Exceptions for Dependability[1]

Emil Sekerinski
McMaster University, Canada

ABSTRACT

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

INTRODUCTION

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

- The specification may be in error: the specification may not be according to the user's requirements or the requirements may have been inconsistent or incomplete.
- There may be errors in the design: there can errors in the correctness argument of the program, like overlooking a case; the design may be based on idealized hypotheses about the programming language and hardware, like a sufficiently large integer range and sufficiently available memory; the design may rely on incorrect assumptions about the run-time system and libraries.
- There can be failures of the underlying software or hardware, like an error in the operating system, disk errors, memory errors, and network failures.

Some failures are always *detected* at run-time by the underlying virtual machine, like indexing an array out of bounds, allocating memory when none is available, or reading a file beyond its end. Some failures can be detected by programmer-added checks, like checking the range of parameters of a procedure, while some failures would be too difficult to detect by any means.

Even with our best efforts to design error-free programs, in the design of any reasonably complex system, there always remains the possibility of a failure. The question then arises how programs should respond to *detected* failures. Suppose that a problem calls for the sequential composition of four statements,

S_1 ; S_2 ; S_3 ; S_4

and statements S_1 and S_3 may fail in a detectable way; in case they fail, the sequence should be abandoned and statement T should be executed instead. In the *a priory scheme*, we add a test before running S_1 and S_3. In the *a posteriori scheme*, we run S_1 and S_3 and test if they were successful:

```
if S1 possible then
    S1 ;
    S2 ;
    if S3 possible then
        S3 ;
        S4
    else
```

```
S1 ;
if S1 successful then
    S2 ;
    S3 ;
    if S3 successful then
        S4
    else
```

[1] Preprint, to appear in *Dependability and Computer Engineering: Concepts for Software-Intensive Systems—a Handbook on Dependability Research*, L. Petre, K. Sere, and E. Troubitsyna, Eds. IGI Global, 2011.

```
        T                                   T
else                                  else
    T                                     T
```

Both schemes are unsatisfactory. Explicit tests clutter up the program with additional variables, parameters, and tests that have to be repeated at different levels of the program structure or complicate the structure, for example if the failing statements are nested inside repetitions and conditionals. The treatment of possible failures dominates the program structure: *the original design is no longer visible*. The solution is a control structure for *exception handling*:

```
try
    S1 ; S2 ; S3 ; S4
catch
    T
```

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

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

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

The purpose of this chapter is to develop a theory of exception handling with try-catch statements and to show its use in the design of dependable systems. To this end, we give a formal account of the patterns of *masking*, *propagating*, *flagging*, *rollback*, *degraded service*, *recovery block*, *repeated attempts*, and *conditional retry*. The theory is based on weakest exceptional preconditions, a generalization of Dijkstra's weakest precondition predicate transformers. To represent programming languages realistically, expressions may be partially defined and may evaluate conditionally. Proof outlines are introduced and used to establish the correctness of the patterns.

BACKGROUND

A treatment of exception handling with predicate transformers is given by (Cristian, 1984): statement have one entry and multiple exits (one of those being the normal one) and are defined by a set of predicate transformers, one for each exit. As (King & Morgan, 1995) point out, this disallows nondeterminism, which precludes the use of the language for specification and design; the solution is to use a single predicate transformer with one postcondition for each exit instead, and which we follow here.

A mechanical formalization of try-catch-finally statements is given by (Jacobs, 2001). However, that formalization includes all the other "abrupt termination" modes of Java, which we do not need here, and uses state transformers, which precludes nondeterminism, and thus is less suited for our needs.

While there is a general agreement that exception handling is needed if programs are to be *correct* and *robust*, there has been a lively debate how exactly exceptions should be used. One view, exemplified by (Liskov & Guttag, 2000), is that exceptions provide an alternative return of procedures (like "item not found"), and as such have their place in interface specifications, together with the condition when they are raised and what the outcome in that case is. The other view, argued by (Meyer, 1997), is that exceptions are only for recovery in case a contract–given by a precondition and a postcondition–is broken. As the interface of a procedure consist of a single pre- and postcondition, this defines at the same time the circumstances when an exception is raised. In this chapter, we focus on exceptions at the level of statements, where issues of interfaces do not occur, but note that the theory by itself does not preclude either view.

For the interested reader we mention that a classification of exception handling mechanisms is given by (Buhr & Mok, 2000), while (Garcia et al., 2001) compare mechanisms specifically in object-oriented languages. A discussion of exception handling vs. the use of return values in operating systems and of failures in common operating systems, is given by (Koopman & DeVale, 2000). The issues of the correct implementation of exception handling is studied by (Börger & Schulte, 2000) and (Wright, 2005), based on an operational semantics of programs; we will not go further into the issue of correct implementation.

The programming language and the treatment of partial expressions in this chapter are inspired by (Hoare et al., 1987). In (Back & Wright, 1998) algebraic properties of single-entry single-exit statements are studied in depth for loops and by considering both angelic and demonic nondeterminism; here we restrict to demonic nondeterminism. In (Leino & Snepscheut, 1994) weakest exceptional preconditions of statements are derived from a trace semantics. Here we start with weakest exceptional preconditions.

UNDEFINEDNESS IN EXPRESSIONS

Expressions may not be defined for some values of its variables; for example, x div y is not defined for $y = 0$. Evaluating such an expression in a program does not return a result, but should–at least for robust programs–instead cause an exception. For expressions E that can appear in programs, we let ΔE be a predicate that is true when E is defined and false otherwise. We do *not* specify what the value of E is if it is undefined, that is we don't extend the range of integers with an "undefined value". Instead, we think of it as being any value, which just happens to be in the register that holds the result, but we have flagged that the result is in error. Likewise, we don't extend booleans or any other type with undefined values. As we define predicates to be boolean expressions, this avoids a "three-valued logic". We write $F(E)$ for function application and $F(E \leftarrow G)$ for modifying function F to be G at E. Functions can be partial and dom F stands for the domain of F; an array of length N is partial function with domain $[0, N)$.

Definition (Δ *for constants, variables, operators*). Let c be a constant, x a variable, E, F, G be expressions of appropriate type:

Δc	$\equiv$ true	$\Delta(E + F)$	$\equiv \Delta E \wedge \Delta F$
Δx	$\equiv$ true	$\Delta(E - F)$	$\equiv \Delta E \wedge \Delta F$
$\Delta F(E)$	$\equiv \Delta E \wedge \Delta F \wedge E \in$ dom F	$\Delta(E \times F)$	$\equiv \Delta E \wedge \Delta F$
$\Delta F(E \leftarrow G)$	$\equiv \Delta E \wedge \Delta F \wedge \Delta G \wedge E \in$ dom F	$\Delta(E$ div $F)$	$\equiv \Delta E \wedge \Delta F \wedge (F \neq 0)$
$\Delta -E$	$\equiv \Delta E$	$\Delta(E$ mod $F)$	$\equiv \Delta E \wedge \Delta F \wedge (F \neq 0)$
$\Delta \neg E$	$\equiv \Delta E$	$\Delta(E$ and $F)$	$\equiv \Delta E \wedge (E \Rightarrow \Delta F)$
$\Delta(E = F)$	$\equiv \Delta E \wedge \Delta F$	$\Delta(E$ or $F)$	$\equiv \Delta E \wedge (\neg E \Rightarrow \Delta F)$

In the definition of Δ for arithmetic expression, we have assumed unbounded arithmetic; if needed, this can be strengthened by requiring that the result is between the maximal and minimal representable values, for example $\Delta(E + F) \equiv \Delta E \wedge \Delta F \wedge$ minint $\leq E + F \leq$ maxint. Equality, written $=$, is interpreted as being *strict*, meaning that both operands need to be defined for the equality to be defined, like with the binary

arithmetic operators. Equivalence, written ≡, is an operator that we use to reason about expressions, rather than an operator that appears in programs. An equivalence is always true or false. For example, $E \equiv E$ is always true, e.g. $1/0 \equiv 1/0$, but the truth of $1/0 = 1/0$ is irrelevant, as it is not defined. The boolean operators and, or, also written as cand, cor and as and then, or else are being interpreted as *conditional* operators, meaning that their second operand does not need to be defined if the first one determines the result. We do not define Δ for further operators, as we will not need them, but only state following properties for transforming conditional boolean operators to *standard* operators:

Property (deMorgan, eliminating and, eliminating or). Let P, Q be predicates.

