg[0];`
`memcpy(msg,buffer+offset,len);`
`  offset-=len;`
`  return len;`
`}`
`inttaskBody2(char *msg, intlen) {`
`myRead(buffer,msg,len);`
`  return 0;`
`}`
	Apart from some other issues not discussed here, the addressed fault potential is related to an overflow of the buffer.
	Obviously, the assumption is that an overflow cannot occur in `task1.c` because a sufficient number of data is removed from buffer by `taskBody2`.
	Verification of this code requires knowledge on the synchronization scheme, at least.

Fig.III-9: Potential for Buffer Overflow

Code	Comment

file.c:
`char buffer[1024];`
task1.c:
`extern char buffer[];`
`intmyTCminLength[]={x,y,z,2};`
`int taskBody1(char *tc, intlen) {`
`intret,len,id;`
`  id =tc[0];`
`len=tc[1];`
`  if (len<=myTCminLength[id])`
`    ret=-1;`
`  else {`
`    ret=0;`
`memcpy(buffer,tc,len);`
`  }`
`  return ret;`
`}`
task2.c:
`extern char buffer[];`
`int taskBody2(char *data) {`
`intlen;`
`len=tc[1];`
`memcpy(data,buffer+2,len-2);`
`  return 0;`
`}`
	A telecommand is received in `taskBody1`. Its length is checked by its id and the contents of `myTCminLength`.
	In `taskBody2` it is expected that the telecommand is checked for correctness in `taskBody1`. The data of the telecommand are expected after the header (of length 2).
	An inconsistency can occur if the length of the header is extended e.g. to 3.
	Now, if 2 is changed to 3 in `taskBody2`, but not in `myTCminLength`, then the check may be passed, but
	`len`-3< 0.
	The result is documented in Fig.III-5.
	A literal instead of the number 3 will fix this issue.

Fig.III-10: Potential Inconsistency

IV. The Planned Activity

The goals of the activity are:

- identification and classification of defect types starting with previously identified defects and continuously extending this set with new findings,
- derivation of footprints of verification tools, to better know what a tool can identify and does report as a defect / error,
- investigation of context-dependent defect / error identification, and to provide a feedback to tool vendors,
- getting a clearer understanding of *false positives* which may change to *true positives* when the context changes,
- identification of hidden *false negatives* and real *false positives*.

The set of tools for evaluation shall cover the following domains and fulfill the criterion defined in Sect. I.B (no manual intervention required for fault identification apart from tool configuration):

- symbolic execution
- static analysis
- compilation as a specific case of static analysis
- dynamic analysis.

The reports of all tools shall be merged into a consolidated report at the end (Fig. IV-1).

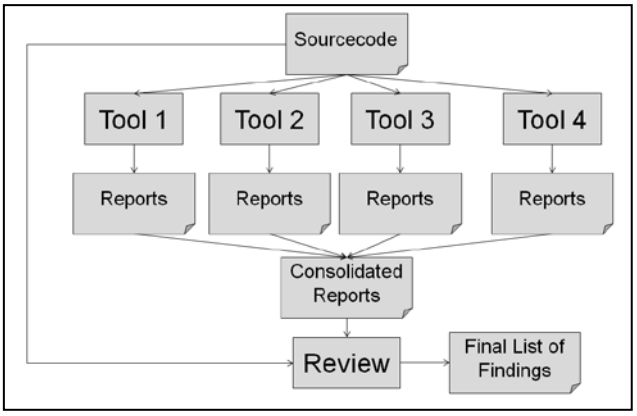

Fig. IV-1: The Evaluation Process

The tools shall be applied to a representative space software package (C and/or C++). The analysis will be carried out without pre-existing knowledge of the defects present. But it will be known which verification activities were previously executed on the software package. This package shall be extended by code examples including known defects serving as a known defect base.

To consider context-dependencies of fault identification (see A3-A4, T3-T4 above) the defects shall be inserted in a complex context. In addition, evaluation shall also consider the case when the defects are present in a simple context.

When applicable, configuration parameters shall be modified to evaluate their impact on fault identification and evaluation effort.

The effort required to identify the source of a defect in the source code shall be recorded.

In a first step the results from every tool are recorded separately. Information is not exchanged in this initial phase (Fig. IV-2).

