System Engineering and Software Exception Handling

Herbert Hecht, Sr. Member, IEEE

Abstract—**Missing or faulty exception handling has caused a number of spectacular system failures and continues to be a major cause of software failures in aerospace systems. Prior work is reviewed and found lacking in a comprehensive approach at the system level as contrasted with programming level exception handling. As a path to better understanding of the problem, the needs for a system engineering approach to exception handling are described as they arise at different times of the development cycle and from different disciplines. It is seen that finding comprehensive solution is difficult but it is essential to pursue this problem. 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**

Index Terms—**Exception Handling, Fault Tolerance, Safety Critical Systems, Software Reliability, Software Requirements and Specifications**

TABLE OF CONTENTS

1. INTRODUCTION

Missing or defective exception handling provisions have caused many failures in critical software intensive systems even though they had undergone extensive review and test. The failures occurred under conditions that had not been covered in the reviews and tests because of incomplete or imprecise system requirements. To curb this cause of failures the 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 that permits 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 is an important part of software development and of the verification and validation activities.

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. An example of the issues dealt with at that level is the following program construct and the comment that follows it:

```
public void someMethod() throws Exception{
}
```

“This method is a blank one; it does not have any code in it. How can a blank method throw exceptions? Java does not stop you from doing this” [2].

Among the fairly extensive publications on this aspect of exception handling is also [3].

In contrast, 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. If there are no explicit requirements for coverage of exception handling it is left up to the program developers to find their own way. The absence of requirements also makes it difficult to verify that all exception handling performs as intended. It is the purpose of this paper to highlight the need for research and guidance on the systems aspects and to present a systematic approach to requirements formulation that may facilitate further discussion.

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 just three will be mentioned here:

- THERAC-25: massive overdoses of radiation were administered to patients 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 (specifically disabling language provided 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].

While these events were spectacular they may be regarded as extremely rare, perhaps like high magnitude earthquakes in California. The following histories show that faulty exception handling is pervasive, has been reported over at least 30 years, and has affected critical systems designed and operated by organizations of high technical competence.

One of the earliest areas that reported concern with exception 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 errors 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 specifically isolate failures in the exception handling provisions (called *defense* in the reference). It showed that exception handling caused 30% of all failures and an amazing 54% of the most severe failures, referred to as *global* failures in the reference.

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 fraction of exception

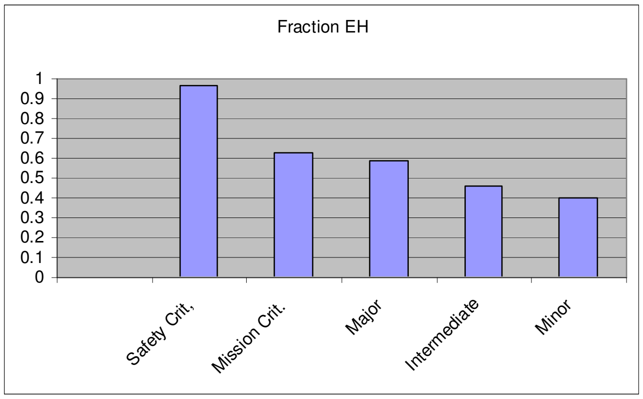

Figure 1 Rare Reports in Final Test of SSA Software

handling (EH) reports for various severity of failure categories derived from Table 3 of the reference.

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 testing had fallen short.

A formal structure for exception handling (primarily addressing exceptions arising in the software) 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."

On a positive note, it has been shown that emphasis on exception handling during the design and implementation of a software program can significantly reduce the probability of failures when exception conditions are encountered. These findings are based on an experiment with students in which a control group received no special instruction while two other groups were either put in teams of three (collaboration) or were individually urged to use a structured aid for generating requirements (dependability case). Both of these groups achieved approximately a doubling of the fault coverage (exception conditions that were correctly identified and handled) [12]. The program involved in this experiment was very simple (computing the mean and standard deviation of an input set), and only well-known software exceptions were dealt with. Thus the findings hardly qualify as a broad spectrum solution to the exception handling problem but it is encouraging that this experiment was carried out at all.

