TM-2022/2809

Resilience Design Patterns

A Structured Approach to Resilience at Extreme Scale - Version 2.0

Christian Engelmann
Rizwan Ashraf
Saurabh Hukerikar
Mohit Kumar
Piyush Sao

December, 2022

OAK RIDGE NATIONAL LABORATORY
MANAGED BY UT-BATTELLE FOR THE US DEPARTMENT OF ENERGY

TM-2022/2809

Computer Science and Mathematics Division

Resilience Design Patterns
A Structured Approach to Resilience at Extreme Scale
Version 2.0

Christian Engelmann
Rizwan Ashraf
Saurabh Hukerikar
Mohit Kumar
Piyush Sao

Date Published: December 16, 2022

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, TN 37831-6283
managed by
UT-Battelle, LLC
for the
US DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ABFT algorithm-based fault tolerance. 19, 37, 38, 41, 42, 58, 59, 61, 63, 104, 105, 108, 109
AMTTF application mean-time to failure. 12
AMTTR application mean-time to repair. 12
API application programming interface. 17, 18, 156
ASIC application-specific integrated circuit. 15
BCH Bose-Chaudhuri-Hocquenghem. 16, 41, 61, 108
BLCR Berkeley Lab Checkpoint/Restart. 16, 38, 59, 100, 156
BMC baseboard management controller. 158
C/R checkpoint/restart. 16, 17, 20, 155, 157
CIFTS Coordinated Infrastructure for Fault-Tolerant Systems. 20
CLE Cray Linux Environment. 54, 86
CMCI Corrected Machine Check Interrupt. 15
CMOS complementary metal oxide semiconductor. 13
CORBA Common Object Request Broker Architecture. 23
CPU central processing unit. 32, 33, 50, 76, 77
CRC cyclic redundancy check. 15, 41, 61, 108
CRMS Cray reliability, availability and serviceability (RAS) and Management Subsystem. 34, 51, 78
DAG directed acyclic graph. 8
DCE detectable correctable error. 7
DECTED double-error correction triple-error detection. 16
DIMM dual in-line memory module. 20, 42, 62, 109
DMR dual-modular redundancy. 16
DMTCP Distributed MultiThreaded CheckPointing. 37, 38, 58, 59, 100
DRAM dynamic random-access memory. 16, 159, 160
DRBD distributed replicated block device. 42, 63, 114
DUE detectable uncorrectable error. 7, 12
ECC error correcting code. 7, 15, 16, 42, 62, 109, 159, 160
EDDI error detection by duplicated instructions. 17
FFT fast Fourier transform. 19
FIT failures in time. 10
FTB Fault Tolerance Backplane. 20
FTI Fault Tolerance Interface. 37, 38, 58, 59, 100
GIB Global Information Bus. 20
GPGPU general-purpose computing graphics processing unit. 34, 51, 54, 82, 85, 86
GVR Global View Resilience. 18, 37, 38, 58, 59, 104, 105
HEC high-end computing. 14
HPC high-performance computing. 1–3, 5, 8, 13–18, 20, 22, 23, 26, 29, 30, 33–35, 37, 39, 41–44, 50–52, 54–56, 58–64, 66, 68, 71, 77, 78, 83, 86, 87, 90–92, 96, 100, 101, 106, 109, 110, 113–115, 117–119, 122, 123, 127, 131, 135, 138–141, 143–145, 147, 148, 151–153, 155, 157–159, 161
HSN high-speed network. 15
HSS Hardware Supervisory System. 15, 55, 95
I/O input/output. 16, 17, 29, 145
IPMI Intelligent Platform Management Interface. 32, 33, 50, 51, 77, 158
MCA Machine Check Architecture. 15
MD molecular dynamics. 19
MDS metadata service. 37, 41, 42, 54, 55, 61–63, 90, 91, 113, 114, 117, 118

MPI Message Passing Interface. 17, 18, 20, 32–34, 37, 38, 41–43, 50, 51, 54, 55, 58, 59, 62, 63, 66, 77, 78, 85, 86, 100, 104, 105, 117, 118, 122, 144, 155, 158
MTBF mean-time between failures. 11, 35, 37, 56, 58, 87, 96, 98, 101, 103
MTTE mean-time to error. 10
MTTF mean-time to failure. 10, 13, 85, 90, 94, 99, 104, 108, 113, 117, 122, 125, 129, 133, 137
MTTR mean-time to repair. 11, 113, 117, 122, 125, 129, 133, 137
NTV near-threshold voltage. 1, 14
OO object-oriented. 21
OODA observe, orient, decide, and act. 33, 46, 50, 72, 82, 130, 133, 135, 136, 138
OPL Our Pattern Language. 21
OS operating system. 8, 16, 17, 29, 32–34, 37, 38, 50, 51, 54, 55, 58, 59, 76, 77, 90, 91, 100, 160
OS/R operating system and runtime. 20
PBS Portable Batch System. 42, 63, 118
PDF probability density function. 10
PGAS partitioned global address space. 18
PLR process-level redundancy. 17
POSIX Portable Operating System Interface. 155, 157, 158
PU planned uptime. 11, 76, 82, 85, 90, 94, 99, 104, 108
PVFS Parallel Virtual File System. 42, 63, 114, 118
RAID redundant array of independent disks. 15, 41, 42, 61–63, 108, 109, 114
RAM random-access memory. 16
RAS reliability, availability and serviceability. xi, 5, 15, 20, 34, 51, 78
SCR Scalable Checkpoint/Restart. 16, 38, 59, 100
SD scheduled downtime. 11, 76, 82, 85, 90, 94, 99, 104, 108
SDC silent data corruption. 7
SECDED single-error correction double-error detection. 7, 15, 16, 42, 62, 63, 109
SGE Sun Grid Engine. 42, 63, 114
SLURM Simple Linux Utility for Resource Management. 41, 42, 61, 63, 113, 114
SMART self-monitoring and reporting technology. 33, 51, 77
SMTTF system mean-time to failure. 12
SMTTR system mean-time to repair. 12
SOI Silicon-on-insulator. 15
SOS sum-of-squares. 19
SRMT software-based redundant multi-threading. 17
SWIFT software implemented fault tolerance. 17, 41, 43, 66, 67, 125, 126
TBB Thread Building Blocks. 22
TMR triple-modular redundancy. 16
UD unscheduled downtime. 11, 76, 82, 85, 90, 94, 99, 104, 108
ULFM user-level failure mitigation. 18, 54, 55, 85, 86, 144
VM virtual machine. 33, 46, 50, 73, 82, 138

ACKNOWLEDGMENTS

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Early Career Research Program under contract number DE-AC05-00OR22725.

ABSTRACT

Reliability is a serious concern for future extreme-scale high-performance computing (HPC) systems. Projections based on the current generation of HPC systems and technology roadmaps suggest the prevalence of very high fault rates in future systems. The errors resulting from these faults will propagate and generate various kinds of failures, which may result in outcomes ranging from result corruptions to catastrophic application crashes. Therefore, the resilience challenge for extreme-scale HPC systems requires coordination between various hardware and software technologies that are capable of handling a broad set of fault models at accelerated fault rates. Also, due to practical limits on power consumption in future HPC systems, they are likely to embrace innovative architectures, increasing the levels of hardware and software complexities. Therefore, the techniques that seek to improve resilience must navigate the complex trade-off space between resilience and the overheads to power consumption and performance. While the HPC community has developed various resilience solutions, application-level techniques as well as system-based solutions, the solution space of HPC resilience techniques remains fragmented. There are no formal methods to integrate the various HPC resilience techniques into composite solutions, nor are there methods to holistically evaluate the adequacy and efficacy of such solutions in terms of their protection coverage, and their performance & power efficiency characteristics. Additionally, few implementations of current resilience solutions are portable to newer architectures and software environments that will be deployed on future systems.

We developed a new structured approach to the management of HPC resilience using the concept of resilience-based design patterns. In general, a design pattern is a repeatable solution to a commonly occurring problem. We identified the well-known solutions that are commonly used to deal with faults, errors and failures in HPC systems. In the initial design patterns specification (version 1.0), we described the various solutions, which address specific problems in the design of resilient HPC environments, in the form of patterns. Each pattern describes a problem caused by a fault, error or failure event in an HPC environment, and then describes the core of the solution of the problem in such a way that this solution may be adapted to different systems and implemented at different layers of the system stack. The catalog of these resilience design patterns provides designers with a collection of design elements. To construct complete resilience solutions using combinations of various patterns, we defined a framework that enhances HPC designers' understanding of the important constraints and the opportunities for the design patterns to be implemented and deployed at various layers of the system stack. The design framework is also useful for establishing interfaces and mechanisms to coordinate flexible fault management across hardware and software components, as well as to consider the trade-off between performance, resilience, and power consumption when constructing a solution. The resilience design patterns specification version 1.1 included more detailed explanations of the pattern solutions, the context in which the patterns are applicable, and the implications for hardware or software design. It also provided several additional examples and detailed case studies to demonstrate the use of patterns to build realistic solutions.

In version 1.2 of the specification document, we have improved the pattern descriptions, including graphical representations of the pattern components. These improvements are largely based on critical comments, feedback and suggestions received from pattern experts and readers of the previous versions of the specification. The pattern classification has been modified to further clarify the relationships between pattern categories. This version of the specification also introduces a pattern language for resilience design patterns. The pattern language presents the patterns in the catalog as a network, revealing the relations among the resilience patterns. The language provides designers with the means to explore alternative techniques for handling a specific fault model that may have different efficiency and complexity characteristics. Using the pattern language also enables the design and implementation of comprehensive resilience solutions as a set of interconnected resilience patterns that can be instantiated across layers of the

system stack. The overall goal of this work is to provide hardware and software designers, as well as the users and operators of HPC systems, a systematic methodology for the design and evaluation of resilience technologies in HPC systems that keep scientific applications running to a correct solution in a timely and cost-efficient manner despite frequent faults, errors, and failures of various types.