$\neg(P \text{ and } Q) \equiv \neg P \text{ or } \neg Q$ $\qquad$ $\Delta P \Rightarrow (P \text{ and } Q \equiv P \wedge Q)$

$\neg(P \text{ or } Q) \equiv \neg P \text{ and } \neg Q$ $\qquad$ $\Delta P \Rightarrow (P \text{ or } Q \equiv P \vee Q)$

Applying Δ to a list of expressions denotes the conjunction of Δ applied to each element of the list. For example, if $\underline{E} = E_1, E_2$ then $\Delta\underline{E} = \Delta E_1 \wedge \Delta E_2$.

WEAKEST PRECONDITIONS

We briefly review standard weakest preconditions with undefined expressions. The statements that we consider are abort, which does not guarantee any particular outcome, stop, which blocks further execution, skip, which does nothing, *multiple assignment* $\underline{x} := \underline{E}$, which assigns the values of the list $\underline{E}$ of expressions to the variables of list $\underline{x}$, *nondeterministic multiple assignment* $\underline{x} :\in \underline{E}$, which assigns any values of the sets $\underline{E}$ to the variables $\underline{x}$, *sequential composition* S ; T of statements S and T, *nondeterministic choice* $S \sqcap T$, *conditional* if B then S else T with *condition* B and *branches* S and T, and *repetition* while B do S with *condition* B and *body* S. Follwing Dijkstra, we introduce:

wp(S, Q) ≡ weakest precondition such that S terminates with postcondition Q

Definition (wp *for straight-line statements*). Let B, Q be predicates, $\underline{x}$ a list of variables, $\underline{E}$ a list of expressions, and S, T statements:

wp(abort, Q)	≡	false
wp(stop, Q)	≡	true
wp(skip, Q)	≡	Q
wp($\underline{x} := \underline{E}$, Q)	≡	$\Delta\underline{E} \wedge Q[\underline{x} \setminus \underline{E}]$
wp($\underline{x} :\in \underline{E}$, Q)	≡	$\Delta\underline{E} \wedge (\forall\, \underline{x}' \in \underline{E} \,.\, Q[\underline{x} \setminus \underline{x}'])$
wp(S ; T, Q)	≡	wp(S, wp(T, Q))
wp($S \sqcap T$, Q)	≡	wp(S, Q) ∧ wp(T, Q)
wp(if B then S else T, Q)	≡	$\Delta B \wedge (B \Rightarrow \text{wp}(S, Q)) \wedge (\neg B \Rightarrow \text{wp}(T, Q))$

In general, $F[\underline{x} \setminus \underline{E}]$ stands for expression F with variables $\underline{x}$ simultaneously replaced by expressions $\underline{E}$. For the simple assignment $x := E$ to terminate with Q, expression E must be defined and $Q[x \setminus E]$ must hold initially. For if B then S else T to terminate with Q, predicate B must be defined and if B holds, S must terminate with Q, otherwise T must terminate with Q. We don't define the weakest precondition for repetitions, but instead give the fundamental rule for repetitions.

Rule for Repetition. Let P be a predicate, the *invariant*, V be an integer expression, the *variant*, and v be an auxiliary integer variable. If

$\Delta B \wedge B \wedge P \wedge (V = v) \Rightarrow \text{wp}(S, \Delta B \wedge P \wedge V < v)$ $\qquad$ (S preserves P, decreases V)

$\Delta B \wedge B \wedge P \Rightarrow V > 0$ $\qquad$ ($V \leq 0$ leads to termination)

then

$\Delta B \wedge P \Rightarrow \text{wp}(\text{while } B \text{ do } S, \Delta B \wedge \neg B \wedge P)$

An alternative formulation, which is methodologically stronger, is obtained if we add to the assumption that whenever the invariant holds, the condition of the repetition must be defined, formally $P \Rightarrow \Delta B$. The alternative rule then follows immediately from above rule.

Alternative Rule for Repetition. Let P be a predicate, the *invariant*, V be an integer expression, the *variant*, and v be an auxiliary integer variable. If

$B \wedge P \wedge (V = v) \Rightarrow \mathsf{wp}(S, P \wedge V < v)$ (S preserves P, decreases V)
$B \wedge P \Rightarrow V > 0$ ($V \leq 0$ leads to termination)
$P \Rightarrow \Delta B$ (B is defined)

then

$P \Rightarrow \mathsf{wp}(\text{while } B \text{ do } S, \neg B \wedge P)$

WEAKEST EXCEPTIONAL PRECONDITIONS

We consider now statements that can *terminate normally*, or *succeed* for short, and can *terminate exceptionally*, or *fail* for short. That is, statements have a single entry and two exits.The abort statement either succeeds in an arbitrary state, fails in an arbitrary state, or does not terminate at all. As previously, the stop statement blocks further execution. The skip statement does nothing and succeeds, while the raise statement does nothing and fails. The multiple assignment $\underline{x} := \underline{E}$ succeeds by assigning the values of $\underline{E}$ to $\underline{x}$ if all expressions of $\underline{E}$ are defined, otherwise it fails and does not change the values of the variables. The nondeterministic multiple assignment $\underline{x} :\in \underline{E}$ is analogously. The sequential composition S ; T starts with S and fails if S fails, or continues with T if S succeeds, in which case it either fails if T fails, or succeeds if T succeeds. The statement try S catch T starts with the *body* S and succeeds if S succeeds, or continues with the *handler* T if S fails, in which case it either succeeds if T succeeds, or fails if T fails. For the nondeterministic choice $S \sqcap T$ to succeeds for sure, both S and T must succeed, and for it to fail for sure, both S and T must fail. The conditional if B then S else T succeeds if B is defined and S succeeds if B is true or T succeeds if B is false, and fails in all other cases.

The weakest exceptional precondition $\mathsf{wp}(S, Q, R)$, sometimes written as $\mathsf{wep}(S, Q, R)$, specifies two postconditions Q and R, one for normal termination and one for exceptional termination.

$\mathsf{wp}(S, Q, R) \equiv$ weakest precondition such that S terminates and
- on normal termination Q holds finally
- on exceptional termination R holds finally

Definition (wp *for straight-line statements*). Let B, Q, R be predicates, $\underline{x}$ a list of variables, $\underline{E}$ a list of expressions, and S, T statements:

$\mathsf{wp}(\text{abort}, Q, R) \equiv \text{false}$
$\mathsf{wp}(\text{stop}, Q, R) \equiv \text{true}$
$\mathsf{wp}(\text{skip}, Q, R) \equiv Q$
$\mathsf{wp}(\text{raise}, Q, R) \equiv R$
$\mathsf{wp}(\underline{x} := \underline{E}, Q, R) \equiv (\Delta\underline{E} \Rightarrow Q[\underline{x} \setminus \underline{E}]) \wedge (\neg\Delta\underline{E} \Rightarrow R)$
$\mathsf{wp}(\underline{x} :\in \underline{E}, Q, R) \equiv (\Delta\underline{E} \Rightarrow \forall\, \underline{x}' \in \underline{E} \,.\, Q[\underline{x} \setminus \underline{x}']) \wedge (\neg\Delta\underline{E} \Rightarrow R)$
$\mathsf{wp}(S\,;\,T, Q, R) \equiv \mathsf{wp}(S, \mathsf{wp}(T, Q, R), R)$
$\mathsf{wp}(\text{try } S \text{ catch } T, Q, R) \equiv \mathsf{wp}(S, Q, \mathsf{wp}(T, Q, R))$
$\mathsf{wp}(S \sqcap T, Q, R) \equiv \mathsf{wp}(S, Q, R) \wedge \mathsf{wp}(T, Q, R)$
$\mathsf{wp}(\text{if } B \text{ then } S \text{ else } T, Q, R) \equiv (\Delta B \wedge B \Rightarrow \mathsf{wp}(S, Q, R)) \wedge (\Delta B \wedge \neg B \Rightarrow \mathsf{wp}(T, Q, R)) \wedge (\neg\Delta B \Rightarrow R)$

The definition shows a duality between skip and raise and between S ; T and try S catch T, which we will exploit in proofs. We may be tempted to introduce a notation that emphasizes this duality, but refrain from doing so because of familiarity with the used notation and, more importantly, because of methodological reasons, as the use of raise and try-catch is reserved for rare and undesired cases.

