Contents

4

Exception Handling and Tolerance of Software Faults

FLAVIU CRISTIAN
University of California, San Diego

ABSTRACT

The first part of this chapter provides rigorous definitions for several basic concepts underlying the design of dependable programs, such as specification, program semantics, exception, program correctness, robustness, failure, fault, and error. The second part investigates what it means to handle exceptions in modular programs structured as hierarchies of data abstractions. The problems to be solved at each abstraction level, such as exception detection and propagation, consistent state recovery and masking are examined in detail. Both programmed exception handling and default exception handling (such as embodied for example in recovery blocks or database transactions) are considered. An assessment of the adequacy of backward recovery in providing tolerance of software design faults is made.

4.1 INTRODUCTION

Programs are designed to produce certain intended, or standard, state transitions in computers and their peripheral devices. Most of the time, these standard state transitions can be effectively provided to program users. However, there exist circumstances which might prevent a program from providing its specified standard service. Since such circumstances are expected to occur rarely, programmers refer to them as exceptions. Exceptions have to be handled with care, since the state of a program can be inconsistent when their occurrence is detected. A normal continuation of the program execution from an inconsistent state can lead to additional exception occurrences and ultimately to a program failure. In operational computer software

[1] An earlier version of this chapter was published in "Dependability of Resilient Computers", T. Anderson, Editor, BSP Professional Books, Blackwell Scientific Publications, UK, 1989, pp. 68–97

Software Fault Tolerance, Edited by Lyu

systems often more than two thirds of the code is devoted to detecting and handling exceptions. Yet, since exceptions are expected to occur rarely, the exception handling code of a system is in general the least documented, tested, and understood part. Most of the design faults existing in a system seem to be located in the code that handles exceptional situations. For instance, field experience with telephone switching systems [Toy82], indicates that approximately two thirds of system failures are due to design faults in exception handling (or recovery) algorithms.

In the early stages of programming methodology development in the 60s, research has mostly focused on mastering the complexity inherent in the usual or standard program behavior. The first papers entirely devoted to exception handling began to appear only in the 70s, e.g. [Goo75, Hor74, Par72b, Wul75]. Early discussions of the issue were often marred by misunderstandings arising from the lack of precise definitions and terminology, but by the end of the 70s [Cri79a, Lis79] it became clear that all proposed exception mechanisms can be classified into two basic categories: termination mechanisms [And81, Bac79, Bes81a, Bro76, Cri79a, Cri80, Hor74, Ich79, Lis79, Mel77, Wul75] and resumption mechanisms [Goo75, Lam74, Lev77, Par72b, Yem82].

The two approaches can roughly be described as follows. With a termination mechanism, signalling the occurrence of an exception E while the body of a command C is executed leads to the (exceptional) termination of C. With a resumption mechanism, signalling an exception E leads to the temporary halt of the execution of C, the transfer of control to a handler associated with E and resumption of the execution of C with the command that follows the one that signalled E if the handler executes a resume command. If the handler does not execute such a command, control goes where the handler directs it to go. Thus, while with a termination mechanism, signalling an exception has a meaning similar to that of an exit command, with a resumption mechanism, it has a meaning similar to that of calling a procedure. For some time it was not clear which kind of mechanism will gain acceptance from programmers. Strong arguments that the termination paradigm is superior to the resumption paradigm are presented in [Cri79a, Lis79]. Roughly, these could be summarized as follows.

While with a termination mechanism, the meaning of calling a procedure of a module implementing some abstract data type depends only on the module state and the arguments of the call, with a resumption mechanism, the meaning also depends on the semantics of exception handlers outside the module, that are in general not known when the module is written. In addition, often such handlers must have knowledge of the module internals to handle the exception. Thus, while a termination mechanism mixes well with the information hiding principles underlying data abstraction, resumption does not.

With a termination mechanism, a programmer is naturally encouraged to recover a consistent state of a module in which an exception occurrence is detected before signalling it, so that further calls to module procedures find the module state consistent. With a resumption mechanism, the programmer does not know if after signalling an exception control will come back or not. If he recovers a consistent state before signalling, for example by undoing all changes made since the procedure start, this defeats the purpose of resumption, which is to save the work done so far between the procedure entry and the detection of the exception. If he does not recover a consistent state, then there is the possibility that the handler never resumes execution of the module after the signalling command, so that the module remains in the intermediate, most likely inconsistent, state that existed when the exception was detected. In the latter case, further module calls can lead to additional exceptions and failures.

The semantics of existing termination mechanisms is by far simpler to understand and

master than the semantics of resumption mechanisms. Moreover, it is the experience of this author that with a termination mechanism one can program all cases that "naturally" call for resumption. To illustrate this, consider a procedure C exported by a module M that composed sequentially of two subcommands C1 and C2, which uses a resumption mechanism to signal E between C1 and C2. If the outside handler RHE resumes C2 after suitably changing the state of M so as to make continuation with C2 meaningful (i.e. ensure that the causes that have lead to the occurrence of E have disappeared), then when the execution of C2 starts all the previous work done by C1 is preserved. If C would use a termination mechanism to signal E, it would have to undo all changes made by C1 before signalling E. The handler THE associated with E would then have to first perform M state changes similar to those performed by RHE (to ensure that the causes that lead to the occurrence of E have disappeared) before invoking C again. Thus, if exception masking is possible, the only advantage of a resumption mechanism over a termination mechanism is that it saves the work done before the exception is signalled, while with a termination mechanism, that work must be undone and repeated. If exception masking is not possible, then termination is clearly advantageous, since resumption is not as inductive to recovering a consistent state for M before signalling an exception as is termination. This small advantage of resumption, namely saving the work done before an exception is signalled in case the exception can be masked, is not worth in this author's view the semantic complexities associated with resumption.

A number of recent developments confirm the view that termination mechanisms are better than resumption mechanisms. Practical feedback from users of of the Mesa programming language [Mit79] incorporating the resumption mechanism of [Lam74] indicates that the use of this type of mechanism can be quite fault-prone [Hor78, Lev85]. Interestingly enough, some of the main proponents of the resumption philosophy (B. Lampson, R. Levin, J. Mitchell, D. Parnas) have abandoned it in favor of the termination philosophy [Lev85, Mit93, Par85]. Widely used programming languages such as Ada and C++ have termination exception handling mechanisms.

The purpose of this chapter is to present a synthesis of the termination exception handling paradigm. We only deal with sequential programs. Exception handling in parallel and distributed programs is still an evolving subject where no clear consensus exists [Cam86, Cri79b, Jal84, Kim82, Lis82, Ran75, Sch89, Shr78, Woo81]. In our discussion we will only examine exceptions detected by programs running on non-faulty hardware. These include exceptions detected and signalled by hardware or by lower level services such as file services. For a text attempting to integrate software and hardware aspects of fault-tolerance, the interested reader is referred to [Cri91].

4.2 BASIC NOTIONS

The goal of this section is to provide rigorous definitions for such basic concepts as program specification, program semantics, exception, program correctness and robustness, failure, fault and error.

4.2.1 Standard Program Specifications and Semantics

When a sequential program P is invoked in some initial storage state s, the *goal* is to make the computer storage reach a final state s', such that some *intended* relationship exists between s and s'.

A storage *state* is a mapping from storage unit names to values storable in those units. Typical storage unit *types* are integer, Boolean, array, disk block, stream of characters, and so on. We denote by s an initial storage state, by s' a final state, and by S the set of all possible storage states. If $s \in S$ is a state, and n is a storage unit name (for instance an integer program variable), s(n) is the value that n has in state s. To keep notations short, the convention is followed of writing n instead of s(n), and n', instead of s'(n). This means that n is used to denote ambiguously both the name of a storage unit and the value stored in that unit. Which meaning is intended should be clear from the context.

A *standard specification* G_σ (G for goal, and "σ" for standard) of a sequential program P is a relation between initial and final storage states:

$$G_\sigma \subseteq S \times S.$$

A pair (s,s') $\in S \times S$ is in G_σ if an *intended* outcome of invoking P in the initial state s is to make P *terminate normally* in the final state s'. (Normal termination in a Pascal-like language means that control returns to the 'next' command, separated by a semicolon from P.)