Version 2.0 expands the resilience design pattern classification and catalog to include self-stabilization patterns and reliability, availability and performance models for each structural pattern.

1. INTRODUCTION

High-performance computing (HPC) systems enable transformative scientific research and discovery in various areas of national importance through computational modeling, simulation, data analysis and prediction. The opportunities to address complex emerging challenges that are important for environmental issues and national security, and to drive fundamental scientific research, are the key motivators behind the HPC community's drive towards extreme-scale HPC systems. Future systems will enable computing at scales in the hundreds of petaflops, exaflops, and beyond, which will provide computing capability for rapid design and prototyping as well as big data analysis for a variety of scientific and engineering disciplines. However, to build and effectively operate extreme-scale HPC systems, there are several key challenges, including management of power, massive concurrency and resilience to the occurrence of faults and failures in system components [62].

In the pursuit of greater computational capabilities, the architectures of HPC systems are expected to change radically. These emerging HPC systems will be innovative systems designed to communicate and compute at unprecedented rates, and will require novel technologies and architectures. Traditional HPC system design methodologies have not had to account for power constraints, or parallelism on the level designers must contemplate for future extreme-scale systems [164]. The evolution of the architectures will also require substantial changes to the programming models and the system software stacks to ensure application scalability. In the midst of these rapid changes, the resilience to faults or defects in system components, which can cause errors and failures, will be critical. While many of the innovations in the architectures will be driven by the continued scaling of transistors made possible by Moore's law, the reliability of these systems will be threatened by a decrease in individual device reliability due to manufacturing defects prevalent at deeply scaled technology nodes, device aging related effects, etc. [33]. The chips built using these devices will also be increasingly susceptible to errors due to the effects of operational and environmental conditions on the reduced noise margins arising from near-threshold voltage (NTV) operation [64] (necessary to meet the limits on system power consumption). These effects are expected to increase the rate of transient and permanent errors in the system, such that applications running on these systems will no longer be able to assume correct behavior from the underlying machine. Due to the complexity of the system environment and the interactions between the numerous hardware and software components, these errors will propagate and generate various other kinds of errors and failures, which may result in HPC application execution outcomes ranging from data corruptions to catastrophic crashes.

Managing the resilience of future extreme-scale systems is a multidimensional challenge. As HPC systems approach exaflops scale, the sheer frequency of faults and errors in these systems will render many of the existing resilience solutions ineffective. Newer modes of failures due to faults and errors, which will only emerge in advanced process technologies and complex multicomponent system environments, will require novel resilience solutions. To remain viable the adaptations of existing solutions, as well as the designs of new solutions, must also navigate the complexity of the hardware and software environments of future systems. Additionally, HPC resilience methodologies, both hardware and software, must optimize for some combination of performance, power consumption and cost while providing effective protection against faults, errors and failures. Therefore, addressing the resilience challenge in extreme-scale HPC systems will require integration and coordination between various hardware and software technologies that are collectively capable of handling a broad set of fault models at accelerated fault rates.

The HPC research community and vendors have developed a number of hardware and software resilience solutions over the years to confront faults and their consequences in a HPC system and to limit their impact on the applications. Most of these solutions are based on a limited set of underlying detection, containment

and mitigation techniques that have persisted through generations of systems and will remain important in the future. Therefore, the key to the design and implementation of HPC resilience solutions is no longer the invention of novel methodologies for dealing with the various fault types that may occur, or to manage the extreme fault rates; rather, it is based on the selection and combination of the most appropriate solutions among the fundamental resilience techniques and adapting them to the design concerns and constraints of the emerging extreme-scale systems. However, there are no systematized methods to adapt the existing solutions to future architectures and software environments, nor are there formalized methods to integrate multiple solutions into composite solutions. There is also a lack of standardized methods to investigate and evaluate the effectiveness and efficiency of such solutions. Therefore, the designers of HPC hardware and software components have a compelling need for a systematic methodology for designing, assessing and optimizing resilience solutions.

In this work, we develop a structured approach for constructing resilience solutions for HPC systems and their applications based on the concept of design patterns. Design patterns are descriptions of well-known solutions to specific, repeatedly occurring problems that are encountered in a specific domain. In an effort to develop resilience design patterns, we identify well-known techniques to handle faults and their consequences in various hardware and software components throughout the HPC system stack. In general, resilience solutions provide techniques for the detection of faults, errors or failures in a system, mechanisms to ensure that their propagation is limited, and for masking of error or failure and recovery of the system. This specification document presents a complete catalog of patterns that capture the solutions for each of these three aspects. Each pattern provides a solution to a recurring HPC resilience problem under a set of clearly defined assumptions about the type of the fault, error or failure it deals with and the constraints about the system behavior it guarantees. The resilience design patterns are specified at a high level of abstraction and describe solutions that are free of implementation details. The patterns have the potential to shape the design of HPC applications' algorithms, numerical libraries, system software, and hardware architectures, as well as the interfaces between layers of system abstraction. Therefore, they are intended to be useful for HPC application, library and tool developers, hardware architects and system software designers, as well as system users and operators.

We codify the resilience design patterns in a layered hierarchy, which classifies the patterns in the catalog, and clearly conveys the relationships among them. The hierarchical scheme enables individual hardware/software component designers to focus on problems and constraints related to detection, containment and mitigation/recovery of specific fault types in specific contexts, while system architects contemplate role of the individual patterns within the context of the overall system architecture and software environment and issues related to stitching the various patterns together and refinement of their interactions. Combining these patterns according to the guidelines given by the classification scheme provides a systematic way to design and implement new resilience solutions, port existing solutions to future architectures and software environments, and to holistically evaluate the scope and efficiency of the solutions. Therefore, using the design patterns as building blocks enables:

- Systematic design and refinement of resilience solutions by using patterns to outline the overall structure of the solution (independent of a specific implementation approach), and incrementally converging towards a detailed implementation.
- Design of solutions with a clear understanding of their protection coverage and performance efficiency.
- Evaluation and comparison of alternative resilience solutions through qualitative and quantitative evaluation of the coverage and handling efficiency of each solution.
- Design of flexible solutions through integration of multiple patterns into complete resilience solutions. The individual patterns may be independently evolved and developed for portability to different HPC

system architectures and software environments.

- Design of cross-layered resilience solutions that combine capabilities from different layers of the system stack.
- Optimization of the trade-off space, at design time or at runtime, between the key system design factors: performance, resilience, and power consumption.

We also develop a systematic methodology to combine an essential set of patterns into productive and efficient resilience solutions. We present a conceptual framework based on the notion of *design spaces* that enables HPC designers to use the patterns as reusable design elements. The framework enables designers to navigate the complexities of composing patterns into complete solutions within the constraints of performance and power overheads, the fault model and its impact on the system, hardware and software implementation challenges, etc. The overall goal of this work is to enable a systematic methodology for the design and evaluation of resilience technologies in HPC systems that keep applications running to a correct solution in a timely and cost-efficient manner despite frequent faults, errors, and failures of various types.

The rest of this document is organized as follows:

- Section 2. provides a summary of the terminology used in fault tolerance and the basic concepts of resilience to enable designers, system operators and users, whether experts in resilience or not, to understand the essence of the patterns and use them in their designs, whether in hardware or in software.
- Section 3. describes the challenges in managing the resilience of future extreme-scale HPC systems.
- Section 4. surveys the various HPC resilience solutions, including those used in production HPC systems, as well as research proposals. The aim of this section is to provide a comprehensive overview of the various HPC resilience techniques.
- Section 5. introduces the design pattern concept and discusses the potential for capturing the HPC resilience techniques in the form of patterns.
- Section 6. describes a classification scheme to organize the various resilience techniques in a layered hierarchy to enable designers to understand the capabilities of each solution and the relationships between patterns.
- Section 7. presents the catalog of resilience design patterns that capture well-understood HPC resilience techniques for error detection, recovery and masking in a structured format.
- Section 8. presents a structured methodology to use the design patterns for the construction of effective and efficient resilience solutions. The design framework introduced in this section guides HPC designers and programmers to select appropriate patterns from the catalog in order to develop complete resilience solutions.
- Section 9. presents a pattern language for the design and implementation of complete, working HPC resilience solutions. The pattern language organizes the patterns in the catalog in a hierarchy or network in order to reveal the relations among the resilience patterns. This provides the means to systematically explore alternative techniques for handling a specific fault model that may have different efficiency and complexity characteristics.
- Section 10. presents case studies that demonstrate how these patterns may be used to understand and evaluate existing resilience solutions, as well as develop new solutions using the pattern-based design framework.

2. TERMINOLOGY AND CONCEPTS

The terminology and concepts used in this document is largely based on prior work that established the definitions for HPC RAS [170, 111, 181, 69, 73, 145].

2.1 RELIABILITY

Reliability is the property of a system that characterizes its ability to perform its required functions under stated conditions for a specified time. It provides information about the probability of occurrence of an error or failure event during normal system operation.

2.2 AVAILABILITY

Availability is the property of a system that defines the readiness of a system for service. It represents the proportion of time a system provides a correct service, rather than incorrect service.

2.3 SERVICEABILITY

Serviceability is the property of a system that enables the identification of faults, errors or failures, the isolation of the root cause, and the provision of hardware or software-based maintenance in order to restore system operation.

2.4 SYSTEMS

- **System:** An entity that performs a specific set of functions.
- **Component:** A subsystem that is part of a larger system.
- **State:** A system's information about its computation, communication, interconnection, and physical condition.
- **Behavior:** What a system does to implement its function, described by a series of states.
- **Service:** A system's externally perceived behavior.
- **Functional specification:** The description of system functionality, defining the threshold between:
  - **Correct service:** The provided service is acceptable, i.e., within the functional specification.
  - **Incorrect service:** The provided service is unacceptable, i.e., outside the functional specification.
- **Life cycle:** A system has life cycle phases in the following order:
  1. **Development:** A system that is in development, which includes its design, construction, deployment and testing.
  2. **Operational:** A system that is in operation and providing correct service.
  3. **Retired:** A system that is no longer in operation since it has reached the end of its operating lifetime.
- **Operational status:** A system has the following operational states:
  1. **Scheduled service outage:** A system delivers no service due to a planned system downtime.