The fundamental rule for repetitions does not require that initially the condition is defined, as in case it isn't, the repetition terminates exceptionally.

Rule for Repetition. Let P be a predicate, the *invariant*, V be an integer expression, the *variant*, and v be an auxiliary integer variable. If

$\Delta B \wedge B \wedge P \wedge (V = v) \Rightarrow \mathsf{wp}(S, P \wedge V < v, R)$ (S preserves P, decreases V, or fails with R)
$\Delta B \wedge B \wedge P \Rightarrow V > 0$ ($V \leq 0$ leads to termination)
$\neg\Delta B \wedge P \Rightarrow R$ ($\neg\Delta B$ leads to R)

then:

$P \Rightarrow \mathsf{wp}(\text{while } B \text{ do } S, \Delta B \wedge \neg B \wedge P, R)$

As a note, if we require the repetition to terminate normally, i.e. take R to be false, then from the third assumption we get $P \Rightarrow \Delta B$ and a generalization of the alternative rule for repetition under "Weakest Preconditions" follows.

A statement that neither contains raise nor try-catch statements, can be equivalently verified through weakest preconditions or weakest exceptional preconditions. Thus we can switch to the simpler weakest precondition formalism.

Theorem (*Reduction*). Let S be a straight-line statement that neither contains raise nor try-catch statements and let Q be a predicate:

$\mathsf{wp}(S, Q) \equiv \mathsf{wp}(S, Q, \mathsf{false})$

The proof of this theorem is straightforward by induction over the structure of statements. As it is rather long, it is left out, as are the proofs of the remaining theorems in this section. If a statement establishes normal postcondition Q and establishes normal postcondition Q', then we have that the statement establishes $Q \wedge Q'$. This is known as (finite) *conjunctivity* and holds for both normal and exceptional postconditions.

Theorem (*Conjunctivity*). Let S be a straight-line statement and let Q, Q', R, R' be predicates:

$wp(S, Q, R) \wedge wp(S, Q', R') = wp(S, Q \wedge Q', R \wedge R')$

The weakest precondition function is monotonic in both the normal and exceptional postcondition. That is, weakening either postcondition will weaken the precondition.

Theorem (*Monotonicity*). Let S be a straight-line statement S and let Q, Q', R, R' be predicates:

if $Q \Rightarrow Q'$ and $R \Rightarrow R'$ then $\mathsf{wp}(S, Q, R) \Rightarrow \mathsf{wp}(S, Q', R')$

Reasoning about the normal and exceptional cases can be separated.

Theorem (*Separation*). Let S be a straight-line statement and let Q, R be predicates:

$\mathsf{wp}(S, \mathsf{true}, R) \wedge \mathsf{wp}(S, Q, \mathsf{true}) = \mathsf{wp}(S, Q, R)$

We might be tempted to conclude from the separation theorem that we can equivalently develop the theory by considering a pair of predicate transformers, say, wpn(S, Q) for normal termination and wpe(S, R) for exceptional termination, as done by (Cristian, 1984). However, (King & Morgan, 1995) argue that having a set of predicate transformers does not allow for nondeterministic choice between different exits to be expressed. Furthermore, these functions are not independent. For example, if we defined wpn(X, Q) = true and wpe(X, R) = false, then X blocks, i.e. is stop, and is abort at the same time. While common

language constructs do not exhibit this anomaly, we prefer to use a single predicate transformer to guarantee consistency. More precisely, the separation theorem states that normal and exceptional reasoning can be separated only when the statement terminates.

A statement *S preserves* a predicate *I* if, provided that *I* holds initially, after termination of *S*, *I* holds again, whether *S* terminates normally or exceptionally. If *S* does not terminate, *S* preserves *I* vacuously. We formalize this in the following way.

Definition (*Preservation*). Statement *S* preserves predicate *I* if for any predicates *P*, *Q*, *R*:
if $P \Rightarrow \mathsf{wp}(S, Q, R)$ then $P \wedge I \Rightarrow \mathsf{wp}(S, Q \wedge I, R \wedge I)$

Theorem (*Preservation by disjointness*). Let *S* be a straight-line statement and let *I* be a predicate that does not contain any variables that are assigned in *S*. Then *S* preserves *I*.

As a note, if we were to add *angelic choice* to the core statements in addition to the *demonic choice* considered above, as elaborated in (Back & Wright, 1998), by
$\mathsf{wp}(S \sqcup T, Q, R) \equiv \mathsf{wp}(S, Q, R) \vee \mathsf{wp}(T, Q, R)$
then conjunctivity would have to be weakened to *sub-conjunctivity*:
$\mathsf{wp}(S, Q, R) \wedge \mathsf{wp}(S, Q', R') \Leftarrow \mathsf{wp}(S, Q \wedge Q', R \wedge R')$
Separation would also have to be weakened, we would only have *sub-separation*, which follows immediately from sub-conjunctivity:
$\mathsf{wp}(S, \mathsf{true}, R) \wedge \mathsf{wp}(S, Q, \mathsf{true}) \Leftarrow \mathsf{wp}(S, Q, R)$
However, this invalidates reasoning separately about the normal and exceptional cases even if the statement terminates, why we do not consider angelic nondeterminism further.

DERIVED STATEMENTS

We extend the language of statements by statements that are defined in terms of the core language. The update $a(E) := F$ modifies function *a* to be *F* at *E*, the conditional if *B* then *S* with a single branch does nothing if *B* does not hold, the statement assert *B* fails if *B* does not hold and otherwise does nothing, the statement try *S* finally *U* executes *S* and then *U*, whether *S* succeeds or fails, and the statement try *S* catch *T* finally *U* is like try *S* catch *T* but additionally executes *U* whether *S* succeeds, *S* fails and *T* succeeds, or *S* and *T* fail. Here, *U* is called the *finalization* of the try-catch-finally statement.

Definition (*update*, if-then, assert, finally). Let *S*, *T*, *U* be statements, *E*, *F* expressions, *B* a predicate, and *a* a partial function variable.

$a(E) := F$	=	$a := a(E \leftarrow F)$
if *B* then *S*	=	if *B* then *S* else skip
assert *B*	=	if $\neg B$ then raise
try *S* finally *U*	=	try *S* catch (*U* ; raise) ; *U*
try *S* catch *T* finally *U*	=	try *S* catch try *T* catch (*U* ; raise) ; *U*

Theorem (wp *of update*, if-then, assert, finally). Let *S*, *T*, *U* be statements, *E*, *F* expressions, *B* a predicate, and *a* a partial function variable:

$\mathsf{wp}(a(E) := F, Q, R)$	$\equiv$	$(\Delta E \wedge \Delta F \wedge E \in \mathsf{dom}\ a \Rightarrow Q[a \setminus a(E \leftarrow F)]) \wedge (\neg\Delta E \Rightarrow R) \wedge (\neg\Delta F \Rightarrow R) \wedge (E \notin \mathsf{dom}\ a \Rightarrow R)$
$\mathsf{wp}(\text{if } B \text{ then } S, Q, R)$	$\equiv$	$(\Delta B \wedge B \Rightarrow \mathsf{wp}(S, Q, R)) \wedge (\Delta B \wedge \neg B \Rightarrow Q) \wedge (\neg\Delta B \Rightarrow R)$
$\mathsf{wp}(\text{assert } B, Q, R)$	$\equiv$	$(\Delta B \wedge B \Rightarrow Q) \wedge (\neg\Delta B \Rightarrow R) \wedge (\neg B \Rightarrow R)$
$\mathsf{wp}(\text{try } S \text{ finally } U, Q, R)$	$\equiv$	$\mathsf{wp}(S, \mathsf{wp}(U, Q, R), \mathsf{wp}(U, R, R))$
$\mathsf{wp}(\text{try } S \text{ catch } T \text{ finally } U, Q, R)$	$\equiv$	$\mathsf{wp}(S, \mathsf{wp}(U, Q, R), \mathsf{wp}(T, \mathsf{wp}(U, Q, R), \mathsf{wp}(U, R, R)))$

CORRECTNESS ASSERTIONS

Hoare's correctness assertion $\{P\}\ S\ \{Q\}$ for *total correctness* means that under precondition P, statement S terminates with postcondition Q. This is now generalized to two postconditions, the normal and exceptional postcondition. Correctness assertions are closely related to weakest preconditions and we will switch between them: we use correctness assertions for outlining the program and proof structure and use weakest preconditions for "mechanical" proofs. We introduce:

$\{P\}\ S\ \{Q, R\}$ $\equiv$ under precondition P, statement S terminates and

- on normal termination Q holds finally
- on exceptional termination R holds finally

Definition (*Correctness assertion*). Let S be a statement and P, Q, R predicates:

$$\{P\}\ S\ \{Q, R\} \equiv P \Rightarrow \mathsf{wp}(S, Q, R)$$
$$\{P\}\ S\ \{Q\} \equiv P \Rightarrow \mathsf{wp}(S, Q, \mathsf{false})$$

Theorem (*Fundamental rules of correctness for straight-line statements*). Let B, Q, R be predicates, $\underline{x}$ a list of variables, $\underline{E}$ a list of expressions, and S, T statements:

$$\{P\}\ \mathsf{skip}\ \{Q, R\} \equiv P \Rightarrow Q$$
$$\{P\}\ \mathsf{raise}\ \{Q, R\} \equiv P \Rightarrow R$$
$$\{P\}\ \underline{x} := \underline{E}\ \{Q, R\} \equiv (\Delta\underline{E} \wedge P \Rightarrow Q[\underline{x} \setminus \underline{E}]) \wedge (\neg\Delta\underline{E} \wedge P \Rightarrow R)$$
$$\{P\}\ \underline{x} :\in \underline{E}\ \{Q, R\} \equiv (\Delta\underline{E} \wedge P \Rightarrow \forall\, \underline{x}' \in \underline{E}\ .\ Q[\underline{x} \setminus \underline{x}']) \wedge (\neg\Delta\underline{E} \wedge P \Rightarrow R)$$
$$\{P\}\ S\ ;\ T\ \{Q, R\} \equiv \exists\, H\ .\ \{P\}\ S\ \{H, R\} \wedge \{H\}\ T\ \{Q, R\}$$
$$\{P\}\ \mathsf{try}\ S\ \mathsf{catch}\ T\ \{Q, R\} \equiv \exists\, H\ .\{P\}\ S\ \{Q, H\} \wedge \{H\}\ T\ \{Q, R\}$$
$$\{P\}\ S \sqcap T\ \{Q, R\} \equiv \{P\}\ S\ \{Q, R\} \wedge \{P\}\ T\ \{Q, R\}$$
$$\{P\}\ \mathsf{if}\ B\ \mathsf{then}\ S\ \mathsf{else}\ T\ \{Q, R\} \equiv \{\Delta B \wedge B \wedge P\}\ S\ \{Q, R\} \wedge \{\Delta B \wedge \neg B \wedge P\}\ T\ \{Q, R\} \wedge (\neg\Delta B \wedge P \Rightarrow R)$$

Theorem (*Fundamental rule of correctness for repetition*). Let B, P, Q, R be predicates, with P the *invariant*, let V be an integer expression, the *variant*, and let v be an auxiliary integer variable:

$$\{P\}\ \mathsf{while}\ B\ \mathsf{do}\ S\ \{Q, R\} \Leftarrow \{\Delta B \wedge B \wedge P \wedge (V = v)\}\ S\ \{P \wedge V < v, R\} \wedge (\Delta B \wedge B \wedge P \Rightarrow V > 0) \wedge (\Delta B \wedge \neg B \wedge P \Rightarrow Q) \wedge (\neg\Delta B \wedge P \Rightarrow R)$$

Theorem (*Fundamental rules of correctness for update,* if-then, assert). Let B, P, Q, R be predicates, a a partial function variable, E, F expressions, and S, T statements:

$$\{P\}\ a(E) := F\ \{Q, R\} \equiv (\Delta E \wedge \Delta F \wedge E \in \mathsf{dom}\ a \wedge P \Rightarrow Q[a \setminus a(E \leftarrow F)]) \wedge (\neg\Delta E \wedge P \Rightarrow R) \wedge (\neg\Delta F \wedge P \Rightarrow R) \wedge (E \notin \mathsf{dom}\ a \wedge P \Rightarrow R)$$
$$\{P\}\ \mathsf{if}\ B\ \mathsf{then}\ S\ \{Q, R\} \equiv \{\Delta B \wedge B \wedge P\}\ S\ \{Q, R\} \wedge (\Delta B \wedge \neg B \wedge P \Rightarrow Q) \wedge (\neg\Delta B \wedge P \Rightarrow R)$$
$$\{P\}\ \mathsf{assert}\ B\ \{Q, R\} \equiv (\Delta B \wedge B \wedge P \Rightarrow Q) \wedge (\neg\Delta B \wedge P \Rightarrow R) \wedge (\neg B \wedge P \Rightarrow R)$$

A *proof outline* is a program in which correctness assertions are interspersed in a systematic way. The rules for ; and for try-catch call for the "invention" of an intermediate assertion, the existentially quantified predicate H in the fundamental rule. A proof outline explicitly states that assertion; we may add further intermediate assertions, typically simplified by weakening, which is allowed by monotonicity of wp. Indentation is used to indicate the "scope" of assertions. The most general form of a proof outline–one which can be matched against any annotated program–for a sequential composition of three statements, together with the required conditions, is:

$\{P\}$	$\Leftarrow$	$\{P_1\}\ S_1\ \{Q_1, R_1\}\ \wedge$
$\quad \{P_1\}$		$\{P_2\}\ S_2\ \{Q_2, R_2\}\ \wedge$
$\quad S_1$		$\{P_3\}\ S_3\ \{Q_3, R_3\}\ \wedge$
$\quad \{Q_1, R_1\}$		$(P \Rightarrow P_1)\ \wedge$
$\quad ;$		$(Q_1 \Rightarrow P_2)\ \wedge$
$\quad \{P_2\}$		$(Q_2 \Rightarrow P_3)\ \wedge$
$\quad S_2$		$(R_1 \Rightarrow R)\ \wedge$
$\quad \{Q_2, R_2\}$		$(R_2 \Rightarrow R)\ \wedge$
$\quad ;$		$(R_3 \Rightarrow R)$
$\quad \{P_3\}$		
$\quad S_3$		
$\quad \{Q_3, R_3\}$		
$\{Q, R\}$		

Thus, the conclusion of this proof outline is $\{P\}\ S_1\ ;\ S_2\ ;\ S_3\ \{Q, R\}$. To avoid "over-annotation", we may leave out intermediate assertions. For example, if $P_1 \equiv P$, then we leave out the line $\{P_1\}$ and if $Q_1 \equiv P_2$, we leave out the line $\{P_2\}$. The most general proof outline for a try-catch statement, together with the required conditions, is:

$\{P\}$	$\Leftarrow$	$\{P_1\}\ S_1\ \{Q_1, R_1\}\ \wedge$
try		$\{P_2\}\ S_2\ \{Q_2, R_2\}\ \wedge$
$\quad \{P_1\}$		$(P \Rightarrow P_1)\ \wedge$
$\quad S_1$		$(R_1 \Rightarrow P_2)\ \wedge$
$\quad \{Q_1, R_1\}$		$(R_2 \Rightarrow R)\ \wedge$
catch		$(Q_1 \Rightarrow Q)\ \wedge$
$\quad \{P_2\}$		$(Q_2 \Rightarrow Q)$
$\quad S_2$		
$\quad \{Q_2, R_2\}$		
$\{Q, R\}$		

The rule for the conditional does not call for inventing intermediate assertions, but still can be easier to follow in a proof outline. The most general proof outline, together with the required conditions, is:

$\{P\}$	$\Leftarrow$	$\{P_1\}\ S_1\ \{Q_1, R_1\}\ \wedge$
if B then		$\{P_2\}\ S_2\ \{Q_2, R_2\}\ \wedge$
$\quad \{P_1\}$		$(\Delta B \wedge B \wedge P \Rightarrow P_1)\ \wedge$
$\quad S_1$		$(\Delta B \wedge \neg B \wedge P \Rightarrow P_2)\ \wedge$
$\quad \{Q_1, R_1\}$		$(\neg \Delta B \wedge P \Rightarrow R)\ \wedge$
else		$(Q_1 \Rightarrow Q)\ \wedge$
$\quad \{P_2\}$		$(Q_2 \Rightarrow Q)\ \wedge$
$\quad S_2$		$(R_1 \Rightarrow R)\ \wedge$
$\quad \{Q_2, R_2\}$		$(R_2 \Rightarrow R)$
$\{Q, R\}$		