For example, the standard goal of a procedure F for computing factorials

procedure F(*in out* n: Integer)

might be expressed by the relation GF_σ (Goal of Factorial procedure) defined over the set Integer of machine representable integers:

$$GF_\sigma \equiv \{(n, n') \mid n, n' \in \ Integer \ \& \ n' = n!\}.$$

(The set Integer contains all integers $i \in \mathcal{Z}$ that are not smaller than a constant min $\in \mathcal{Z}$ and that are not greater than a constant max $\in \mathcal{Z}$, min $\leq 0 \leq$ max, where $\mathcal{Z}$ denotes the infinite set of mathematical integers.) The specification GF_σ associates initial values of the parameter n with final values n', such that n'=n!, where the mathematical factorial function, denoted "!", might be defined recursively by the equation

$$n! \equiv if \ n = 0 \ then \ 1 \ else \ n \times (n-1)!.$$

In most cases encountered in practice, standard program specifications are partial. A specification G_σ is *partial* if its domain dom(G_σ) is a strict subset of the set S of all possible initial states: $dom(G_\sigma) \subset S$. The domain of a relation G_σ is the set of all initial states $s \in S$ for which there exist final states $s' \in S$ in G_σ :

$$dom(G_\sigma) \equiv \{s \in S \mid \exists s' \in \ S : (s, s') \ \in \ G_\sigma\}.$$

For example, GF_σ is partial. Indeed, GF_σ does not define a final value for n when its initial

value is negative, because the mathematical factorial function "!" is undefined for negative integers:

$$GF_\sigma = \{(0,1),(1,1),(2,2),(3,6),(4,24),(5,120),(6,720),(7,5040),(8,40320),...\}$$

To emphasize the partial nature of a standard specification G_σ, it is customary to structure it into a standard *precondition* pre_σ that characterizes the domain of the specification

$$pre_\sigma : S \rightarrow \{true, false\}, \qquad s \in dom(G_\sigma) \equiv pre_\sigma(s) = true$$

and a standard *postcondition* $post_\sigma$ that is the characteristic predicate of G_σ

$$post_\sigma : S \times S \rightarrow \{true, false\} \qquad (s, s') \in G_\sigma \equiv post_\sigma(s, s') = true.$$

Thus, a precondition is used to indicate *when* a service can be provided, and a postcondition is used to describe *what* service will be provided.

A *program* is a syntactic object that is built according to a certain programming language *grammar*. For example, the text in Figure 4.1 (written in accordance with some Pascal-like grammar) might be taken as being a procedure that attempts to accomplish the standard goal GF_σ mentioned above:

```
 procedure F(in out n: Integer);
var k,m: Integer;
 begin
    k:=0; m:=1;
    while k < n
    do k:=k+1 ;
      m:=m × k
    od;
    n:=m
 end;
```

Figure 4.1 A standard factorial program

The *standard semantics* $[P]_\sigma$ of a program P is the *actual* function from input to output states that P computes *when* it terminates normally:

$$[P]_\sigma \subseteq S \times S.$$

A pair of states (s,s') is in $[P]_\sigma$ if, when invoked in the initial state $s \in S$, the program P *terminates normally* in the final state $s' \in S$.

For example, the termination of the procedure F is normal either if n is negative (in which case the final value n' is 1) or if n is positive and n! is a machine representable integer (in which case the final value n' is n!). An overflow occurrence when trying to compute n! does

not result in normal termination, as will be discussed later. On a microcomputer using signed 16-bit integer representation (where max $<$ 8!) the standard semantics of the procedure F is the function:

$$[F]_\sigma = \{(min, 1), ..., (0, 1), (1, 1), ..., (6, 720), (7, 5040)\}$$

That is, on such a microcomputer, F terminates normally whenever it is invoked with an argument smaller than 8.

The set of all initial states $s \in S$ for which a program P terminates normally in some final state $s' \in S$ which satisfies the standard specification G_σ is the *standard domain* SD of P (with respect to G_σ):

$$SD \equiv \{s \mid \exists s' : (s, s') \in [P]_\sigma \ \& \ (s, s') \in G_\sigma\}.$$

For example, the standard domain of the program F with respect to the specification GF_σ is the domain $\{0,1,...,7\}$ of the relation $[F]_\sigma \cap GF_\sigma$. The characteristic predicate of the standard domain can be computed as being the weakest precondition for which P terminates normally in a final state satisfying G_σ [Dij76, Cri84].

4.2.2 Exceptional Program Specification and Semantics

If a program P is invoked in an initial state which is outside the standard domain SD, the standard service G_σ specified for P *can not* be provided by P. The set of all states which are not in the standard domain is the *exceptional domain* ED of that program:

ED $\equiv$ S - SD .

For example, the exceptional domain of the factorial procedure F with respect to its standard specification GF_σ is Integer-$\{0,...,7\}$, that is, $\{$min,...,-1$\} \cup \{$8,...,max$\}$.

An invocation of a program in its exceptional domain is an *exception occurrence*. (Note that no actual detection is implied.) By the above definition, an exception occurrence is synonymous with impossibility of delivering the standard service specified for a program. If the standard domain associated with a program and specification includes the set of all possible input states, there will be no exception occurrences when that program is invoked. Unfortunately, such programs and specifications are rarely encountered in practice. Most often, the exceptional domains associated with programs and specifications are not empty.

A characteristic of an exception occurrence is that, once such an event is *detected* in a program, it is not sensible to continue with the sequential execution of the remaining operations in that program. For example, an exception occurrence (say, for an initial state $i < 0$) detected during the execution of the first operation F(i) of the program below

```
F(i); F(j); m:= i+j;
```

reveals that the standard goal $m' = (i! + j!)$ cannot be achieved, and hence, it does not make sense to continue normal execution of the program by invoking the next operation F(j). Thus, to handle exception occurrences, it is convenient to allow for occasional (exceptional) alterations of the sequential (standard) composition rule for operation invocations.

An *exception mechanism* is a language control structure which allows a programmer to express that the standard continuation of a program is to be replaced by an exceptional continuation when an exception is detected in that program. A direct way of associating several continuations with a single program is to make that program have several *exit points*: one standard exit point, to which a standard continuation may be associated, and zero or more exceptional exit points, to which exceptional continuations may be associated.

The intention is that the program should return normally if it *can* provide its specified standard service, and should return exceptionally if it *cannot*. In this way, a program can endeavor to notify its invoker *directly* that a requested standard service is (or is not) provided by simply returning normally (or exceptionally). To let a user of a program P distinguish among different exceptional returns from P, alphanumeric *exception labels* can be used to label distinct exceptional exit points of P. The symbol σ (which cannot be confused with an exception label) will be used to denote the standard exit point of a program.

Since all examples to be given in this chapter are phrased in terms of the simple exception mechanism defined in [Cri79a, Cri84], it is appropriate at this point to briefly recall its main characteristics.

The designer of a procedure P indicates that P has an exceptional exit point "e" by declaring "e" in the header of P as follows:

procedure P *signals* e.

An invoker of P defines the exceptional continuation (if e is signalled by an invocation of P) to be some operation K by writing

P[e:K] .

To detect and handle the occurrence of e, the designer of P may explicitly insert in the body of P the following syntactic constructs:

(a) [B:H]
(b) O[d:H]

In the first, B is a Boolean expression (or run-time check, or executable assertion). In the second construct, O is an operation which can signal some exception d. The handler H may be a (possibly empty) sequence of operations and may terminate with a "*signal* e" exceptional sequencer. The meaning of an (a) or (b) construct inserted in the body of P may be explained informally as follows. If B evaluates to *true* or O signals d, then H is invoked. If H terminates with a "*signal* e" sequencer, then the standard continuation of the (a) or (b) construct is abandoned in favor of an exceptional continuation (*e.g.*, K) associated with the e exit point of P. In the remaining cases, *i.e.*, if B evaluates to *false* or O terminates normally or the execution of H does not terminate with a "*signal* e" sequencer, the standard continuation of the (a) or (b) construct is taken. If the designer of P did not associate the handler H with the exception d which can be signalled by O, then d would be an exit point for P too. Such an exceptional exit (not explicitly declared for P by its designer) would be taken whenever O signals d after being invoked from P.

Note that the occurrence of e can be *detected* in P either because some Boolean expression B evaluates to true, or because an operation O invoked from P signals a lower level exception d. In the latter case, the detection of e in P *coincides* with the *propagation* of the (lower level) exception d in P. The problem of systematic placement of Boolean expressions in programs

so as to detect all possible exception occurrences is investigated in [Bes81a, Sta87]. The verification methods described in [Cri84] can be used to prove that all exceptions, whether detected by Boolean expression evaluations or lower level exception propagation, are correctly detected in a program.

As an example, Figure 4.2 contains a variant FE (Factorial with exceptions) of F which signals a "negative" exception whenever the input is negative.

```
procedure FE(in out n: Integer) signals negative;
var k,m: Integer;
begin
   [n<0: signal negative];
   k:=0; m:=1;
   while k < n
   do k:=k+1 ;
      m:=m × k
   od;
   n:=m
end;
```

Figure 4.2 A factorial program with exceptions

Software designers often anticipate that the exceptional domains of the programs they write may be nonempty, and decide to provide alternative *exceptional services* when the intended standard services cannot be provided. Let E be the set of exception labels that the designer of a program P *declares* for P in order to identify a set of specified exceptional services that P will deliver when the standard service G_σ cannot be provided.

As discussed above, a program P can also signal exceptions that are declared for some component operations invoked from P, but are not declared for P itself. These are the exceptions signalled by lower level operations invoked from P and for which there are no associated handlers in P. As an example, assume that the definition of the language used to write the FE procedure specifies that an integer assignment, such as m := m × k, signals the language defined exception "intovflw" when the result of evaluating the right hand side expression is an integer that is not machine representable. Although the designer of the procedure FE did not declare an "intovflw" exceptional exit point, the procedure can signal this exception whenever the execution of m := m × k results in an arithmetic overflow. In such a case, the entire procedure FE terminates at the "intovflw" exit point, which was not declared for FE. We denote by X the set of *all* (declared and undeclared) exception labels that a program P can signal.