2. **Unscheduled service outage:** A system delivers incorrect or no service due to an unplanned outage that is caused by an error or failure.

3. **Service delivery:** A system delivers correct service under normal operating conditions.

The terms **fault**, **error** and **failure** are sometimes used interchangeably. However, in fault tolerance literature [25], these terms are associated with distinct formal concepts, which are defined in the following:

2.5 FAULTS

Fault is an underlying flaw or defect in a system that has potential to cause problems. A fault can be dormant and can have no effect. When activated during system operation, a fault leads to an error. Fault activation may be due to triggers that are internal or external to the system.

- **Fault classes:** *{benign, dormant, active} {permanent, transient, intermittent} {hard, soft}*

These fault classes have the following categories:

- **Benign:** An inactive fault that does not activate.
- **Dormant:** An inactive fault that potentially does become active at some point in time.
- **Active:** A fault that causes an error at the moment it becomes active.
- **Permanent:** The presence of the fault is continuous in time.
- **Transient:** The presence of the fault is temporary.
- **Intermittent:** The presence of the fault is temporary and recurring.
- **Hard:** A fault that is systematically reproducible.
- **Soft:** A fault that is not systematically reproducible.

The following common terms map to these fault classes:

- **Latent fault:** Any type of *dormant fault*.
- **Solid fault:** Any type of *hard fault*.
- **Elusive fault:** Any type of *soft fault*.

An example of a fault is a radiation-induced bit-flip in memory is a *dormant transient soft fault* that becomes an *active transient soft fault* when the memory is read. The fault disappears when the memory is written. A radiation-induced bit-flip in memory is a *dormant permanent soft fault* if the memory is never written. It becomes an *active permanent soft fault* when the memory is read.

2.6 ERRORS

Errors result from the activation of a fault and cause an illegal system state.

The following error classes exist:

- **Error classes:** *{undetected, detected} {unmasked, masked} {hard, soft}*

These error classes have the following categories:

- **Undetected:** An error whose presence is not indicated.

- **Detected:** An error whose presence is indicated by a message or a signal.
- **Masked:** An error whose impact is compensated so that the system specification is satisfied despite the incorrect state; the propagation of the error is limited.
- **Unmasked:** An error that has not been compensated and has the potential to propagate.
- **Hard:** An error caused by a permanent fault.
- **Soft:** An error caused by a transient or intermittent fault.

The following common terms map to these error classes:

- **Latent error:** Any type of *undetected error*.
- **Silent error:** Any type of *undetected error*.
- **SDC:** An *undetected unmasked hard* or *soft error* that affects the system state.

In an application program, the faulty assignment of a value to a loop counter variable may result in an error that results in an illegal value for that variable. When the variable is used for control of a for-loop's execution, it may lead to incorrect program behavior. In the memory of a system, an *active transient soft fault*, created by a radiation-induced bit-flip in memory being read, causes an *undetected masked soft error*, when the read value is used in a multiplication with another value that happens to be 0. This causes an *undetected unmasked soft error*, or silent data corruption (SDC), when the read value is used as an index in a memory address calculation. A *detectable correctable error (DCE)* is often transparently handled by hardware, such as a single bit flip in memory that is protected with single-error correction double-error detection (SECDED) error correcting code (ECC) [131]. A *detectable uncorrectable error (DUE)* typically results in a failure, such as multiple bit flips in the same addressable word that escape SECDED ECC correction, but not detection, and ultimately cause an application abort. An undetectable error may result in SDC, e.g., an incorrect application output.

2.7 FAILURES

Failure occurs if an error reaches the service interface of a system, resulting in system behavior that is inconsistent with the system's specification.

The following failure classes exist:

- **Failure classes:** *{undetected, detected} {permanent, transient, intermittent} {complete, partial, Byzantine}*

These failure classes have the following categories:

- **Undetected:** A failure whose occurrence is not indicated.
- **Detected:** A failure whose occurrence is indicated by a message or a signal.
- **Permanent:** The presence of the failure is continuous in time.
- **Transient:** The presence of the failure is temporary.
- **Intermittent:** The failure is temporary but recurring in time.
- **Complete:** A failure that causes service outage of the system.
- **Partial:** A failure causing a degraded service within the functional specification.

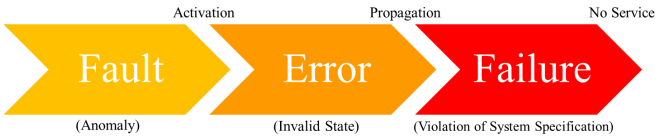


Figure 1. Relationship between fault, error and failure

- **Byzantine:** A failure causing an arbitrary deviation from the functional specification.

The following common terms map to these failure classes:

- **Fail-stop:** An *undetected* or *detected failure* that completely halts system operation, which often causes an irretrievable loss of state.
- **Fail-safe:** A mode of system operation that mitigates the consequences of a system failure.

In an application program, a faulty value assignment to a pointer variable leads to erroneous accesses to a data structure or buffer overflow, which in turn may cause the program to crash due to an attempt to access an out-of-bound memory location. In the hardware, an *active transient soft fault*, created by a radiation-induced bit-flip in memory being read, causes an *undetected unmasked soft error*, when the read value is used as an index of a memory address calculation. A memory access violation caused by using a corrupted address results in a *detected permanent failure*, as the executing process is killed by the operating system (OS), and a message is provided to the user. However, if using the corrupted calculated address results in an incorrect service that is not indicated, such as erroneous output, an *undetected intermittent Byzantine failure* occurred.

2.8 THE RELATIONSHIP BETWEEN FAULTS, ERRORS AND FAILURES

While a fault is the cause of an error, its manifestation as a state change is considered an error, and the transition to an incorrect service is observed as a failure [170]. A failure in a HPC system is typically observed through an application abort or a full/partial system outage. For example, a bug or stuck bit is a fault that manifests as an error in the form of a bad value or incorrect program execution; a failure is when an application aborts or a system crashes. There is a causality relationship between fault, error and failure, as shown in Figure 1. A **fault-error-failure chain** is a directed acyclic graph (DAG) representation with faults, errors and failures represented by its vertices. When the system is composed of multiple components, errors may be transformed into other errors and propagate through the system generating further errors, which may eventually result in a failure in the system. The failure of a single component causes a permanent or transient external fault for any other components that receive service from the failed component, which potentially causes errors and failures in those components. A **failure cascade** occurs when the failure of a component A causes an error and subsequently a failure in component B interfaced with A, and the sequence of events leads to failures in other system components interfaced with B. For example, a faulty procedure argument leads to erroneous computation and may manifest as an error in the form of an illegal procedure return value. To the caller of the function, this activates a chain of errors as the incorrect return value is passed to other functions, and the error propagates until service failure occurs, i.e., a program crash. In a parallel application, the failure of a process may potentially lead to failure of other processes communicating with the failed process, which causes the parallel application to hang.

2.9 RESILIENCE CAPABILITIES

There are three key components to designing a resilience strategy:

2.9.1 Detection

Detection entails the discovery of an error in the state of the system, either in the data, or in the instructions. It is typically accomplished with the help of redundancy; the extra information enables the verification of correct values.

Errors are detected by identifying the corresponding state change. Failures are detected by identifying the corresponding transition to an incorrect service. An error or a failure is indicated by a detector. This detector itself can experience errors or failure as well, which may lead to undetected errors or failures, or the false positive indications of error or failure events.

The following detection classes exist:

- **Detection classes:** *{true, false} {positive, negative}*

These detection classes have the following categories:

- **True:** A correct detection.
- **False:** An incorrect detection.
- **Positive:** An indication, such as a message or a signal.
- **Negative:** No indication.

2.9.2 Containment

A containment capability enables limiting the effects of an error from propagating. Containment is achieved by reasoning about the modularity of components or sub-systems that make up the system. In terms of resilience of the system, a containment module is a unit that fails independently of other units and it is also the unit of repair or replacement.

2.9.3 Masking

Masking may occur naturally when the erroneous state does not propagate, such as when an erroneous value is multiplied with 0. Masking may also entail *recovery* or *mitigation*, which ensures correct operation despite the occurrence of an error. Masking is usually accomplished by providing additional redundant state information in order to construct correct, or at least acceptably close, values of the erroneous state. When the masking involves the change of incorrect state into correct state, it is called *error correction*.

