Chapter 6

Software fault tolerance

The need for highly reliable and safe software is becoming more and more important for software users and engineers. Traditionally, software dependability is achieved by *fault avoidance* (formal methods, structured and object oriented programming etc.) or *fault removal* (testing, verification and validation). However, despite of rigorous fault avoidance and removal, design faults often remain in software upon delivery to users. As the complexity of software increases, the probability of existence of faults increases, real world software cannot be assured to be free of errors. The capability to tolerate faults becomes crucial, at least for critical applications. Highly reliable software is possible in many applications only through *software fault tolerance.*

The term “software fault tolerance” has two interpretations: “software-fault tolerance” means the tolerance of software faults (i.e. design faults) while “software fault-tolerance” means the control of (usually hardware) fault tolerant mechanisms through software implemented functions. In this chapter, the first interpretation is discussed: the techniques necessary to enable the system to tolerate software (design) faults, and provide mechanisms to avoid system failure.

In a single version software environment, fault tolerance involves error detection and recovery features (exception handling, atomic actions, monitoring mechanisms, checkpointing and forward or backward recovery). Often the robustness and temporal redundancy is mentioned as a technique of fault tolerance. *Robustness* means all techniques that do not involve redundancy. A robust program properly handles inputs out of range, different types or formats than defined, checks the control sequence (e.g. limits on loop operations) as well as input and output. *Temporal redundancy* consists of the re-execution of the program when an error is encountered. Some errors arising from the state of execution environment (e.g. busy communication channels, full buffers) can be cleared in this way.

In more advanced software architectures *design diversity* is utilized, where functionally equivalent but independently developed software versions are applied to provide tolerance to design faults. The schemes of fault tolerant software are similar to hardware fault tolerant architectures. Typical hardware architectures include static redundancy (N-modular redundancy), active redundancy (stand-by spare) and dynamic active redundancy. The equivalent software fault tolerance schemes are *N-version programming*, *recovery blocks* and *N self-checking program-*

ming. After discussing some basic mechanisms and structuring techniques, these schemes will be described in the following sections in details.

6.1 Exception handling

Exception handling is the interruption of the normal execution flow of a program upon detection of a predefined anomalous condition and the branching to a predefined section of code [Pra96]. An exception is usually an indication that something out of the ordinary has occured; however, not all exceptions are faults (e.g. end of tape).

Exception handling was first implemented in PL/1 and Ada as a formal language construct, but some forms of it can be found also in other languages. Effective use of exception handlers requires the understanding of

- anticipated triggering events
- the manifestation of the triggering events
- the means of avoiding undesired effects.

These factors are examined through the following design process:

1. Identification of the levels of handling exceptions (e.g. instruction, module, task)
2. Classification of triggering events for each level (e.g. invalid instruction, device failure, late response, arithmetic overflow)
3. Enumeration of the events and avoidance of effects (e.g. retry, terminate task, restore checkpoint)

The avoidance of undesired effects is a nontrivial activity. A lot of side effects has to be taken into account. E.g. if a task is terminated, the open files (streams) has to be closed, locks have to be released, the partner task have to be notified etc.

At different levels of the system different philosophies are defined. At lowest (instruction) level, retry is a common technique. At higher levels (modules) the error can be masked by triggering the built-in fault tolerance mechanism (e.g. recovery block scheme). At highest levels (task, system) an orderly shutdown may be the primary goal of the exception handler.

Some general objectives of implementing exception handlers are listed as follows:

- Exceptions should not be propagated out of scope (so that higher levels could not handle it).
- Exceptions should not be mapped to incorrect handlers.
- Exceptions to be handled should not be raised prior to the start of the program (handler) code (e.g. functions or subprograms called in the declaration part).

Exception handling and system structuring

In operational programs often more than the half of a program is devoted to detect and handle exceptions. Since exceptions are expected to occur rarely, the exception handling code of a system is the less documented, tested and understood part. Most of the design faults seem to be in the code which handles exceptional situations. Thus, structuring is a crucial requirement.