The intended state transition that a program P should perform when an *anticipated* exception e ∈E is detected can be specified by an *exceptional specification* G_e:

$$G_e \subseteq S \times S\ .$$

A pair of states (s,s') is in G_e if the *intended* outcome of invoking P in the initial state s ∈ S is to make P terminate at its declared "e" exceptional exit point in the final state s' ∈ S.

Like a standard specification, an exceptional specification may be partial, and may be structured into an exceptional precondition pre_e

$$pre_e : S \to \{true, false\}\ , \qquad s \in dom(G_e) \equiv pre_e(s) = true$$

and an exceptional postcondition $post_e$

$$post_e : S \times S \rightarrow \{true, false\}\,, \qquad (s, s') \in G_e \equiv post_e(s, s') = true.$$

The exceptional preconditions pre_e, e ∈ E, divide the input space of P into several labeled exceptional sub-domains. When a program P is invoked in the e-labeled exceptional sub-domain, one says that the exception e occurs.

We illustrate the notion of an exceptional specification, by specifying that the parameter n of FE should remain unchanged when it is initially negative. This can be done either by directly giving $GF_{negative}$

$$GF_{negative} \equiv \{(n, n') \mid n, n' \in Integer \,\&\, n < 0 \,\&\, n = n'\}$$

or by giving a pair of pre- postconditions:

$$pre_{negative} \equiv n < 0, \qquad post_{negative} \equiv n = n'.$$

Although the practice of structuring specifications into pre- and postconditions is very common, to keep notations short, it will not be used further in this chapter. The definitions to be given can be translated in terms of pre- and postconditions (if desired) by using the pre- and postcondition definitions given above.

The *exceptional semantics* $[P]_e$ of a program P with respect to an exception label e ∈X (either declared or undeclared for P) is the function (from input to output states) that P *actually* computes between its start and its termination at the "e" exit point:

$$[P]_e \subseteq S \times S\,.$$

A pair of states (s,s') is in $[P]_e$ if, when invoked in the initial state s ∈ S, P *terminates* at its "e" exit point in the final state s' ∈ S.

For example, the function computed by FE when the "negative" exception is signalled is the identity function over the set {min,...,-1}:

$$[FE]_{negative} = \{(min, min), ..., (-1, -1)\}.$$

The programs F and FE also compute the functions:

$$[F]_{intovflw} = [FE]_{intovflw} = \{(8, 8), (9, 9), ..., (max, max)\}$$

$$[FE]_{\sigma} = \{(0, 1), (1, 1), ..., (6, 720), (7, 5040)\}\,.$$

4.2.3 Program Failures, Faults, and Errors

Consider a program P whose specification is structured into a standard service G_σ and zero or more exceptional services G_e, $e \in E$. The *specification* G of P is the set of all standard and exceptional specifications defined for P:

$$G \equiv \{G_x \mid x \in E \cup \{\sigma\}\},$$

where E is the set of exception labels *declared* for P, that is, the set of exceptions *anticipated* by the designer of P. By convention, the standard exit point σ is always *declared* for any program. We assume that such a specification is implementable by a deterministic program, that is we assume that:

$$\forall x, y \in (E \cup \{\sigma\}) : (x \neq y) \Rightarrow (dom(G_x) \cap dom(G_y) = \{\ \}).$$

Let AI (Anticipated Inputs) denote the set of all inputs for which the behavior of P is specified:

$$AI \equiv \bigcup_{x \in E \cup \{\sigma\}} dom(G_x)$$

We denote by UI (Unanticipated Inputs) the remaining possible input states, that is, the input states for which the behavior of P was left unspecified:

$$UI \equiv S - AI.$$

A specification is *complete* if it prescribes the behavior of P for all possible input states $s \in S$:

$$S \subseteq AI\ .$$

For instance, the specification $GF = \{GF_\sigma, GF_{negative}\}$ is not complete, since it does not specify the result to be produced when FE is invoked with a positive argument whose factorial is not machine representable. An example of a complete specification is $CGF = \{GF_\sigma, GF_{negative}, GF_{overflow}\}$, where

$$GF_{overflow} \equiv \{(n, n') \mid n, n' \in Integer \,\&\, n! > max \,\&\, n = n'\}\ .$$

The *semantics* of a program P with exceptional exit points X is the set of all semantic functions $[P]_x$, $x \in X \cup \{\sigma\}$, that P computes between a start and a termination at some (declared or undeclared) exit point x:

$$[P] \equiv \{[P]_x \mid x \in X \cup \{\sigma\}\}.$$

A program P is termed totally *correct* with respect to a specification G if its actual semantics [P] is *consistent* with the intended semantics G:

$$\forall x \in E \cup \sigma : dom(G_x) \subseteq dom([P]_x) \& \forall s, s' \in dom(G_x), S : (s, s') \in [P]_x \Rightarrow (s, s') \in G_x$$
.

That is, P is correct if it actually terminates at some declared exit point x every time the specification G requests that it terminate at x. Moreover, any final state s' actually produced by P from an anticipated initial state $s \in dom(G_x)$, for some $x \in E \cup \{\sigma\}$, is always consistent with the stated intention G_x. Correctness does not necessarily imply that [P] and G are equal. While P must be deterministic (i.e. [P] must be a function) G might be nondeterministic (i.e. G might not be a function). Moreover, there can be states s,s' such that (s,s')$\in [P]_x$ but s$\notin dom(G_x)$. For example, P might terminate normally ((s,s')$\in [P]_\sigma$) if invoked in initial states s for which the specification does not prescribe normal termination (s$\notin dom(G_\sigma)$). Methods for proving the total correctness of programs with exceptions are discussed in [Cri84].

For example, the program FE is totally correct with respect to the specification GF, but is incorrect with respect to the complete specification CGF. An example program RFE (Robust Factorial with Exceptions) that is totally correct with respect to the complete specification CGF is given in Figure 4.3. The overflow exception declared for RFE is detected *when* the lower level machine exception intovflw is propagated into RFE.

```
procedure RFE(in out n: Integer) signals negative, overflow;
var k,m: Integer;
begin
   [n<0: signal negative];
   k:=0; m:=1;
   while k<n
   do k:=k+1;
       m:=m × k[intovflw: signal overflow]
   od;
   n:=m
end;
```

Figure 4.3 A robust factorial program with exceptions

A program that is totally correct with respect to a complete specification is *robust*, in that its behavior is predictable for *all* possible inputs. Besides other characteristics such as functionality, ease of use, and performance, robustness is one of the most important aspects of a program and its documentation. The procedure RFE is robust since its behavior is correctly predicted by the specification CGF for all possible initial values of n.

A robust program whose exceptional specifications G_e, $e \in E$, are identity relations is called *atomic* with respect to the exceptions $e \in E$. For an external observer, any invocation of such a program has an "all or nothing" effect: either the specified standard state transition is produced or an exception is signalled and the state remains unchanged. Methods for proving the correctness of data abstractions with atomic operations are discussed in [Cri82].

Remark: The adjective "atomic" is over-used in the programming community and one has to carefully distinguish among the different meanings it takes in different contexts. In a multiprocessing context, it is used to qualify the interference-free or serializable execution of parallel operations [Ber87, Bes81b]. In a context in which program interpreter crashes can occur, a command C is said to be atomic with respect to crashes if a crash occurrence during the execution of C either causes no stable state transition or causes the stable state transition specified for C [Cri85]. Clearly atomicity with respect to concurrency on one side and atomicity with respect to exceptions or crashes on the other side are fairly distinct, orthogonal concepts. Although the basic idea behind atomicity with respect to exceptions and atomicity with respect to crashes is the same, work on verifying atomicity with respect to crashes and atomicity with respect to exceptions shows that these are two fairly distinct concepts, usually implemented by distinct run-time mechanisms [Cri84, Cri85]. *End of remark*.

If a program P is not totally correct with respect to a specification G, there exist (anticipated) input states $s \in$ AI for which P's behavior *contradicts* G. The set of all input states for which the actual behavior of P contradicts the specified behavior G is the *failure domain* FD of P with respect to G:

$$FD \equiv AI - (SD \cup AED) ,$$

where AED denotes the set of all input states for which correct exceptional results are produced

$$AED = \bigcup_{e \in E} \{ s \mid \exists s' : (s, s' \in [P]_e \ \& \ (s, s') \in G_e \} .$$

The characteristic predicate of the AED domain can be computed as being the disjunction of the weakest preconditions for which P terminates at declared exit points e in final states satisfying the exceptional specifications G_e [Cri84].

As an example, observe that the failure domain of the program FE with respect to the specification CGF is the set of all positive integers with non machine representable factorials. Note also that for a program P to fail for an input, the behavior of P for that input must be described by the specification G. One can talk about a *specification failure* whenever a specification fails to prescribe the program behavior for some inputs, that is, whenever UI $\neq \{\}$. Specification failures are in fact as annoying as program failures.