2.10 RESILIENCE METRICS

2.10.1 Reliability Metrics

The following metrics are widely used for quantifying various aspects of a system's reliability:

- **Error** or **failure reliability:** The probability of a system not experiencing an error or failure during $0 \leq t$, $R(t)$.

$$R(t) = 1 - F(t) = \int_t^\infty f(t)dt \tag{1}$$

- **Error** or **failure distribution:** The probability of the occurrence of an error or failure in the system during $0 \leq t$, $F(t)$.

$$F(t) = 1 - R(t) = \int_0^t f(t)dt \tag{2}$$

- **Probability density function (PDF):** The relative likelihood of an error or failure, $f(t)$. A normalized exponential PDF of $\lambda e^{-\lambda t}$ with a constant rate λ is typically assumed for the prolonged center period of the "bathtub curve".

$$R(t) = e^{-\lambda t} \tag{3}$$

- **Error** or **failure rate:** The frequency of errors or failures in a system, $\lambda(t)$.

$$\lambda(t) = \frac{f(t)}{R(t)} \tag{4}$$

- **Mean-time to error (MTTE):** A system's expected time to error.
- **Mean-time to failure (MTTF):** A system's expected time to failure.

$$MTTE\ or\ MTTF = \int_0^\infty R(t)dt \tag{5}$$

- **Failures in time (FIT):** The number of expected failures per billion hours (10^9) of a system's operation, FIT.

$$FIT = \frac{10^9}{MTTF} \tag{6}$$

- **Serial reliability:** The reliability of a system with n dependent components, $R(n,t)_s$.

$$R(n,t)_s = \prod_{i=1}^{n} R_i(t) \tag{7}$$

- **Parallel reliability:** The reliability of a system with n redundant components, $R(n,t)_p$.

$$R(n,t)_p = 1 - \prod_{i=1}^{n} (1 - R_i(t)) \tag{8}$$

- **Identical serial reliability:** The serial reliability with n identical components, $R(n,t)_{is}$.

$$R(n,t)_{is} = R(t)^n \tag{9}$$

- **Identical parallel reliability:** The parallel reliability with n identical components, $R(n,t)_{ip}$.

$$R(n,t)_{ip} = 1 - (1 - R(t))^n \tag{10}$$

2.10.2 Availability Metrics

Availability is the property of a system that defines the proportion of time it provides correct service. The following availability metrics are used:

- **Availability:** The proportion of time a system provides a correct service, A.

$$A = \frac{t_{pu}}{t_{pu} + t_{ud} + t_{sd}} \tag{11}$$

$$= \frac{MTTF}{MTTF + MTTR} \tag{12}$$

$$= \frac{MTTF}{MTBF} \tag{13}$$

$$\tag{14}$$

- **Planned uptime (PU):** A system's *service delivery* time, t_{pu}.
- **Unscheduled downtime (UD):** A system's *unscheduled service outage* time, t_{ud}.
- **Scheduled downtime (SD):** A system's *scheduled service outage* time, t_{sd}.
- **Mean-time to repair (MTTR):** The expected time to repair/replace a system, $MTTR$.
- **Mean-time between failures (MTBF):** The expected time between failures in a system, $MTBF$.

$$MTBF = MTTF + MTTR \tag{15}$$

- **Recovery rate:** The frequency of recoveries in a system, $\mu(t)$.
- **Serial availability:** The availability of a system with n dependent components, A_s.

$$A_s = \prod_{i=1}^{n} A_i \tag{16}$$

- **Parallel availability:** The availability of a system with n redundant components, A_p.

$$A_p = 1 - \prod_{i=1}^{n} (1 - A_i) \tag{17}$$

- **Identical serial availability:** The availability of a system with n identical components, A_{is}.

$$A_{is} = A^n \tag{18}$$

- **Identical parallel availability:** The availability of a parallel system with n identical redundant components, A_{ip}.

$$A_{ip} = 1 - (1 - A)^n \tag{19}$$

A system can also be rated by the number of 9s in its availability figure (Table 1). For example, a system with a five-nines availability rating has 99.999% availability and an annual downtime of 5 minutes and 15.4 seconds.

Table 1. Availability measured by the "nines"

9s	Availability	Annual Downtime
1	90%	36 days, 12 hours
2	99%	87 hours, 36 minutes
3	99.9%	8 hours, 45.6 minutes
4	99.99%	52 minutes, 33.6 seconds
5	99.999%	5 minutes, 15.4 seconds
6	99.9999%	31.5 seconds

2.10.3 Error and Failure Detection Metrics

- **Precision:** The fraction of indicated errors or failures that are actual errors or failures.
- **Recall:** The fraction of all errors or failures in the system that are detected and indicated.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} = \frac{True\ Positives}{Indicated\ Errors\ or\ Failures} \tag{20}$$

$$= 1 - \frac{False\ Positives}{True\ Positives + False\ Positives} = 1 - \frac{False\ Positives}{Indicated\ Errors\ or\ Failures} \tag{21}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} = \frac{True\ Positives}{Errors\ or\ Failures} \tag{22}$$

$$= 1 - \frac{False\ Negatives}{True\ Positives + False\ Negatives} = 1 - \frac{False\ Negatives}{Errors\ or\ Failures} \tag{23}$$

For example, a *true positive* detection corresponds to an existing error or failure being indicated, while a *false positive detection* corresponds to a non-existing error or failure being indicated. A *true negative* detection corresponds to a non-existing error or failure not being indicated, while a *false negative* detection corresponds to an existing error or failure not being indicated.

2.10.4 Mean Time to Failure and Mean Time to Repair

Resilience is measured by vendors and operators from the system perspective, e.g., by system mean-time to failure (SMTTF) and system mean-time to repair (SMTTR). Users measure resilience from the application perspective, using the metric application mean-time to failure (AMTTF) and application mean-time to repair (AMTTR) [170]. Both perspectives are quite different [53]. For example, an application abort caused by a main memory DUE does not require the system to recover, i.e., the SMTTR is 0. However, the aborted application needs to recover its lost state after it has been restarted, i.e., the AMTTR may be hours. Conversely, a failure of a parallel file system server may only impact a subset of the running applications, as the other ones access a different server. In this case, the server failure is counted toward the SMTTF, while the AMTTF differs by application.

3. THE RESILIENCE CHALLENGE FOR EXTREME-SCALE HPC SYSTEMS

Various studies that analyze faults, errors and failures in HPC systems indicate that faults are not rare events in large-scale systems and that the distribution of failure root cause is dominated by faults that originate in hardware. These may include faults due to radiation-induced effects such as particle strikes from cosmic radiation, circuit aging related effects, and faults due to chip manufacturing defects and design bugs that remain undetected during post-silicon validation and manifest themselves during system operation. With aggressive scaling of complementary metal oxide semiconductor (CMOS) devices, the amount of charge required to upset a gate or memory cell is decreasing with every process shrink. For very fine transistor feature sizes, the lithography used in patterning transistors causes variations in transistor geometries such as line-edge roughness, body thickness variations and random dopant fluctuations. These effects lead to variations in the electrical behavior of individual transistor devices, and this manifests itself at the circuit-level in the form of variations in circuit delay, power, and robustness [22]. The challenge of maintaining resilience continues to evolve as process technology continues to shrink and system designers will use components that operate at lower threshold voltages. The shrinking noise margins makes the components inherently less reliable and leads to a greater number of manufacturing defects, as well as device aging-related effects. The use of system-level performance and power modulation techniques, such as dynamic voltage/frequency scaling, also tend to induce higher fault rates. It is expected that future exascale-capability systems will use components that have transistor feature sizes between 5 nm and 7 nm, and that these effects will become more prevalent, thereby causing the system components to be increasingly unreliable [51]. The modeling and mitigation of these effects through improved manufacturing processes and circuit-level techniques might prove too difficult or too expensive.

Today's petascale-class HPC systems already employ millions of processor cores and memory chips to drive HPC application performance. The recent trends in system architectures suggest that future exascale-class HPC systems will be built from hundreds of millions of components organized in complex hierarchies. However, with the growing number of components, the overall reliability of the system decreases proportionally. If p is the probability of failure of an individual component and the system consists of N components, the probability that the complete system works is $(1 - p)^N$ when the component failures are independent. It may therefore be expected that some part of an exascale class supercomputing system will always be experiencing failures or operating in a degraded state. The drop in MTTF of the system is expected to be dramatic based on the projected system features [110]. In future exascale-class systems, the unreliability of chips due to transistor scaling issues will be amplified by the large number of components. For long-running scientific simulations and analysis applications that will run on these systems, the accelerated rates of system failures will mean that their executions will often terminate abnormally, or in many cases, complete with incorrect results. Finding solutions to these challenges will therefore require a concerted and collaborative effort on the part of all the layers of the system stack.

Due to the constraints of power, resilience and performance, emerging HPC system architectures will employ radically different node and system architectures. Future architectures will emphasize increasing on-chip and node-level parallelism, in addition to scaling the number of nodes in the system, in order to drive performance while meeting the constraints of power [165]. Technology trends suggest that present memory technologies and architectures will yield much lower memory capacity and bandwidth per flop of compute performance. Therefore, emerging memory architectures will be more complex, with denser memory hierarchies and utilize more diverse memory technologies. The node and system architectures will also become increasingly heterogeneous. The consequence of these architectural shifts is the increase in complexity of the system software, and hence more errors are likely. The software stack must also contend with power and resilience management, which also increase the complexity of the system software. Furthermore, the scale of extreme-scale systems requires system software services to be decentralized to

remain scalable, which also increases the complexity and susceptibility to errors. Additionally, application codes are also becoming increasingly intricate, employing several separately developed software components and modules for numerical analysis, visualization, etc. The growing complexity of the software stack compounds the resilience challenge by making it difficult to analyze the origin of faults, the propagation of errors and limiting their impact.

Resilience is an approach to fault tolerance for high-end computing (HEC) systems that seeks to keep the application workloads running to correct solutions in a timely and efficient manner in spite of frequent errors [58]. The emphasis is on the application's outcome and the reliability of application level information in place of or even at the expense of reliability of the system. Resilience technologies in HPC embrace the fact that the underlying fabric of hardware and system software will be unreliable and seek to enable effective and resource efficient use of the platform in the presence of system degradations and failures [51]. A complete resilience solution consists of detection, containment and mitigation strategies.

Performance, resilience, and power consumption are interdependent key system design factors. An increase in resilience (e.g., though redundancy) can result in higher performance (as less work is wasted) and in higher power consumption (as more hardware is being used). Similarly, a decrease in power consumption (e.g., through NTV operation) can result in lower resilience (due to higher soft error vulnerability) and lower performance (due to lower clock frequencies and more wasted work). The performance, resilience, and power consumption cost/benefit trade-off between different resilience solutions depends on individual system and application properties. Understanding this trade-off at system design time is a complex problem due to uncertainties in future system hardware and software reliability. It is also difficult due to a needed comprehensive methodology for design space exploration that accounts for performance, resilience, and power consumption aspects across the stack and the system. Similarly, runtime adaptation to changing resilience demands, while staying within a fixed power budget and achieving maximum performance, is currently limited to chekpoint placement strategies. While resilience technologies seek to provide efficient and correct operation despite the frequent occurrence of faults and defects in components that lead to errors and failures in HPC systems, there is no standardized methodology for optimizing the trade-off, at design time or runtime, between the key system design factors: performance, resilience, and power consumption.