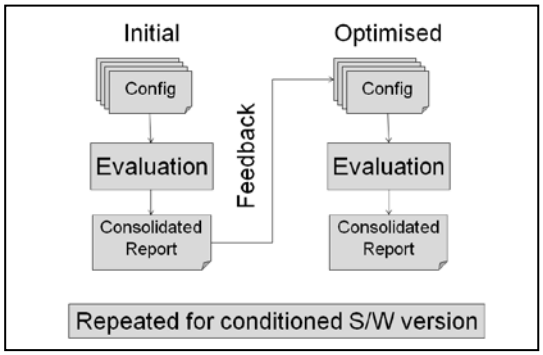

Fig. IV-2: Logic Flow of Defect Reporting

Then the results will be compared, and reports will be analyzed for true or *false positive* status. *False negatives* shall be approximately identified by cross-comparison of reports. Further, the review of reports may lead to additional findings of previously unreported defects, which will be registered as well.

In a second – optimizing – step all defect reports will be considered. In case of *false negatives*, the reason shall be identified and, if possible, evaluated for whether a *true positive* can be obtained by varying configuration parameters.

To let each project decide which criteria for fault identification are of relevance regarding *true* or *false positives*, all *false positives* of type *temporarily disabled* are considered as *true positives* because there is a non-zero probability for fault activation in the sense that the software may enter an invalid state.

Final results shall be available before end of 2015.

V. Conclusions

We expect a major step forward to achieve higher fault coverage by knowing in advance which defects can be identified by the chosen set of verification tools, resulting in a more deterministic strategy for software verification and in minimization of risks.

Minimization of risks requires minimization of *false negatives. False positives* cause an overhead, which in turn may convert *true positives* to *false negatives* because not all positives may be considered within the project timeframe. Mastering of risks therefore requires both minimization of *false positives* and good knowledge on *false negatives* which shall be achieved by diversification of verification methods and tools.

The derived footprints also shall provide a feedback to the tool vendors where they can improve fault identification capabilities of tools and efficiency of report evaluation.

A systematic analysis of fault identification capabilities is a pre-condition to improve tools and verification strategies in the sense of DeMarco: “what you don’t measure, you can’t control [and improve]”.

Due to classification of tools regarding identification of defect types and the required effort, a strategy can be derived how a high number of defects can be identified at a minimum of effort.

In consequence, this should lead to a higher efficiency of the software verification and quality assurance processes as well as increased risk reduction, higher fault coverage and higher quality at reduced costs.

The consideration of context dependency of fault identification suggests that deeper knowledge on fault activation conditions can be achieved by unit testing. Although more defects may be identified than in the system context, it may be worthwhile to know about such conditions, especially regarding maintenance and reuse. However, it is the decision of the project whether this is considered as an overhead or useful information.

The effort for tool qualification should be reduced when tool characteristics are provided in terms of footprints.

REMARK

The community is invited to contribute to the collection of defect types and to suggest verification tools as candidates for future evaluation which fulfil the criterion of Sect. I.B (no manual intervention required for fault identification apart from tool configuration).

REFERENCES

[1] ECSS-Q-HB-80-03A, Handbook on Space Software Dependability and Safety

[2] ECSS-E-ST-40C, Space Engineering / Software

[3] ECSS-E-HB-40A, Software Engineering Handbook

[4] ECSS-Q-ST-80C, Software Product Assurance

[5] ESA ISVV Guide, ESA Guide for Independent Software Verification and Validation

[6] DO178, Software Considerations in Airborne Systems and Equipment Specification

[7] DO330, Software Tool Qualification Considerations

[8] EN 50128, Software for Railway Control and Monitoring Systems

[9] P. Emanuelsson, U. Nilsson, A Comparative Study of Industrial Static Analysis Tools, Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden, Januar 2008

[10] P. Hellström, Tools for static code analysis: A survey, Department of Computer and Information Science, Linköping University, SE-581 83 Linköping, Sweden, Februar 2009

[11] R.Gerlich, R.Gerlich, C.Dietrich: "Fault Identifcation Strategies", Eurospace Symposium DASIA'09 "Data Systems in Aerospace", 26 – 29 May, 2009, Istanbul, Turkey.

[12] M.Temmermann, A Comparative Study of MISRA-C compliancy Checking Tools, Karel de Grote University College, Antwerp, July 2013

[13] G.J.Holzmann, Mars Code, CACM, Feb. 2014, Vol. 57, No. 2, pp. 64-73