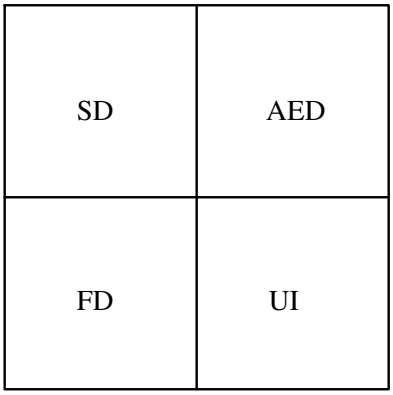

Figure 4.4 A partition over the set of all input states

The domains SD, AED, FD, and UI introduced previously define a partition over the set of all input states, in that they are pairwise disjoint and their union is the set of all possible input states S (see Figure 4.4). A goal of good software practice is to make sure that the FD and UI domains are empty. Methods for computing the domains SD, AED, FD for programs with exceptions are described in [Cri84].

A *program failure* occurs when a program is invoked in its failure domain FD. Thus, a program failure is synonymous with *divergence* between specified and actual program behavior. A failure of a sequential program P for an input $s \in$ FD can be of one the following four types:

1) P loops indefinitely: $\neg\exists x \in X \cup \{\sigma\} : s \in dom([P]_x)$
2) an exception u that was not declared for P is detected: $\exists u \in X - E : s \in dom([P]_u)$
3) P terminates normally (*i.e.*, at its standard exit point) in a final state s' which does not satisfy the standard specification of P: $\exists s' : (s, s') \in [P]_\sigma \,\&\, (s, s') \notin G_\sigma$,
4) P terminates by signalling a declared exception $e \in E$ in a final state s' which does not satisfy the exceptional specification $G_e : \exists s' : (s, s') \in [P]_e \,\&\, (s, s') \notin G_e$.

Note that the definition given to the notion of program failure does not imply that an occurrence of a program failure is actually recognized (detected) by a program user, either human or another program. Typically, failures of type (3) or (4) which result in *proper* program termination in some erroneous state (that is, termination at a declared exit point) are more difficult to detect than failures of type (1) or (2) which result in *improper* program termination. Indeed,

non-termination (detected by a timeout) or termination at an undeclared exit (e.g. with a run-time "error message") are obvious indications of a faulty program for an observer external to the program (e.g. human user, other program, or operating system), while proper program termination is a behavior that an external observer expects from a correct program.

Not only are failures of type (3) and (4) more difficult to detect, but they also have a greater potential for being disruptive than those which manifest themselves by improper program termination, since they may result in further failures. Often, proper termination in some unrecognized erroneous state is followed by further program invocations from that state. These invocations can result in further unpredictable behavior, until an external human observer discovers at some later time a discrepancy between specified and actual program results. At that time, very little can be said about the consistency of the program state.

Failures of type (1) or (2) are referred to as *confined failures*, while failures of type (3) or (4) are referred to as *unconfined failures*. Corresponding to these two general failure classes, the failure domain FD can be divided into a *confined failure domain* CFD, and an *unconfined failure domain* UFD:

$$FD = CFD \cup UFD\,.$$

A program P which, for every input, either terminates properly in a state satisfying the specification G or suffers a confined failure will be termed *partially correct* with respect to the specification G. In other terms, a partially correct program is one that has empty UI and UFD domains. This notion of partial correctness is somewhat different from the classical notion of partial correctness [Flo67, Hoa69], where partial correctness is defined in term of a pre and postcondition. In this chapter, when we talk about partial correctness we assume a constantly true precondition and a complete specification. Our notion of partial correctness is however such a natural extension of the classical notion that we feel it does not justify the introduction of a new term for it (in [Gra93] programs that are partially correct — in the sense of having empty UI and UFD domains — are termed "fail-fast").

The interest in partially correct programs comes from the fact that such programs are *safe*, in the sense that they never output erroneous results to their users. Methods for verifying that programs with exceptions are partially correct are described in [Bac79, Bro76, Luc80]. In combination with a run-time mechanisms for detecting improper program termination and an alternate program for outputting a default "safe value" (when correct primary output is unavailable from a primary partially correct program in a timely manner), partially correct programs can be used to build *fail-safe* programs, that is, programs that either deliver a correct output in a timely manner or otherwise deliver a predefined output considered "safe" for the application at hand (like "close ATM window" when the procedure that identifies current client fails). Note also that in the literature on database transactions [Ber87, Gra93] it is standard to assume that *transactions* are implemented by partially correct programs.

The occurrence of program failures can be attributed to the *existence* of design faults. Thus, if the failure domain FD of a program P with respect to a specification G is not empty, one says that the program P has a *design fault* with respect to the specification G.

For example, any invocation of the program F (which is incorrect with respect to the complete specification CGF) with an initial value n such that n! is not machine representable results in improper termination of F. The absence of a handler associated with the intovflw language defined exception in Figure 4.1 is thus a design fault. This (confined) design fault leads to the existence of a nonempty confined failure domain (the set of all positive integers with non-machine representable factorials) for F.

```
procedure FFE(in out n: Integer) signals negative;
var k,m: Integer;
begin
   [n<0: signal negative];
   k:=0; m:=1;
   while k < n
   do m:=m × k;
     k:=k+1;
   od;
   n:=m
 end;
```

Figure 4.5 A faulty factorial program with exceptions

Consider now the case when the unconfined failure domain of a program P contains a state s such that, when started in s, P terminates properly at some declared exit point x in a final state s" different from the intended final state s' prescribed by the specification G_x for s. Then there exist program variables v whose state is *erroneous*, in that their value s"(v) is different from the value s'(v) prescribed by the specification G_x. The value that such a variable v possesses in s" is called an *error* (with respect to the specification G_x).

As an example of an unconfined design fault which can lead to output errors that can spread to other programs, consider the version FFE (Faulty Factorial with Exceptions) of the procedure FE, given in Figure 4.5, in which the two operations of the loop body have been transposed. The standard domain of FFE with respect to the specification CGF is $\{0\}$. Whenever FFE is invoked with an actual parameter that is strictly positive, the final value of n (0) is an error since it is different from the final value prescribed by CGF.

4.3 EXCEPTION HANDLING IN HIERARCHICAL MODULAR PROGRAMS

In this section, we investigate what it means to handle exceptions in modular programs structured as hierarchies of data abstractions. The basic problems to be solved at each abstraction level, such as exception detection, consistent state recovery, exception masking and propagation are discussed. Both programmed and default exception handling methods are considered. An assessment of the effectiveness of backward recovery based default exception handling (as embodied for example in the recovery block mechanism [Hor74]) in providing effective tolerance of residual design faults is provided.

The scope of this section is limited to discussing tolerance of program design faults, not tolerance to specification faults or lower level service failures, so we assume that specifications are correct and the lower level services on which a hierarchical modular program depends are also correctly functioning. Such lower level services might of course signal exceptions, but we assume that the standard and exceptional state transitions that these services undergo are consistent with their specification. For example if the program depends on a file service, exceptions such as "no-such directory" or "end-of-file" can be signalled, but we assume that they are detected and handled correctly at the file service level before being propagated to our hierarchical program. This topic of providing tolerance to software design faults affecting a given program is sufficiently complex to deserve consideration separate from other interesting

areas like tolerance of hardware failures or lower level service failures. Our opinion is that responsibility for coping with faults specific to each interpretation level (i.e. detection and at least signalling) must fall on the designers of the level concerned. For an attempt to integrate software and hardware fault-tolerance, the interested reader is referred to [Cri91].

To keep the presentation short, we give fewer examples than in the first, more introductory, part. Detailed examples of the often tricky problems posed by exception handling in programs structured as hierarchies of abstract data types can be found in [Cri82].

4.3.1 Hierarchical Program Structure

In the 70s, it became clear that *data abstraction* is a powerful mechanism for mastering the complexity of programs [Hoa72, Lis74, Par72a, Wul76]. Researchers in programming methodology suggested that the right way to solve a programming problem was to repeatedly decompose the problem into sub-problems, where each sub-problem could be easily solved by writing an "abstract" program module in an "abstract" language which possessed all the right data types needed to make the solution to that sub-problem simple. Those data types assumed in such modules that were not available in the programming language used were called "abstract", as opposed to the built-in "concrete" types. The implementation of these abstract types then was just a new programming problem, which had to be solved recursively by writing other program modules in terms of other (possibly abstract) data types. The programming process would then continue until the problem of implementing all assumed abstract data types was solved only in terms of concrete types.

The programming methodology outlined above leads to programs which are structured into a *hierarchy of modules* [Par72a, Par74], where each module implements some instance of an abstract data type. Visually, such a hierarchy may be represented by an acyclic graph as in Figure 4.6. Modules are represented by nodes. An arrow from a node N to a node M means that N is a *user* of M, that is, the successful completion of (at least) an operation N.Q exported by N depends on the successful completion of some operation M.P exported by M. In what follows we frequently refer to the hierarchy illustrated in Figure 4.6, by using O, P, and Q as names for operations exported by the modules L, M, and N, respectively, and by using d, e, f as generic exception names for the operations O, P, and Q, respectively.