4. SURVEY OF HPC RESILIENCE TECHNIQUES

This section surveys various fault-tolerance techniques used in practical computing systems, as well as research proposals.

4.1 HARDWARE-BASED TECHNIQUES

Various HPC vendors have developed several hardware-based resilience technologies. The use of SECDED ECC for main memory, caches, registers and architectural state is typical in most modern HPC installations. ECC uses a flavor of redundancy in memory structures that typically add additional bits to enable detection and correction of memory errors. Systems such as the Cray XC40 also include independent, redundant power supplies, redundant voltage regulator modules, redundant paths to all system, redundant array of independent disks (RAID) and redundant hot swap blowers to ensure continuous operation in the event that one of these units experiences malfunction or failure [103].

Production HPC systems also include reliability, availability and serviceability management systems for monitoring and control. The Cray XT6 and newer series of machines include an integrated Cray Hardware Supervisory System (HSS) that monitors operation of all operating system kernels. The HSS provides hardware and software-based monitoring for all major hardware components in the system; it also controls power-up, power-down and boot sequences, manages the interconnect, and displays the machine state to the system administrator. The HSS system also includes the NodeKARE, which automatically runs diagnostics on all compute nodes involved in the application removing any unhealthy nodes from the compute pool [101]. Generations of the IBM Blue Gene series have included monitoring systems that generate online information about the state of hardware and software of the system and store such information in the RAS event log [112]. The RAS log information may be used for post-hoc analysis [117] to understand system's resilience behavior, or even in online analysis to discover trends in the failure events [193].

The Cray XE6 architecture was designed to tolerate the failure of compute nodes or services nodes [102]. The Cray Gemini high-speed network (HSN) is designed to dynamically route around the failed nodes [183]. The Cray Gemini support out-of-band network management. Each torus link in the Gemini comprises four groups of three lanes and the cyclic redundancy check (CRC) code for each packet are checked by each device with automatic link level retry upon the occurrence of an error. Additionally, Gemini uses ECC to protect major memories and data paths for the protection of all packet traffic in the network. The Cray Aries network [17] also contains similar features.

In the IBM Blue Gene/Q, many mechanisms are deployed to protect the chip against errors. The compute node of a Blue Gene/Q contains an application-specific integrated circuit (ASIC) compute chip that consists of 18 PowerPC-A2 processor cores. Sixteen of these processor cores are designated as 'user' cores, one core is designated for system management (handles interrupts message passing, etc.) and the eighteenth core is a spare [91]. Additionally, the manufacturing process of the chip uses of Silicon-on-insulator (SOI) technology, radiation-hardened latches, and detection and correction for on-chip arrays, register files and caches [46].

The architectures of processors used in modern HPC systems contain a range of features for reliability management. The IBM RAS features of the Power8 architecture contain features for soft error handling, active memory mirroring, dynamic substitution of unused memory for predictive memory faults, memory buffer replay, and triple redundant global clocks and chip-level thermal sensors [94]. The Intel Xeon series of processors includes RAS features that enable co-operative hardware/software management of errors in the processors. This includes Corrected Machine Check Interrupt (CMCI) for predictive failure analysis, Machine Check Architecture (MCA) interrupt-based recovery, as well as capabilities to add processors,

memory during system operation and socket migration in collaboration with the operating system [54].

Dual-modular redundancy (DMR) and triple-modular redundancy (TMR) hardware-based approaches have been successfully used in mission-critical systems through hardware replication. Examples of fault-tolerant servers include the Tandem Non-Stop [127] and the HP NonStop [30] that use two redundant processors running in locked step. The IBM G5 [167] employs two fully duplicated lock-step pipelines to enable low-latency detection and rapid recovery. While these solutions are transparent to the supervisor software and application programmer, they require specialized hardware. While SECDED is the most widely used variant of ECC, researchers have also explored Bose-Chaudhuri-Hocquenghem (BCH) and double-error correction triple-error detection (DECTED) [135] for multi-bit error detection and correction. Chipkill [60] is a stronger memory protection scheme that is widely used in production HPC systems. The scheme accommodates single dynamic random-access memory (DRAM) memory chip failure as well as multi-bit errors from any portion of a single memory chip by interleaving bit error-correcting codes across multiple memory chips.

4.2 SOFTWARE-IMPLEMENTED TECHNIQUES

Software-based redundancy promises to offer more flexibility and tends to be less expensive in terms of silicon area as well as chip development and verification costs; it also eliminates the need for modifications of architectural specifications.

4.2.1 Operating System & Runtime-based Solutions

The most widely used strategies in production HPC systems are predominantly based on checkpoint/restart (C/R). In general, C/R approaches are based on the concept of capturing the state of the application at key points of the execution, which is then saved to persistent storage. Upon detection of a failure, the application state is restored from the latest disk committed checkpoint, and execution resumes from that point. The Condor standalone checkpoint library [120] was developed to provide checkpointing for UNIX processes, while the Berkeley Lab Checkpoint/Restart (BLCR) library [65] was developed as an extension to the Linux OS. The libckpt [148] provided similar OS-level process checkpointing, albeit based on programmer annotations.

In the context of parallel distributed computing systems, checkpointing requires global coordination, i.e., all processes on all nodes are paused until all messages in-flight and those in-queue are delivered, at which point all the processes' address spaces, register states, etc., are written to stable storage, generally a parallel file system, through dedicated input/output (I/O) nodes. The significant challenge in these efforts is the coordination among processes so that later recovery restores the system to a consistent state. These approaches typically launch daemons on every node that form and maintain communication groups that allow tracking and managing recovery by maintaining the configuration of the communication system. The failure of any given node in the group is handled by restarting the failed process on a different node, by restructuring the computation, or through transparent migration to another node [13] [42] [114].

Much work has also been done to optimize the process of C/R. A two-level recovery scheme proposed optimization of the recovery process for more probable failures, so that these incur a lower performance overhead while the less probable failures incur a higher overhead [179]. The Scalable Checkpoint/Restart (SCR) library [130] proposes multilevel checkpointing where checkpoints are written to storage that use random-access memory (RAM), flash, or local disk drive, in addition to the parallel file system, to achieve much higher I/O bandwidth. Oliner *et al.* propose an opportunistic checkpointing scheme that writes checkpoints that are predicted to be useful - for example, when a failure in the near future is likely [143]. Incremental checkpointing dynamically identifies the changed blocks of memory since the last checkpoint

through a hash function [12] in order to limit the amount of state required to be captured per checkpoint. Data aggregation and compression also help reduce the bandwidth requirements when committing the checkpoint to disk [104]. Plank *et al.* eliminate the overhead of writing checkpoints to disk altogether with a diskless in-memory checkpointing approach [150].

Process-level redundancy (PLR) [166] creates a set of redundant application processes whose output values are compared. The scheduling of the redundant processes is left to the OS. The RedThreads application programming interface (API) [99] provides directives that support error detection and correction semantics through the adaptive use of redundant multithreading.

4.2.2 Message Passing Library-based Solutions

In general, automatic application-oblivious checkpointing approaches suffer from scaling issues due to the considerable I/O bandwidth for writing to persistent storage. Also, practical implementations tend to be fragile [68]. Therefore, several Message Passing Interface (MPI) libraries have been enabled with capabilities for C/R [115]. The CoCheck MPI [172], based on the Condor library, uses synchronous checkpointing in which all MPI processes commit their message queues to disk to prevent messages in flight from getting lost. The FT-MPI [74], Open MPI [100], MPICH-V [34] and LAM/MPI [160] implementations followed suit by incorporating similar capabilities for C/R. In these implementations, the application developers do not need to concern themselves with failure handling; the failure detection and application recovery are handled transparently by the MPI library, in collaboration with the OS.

The process-level redundancy approach has also been evaluated in the context of a MPI library implementation [77], where each MPI rank in the application is replicated and the replica takes the place of a failed rank, allowing the application to continue. The RedMPI library [70] [78] replicates MPI tasks and compares the received messages between the replicas in order to detect corruptions in the communication data. Studies have also proposed the use of proactive fault tolerance in MPI [133] [186]. However, with the growing complexity of long running scientific applications, complete multi-modular redundancy, whether through hardware or software-based approaches, will incur exorbitant overhead to costs, performance and energy, and is not a scalable solution to be widely used in future exascale-class HPC systems.

4.2.3 Compiler-based Solutions

Software implemented fault tolerance (SWIFT) [153] is a compiler-based transformation which duplicates all program instructions and inserts comparison instructions during code generation so that the duplicated instructions fill the scheduling slack. The DAFT [195] approach uses a compiler transformation that duplicates the entire program in a redundant thread that trails the main thread and inserts instructions for error checking. The software-based redundant multi-threading (SRMT) [188] uses compiler analysis to generate redundant threads mapped to different cores in a chip multi-processor and optimizes performance by minimizing data communication between the main thread and trailing redundant thread. Similarly, error detection by duplicated instructions (EDDI) [142] duplicates all instructions and inserts “compare" instructions to validate the program correctness at appropriate locations in the program code. The ROSE::FTTransform [118] applies source-to-source translation to duplicate individual source-level statements to detect transient processor faults.

4.2.4 Programming Model Techniques

Most programming model approaches advocate a collaborative management of the reliability requirements of applications through a programmer interface in conjunction with compiler transformations, a runtime

framework and/or library support. Each approach requires different levels of programmer involvement, which has an impact on amount of effort to re-factor the application code, as well as on the portability of the application code to different platforms.

