A Systems Engineering Approach to Exception Handling

Herbert Hecht
SoHaR Incorporated
herb@sohar.com

Abstract

Missing or faulty exception handling has caused a number of spectacular system failures and is a major cause of software failures in extensively tested critical systems. Prior work is reviewed and found lacking in a comprehensive approach at the system level as contrasted with details of exception handling at the programming level. As a path to better understanding of the problem, the needs for exception handling are described as they arise at different times of the development cycle and from different disciplines. It is seen that a comprehensive solution is difficult but is essential. The details of stating requirements for exception handling are addressed and a methodology for verifying the effectiveness and completeness is described. Further research needs are discussed and the formation of a working group for a best practice or standard on the subject is suggested.

1. Introduction

Missing or defective exception handling provisions have caused many failures in critical software intensive systems, systems that had been extensively reviewed and tested. The failures occurred under conditions that had not been covered in the reviews and tests, and deficient system requirements are a probable cause for this lack of coverage and the resulting failures. Therefore this paper addresses the generation of system requirements for exception handling.

The systems most in need of precise exception handling requirements are real-time control systems because in these there is usually no opportunity to roll back and try a second time. In keeping with the EWICS TC7 convention [1] such systems are in the following called critical systems. They are found in aerospace, process control, and increasingly in automotive applications. Software for critical systems is expected to protect against a wide range of anomalies that can include

- Unusual environmental conditions
- Erroneous inputs from operators
- Faults in the computer(s), the software and communication lines
- Sensor and actuator failures

The portions of the programs that are charged with providing this protection are called exception handling provisions or exception handlers. Their purpose is (a) to detect that an anomalous condition has been encountered and (b) to provide a recovery path for continued system operation, sometimes with reduced capabilities. In critical systems a large part of the software can be devoted to exception handling and in some cases a substantial part of the failures in these systems have been traced to deficiencies in the exception handlers. Thus, exception handling should be a major concern of system engineering.

The programmer views exception handling as a task that requires detecting an abnormal condition, stopping the normal execution, saving the current program state, and locating the resources required for continuing the execution. Examples of the fairly extensive publications on this aspect of exception handling are [2, 3]. This paper focuses on the systems aspects of exception handling and particularly on the need to establish requirements that assure that exception handling is provided for all the bulleted conditions listed above. Where there are gaps in requirements for exception handling the programmers are left to find their own way, making verification difficult or impossible.

The body of this paper first describes how missing or faulty exception handling causes system failures and summarizes prior research in this field. It then analyzes the sources of exception handling provisions and finds that these are distributed in time and come from a variety of disciplines. Next, the details of generating requirements are explored, and it is found convenient to discuss three phases: objectives of exception handling, algorithmic description, and assignment of the exception handling to a specific software function. After this the verification of exception handling is covered. In the concluding

section we discuss further research needs and suggest the formation of a working group for a best practice or standard on the subject.

2. Exception handling as a cause of failures

During the last 10 years there have been widely publicized accidents due to faulty exception handling of which three will be mentioned here:

- THERAC-25: massive overdoses of radiation as a result of not suppressing operator inputs while magnets were being repositioned resulting in deaths and serious injuries [4].
- Ariane 5, Flight 501: The launch vehicle self-destructed after 37 seconds due to faulty exception handling (and disabling the language's inherent exception handling for efficiency reasons) and failure to deal with simultaneous shut-down of both inertial navigation systems [5].
- Mars Polar Lander: Crashed rather than landed on Mars due to failure to de-bounce a signal generated by mechanical contacts. De-bouncing such signals is a common practice [6].

Such spectacular events are rare, but the following histories show that faulty exception handling is pervasive even in organizations of high technical competence, and has been reported over at least 30 years

One of the earliest areas that reported concern with exceptions were computerized telephone switching centers. A study of failures in AT&Ts Electronic Switching System showed the following distribution of causes [7]:

Recovery	35%
Processing errors	30%
Hardware	20%
Software	15%

It seems reasonable to postulate that all of the failures due to recovery problems and a fair proportion of those attributed to processing and software involved faulty exception handling.

In the same timeframe a review of software dependability in the French telephone switching system [8] was among the first to isolate failures in the exception handling provisions (called *defense* in the reference). It showed that exception handling had a 32% higher fault density than the operational parts of the program (*telephony*) but that relative to its size it caused 2.33 times as many global failures as the operational programs.