When observed from a user's point of view (*e.g.*, N), a module M is perceived as being an (abstract) *variable* declared to be of some *abstract data type.* To make use of a module M, it is only necessary to know the set of *abstract states* that may be assumed by M and the set of abstract state *transitions* that are produced when the operations exported by M are invoked. The internal structure of a module is not visible to a user. When seen from inside, a module M is a set of *state variables* and a set of *procedures.* A state variable may be either of a predefined type (*e.g.*, integer, Boolean, array) directly provided by the programming language being used, or may be of some programmer defined abstract type, in which case, it is implemented by some lower level module (*e.g.*, L).

The *internal state* of a module M is the aggregation of the abstract states of its state variables. The *abstract state* of M is the result of applying an *abstraction function* A to its internal state [Hoa72, Wul76]. In general, A is a partial function defined only over a subset $I \subseteq S$ of the set of all possible internal states of the module. (In practice, this subset is defined by using an *invariant* predicate [Hoa72, Wul76].) The internal states in I are said to be *consistent* with the abstraction that the module is intended to implement. During a procedure execution,

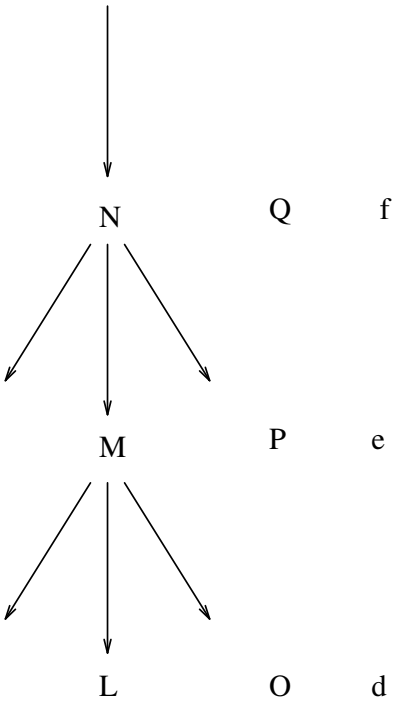

Figure 4.6 An acyclic graph representing a program hierarchy

a module may pass through a set of intermediate internal states i which are inconsistent (i.e. i $\notin$ I) and for which A, and hence the abstract state, are not defined.

4.3.2 Programmed Exception Handling

Every procedure P exported by a module M is designed to accomplish a specified standard service: some intended internal, and hence abstract, state transition. As discussed in the first part, P can also be required to provide zero or more exceptional services in addition to its standard service. Let

$$G \equiv \{G_x \mid x \in E \cup \{\sigma\}\}$$

denote the global specification of M.P, where E is the set of all exceptions declared for P. We require such specifications to be strong enough to exclude an inconsistent state being an intended outcome when P is invoked in a consistent state (remember that we are interested in discussing the adequacy of default exception handling in providing tolerance of program design — not specification — faults):

$$\forall x \in E \cup \{\sigma\} : \forall s \in I : \forall s' \in S : (s, s') \in G_x \Rightarrow s' \in I.$$

Usually, the standard and exceptional specifications of P are not defined by enumerating all the component pairs of each G_x, $x \in E \cup \{\sigma\}$, but by using pre and post conditions

as mentioned in the first part of the chapter. We use sets and relations to specify the operations we present in our discussion instead of predicates because they provide us with a more compact representation. Of course, the entire discussion can be translated in terms of pre, postconditions, and invariants without any difficulty.

Consider now an exception e declared for a procedure P and let G_e be the specification of the state transition to be produced when e occurs. As discussed in the first part, an occurrence of e may be *detected*: (a) either by a run-time check, or (b) because a lower level exception d is propagated in P by a lower level operation O invoked from P. In the latter case the detection of e coincides with the propagation of d in P, that is, with the invocation of a handler H *of* e. Although this handler is syntactically *associated* with the lower level propagated exception d by using a (b) language construct of the form O[d:H], it is essential to understand that its semantics (the exceptional state transition it has to accomplish) is determined solely by the exceptional specification G_e. We use the phrase "handler associated with" to state a syntactic fact and the phrase "handler of" to reflect a semantic knowledge.

When an exception occurrence is detected in a module M, an intermediate *inconsistent state* outside the set I may exist. An example in [Cri82] illustrates that further invocations of a module left in such a state (by some exception occurrence not appropriately handled) can lead to unpredictable (*i.e.*, unspecified) results and to subsequent unanticipated exception occurrences. To avoid such consequences, it is necessary that measures for the recovery of a consistent state are taken by the handler H *of* e.

Let $s \in$ I be the consistent state prior to the invocation of P and $i \in$ S be the state of M when e is detected. A set of state variables of M is called a *recovery set* RS if by modifying the state that these variables have in i, a final state s' such that (s,s') $\in G_e$ can be reached. Note that according to our earlier definition of an error, the values assumed by the variables of RS in the intermediate state i are not erroneous with respect to the specification G of P, since G does not prescribe through what intermediate states P should transit between successive invocations. In general, there exist several recovery sets for an exception detection. From a performance point of view, the most interesting recovery set is the one with the fewest elements. An *inconsistency set* IS is a recovery set such that for any other recovery set RS: $\mid IS \mid \leq \mid RS \mid$, where $\mid \ \mid$ denotes set cardinality. Because of this minimality property, an IS can be regarded as being a characterization of that part of the state which is "effectively" inconsistent when the occurrence of e is detected. For nontrivial examples of inconsistency and recovery sets, the interested reader is referred to [Cri82].

If the decision is taken that module operations should be atomic with respect to exceptions, then two other kinds of recovery sets may be of interest. Let us define the *inconsistency closure* IC associated with the intermediate state i, existing when e is detected, to be the set of all state variables modified between the entry in P and the detection of e. An IC is a recovery set (for any abstraction function A and any invariant I), since by resetting all the modified variables to their initial (abstract) states, a final internal state s' identical to the initial internal state s is obtained. The second kind of recovery set is the crudest approximation one can imagine for an IS (an inconsistency closure is a better one). This approximation is obtained by taking the whole set of state variables of M (with their state in s) to form a complete *checkpoint* CP of the initial internal state s of M. Clearly, by restoring all variables of M (whether modified or not between the entry in P and the detection of e) to their state prior to the invocation of P, a final internal state s' identical to the initial state s is obtained.

After the above discussion on recovery sets, we can now describe the task of a handler H of e as being to *recover* some RS before *signalling* e. Of course, if the state i in which e

```
procedure P signals e;
begin
   .
   .
   [DET: recover RS; signal e]
   .
   .
end;
```

Figure 4.7 Recovery of a consistent state before signalling an exception

is detected already satisfies the specification G_e, i.e. (s,i) $\in G_e$, then no recovery action is necessary, that is, the IS associated with such an exception detection is empty.

If the exceptional postcondition G_e specified for the detected exception e is not the identity relation *i.e.*, P is not intended to behave atomically with respect to exceptions, then the recovery action of H is said to be *forward* [Ran78]. From an internal point of view, the recovery of an RS is "forward" if the final state of at least one variable in RS is different from its state when P was invoked. A forward recovery action is based on knowledge about the module semantics (captured by the internal invariant I, the abstraction function A, and the specification G_e) and, thus, has to be explicitly programmed by the implementer of P. However, if P is intended to have an atomic behavior, then the determination of the IC or CP recovery sets (which are independent of I, A, and G_e) can be done automatically at run-time. Checkpointing techniques have long been used for recovering consistent system states. Later, it has been proposed [Ber87, Gra93, Hor74], to leave the task of computing the inconsistency closures associated with the intermediate inconsistent states i through which a system may pass to special mechanisms, called *recovery caches* or *log managers*.

The (automatic) recovery of inconsistency closures or checkpoints is referred to as *backward recovery* [Ran78]. More generally, one can view the recovery of some RS as being "backward" if all the variables in RS recover their states prior to the invocation of P. To avoid confusion between explicitly programmed "backward" recovery and that performed by a recovery cache, a log manager, or a checkpointing mechanism, we will call the latter *automatic backward recovery*.

To conclude this discussion on the detection and recovery issues raised by the handling of an exception e in a procedure P, let us denote by "[DET:" the "[B:" or "O[d:" syntactic construct used to detect an occurrence of e. The handling of e in the procedure M.P where it is detected may be summarized as shown in Figure 4.7: a consistent state must be recovered for module M before the exception e is signalled to the user of M.P. Let us now investigate the consequences that a *propagation* of e by $M.P$ may have for the invoking procedure $N.Q$ (see Figure 4.6). In some cases, the propagation of a lower level exception e in a procedure Q is a consequence of invoking Q within its own exceptional domain. Such a situation was illustrated by the example of Figure 4.3 where the lower level exception intovflw is propagated in RFE whenever RFE is invoked in the exceptional subdomain n!>max. However, there exist cases in which a lower level exception may be propagated in a procedure even though that procedure was invoked within its standard domain.