If the line $\{P_1\}$ is left out, P_1 is assumed to be $\Delta B \wedge B \wedge P$. If the line $\{P_2\}$ is left out, P_2 is assumed to be $\Delta B \wedge \neg B \wedge P$. The proof outline for the if-then conditional is similar:

$\{P\}$	$\Leftarrow$	$\{P_1\}\ S_1\ \{Q_1, R_1\}\ \wedge$
if B then		$(\Delta B \wedge B \wedge P \Rightarrow P_1)\ \wedge$
$\quad \{P_1\}$		$(\Delta B \wedge \neg B \wedge P \Rightarrow Q)\ \wedge$
$\quad S_1$		$(\neg \Delta B \wedge P \Rightarrow R)\ \wedge$
$\quad \{Q_1, R_1\}$		$(Q_1 \Rightarrow Q)\ \wedge$
$\{Q, R\}$		$(R_1 \Rightarrow R)$

The rule for the repetition calls for the invention of an invariant and variant; the most general proof outline for a loop, together with the required conditions, is:

$\{P\}$	$\Leftarrow$	$\{P_0\}\ S\ \{Q_0, R_0\}$
{invariant: I}		$(\Delta B \wedge B \wedge I \wedge (V = v) \Rightarrow P_0)$
{variant: V}		$(Q_0 \Rightarrow I \wedge V < v)$
while B do		$(\Delta B \wedge B \wedge I \Rightarrow V > 0) \wedge$
$\{P_0\}$		$(P \Rightarrow I) \wedge$
S		$(\Delta B \wedge \neg B \wedge I \Rightarrow Q) \wedge$
$\{Q_0, R_0\}$		$(\neg\Delta B \wedge I \Rightarrow R) \wedge$
$\{Q, R\}$		$(R_0 \Rightarrow R)$

If the line $\{P_0\}$ is left out, P_0 is assumed to be $\Delta B \wedge B \wedge I \wedge (V = v)$. If the line $\{Q_0, R_0\}$ is left out, Q_0 is assumed to be $I \wedge V < v$ and R_0 is assumed to be false.

If in a postcondition $\{Q, R\}$ the exceptional postcondition R is false, we write the postcondition simply as $\{Q\}$; the normal postcondition is always stated. The application of proof outlines is illustrated by two examples.

Example (*Linear Search*). Let a be an array of integers of length n and let x be an integer. The task is to assign to boolean variable *found* if x occurs in a, and if it occurs, to integer variable i the index of the first occurrence. In the proof outline below, each assertion is labelled:

```
1       {A: true}
2       try
3           {B: true}
4               i := 0
5               {C: i = 0}
6               ;
7               {invariant D: 0 ≤ i ≤ n ∧ ∀ j ∈ [0, i) . a(j) ≠ x}
9               {variant V: n – i}
10              while i < N do
11                  {E: 0 ≤ i < n ∧ ∀ j ∈ [0, i) . a(j) ≠ x}
12                      if a(i) = x then
13                          {F: 0 ≤ i < n ∧ a(i) = x}
14                          raise
15                      {G: 0 ≤ i < n ∧ ∀ j ∈ [0, i] . a(j) ≠ x, H: 0 ≤ i < n ∧ a(i) = x}
16                      ;
17                      i := i + 1
18                  {I: 0 ≤ i ≤ n ∧ ∀ j ∈ [0, i) . a(j) ≠ x, J: 0 ≤ i < N ∧ a(i) = x}
19              {K: ∀ j ∈ [0, n) . a(j) ≠ x, L: 0 ≤ i < n ∧ a(i) = x}
20              ;
21              found := false
22          {M: ¬found ∧ ∀ j ∈ [0, n) . a(j) ≠ x, N: 0 ≤ i < n ∧ a(i) = x}
23      catch
24          {O: 0 ≤ i < n ∧ a(i) = x}
25          found := true
26          {P: found ∧ 0 ≤ i < n ∧ a(i) = x}
27      {Q: (found ∧ 0 ≤ i < n ∧ a(i) = x) ∨ (¬found ∧ ∀ j ∈ [0, n) . a(j) ≠ x)}
```

The condition for the whole statement is:

$\{A\}$ (lines 2-26) $\{Q\}$

For this, the required conditions by the rule for try-catch are:

$\{B\}$ (lines 4-21) $\{M, N\}$

$\{O\}$ (line 25) $\{P\}$

$A \Rightarrow B$
$N \Rightarrow O$
$M \Rightarrow Q$
$P \Rightarrow Q$

This process continues as long as rules for proof outlines can be applied. The remaining conditions are either plain boolean expressions, as the last four implications above, or are correctness assertions about primitive statements, as $\{O\}$ (line 25) $\{P\}$ above. For these, the fundamental rule of correctness is applied, for example:

$\{O\}$ *found* := true $\{P\}$
$\equiv$ $\{O\}$ *found* := true $\{P, \text{false}\}$
$\equiv$ $(\Delta\text{true} \wedge O \Rightarrow P[\mathit{found} \setminus \text{true}]) \wedge (\neg\Delta\text{true} \wedge O \Rightarrow \text{false})$
$\equiv$ $O \Rightarrow P[\mathit{found} \setminus \text{true}]$
$\equiv$ $(0 \le i < n \wedge a(i) = x) \Rightarrow (\mathit{found} \wedge 0 \le i < n \wedge a(i) = x)[\mathit{found} \setminus \text{true}]$
$\equiv$ $(0 \le i < n \wedge a(i) = x) \Rightarrow (\text{true} \wedge 0 \le i < n \wedge a(i) = x)$
$\equiv$ true

The proof leads to numerous, but simple conditions, which we leave out, except for one condition that arises at line 12. According to the rule for if-then, a condition is $\neg\Delta(a(i) = x) \wedge E \Rightarrow H$, that is, if $a(i) = x$ is not defined then the exceptional postcondition has to hold. However, we know from E that i is in the range for $a(i)$ to be defined, so this holds vacuously:

$\neg\Delta(a(i) = x) \wedge E$
$\equiv$ $\neg(\Delta a(i) \wedge \Delta x) \wedge E$
$\equiv$ $\neg\Delta a(i) \wedge E$
$\equiv$ $\neg(0 \le i < n) \wedge 0 \le i < n \wedge \forall\, j \in [0, i)\ .\ a(j) \ne x$
$\equiv$ false

Example (*Dividing Vectors*). Let a, b, c be arrays of integers of length n. The task is to assign to c vector a divided by b, where division by zero should result in maxint being assigned instead. The invariant and variant are named, so they can be referred to in the assertions.

```
{true}
    i := 0
    {i = 0}
    ;
    {invariant I:
        i ∈ [0, n] ∧ ∀ j ∈ [0, i) . (b(j) ≠ 0 ∧ c(j) = a(j) div b(j)) ∨ (b(j) = 0 ∧ c(j) = maxint)}
    {variant V: n – i}
    while i < n do
        {i < n ∧ I ∧ V = v}
            try
                c(i) := a(i) div b(i)
                {i < n ∧ I ∧ b(j) ≠ 0 ∧ c(i) = a(i) div b(i) ∧ V = v, i < n ∧ I ∧ b(i) = 0 ∧ V = v}
            catch
                {i < n ∧ I ∧ b(i) = 0 ∧ V = v}
                c(i) := maxint
                {i < n ∧ I ∧ b(i) = 0 ∧ c(i) = maxint ∧ V = v}
            {i < n ∧ I ∧ ((b(i) ≠ 0 ∧ c(i) = a(i) div b(i)) ∨ (b(i) = 0 ∧ c(i) = maxint)) ∧ V = v}
            ;
            i := i + 1
        {I ∧ V < v}
    {i ≥ n ∧ I}
{∀ j ∈ [0, n) . (b(j) ≠ 0 ∧ c(j) = a(j) div b(j)) ∨ (b(j) = 0 ∧ c(j) = maxint)}
```

PATTERNS OF EXCEPTION USE

We motivate and then present several patterns and their combinations for using exception handling to increase dependability.

