IAEA-TECDOC-1140

Effective handling of software anomalies in computer based systems at nuclear power plants

Report prepared within the framework of the International Working Group on Nuclear Power Plant Control and Instrumentation

INTERNATIONAL ATOMIC ENERGY AGENCY IAEA

March 2000

The originating Section of this publication in the IAEA was:

Nuclear Power Engineering Section
International Atomic Energy Agency
Wagramer Strasse 5
P.O. Box 100
A-1400 Vienna, Austria

EFFECTIVE HANDLING OF SOFTWARE ANOMALIES IN COMPUTER BASED SYSTEMS AT NPPs
IAEA, VIENNA, 2000
IAEA-TECDOC-1140
ISSN 1011–4289

Printed by the IAEA in Austria
March 2000

FOREWORD

This report reviews possible types of anomalies that are related to software in nuclear power plants, outlines techniques that can be used to identify anomalies throughout the entire software life-cycle, and discusses important issues that must be considered during anomaly investigation and resolution. Typically, anomalies are identified, investigated and resolved during the normal process of developing or maintaining plant software, where these activities are covered by procedures and tools that are part of this process. Nevertheless, to reduce the number and impact of anomalies under plant operating conditions, it is important to ensure that good plans, procedures and tools are in place throughout the software life-cycle. The need for this was pointed out by the IAEA International Working Group on Nuclear Power Plant Control and Instrumentation (IWG-NPPCI).

The report is the result of a series of consultants meetings held by the IAEA in 1997 and 1998 in Vienna. It was prepared with the participation and contributions of experts from Austria, Canada, Germany, Hungary, the United Kingdom and the United States of America.

The scope of activities described in this report covers a methodology for anomaly identification, anomaly investigation and anomaly resolution. The activities to be done within these steps strongly depend on the safety category of the software, the actual life-cycle phase of the software, the type of the software and the severity of the anomaly.

Special thanks are due to J. Naser (USA) who chaired the working meetings and co-ordinated the work. A. Kossilov of the Division of Nuclear Power was the IAEA officer responsible for this publication.

EDITORIAL NOTE

The use of particular designations of countries or territories does not imply any judgement by the publisher, the IAEA, as to the legal status of such countries or territories, of their authorities and institutions or of the delimitation of their boundaries.

The mention of names of specific companies or products (whether or not indicated as registered) does not imply any intention to infringe proprietary rights, nor should it be construed as an endorsement or recommendation on the part of the IAEA.

CONTENTS

1. INTRODUCTION

The IAEA report "Verification and Validation of Software Related to Nuclear Power Plant Instrumentation and Control" has recently been prepared within the framework of the International Working Group on Nuclear Power Plant Control and Instrumentation and published as Technical Reports Series No. 384 [1]. It provides information on how effective verification and validation (V&V) of computer based control and instrumentation systems can be achieved, and on methods available to support V&V activities. However, to complete the process, further information and guidance is required to assist suppliers, users, and regulators in dealing with the identification, investigation, and resolution of anomalies (deviations from expectation). These anomalies may be found during any phase of the software life-cycle, including the requirements, design, coding, V&V, operation, maintenance, and modification phases. This report addresses these issues and is designed to complement Ref. [1].

1.1. PURPOSE

The purpose of this report is to provide guidance and to discuss the issues that should be considered when carrying out the identification, investigation, and resolution of anomalies. A generalized procedure of this is shown in Fig. 1. The anomalies of interest are those detected in computer based control and instrumentation systems during all phases of the software life-cycle.

This publication is intended to complement related reports and standards such as IAEA Technical Reports Series No. 384 [1], IEEE 1044 standard [2], IEC 880 standard [3], IAEA Technical Reports Series No. 367 [4], IAEA technical documents IAEA-TECDOC-808 [5], IAEA-TECDOC-780 [6], IAEA-TECDOC-952 [7], the IAEA Safety Code 50-C-O [8] and Safety Code 50-C-QA [9].

1.2. SCOPE AND AUDIENCE

This publication is primarily applicable to computer based systems and subsystems of nuclear power plant applications that are important to either safety or operation. This report discusses how to identify software anomalies in the system and subsystem, but other types of anomalies may also be identified when these techniques are used. In the investigation and resolution phases, this report only considers the situations when the fault is in the software or when software modifications can be implemented to work around hardware or interaction faults. The intended audience for this report includes plant management, system responsible engineers, operations staff, maintenance staff, software engineers, suppliers, and regulatory staff.

1.3. REPORT OVERVIEW

This report is concerned with the identification, investigation, and resolution of anomalies. It first describes a process to determine that an anomaly exists through analysis (static and dynamic) and behavioural responses. The anomaly detected through the behaviour of the system can be manifested by either an unexpected behaviour or by the lack of an expected behaviour. After an anomaly has been recognized, the next step described is the investigation of the anomaly to determine the cause of the anomaly, the source of the anomaly, and the consequences of the anomaly. Causes of the anomaly can come from a fault in the software, hardware, or in interactions between various elements of the system. The final step described is the resolution of the anomaly. This resolution can be achieved in several ways including removing the fault, mitigating the fault, changing procedures to work around the fault, and documenting the fault without modification. The appropriate resolution is dependent on the consequences of the anomaly and the circumstances in the plant and must be determined on a case-by-case basis. Configuration management plays an integral role in the identification, investigation, and resolution activities.

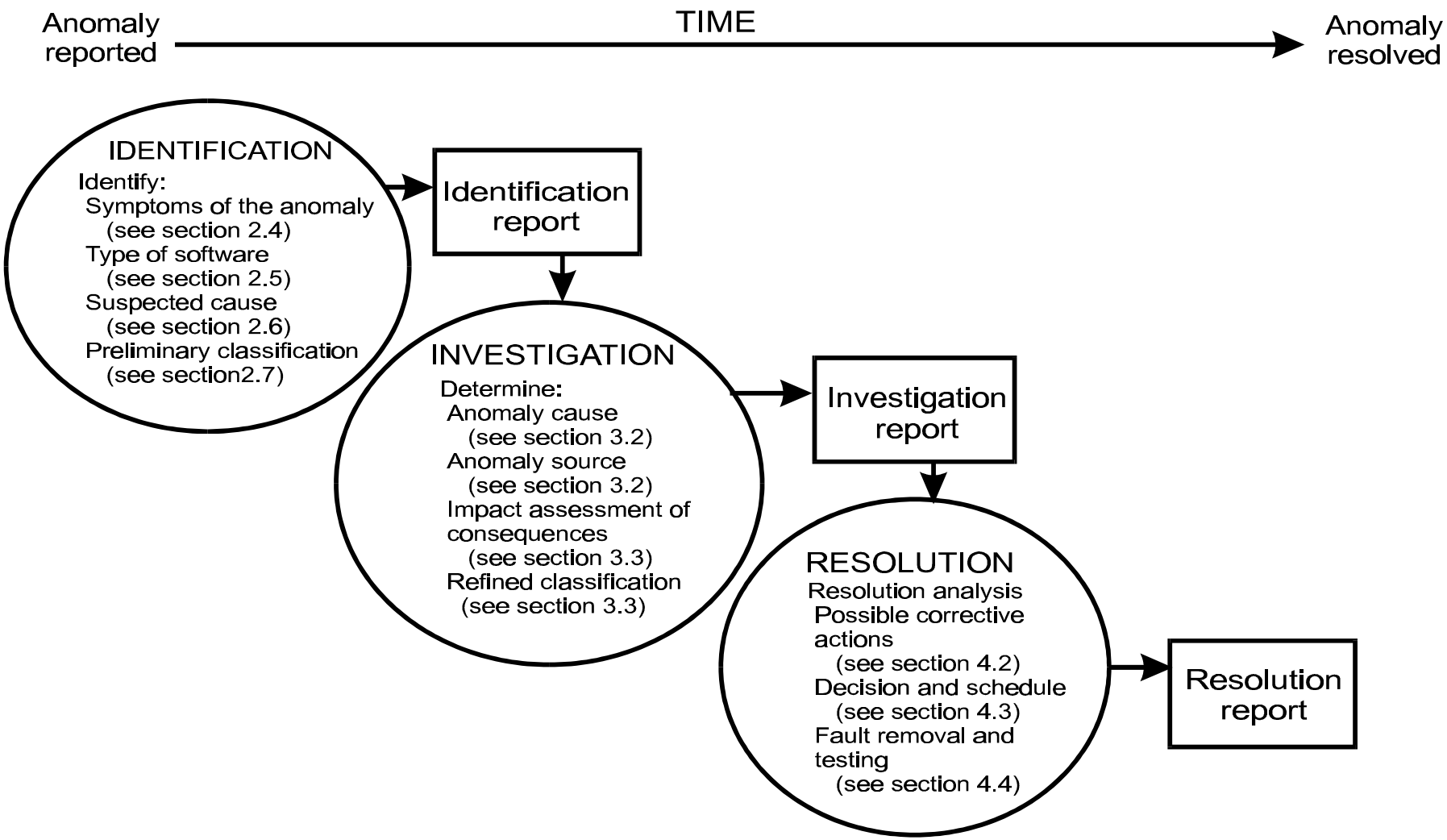

FIG. 1. Identification, investigation and resolution of software anomalies

A policy for the treatment of anomalies is required after software development concludes. After this time, unnecessary corrections might require an extensive revalidation effort and could cause the unintentional insertion of new faults when removing old ones. In this case, the priority of the severity is likely to provide rational guidance for deciding to do one of the following:

- Investigate and resolve the anomaly immediately.
- Wait to investigate the anomaly at the next best opportunity (e.g., maintenance phase) postponing the decision on its removal until full analysis information is available; or
- Consider the anomaly significance too low to justify any additional investigation and resolution effort. For this case, detailed documentation of the anomaly should be produced and logged for consideration during the next revision of the document or code..

The structure and linkage of the sections of this report are summarized as follows:

The remainder of Section 1 defines the terms used in this report, discusses safety categories of systems, severity levels of anomalies, phases of the software life-cycle, and types of software. These four parameters are important due to the effect they have on the identification, investigation, and resolution activities described in the remainder of this report. Section 2 provides guidance for the identification of anomalies, mainly the gathering of evidence and the preliminary classification. The evidence is categorized based on the techniques applied to identify the anomaly. Section 3 provides guidance for the anomaly investigation and consequence assessment by building upon the information gathered from the anomaly identification phase. In particular, the cause of the anomaly, the source of the anomaly, and the consequences of the anomaly are discussed. Section 4 provides guidance on the resolution of the anomaly based upon the consequence assessment described in Section 3. Corrective actions and the impacts of the changes are discussed. Section 5 provides the recommendations from this report. The appendix provides an example of the data that should be recorded during anomaly identification, anomaly investigation, and anomaly resolution.

1.4. DEFINITIONS

Definitions of some of the terms used in this publication are given below. They are consistent or identical if marked *, with those used in other current IEEE, IEC or IAEA codes, guides and standards.

