Establishing Requirements for Exception Handling

Herbert Hecht
SoHaR Incorporated

1. Introduction

Software for embedded systems is expected to protect the system from a wide range of conditions 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 frequently 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 to continued system operation, sometimes with reduced capabilities. In safety 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 verification.

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 [1] [2]. In this paper we address the systems concerns with exception handling and particularly the need to establish requirements that assure that exception handling is provided for all the bulleted conditions listed above. The lack of explicit requirements for coverage of exception handling leaves it up to each program developer to find his own way and makes it difficult to verify that all required exception handling performs as intended. It is the purpose of this paper to highlight the need for guidance on the systems aspects, particularly coverage, of exception handling and to possibly promote the formation of a working group for a best practice or standard on the subject.

2. Exception handling as a cause for failures

A study of software dependability in the French telephone switching system[3] was among the first to isolate failures in the exception handling provisions (called *defense* in the reference). It showed that exception handling had a higher fault density than the program as a whole (it accounted for 30% of the faults but constituted only 26% of the program volume) and that the consequences of its failures were much more severe (20% resulted in global unavailability vs. 13% for the program as a whole).

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[4]. 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 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. Those found in final test showed where the review had not been perfect.

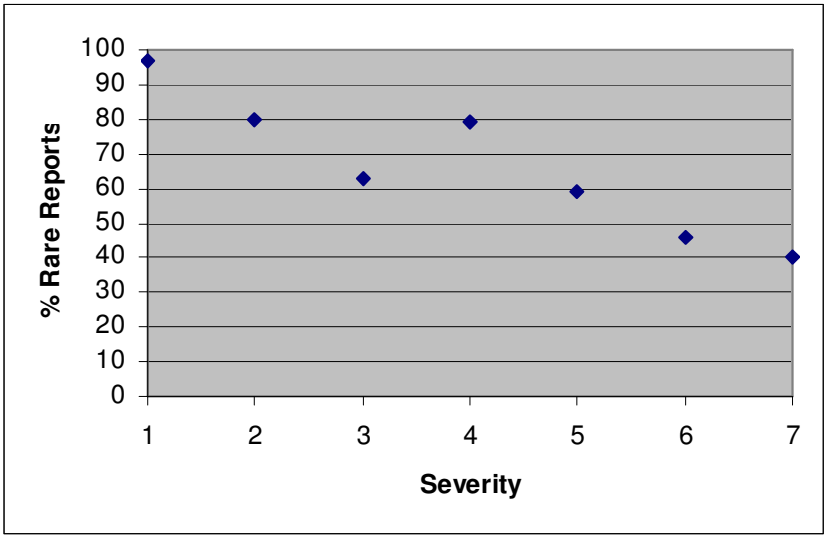

Figure 1 Rare Reports in Final Test of SSA Software

A review of the extensive software testing conducted as part of the Y2K effort showed similar results[5]: "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."

Thus there is considerable evidence that deficiencies in exception handling are responsible for a significant fraction of the failures is 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 the formulation of system oriented requirements for exception handling. Such requirements may exist within companies or project organizations but they are not widely available. 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 need for exception handling and they have to be tailored for each system and software life cycle phase. These topics will be addressed in the following two sections of this paper.

3. Needs for Exception Handling

Needs for exception handling typically arise from one of the following sources:

- External to the system being developed
- Protective measures arising from system function and structure
- Protective measures arising from the computing environment
- Support for monitoring and self-test
- Support for the primary software function

Details of each of these will now be discussed.

3.2 External needs

The most frequently encountered external needs for exception handling arise from start-up and shut-down provisions. Calibration, mode change commands and support for exception handling at a higher level may also be encountered. External needs for exception handling are usually known early in the life cycle and are less likely to change than other needs.

3.3 System protective measures

This heading covers the most diverse and demanding needs for exception handling. It includes protection against unsafe or undesirable states, redundancy management and activation of alarms and physical barriers. Aerospace systems that are "essential for continued safe flight and landing [of aircraft][6]" are frequently made redundant in their entirety or on a channel basis. Exception handling is required for isolation of the failed component or channel and reconfiguration of 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 for failures where no redundant

coverage is available. Exception handling may be required to avoid undesirable states (e. g., excess fuel consumption) by initiating corrective measures or by activating an alarm.

Exception handling under this heading may also be required to protect from unsafe or undesirable operator actions or to prevent unauthorized personnel from operating the system or equipment. Similarly, protection against inappropriate maintenance actions may use exception handling.

Requirements for exception handling from system protective measures are usually known at least in outline form early in the development cycle and will evolve and possibly change through much later stages.

3.4 Computing environment protective measures

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 the higher level applications are usually more informed about the consequences of a missed cycle and about alternative means of accomplishing a function it may be desirable to disable the low level exception handling lest it interfere with the more beneficial actions programmed in the applications. Commercial operating systems include watchdog timers and resource monitors that serve a similar purpose. Again, requirements for exception handling must be aware of all these native protective measures lest they interfere with the specifically programmed ones.

Requirements for software environment protective measures will usually follow those arising from the hardware environment and are likely to change throughout the application software development.