Generally, the system is viewed as a set of interacting components. Each component is a system in itself and may have sub-components. These components receive requests and produce responses. When a component cannot satisfy a request then it returns an exception. Three classes of exceptional situations can be distinguished. *Interface exceptions* are raised when the interface checks detect that an invalid request has been made. These exceptions must be treated by the component which made the request. *Local exception* is raised when the component detects an error in itself. The local fault tolerance capabilities of the component have to be triggered to handle this exception. *Failure exception* is raised to the requester if the component, despite of the fault tolerant capabilities, is unable to provide the service.

Exception mechanisms can be classified in two basic categories: *termination mechanisms* and *resumption mechanisms* [Cri95]. With a termination mechanism, if an exception E was signaled while in the body of a command C then the exception leads to the termination of C. With a resumption mechanism, signaling an exception E leads to the temporary halt of C, the execution of the handler of E and resumption to the code of C (if the handler executed a resume command then to the instruction following C, otherwise to a specific command decided by the handler). Thus, in the termination mechanism an exception is similar to an `exit` command, in the resumption mechanism it is similar to a procedure call.

In the literature, the termination mechanism is strongly recommended, due to the following properties:

- With a resumption mechanism, the meaning of a component depends also on the semantics of the exception handlers outside of the component (which may not be known when the component is implemented). It does not fit well to the information hiding and encapsulating principles of the components.

- With a termination mechanism, the programmer is constrained to recover a consistent state of the component. This way further requests find the component in a consistent state. With a resumption mechanism the programmer does not know whether the control resumes to the component (in this case the original state should be kept in order to continue the execution) or not (a consistent state should be restored). In the latter case, new requests may find the component in an inconsistent state.

- The semantics of terminating mechanisms is simpler to understand than the semantics of resumption mechanisms. Moreover, with a termination mechanism all cases can be programmed that call for resumption. The only disadvantage is, that some of the computation executed by the component has to be repeated.

The conclusion is, that exception handling in a component should consist of the following steps:

1. Detection
2. Attempt of masking; if it fails then
3. Consistent state recovery
4. Propagation to upper levels.

6.2 The recovery block scheme

A study of large software systems shows that the principal cause of residual design faults is system complexity. The use of appropriate structuring techniques is necessary - otherwise the additional software components assigned to the system to provide fault tolerance might lead to even decrease of the reliability.

A simple, coherent and general strategy for fault tolerance is the *recovery block scheme* (RB) developed in the early 70's [Ran75]. This approach attempts to prevent residual software faults from impacting the system by masking the effects using alternate software modules. The alternate modules are nested into a sequential program in the following way:

```
ensure { acceptance test }
by { primary alternate }
else by { alternate 2 }
...
else by { alternate 3 }
else { failure exception }
```

On entry to a recovery block the state of the system has to be saved (establishing a checkpoint). Then the primary alternate is executed and the acceptance test is evaluated. If it is passed then the execution of the primary alternate was successful and the recovery block can be exited (the checkpoint can be discarded). If the acceptance test fails, or internal exceptions occur during the execution of the primary alternate, then the second alternate has to be executed. To provide the same environment for the alternate, the error recovery is started and the initial state is restored (using the checkpoint). The acceptance test is applied to the output of the secondary module. This sequence continues until either the acceptance test is passed or there is no further alternate exists (or a time-out/deadline occurs). If all alternates fail (by the acceptance test or by raising an internal exception) then a failure exception is signaled to the requester component (environment). The operation is illustrated in Figure 6.1.