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. As implied in the discussion of the Space Shuttle data, the NASA Office for Safety and Mission Assurance (OSMA) has for many years devoted extensive efforts to improving and demonstrating software reliability and as part of this has produced important guidance documents that are in the public domain.

NPR 7150.2	Software Engineering Requirements
NPR 7120.5C	Program and Project Management Processes
NPR 7120.6	Lesson Learned Requirements
NPD 8700.1C	Policy for Safety and Mission Success
NPD8700.3A	SMA for Spacecraft, Instruments and Launch Services
NPD8705.2A	Human rating Requirements for Space Systems
NPR 8705.6	SMA Audits, Reviews and Assessments

None of these contain explicit material for organizing or evaluating requirements for exception handling. Issuing an edict "There shall be requirements for exception handling" is not likely to produce desirable results. The requirements have to be specific with regard to the need for exception handling and they have to be capable of tailoring for each system and life cycle phase. Suggestions for these topics will be addressed in the following two sections of this paper.

3. SOURCES OF EXCEPTION HANDLING REQUIREMENTS

In control and information systems, and particularly in aerospace 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 are discussed below. But just by looking at this list it is fairly obvious that (a) the totality of the requirements cannot be expected to be available at the start of the project, (b) the requirements will originate from multiple organizational units and disciplines, and (c) there will inevitably be differences in format and the level of detail in which the requirements are stated. All of this is in addition to the acknowledged difficulties of requirements formulation for critical systems [13].

3.1 Operational Requirements of the System

The most frequently encountered operational requirements for exception handling arise from the need for power, communications and thermal control. How should the system respond when these fail and how much time is available for recovery? Also, where the system can operate in several modes, safeguards for the issuance of mode changes and verification of the accomplishment of mode changes give rise to exception handling requirements.

Aerospace systems that are "essential for continued safe flight and landing [of aircraft]" [14] are frequently made redundant in their entirety or on a channel basis. Exception handling is required to isolate the failed component or channel and to reconfigure the remaining units into a survivable structure. In process control applications similar redundancy management may be applied to sensors, and shut-down may have to be initiated where no redundant coverage is available. Exception handling may also be required to avoid undesirable states (e. g., excess fuel consumption) by initiating corrective measures or by activating an alarm.

These operational requirements for exception handling are frequently known early in the life cycle and are less likely to change than other needs. Also, these provisions may exist in predecessor systems and thus not only the requirements but also parts of the implementation can be specified early.

3.2 Implementation Details

As the system enters design additional operational and structural details will emerge that give rise to exception handling requirements. In sensor calibration and in monitoring of internal temperatures, of actuator states or of output channels exception conditions may be encountered for which the handling needs to be specified. Where operator input is required the authorization of the person generating the input may have to be verified. Safeguards against potentially harmful operator commands will have to be provided. Also, as more detail about interfaces with cooperating systems emerges checks on acceptance or delivery of information will need to be implemented. All of these features can result in requirements for exception handling.

Maintenance provisions are another important area under this heading.. Is a distinct maintenance mode required? Can spares be hot-swapped? Do maintenance personnel require authentication? How will the system be restored to operational use after maintenance, and how will the completion of this transition be verified? The response to every one of the questions can indicate a condition that requires exception handling, including initiation of remedial action to restore the system to operational use.

As the heading implies, these requirements for exception handling are frequently not foreseen at the project initiation. However, once recognized they can be expected to remain reasonably stable during the rest of the development phase.

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 and underflows. These requirements are usually known shortly after the hardware is specified and can be expected to remain stable unless the computer is replaced.