— *Anomaly** [2]
An apparent deviation from a general rule, method, analogy, specification, or result. Anomalies may be found during the review, test, analysis, compilation, or use of software products or applicable documentation.

— *Behavioural response*
Observable system reaction to specific input stimuli under particular operating conditions.

— *Cause*
The cause is the static feature in one of the formal or informal documents responsible for the identified anomaly.

— *Consequences*
The consequences are the potential impacts of the anomalous behaviour on the plant.

— *Dynamic analysis*
Class of methods determining input dependent behavioural properties of a software document by executing the software under particular input and state conditions, or simulating its behaviour by animation.

— *Equipment** [10]
One or more parts of a system. An item of equipment is a single definable (and usually removable) element or part of a system.

— *Fault*
A fault is an anomaly such that the performance would not be as specified if the fault is activated.

— *FSE* (Functions, and the associated systems and equipment)* [10]
Functions are carried out for a purpose or to achieve a goal. The associated systems and equipment are the collections of components and the components themselves that are employed to achieve the functions.

— *Function** [10]
A specific purpose or objective to be accomplished, that can be specified or described without reference to the physical means of achieving it.

— *Postulated initiating events (PIEs)** [11]
Identified events that lead to anticipated operational occurrences or accident conditions and their consequential failure effects.

— *Resolution*
Any activity that results in the removal or mitigation of the effect of the discovered fault.

— *Signal trajectory** [1]
A signal trajectory is a set of time histories of a group of signals, or equipment conditions, which result in a particular output state of the system.

— *Source (of an anomaly)*
The life-cycle phase when and reason why the cause of the anomaly was introduced.

— *Static analysis*
Class of methods determining invariant properties of a software document by processing its contents, without executing the software, nor simulating its behaviour by animation.

— *System** [10]
A set of interconnected elements constituted to achieve a given objective of carrying out a specified function.

1.5. PRINCIPLES OF CLASSIFICATION

Classification of FSEs, anomalies, life-cycle phase, and type of software will help prioritise the investigation and resolution of anomalies and provide support and guidance in determining process improvements. Early in the design process, it may be straightforward to identify the cause of anomaly but difficult to identify the severity. Later in the process, it may be straightforward to identify the severity, but difficult to identify the cause, particularly when the cause involves contributions from more than one source.

The categorization in terms of safety of the software based system containing the anomaly and the severity of the anomaly in terms of safety and performance are important factors in determining anomaly investigation and resolution priority [12].

Recording of the life-cycle phase during which the anomaly has been detected and the life-cycle phase in which the anomaly was introduced is useful in determining what investigation and resolution activities are necessary. The recording of this life-cycle information would also help highlight what areas might need process improvements.

The type of software (new, reused, proprietary, etc.) in which an anomaly has been observed will be of concern. The availability of information, or lack thereof, could limit the completeness of the anomaly investigation. Similarly the amount of access to the software could limit the resolution options. The type of software could also indicate what other systems using the same software might be affected by any faults associated with an anomaly since the systems use the same software.

1.5.1. Safety categories of functions and the associated systems and equipment

IAEA Safety Code 50-C-D [11] and Safety Guides 50-SG-D3 [13] and 50-SG-D8 [14] establish the concept of classification of nuclear power plant systems according to their importance to safety. The principles of IAEA classification have been interpreted by the IEC in IEC 1226 [10] which identifies categories A, B and C for FSEs that are important to safety. It is assumed within this publication that FSEs have already been categorized before any anomaly identification, investigation, or resolution activities are attempted. The category descriptions are the following:

— Category A
Category A denotes the FSEs which play a principal role in the achievement or maintenance of nuclear power plant safety. These FSEs prevent PIEs from leading to a significant sequence of events, or mitigate the consequences of PIEs. Category A FSEs may be accomplished automatically or via manual actions, providing such actions are within the capabilities of human operators. Category A also denotes FSEs whose failure could directly cause a significant sequence of events. Category A FSEs have high availability and reliability requirements. They may be limited in their functionality so that their availability and reliability can be very confidently guaranteed.

— Category B
Category B denotes FSEs that play a complementary role to the Category A FSEs in the achievement or maintenance of nuclear power plant safety. The operation of a Category B FSE may avoid the need to initiate a Category A FSE. A Category B FSE may improve or complement the execution of a Category A FSE in mitigating a PIE, so that plant or equipment damage or activity release may be avoided or minimized. Category B also denotes a FSE whose failure could initiate or worsen the severity of a PIE. Because of the presence of Category A FSEs to provide the ultimate prevention or mitigation of PIEs, the safety requirements for the Category B FSEs need not be as high as those for the Category A FSEs. This allows, if necessary, the Category B FSEs to be of higher functionality than Category A FSEs in their method of detecting a need to act or in their subsequent actions.

— Category C
Category C denotes FSEs that play an auxiliary or indirect role in the achievement or maintenance of nuclear power plant safety. Category C includes FSEs that have some safety significance, but are not Category A or B. They can be part or the total response to an accident but not be directly involved in mitigating the physical consequences of the accident.

1.5.2. Severity of anomaly

The three anomaly severity levels defined below are primarily based on safety implications and serve as examples of classifying anomalies. However, other considerations such as plant performance, reliability, impact on other similar FSEs, and the cost of repair could be given more weight and different severity levels could then be defined.

The degree of severity of the anomaly is classified according to the following criteria applied to the actual or expected on-line operation of the software.

A *severity level 1* anomaly is one which either:

— directly prevents a function important to safety from operating;
— directly causes a function important to safety to operate when not required; or
— directly reduces the system reliability or availability by incorrectly preventing equipment from operating when required to do so.

A *severity level 2* anomaly is one which is not of level 1 and either:

— reduces the possible reliability or availability of a required function important to safety, by degrading a support function, maintenance, test or calibration function;
— prevents information transfer or provides incorrect information transfer to an external system of a more severe category;
— is software information, configuration information, or software issue level and is incorrect such that interfacing, modifications or revisions would be incorrect;
— implements a function in code or appears as documentation significantly below the agreed standard for the system concerned; or
— implements a function in a manner which could fail in expected abnormal hardware conditions, but does not fall into severity level 1.

A *severity level 3* anomaly is one which is not of level 1 or 2 and either:

— could significantly impair documentation clarity or quality, where no functional defect exists;
— could impair clarity of auxiliary information on performance, operation or function; or
— could impair auxiliary aspects not central to the functions important to safety.

If the anomaly does not fall into the above levels, then it is identified as not significant. However, a fourth category (severity level 4) could be introduced for anomalies related to faults or failures of a trivial nature (e.g., spelling mistakes in code comments). Such anomalies identify errors which one may wish to correct when an opportunity presents itself.

1.5.3. Phases of the life-cycle

The life-cycle phases described in IEC 880 [3] and illustrated in the IAEA Manual on Quality Assurance, Technical Reports Series No. 282 [15] are:

— System requirements specification;
— Computer system specification;
— Software design;
— Coding;
— Computer system integration;
— Integrated computer system test;
— Validation and commissioning test;
— Handover; and
— Operation, maintenance, and modification.

Note, that in the context of this report the “coding phase” includes the module testing (through the use of static analysis, as well as dynamic tests), i.e., every activity to implement the requirements of the Software Design Document to make the fully formalized documents ready for system integration. It also should be noted that modifications can be required at any phase in the life-cycle. Any modifications that are done should be taken through all of the appropriate earlier phases.

Detailed descriptions of these life-cycle phases are given in Section 3.1 of Ref. [1].

1.5.4. Types of software

The identification, investigation, and resolution of any software anomaly is dependent on the type of software involved. For example, investigation and resolution of an anomaly suspected in proprietary software could be difficult due to the lack of the source code and design documentation. The code supplier may need to be consulted either to help investigate and resolve the anomaly or to

provide extra material to aid investigation and resolution. The following four types of software are described in Ref. [1] and will be used in later sections of this publication:

— New software — all software written specifically for the application;
— Existing accessible software — typically software from a similar application that is to be reused and for which all the documentation is available;
— Existing proprietary software — typically a commercial product or software from another application that meets all or part of the current application requirements but for which little documentation is available; and
— Configurable software — typically software that already exists but is configured for the specific application using data or an application specific input language.

2. IDENTIFICATION OF SOFTWARE ANOMALIES

2.1. INTRODUCTORY REMARKS

This section is devoted to the description of the task of identifying the presence of software anomalies throughout the software life-cycle. In particular, the scope of this discussion will include:

— the dependency of this task on project specific characteristics such as the phase of the software life-cycle during which the anomaly is detected, and the technique applied to detect it; and
— the provision of input to the successive tasks of investigation and resolution, which make use of information made available during the identification task.

As already mentioned (see definitions), the term *anomaly* is used (in accordance with IEEE 1044 [2]) as "an apparent deviation from a general rule, method, analogy, specification, or result". This definition applies to the complete software life-cycle and includes, beside actual faults, any deviating aspect requiring further analysis or explanation. The term *identification* denotes the process of recognizing, by provision of evidence, the presence of some anomaly.

The term *evidence* may denote observations of a different nature, according to the identification techniques applied, as described below. It includes observations based on the mental review of documents, code execution, and indications provided by automatic analysis tools.

The term *presence of anomaly* indicates some evidence is provided that an anomaly exists, according to which further study or justification is required. It does not necessarily indicate that detailed information on the nature of the anomaly, its cause, or location, is available.

The type of evidence confirming the presence of an anomaly mainly depends on the class of identification technique providing this evidence, and on the life-cycle phase, during which the anomaly identification takes place. On the other hand, the evidence available has a fundamental impact on the ease of diagnosing during the ensuing investigation phase. In view of such considerations, the main aspects characterizing anomaly identification and evidence will be discussed in the following sections.

2.2. LIFE-CYCLE PHASE AT IDENTIFICATION TIME

The following (possibly overlapping) classes of life-cycle phases are grouped for convenience in this subsection to reflect their common characteristics in terms of anomaly identification:

— development phases of the life-cycle, in which software development has only progressed up to the production of *documentation in informal language*, such as the specification and design phases in natural language;

— development phases of the software life-cycle, where *documents in formal language* are available, i.e., documents written in a rigorous language with a well defined semantics, which can be syntactically and logically analysed, such as the formal specification and design phases, and the coding phase for arbitrary programming languages (i.e., both source and object code levels);
— development phases of the software life-cycle, where *executable software documents* are available, i.e., documents allowing simulation of dynamic behaviour, such as the conventional coding phase including compilation (object code level), and formal or semi-formal specification phases allowing dynamic animation (e.g., animation of Z specs, simulation by Petri nets); and
— usage phases, after the system was *validated for use*, and all documents required for its validation are available, such as the operation and maintenance phases.