As an example, suppose that module N is a file management module which exports a procedure "CREATE a file containing Z disk blocks," where Z is of type positive integer. Assume that each file is completely stored either on a disk d_1 or on a disk d_2 and that M_1, M_2 are the modules which manage the free blocks left on d_1 and d_2, respectively. An initial state in

procedure CREATE (Z:positive-integer) *signals* ns;
begin
.
M_1.AL(Z)[do:M_2.AL(Z)[do: recover RS; *signal* ns]];
.
.
end;

Figure 4.8 Space allocation in a program

which at least one disk has more than Z free blocks will be in the standard domain of CREATE and a state in which both disks have less than Z free blocks will be in its exceptional domain. Suppose that the space allocation within CREATE is programmed as shown in Figure 4.8.

If CREATE is invoked in a state in which d_1 has less than Z free blocks, then the handler associated with the "do" (Disk Overflow) exit point of the space allocation procedure M_1.AL is invoked. Now there remain two possibilities. If the initial state was in the standard domain, that is, d_2 has at least Z free blocks, then M_2.AL terminates normally and the continuation is standard (*i.e.*, the handler associated with the "do" exit point of M_2.AL , is not invoked). Otherwise, if the initial state was in the exceptional domain of CREATE, the propagation of the disk overflow exception by M_2.AL coincides with the detection of the "ns" (No Space) exception declared for CREATE. The handler *of* "ns" (the sequence "recover RS; *signal* ns") recovers a consistent state before propagating "ns" higher up in the hierarchy.

This example illustrates two points. First, the "[DET:" symbol used previously in Figure 4.7 may sometimes be a sequence of "O[d:" detection symbols (this is frequently the case when dealing with exceptions due to transient input/output faults [Cri79a]). Second, lower level exception propagations can be stopped by higher lever procedures.

If a procedure Q can provide its standard service despite the fact that a lower level exception e is propagated in Q, we say that Q *masks* the propagation of e.

4.3.3 Default Exception Handling

As mentioned in the first part, one of the main goals in program design is to achieve correctness and robustness. Despite recent advances in understanding the issues involved in the production of correct and robust programs, the design of software that is correct and robust remains a nontrivial task. In practice, instead of relying on rigorous programming and validation methods, many software designers rely upon their intuition and experience to deal with possible exception occurrences. Therefore, the identification and handling of the exceptional situations which might occur is often just as (un)reliable as human intuition.

Consider now the case of a faulty procedure P exported by a module M. Let us assume for the moment that M.P is partially correct, that is, any invocation of M.P in its (non-empty) failure domain is detected because an unanticipated exception u is propagated by a lower level operation L.O in P (in particular, u might be a time-out exception). The case when a failure of M.P remains undetected after a proper termination of M.P will be discussed later.

Now, what is a sensible reaction to such a situation? For example, what exceptional continuation should be associated with the exception u propagated from a lower level? One possible solution (adopted in ADA [Ich79]) is to continue the propagation of u in the higher level module N. Such free exception propagations across module boundaries may have dangerous consequences. First, according to the "information hiding principle" of modular programming

```
procedure P signals e;
begin
   .
   .
   .
   .
end[ :DH];
```

Figure 4.9 A default handler implicitly provided by the compiler

[Par72a], the designer of N is not supposed to know anything about the modules L used by M. Thus, an exception label u, declared for an operation O of a lower level module L is likely to be meaningless to the designer of N and it is probable that there will be no handler explicitly associated with u in N.Q. Second, propagating u from L.O directly into N.Q violates the basic principle that after any procedure invocation from M.P control should return back in the invoking procedure M.P. Indeed, any L.O invocation which results in a propagation of u is a definitive exit from M.P (through an exit point which has not been declared for M.P!). Third, and this is perhaps the most serious consequence, if the lower level procedure L.O was invoked from M.P when M was in an intermediate inconsistent state, then the propagation of u in N.Q leaves M in that inconsistent state. Thus, there is a danger that later invocations of M will lead to unpredictable results and to additional unanticipated exception propagations.

A different approach to the problem of handling detectable failure occurrences is discussed in [Cri79a, Hor74, Lis79]. The basic idea is quite simple: associate a *default handler* DH, with any lower level (unanticipated) exception u propagated in a procedure exported by a module M. The default handler DH is implicitly provided by the compiler (Figure 4.9).

The " " before the ":" symbol stands for any exception which can be propagated in P and which has no explicitly associated handler in P. The exceptional service that such a handler attempts to provide can be identified by a language defined exception label "failure" [Cri79b, Lis79], or "error" [Hor74]. The systematic addition of default handlers to all procedures exported by modules, written in a language in which the "failure" exception is predefined, has the following consequences. For any lower level exception which may be propagated in a procedure $M.P$, there exists an exceptional continuation in M.P (either one explicitly defined or the default continuation DH). A "failure" (or "error") exit point is implicitly added to any procedure exported by a module.

Default exception handlers can be designed to solve the same problems as those mentioned previously for programmed exception handlers. These are (1) *masking*, (2) consistent state *recovery*, and (3) *signalling*. But while the programmer of an explicit handler H, specifically inserted in a specific procedure M.P, knows the intended semantics (captured by I, A, G) of M.P, and, therefore, can provide a specific masking algorithm or determine an inconsistency set to be recovered, this knowledge is not available to the programming language designer who decides on a general default exception handling strategy for *all* programs which will be written in that language.

The default exception handling strategy embodied in the CLU programming language developed at MIT [Lis79] is oriented towards solving problem (3), related to the (proper) propagation of "failure" exceptions across module boundaries, *i.e.*, each default handler is of the form DH $\equiv$ *signal* failure. In CLU, a suitable error message may be passed as a parameter to a *signal* failure sequencer to help in fixing off-line the cause of the failure detection. However, according to terminology introduced in [Mel77, Ran78], tolerance of failure detections

implies at least the resolution of problems (2) and (3). Thus, one can regard the default exception handling strategy of CLU as being more oriented towards off-line debugging rather than towards the provision of on-line software-fault tolerance.

The default exception handling strategy proposed for the SESAME programming language developed at the University of Grenoble [Cri79a] was oriented towards solving the consistent state recovery (2) and propagation (3) problems. (The masking problem (1) can also be solved by using our mechanism, as will be shown later, but we have not dealt with this issue in [Cri79a].) The solution proposed to problem (2) is based on the fact that, for any exception which can be detected in a procedure M.P, there exists a recovery set, *i.e.*, the *inconsistency closure*, which can be determined at run-time without having any knowledge about the semantics of M.P. A recovery cache mechanism (more simple than that of [Hor74] because of the modular scope rules of SESAME) was designed for the automatic update of the inconsistency closures associated with all intermediate states through which a system may pass. A detailed description of this mechanism has already been published [Cri79b], so we will not repeat it here. To enable the automatic recovery of inconsistency closures, a *reset* primitive was made available in the SESAME language (as a compilation option). When invoked, *reset* recovers the "current" IC and returns normally. This primitive is mainly used in default handlers, but is also available to a programmer. (If the exceptional state transition G_e specified for some anticipated exception e is the identity relation, then by inserting a *reset* primitive in the handler of e, the programmer is relieved from the burden of explicitly identifying and restoring some recovery set.) Problem (3) is solved by requiring the propagation of "failure" exceptions to obey the same rules as the propagation of anticipated exceptions. Thus, a DH handler in SESAME is defined as DH $\equiv$ *reset*; *signal* failure.

Default handlers can be inserted only by the complier, *i.e.*, "failure" exceptions cannot be explicitly handled. Programmers can nevertheless explicitly signal "failure" exceptions. This often happens when a Boolean check for an invariant relation, which should be true if the program were correct, is actually found false at run-time. Termination of a program with a "failure" exception is *improper*. Thus, for a language which incorporates the notion of a "failure" exception, one can extend the definition of a partially correct program, given earlier, as follows:

> A program is *partially correct* if, for any possible input, it either terminates properly in a final state satisfying the program specification, or it fails to terminate properly.

It is interesting to note that the default exception handling strategy embodied in SESAME is very similar to the undo-log based strategy used to abort transaction executions that result in unanticipated exception detections or assertion violations [Ber87, Gra93]. Most database systems do not attempt to mask transaction aborts caused by exception detections to users.

The recovery block mechanism, devised at the University of Newcastle upon Tyne [Hor74], was designed to solve all the problems (1)–(3) mentioned above. Unlike the mechanisms described in [Cri79a, Lis79] which support *both* explicit and default exception handling, the recovery block mechanism is a pure default exception handling mechanism based on automatic backward recovery. To deal with a possible "failure" detection (the label "error" is used in [Hor74]) in a procedure P designed to provide some specified standard service $post_\sigma$, a programmer can define P to be the primary block P_0 of a *recovery block* possessing zero or more alternate blocks P_1, P_2, ..., P_k. and an acceptance test *at* that is supposed to check $post_\sigma$. Assume (for simplicity) that a single alternate P_1 is provided. The syntax of a recovery block construct RB in this case is

RB $\equiv$ *ensure at by* P_0 *else by* P_1 *else* failure.