HPC programs usually deploy a large number of nodes to implement a single computation and use MPI with a flat model of message exchange in which any node can communicate with another. Every node that participates in a computation acquires dependencies on the states of the other nodes. Therefore, the failure of a single node results in the failure of the entire computation since the message passing model lacks well-defined failure containment capabilities [68]. User-level failure mitigation (ULFM) [31] extends MPI by encouraging programmer involvement in the failure detection and recovery by providing a fault-tolerant API for MPI programs. The error handling of the communicator has changed from MPI_ERRORS_ARE _FATAL to MPI_ERRORS_RETURN so that error recovery may be handled by the user. The proposed API includes MPI_COMM_REVOKE, MPI_COMM_SHRINK to enable reconstruction of the MPI communicator after process failure and the MPI_COMM_AGREE as a consistency check to detect failures when the programmer deems such a sanity check necessary in the application code.

The abstraction of the *transaction* has also been proposed to capture a programmer's fault-tolerance knowledge. This entails division of the application code into blocks of code whose results are checked for correctness before proceeding. If the code block execution's correctness criteria are not met, the results are discarded and the block can be re-executed. Such an approach was explored for HPC applications through a programming construct called *Containment Domains* by Sullivan *et al.* [48] which is based on weak transactional semantics. It enforces the check for correctness of the data value generated within the containment domain before it is communicated to other domains. These containment domains can be hierarchical and provide the means to locally recover from an error within that domain. A compiler technique that, through static analysis, discovers regions that can be freely re-executed without checkpointed state or side-effects, called idempotent regions, was proposed by de Kruijf *et al.* [57]. Their original proposal [56], however, was based on language-level support for C/C++ that allowed the application developer to define idempotent regions through specification of *relax* blocks and *recover* blocks that perform recovery when a fault occurs. The FaultTM scheme adapts the concept of hardware-based transactional memory where atomicity of computation is guaranteed. The approach requires an application programmer to define *vulnerable* sections of code. For such sections, a backup thread is created. The original and the backup thread are executed as an atomic transaction, and their respective committed result values are compared [191].

Complementary to approaches that focus on resiliency of computational blocks, the Global View Resilience (GVR) project [47] concentrates on application data and guarantees resilience through multiple snapshot versions of the data whose creation is controlled by the programmer through application annotations. Bridges *et al.* [36] proposed a `malloc_failable` that uses a callback mechanism to handle memory failures on dynamically allocated memory, so that the application programmer can specify recovery actions. The Global Arrays implementation of the partitioned global address space (PGAS) model presents a global view of multidimensional arrays that are physically distributed among the memories of processes. Through a set of library API for checkpoint and restart with bindings for C/C++/FORTRAN, the application programmer can create checkpoints of array structures. The library guarantees that updates to the global shared data are fully completed and any partial updates are prevented or undone [61]. Rolex [98] provides various resilience semantics for error tolerance and amelioration through language-based extensions that enable these capabilities to be embedded within standard C/C++ programs.

4.2.5 Algorithm-based Fault Tolerance

Algorithm-based fault tolerance (ABFT) schemes encode the application data to detect and correct errors, e.g., the use of checksums on dense matrix structures. The algorithms are modified to operate on the encoded data structures. ABFT was shown to be an effective method for application-layer detection and correction by Huang and Abraham [96] for a range of basic matrix operations including addition, multiplication, scalar product, transposition. Such techniques were also proven effective for LU factorization [55], Cholesky factorization [89] and QR factorization [107]. Several papers propose improvements for better scalability in the context of parallel systems, that provide better error detection and correction coverage with lower application overheads [155] [149] [158]. The checksum-based detection and correction methods tend to incur very high overheads to performance in sparse matrix-based applications. Sloan *et al.* [168] have proposed techniques for fault detection that employ approximate random checking and approximate clustered checking by leveraging the diagonal, banded diagonal, and block diagonal structures of sparse problems. Algorithm-based recovery for sparse matrix problems has been demonstrated through error localization and re-computation [169] [44].

Various studies have evaluated the fault resilience of solvers of linear algebra problems [38]. Iterative methods including Jacobi, Gauss-Seidel and its variants, the conjugate gradient, the preconditioned conjugate gradient, and the multi-grid begin with an initial guess of the solution and iteratively approach a solution by reducing the error in the current guess of the answer until a convergence criterion is satisfied. Such algorithms have proved to be tolerant to errors, on a limited basis, since the calculations typically require a larger number of iterations to converge, based on magnitude of the perturbation, but eventual convergence to a correct solution is possible. Algorithm-based error detection in the multigrid method shown by Mishra *et al.* [128], uses invariants that enable checking for errors in the relaxation, restriction and the interpolation operators.

For fast Fourier transform (FFT) algorithms, an error-detection technique called the sum-of-squares (SOS) was presented by Reddy *et al.* [152]. This method is effective for a broader class of problems called orthogonal transforms and therefore applicable to QR factorization, singular-value decomposition, and least-squares minimization. Error detection in the result of the FFT is possible using weighted checksums on the input and output [189].

While the previously discussed methods are primarily for numerical algorithms, fault tolerance for other scientific application areas has also been explored. In molecular dynamics (MD) simulations, the property that pairwise interactions are anti-symmetric ($F_{ij} = - F_{ji}$) may be leveraged to detect errors in the force calculations [190]. The resilience of the Hartree-Fock algorithm, which is widely used in computational chemistry, can be significantly enhanced through checksum-based detection and correction for the geometry and basis set objects. For the two-electron integrals and Fock matrix elements, knowing their respective value bounds allows for identifying outliers and correcting them with reasonable values from a range of known correct values. The iterative nature of the Hartree-Fock algorithm helps to eliminate the errors introduced by the interpolated values [180]. The fault-tolerant version of the 3D-protein reconstruction algorithm, FT-COMAR, proposed by Vassura *et al.* [182] is able to recover from errors in as many as 75% of the entries of the contact map.

4.3 INTEGRATED APPROACHES

Cross-layer resilience techniques [129] employ multiple error resilience techniques from different layers of the system stack to collaboratively achieve error resilience. The design of cross-layer resilience solutions distribute the responsibility of detection, containment and masking/recovery across multiple layers of the system in contrast to the traditional approaches that seek to implement the solution within a single layer of

abstraction. While the coordination across system layers increases the complexity of the design, the cross-layer approach enables the creation of flexible resilience solutions. Based on the fault model, frequency of fault events and their impact on the application running on a system, certain resilience tasks may be implemented in software to save chip area and system power consumption. Additionally, the system may use contextual information about the impact of an error available at each layer of the system to optimize fault coverage, performance and power consumption. To formalize the design process for resilience solutions, the cross-layer resilience study specifies a set of key tasks: detection, diagnosis, reconfiguration, recovery, and adaptation, which may be implemented using hardware or software mechanisms at different levels of the system stack [41]. The CLEAR framework [45] enables systematic analysis of resilience techniques across various layers of the system stack through exploration of the large space of combinations of resilience techniques. The framework combine selective circuit-level hardening and logic-level parity checking with algorithm-based fault tolerance methods to provide resilient operation.

Software-only cross-layer resilience techniques enable fault management through collaboration between the layers of the software stack, including operating systems, runtimes, libraries, middleware, and the applications. The Coordinated Infrastructure for Fault-Tolerant Systems (CIFTS) infrastructure is based on the notion of an information backplane (the Fault Tolerance Backplane (FTB)) that enables fault notification and response information to be propagated through a uniform interface between layers of the system stack [88]. The availability of fault information on the backplane enables the various software modules to proactively coordinate fault management. The Hobbes operating system and runtime (OS/R) provides the Global Information Bus (GIB) for sharing status information, including RAS events, that is needed by other software components [37]. The ARGO OS/R system provides publish-subscribe framework called BEACON that enables applications and system services with access to reporting and resource monitoring [144].

Much of the research in HPC resilience has focused on fault-model driven approaches, in which designers identify the physical effects (such as transient errors, permanent errors etc.), or their impact on a system (such as node failure, dual in-line memory module (DIMM) failure, etc.) that their design must tolerate. Many of these existing approaches develop techniques to address each of these individual effects. Such techniques may be very effective when system resilience must be guaranteed for only a small number of error sources. In an effort to address extreme rates and diverse sources of faults and errors, recent efforts have explored the strategies that seek to address more than one fault model. The combination of algorithm-based verification mechanisms and in-memory checkpoints for silent errors and disk checkpoints for fail-stop errors has been proved effective [29]. In the context of iterative solvers, such as the conjugate gradient solver, the combination of algorithm-based fault tolerance and disk-based checkpointing results in more resilient and more performant solution than with the use of a pure disk-checkpointing based solution [75]. The ACR is a C/R library-based solution integrated with Charm++ that accounts for failures due to hard errors as well as undetected silent data corruptions [137]. There are also solutions that leverage multiple techniques in an effort to mitigate the overhead of the resilience solution. In the MPI layer, the combination of partial redundancy and checkpointing yields lower overheads in comparison to a full redundancy approach [67].

5. DESIGN PATTERNS FOR RESILIENCE

5.1 INTRODUCTION TO DESIGN PATTERNS

A *design pattern* describes a generalizable solution to a recurring problem that occurs within a well-defined context. Patterns are often derived from best practices used by designers and they contain essential elements of the problems and their solutions. They provide designers with a template on how to solve a problem that may be used in many different situations. The patterns may also be used to describe design alternatives to a specific problem. The original concept of design patterns was developed in the context of civil architecture and engineering problems. With the goal of designing functional and aesthetically beautiful living spaces and structures, patterns captured the detailed designs of towns and neighborhoods, houses, gardens and rooms. These patterns identify and catalog solutions to recurrent problems and solutions encountered during the process of building and planning. Each pattern described a problem, which occurs repeatedly in our environment, and then described the core of the solution to that problem, in such a way that this solution may be used a million times over, without ever doing it the same way twice [15].