2.3. TECHNIQUES SUPPORTING ANOMALY IDENTIFICATION

2.3.1. Informal checks

Documents that are only written using informal language (typically natural language) merely allow for checks in a quite unsystematic, rather 'ad hoc' style. The main implication of this is that such checks consist of mental activity, since language ambiguity does not allow for machine support for either symbolic execution nor for analysis. The only systematic procedure underlying such techniques is a rigorous distribution of roles to be fulfilled by each of the participants (moderator, author, testers, etc.). According to the roles, at high level design phase the following checking activities should be carried out:

— *inspections*, where checks are confined to the examination of existing documents being mutually consistent and complete, and fulfilling predefined rules for document control; and
— *reviews*, which, in addition to the above, also include plans for future phases, and constructive suggestions for improvement.

At low level design and coding phases, a check of finer granularity is required; this should be carried out either in the course of:

— *walkthroughs*, where the author explains the details of the document to the participants, who check the explanation by mental activity; or by
— *desktop inspections*, where only one participant (typically an independent assessor) mentally reviews the document.

The main advantage of this identification technique is that it is applicable to the earliest development phases and thus allows (at least in principle) for a timely detection of any kind of fault created in the initial phases. However, reality often also reveals the limits of such informal checks. Their success tends to be extremely subjective and, due to the non-reproducibility of the results, even dependent on unpredictable, momentary circumstances. On the whole, in the case of informal development, this class of identification techniques (desktop inspections especially) cannot be replaced by any other alternative in the earliest phases. Therefore, the use of this class of techniques is highly recommended. Nonetheless, its potential for errors and subjectivity should be kept in mind.

Since this identification technique is mainly characterized by mental activities focused on static objects (i.e., without code execution), it has a major advantage in terms of its impact on anomaly investigation. In fact, anomaly identification by informal checks is cause oriented, rather than effect oriented. In other words, most findings are explicitly related to the location and the cause of the anomaly identified.

2.3.2. Static analysis

The goal of static analysis is the determination of static document properties (in particular of source code), i.e., properties not related to the execution under specific operational conditions, but rather to invariant properties with respect to syntactic or semantic rules of the underlying language.

During development phases that provide documents in a logically well defined notation, systematic procedures can be applied to support the automatic analysis of documents by automatic tools. In fact, the use of logically well defined notation offers the additional advantage of overcoming the main drawbacks of human subjectivity of informal checks by standardizing the checking procedure in an objective, repeatable way. Apart from ensuring objectivity and reproducibility, this approach permits the process to be strongly supported by automatic tools, allowing the performance of analysis of greater complexity than is possible manually.

2.3.2.1. Syntactic analysis

With respect to syntactic rules, static analysers examine a formal document to identify its syntax and provide relevant information on control and data flow. Basic checks on syntax compliance to predefined coding rules are already offered as standard support, typically as a compiler analysis option or stand-alone program (e.g., C-LINT). Also classical software measures, for example those related to module size, are usually automatically provided as a matter of routine even in projects with modest dependability demands. Deeper information is offered by more sophisticated static analysis techniques which are described in Ref. [16]. These techniques present their results in form of control flow graphs, cross-reference tables for variables, structural complexity metrics, etc.

The information collected by static analysers helps to identify *anomalies* in the broader sense of the terms, i.e., specific faults, as well as features of the anomaly, which point to suspicious parts of the code where it is worth devoting further verification effort. A typical fault that is easily detectable by static analysis is the use of undefined variables. Also the successive (re-) definition of the same variable without any intermediate use is easily verifiable by syntax checkers and, though not necessarily incorrect, is likely to provide an indication of unintended behaviour. Also complexity metrics can be useful in identifying suspicious looking *outliers*, that is when software modules differ too much in terms of their size or in their structural properties. This information can provide an indication of the software sections requiring a particularly extensive verification effort.

2.3.2.2. Semantic analysis

With respect to semantic checks, static analysers can provide considerable support by identifying so-called *path expressions*. These indicate logical relationships among variable contents, in order for a program input to traverse a predefined part of the code control flow. This information is in itself insufficient to detect faults, but again it can efficiently serve the purposes of

— supporting completeness and consistency checks by analysing and comparing decision criteria for different control flow paths;
— indicating the existence of inaccessible code, i.e., code portions not reachable by any input configuration (identifiable by non-satisfiable path expressions); and
— supporting test case selection during structural testing (thus indirectly contributing to anomaly identification, as described below).

One of the main drawbacks of semantic analysis is caused by its limited applicability to complex code structures. For example, path expressions are only easily resolvable for non-complex code portions, such as linear instruction sequences or ‘for’-loops, whose number of iterations can be statically determined, as in the case of constant boundaries and for the incrementation of loop counters. In more complex cases, where this condition is not fulfilled (e.g., ‘while’-loops, whose

repeating condition depends on the value of variables changed in the loop body), compact and complete information on input conditions for path traversal cannot generally be provided automatically. In such cases, semantic analysis must be restricted to path segments, by checking that particular logical expressions of variable values are fulfilled at specific points in the program. These so-called *assertions*, typically loop invariants, have to be provided manually by the program developer (in form of pre- and post-conditions), and can be successively verified by automatic theorem proving. More generally, assertions also provide suitable rationality checks making explicit assumptions about the software system state at specific execution points. Apart from supporting semantic analysis, assertions can also contribute to anomaly identification during operation, when they are used as detection mechanisms for exception handling procedures.

Generally, theorem proving aims at verifying a logical statement by repeatedly applying logical transformations from a set of given logical rules on theorem assumptions (pre-conditions), in order to derive a stated theorem (a post-condition). In general, theorem proving cannot be fully automatic, but often needs to be supported by intermediate manual steps in an interactive way. The complete proof can then be processed by so-called *proof checkers*, which are capable of verifying the syntactic proof correctness. A particular class of theorem provers is devoted to the verification of correct refinement from one development phase to the next by discharging so-called *proof obligations*. These are conditions that ensure that lower, more detailed design levels refine the higher ones by fulfilling all the higher level requirements while reducing their original degrees of freedom through implementation of abstract data structures.

For the purpose of anomaly identification, human interaction offers certain advantages. In fact, usually automatic theorem provers do not provide the user with helpful information in case of proof failures, which might be due to incorrect theorems or to the inability of the prover. One type of automatic proving technique, which is capable of overcoming this limit, is the so-called *model checking*, whose application scope is restricted to finite state automata. In order for safeness or liveness properties to be verified for arbitrarily long system state sequences, which are usually expressed in temporal logic, efficient tools have been developed in recent years to check the validity of these properties. In this particular case, automatic support is not only able to prove correct system properties, but it can also provide counter-examples for invalid logical attributes. Though model checking is not aiming at verifying complete functional correctness, its ability to provide counter-examples offers valid support for anomaly detection concerning specific system requirements, which are often related to system safety and availability.

Summarizing, the advantages of static analysers are their full reproducibility and the static universality of their outcome. This outcome is based on the analysed documents as such, and not on the expected behaviour of the software under particular running conditions. For the same reason, the potential anomalies identifiable with these techniques are exclusively those related to the logical consistency of one or more formalized levels. Checks for validity of system properties and for functional correctness are based on reference documents on system or software requirements, assumed to be correct themselves. Further dependencies of software behaviour on the envisaged computer environment (e.g., hardware and operating system) are not covered at this stage. They have to be addressed by the activities described in the following sections.

2.3.3. Dynamic analysis

During development phases of the software life-cycle providing executable documents, i.e., those documents allowing the simulation of dynamic behaviour, systematic procedures may be applied to verify behavioural correctness under specific circumstances. Usually this involves code execution by testing. However, it is applicable to earlier development phases, if appropriate formal languages or semi-formal descriptions were used, allowing the simulation of dynamic behaviour by animation techniques.

The main advantage of this class of anomaly identification techniques is the fact that they can actually relate (as much as possible, depending on the testing phase) to the real behaviour (which may deviate from the envisaged one). The more testing has progressed from unit testing through integration testing towards system testing, the better dynamic analysis can reveal anomalies statically unidentifiable, since they refer to not strictly functional, and possibly non-formalized, requirements (e.g., timing aspects, operating system performance, hardware constraints, module interfaces, incomplete requirements, etc.). On the other hand, any observation provided by dynamic analysis exclusively refers to particular running conditions, represented by input data selection, internal states, software environment, etc., and should not be taken as representative of general behaviour. Dynamic analysis allows the identification of the presence of anomalies, but cannot guarantee their absence.

Among the main limitations of testing is the reproducibility of the testing outcome, for example in case of non-deterministic software systems or in the case of systems which do not have a well defined initial state. To avoid the latter example, it is recommended to specify and implement software based systems such as to ensure their re-initialization to a starting state (typically by reset). In this way, anomalies observed by dynamic testing can be reproduced by recording the complete signal trajectory to reproduce the symptom.

Dynamic analysis requires an *oracle* (or gold standard), i.e., some mechanism capable of determining whether the observed behaviour corresponds to the expectations, or not. This decision is not always obvious and may require some simulation, usually based on simplification of the system.

Apart from the level of testing, the following different strategies for test case selection can be distinguished:

— functional testing;
— structural testing;
— statistical and random testing;
— full-scope simulation testing;
— performance testing;
— back-to-back testing.

Due to the complementary nature of these tests, it is recommended that as many of these tests be applied as practical.

Functional testing aims at verifying that all functionalities specified in the higher level requirements are actually implemented at lower level design and code. To verify this property, the lower level executable document is stimulated in a *black-box* fashion, i.e., without regard of the internal structure, focusing on input selection and output determination. Anomalies identifiable by this dynamic procedure mainly consist of missing or inoperable functionalities which are required to be included in the system.

Structural testing aims at verifying that only functionalities specified in higher level requirements are actually implemented at lower level design and code. To verify this property, the lower level executable document is stimulated in a *white-box* fashion, i.e., focusing on the internal structure. Anomalies identifiable by this dynamic procedure mainly consist of unintended features which need to be removed. At code level, automatic support is provided by testing tools and static analysers to aid in test case selection and to record the gradual achievement of structural coverage. Usually at least branching coverage is recommended, path coverage often requiring a prohibitive testing effort.

Statistical and random testing aim at verifying that the specified functionalities are correctly performed for particular input configurations, which may not be explicitly incorporated in the specification documents. More specifically, statistical strategies are based on black-box testing with

inputs chosen to simulate expected operation by representative signal trajectories. In contrast, random strategies are based on arbitrary random test case generation. In both cases, anomalies identified are likely to concern requirements ambiguity or incompleteness. In spite of the considerable effort required to perform them, both strategies have proved to be able to support anomaly identification in real-world applications by encouraging the testers to observe system behaviour under realistic or improbable data combinations that had not been considered explicitly during previous, more focused testing phases.

Full scope simulation testing aims at verifying the full system functionality for validation testing, especially for control or operator interface systems. These tests are performed on hardware identical to the final system, with all inputs stimulated and outputs read by a simulator that models the controlled system as completely as possible. For example, in some cases plant control systems can be tested over most of the plant operating range, including many design basis incidents, using full scope training simulators. In other cases, simulation models of the controlled system are developed in parallel with the control system software to perform full scope simulation testing before delivery and commissioning. Applying simulation testing one has to be aware that noise free data generated by full scope simulators may differ from noisy behaviour to be expected in real operation. Therefore, dynamic analysis based on simulation should be expanded by additional runs especially conceived to test for system robustness.