The basic means of responding to an exception are *masking*, *propagating*, and *flagging*. When an exception is masked, it is not visible to the outside. That is, the handler has to establish the desired postcondition if the body fails to do so and the handler must not fail. As an example, the body may request the next command from a user in an interactive program; one of the valid commands is *help* for displaying instructions. If the user does not enter a valid command, instructions should be displayed:

```
try request next command
catch command := help
```

The normal postcondition of *request next command* is that *command* is a valid command. If that postcondition cannot be established, an exception has to be raised and the handler *command* := *help* establishes the desired postcondition. From the outside, the occurrence of an exception is not visible. In the following theorem, Q is the desired normal postcondition of the body, H the exceptional postcondition in which the body terminates if it fails and from which the handler has to establish Q.

Theorem (*Masking*). Let H, P, Q be predicates and S, T statements. If

$\{P\}\ S\ \{Q, H\}$
$\{H\}\ T\ \{Q\}$

then:

$\{P\}$ try S catch T $\{Q\}$

Proof. The proof outline is:

$\{P\}$
try
    S
    $\{Q, H\}$
catch
    $\{H\}$
    T
    $\{Q\}$
$\{Q\}$

When masking an exception, it may be necessary to weaken the desired postcondition such that the handler may always establish it. If that is not possible, the exception can be "passed" to the caller by propagating it. In that case the handler may "do some repair", like establishing a local invariant, but must terminate exceptionally.

Theorem (*Propagating*). Let H, P, Q, R be predicates and S, T statements. If

$\{P\}\ S\ \{Q, H\}$
$\{H\}\ T\ \{\text{false}, R\}$

then:

$\{P\}$ try S catch T $\{Q, R\}$

Proof. The proof outline is:

$\{P\}$
try
    S
    $\{Q, H\}$

catch
    {H}
    T
    {false, R}
{Q, R}

A direct way for making the handler T always fail is to put it in the form U ; raise. In this case, U may either terminate normally or exceptionally. As an example, the body may process file A and output file B. Writing a file may fail. If a failure occurs, the handler deletes file B, and *re-raises* the exception:

try *process file A and output B*
catch (*delete file B* ; raise)

Thus the normal postcondition of the body and the whole statement is that file B is successfully output, the exceptional postcondition of the body is that file B is partially output, and the exceptional postcondition of the whole statement is that file B is not output.

Corollary (*Propagating with re-raising*). Let H, P, Q, R be predicates and S, T statements. If

{P} S {Q, H}
{H} U {R, R}

then:

{P} try S catch (U ; raise) {Q, R}

The corollary is an example of the technique of re-raising an exception: this allows a local, partial treatment of an exception that is then passed to the caller, where the exception can be further treated. In a *modular* or *layered design*, this allows each module or layer to restore a consistent state before passing on the exception.

When an exception is flagged, it is masked, but its occurrence is recorded in a boolean variable. This way, further actions of the program may depend on whether that exception occurred or not. The above example of processing file A and outputting file B may be rephrased with flagging:

try (*process file A and output B* ; *done* := true)
catch (*delete file B* ; *done* := false)

Theorem (*Flagging*). Let H, P, Q, R be predicates, S, T statements, and *done* a boolean variable. If

{P} S {Q, H}
{H} T {R}

then

{P} try (S ; *done* := true) catch (T ; *done* := false) {(*done* ∧ Q) ∨ (¬*done* ∧ R)}

Proof. The proof outline is:

{P}
try
    {P}
        S
        {Q, H}
        ;
        {Q}
        done := true
    {*done* ∧ Q, H}
catch
    {H}

T
{*R*}
;
done := false
{¬*done* ∧ *R*}
{(*done* ∧ *Q*) ∨ (¬*done* ∧ *R*)}

We note that masking, propagating, and flagging can be combined within one try-catch statement. For example, a handler may in some cases mask the exception and in some cases propagate it.

When a statement fails, it may leave the program in an inconsistent state, for example one in which an invariant does not hold and from which another failure is likely, or an undesirable state, for example one in which the only course of action is termination of the program. We give patterns for *rolling back* to the original state. In the first pattern failure is masked. As an example, consider an interactive program that displays a form for the entry of values *u*, *v*, *w*. If invalid values are entered, the form is cancelled, or some other kind of failure occurs, the original values of *u*, *v*, *w* are restored:

*u*0, *v*0, *w*0 := *u*, *v*, *w* ;
try *display form for entering u*, *v*, *w*
catch *u*, *v*, *w* := *u*0, *v*0, *w*0

Here, the precondition of the whole statement is that *u*, *v*, *w* are valid values and the postcondition of the whole statement is again that *u*, *v*, *w* are valid values. If the body cannot establish that postcondition, an exception is raised and the hander will establish it.

In general, suppose statement *S* operates on some variables; statement *backup* makes a copy of those and statement *restore* copies those back. We formalize this by requiring that *backup* establishes a predicate *B*, which *restore* requires to roll back and which *S* has to preserve in case of failure. The backup may consist of a copy of all variables in main memory or secondary storage, or a partial or compressed copy, as long as a state satisfying *P* can be established. Statement *S* does not need to preserve *B* in case of success, e.g. can overwrite the backup of the variables. In the formulation of *rollback with masking* below, we let a statement *T* (which can be empty) do some "clean up" after restoring to achieve the desired postcondition.

Theorem (*Rollback with masking*). Let *B*, *P*, *Q* be predicates and let *backup*, *restore*, *S* be statements. If

{*P*} *backup* {*P* ∧ *B*}
{*B*} *restore* {*P*}
{*P* ∧ *B*} *S* {*Q*, *B*}
{*P*} *T* {*Q*}

then:

{*P*} *backup* ; try *S* catch (*restore* ; *T*){*Q*, *P*}

Proof. We give the proof outline:

{*P*}
backup
{*P* ∧ *B*}
;
try
S
{*Q*, *B*}
catch
{*B*}
restore
{*P*}

;
T
{*Q*}
{*Q*}

The formulation of *rollback with propagation* below simply restores the original state without cleaning up and then "passes" the exception to the caller.

Theorem (*Rollback with propagation*). Let *B*, *P*, *Q* be predicates and let *backup*, *restore*, *S* be statements. If

{*P*} *backup* {*P* ∧ *B*, *P*}
{*B*} *restore* {*P*, *P*}
{*P* ∧ *B*} *S* {*Q*, *B*}

then:

{*P*} *backup* ; try *S* catch (*restore* ; raise) {*Q*, *P*}

Proof. We give the proof outline:

{*P*}
backup
{*P* ∧ *B*, *P*}
;
try
    S
    {*Q*, *B*}
catch
    {*B*}
    restore
    {*P*, *P*}
    ;
    raise
    {false, P}
{*Q*, *P*}

Alternatively to re-raising an exception, failure may be indicated by flagging.

Theorem (*Rollback with flagging*). Let *B*, *P*, *Q* be predicates, let *backup*, *restore*, *S* be statements, and let *done* be a boolean variable that is not assigned in any of the statements. If

{*P*} *backup* {*P* ∧ *B*, *P*}
{*B*} *restore* {*P*, *P*}
{*P* ∧ *B*} *S* {*Q*, *B*}

then

{*P*} *backup* ; *done* := false ; try (*S* ; *done* := true) catch *restore* {(*done* ∧ *Q*) ∨ (¬*done* ∧ *P*), *P*}

In the formulation of the last two rollback theorems we have allowed that *backup* and *restore* fail. Statement *backup* may either establish the backup predicate *B* or may fail, but in any case must preserve *P*. Statement *restore* may succeed or fail, but in any case must establish the original predicate *P* given only *B* initially. In what follows, for simplicity we will assume that *backup* and *restore* always succeed, but note that this may be relaxed.