The semantics of the recovery block can be expressed in terms of our exception handling notation as follows:

RB = PP_0[: *reset*; PP_1[: *reset*; *signal* failure]];

where

$PP_i \equiv$ *begin* P_i; [$\neg$ *at*: *signal* failure] *end*

If a "failure" exception is detected during the execution of the P_0 procedure (because some lower level exception is propagated in P_0 or because the acceptance test *at* evaluates to false when P_0 terminates), then the *inconsistency closure* associated with this failure detection is restored by a recovery cache device and the alternate P_1 is invoked. The aim of the alternate is to *mask* the failure detected in PP_0 by achieving the specified state transition $post_\sigma$ in a different way. Since no attempt is made at elucidating the reason why P_0 could not achieve *post*, the construction of an alternate P_1 is based on the sole assumption that, when invoked, P_1 starts in the same state as the primary P_0. If the invocation of P_1 leads to another failure detection, then the masking problem (1) cannot be successfully solved at the level of RB. Problem (2) is solved by invoking again the recovery cache to restore the inconsistency closure associated with the "failure" exception detected in PP_1. Problem (3) is dealt with by propagating a failure signal to the user of RB. The termination of an RB is standard if no failure is detected during the execution of PP_0 or if a failure detection in PP_0 can be masked by the normal termination of PP_1 in a final state in which *at* is true.

The above discussion assumes that a precise monolithic run-time check *at* equivalent to $post_\sigma$ can be programmed. In practice, postconditions usually contain logical quantifiers and other expressions not directly available in a programming language. Thus, to program a Boolean (executable) expression *at* with the same truth value as $post_\sigma$ may turn out to be at least as difficult as programming an alternate. (In [Bes81a], a methodology for splitting such monolithic acceptance checks into sets of simpler assertions without quantifiers spread among the intermediate operations which compose operations like P_0, P_1 is investigated, but pursuing such a verification-oriented approach leads naturally to a programmed, rather than default, exception handling style.) What can happen in practice is that the acceptance test at is an approximation of $post_\sigma$: only some, but not all, invocations of P_0, P_1 in their failure domain will be detected by *at* or by the occurrence of an unanticipated exception at run-time. In such a case, the recovery block program RB, resulting from combining the alternates P_0, P_1 with the acceptance test *at* in the manner described above, will not be partially correct, in the sense that certain invocations of RB in its failure domain will result in proper termination of the RB in an erroneous final state.

4.3.4 Exception Handling in Hierarchies of Data Abstractions

Consider a software system structured into a hierarchy of data abstractions (Figure 4.6). Let $\{C_i\}$ be the set of operations exported by data abstractions visible to system users. These data abstractions, storing information which is significant to the users, are generally implemented by high-level modules. Let us distinguish a C_i operation from other (lower level) hidden operations by calling C_i a system *command*. (If the data abstractions visible to users are

stored in a data base on stable storage, what we call a command would probably corresponds to a database transaction.) A purpose of programmed and default exception handling is to ensure that system command executions preserve the internal invariant properties inherent to the data abstractions which compose the system in spite of possible exception occurrences.

Suppose that the invocation of a command C_i leads to the occurrence of an (anticipated or unanticipated) exception d when some lower level operation L.O is invoked. The operation L.O is said to be *tolerant* to the occurrence of d if d is detected and the (programmed or default) handler of d recovers a consistent state for L before propagating d to the invoking procedure M.P. If this procedure can stop the propagation of d, then M.P is said to be *mask* the occurrence of d. Otherwise, if the propagation of d coincides with the detection of a higher level exception e in M.P, M.P in its turn must be tolerant to e. In general, if an exception propagation d, e, f, ... takes place across modules L, M, N, ... and none of the traversed modules can perform a successful masking, then each module must be tolerant with respect to that propagation (*i.e.*, each must contain programmed or default handlers able to recover a consistent module state and continue the propagation).

Default exception handling based on automatic backward recovery can be used to tolerate or mask the unanticipated exceptions detected during the execution of system commands. After a command execution is terminated, the recovery data maintained by the recovery cache has to be discarded to allow the cache to keep track of the inconsistency closures associated with further potential exception detections during the next command execution.

4.3.5 Tolerance of Design Faults

Assume now that there is a design fault in a procedure M.P. A failure occurrence when P is invoked (in some state within its failure domain) is a *manifestation* of the design fault. Between a manifestation and a detection of the consequences of this manifestation (either by a run-time check or by a human user who observes a discrepancy between the actual and specified behavior of the system containing P), a fault is called *latent*.

A system can be called *design-fault tolerant* if its commands tolerate or mask lower level failure occurrences caused by design faults. As discussed previously default exception handling based on automatic backward recovery can be used to provide design fault tolerance, but the question is: to what extent can one depend on this technique to make tolerable the consequences of human mistakes made during the design (or debugging) of a system?

Let us call the time interval between the beginning and the termination of a command a *command execution interval* and let us call the time elapsed between a manifestation of a design fault and a detection of the consequences of this manifestation a *latency interval*. Suppose that when a command C_i is started, the internal states of the system modules are consistent, and that during the execution of C_i a design fault manifests itself. If this manifestation leads to a failure exception detection before the termination of C_i, then by invoking automatic backward recovery it is possible to restore, for all system modules invoked since the beginning of C_i, internal states which are equivalent to those which existed at the beginning of C_i. These recovered internal states are then consistent, and the danger of later additional unanticipated exception detections is avoided.

However, it is possible that the manifestation of a design fault does not cause some explicitly checked assertion to be violated, so that no failure exception is detected during the execution of the C_i command. In such a case, when C_i terminates some of the component modules of the system can be in an inconsistent state. It is then possible that a failure ex-

ception caused by the design fault which has manifested itself during the execution of C_i is detected during some later command execution C_j. The invocation of automatic backward recovery will then restore internal module states which are equivalent to those which existed at the beginning of C_j. But since these states were already inconsistent, the recovered system state will be inconsistent and the danger of further unpredictable behavior and additional unanticipated exception detections persists.

Thus, while default exception handling based on automatic backward recovery *guarantees* tolerance of design faults with latency intervals contained within command execution intervals, it is *not adequate* for coping with design faults having latency intervals which stretch over successive command executions. In other terms: in a system where the user visible commands are implemented by using recovery blocks (or database transactions), backward recovery based default exception handling guarantees proper behavior despite design faults only if the outer-most recovery blocks (or database transactions) are *partially correct*. Clearly, the use of automatic backward recovery improves the chance that crucial data will remain consistent in the presence of failure detections, since it provides tolerance for all confined design faults. Experimental studies confirm this [And85]. However, to acquire confidence that a recovery block is capable of tolerating *all* design faults that might be contained in its alternates and acceptance test is in fact as hard as proving that these alternates together with the acceptance test are partially correct.

4.4 CONCLUSIONS

This chapter gives mathematically rigorous definitions for notions basic to the design of dependable software such as specification, program semantics, exception, program correctness and robustness. It also defines with precision concepts fundamental to fault-tolerant computing, such as program failure, program design fault, and error. To define precisely these often used — but rarely defined — terms, we introduced a number of other concepts that are useful for future discussions of software-fault tolerance issues, such as standard domain, anticipated exceptional domain, failure domain, and unanticipated input domain.

The notion of an exception is defined in terms of the set of possible input states of a program and the standard specification for that program, to mean "impossibility of obtaining the specified standard service". It therefore depends on how the states of a program are defined and how the standard service of that program is specified. If the probability of invoking the program in its standard domain is in general greater than that of invoking it in its exceptional domain, this definition is consistent with the probabilistic point of view adopted in [Che86]. However, unlike in [Che86], we view exceptions purely as a specification and program *structuring tool*, and we refrain from discussing the criteria to be used when deciding on how to use exceptions to structure programs. Our definitions are general precisely because they are independent from any such criteria. Probability of successfully completing a state transition is one such criterion. This criterion might be useful when a great deal of statistical information about program inputs is available. Often, at the beginning of a design such information does not exist, and other criteria for partitioning the input domains of programs into subdomains must be adopted.

Exception occurrences can result in delivery of specified exceptional services (when anticipated) or in the delivery of unspecified results or program failures (when unanticipated). While anticipated exceptional program responses share with failures the characteristic "impossibility

of obtaining the requested standard service", they also share with correct standard program responses the characteristic "the program behaves as specified". Exceptions can therefore be viewed as being a software structuring concept that helps bridge the conceptual gap which exists between behaviors as opposite as "correct standard service provided" at one extreme, and "program failure" at the other extreme.