The full scope simulator testing is very effective for validating effectiveness and adequacy of the human–machine interface (HMI) of a control software. It is usually done by using specifically selected scenarios simulating incidents which require complex actions and decisions from the plant operators.

The different activities testing a system's performance fall into three categories: tests aiming at assessing the extent of usage of different system resources, tests to evaluate the timing properties of the system, and the tests to evaluate the numerical accuracy of the results of the system. The typical system resources are the CPU capacity, the operational memory and the backup storage capacity. Typically, the system requirements prescribe limits for using the system resources, though there are some general rules that have to be followed in any case:

— the CPU load, averaged over a time period characteristic for the system speed, must not exceed 80% with the highest anticipated load, to allow some margin for unforeseen events and modifications;
— the usage of operational memory must converge to a constant value after some period of continuous running of the system;
— the backup storage usage also must allow some margin (e.g. 10–20%).

The validation of timing requirements is typically rather complicated in case of distributed systems. In general, the timing characteristics of the system is measured under typical conditions and under some worst case conditions. The numerical accuracy of the results produced by the system can be evaluated, for example, by varying artificially all the input parameters through their valid ranges.

A particular testing strategy applicable to redundant systems consists of automatically comparing the outputs of two or more diverse software variants executed with identical inputs and in initially identical internal states (so-called "back-to-back testing" or "comparison checking"). Failures are analysed in case of discrepancies. The drawback of this technique lies in its fault detection capability, which is limited to the case of result disagreement; therefore, it is unable to reveal identical failures. On the other hand, the effort required to carry out such a testing phase is considerably lower than for conventional, non-redundant tests. In fact, a simple result comparison used as fault detection mechanism avoids a laborious result verification. Thanks to the lower cost demands, this testing strategy can be carried out for considerably longer runtimes than usually possible for other dynamic analysis techniques.

2.3.4. Operational behaviour

Once the system is in use, it is not usual to repeat software testing periodically to identify anomalies in validated programs is not normally done. Due to the static nature of software, these test results should not vary with time. Nevertheless, the environment in which the software operates can vary, possibly resulting in unexpected stimuli from the environment that may cause a previously dormant fault to produce an anomaly.

Therefore, deviations from expected behaviour might be identified by:

— *on-line* observations during operation,
— *off-line* review of plant data acquired during operation, or
— *off-line* observations during periodic maintenance tests.

In all cases, it is important that the software and system documentation accurately reflect the intended behaviour of the system, and that plant operators and maintainers have adequate training on the system, so that anomalies will be recognized when they occur.

During operation, for example, software or hardware faults may result in unintended or unexpected system behaviour. This behaviour may include not responding to stimuli or error conditions, as well as responding inappropriately. Fault reports automatically generated by computer systems should also be reviewed for possible indications of software anomalies. These circumstances include, for example, the overriding of one subsystem in a triplicated diverse system, or fault reports from internal self-consistency checking.

Historical plant data acquired during operation can also be reviewed off-line to check for anomalies in the system or components. While these anomalies are often hardware related or indicative of changes in the operation state of the components, software faults can also result in small deviations from expected operation that is not immediately apparent on-line.

The possibility of an unexpected interaction between the computer hardware, software, and plant environment (including other computer based systems) can result in system anomalies that, at first glance, are not believed to be software related. However, hardware faults can cause the software to behave in an unexpected manner. Similarly, software faults can cause hardware to behave in an unexpected manner. Thus all observed anomalies involving the system should be identified, and the cause of the anomalies investigated. The hardware software interaction can also result in software faults being resolved through hardware changes, and hardware faults being resolved through software changes.

Anomalies may also be observed during periodic system maintenance tests that may result in the activation of a hardware software interaction not exposed during system installation, commissioning and operation.

2.4. SYMPTOMS

The symptom of a software anomaly is the collection of evidence available for the presence of the anomaly. As mentioned in earlier sections, the nature of the evidence mainly depends on the techniques used to identify the anomaly. It may include

— the observation of an *inconsistency* with the procedures or with best practice rules of programming which may negatively affect program understandability and maintainability;
— the observation of *ambiguity or incompleteness* in a document likely to propagate to later life-cycle phases by causing an actual fault;

— the observation of *incorrect*, or at least unexpected behaviour (this may be based on subjective judgement);
— the *failure to prove* a theorem on functional correctness, which might be due to invalid document annotations; or
— the provision of a *counter example* to a system requirement, which may be manually provided by the analysing personnel, or automatically produced (e.g., by a model checker).

It is extremely important, once the presence of an anomaly is suspected, that all the information available from behavioural observations should be recorded, not just the actual behaviour observed (see for example Section A.1 of the Appendix). The records should also include all the circumstances at the time of the anomaly observation, in particular the values of input trajectories and internal states, to ensure the symptoms can be reproduced and a thorough analysis of the anomaly can be performed. Such an accurate record evidently depends on a rigorous and transparent configuration management.

2.5. TYPE OF SOFTWARE

The completion of the activities described above can be limited by the amount of documentation available for specific software components. In the case of proprietary software, typically commercial products, it may not be possible to obtain all of the desired information. In such cases often just the object code is available from the vendor so that the applicable identification techniques are mainly limited to *black-box* behavioural observations.

It might only be possible to localize the fault in commercial packages by means of checking intermediate results. In some cases, no more than an estimate on which proprietary section of the software is responsible for the anomaly might be possible. The expansion of the use of commercial packages to provide standard services is becoming of crucial importance. Support may be provided by the vendor in the form of detailed information on the operational experience gained so far by other users of the same product, as well as on problems reported during its usage.

In the case of reuse of pre-existing accessible software, typically software from a similar application, in general all the documentation required for the purpose of anomaly identification can be assumed to be available. Therefore, with respect to the information needed, reuse of this type of software does not differ from the case of software newly written specifically for an application. However, positive experience with a module may lead to cursory verification and an overestimate of its reliability, especially if the module has already been validated for a safety-critical application. In such cases, one tends to identify the intrinsic reliability of a reusable function with its suitability to provide a similar service within a new software system. The importance of a detailed analysis of the module interface with other software components is often underestimated when only relying on the correct service provided in similar past projects. In such cases, it is recommended that verification techniques are applied with focus on the interactions of reusable modules with other system sections. In particular, it should be considered whether the parameters the module depends upon actually reflect the new process characteristics. Failure to do this can lead to parameters that are no longer valid (e.g., Ariane 4 and Ariane 5 [17]). Also the safety relevance of reusable software may vary with the application environment; for instance, when electromechanical interlocks originally used are removed from a new system version, like in the case of the medical device Therac-25 [18, 19]. It is important to demonstrate that the reusable software achieves the requirements for the new application.

An important set of reusable software is design, support and analysis tools. Even though there has been considerable past experience with them on other applications, it is important to evaluate and confirm their applicability to the present application.

2.6. SUSPECTED CAUSE

Depending on the evidence available, the cause of the anomaly identified may already be suspected at this stage, for instance when using informal checks. In fact, these techniques explicitly localize the anomaly within the document observed and indicate the reason for a potential deviation.

Static analysers may (but not always) provide the explicit reason for the identified anomaly. More often, they merely point at some irregularity implying the presence of the anomaly, for example outlying modules with respect to given metrics, or inaccessible code portions. The task of determining why the anomaly occurred has to be completed in a later phase.

Theorem proving failures and counterexamples are also likely to provide insufficient information for a diagnosis at this stage. Depending on the granularity of code assertions, or on the particular counter example produced, it may be possible to identify where the fault might be isolated during investigation. It is even more difficult to determine the responsible faults in the case of unexpected observations, especially if they do not relate to specific functional test cases.

On the whole, progress achieved in the last decade by automatic analysis support tools and by powerful techniques for formal deduction has considerably improved the process of timely anomaly identification and cause determination with respect to purely logical faults. In fact, as soon as a development phase (be it specification or design) is completely formalized, the process of mapping this formalized description to an adequate code language (be it source or machine code) is well supported by theorem proving and automatic code generation. This process compresses the development phase, by reducing the number of intermediate design levels.

On the other hand, due to the progress achieved at design and coding phase and to the flexibility offered by software, a tendency to implement systems with increasingly complex requirements is observed. Thus, while anomaly sources are reduced in the purely logical domain of program design and coding, new problems tend to originate at the level of requirements specification. The interaction between the digital system and the plant has to be defined on the basis of both the physical properties of the technical process and the logical properties of digital control.

In recent times, the growing complexity of embedded control systems tends to cause difficulties in specifying correct system requirements. When dealing with high logical complexity, there is a remarkable and partly understandable tendency to avoid the explicit inclusion of assumptions about the underlying physics. On one hand, this allows concentration on (or even to formally verify) the validity of logical system properties. On the other hand, the validity of the properties verified is restricted by their level of abstraction, which for the reasons above may exclude fundamental physical dependencies.

A number of safety related events in recent years (e.g., the Warsaw accident of an LH-Airbus 320 [20]) were related to unforeseen behaviour under particularly rare input combinations. It is extremely difficult to identify such anomalies during software development. Even if very infrequent scenarios are generated by random testing, their output is likely to be checked against the software specification document, and therefore be classified as behaving according to requirements. For this reason, it is recommended starting the requirements analysis at a description level as early and exhaustively as possible. An optimal combination of logical discreteness and physical continuity involves a trade-off between decidability enforced by formal deduction and expressiveness supported by physical modelling. Some recent approaches in this direction make use of hybrid modelling languages and analysis techniques such as Time Petri Nets [21], Data Flow Methodology [22], Hybrid Automata [23].

2.7. PRELIMINARY CLASSIFICATION

A preliminary estimation of the severity of the consequences of the anomaly may already be possible at this point. Such an estimation may provide support for decision-making by guiding the developer to classify anomalies in terms of their priority and urgency with which they need to be investigated and possibly removed.

This priority depends both on:

— the *safety* category of the system (e.g., in accordance with IEC 1226 [10]), and
— the *severity* of the anomaly in terms of its potential impact on functionality or economics.

In general, it may not possible to determine the severity of the anomaly at this stage. Even in the case of a complete report on an anomalous system behaviour, the information on the failure occurrence observed is usually insufficient to classify the severity of the fault (unless the failure effects are of highest criticality). In fact, the same fault causing a minor failure may result in much more severe consequences for slightly different data trajectories, or in no failure at all. For this reason, the preliminary classification of the severity of an anomaly carried out in this phase should be justified and refined during the ensuing investigation activities.

The potential impact of an apparently minor anomaly from an earlier development phase on the future development phases should not be underestimated. Even an anomaly class, which at coding level might be considered insignificant, could indicate that the development has been crucially compromized during earlier stages, and consequently dramatically affect software reliability. Therefore, it is recommended that the analysis and removal of anomalies in the earlier development phases are carried out as soon as the anomalies are identified. This provides the most economic way of achieving highly reliable code, since it reduces the need for costly and error-prone corrections of earlier documentation.