In general, a design pattern identifies the key aspects of a solution and creates an abstract description that makes it useful in the creation of a reusable design element. Patterns don't describe a concrete design or an implementation - they are intended to be templates that may be applied by a designer in various contexts and modified to suit the problem at hand. Patterns are also free from constraints of detail associated with the level of system abstraction at which the solution is implemented. Patterns also describe the design decisions that must be made when applying a certain solution. This enables a designer to reason about the impact of the design decisions on a system's flexibility or scalability as well as consider implementation issues. The design patterns must address a specific problem at hand, and yet must be general enough to remain relevant to future requirements of systems.

In the domain of software design, patterns were introduced in an effort to create reusable solutions in the design of software and bring discipline to the art of programming. The intent of software design patterns isn't to provide a finished design that may be transformed directly into code; rather, design patterns are used to enhance the software development process by providing proven development paradigms. With the use of design patterns, there is sufficient flexibility for software developers to adapt their implementation to accommodate any constraints, or issues that may be unique to specific programming paradigms, or the target platform for the software. Related to design patterns, the concept of algorithmic skeletons was introduced [49] and further refined [50].

In the context of object-oriented (OO) programming, design patterns provide a catalog of methods for defining class interfaces and inheritance hierarchies, and establish key relationships among the classes [82]. In many object-oriented systems, reusable patterns of class relationships and communications between objects are used to create flexible, elegant, and ultimately reusable software design. There are three categories of OO patterns: (i) **creational** patterns for ways to do instantiation of objects (ii) **structural** patterns concerned with class and object composition, and (iii) **behavioral** patterns for communication between objects. Design patterns have also been defined in the design of software architectures [40] to capture repeatedly used methodologies in software engineering practice. Pattern systems have also been developed for cataloging concurrent and networked object-oriented environments [163], resource management [109], and distributed software systems [39].

In the pursuit of quality and scalable parallel software, patterns for programming paradigms were developed [124] as well as a pattern language, called Our Pattern Language (OPL) [108]. These parallel patterns are used as means to systematically describe parallel computation and communication when

architecting parallel software. In an effort to enable a more structured approach to designing and implementing parallel applications, particularly for many-core processors, a catalog of parallel patterns enables programmers to compose parallel algorithms, which may be easily implemented using various programming interfaces such as OpenMP, OpenCL, Cilk Plus, ArBB, Thread Building Blocks (TBB) [125]. For the design of parallel algorithms, deterministic patterns support the development of systems that automatically avoid unsafe race conditions and deadlock [126].

Design patterns have been in a variety of other domains for codifying the best-known solutions to common problems, including natural language processing [176], user interface design [32], web design [66], visualization [93], and software security [63]. Patterns have also been defined for enterprise applications that involve data processing in support or automation of business processes [80] in order to bring structure to the construction of enterprise application architectures. In each of these domains of design, patterns capture the essence of effective solutions in a succinct form that may be easily applied in similar form to other contexts and problems.

5.2 DESIGN PATTERNS FOR HPC RESILIENCE SOLUTIONS

The occurrences of various types of faults, errors and failures are not rare events in modern large-scale HPC system environments. HPC resilience solutions seek effective and efficient management of the different types of fault and errors to ensure that HPC applications produce reliable outcomes despite system degradations and component failures. The focus of resilience solutions is on application correctness lieu of, or even at the expense of, reliability of state of the system.

In general, every resilience solution consists of the following core capabilities:

- **Detection:** Identifying the presence of an anomaly in the data or control value is an important aspect of any resilience management strategy. The detection and diagnosis of faults in a system may allow the remedy of the underlying defect, which may prevent the activation of an error or failure. The timely detection of errors or failures enables recovery of the system.
- **Containment:** When an error or failure is discovered in a system, containment strategies assist in limiting the impact of the event on other components in the system. Limiting the propagation enables simplified recovery strategies and prevents the error from affecting other components in a system.
- **Recovery:** The recovery aspect of any resilience solution is necessary to ensure that the application outcome is correct in spite of the presence of an error or a failure in a system. The recovery may entail a workaround to isolate and bypass the presence of an error or a failed component, complete elimination of the error or failure, and may also seek to prevent the root cause of the underlying fault from resurfacing.

Often the solutions used to achieve these capabilities are based on well-known techniques, which have been repeatedly used by hardware and software designers to guarantee that anomalous events in a system are detected, their propagation limited and their impact on the correct operation of the system is minimized. Many of these techniques that increase system reliability have been used since the early days of computing systems. These techniques are based on the use of redundant structures to mask failed components, error-control codes and duplication or triplication with voting to detect or correct information errors, diagnostic techniques to locate failed components, and automatic switchovers to replace failed subsystems [24]. Many of the hardware- and software-based resilience solutions used in HPC environments over the past three decades are also largely based on these set of techniques.

The goal of this work is to capture the fundamental techniques that are used in the design of HPC resilience

solutions and write them down in the form of design patterns. The patterns describe the techniques for detection, containment and mitigation in a highly structured format. A resilience pattern systematically names, describes the capability and explains the semantics of such a technique used in supporting resilience in HPC systems.

Each resilience technique provides different guarantees in terms of its capabilities and impact on the system operation (in the absence of any events as well as in response to a fault, error or failure). Using design patterns to construct resilience solutions forces designers to contemplate the time or the space overhead introduced, the efficiency of the reaction to a fault, error or failure event, as well as practical considerations for implementing a pattern, such as choosing the appropriate layer of system abstraction, the assessing design complexity added to the system, etc.

Therefore, we organize the resilience patterns in a catalog to provide designers with an accessible collection of well-established techniques that may be reused or adapted to create new resilience solutions. The purpose behind creating a resilience design pattern catalog is to capture the experiences and insights of HPC hardware & software designers and practitioners in designing resilience capabilities for large-scale parallel HPC systems. The patterns enable designers to build upon previous experience of using similar solutions to new HPC environments. These patterns solve specific design problems for HPC resilience and seek to make design of the solution flexible and elegant.

Previous efforts to define design patterns for fault tolerance have described a set of patterns for failure detection, recovery and mitigation. These patterns were also defined based on well-known fault tolerance solutions used in mission-critical systems such as telecommunication systems and space programs [90], distributed systems [162] and enterprise data processing systems [81]. While the capabilities of some of the patterns in these domains overlap with the capabilities of the patterns described in this document, these patterns solve problems that are significantly different from those encountered in high-performance computing systems in terms of the system's architectures, the software environments, and the nature of the applications that run on these computing systems. The patterns in this document specifically address the challenges for maintaining resilient operation in HPC systems, which entails keeping scientific applications running to a correct solution in a timely and efficient manner in spite of faults, errors, and failures. The Common Object Request Broker Architecture (CORBA) [87] defines a set of standard services and protocols defined by distributed object computing middleware. The fault tolerant version of CORBA [136] supports a range of fault tolerance strategies, including request retry, redirection to an alternative server, passive (primary/backup) replication, and active replication which provides applications with capabilities for rapid recovery from faults.

5.3 ANATOMY OF A RESILIENCE DESIGN PATTERN

The basic template of a resilience design pattern is defined in an event-driven paradigm based on the insight that any resilience solution is necessary in the presence of, or sometimes in the anticipation of an anomalous event, such as a fault, error, or failure. The abstract resilience design pattern consists of a *behavior* and a set of *activation* and *response interfaces* (Figure 2). The appeal of defining the resilience design patterns in such an abstract manner is that they are universal. The abstract definition of the resilience design pattern behavior enables description of solutions that are free of implementation details. The instantiation of pattern behaviors may cover combinations of detection, containment and mitigation capabilities. The individual implementations of the same pattern may have different levels of performance, resilience, and power consumption. Also, the resilience pattern definition abstracts a pattern's interfaces from the implementation of these interfaces. In specification of the resilience patterns that serve as reusable design elements, it is important to find pertinent techniques at the appropriate granularity, define their

activation and response interfaces in a manner that enables designers to establish key relationships among the patterns.

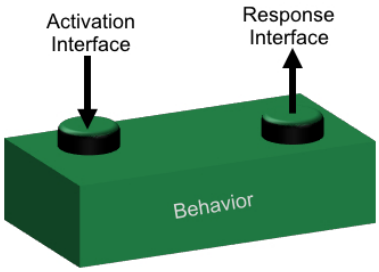


Figure 2. Anatomy of a resilience design pattern

6. CLASSIFICATION OF RESILIENCE DESIGN PATTERNS

The patterns presented in the resilience design patterns catalog provide solutions to problems that repeatedly appear in the design of fault tolerance capabilities. The solutions offered by these patterns may shape the design of hardware, system software as well as the applications and libraries independent of the scope and scale of the problem. Due to the variety in the granularity and level abstraction at which each of the patterns may be implemented, we must organize the patterns. A classification scheme codifies the relationships between the various patterns in the catalog, which enables designers to understand their individual capabilities and the relationships among the patterns, and to compose different patterns into complete resilience solutions.

The most straightforward classification scheme for the resilience solutions is one based on the layer of system abstraction at which the solution is implemented. The survey of resilience solutions presented in section 4. is organized in this manner. However, the resilience patterns have various properties each of which can be used as a basis for developing a classification scheme. These properties include:

- **Capability:** Whether the pattern offer detection, containment, recovery or masking semantics, or a combination of these capabilities.
- **Protection Coverage:** The scope of the system that the pattern protects.
- **Fault Model:** The type of event that the pattern is equipped to handle.
- **Impact:** How a fault, error or failure event affects the outcome of a system, and the capabilities of the pattern to deal with the consequences of the event.
- **Design complexity of the solution:** The effort necessary to incorporate the patterned solution in the overall design of a system.
- **Time overhead in the absence of fault/error/failure events:** The impact of the pattern (in terms of time to solution) on the fault-free operation of a system.
- **Time overhead to manage fault/error/failure events:** The impact on time to solution on account of the actions required to manage an event.
- **Space overhead of the solution:** The number of additional components or sub-systems that the solution requires.
- **Power overhead of the solution:** The impact of applying the pattern on the system's power consumption.