Suppose that two or more statements are supposed to achieve the same goal, but some statements are preferred over others–the preferred one may be more efficient, may achieve a higher precision of numeric

results, may transmit faster over the network, may achieve a higher sound quality. If the most preferred one fails, we may *fall back* to one that is less desirable, but more likely to succeed, and if that fails, fall back to a third one, and so forth. The least preferred one may simply inform the user of the failure. We call this the pattern of *degraded service*; it is the basis for further patterns. For example, assume we want to evaluate the function $\sqrt{(x^2 + y^2)}$ with arguments x and y in a robust way (Hull et al., 1994). In most cases, the evaluating that formula directly will work, but if there is an overflow or underflow, then the arguments are first scaled, the same formula is attempted, and the result unscaled. If scaling fails, then that is because of an underflow, and the result can be determined from the larger argument. If unscaling fails, the whole pattern fails. In the formulation below, the occurrence of underflow is masked but the occurrence of an overflow is propagated:

```
try       -- try the simplest formula, will work most of the time
    z := √(x² + y²)
catch     -- overflow or underflow has occurred
    try
        m := max(abs(x), abs(y)) ;
        try       -- try the formula with scaling
            t := √((x / m)² + (y / m)²)
        catch     -- underflow has occurred
            t := 1 ;
        z := m × t
    catch     -- overflow on unscaling has occurred
        z := +∞ ;
        raise
```

In the simplest form, there is one main alternative and one degraded alternative. Following theorem formalizes degraded service for three alternatives; it generalizes to more than three in a natural way. We require that all alternatives try to establish the same normal postcondition Q and statement S_{n+1} starts in a state in which S_n has failed.

Theorem (*Degraded service*). Let H_1, H_2, P, Q, R be predicates and S_1, S_2, S_3 statements. If

$\{P\}\ S_1\ \{Q, H_1\}$
$\{H_1\}\ S_2\ \{Q, H_2\}$
$\{H_2\}\ S_3\ \{Q, R\}$

then

$\{P\}$ try S_1 catch (try S_2 catch S_3) $\{Q, R\}$

Proof. The proof outline is:

```
{P}
try
    S1
    {Q, H1}
catch
    {H1}
    try
        S2
        {Q, H2}
    catch
        {H2}
        S3
        {Q, R}
    {Q, R}
{Q, R}
```

Degraded service may be used with masking, propagating, and flagging the exception. When masking, the condition for the last alternative specializes to $\{H_2\}\ S_3\ \{Q\}$. When propagating, the condition for the last alternative specializes to $\{H_2\}\ S_3\ \{\mathsf{false}, R\}$. For flagging, the boolean variable has to be generalized to an enumeration variable that indicates which alternative was taken.

Degraded service can be combined with rollback such that each attempt starts in the original state, rather than in the state that was left from the previous attempt. Hence, all alternatives have to adhere to the same specification, but try to satisfy that by different means. We give a formulation with partial propagation, causing failure in case that the last alternative fails, and leave a formulation with complete masking as an exercise to the reader. Here, *restore* has to preserve the backup predicate B to allow subsequent restores.

Theorem (*Degraded service with rollback*). Let P_0, P_1, P_2, Q be predicates and S_1, S_2 statements. If

$\{P\}$ *backup* $\{P \wedge B\}$
$\{B\}$ *restore* $\{P \wedge B\}$
$\{P \wedge B\}\ S_1\ \{Q, B\}$
$\{P \wedge B\}\ S_2\ \{Q, B\}$

then

$\{P\}$ *backup* ; try S_1 catch (*restore* ; try S_2 catch (*restore* ; raise)) $\{Q, P\}$

The *recovery block* structure specifies N alternatives together with an *acceptance test* (Horning et al., 1974). The alternatives are executed in the specified order. If the acceptance test at the end of an alternative fails or an exception is raised within an alternative, the original state is restored and the next alternative attempted. If an acceptance test passes, the recovery block terminates. If the acceptance test fails for all alternatives, the recovery block fails, possibly leading to alternatives taken at an outer level. Here is the originally suggested syntax of (Randell, 1975) and our formulation with try-catch statements; predicate A is the acceptance test:

```
ensure A                        backup ;
by S1                           try (S1 ; assert A)
else by S2                      catch
else by S3                          restore ;
else error                          try (S2 ; assert A)
                                        catch
                                            restore ;
                                            try (S3 ; assert A)
                                            catch (restore ; raise)
```

The reason for having acceptance tests is that we may not sure that the alternatives establish the desired postcondition. This may be because the alternatives use approximate algorithms that are known sometimes to fail, are based on unreliable hardware or software components, we don't have confidence in their design, or because we want to have a redundant check in a highly trusted program. The acceptance test does not have to be the complete postcondition–that would be rather impractical in general. However, suppose that we know that alternative S_i establishes normal postcondition Q_i. If we can devise a predicate A_i such that $Q_i \wedge A_i$ implies the desired postcondition Q, then A_i is an *adequate* acceptance test for S_i; hence each alternative has to have its own acceptance test, a possibility already mentioned in (Randell, 1975):

Theorem (*Recovery block*). Let B, P_0, P_1, P_2, P_3, Q be predicates, S_1, S_2, S_3 statements that preserve B, and let *rb* be defined by:

```
rb = backup ;
     try (S1 ; assert A1)
     catch
```

restore ;
try (S_2 ; assert A_2)
catch
restore ;
try (S_3 ; assert A_3)
catch (*restore* ; raise)

If

$\{P\}$ *backup* $\{P \wedge B\}$ $\quad\quad$ $\{B\}$ *restore* $\{P \wedge B\}$
$\{P \wedge B\}\ S_1\ \{Q_1, B\}$ $\quad\quad$ $Q_1 \wedge A_1 \Rightarrow Q$
$\{P \wedge B\}\ S_2\ \{Q_2, B\}$ $\quad\quad$ $Q_2 \wedge A_2 \Rightarrow Q$
$\{P \wedge B\}\ S_3\ \{Q_3, B\}$ $\quad\quad$ $Q_3 \wedge A_3 \Rightarrow Q$

then

$\{P\}$ *rb* $\{Q, P\}$

More generally, *partial acceptance tests* in form of additional assert-statements to be carried out anywhere within an alternative, rather than only at the end; failure should be detected early such that resources are not wasted. The acceptance tests may need to refer to the initial values of the variables. If the alternatives preserve the predicate B, the acceptance test may refer to the backup. We do not elaborate on these issues further.

Failures may be transient, e.g. because environmental influences, unreliable hardware, or temporary usage of resources by other programs. In such cases, a strategy is to repeat the failing statement, perhaps after a delay. In the pattern of *repeated attempts*, statement S is attempted at most n times, $n \geq 0$. When S succeeds, the whole statement succeeds, if S fails n times, the whole pattern fails.

Theorem (*Repeated attempts*). Let P, Q be predicates in which integer variable n does not occur, S, T be statements that do not assign to n, and let *ra* be defined by:

ra = while $n > 0$ do
try (S ; $n := -1$)
catch (T ; $n := n - 1$) ;
if $n = 0$ then raise

If

$\{P\}\ S\ \{Q, R\}$
$\{R\}\ T\ \{P\}$

then

$\{n \geq 0 \wedge P\}$ *ra* $\{Q, P\}$

Statement S may terminate exceptionally in an intermediate state satisfying R, from which T has to repair by re-establishing P, the precondition that S requires. Of course, if S does not modify the state when failing, then $R \equiv P$ and T can be reduced to skip.

Proof. The proof outline is:

$\{n \geq 0 \wedge P\}$
{invariant I: $(n = -1 \wedge Q) \vee (n \geq 0 \wedge P)$}
{variant: n}
while $n > 0$ do
$\{n > 0 \wedge P \wedge n = v\}$
try
S
$\{Q, n > 0 \wedge R \wedge n = v\}$

```
                ;
                n := –1
                {n = –1 ∧ n < v ∧ Q}
            catch
                {n > 0 ∧ R ∧ n = v}
                T
                {n > 0 ∧ P ∧ n = v};
                n := n – 1
                {n ≥ 0 ∧ P ∧ n + 1 = v}
            {I ∧ n < v}
        {I ∧ n ≤ 0}
        ;
        if n = 0 then
            {P}
            raise
            {false, P}
        {Q, P}
    {Q, P}
```

The theorem assumes that if *S* fails, *T* can re-establish the original state. This can be achieved by rolling back, provided that an initial backup is made.

Theorem (*Repeated attempts with rollback*). Let *P*, *Q* be predicates in which integer variable *n* does not occur and let *S*, *backup*, *restore* be statements that do not assign to *n*, and let *rr* be defined by:

```
rr  =  backup ;
       while n > 0 do
           try (S ; n := –1)
           catch (restore ; n := n – 1) ;
       if n = 0 then raise
```

If

$\{P\}$ *backup* $\{P \wedge B, P\}$
$\{B\}$ *restore* $\{P \wedge B\}$
$\{P \wedge B\}$ *S* $\{Q, B\}$

then

$\{n \geq 0 \wedge P\}$ *rr* $\{Q, P\}$

The requirement on *S* is now weakened, as in case of failure *S* has only to preserve the backup *B*; *S* does not have to preserve *B* in case of successful termination.