3. INVESTIGATION OF ANOMALY

The investigation of an anomaly involves determining its:

— *Cause*
The cause is the static feature in one of the formal or informal documents responsible for the identified anomaly (e.g., the observation is that no warning message is given when a limit is exceeded on an operating parameter and its cause could be that the operating parameter is checked against the wrong limit value within the software);

— *Source*
The source is the life-cycle phase when and reason why the cause of the anomaly was introduced (e.g., based on the scenario above, the source of the anomaly could be that the software programmer put the decimal point in the wrong place within the limit value and that the verification exercise failed to identify the error);

— *Consequences*
The consequences are the potential impacts of the anomalous behaviour on the plant (e.g., based on the scenario above, if no warning has been given that an operating parameter is approaching its trip limit the consequence might be that no action is taken to prevent a component being shut down unnecessarily).

The preliminary estimate of the severity of the anomaly and the safety category of the associated FSE are important inputs to the process of determining the priority and extent of the

investigation. An estimated severity level 1 anomaly occurring in a Category A FSE should result in an immediate investigation to determine consequences which will either confirm the severity level (e.g., one such consequence of the anomaly could be to directly prevent a function important to safety from operating) or to reduce it (i.e., no consequences found would lead to any of the criteria listed for a severity level 1 occurring).

Note also that the reliability and performance implications of any consequences arising from the anomaly are also important in determining the priority and extent of the investigation. The cost of downtime or of poor performance may determine that the investigation should start immediately.

Great care needs to be taken during the investigation of an anomaly to avoid being misled by the symptoms and consequences noticed so far. The associated fault could lead, under given different conditions, to other symptoms and consequences of greater severity.

As a general rule, the sooner an anomaly is identified after its introduction, the easier and cheaper will be its investigation and resolution. Hence, the cost of carrying out proper verification during each life-cycle phase is money well spent, since it could save much greater sums being spent later on rectifying faults that could have been detected earlier.

A number of studies have demonstrated that the cost of fixing problems increases as the life-cycle progresses. As an example, A.M. Davis [24] reported the following relative cost of repairs:

TABLE I. RELATIVE COST OF SOFTWARE REPAIRS

LIFE-CYCLE PHASE	RELATIVE COST OF REPAIR
Requirements	0.1–0.2
Design	0.5
Coding	1.0
Unit test	2.0
Acceptance test	5.0
Maintenance	20.0

Other researchers have found different absolute values for the cost of repair but they have all found the same trend.

Determining the source of the anomaly as early as possible is important. The source may reveal a deficiency in the development process, which if not corrected could lead to more serious anomalies being introduced.

For an anomaly that has been identified late in the life-cycle, during plant operation in particular, the investigation could be complex. The identification of the cause and all possible consequences could require examination of several levels of documentation (including code), using a number of support tools. As already explained above, the severity of the anomaly, in terms of safety, reliability and performance, needs to be kept under continual review when determining the amount of extra effort to be put into the investigation.

For the operations phase, it is very important to have in place a software maintenance plan, which defines responsibilities and procedures to be followed for anomaly investigation and resolution. Equally important to the investigation is the availability of personnel with the appropriate training. The software maintenance plan should list what training is required for each aspect of the software maintenance including anomaly detection.

The methodology to be followed for investigation is concerned with the following:

— The inputs required for anomaly investigation both from the anomaly identification activity and other sources;
— The determination of the cause and source of the anomaly;
— The determination of the consequences of the anomaly in terms of safety, reliability and performance; and
— The outputs from the anomaly investigation needed for the anomaly resolution activity and recorded to aid future anomaly investigations and process improvements.

These will be described in the following sections.

3.1. INPUTS TO THE ANOMALY INVESTIGATION

The input to the anomaly investigation is the Anomaly Identification Report (see Section A.1 of the Appendix) containing the information:

— Life-cycle phase at the time the anomaly was reported;
— Technique (e.g., automated tool) used in detecting the anomaly;
— Symptoms of the anomaly;
— Suspected cause of the anomaly (may be unavailable);
— Suspected source of the anomaly (may be unavailable);
— Preliminary estimate of the severity of the anomaly (to help determine the priority and extent of the investigation);
— Category of associated FSE (to help determine the priority and extent of the investigation); and
— Type of software related to the anomaly if relevant (the identification phase may not be able to explicitly identify the software component responsible for the anomaly).

Other inputs need to be collected prior to the investigation and should include:

— Configuration details of the associated FSE, both hardware and software, to be used to determine material to be examined during the investigation and to help reproduce the anomalous behaviour observed;
— Fault tolerance, including diversity, details of the sub-systems (this information will help determine the severity of the anomaly, e.g., reducing it if alternative sub-systems do not display the same anomalous behaviour);
— System tolerances for data inaccuracies related to the anomaly (can be used in the impact assessment to help determine consequences); and
— Any relevant information from previous similar anomaly investigations.

3.2. ANOMALY CAUSE AND SOURCE DETERMINATION

Once the cause is determined and recorded, it should be decided if the source (if not already known) needs also to be determined and recorded. For an anomaly associated with a minor fault, this may not be necessary unless a large number of similar anomalies are being identified and together are likely to cause a problem. The source (e.g., the ambiguity in the design that resulted in a code error) might reveal a deficiency in the development, operating or maintenance process or of non-conformance to the potential procedure. The identification and documentation of the source may prevent more serious and costly anomalies from occurring.

Determination of the source of the anomaly will generally be straightforward once the cause has been detected. Likely sources include:

— Human error (e.g., a programmer has inserted a “>” into the code instead of “<” as clearly stated in the design);
— Documentation deficiency (e.g., the design describes what should happen if any operating parameter is either greater or less than a limit but not what should happen if it equals the limit);
— Tool error (e.g., an automatic code generator incorrectly translates a design instruction into code); and
— Inadequate procedures (e.g., not requiring documentation of input parameters).

The life-cycle phases and processes associated with the source of the anomaly should be recorded to aid the anomaly resolution activity (in determining the corrective action to undertake) and to aid any process improvements undertaken (in determining what phase and processes should be targeted).

The following sections give guidance in determining the cause and source of an anomaly. The sections are based on the life-cycle phases at the time of the anomaly identification. The case involving new software (all software written specifically for the application) is discussed first for each sub-section. The points to consider for other types of software are given at the end of each sub-section. Whenever possible, existing development procedures should be used during the anomaly cause and source investigation.

For all phases, the determination process will be simplified if electronic versions of the code and associated documentation are available. Electronic searches for words and phases are then possible. Other statistical measures may also be useful.

3.2.1. Specification and design phases

If the anomaly has been identified during the specification and design phases, it is likely that only specification and design documents need to be checked to discover the fault. However, other materials (e.g., animation) may also need to be examined.

The symptoms of the anomaly should be examined to help localise the cause. If an automated tool has been used, identification of the possible cause may also have been provided.

The source of the anomaly may be an ambiguity in a document that leads to an incorrect interpretation in a subsequent document. The ambiguity could be caused by a human error or possibly a fault in a tool. The configuration details of the documents can provide important clues (such as authorship, and traceability between documents) in the determination of the source of the anomaly.

Documentation for proprietary software may not be available for investigation, and effective responses by the suppliers of the proprietary software to any queries will depend on what software maintenance agreement exists.

3.2.2. Coding phase

Since the result of the coding phase is the set of system modules ready for system integration, this phase obviously includes all necessary static and dynamic analyses related to the modules, or group of related modules. In the classical model of software development; therefore, this phase is the most complicated one. For high criticality systems, the module tests are often performed by a group, independent of the personnel carrying out the coding itself, making the phase even more complex.

If the anomaly has been identified during the coding phase (e.g., during code testing), generally only a limited amount of code (e.g., the module under test) and associated design documentation needs to be examined to determine the cause. The evidence collected during the anomaly identification phase will in most circumstances pinpoint the areas to check.

The person carrying out the checking should also be familiar with the code being used. If the cause is suspected to be an error in a complicated algorithm, it may not be easy to pinpoint the error. Extra test cases running the dynamic test tool used in the project may need to be generated and/or the examination of the suspect code using a static testing tool may need to be carried out. These additional tests should be incorporated in the overall test suite.

If an anomaly has been detected by a specific verification activity (e.g., dynamic testing) it should be checked if any other activity (e.g., static analysis) has detected a related anomaly. This check can provide further information to help determine the cause and fault. Understanding that these two anomalies are related can reduce unnecessary additional investigative effort.

If the cause is suspected to be within proprietary software, further complications exist. Gathering sufficient evidence to pinpoint the cause could depend very much on the co-operation of the proprietary software supplier. The ability to determine of the source of the anomaly when proprietary software is involved may be limited.

It would be wise to ensure that a maintenance agreement with the supplier exists and has acceptable response times for critical proprietary software. Also the supplier should be obliged to either correct the cause or help provide a suitable workaround as part of the maintenance agreement.

3.2.3. Integration and system test phases

If the anomaly has been identified during the integration phase (e.g., during integration testing) the determination of the associated cause and source could be more complex than for earlier phases. The cause could lie in the interface between two sub-systems or totally within one sub-system. An interface cause will be more likely, assuming the sub-systems have been thoroughly tested before integration.

If the cause is suspected to be within one sub-system, the same guidance outlined in section 3.2.2 applies. If the cause is suspected in the interface between two sub-systems, extra integration test cases may need to be generated to pinpoint the cause. The determination of the source of the anomaly may indicate deficiencies in the definition of the sub-system interfaces.

The types of anomalies that could occur during system testing include CPU overload, delay times, stack overflow, time sequencing and interrupts problems. Extra system tests may pinpoint the cause. However, a combination of factors rather than an individual cause could lead to an anomaly (e.g., timing problems) being observed.

The involvement of proprietary software in this investigation will pose the same problems as those outlined in Section 3.2.2.

3.2.4. Validation, commission and handover phases

Anomalies observed during the validation, commissioning and handover phases are likely to occur during specific test sequences.

Identification of the relevant sub-system containing the suspected fault as early as possible will enable the investigation effort to be properly focused. The methodology described in the sections above can then be followed to detect the cause and source, if it is suspected to be within the software.

3.2.5. Operation, maintenance and modification phase

Anomalies observed during plant operation and maintenance are likely to be discrepancies between actual plant operation and expected behaviour (e.g., expected behaviour given in the

operations manuals or based upon operators' experiences). For such a discrepancy, it is important to determine whether the underlying cause is within the system or external to this system such as within the plant itself (including components such as sensors) or is simply an error in the plant documentation. Expert knowledge of the plant and the computer based system will be necessary and should be accessible.

The configuration of the computer system (e.g., hardware versions, software versions, databases, state of all variables and configurable parameters, etc.) and all relevant plant systems, at the time of the anomaly occurrence, is an important factor during anomaly investigation. This information may be needed to trace the cause of the anomaly and, if necessary, to reproduce the behaviour of the anomaly. The computer system hardware and plant configuration information is essential input in determining if there is a complex dependency between the software, hardware and plant configuration that has not been considered during the software development.