The critical point in the recovery block scheme is the error detection (acceptance test). It must be simple (without design faults), complete and exact (based on the specification). However, there is no requirement that each alternate must produce the same output, only the *acceptance* is important. The acceptance test is the last step in the checking, the alternates itself

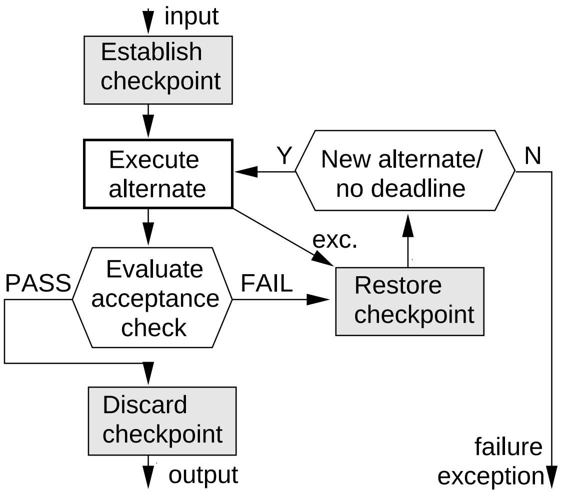

Figure 6.1: The recovery block scheme

may contain error detection facilities (e.g. assertions, timeout, exceptions from sub-components). A forward recovery can also be incorporated into the scheme by defining time-out and the corresponding actions (`timeout` statement in syntactic form).

The alternates may have different properties. The primary module is usually the most effective and fast, since most of the time only this module is executed. The alternate modules are more robust but less effective implementations.

Recovery blocks in concurrent systems

When the recovery block scheme is applied in concurrent system, the processes should continually establish, discard and restore checkpoints. If these actions are not performed in a coordinated way, then the *domino effect* can occur, i.e. due to the dependency among processes (communication) the rollback of a process can result in a cascade of rollbacks that could bring each process to the beginning of the computation (Figure 6.2).

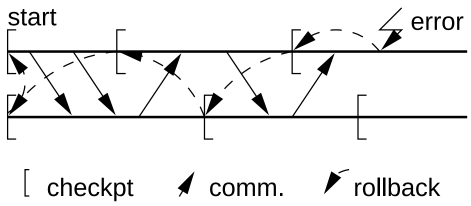

Figure 6.2: The domino effect

The structuring of communication by forming *atomic actions* can avoid the domino effect. Communication can only take place between processes that are participating in an atomic action (Figure 6.3). There are two approaches to structure the communication and establish checkpoints in a coordinated manner. In the first, *coordination-by-programmer approach*, the programmer is fully responsible for designing and structuring the communication. The *coordination-by-*

machine approach attempts to include in the system automatic user-transparent mechanisms that establish checkpoints of interacting processes. However, it introduces loss in efficiency by additional synchronization mechanisms.

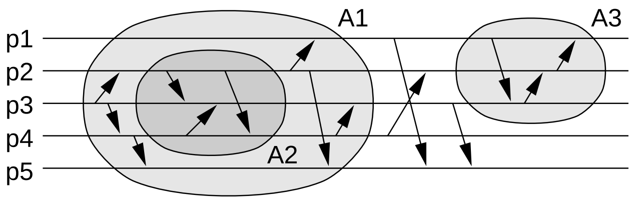

Figure 6.3: Atomic actions

The operation of the recovery block scheme in an atomic action is as follows:

1. On entry to an atomic action the processes establish a checkpoint.
2. If an error is detected by any process then all processes restore their checkpoints; after restoration all processes start their alternates.
3. All processes leave the atomic action together.
4. The results of processes are checked upon leaving the atomic action; if the acceptance check fails then all processes restore their checkpoints and start their alternates.
5. If the acceptance check is passed then the checkpoints saved on entry of the atomic action can be discarded.

The extension of the above scheme introduces two types of acceptance checks: one for each process within the atomic action to check its own goal and one for the whole structure of processes to check the global goal. If an acceptance check fails then the necessary recovery and the set of processes to be started using alternates can be decided dynamically.

The Distributed Recovery Block

The Distributed Recovery Block [Kim84] is an extension of the recovery block scheme, running on different computing nodes. In the basic scheme there are two network nodes called primary and backup. As the name suggests, each node contains an initial module and one or more alternate modules and acceptance checks (in function and time).

Under fault-free circumstances the primary node runs the primary module and the backup node runs the alternate module concurrently. Both pass the acceptance check and update their local data. Upon success of a primary node it informs the backup and outputs the results.