3.5 Support for monitoring and self-test

The original system design may include monitoring or self-test provisions to facilitate the operator's task or to aid in maintenance. Frequently such functions are added or enhanced 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.

3.6 Support for the primary software function

The exception conditions under this heading arise directly from an anomalous state or transition of the application software but indirectly they may respond to an anomaly in the system. A source for the detection and mitigation of software failures can be found in the taxonomy compiled by senior figures in the field of dependable computing[7]. 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. The reference also provides a taxonomy of recovery provisions in which a distinction is made between *error handling* (replacing erroneous data values) and *fault handling* (the isolation, removal and replacement of permanently damaged data stores or instructions).

Because these exception conditions are closely tied to the software development the requirements they can only be formulated in very general terms in the development stages prior to software design. They are also the most likely to change as a result of software test and operation.

3. Exception handling in the system life cycle.

Requirements for exception handling will normally evolve during the system life cycle, and the current heading addresses the form in which the requirements can be expected at various life cycle stages. First we discuss the typical steps in the evolution of the exception handling requirements, and then how these fit into the system life cycle. Evolutionary steps usually include

> *Objectives* – 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 and completes the input to software requirements
>
> *Assignment to a software function* – defines the software 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) may be included in this step. The document generated at the end of this step is an input to software design.

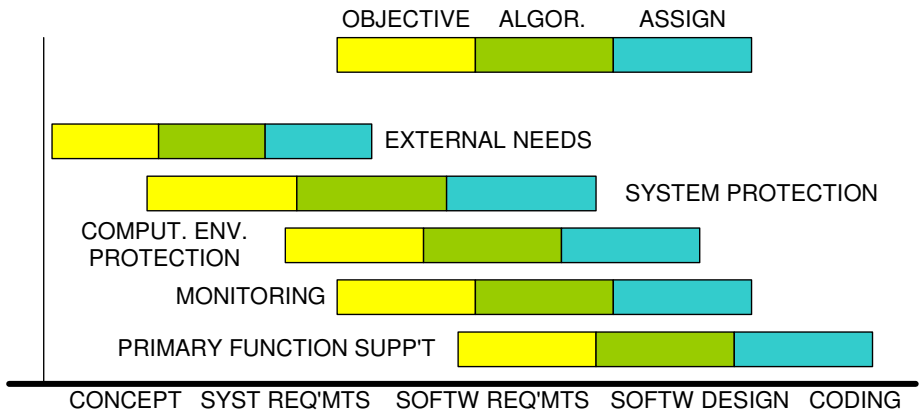

Figure 2 Exception handling requirements in software development

Figure 2 shows a typical interface between the requirements for exception handling and 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 primary function 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.

4. 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

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 frequency of occurrence, decreasing severity of effects, or decreasing risk assessment[8]. It is desirable to locate the most frequent or most severe 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.[9]

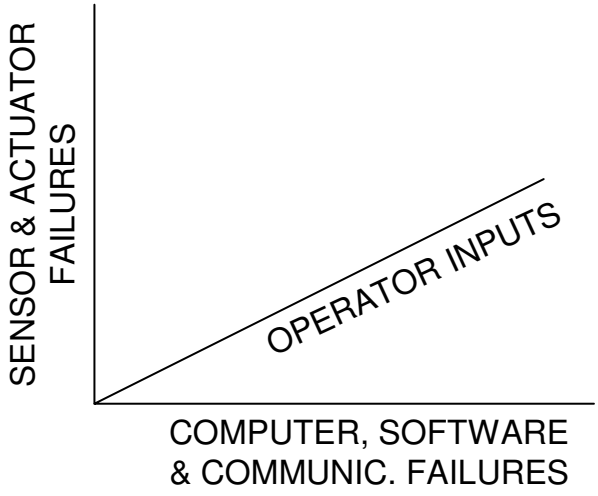

Figure 3 Verification Matrix

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.

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 scheduled approach to exception handling and the

The verifier will need to check 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 handling works as expected and without interfering with other system functions that are critical for mission success.

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

Acknowledgement

Portions of this research have been conducted under contract W911QX-06-C-0116 with the U. S. Army Research Laboratory. The encouragement for this work received from Mr. Jeffrey M. Dehart is much appreciated.

References

[1] Doshi, Gunjan *Best Practices for Exception Handling*, http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html
[2] Christophe de Dinechin, *C++ Exception Handling for IA-64* https://db.usenix.org/events/wiess2000/full_papers/dinechin/dinechin.pdf
[3] Kanoun, K. and T. Sabourin, "Software Dependability of a Telephone Switching System", *Digest of Papers, FTCS-17,* Pittsburgh PA, July 1987, pp. 236 – 241.
[4] 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.
[5] C. K. Hansen, *The Status of Reliability Engineering Technology 2001*, Newsletter of the IEEE Reliability Society, January 2001
[6] Federal Aviation Regulation (FAR) 25.1309
[7] Avizienis, A., J. C. Lapried, 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
[8] Department of Defense, *Standard Practice for System Safety*, MIL-STD-882D (Table A-III), February 2000
[9] 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