BRON84b On the use of exception handling in Modular Pascal C. Bron and E.J. Dijkstra University of Kent at Canterbury, 1984

This paper presents Bron and Dijkstra's ideas on the design of an exception handling mechanism based on several years of practical experience with exception handling facilities of the programming language Modular Pascal.

The value of an exception handling mechanism is the link between the robustness of a program and how well exceptional situations are handled. A short example illustrating causes and effects of errors leads to the identification of three properties of exceptions: an exception is a named abortion (of an expression or program control flow), one must be able to explicitly invoke the abort in a program, and since blocks are a natural delimeter of the reason for a particular abort, blocks may have associated actions to be performed in case of an abort within the block.

These principles are the basis of the exception handling mechanism described in the paper, and are concretized through the syntax used in Modular Pascal to achieve them. Exceptions are referred to by global identifiers, similar to enumerated types. Each block of code may designate actions for the exceptions to be processed at the level of the block. A set of actions for an exception is known as the "handlers" for the exception at that level, and is in some ways similar to a local procedure.

When an exception occurs, the active blocks in the program are aborted one at a time until there is an active block that specifies a handler for the raised exception. The handler is invoked and that block is terminated normally when the handler finishes. The handler is responsible for establishing the postcondition of its associated block. Dynamic scoping of handlers for a particular exception allows exceptions of local significance to be handled differently in various blocks of the program.

An advantage of the transparency of the discussed exception handling mechanism in connection with libraries, is that the burden to maintain robustness isn't entirely on the application. If some precondition fails in a library program, it can be handled as if the same check was done in the application, and the exception raised there. The appropriateness of this method also pointed out for cases where the precondition cannot be checked by the calling program, e.g. the validity of an I/O operation before it is actually performed.

The authors emphasize that the exception handling mechanism should be independent of the original source of the exception (hardware, system software, the application). Apart from the conceptual simplification, this allows application signalled errors to be treated in the same manner as errors generated by system software.

A warning is made against relying on exception handling to achieve some purpose; for example defining a handler for an end-of-file condition, and then going off in an infinite loop reading from the file using the exception handling mechanism to terminate the loop. This kind of misuse is chastised because the program is not correct on its own. Exception handling is meant to increase robustness after the program is correct.

Considerations on how to handle externally (to the program) generated exceptions lead to the concepts of "mortal" and "immortal" process states. A process is only "immortal" when it is executing an exception, because aborting an exception to execute another could cause an inconsistant system state. Instead, exceptions occurring in the "immortal" state are queued until the executing exception handler has finished.

The paper ends with a discussion of some of the implementation details encountered by the authors when adding exception handling features to Modular Pascal. They make an interesting comment, that the activation of an exception handler is very similar to a call to a procedural parameter, the only difference being the simultaneous termination of the handler and the block defining it.

The context from which the handler is called must be either the context of the block in which the exception occurred, or the block in which the exception handler is defined (if different). For practical reasons, access

to the process state as the exception occurred can be helpful for debugging or for producing error messages, so the first method was chosen for Modular Pascal.

Bron and Dijkstra view exceptions as one-way traps that must establish the postconditions of their environment (block defining associated handler). An implicit assumption is that an exception is raised because of an unusual condition at the local level, and therefore it is not desirable to continue processing at the same level.

This view is fine as long as the program is intended for a closed system at the process level, but practicality of this restriction is questionable in a system that interacts with the "world" through interrupts. The interrupt is conceptually close to the English meaning of the word "exception" of being an unusual case. In contrast to the view taken in this paper, that exceptions are abort conditions, interrupts often serve to communicate an external state change to the program, i.e. affecting the program in a manner that is independent of program correctness. I would be interesting to see a language design that consistently integrated these facets of exception handling.

The implicit stack of exception handlers and the way of finding the handler for a raised exception, are the exact methods that have been used for some time in dialects of the dynamically scoped language LISP (Franz Lisp, Zetalisp and others). Although the ideas are not new, this paper presents them in a new context, that of a traditional programming language (structured, static scoping, etc.). With the recent development of Pascal family languages that include an exception handling mechanism (Modular Pascal, Ada, Numerical Turing), language designers may be paying more attention to this area in the future.