The notions defined in Section 4.2 of this chapter are central to many programming related areas such as testing, stochastic reliability estimation, program verification, and design-fault tolerant programming. Testing attempts to hit the failure domain of a program with test data to reveal design faults. Often testing helps discover possible inputs in the unanticipated input domain of a program. Stochastic software reliability estimation methods attempt to predict the "size" of the failure domain, given the estimated sizes of the failure domains of successive program versions during a testing period. Program verification, like testing, aims at discovering program design faults, if they exist. It differs from testing in that it also attempts at proving the absence of such faults, if they do not exist. Design-fault tolerant programming techniques start from the premise that the failure domains associated with program designs are never empty, and attempt to mask component program failures by relying on the use of design diversity [Avi84, Hor74]. The intention is to construct several program versions for a single specification so that the failure domain of the resulting multi-version program is smaller than the failure domains of the individual program versions used. Empirical investigations of the likelihood of this goal being achieved for actual programs can be found in [And85, Eck91].

Section 4.3 of this chapter investigates *what is* exception handling in programs structured as hierarchies of data abstractions. The answer proposed is a simple one. At each level of abstraction, exception handling consists of: detection, attempt at masking, consistent state recovery, and propagation. Several problems posed by default exception handling in programming languages which support data abstraction (such as Ada) are. Finally, an assessment of the adequacy of automatic backward recovery based default exception handling (such as embodied in recovery blocks [Hor74] or database transactions [Ber87, Gra93]) in providing design-fault tolerance was provided: automatic backward recovery guarantees tolerance of design faults only in partially correct programs.

REFERENCES

[And81] T. Anderson and P. A. Lee. *Fault-Tolerance: Principles and Practice*. Prentice Hall, December 1981.

[And85] T. Anderson, P. A. Barrett, D. N. Hallivell, and M. R. Moulding. An evaluation of software fault tolerance in a practical system. In *Proc. 15th International Symposium on Fault-Tolerant Computing*, pages 140–145, Ann Arbor, Michigan, 1985.

[Avi84] A. Avižienis and J. P. Kelly. Fault-tolerance by design diversity. *IEEE Computer*, 17(8):67–80, 1984.

[Bac79] R. J. R. Back. Exception Handling with Multi Exit Statements. Technical report IW125, Math. Cent. Amsterdam, November 1979.

[Ber87] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, February 1987.

[Bes81a] E. Best and F. Cristian. Systematic detection of exception occurrences. In *Science of Computer Programming,* 1(1):115–144, 1981.

[Bes81b] E. Best and B. Randell. A formal model of atomicity in asynchronous systems. In *Acta Informatica,* 16:93–124, 1981.

[Bro76] C. Bron, M. M. Fokkinga, and A. C. M. de Haas. A Proposal for Dealing with Abnormal Termination of Programs. Memorandum Nr. 150, Department of Applied Mathematics, Twente University of Technology, Netherlands, 1976.

[Cam86] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering* SE-12(8):811–826, 1986.

[Che86] D. Cheriton. Making exceptions simplify the rule and justify their handling. In *Proc. IFIP Congress 86*, pages 27–33, 1986.

[Cri79a] F. Cristian. *Le Traitement des Exceptions dans les Programmes Modulaires*. PhD Dissertation, University of Grenoble, Grenoble, France, 1979.

[Cri79b] F. Cristian. A recovery mechanism for modular software. In *Proc. 4th International Conference on Software Engineering*, Munich, Germany, 1979.

[Cri80] F. Cristian. Exception handling and software fault-tolerance. In *Proc. 10th International Symposium on Fault-Tolerant Computing*, pages 97–103, 1980, Kyoto, Japan; also in *IEEE Transactions on Computers*, C-31(6):531–540, 1982.

[Cri82] F. Cristian. Robust data types. *Acta Informatica*, 17:365–397, 1982.

[Cri84] F. Cristian. Correct and robust programs. *IEEE Transactions on Software Engineering*, SE-10(2):163–174, 1984.

[Cri85] F. Cristian. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering*, SE-11(1):23–31, 1985.

[Cri91] F. Cristian. Understanding fault-tolerant systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[Dij76] E. W. Dikstra. *A Discipline of Programming*, Prentice Hall, 1976.

[Eck91] D. Eckhardt, A. Caglayan, J. Knight, L. Lee, D. McAllister, M. Vouk, and J. P. J. Kelly. An experimental evaluation of software redundancy as a strategy for improving reliability. *IEEE Transactions on Software Engineering*, 17(7):692–701, July 1991.

[Flo67] R. Floyd. Assigning meaning to programs. in *Mathematical Aspects of Computer Science.*, 19:19–31, 1967. American Mathematical Society.

[Goo75] J. Goodenough. Exception handling, issues and a proposed notation. *Communications ACM*, 18(12):683–696, 1975.

[Gra93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[Hoa69] C. A. R. Hoare. An axiomatic approach to computer programming. *Communications ACM*, 12(10):576–580, 1969.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[Hor74] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Lecture Notes in Computer Science*, volume 16, Springer-Verlag, New York, 1974.

[Hor78] J. J. Horning. Language features for fault-tolerance. In *Lecture Notes, Advanced Course on Computing Systems Reliability*, University of Newcastle upon Tyne, August, 1978.

[Ich79] J. Ichbiah *et al.*. Rationale for the design of the ADA programming language. In *SIGPLAN Notices*, 14(6), 1979.

[Jal84] P. Jalote and R. H. Campbell. Fault-tolerance using communicating sequential processes. In Proc. *14th International Conference on Fault-Tolerant Computing*, pages 347–352, 1984.

[Kim82] K. H. Kim. Approaches to mechanization of the conversation scheme based on monitors. *IEEE Transactions on Software Engineering*, SE-8:189–197, May, 1982.

[Lam74] B. Lampson, J. Mitchell, and E. Satterthwite. On the transfer of control between contexts. In *Lecture Notes in Computer Science*, 19:181–203, 1974.

[Lev77] R. Levin. *Program Structures for Exceptional Condition Handling*, PhD Dissertation, Carnegie-Mellon University, 1977.

[Lev85] R. Levin, P. Rovner, and J. Wick. On extending Modula-2 for building large integrated systems. (B. Lampson is acknowledged as making major design contributions to this extension.) DEC Systems Research Center Technical Report number 3, January 11, 1985.

[Lis74] B. H. Liskov and S. Zilles. Programming with abstract data types. In *Proc. ACM SIGPLAN Conference on Very High Level Languages*, SIGLAN Notices, 9(4):50–59, 1974.

[Lis79] B. H. Liskov and A. Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, SE-5:546–558, 1979.

[Lis82] B. H. Liskov. On linguistic support for distributed programs. *IEEE Transactions on Software Engineering*, SE-8:203–210, 1982.

[Luc80] D. Luckham and W. Polak. ADA exception handling: an axiomatic approach. *ACM TOPLAS*, volume 2, 1980.

[Mel77] M. Melliar-Smith and B. Randell. Software reliability: the role of programmed exception handling. In *Proc. ACM Conference on Lang. Design for Reliable Software*; also in *SIGPLAN Notices*, 12:95–100, 1977.

[Mit79] J. Mitchell *et al.* Mesa Language Manual. Report CSL-79-3, Xerox PARC, Palo Alto, California, 1979.

[Mit93] J. Mitchell. Private Communication, 1993.

[Par72a] D. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.

[Par72b] D. Parnas. Response to Detected Errors in Well-Structured Programs. Technical report, Carnegie-Mellon University, Dept. of Computer Science, 1972.

[Par74] D. Parnas. On a buzzword: hierarchical structure. In *Proc. IFIP Congress 1974*, North Holland Publication Company, 1974.

[Par85] D. Parnas, Private Communication, 1985.

[Ran75] B. Randell. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, SE-1(2), 1975.

[Ran78] B. Randell, P. A. Lee, and P. C. Treleaven. Reliability issues in computing systems design. *Computing Surveys*, 10(2):123–165, 1978.

[Sch89] R. Schlichting, F. Cristian and T. Purdin. Mechanisms for failure handling in distributed programming Languages. In *Proc. 1st International Working Conference on Dependable Computing for Critical Applications*, Santa Barbara, California, 1989.

[Shr78] S. K. Shrivastava and J. P. Banatre. Reliable resource allocation between unreliable processes. *IEEE Transactions on Software Engineering*, SE-4:230–241, May, 1978.

[Sta87] M. E. Staknis. *A Theoretical Basis for Software Fault Tolerance*. PhD Thesis, University of Virginia, Charlottesville, CS Report RM-87-01, February 26, 1987.

[Toy82] W. N. Toy. Fault-tolerant design of local ESS processors. In *The Theory and Practice of Reliable System Design*, D. P. Siewiorek and R. S. Swarz, Eds., Digital Press, 1982.

[Woo81] W. G. Wood. A decentralized recovery control protocol. In *Proc. 11th International Conference on Fault-tolerant Computing*, pages 159–164, 1981.

[Wul75] W. Wulf. Reliable hardware-software architecture. In *Proc. International Conference on Reliable Software*, SIGPLAN Notices, 10(6):122–130, 1975.

[Wul76] W. Wulf, R. London and M. Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Transactions on Software Engineering*, SE-2:253–265, July 1976.

[Yem82] S. Yemini. An axiomatic treatment of exception handling. In *Proc. 7th ACM Symposium on Principles of Programming Languages*, 1982.