A substantial part of the exception handling originates in the software component of the computing environment, including the executive or operating system and the middleware. For real-time systems that operate on fixed cycle times these software components detect when timing constraints are about to be violated and they may invoke their own exception handling to deal with these conditions, not always with desirable results. Higher level application programs are usually better informed about the consequences of a missed cycle and about alternative means for accomplishing a function. In some instances it is therefore necessary to modify or disable the native time-out provisions. Commercial operating systems include watchdog timers that serve a similar purpose. In establishing and reviewing requirements for exception handling these possibilities of undesirable interactions must be kept in mind.

Requirements arising out of the software components of the computing environment may become known later than those from the hardware part, and they are more likely to change during subsequent development phases.

3.4 Monitoring and Self-Test of System Functions

Although the original system concept may include monitoring and self-test functions, the details of these provisions emerge only after selection of the components to be monitored. Even for functions that were fully identified during the concept phase, the full scope of the exception handling requirements will become known only much later.

Monitors are usually active at all times and exception handling is required only when they indicate anomalous conditions. Exception handling must distinguish between failures in the monitor and failures in the monitored component. Self-test may be invoked for any system component but it is particularly important where passive sensors guard against potentially harmful system states, such as an over-temperature sensor for a critical electronic component. Exception handling must provide for appropriate action under all test outcomes (which may include some unforeseen combination or sequence of test results).

3.5 Application Software

The exception conditions under this heading may be native (arising from a programming error that induces an anomalous state or transition) or external (responding incorrectly to an anomaly in the system). 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 [15]. Figure 5 of the reference lists 12 types of software faults that can impede the intended execution of a program, including maliciously introduced faults that affect the execution during particularly critical operational states. The same reference also includes a taxonomy of recovery provisions that is significant for exception handling. In it a distinction is made between *error handling* (replacing erroneous data values) and *fault handling* (isolation, removal and replacement of permanently damaged data stores or instructions).

Because these exception conditions are closely tied to the software development their requirements can only be formulated in general terms prior to software design and must be finalized during the design phase. These requirements are also the ones most likely to change as a result of software test and operation.

3.6 Summary of Sources for Requirements

In this section we have identified typical sources of requirements for exception handling. It is not claimed that these represent an exhaustive listing for aerospace systems, and they are certainly not exhaustive for other critical systems. Rather 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, usually in algorithmic format. 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 in this step. The document generated at the end of this step is an input to software design.

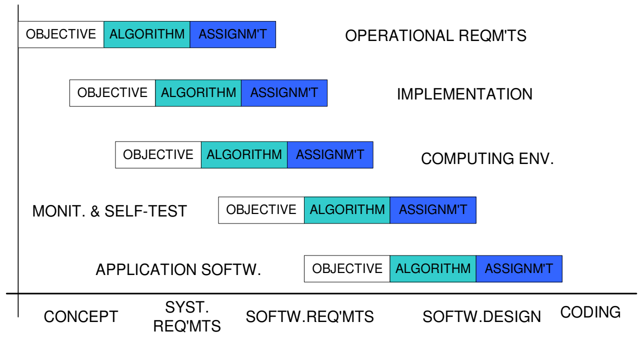

Figure 2 Phasing of Exception Handling Requirements

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 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. VERIFICATION OF EXCEPTION HANDLING REQUIREMENTS

The introduction listed the following conditions that may give rise to the need for exception handling

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

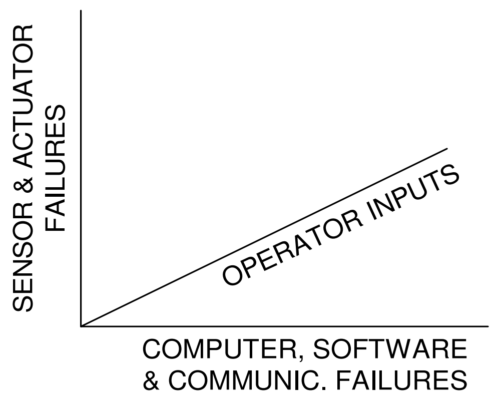
SENSOR & ACTUATOR FAILURES
OPERATOR INPUTS
COMPUTER, SOFTWARE & COMMUNIC. FAILURES