If the acceptance test detects an error in the primary module then the primary node attempts to inform the backup node. When the backup receives the notice, it assumes the role of the primary node. Because it has been executed the alternate module concurrently, a result is available for output. Thus, the switching time is short.

If the primary node stops executing (crash) then no message will be passed to the backup node which detects it by the expiration of a local timer. (This way hardware failures of the primary node can also be detected.)

If the primary node failed due to a software fault in the module, then this node attempts to become the backup. It executes a rollback and retry with the alternate module to process the original data once again (it is necessary to keep its state consistent with the other node). When the retry is successful then it becomes the backup node.

It is desired that the two nodes execute different modules whenever possible. If, due to the recovery from a failure, both execute the same module then one of them should attempt to change to the other module.

6.3 The N-version programming scheme

The concept of N-version programming (NVP) was introduced in 1977 [AC77]: "N-version programming is defined as the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification. The N programs possess all the necessary attributes for concurrent execution". The N programs (variants) are executed in parallel, providing the same input by the executive environment. The results of the variants are then collected and evaluated.

When $N = 2$, only error detection is available. If the results are different then one of the variants can be treated as more reliable and its results can be used, or a failure exception can be raised to the environment. When $N > 2$, a majority voting scheme can be implemented.

The necessary system elements of an N-version scheme (Figure 6.4) are

- the N independent variants
- the decision algorithm that determines a consensus result
- the executive environment (driver) that provides the input for the modules, collects and synchronizes the output.

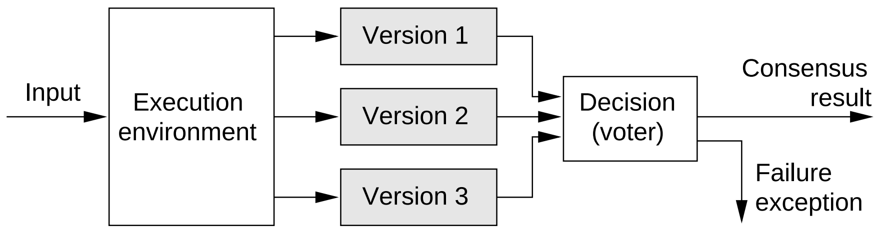

Figure 6.4: The N-version programming scheme

The variants

Crucial requirement of the N-version programming scheme is the *design diversity* of the variants. The purpose of the required diversity is to minimize the chance of common causes of software faults (e.g. compiler bugs, ambiguous arithmetic statements etc.). Diversity may be specified in the following elements [Avi95]:

- training, experience of the programmer
- application algorithms and data structures
- programming languages
- software development methods
- programming tools and environments
- testing methods and tools.

Since the variants have to be executed in the same environment, the *common specification* has to define the following features:

- functions to be implemented
- requirements for internal error detection and exception handling
- the points at which the results are checked
- the decision mechanism.

The most promising way of writing specification is using formal specification languages. If these specifications can be executed, then they can be tested for defects and may serve as prototypes.

The decision algorithm

The decision and the generation of the consensus result requires a sophisticated algorithm in order to be able to deal with the (slightly) different results of the diverse variants. The allowable ranges (for integers, floating point numbers etc.), acceptable differences (e.g. extra spaces in strings) has to be specified.

Comparison and voting can be performed also on intermediate results (if the algorithms of the diverse modules enable it). Infrequent comparisons minimize the overhead but the error detection latency is high. Frequent comparison requires commonality of program structure and intermediate data, which reduces the degree of independence.

The execution environment

The execution environment of the N-version programming scheme must provide the following functions:

- data collection and the call of the decision process
- input consistency for all variants
- enforcement of timing constraints
- supervision of the variants

- possible recovery of the variants

These functions are generic, i.e. they can execute any given set of variants. The implementation may include hardware as well as software elements. The principal requirement is a highly dependable and fast operation.