A similar picture emerged from a study of failures during final test of the Space Shuttle Avionics (SSA) software prior to the first re-launch after the *Challenger* accident [9]. Exceptions are there referred to as *rare events*, and a *rare report* is a failure report in which the cause is traced to faulty exception handling. Figure 1 is a plot of the percentage of rare reports against a severity (of failure consequences) scale derived from Table 3 of the paper. In the highest severity categories the vast majority of failures were due to deficiencies in the exception handling provisions. In the lowest severity categories most failures were due to other causes but exception handling still accounted for a large fraction. The most likely factor that led to the greater prevalence of failures caused by exception handling was the very careful review and testing of all software, and particularly of those modules that could cause safety critical failures, prior to the final test. This review removed practically all failures in the mainline code, as well as probably many in exception handling. The final test failures showed where the reviews and prior

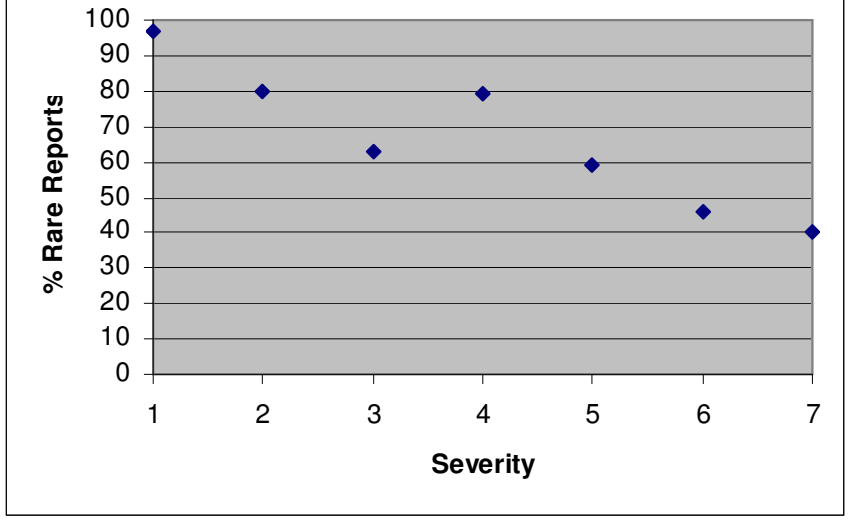

Figure 1 Rare Reports in Final Test of SSA Software

testing had fallen short.

A formal structure for exception handling (primarily addressing exceptions arising in the program) is presented as a chapter in a 1995 volume on software fault tolerance [10]. The author sees the ad hoc design of exception provisions as a major cause of failures. Under the heading of Default Exception Handling he states: "Therefore the identification and handling of the exceptional situations that might occur is often just as (un)reliable as human intuition." The proposals set forth later in this paper aim to arrive at a more disciplined approach.

A review of the extensive DoD software testing conducted as part of the Y2K effort shows that the problem continues [11]: "The main line software code usually does its job. Breakdowns typically occur when the software exception code does not properly handle abnormal input or environmental conditions – or when an interface does not respond in the anticipated or desired manner."

In summary. there is much evidence that deficiencies in exception handling are responsible for a significant fraction of the failures in well-tested systems and very little published material on a major effort to attack this problem. A first step in such an effort should be guidance for the formulation of system oriented requirements for exception handling. Such guidance may exist within companies or project organizations but it is not widely available.

3. Sources of exception handling requirements

In control and information systems needs for exception handling typically arise from the following sources:

- Operational requirements of the system
- Implementation details
- Computing environment
- Monitoring and self-test of system functions
- Application Software

Details of each of these will be discussed below. However, it is fairly obvious from this enumeration (a) that most of these requirements will arise only during the design and cannot be expected to be available at the beginning of the project, (b) that the requirements will originate from multiple organizational units and disciplines, and (c) that there will inevitably be differences in format and the level of detail.

3.1 Operational requirements of the system

The most frequently encountered operational requirements for exception handling arise from the need for power, communication and thermal control. How should the system respond when these fail and how much time is available for recovery? Where the system can operate in several modes, safeguards for the issuance of mode changes and verification of the execution of mode change commands may be required, and exception handling when the conditions are not met.

These operational requirements for exception handling are usually known early in the life cycle and are less likely to change than other needs.

3.2 Implementation details

As the system is being implemented additional operational and structural details will emerge that can give rise to exception conditions. Sensor calibration and monitoring of internal temperatures, of actuator states or of output channels can give rise to exception handling needs. Where operator input is required, the authorization of the operator needs to be verified and the action to be taken on failure of verification needs to be specified. Another important area under this heading are maintenance provisions. Is a distinct maintenance mode required? Can spares be "hot-swapped"? Do maintenance personnel require authentication? All of these questions may identify conditions that do not permit resumption of operations, and that require exception handling. These requirements usually arise during the preliminary design phase of a project.

3.3 Computing environment

Some requirements for exception handling arise from the computer hardware, such as handling of memory errors, divide-by-zero exceptions, and overflows or underflows. These requirements are usually known as soon as the hardware is specified and can be expected to remain stable unless the hardware is replaced.