If the cause of the anomaly is suspected to be a software related problem within the computer system, the methods described in the sections above should be followed to determine the cause and its source. These methods should use the same computer system configuration and knowledge of the plant configuration as that in place when the anomaly was observed.

The plant operating history should also be examined to see if the same or similar anomaly has occurred during similar operating conditions. If such incidents have occurred, extra symptoms may then have been noted previously which could provide useful additional input into the current anomaly investigation phase. Also an identification, investigation and resolution exercise may have been carried out previously which could provide guidance for the direction of the current investigation.

Assuming that any modification undertaken during the modification phase follows a similar life-cycle as that described in this report, any anomalies identified during modification should be investigated as described in the sections above. If a different life-cycle is to be followed, the methodology of how to carry out identification, investigation and resolution of anomalies during the various life-cycles phases should be determined before the modification is implemented. The methodology described in this report is likely to form the basis of any other methodologies.

3.3. IMPACT ASSESSMENT AND FINAL CLASSIFICATION

The purpose of an impact assessment is to provide enough information about the possible consequences of an anomaly, in terms of plant safety, reliability and performance, to enable the resolution to determine the best course of action. During the early life-cycle phases, an impact assessment might not be necessary if the anomaly is to be corrected anyway before the start of the next life-cycle phase. An impact assessment should be carried out for any anomaly identified during the operation and maintenance phases.

The impact assessment should determine what severity level (1, 2, 3 or 4) the anomaly actually is. If, for example, a consequence of the anomaly is to prevent a function important to safety from operating, the severity level would be 1, and the resolution would be to either immediately correct the cause of the anomaly or to provide a workaround (in really severe cases this might actually be a shutdown until the cause of the anomaly is corrected). See Section 1.5.2.

During the impact assessment, traceability matrices which link requirements, design and code can be examined, if available, to check for any side effects from the cause of the anomaly.

The determination of the consequences of an anomaly involves a systematic analysis of any potential anomaly propagation throughout the software system in order to identify the worst case impact on safety, reliability and performance. This activity should be based on a thorough hazard

analysis, supported by fault tree techniques. For all significant consequences, it is desirable to consider listing some or all of the following information:

— A general overview of the consequence (e.g., a plant component could be shut down before manual operator intervention could prevent it, this is a consequence of an anomaly of no warning message being displayed when it should have been);
— Operating and configuration conditions required for consequence to occur (e.g., the plant component operating temperature goes above 40 degrees centigrade, version 4 of the computer system software is installed etc.);
— Probability of the consequence occurring. It may only be possible to express the probability as high, medium or low, because it is difficult to calculate it exactly (e.g., a plant component would only be shutdown if its operating temperature exceeded its limit and this is dependent on the probability of the air fans breaking down);
— Safety implications of the anomalous behaviour occurring which is used to determine the severity level of the anomaly (e.g., the reactor core could overheat due to emergency cooling not operating when expected due to a software fault, this would be classified as severity level 1);
— Reliability and performance implications of the anomalous behaviour (e.g., a reactor may shut down unnecessarily due to lack of warning to enable manual preventative measures once a month or plant performance may only be 80% of that expected);
— Cost implication of the anomalous behaviour (e.g., if the reactor is shut down unnecessarily, the equivalent cost of two days of output might be lost before the reactor is brought on line again);
— Likelihood of the same anomaly occurring elsewhere. The likelihood may only be able to be expressed as high, medium or low since it could be very difficult to work it out exactly (e.g., the anomaly could occur in similar software in use in other plants);
— Human involvement affecting the consequence. (e.g., the consequence will not occur if the manual operator routinely notices an increasing temperature despite no warning message being issued);
— Any mitigation of the consequence due to fault tolerance, including diversity, of sub-systems (e.g., a mechanical device might be able to turn a cooling fan on when the software based device displaying the anomaly does not, this will affect the probability of the consequence occurring since the probability of the mechanical device not working when required needs to be considered as well); and
— Effect the system tolerance to inaccurate data has on the consequence, if any.

During the assessment the cost and safety implications of some of the consequences analysed may already provide some preliminary information on the anomaly severity and on the urgency for anomaly resolution actions. Nevertheless, it might be advisable to continue and conclude the impact assessment process by considering all possible anomaly consequences, even in case that the highest severity level is already perceived. Only a complete hazard analysis may fully support diagnosis, resolution and revalidation activities.

3.4. OUTPUTS FROM THE ANOMALY INVESTIGATION

The outputs of the anomaly investigation should be placed into an investigation report (see Section A.2 of the Appendix). The portion of the outputs which are used in the determination of the anomaly resolution, consist of:

— Details of the actual cause;
— Details of the source of the anomaly including associated life-cycle and processes;
— Impact assessment results (see Section 3.3 above);
— Severity level (will be based on the most severe consequences determined during the impact assessment); and
— Confirmation of category of the associated FSE.

The above outputs will also be useful inputs into future anomaly investigations and process improvement activities. For these purposes, further outputs from the investigation phase should include:

— List of persons involved in the investigations;
— Details and results of any investigation tests carried out; and
— Details about any tools used during investigation.

All outputs from the anomaly investigation phase should be put under configuration management to enable the information to be retrieved properly at later stages.

4. RESOLUTION OF ANOMALIES

The term *resolution* includes any activity that results in removing or mitigating the effect of the discovered fault, i.e., to avoid the recurrence of the observed anomaly. The anomaly resolution includes all activities necessary to resolve the immediate anomaly and those activities required to revise processes, policies or other conditions necessary to prevent the occurrence of similar anomalies. Thus it may range from mere administrative rules to avoid specific situations, such as signal trajectories, to mitigating the effect of the fault (e.g., by modifying the hardware configuration), to modifying the software in order to remove the actual cause of the anomaly. In some cases software modifications may be applied to neutralize or mitigate the effect of a hardware defect.

The standards [2] describe in detail how to carry out the resolution. For Category A safety systems, the requirements of these standards should be applied in full to maintain the appropriate *change control or configuration control*. For systems of lower safety requirements some relaxation in the scope of activities may be acceptable.

The standard IEC 880 [3] in Section 9 provides for systems of safety Category A, a proper guideline of the necessary activities after a software fault has been identified; however, these recommendations are restricted to the maintenance (operational) phase of the software life-cycle. The goal of this section is to cover the whole life-cycle of the software and to take into account the systems of lower safety category, as well.

The inputs for this phase include:

— anomaly identification report
— system documentation (which may include the functional requirements, software specification and design documents, maintenance plan, etc.),
— test results, and
— anomaly investigation report.

The outputs of this phase may include:

— anomaly resolution report, containing
  - resolution analysis report,
  - fault removal report;
— modified source code;
— test results with the modified version;
— configuration documents;
— modified documentation including requirements, design documents, user's manuals etc. as applicable;
— modified procedures (either for the development process or for maintenance).

4.1. THE PROCESS OF RESOLUTION

4.1.1. General requirements

For implementing any means to resolve an anomaly (modification, addition, deletion) the following general requirements should be fulfilled:

— the changed object (software, documentation) must comply with the same or higher quality standards as required for the original design and implementation;
— the modifications must be fully documented and completely understandable;
— the modification procedure must comply with the procedures prescribed by the quality requirements of the project;
— V&V, including any required additional tests, must be done;
— the original status of any modified data or program component must be reconstructable by means of appropriate backup copies.

4.1.2. Stages of the resolution process

In general the anomaly resolution process can be subdivided into three stages:

— the resolution analysis,
— decision and schedule,
— fault removal, testing and configuring the modified components.

The first step of the resolution process is the analysis considering all possible options as well as their conditions and consequences. The result of this analysis should be described in an appropriate report (e.g., resolution analysis report) which should provide the necessary technical basis for making the decision concerning an actual action.

The decision may be a simple and adequate action as a result of the analysis, though in some cases the decision can not be reached without making trade-offs between several conflicting aspects. Often the full resolution can only be reached in two or more steps to minimize the consequences of the changes as well as of the fault. That is an intermediate step is taken immediately to mitigate the situation with the final resolution being done later.

The action to remove a fault should be documented in a report (e. g. anomaly resolution report), which typically contains the following information:

— reference to the previous reports concerning the given anomaly (e.g., anomaly identification report, anomaly investigation report, resolution analysis report, etc.);
— the name of the person(s) being responsible for the activity;
— the description of the action to remove the fault;
— the list of all configuration elements affected by the change(s);
— the list of already issued documents requiring modification due to the changes;
— a test plan describing the test procedures in order to take into account the change(s) and to validate the modified version;
— (reference to) reports on the test results.

Note, that the above structure of the report may fully be required only in the maintenance or operational phases of the software, in earlier phases appropriate omissions are acceptable. The whole procedure of modification should be compliant with the general configuration control requirements of the given software system.

4.2. RESOLUTION ANALYSIS — POSSIBLE CORRECTIVE ACTIONS

Once the anomaly has been identified and fully investigated, it is necessary to perform an analysis taking into account all possible means of correcting or mitigating the impact of the anomaly. In many cases, several options are available to solve the problem. The most straightforward choices are usually available in the requirements phase and in the design phases, later more and more trade-offs need to be taken. Naturally, the number of different choices strongly depends on the nature of the fault (e.g., conceptual error, data error, syntax error, non-compliance error, etc.)

A list of different types of corrective actions includes the following:

— requirement modification;
— modification of the software design and/or the technical specification;
— source code modification;
— adding new modules to the system;
— binary patch (a rather obsolete approach);
— modification of manuals;
— modification of the procedures for the development;
— modification of the plant operational procedures;
— data modification;
— extended training, etc.

In many cases, more than one of the above have to be applied. Also more than one modification of a given type may be necessary.

When choosing from the possible options to correct a fault, an appropriate trade-off should be achieved between the following requirements:

— keep all formerly proven safety related features of the system without compromising and retain or improve the qualities of the software concerning its stability and maintainability;
— minimise the non-local consequences of the corrective action, avoid destroying the original structure of the system;
— minimise the cost of the action (in terms of manpower, down-time, etc.);
— minimise the modifications in the user interfaces (i.e. minimise the necessary re-training);
— clarify all re-licensing requirements of the planned action.

The following sections discuss the main factors influencing the choice of corrective actions, i.e., the issues to be taken into account during the resolution analysis.

4.2.1. Type of software

The type of software (or software component in a system) may influence the available choice. Usually the greatest freedom is available in the case of new software, which is specifically developed for a single or a small number of customers. The number of options in the case of proprietary software is usually very limited, the only option may be to find a workaround solution. In the case of existing accessible software, it is necessary to investigate the influence of the discovered error, as well as of the proposed modification, on other applications where the software component is already in use.