6.4 The N self-checking programming scheme

Self-checking program results from the addition of redundancy to check its own dynamic behavior during execution. In the N self-checking programming scheme (NSCP), faults tolerance is provided by active dynamic redundancy. The $N \geq 2$ self-checking components are executed in parallel. At each execution, one of the components is active (i.e. it delivers service, produces the responses), the others are considered as hot spares. If a failure exception occurs in the active component then the execution is switched to a spare self-checking component. If a spare component raises a failure exception then the execution is switched to further spares (or back to the originally active component). Error processing is performed by error detection and switching of the results. If all components fail then a failure exception is raised towards the environment.

In fact, most of the practical highly dependable systems implement this approach (Airbus, railway interlocking systems).

A few differences between the previous two schemes and the N self-checking programming scheme have to be emphasized.

- A self-checking component is often implemented by a two variants, where the first is responsible to deliver the service while the second one performs an extended acceptance test.
- The N components are being executed in parallel, this way a consistent input have to be provided for them.
- The acceptance tests of the components can be the same, derived from a common specification.

6.5 Self-configuring optimistic programming

Self-configuring optimistic programming (SCOP) scheme [BGX93] provides a trade-off between space and time redundancy needed to implement and execute software variants. A redundant architecture is provided where dependability and efficiency could be combined dynamically.

SCOP is organized in phases, each one involving the execution of an appropriate subset of dynamically configured variants. At the end of each phase, an adjudication is performed which contains all the relevant information collected so far. If the conditions for the release of the result are satisfied then the operation is ended, otherwise a new phase is performed (until variants are exhausted).

The SCOP scheme has high run-time efficiency since it requires only the necessary amount of redundancy which is needed to solve the problem (e.g. the amount of redundancy to detect an error is considerably smaller than those necessary to tolerate it). In general-purpose distributed system the significance of dynamic control increases as saved resources can be used by other applications. However, in specific applications the scheme has no significant advantages, since saved hardware resources merely left idle.

6.6 Language support for software fault tolerance

The key problem is how a set of *simple programming features* can be developed which enable the implementation of software fault tolerance and how the *supporting mechanisms* can be provided in a more natural and modular manner than by ad-hoc methods or traditional system calls [RX95].

Object oriented programming may support the above described schemes of software fault tolerance [RX93]. Forward and backward error recovery is supported by generic classes and functions ("templates") and exception handling ("catch" and "throw" in C++) which make possible to separate the application from its fault tolerance [RCS93]. High levels of reuse is also achieved, which could apply both to general software components implementing fault tolerance schemes (recovery blocks, N-version programming, N self-checking programming) and application-specific components. A set of predefined classes could support the construction of a fault tolerance framework.

However, there remain certain types of structuring that cannot be implemented elegantly in a language like C++ (e.g. explicit calls to restore the state of an object). To constrain the programmer to comply with a given convention is regularly a source of residual program faults. The pre-processor approach is often more practical.

6.7 Hardware architecture and software fault tolerance

This section presents examples of architectures tolerating single software and hardware faults [LABK95]. The architectures will be identified by $X/i/j$ where X denotes the software fault tolerance scheme (RB, NVP or NSCP), i is the number of hardware faults and j is the number of software faults tolerated. The architectures are depicted in Figure 6.5.

RB/1/1 architecture is obtained by the duplication of two variants on two hardware components. Each hardware component is software fault tolerant. Since they operate in hot standby redundancy, error of a hardware component is also tolerated.

NSCP/1/1 architecture is made up of four hardware components grouped into two pairs in hot standby redundancy and four software variants grouped into two pairs constituting a software self-checking component.

Besides the nominal tolerance, the architecture ensures the tolerance of two simultaneous