A substantial part of exception handling arises from the software component of the computing environment, including the executive or operating system and middleware. For real-time systems that operate on fixed cycle times these software components signal when time limits are about to be violated and they may also invoke their own exception handling to deal with these conditions. Because higher level applications are usually more informed about the consequences of a missed cycle and about alternative

means of accomplishing a function it may in some instances be desirable to disable the low level exception handling lest it interfere with the more beneficial actions programmed in the applications.

3.4 Monitoring and self-test of system functions

The original system requirements may include monitoring or self-test provisions but the details of such functions are added during the course of the development. Monitors are usually active at all times and exception handling is required only when they indicate anomalous conditions. Self-test may be invoked for any system function but it is particularly important where passive sensors monitor critical system states, such as an over-temperature sensor for a temperature sensitive electronic component. Self-test may be initiated autonomously, as a result of an indication by a monitor or due to an exception condition encountered by the application. In either case the requirements for exception handling must provide for appropriate action under all test outcomes.

3.5 Application software

The exception conditions under this heading may be native (arising directly from an anomalous state or transition of the application software) or external (responding to an anomaly in the system). An example of a native software exception is to protect against exceeding array bounds. But the software implementation of any of the conditions mentioned in 3.1 – 3.4 is an external exception. A source for the detection and mitigation of native software failures can be found in the taxonomy compiled by senior figures in the field of dependable computing [12]. The reference lists 12 types of software faults that can impede the intended execution of a program (Figure 5 of the reference), not all of which may be applicable in a given environment but that need to be at least considered. Included in the list are maliciously introduced faults that can be expected to affect the system in a particularly critical operational state.

3.6 Summary of sources for requirements

In this section we have identified typical sources of requirements for exception handling. The discussion was intended to show the distribution in time over the development cycle, the diversity of disciplines from which they originate, such as system planners, system and component engineers, and software professionals and these factors can be assumed to be near universal. Also, as was pointed out in Section 3.3, requirements for exception handling can be in conflict, such as when a recovery from a failure causes the allowed time for a function to be exceeded, activating a watchdog timer exception that may negate the recovery. For all of these reasons, a unified approach to requirements for exception handling is difficult but it is also essential. A more detailed discussion of how requirements are formulated will be found in the following section

4. The evolution of requirements for exception handling.

Requirements for exception handling will normally arise over most of the system development cycle, and the current heading addresses the form in which the requirements can be expected to be formulated. First we discuss the typical steps in the evolution of the exception handling requirements, and then how these fit into the development cycle. Evolutionary steps for a given requirement usually include

Objectives – these represent the conditions to be prevented or to be achieved; they may reference regulatory or system level requirements. The operating conditions to which the objectives apply also need to be stated. The document generated at the end of this step permits initiation of the algorithmic descriptions and is a preliminary input to software requirements.

Algorithmic descriptions – these identify how the anomalous condition is to be detected and the action(s) to be taken subsequent to detection. The document generated at the end of this step permits initiation of the assignment for exception handling and completes the input to software requirements

Assignment to a software function – this identifies the software (and sometimes also hardware) function responsible for the execution of the algorithm. Requirements for sampling frequency, execution time for exception handling, and other implementation constraints (e. g., use of a specific sensor output) are usually included included in this step. The document generated at the end of this step is an input to software design.

Figure 2 shows a typical distribution of requirements for exception handling over the system development phases. The phase designations are shown along the horizontal axis. Each of the needs categories discussed in Section 3 is represented by a horizontal bar. The colored divisions in each bar represent the three steps of requirements development described above. Most

