ABSTRACT

Exception handling allows (1) a program to be structured such that the original design is preserved in presence of possibly failing components; (2) rare or undesired cases to be treated in an unobtrusive manner; and (3) imperfections to be handled systematically. This chapter develops a theory of exception handling with try-catch statements, and demonstrates its use in the design of dependable systems by giving a formal account of the patterns of masking, propagating, flagging, rollback, degraded service, recovery block, repeated attempts, and conditional retry. The theory is based on weakest exceptional preconditions, which are used for both defining statements and proofs. Proof outlines are introduced and used to establish the correctness of the patterns.

INTRODUCTION

A program may fail to perform its intended task for three reasons:

- The specification may be in error. It may not capture the user’s requirements, or the requirements are inconsistent or incomplete.
- There may be errors in the design. There errors arise from incorrect arguments that the program meets its specification (e.g. overlooking a case), or from idealized or incorrect assumptions about the programming language, libraries, and the hardware (e.g. a sufficiently large integer range and sufficiently available memory).
- The underlying software or hardware may fail, (e.g. operating system, disk, memory, or network).

Some failures are always detected at run-time by the underlying (virtual) machine (e.g. indexing an array out of bounds, allocating memory when none is available, or reading a file beyond its end).
Exceptions for Dependability

Other failures may be detected by programmer-added checks (e.g. checking the range of parameters of a procedure). Finally, some failures are too difficult to detect by any means.

The possibility of a failure is present even when the best effort is put forth to design error-free programs. The question then arises how programs should respond to detected failures. Suppose that a problem calls for the sequential composition of four statements,

\[ S_1 ; S_2 ; S_3 ; S_4 \]

and statements \( S_1 \) and \( S_3 \) may fail in a detectable way. In case they do fail, the sequence should be abandoned and statement \( T \) should be executed instead. In the a priory scheme, we add a test before running \( S_1 \) and \( S_3 \). In the a posteriori scheme, we run \( S_1 \) and \( S_3 \) and test if they were successful (see Box 1).

Both of these schemes are unsatisfactory! Adding explicit tests clutters up the program with additional variables and parameters. Tests may have to be repeated at different levels of the program structure, for example when the failing statement is nested inside repetitions and conditionals. The treatment of possible failures dominates the program structure to the point that the original design is no longer visible. The solution is to use a control structure for exception handling.

In addition to dealing with failures, there are two further uses of exception handling. The second use is for an unobtrusive treatment of rare or undesired cases—cases that are known to happen but that would otherwise affect the program structure in the similar sense as possible failures, in that the structure for common or desired case is no longer visible. Thus, exception handling can be used to simplify the design process by separating the concerns of common and exceptional cases.

The third use of exception handling is to allow for imperfections in implementations, like missing parts in a prototype or features that are planned for a future release. These imperfections are initially treated like a failure. Later on, when the implementation is completed, its structure does not need to change, or perhaps only a top-level handler informing the user that a feature is missing can be removed. Dually, obsolete features may be removed by replacing their implementation with one that fails. Thus, exception handling helps in evolutionary development and maintenance.

The purpose of this chapter is to develop a theory of exception handling and to show its use in the design of dependable systems. We consider exception handling in the form of \texttt{try-catch} statements, for example:

\begin{verbatim}
try
  S_1 ; S_2 ; S_3 ; S_4
catch
  T
\end{verbatim}

The meaning is that the \texttt{body} of the \texttt{try-catch} statement, here the sequential composition \( S_1 ; S_2 ; S_3 ; S_4 \), is attempted; if any of its components fails, the statement \( T \), known as the \texttt{exception handler}, is executed immediately. If the body succeeds, the exception handler is ignored. The failing statement may be nested at any level inside the body, or may be syntactically outside the body in a procedure that is called from within the body: the exception handler is determined by the dynamic call chain.