For configurable software (i.e., the investigation has proven that the configurable software had been used according to its specification, though it did not react the expected way for some signal trajectories) two options may be available: trying to re-structure the controlling data/code structures in such a way that the software would not be sensitive for the given trajectory, or request a new version from the supplier of the software, in which the given error is fixed. In fact, both approaches may be

risky to an extent, since there is no guarantee that after the modification there would not be some other trajectories, where different anomalies would show up. A good approach may be to get advice from the vendor of the configurable software on how the problem can be worked around.

4.2.2. Software life-cycle phase

The options strongly depend on the software life-cycle phase in which the anomaly has been discovered. During the different software development phases the anomaly removal should be a standard action which is well controlled by the development procedures. In the operational/maintenance phase of the software the procedure of anomaly removal should be performed in an appropriately controlled manner (e.g., by following the Software Maintenance Plan). It is necessary to take into account in this maintenance plan that a revision of the related documents has to be carried out up to the life-cycle phase when the fault had been introduced.

4.2.3. Interface error

In case of an *interface error*, it is potentially possible to modify either side of the interface. When two subsystems of different safety category are being interfaced, then the subsystem of lower safety category should be modified, whenever it is possible. The number of different components using a given side of the interface can be another issue to influence the resolution.

4.2.4. Diverse system

In the case of a diverse safety system, a special analysis should be carried out, in addition to all the above, to maintain or even enhance the diversity of the system with the proposed corrective action. Should the failure be of a common mode type, increasing the diversity needs to be among the goals.

4.2.5. Hardware error

In some cases, when the investigation has revealed that the source of the anomaly was in some hardware component, there still may be one or more options of software modifications which would avoid the recurrence of the anomaly. Such solutions are usually acceptable only when the hardware modification, repair, or replacement is impossible or is not feasible.

4.2.6. Impact of changes

Most of the changes made in a complicated and strongly interrelated system imply impacts on other parts of the system. In some cases rather unexpected impacts may be experienced. The most typical and direct impacts are the necessary modifications in the related documents (including re-issue, re-testing, etc.). In some cases, especially in the case of software systems of lower safety categories as well as of higher complexity, the introduction of a modification in one part of the software system may require modifications in some other parts of the system. Typically, such a case can occur when a functionality shortcoming is discovered after the design phase and the required new functionality contradicts the originally planned or already realized structuring scheme.

The following sections discuss other factors to consider when determining the impact.

4.2.6.1. Software life-cycle phases

In the *requirements phase* the most typical anomalies are contradictions, ambiguity and incompleteness. The main task of reviewing and formally testing the requirement documents is to locate and remove these anomalies, since they are potential sources of problems which could only be

removed at much higher costs later. Thus one goal of removing these errors is to avoid greater impacts later.

In the different *design phases* it is still possible to reveal shortcomings of the requirement documents, though the most typical problems are non-conformance with the requirements and inconsistencies between the different parts of the design documents. It is essential for any design document modification to take into account the possible impacts on the other parts of the design document, otherwise the late discovery of such contradictions could be more costly to correct.

The *coding phase* typically includes the module tests. In most cases the necessary corrective action directly follows from the revealed anomaly. Typically, in this phase the anomalies have little chance of causing any impact on other modules of the system. However, it is possible that the changes in the module will cause interaction type anomalies with other modules in the system. The change could also affect other applications using this module.

The *system integration phase* tends to be more sensitive to the problems in the interactions of different components, thus the careful review of possible dependencies in case of a modification is of great importance. It is a general practice that, during integration and prototype testing, the test is continued without correcting the revealed anomalies, unless the testing team runs into a problem that prevents the continuation. The re-iteration of most of the system-wide tests is usually unavoidable after the deficiencies revealed in the last testing period have been corrected. Typically, the system delivery occurs after the system integration tests have been accepted. Still it may happen that during the site acceptance tests and the final installation anomalies will show up. These cases potentially influence some of the earlier phases and/or some other parts of the system.

Normally, the frequency of system anomalies during the *operational phase* is low. However, the most complicated errors resulting from unforeseen data trajectories may show up during operation. In some cases, especially if the investigation has shown that the error relates back to the specification or design, such anomalies and/or the appropriate corrective actions may have complicated impacts on many different parts of the system and the associated documents. It is a crucial part of software maintenance that all of the possible impacts of any modification of an operational software are carefully taken into account.

4.2.6.2. Type of software

In the case of *existing accessible software* components, the impacts of the discovered anomaly and the required modification may affect some other software systems where the same module is being applied. In this case, the original vendor or developer should be able to judge the necessity of notifying other users of the software component and supplying them with the appropriate corrections.

When the identified anomaly is located in a *proprietary software* component obtained from another vendor, and *full documentation* (including source code) is available, then the corrective action could be carried out either by the original vendor of the software or by the developer of the overall plant system. The selection between these options depends on which party is responsible for the maintenance and configuration of the software component in question. The opposite situation may also occur. The original vendor of the software component may provide the vendor of the overall plant system with information about an anomaly and offer a new release of the software component. In this case the organization, which is responsible for the whole safety system, should make a decision about the inclusion of the new version. For a system already in the operational phase, such a decision usually implies a full re-testing and re-configuring the whole system.

In case of identifying an anomaly in a *proprietary software* component for which *no complete documentation* is available (commercial software, operating system, compilers etc.), then most of the time the only plausible resolution to the problem is to work around it. In safety critical software

systems, the upgrading of the underlying system software components is usually avoided, unless there are very definite motivations to do so. In such cases, the complete repetition of the integration tests is unavoidable.

The main draw back of a commercial *configurable software* is that when a fault is identified within the configurable software, the configurator of the application software can not determine how the fault will be removed from the underlying configurable software, or how it will be tested after the modification. Even a work-around solution can be risky, since unpredictable failures may show up in other functions. Thus, no matter what the solution, the repetition of the integration tests in full should be done for systems of Category A or B, since the limited knowledge concerning the internal structures of the configurable software usually prevents any explicit preliminary judgement about the independence of the functionalities.

4.2.6.3. Safety relevance

Though the strict and rigorous structuring of software systems of high safety impact is a high priority requirement, and a general practice as well, the investigation and exclusion of any side effect of the planned corrective action should be carried out. The higher the safety relevance of the software system is in the scale according to IEC 1226 [10], the more rigorous analysis should be performed. For a Category A software system after delivery, the full integration test procedure should be repeated to prove that the applied corrective action did not cause any unexpected side effects, as it obviously needs to be prescribed by the appropriate procedures.

4.2.6.4. Severity of the given anomaly

The severity of the particular fault is not necessarily correlated directly with the severity of some interrelated anomalies that can be caused as side effects by the proposed modification. It is also important to note that a given fault in the software may have caused a low severity anomaly with one signal trajectory, though another trajectory might trigger a much more severe effect. Thus, in case of a Category A or B software system, the correction of even a very low ranked fault needs to be scrutinized very carefully to exclude any possible side effects which may potentially result in a more significant severity level (see Section 1.5). A system that is well designed structurally is of a great help in this respect.

4.2.7. Estimation of costs

Note, that the investigation of the software anomaly also has cost implications for the vendor, as well as the user. We do not take the vendor's cost implications into account here, since it does not usually influence the decision and schedule of the anomaly resolution.

The costs of resolving a software anomaly can be grouped as follows:

— costs related to the time and manpower requirements to perform further investigation;
— costs related to the corrective actions and the necessary V&V activities;
— costs of re-licensing;
— costs related with the late delivery of the system;
— costs arising at the application site related to the outage time or degraded operation of the plant due to the software error;
— costs arising at the application site related to the implementation of the corrected software;
— costs related to re-training;
— costs arising due to the operational limitations in preventing the activation of the fault (the option of not removing the fault).

The appropriate trade-off required to make the decision is likely to be influenced by some or all of the above cost components. Some unaccountable hidden costs may also be present. These are related to the risks of carrying out the necessary corrective actions and causing unintended side effects.

4.2.7.1. Costs vs. life-cycle phases

It is well known that the sooner an anomaly is discovered in the life-cycle of the software, the more cost effectively its correction can be performed. This rule emphasizes the importance of very careful reviews of the requirement and design documents. Following good coding standards helps to minimize the number of faults that would be revealed by static tests and reviews. Well established system design will reduce the number and severity of the faults that would be revealed during the dynamic integrated tests. Detailed system tests with broad coverage will reduce significantly software anomalies that would be revealed during the operational phase of the system when the price of any corrective action is the highest.

4.2.7.2. Severity of the fault

There is no direct correlation between the severity of the anomaly caused by a software fault and the complexity of the required action to remove the fault. The removal of a severe fault may not be a sufficient action. Extra effort may be necessary to review the software development process to determine how the fault was introduced and why it was not revealed. In order to regain confidence in the product, the verification and validation process should be reviewed and extended as necessary. All of this implies extra costs. It is also clear that a fault with severe consequences as well as a fault in any Category A system requires, for even the simplest modification, extensive verification and validation activities which may be rather costly.

4.2.7.3. Type of software

To take corrective action in proprietary software is usually more expensive, since the depth of knowledge related to such software is usually limited at the system supplier's site. This requires more manpower and time to identify the problem and also to reveal or exclude any possible side effects of the proposed correction. It may be necessary to hire the original vendor of the software to help take the appropriate corrective actions.

4.2.8. Human–machine interface issues

In many cases, especially in the case of control application software, the impact of any change on the human–machine interface (HMI) of the system has to be taken into account. In the case of an already operational system, the corrective action should modify the HMI to the minimum extent, unless the anomaly is directly related to the HMI of the system. The user's manual and procedures have a great influence on the acceptability of the human–machine interface as well as the system itself; therefore, their appropriate maintenance is an integral part of any corrective action.

4.3. DECISION AND SCHEDULE

Whenever there is more than one possible resolution of a software anomaly, all the above given conditions should be taken into account before a decision is made. The decision can be influenced strongly by the actual stage of the software within its life-cycle. For some mild anomalies revealed after the system design has been completed, it may not be cost effective to remove them if they have no safety implications. In this case, they should be documented as a "feature" or limitation of the system.

The main items influencing the decision are the results of the investigation phase, the urgency of the correction of the anomaly, the cost related to the different solutions, and the risk caused by the possible side effects. The general method to make the resolution decision is to optimize the best trade-off between costs associated with doing or not doing a particular action. For anomalies having no direct safety implications, the resolution may be done in two (or more) steps. In such cases, a temporary solution is applied without fully resolving the problem, and in a later phase the full solution will be elaborated and implemented.

4.3.1. Life-cycle phases

In the later phases of the life-cycle, the grouping of corrections of different anomalies is a general and acceptable practice, unless the correction is needed immediately due to safety or other considerations. Such an approach can minimize the cost and effort related to the corrective actions as well as the necessary testing. In the later phases of the life-cycle, a work-around solution can also be accepted, especially when the consequences of the anomaly are not severe and the software system is not of Category A.

4.3.2. Severity

The severity of the revealed anomaly is of great importance to the urgency of the resolution. Obviously, the resolution of high severity faults can not be delayed, especially if it is discovered during the operational phase of the software. For faults having no severe safety implications, temporary administrative modifications may be implemented at the site of application to avoid challenging the fault, until the fault is ultimately removed.