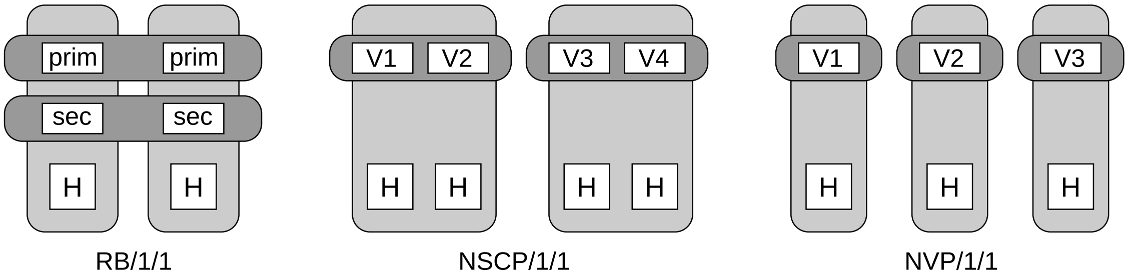

Figure 6.5: Architectures tolerating a single hw and sw fault

independent software faults in a group, the detection of related faults among two variants and the detection of three or four simultaneous independent software faults.

NVP/1/1 architecture is a direct implementation of the NVP method: three hardware components each running a distinct variant. The handling of software and hardware faults is common and performed at the software layer by the decision process. The architecture allows the detection of independent faults in two or three variants.

6.8 Comparison of the schemes

The basic schemes are compared in Table 6.1. While having essential similarities (multiple variants are needed, acceptance check or error detection is required in components) there are differences in efficiency and requirements towards the execution environment. This is why optimization techniques (SCOP) were developed.

	Recovery block scheme	**N-version programming**
Checking of results	Acceptance check Absolute (specification) Results cannot be predicted	Voting check Relative (variants) Multiple solutions (ranges)
Execution	Sequential	Parallel (efficient) sequential (slow)
Redundancy	In case of an error	Always
Execution time	Summary of modules	Time of longest module
Error handling	Backward recovery (slow)	Error is masked (fast)
Minimal overhead	Acceptance test	Voting
Problems	Recovery	Distributing and collecting data
Fault tolerance	f faults by f+1 modules	f faults by f+2 modules

Table 6.1: Comparison of recovery block scheme and N-version programming

Bibliography

[AC77] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proc. IEEE COMPSAC 77*, pages 149–155. IEEE, 1977.

[Avi95] A. Avizienis. The methodology of n-version programming. In M. R. Lyu, editor, *Software Fault Tolerance*, volume 3 of *Trends in Software*, pages 23–46. John Wiley & Sons Ltd., New York, 1995.

[BGX93] A. Bondavalli, F. Di Giandomenico, and J. Xu. A cost-effective and flexible scheme for software fault tolerance. *Int. Journal of Computer Systems Science and Engineering*, 8:233–244, 1993.

[Cri95] F. Cristian. Exception handling and tolerance of software faults. In M. R. Lyu, editor, *Software Fault Tolerance*, volume 3 of *Trends in Software*, pages 81–107. John Wiley & Sons Ltd., New York, 1995.

[Kim84] K. H. Kim. Distributed execution of recovery blocks: An approach to uniform treatment of hardware and software faults. In *Proc. 4th Int. Conference on Distributed Computing Systems*, pages 526–532, 1984.

[LABK95] J. C. Laprie, J. Arlat, C. Beounes, and K. Kanoun. Architectural issues in software fault tolerance. In M. R. Lyu, editor, *Software Fault Tolerance*, volume 3 of *Trends in Software*, pages 47–80. John Wiley & Sons Ltd., New York, 1995.

[Pra96] D. K. Pradhan. *Fault-Tolerant Computer System Design.* Prentice Hall PTR, New Jersey, 1996.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[RCS93] C. M. F. Rubira-Calsavara and R. J. Stroud. Forward and backward recovery in C++. In *Proc. 1st Predictably Dependable Computing Systems 2 Open Workshop*, pages 147–164, Tolouse, 1993.

[RX93] B. Randell and J. Xu. Object-oriented software fault tolerance: Framework, reuse and design. In *Proc. 1st Predictably Dependable Computing Systems 2 Open Workshop*, pages 165–184, Tolouse, 1993.

[RX95] B. Randell and J. Xu. The evolution of the recovery block concept. In M. R. Lyu, editor, *Software Fault Tolerance*, volume 3 of *Trends in Software*, pages 1–21. John Wiley & Sons Ltd., New York, 1995.