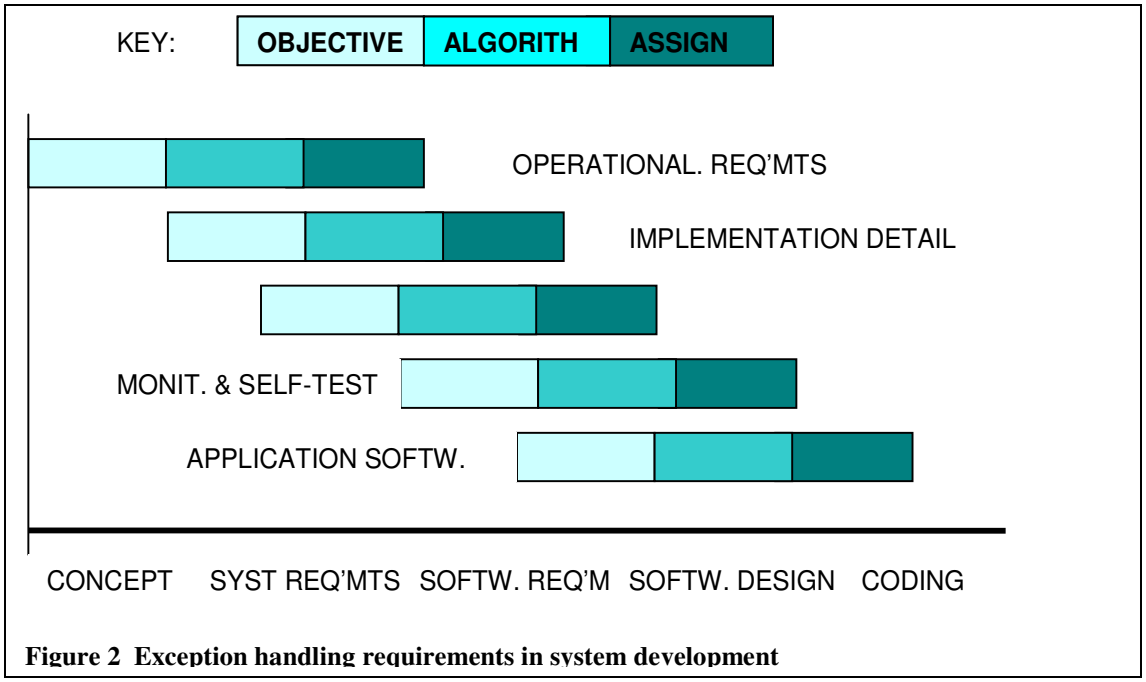

Figure 2 Exception handling requirements in system development

requirements inputs will have been completed prior to the start of the coding phase with only those originating from native application software support showing an overlap. This scheduling should allow for orderly implementation of the exception handling requirements. As mentioned in the introduction of this paper, guidelines exist for the coding of exception handling in most currently used programming languages.

During testing and in the operation and maintenance phase the need for modifying the exception handling requirements may arise, and sometimes new requirements may be added. These events are not shown in the figure. Good configuration management demands that the changes not only be made in the program but that the requirements documentation is updated as well. Verification that is usually required for all critical software will be much easier if requirements documents and the code represent the same revision stage.

5. Further steps

Observations from several sources have shown that exception handling accounts for a disproportionate number of failures in critical programs. The lack of requirements for exception handling in the public domain is a highly likely cause of this condition. In this paper we have developed one possible approach to generating requirements for exception handling and it is hoped that this can serve as a starting point for a volunteer effort to create a consensus document.

Establishing requirements for exception handling in a consensus format will benefit both software development and verification activities. The developer has a systematic and schedulable approach to exception handling. The verifier's task will also become more definable in that it will consist of checking that

1. Detection of anomalous conditions is provided for all hazards identified in the system documentation and in review of previous software failures.
2. The software invokes mitigation provisions at a level of the hierarchy that is specific to the sensed exception and prevents dissemination of faulty data and states
3. Testing shows that exception handling works as expected and without interfering with other system functions that are critical for mission success.

This structure will permit a much more focused approach to at least one major aspect of V&V than can be achieved with current practice.

References

[1] F. J. Redmill, ed., *Dependability of Critical Computer Systems,* Elsevier Applied Science, 1988

[2] Doshi, Gunjan *Best Practices for Exception Handling*, http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html
[3] Christophe de Dinechin, *C++ Exception Handling for IA-64* https://db.usenix.org/events/wiess2000/full_papers/dinechin/dinechin.pdf
[4] Leveson, Nancy G. and Clark S. Turner, "An Investigation of the Therac-25 Accidents", *IEEE Computer,* vol 26, no. 7, July 1993
[5] Wikipedia: Ariane 5/Launch History
[6] NASA Press Release 00-46, March 26, 2000
[7] Toy, W. N., "Fault-Tolerant Design of AT&T Telephone Switching Systems" in *Reliable Computer Systems: design and evaluation,* Siewiorek and Swarz, eds., Digital Press, Burlington MA, 1992.
[8] Kanoun, K. and T. Sabourin, "Software Dependability of a Telephone Switching System", *Digest of Papers, FTCS-17,* Pittsburgh PA, July 1987, pp. 236 – 241.
[9] Hecht, H. and P. Crane, "Rare Conditions and their Effect on Software Failures", *Proc. of the 1994 Annual Reliability and Maintainability Symposium",* January 1994, pp. 334 – 337.
[10] Cristitan, Flaviu "Exception Handling and Tolerance of Software Faults" in *Software Fault Tolerance,* Michael R. Lyu, ed., Wiley, New York, 1995
[11] C. K. Hansen, *The Status of Reliability Engineering Technology 2001*, Newsletter of the IEEE Reliability Society, January 2001
[12] Avizienis, A., J. C. Laprie, B. Randell and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing", *IEEE Transactions on Dependable and Secure Computing,* vol. 1, No.1, Jan 2004