4.4. FAULT REMOVAL AND TESTING

An important aspect in the resolution is assuring the traceability of the actions taken. A change control and configuration management system should be used to assure this traceability. An important aspect of re-testing the system after the modification is that most often the original test set has to be complemented to include additional tests which are sensitive to the newly revealed anomalies. Upon completion of a given modification, the appropriate record (i.e. anomaly resolution record as given in Section A.3 of the Appendix) should be filled.

The new version of the system (corresponding to its actual phase in its life cycle) should be issued only when all the related documents and training, if needed, are made consistent. If the system is already in use, the installation of the new version should follow the procedures and requirements in the system maintenance plan.

5. CONCLUSIONS

This report provides a general model for handling anomalies related software based systems in nuclear power plants. The process of anomaly handling is structured into three main steps:

— anomaly identification,
— anomaly investigation,
— anomaly resolution.

The activities to be done within these steps strongly depend on the *safety category of the software*, the actual *life-cycle phase* of the software, the *type of the software* and the *severity of the anomaly*. This structural approach of the anomaly handling process involves producing documents, which are being passed from one step to the other. It is important to handle these documents in correspondence with the general configuration control rules and procedures of the project.

The main goal of the well structured and documented anomaly handling is to prevent the recurrence of identical and similar anomalies, and to provide input to improving the actual and the subsequent development procedures.

The report gives a guideline for classifying the *severity anomalies* according to their safety and operational relevance. The systematic use of this categorization helps controlling the whole software development and maintenance process.

It is of great importance to *identify* every anomaly in the system as early within the software life cycle, as possible. Therefore the extensive and effective use of computer based tools for analysis and testing during the different phases of software development is strongly recommended.

The *investigation* step needs to reveal the *cause* and *source* and most important, potential *consequences* of the anomaly, and to finalize the anomaly classification.

The decision and schedule concerning the *resolution* measures are strongly dependent on the details revealed in the investigation step. A great care has to be taken that any modification taken to resolve an anomaly should be well documented and must not jeopardize in any way the original quality standards of the product. It is also important to keep the original version in form of appropriate backup copies.

The personnel taking part in the development process should be well trained to be able to follow the development procedures, including the anomaly management.

Appendix

EXAMPLE PROCEDURES

The procedures to be applied for identifying, analysing and correcting software anomalies may be formalized by specifying the data to be recorded in each activity. The records below provide the basic contents of the associated reports. The report puts the information in these records into context and provide a rationale for the decisions taken. These procedures should be matched with the appropriate plant procedures and guidelines. Each anomaly identified is uniquely denoted by a reference number according to an agreed numbering convention. The following three records (Sections A.1–A.3) give examples of the important topics to be covered. The contents of these example records represent a minimum of the topics only and should be augmented as necessary.

A.1. Anomaly Identification Record

A1.1. *Anomaly Title*

Textual sum-up of anomaly as aide-memoir.

A1.2. *Anomaly Reporters*

Names of the persons involved with the identification.

A1.3. *Date of Anomaly Identification*

Calendar time at which the anomaly was identified.

A1.4. *Product Type and Version*

Document type (with version number) affected by anomaly (e.g., specification, design, code).

A1.5. *Identification Phase*

Phase of the software life-cycle when the anomaly was identified.

A1.6. *Identification Method*

Technique adopted allowing to identify the anomaly.

A1.7. *Anomaly Description*

Free-text indication of reason for assuming the presence of an anomaly.

A1.8. *Anomaly Symptoms*

Indication of full information capable of reproducing the anomaly (e.g., input trajectory, internal states).

A1.9. *Priority*

Preliminary assessment of severity level of the anomaly identified.

A1.10. *Policy for Anomaly Investigation*

Decision on whether and when to investigate the anomaly (with indication of the reasons).

A.2. Anomaly Investigation Record

A2.1. Investigation Methods

Techniques adopted allowing to analyse the anomaly identified.

A2.2. Investigation Effort

Person-power required to investigate the anomaly identified.

A2.3. Investigation Phase

Phase of the software life-cycle when investigation was carried out.

A2.4. Personnel

Name of persons involved with investigation

A2.5. Date of Investigation

Calendar time when investigation was carried out.

A2.6. Internal Function

Operation or product containing the anomaly.

A2.7. Investigation Result

Complete description of the anomaly.

A2.8. External Effect

Observable anomaly effects, if any.

A2.9. Priority

Assessment of severity level.

A2.10. Policy for Anomaly Resolution

Decision on whether and when to resolve the anomaly (with indication of the reasons).

A.3. Resolution Action Record

A3.1. Change Number

Numbering of change for this anomaly.

A3.2. Correction Phase

Phase of the software life-cycle when change is carried out.

A3.3. Resolution Personnel

Name of persons involved with anomaly resolution.

A3.4. Items Changed

Parts of the product subjected to changes.

A3.5. Resolution Effort

Person-power required to implement and carry out the resolution.

A3.6. Resolution Description

Free-text explanation of change carried out.

A3.7. Validation Method

Techniques adopted allowing to validate the anomaly resolution.

A3.8.Validation Effort

Person-power required to validate the anomaly resolution.

A3.9. Validation Personnel

Name of persons involved with validation.

A3.10. Validation Justification

Free-text indication of the reasons why the change can be considered to have been validated.

REFERENCES

[1] INTERNATIONAL ATOMIC ENERGY AGENCY, Verification and Validation of Software Related to Nuclear Power Plant Instrumentation and Control, Technical Reports Series No. 384, IAEA, Vienna (1999).

[2] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS, A Standard Classification for Software Anomalies, IEEE-1044, IEEE, Piscatawnay, NJ (1992).

[3] INTERNATIONAL ELECTROTECHNICAL COMMISSION, Software for Computers in the Safety Systems of Nuclear Power Stations, Standard 880, IEC, Geneva (1986).

[4] INTERNATIONAL ATOMIC ENERGY AGENCY, Software Important to Safety in Nuclear Power Plants, Technical Reports Series No. 367, IAEA, Vienna (1994).

[5] INTERNATIONAL ATOMIC ENERGY AGENCY, Computerization of Operation and Maintenance for Nuclear Power Plants, IAEA-TECDOC-808, IAEA, Vienna (1995).

[6] INTERNATIONAL ATOMIC ENERGY AGENCY, Safety Assessment of Computerized Control and Protection Systems, IAEA-TECDOC-780, IAEA, Vienna (1994).

[7] INTERNATIONAL ATOMIC ENERGY AGENCY, Advanced Control Systems to Improve Nuclear Power Plant Reliability and Efficiency, IAEA-TECDOC-952, IAEA, Vienna (1997).

[8] INTERNATIONAL ATOMIC ENERGY AGENCY, Code on the Safety of Nuclear Power Plants: Operation, Safety Series No. 50-C-O (Rev.1), IAEA, Vienna (1988).

[9] INTERNATIONAL ATOMIC ENERGY AGENCY, Code on the Safety of Nuclear Power Plants: Quality Assurance, Safety Series No. 50-C-QA (Rev.1), IAEA, Vienna (1988).

[10] INTERNATIONAL ELECTROTECHNICAL COMMISSION, Nuclear Power Plants — Instrumentation and Control Systems Important to Safety — Classification, Standard 1226, IEC, Geneva (1993).

[11] INTERNATIONAL ATOMIC ENERGY AGENCY, Code on the Safety of Nuclear Power Plants: Design, Safety Series No. 50-C-D (Rev.1), IAEA, Vienna (1988).

[12] INTERNATIONAL ATOMIC ENERGY AGENCY, Reliability of Computerized Safety Systems at Nuclear Power Plants, IAEA-TECDOC-790, IAEA, Vienna (1995).

[13] INTERNATIONAL ATOMIC ENERGY AGENCY, Protection System and Related Features in Nuclear Power Plants: A Safety Guide, Safety Series No. 50-SG-D3, IAEA, Vienna (1980).

[14] INTERNATIONAL ATOMIC ENERGY AGENCY, Safety-Related Instrumentation and Control Systems for Nuclear Power Plants: A Safety Guide, Safety Series No. 50-SG-D8, IAEA, Vienna (1984).

[15] INTERNATIONAL ATOMIC ENERGY AGENCY, Manual on Quality Assurance for Computer Software Related to the Safety of Nuclear Power Plants, Technical Reports Series No. 282, IAEA, Vienna (1988).

[16] BISHOP (Ed.)., "Dependability of critical computer systems", Part 3: Techniques Directory, Guidelines produced by the European Workshop on Industrial Computer Systems (EWICS TC7), Elsevier Applied Science (1990).

[17] ARIANE 5 Flight 501 Failure, Report by the Inquiry Board, European Space Agency, Press Release (1996).

[18] LEVESON, Safeware: System Safety and Computers — A Guide to Preventing Accidents and Losses Caused by Technology, Addison-Wesley (1995).

[19] THOMAS, "The story of the Therac-25 in LOTOS", High Integrity Systems, Vol. 1, Oxford University Press, Oxford (1994).

[20] COOMBES, J., MCDERMID, J., MOFFETT, "Requirements Analysis and Safety: A Case Study (using GRASP)", Proc. SAFECOMP'95 (RABE, G., Ed.). Springer-Verlag (1995).

[21] BERTHOMIEU, M. DIAZ, "Modeling and verification of time dependent systems using time Petri nets", IEEE Transactions on Software Engineering, Vol. 17, IEEE Computer Society (1991).

[22] GARRETT, J., GUARRO, S.B., APOSTOLAKIS, G.E., "The dynamic flowgraph methodology for assessing the dependability of embedded software systems", IEEE Transactions on Systems, Man, and Cybernetics, Vol. 25, IEEE Computer Society (1995).

[23] LYNCH, “Modelling and Verification of Automated Transit Systems, using Timed Automata, Invariants and Simulations”, Lecture Notes in Computer Science, Vol. 1066, Springer-Verlag (1996).
[24] DAVIS, M., Software Requirements, Prentice Hall (1993).

CONTRIBUTORS TO DRAFTING AND REVIEW

Adorjan, F.	Atomic Energy Research Institute, Hungary
Buckle, G.	Rolls Royce and Associates Ltd, United Kingdom
De Jong, M.	AECL Chalk River Laboratories, Canada
Duong, M.	International Atomic Energy Agency
Elder, M.	AECL Chalk River Laboratories, Canada
Naser, J.	EPRI, United States of America
Neboyan, V.	International Atomic Energy Agency
Saglietti, F.	Institute for Safety Technology ISTec GmbH, Germany
Schildt, G.H.	Institute for Automation, Technical University Vienna, Austria
Tate, R.	British Energy, United Kingdom
Wall, N.	Health & Safety Executive, Nuclear Safety Directorate, HM Nuclear Installations Inspectorate, United Kingdom

Consultants Meetings
Vienna Austria: 6–10 October 1997, 18–22 May 1998, 27–30 October 1998