LISKOV79
Liskov, B.H., and Snyder, A.; Exception Handling in CLU; IEEE TSE SE-5, 6 (November 1979), pp. 546-558.

In the quest for reliable (robust, fault tolerant) programs, ones that behave "reasonably" under a wide range of circumstances, linguistic structures called *exception handling mechanisms* have been designed. To enhance reliability, procedures should be defined to behave as generally as possible, responding with well-defined responses to all possible combinations of legal inputs (inputs satisfying the type constraints), even when lower level modules on which those procedures are depending fail. Exception handling mechanisms facilitate communication of information, implicit or explicit, that can be used to recover from faults such as erroneous data and failures of lower level modules. Liskov and Snyder discuss various models of exception handling and describe the exception handling facilities in the CLU programming language.

The CLU programming language was designed ". . . to provide programmers with a tool that would enhance their effectiveness in constructing programs of high quality — programs that are reliable and reasonably easy to understand, modify, and maintain. CLU aids programmers by providing constructs that support the use of abstractions in program design and implementation" [Liskov et al. 1977]. Three kinds of abstractions were recognized: data abstractions, procedural abstractions, and control abstractions.

The authors take as a basic assumption that for each procedure there is a set of circumstances in which it will terminate "normally"; in general, this is when the input arguments satisfy certain constraints and the lower level modules (implemented in both hardware and software) on which the procedure depends work properly. In other circumstances, the procedure is unable to perform any action that would lead to normal termination, but instead must notify some other procedure (for example, the invoking procedure) that an exceptional condition has occurred. The term "exception" is chosen over "error" because "exception" implies not that something is wrong, but rather that an unusual situation has arisen. An example given is that of reading character input from a file. The usual method of checking for end-of-file in a language without exception handling mechanisms is to check the character returned by the *getcharacter* routine for an "eof" character. In CLU, end-of-file can be an exceptional condition raised by *getcharacter*; only legitimate characters are then returned by the normal return mechanism.

The CLU model of exception handling involves the communication of information from the procedure activation that detects an exceptional condition (the *signaler*) to some other procedure activation that is prepared to handle an occurrence of that condition (the *catcher*). The CLU mechanism is based on a *single-level termination model*. Only the immediate caller of a procedure may handle exceptions signalled by that procedure, because, in the authors' view, all exceptions that may be raised by a procedure, whether explicitly or implicitly, must be considered part of that procedural abstraction. A multilevel mechanism, in which activations other than the immediate caller of the signaler may handle the exception, does not hold to that principle. In CLU, signals may be propagated up an activation hierarchy. In the termination model, the signalling activation ceases to exist. The resumption model allows the signaler to continue to exist after signalling; if the catcher can fix up the exceptional condition, the processing of the signaler may in fact resume. The handler may be viewed as an implicit procedure parameter of the signaler, called by the signaler when the exception handled is raised. The resumption model has more expressive power but is more complex The authors consider the relative expressive powers of the termination and resumption models, and they conclude that resumption, while the more powerful technique, is truly useful only in very limited situations and therefore does not justify the added complexity and inconvenience such a technique imposes.

In CLU, a procedure can terminate in the normal way by returning, or it can terminate in one of several exceptional conditions by signalling. In each case, result objects, differing in number and type, may be returned. A procedure heading must state the names of signals raised in the procedure and the number and types of result objects returned by each kind of signal. This facilitates static checking both that the procedure indeed signals the named exceptions and that calling procedure exception handlers are compatible with the exceptions in the procedures it calls.

Exceptions arise only from invocations of user defined procedures or built-in operators. Within a procedure, a *handler* is code which is executed when its associated exception is raised by a called procedure. Handlers are statically associated with invocations, and handlers may be attached only to statements, not to expressions, operators, or type or precision conversions. The latter decision was taken to simplify the mechanism. Each handler names one or more exceptions to be handled, followed by a list of statements

describing the appropriate actions; several handlers may be attached to a statement. The handler body may contain any legal CLU statement. If the handler body returns or signals, the containing procedure will be terminated. The handler may also be terminated by an **exit** (see below) or because it cannot handle some exception raised by an invocation within the handler itself. Otherwise, when the handler body is finished, the next statement following the handler in the normal flow is executed. The CLU **exit** statement may be used within handlers to raise directly a condition so that condition can be handled in the same procedure activation containing the handler. The **exit** is in contrast and complementary to the **signal** statement, which signals the condition to the calling procedure activation. Handlers may be placed flexibly within procedures; a handler must, however, be placed on the statement whose execution is to be terminated if the handler body terminates without returning or signalling.

If a procedure provides no handler for an exception raised by some contained invocation, a language-defined exception named *failure* is signalled. *failure* is implicitly an exception of every procedure and has one result object, a string that may contain a description of the cause of the failure. *failure* may be explicitly or implicitly signalled.

Exceptions may not be disabled. The CLU designers viewed disabling as inconsistent with encouraging good programming practice, because errors that would otherwise raise exceptional conditions may still occur without being recognized.

The authors discuss possible implementation techniques for exception handling mechanisms, including methods for associating invocations and handlers. The CLU exception handling mechanism was designed explicitly to provide information that programs, not programmers, can use to recover from exceptional conditions. The mechanism can, however, mesh smoothly with both debugging environments and production diagnostic environments.

Liskov and Snyder present a lucid, comprehensive discussion of issues in the design of exception handling mechanisms, and by justifying their design decisions in light of that discussion, the authors give the reader a clear insight into the design philosophy of the CLU exception handling mechanism. The authors contend that previous mechanisms were overly powerful and ill-structured. The CLU mechanism itself appears simple yet powerful; the underlying design is rational and appears to be consistent with the overall CLU design. The discussion of the scope of exception handlers within procedures is, unfortunately, weak; specifically, the rules for placement of exception handlers are not clear, and an example demonstrating the use of exception handling is ambiguous. The authors provide good references for further reading on CLU, exception handling, and reliable software design. The CLU exception handling effort is an important part of the evolution of exception handling mechanisms.

[Liskov et al. 1977] Liskov, Barbara H., et al.; Abstraction Mechanisms in CLU; CACM 20, 8 (August 1977) pp. 564-576.