Figure 3 Verification Matrix

For the purpose of verification it is convenient to enumerate the “unusual environmental conditions” as a high level independent discrete variable, allocating a separate verification session to each specified event. The remaining conditions can then be arranged in a three-dimensional space as shown in Figure 3. The listing along each of the axes can be in the order of decreasing severity of effects or decreasing risk assessment [16]. It is desirable to locate the most severe or highest risk events close to the origin so that these can become the focus of verification and test activities. Testing under multiple exception conditions is a very effective way of finding software weaknesses as shown by an analysis of failures of a redundancy management program conducted under NASA sponsorship [17].

Twenty versions of a redundancy management program, written in Pascal, were developed at four universities (five versions at each) from the same requirements, and the versions were then tested individually to establish the probability of correlated errors. The specifications for the program were very carefully prepared and then independently validated to avoid introduction of common causes of failure. Each programming team submitted their program only after they had tested it and were satisfied that it was correct. Then all 20 versions were subjected to an intensive third party test program. The objective of the individual programs was to furnish an orthogonal acceleration vector from the output of a non-orthogonal array of six accelerometers after up to three arbitrary accelerometers had failed. Table 1 shows the results of the third-party test runs in which an accelerometer failure was simulated. The software failure statistics presented below were computed from Table 1 of the reference.

Table 1. Tests of Redundancy Management Software

No. of anomalies	Observed Failures	Total Tests	Failure Fraction
1	1,268	134,135	0.01
2	12,921	101,151	0.13
3	83,022	143,509	0.58

In slightly over 99% of all tests a single anomaly could be handled as indicated by the first row of the table. Two anomalies produced an increase in the failure fraction by more than a factor of ten, and the majority of test cases involving three anomalies resulted in failure. Thus, a significant conclusion from this work is that test cases containing multiple exception conditions greatly increase the probability of finding latent faults, including those not due to the multiplicity of conditions. The verification of exception handling can benefit from this finding by using a test program that emphasizes test cases with multiple exceptions.

6. 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 an effort to create a consensus document.

Establishing requirements for exception handling in a consensus format will benefit both software development and verification activities. For the developer it provides 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] Cristian, 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] Maxion, Roy A. and Robert T. Olszewski, "Eliminating Exception Haandling Errors with Dependability Cases: A Comparative, Empirical Study, *IEEE Transactions on Software Engineering,* vol. 26 no. 9, September 2000.
[13] Leveson, N. G., "Software Safety: what, why and how?" *Computing Surveys of the ACM,* vol 18(2), pp. 125 – 63, 1986
[14] Federal Aviation Regulation (FAR) 25.1309
[15] 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

[16] Department of Defense, *Standard Practice for System Safety*, MIL-STD-882D (Table A-III), February 2000
[17] Eckhardt, D. E., A. K. Caglayan, J. C. Knight, et al., "An experimental evaluation of software redundancy as a strategy for improving reliability", *IEEE Trans. Software Engineering,* vol 17 no 7, July 1991, pp. 692 – 702

BIOGRAPHY

Herb Hecht *is Chief Engineer of SoHaR Incorporated. In prior employment he worked at The Aerospace Corporation and at Sperry Gyroscope Company (now Honeywell Flight Systems). His primary interest is the reliability of critical systems, such as aircraft controls, space systems, and safety systems of nuclear power plants and process industries.*

He is the author of System Reliability and Failure Prevention, *Artech House, 2003*, and numerous publications in technical journals and conference proceedings.

He received a BSEE degree from CCNY, a Masters from Polytecnhic University of New York, and a Ph. D in Engineering from UCLA. He is a member of the Golden Core of the IEEE Computer Society.

Herbhecht@sbcglobal.net