Proof. The proof outline is:

```
{n ≥ 0 ∧ P}
    backup
    {n ≥ 0 ∧ P ∧ B, P}
    ;
    {invariant I: (n = –1 ∧ Q) ∨ (n ≥ 0 ∧ P ∧ B)}
    {variant: n}
    while n > 0 do
        {n > 0 ∧ P ∧ B ∧ n = v}
        try
            S
```

```
                {Q, n > 0 ∧ B ∧ n = v}
                ;
                n := –1
                {n = –1 ∧ n < v ∧ Q}
            catch
                {n > 0 ∧ B ∧ n = v}
                restore
                {n > 0 ∧ P ∧ B ∧ n = v}
                ;
                n := n – 1
                {n ≥ 0 ∧ P ∧ B ∧ n + 1 = v}
            {I ∧ n < v}
        {I ∧ n ≤ 0}
        ;
        if n = 0 then
            {P}
            raise
            {false, P}
        {Q, P}
    {Q, P}
```

Instead of attempting a statement a fixed number of times, we may need to make attempts dependent on a condition. However, that condition has eventually to be become false. In the pattern of *conditional retry*, we ensure termination of attempts by requiring that the handler decreases a variant. This pattern mimics the rescue and retry statements of Eiffel (Meyer, 1997).

Theorem (*Conditional retry*). Let P, Q be predicates in which boolean variable *done* does not occur, let S, T be statements that do not assign to *done*, let V be an integer expression and let *cr* be defined by:

```
cr =  done := false ;
      while ¬done and B do
          try (S ; done := true)
          catch T ;
      if ¬done then raise
```

Assume that S preserves $V = v$. If

$\{\Delta B \wedge B \wedge P\}\ S\ \{Q, R\}$

$\{R \wedge V = v\}\ T\ \{P \wedge V < v\}$

$\Delta B \wedge B \wedge P \Rightarrow V > 0$

then

$\{P\}\ cr\ \{Q, P\}$

Proof: For the purpose of this proof, we allow booleans to be implicitly converted to integers, with false being 0 and true being 1. Thus V – *done* becomes a valid arithmetic expression, to be used as the variant of the repetition. The proof outline is:

```
{P}
    done := false ;
    {invariant: (¬done ∧ P) ∨ (done ∧ Q)}
    {variant: V – done}
    while ¬done and B do
        try
```

```
                {ΔB ∧ B ∧ P ∧ V = v}
                S
                {Q, R ∧ V = v}
                ;
                {Q}
                done := true
                {Q ∧ done}
            catch
                {R ∧ V = v}
                T
                {P ∧ V < v}
            {(Q ∧ done) ∨ (P ∧ V < v)}
        {ΔB ∧ (done ∨ ¬B) ∧ ((¬done ∧ P) ∨ (done ∧ Q))}
        if ¬done then
            {¬done ∧ ΔB ∧ ¬B ∧ P}
            raise
            {false, P}
        {done ∧ ΔB ∧ Q}
    {Q}
```

FUTURE RESEARCH DIRECTIONS

Several issues have not been touched in this chapter. Not all exceptions can be handled uniformly. For example, the pattern of repeated attempts continues a fixed number of times, but some failures may be fatal and should cause immediate exit. For this, different exception types can be introduced, such that each type has its own handler, thus generalizing statements to having one entry and *N* exits, as in (Jacobs, 2001). Programming languages offer different exception types or allow values to be passed with exceptions. Extending the theory accordingly remains future work.

While we have postulated a rule for the repetition, a formal derivation of that rule is missing. The standard definition of the repetition in terms of least fixpoints (Back & Wright, 1998), from which the rule for repetition can be derived, requires a refinement order, which is outside the scope of this chapter. A definition of recursion in terms of ordinals for a language with exists in given by (King & Morgan, 1995).

An omission in the rules for correctness is rule for the try-catch-finally statement, the difficulty being that the finalization has three possible entries–after the body succeeds, the body fails and the handler succeeds, the body fails and the handler fails–raising the question what the correctness condition for the finalizer is. This is left as future work.

We have only mentioned issues of layered and modular design in passing and not touched object-oriented design in particular. The recommendation of (Parnas & Würges, 1976) is based on the structure of a program by a layer of abstraction; each layer has the responsibility of dealing with "undesired events" at that level of abstraction, such that the abstraction hierarchy is preserved. The programming language Eiffel ties classes with exception handling (Meyer, 1997), the point being that an exception handler should re-establish the class invariant. For concurrent object-oriented programs, *conversations* have been suggested as a mechanism for coordinated distributed error recovery (Xu et al., 1995). Extending the theory to object-oriented and to concurrent designs remains future work.

In real-time systems, a further source of failure is a *time-out*, i.e. an implicitly raised exception after a specified time has elapsed. The original suggestion for recovery blocks also includes detecting time-outs for each alternative (Randell, 1975). Extending the theory to include time-outs also remains future work.

CONCLUSION

The contributions of this chapter are a theoretical explorations of statements with normal and exceptional exits and a formalization of the patterns of masking, propagating, flagging, rollback, degraded service, recovery block, repeated attempts, and conditional retry. An observations is that *procedural abstraction* in form of pre- and postconditions was used, there was no need for *data abstraction*. Following Dijkstra, the use of weakest exceptional preconditions on one hand provides the semantics of statements and at the same time proof conditions for their correctness, thus keeping the "formal overhead" minimal.

ACKNOWLEDGMENTS

The author is grateful to Ned Nedialkov for suggesting numerical computations as an example for exception handling.

REFERENCES

Back, R.-J. J. and Wright, J. v. (1998). *Refinement Calculus: A Systematic Introduction.* Springer-Verlag.

Börger, E. and Schulte, W. (2000, September). A practical method for specification and analysis of exception handling-a Java/JVM case study. *Software Engineering, IEEE Transactions on, 26*(9), 872-887.

Buhr, P. A. and Mok, W. Y. R. (2000). Advanced Exception Handling Mechanisms. *IEEE Transactions on Software Engineering, 26*(9), 820-836.

Cristian, F. (1984). Correct and Robust Programs. *IEEE Transactions on Software Engineering, 10*(2), 163-174.

Garcia, A. F., Rubira, C. M. F., Romanovsky, A., and Xu, J. (2001). A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software, 59*(2), 197-222.

Hoare, C. A. R., Hayes, I. J., Jifeng, H., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sørensen, I. H., Spivey, J. M., and Sufrin, B. A. (1987). Laws of Programming. *Communications of the ACM, 30*(9), 672–686.

Horning, J. J., Lauer, H. C., Melliar-Smith, P. M., and Randell, B. (1974). A program structure for error detection and recovery. In *Operating Systems, Proceedings of an International Symposium* 171-187.

Hull, T. E., Fairgrieve, T. F., and Tang, P.-T. P. (1994). Implementing complex elementary functions using exception handling. *ACM Transactions on Mathematical Software, 20*(2), 215–244.

Jacobs, B. (2001). A Formalisation of Java's Exception Mechanism. In D. Sands (Ed.), *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems* 284-301.

King, S. and Morgan, C. (1995). Exits in the Refinement Calculus. *Formal Aspects of Computing, 7*(1), 54-76.

Koopman, P. and DeVale, J. (2000). The Exception Handling Effectiveness of POSIX Operating Systems. *IEEE Transactions on Software Engineering, 26*(9), 837-848.

Leino, K. R. M. and Snepscheut, J. L. A. v. d. (1994). Semantics of Exceptions. In E.-R. Olderog (Ed.), *PROCOMET '94: Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi* 447-466.

Liskov, B. and Guttag, J. (2000). *Program Development in Java: Abstraction, Specification, and Object-Oriented Design.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Meyer, B. (1997). *Object-Oriented Software Construction.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Parnas, D. L. and Würges, H. (1976). Response to undesired events in software systems. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering* 437-446.

Randell, B. (1975). System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software* 437-449.

Wright, J. J. (2005, June). Compiling and Reasoning about Exceptions and Interrupts. .

Xu, J., Randell, B., Romanovsky, A., Rubira, C. M. F., Stroud, R. J., and Wu, Z. (1995). Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing* 499–508.