Our goal is to develop a classification scheme that facilitates the incorporation of resilience capabilities which are an essential part of the design process. The scheme should enable system designers to easily discover an appropriate pattern for a specific design problem. Therefore, we propose a classification that organizes the resilience design patterns in a layered hierarchy. Such an organization of the patterns helps designers to systematically evaluate the features of various patterns, and analyze the impact on coverage and overhead of applying a pattern to a specific problem. Each layer in the hierarchy, which is illustrated in Figure 3, enables designers to view the design problem at a different granularity. This permits the different stakeholders to reason about resilience capabilities based on their view of the system and their core expertise - system architects may analyze protection coverage for the various hardware and software components that make up the system based on the patterns applied by each component; the designers of individual components may operate within a single layer of system abstraction and focus on instantiation of patterns based on local constraints, without the need to understand the overall system organization. Each

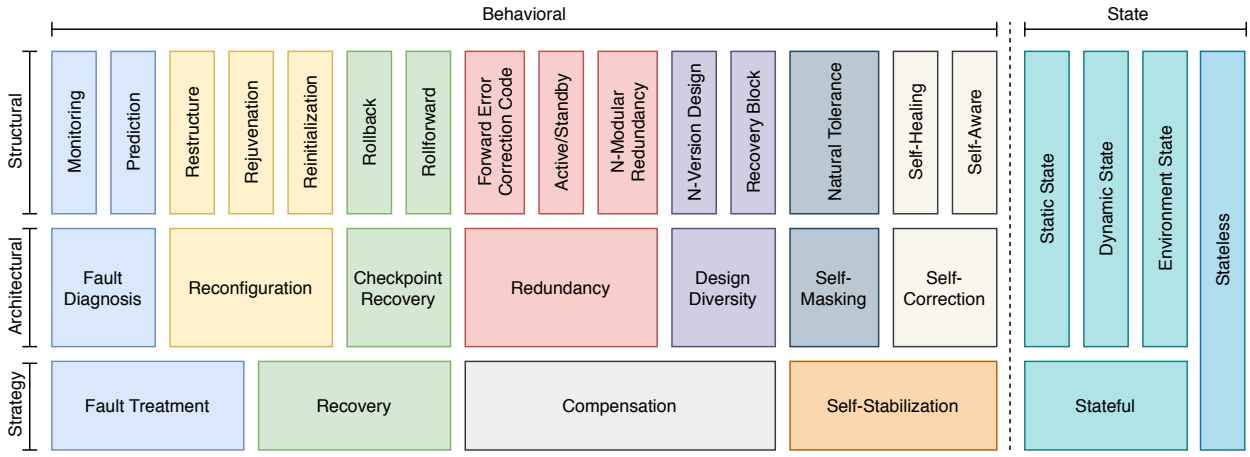


Figure 3. Classification of resilience design patterns

level in the hierarchy also addresses one or more of the design considerations, including the type of event handled, potential impact on a system, the capabilities supported by a pattern, design complexity, etc. Therefore, the ordering of the patterns within a level in the hierarchy is based on these considerations.

Resilience in the context of HPC systems and applications has two key dimensions: (1) forward progress of the system and (2) data consistency in the system. Based on these two aspects, the resilience design patterns are broadly classified into:

- **State Patterns:** These patterns describe all aspects of the system structure that are relevant to the forward progress of the system. The correctness and consistency of the system state ensures the correct operation of the system. The state patterns implicitly define the scope of the protection domain that must be covered by a resilience mechanism.
- **Behavioral Patterns:** These design patterns identify common detection, containment, or mitigation actions that enable the components in a system that realize these patterns to cope with the presence of a fault, error, or failure event.

In the classification scheme in Figure 3 the state patterns are placed separately from the behavioral patterns. This classification enables designers to separately reason about scope of the protection domain and the semantics of the pattern behavior.

6.1 STATE PATTERNS

The notion of state for an HPC application may be classified into three aspects [113]:

- *Persistent/Static State*, which represents the data that is computed once in the initialization phase of the application and is unchanged thereafter.
- *Volatile/Dynamic State*, which includes all the system state whose value may change during the computation.
- *Operating Environment State*, which includes the data needed to perform the computation, i.e., the program code, environment variables, libraries, etc.

Within the layer of state patterns, the patterns are organized to capture each of these aspects of state. This organization enables the behavioral patterns to be applied to individual aspects of the system state.

However, the state patterns may also be fused to enable the application of a single behavioral pattern to more than one state pattern. Certain resilience behaviors may be applied without regard for state; such patterns are concerned with only the forward progress of the system. Therefore, the classification of state patterns also includes a *stateless* pattern to enable designers to create solutions that define behavior without state.

6.2 BEHAVIORAL PATTERNS

The behavioral patterns are presented in a layered hierarchy to provide designers with the flexibility to organize the patterns in well-defined and effective solutions:

6.2.1 Strategy Patterns

These patterns define high-level policies of a resilience solution. Their descriptions are deliberately abstract to enable hardware and software architects to reason about the overall organization of the techniques used and their implications on the full system design. However, these patterns describe the overall structure of each pattern and the key components in the solution and their capabilities independent of the layer of system stack and hardware/software architectural features.

The strategy patterns are organized by the type of event that they intend to handle - fault, error or failure; the techniques to handle these events are fundamentally different. Without delving into the specifics of the precise type of fault, error, or failure and their impact on a system, these patterns describe the overall plan for handling each event type.

The classification of the strategy patterns also captures the intent behind each solution. The fault treatment patterns are concerned with diagnosing and preventing an imminent error or failure. The recovery and compensation patterns must limit and remove an error or failure state in the system. The recovery pattern aims to substitute an error/failure-free state in place of the erroneous/failed system state. The compensation pattern seeks to tolerate the presence of an error or failure by providing redundancy in the system.

6.2.2 Architectural Patterns

These patterns convey specific methods necessary for the construction of a resilience solution. They explicitly convey the type of fault/error/failure event that they handle and provide detail about the key components and connectors that make up the solution. Since the architecture patterns are a sub-class to the strategy patterns, they are also organized by the type of event they handle.

The architectural patterns are also organized to reflect the specific action that its solution takes to handle the event, and the intended impact of the action on the system resilience. Consequently, there exists an overlap between the sub-class relationships for one of the patterns.

The classification of these architectural patterns based on the core solution is also suggestive of the design time complexity and runtime complexity of a pattern. However, architectural patterns are independent of the precise fault model and may be implemented at any layer of the system stack.

6.2.3 Structural Patterns

These patterns provide concrete descriptions of the solution rather than high-level strategies. They comprise of instructions that may be implemented in hardware/software components. While the strategy and architectural patterns serve to provide designers with a clear overall framework of a solution and the type of events that it can handle, the structural patterns express the details so they can contribute to the

development of complete working solutions. Structural patterns provide a concrete description of the components that make up the solution.

The structural patterns are also sub-classes of the strategy and architectural patterns. Therefore, their first-order organization is also based on the type of fault event that their solution handles. However, the pattern descriptions include the details of the fault model that the pattern protects the system against. Although the structural patterns provide more detailed solutions, their descriptions are still independent of the layer of system abstraction. However, based on the description of the modules, a designer may be able to identify the layer of system stack at which the pattern may potentially be instantiated.

6.2.4 Implementation Patterns

These patterns bridge the gap between the design principles and the concrete details required for realization of a complete resilience solution. These are often compound patterns, i.e., patterns of patterns, and consist of a structure pattern and a state pattern. The implementation patterns explicitly specify the layer of system abstraction at which they are implemented, and the activation and response interfaces. These patterns enable the designer to understand the overhead of a solution in terms of time and space, as well as the trade-off between design complexity and runtime complexity.

7. CATALOG OF RESILIENCE DESIGN PATTERNS

The resilience solutions presented in this document have been extensively studied and/or used. The primary objective is to capture them in a standardized and accessible format. The resilience design patterns are based on practical solutions that have been or can be applied to HPC systems and their applications. Each design pattern focuses on a specific problem in HPC resilience.

7.1 DESCRIBING DESIGN PATTERNS

Patterns are expressed in a written form and in a highly structured format to enable HPC architects and designers to quickly discover whether the pattern is relevant to the problem being solved. Every pattern has the following key attributes [82]:

- A descriptive **name** that distinctly identifies the pattern and enables designers to think about designs in an abstract manner and communicate their design choices to others.
- A description of the **problem** that provides insight on when it is appropriate to apply the pattern. Multiple patterns may address the same problem differently.
- A description of the **solution** that defines the abstract elements that are necessary for the composition of the design solution as well as their relationships, responsibilities, and collaborations.
- The **consequences** of applying the pattern in terms of the protection coverage that the pattern offers and the involved resource and execution time trade-offs.

For convenience and clarity, each resilience pattern presented in the catalog follows the same prescribed format. There are three key reasons behind this pattern format: (1) to present the pattern solution in a manner that simplifies comparison of the capabilities of patterns and their use in developing complete resilience solutions, (2) to present the solution in a sufficiently abstract manner that designers may modify the solution depending on the context and other optimization parameters, and (3) to enable these patterns to be instantiated at different layers in the system.

Defining a catalog of resilience design patterns provides reusable solutions to specific problems in a way that they may be instantiated in various ways, in hardware and software. Hardware design covers design of microarchitecture blocks, processor architecture, memory hierarchy design, network interface design, as well as design of racks, cabinet and system-level design. The scope of software design spans the spectrum of OSs, runtimes for scheduling, memory management, communication frameworks, performance monitoring tools, computational libraries, compilers, programming languages, and application frameworks.

In order to make the resilience pattern relevant to these diverse domains of computer system design, we describe solutions in a generic manner. The descriptions use the term *system* to refer to an entity that has the notion of a well-defined structure and behavior. A *subsystem* is a set of elements, which is a system itself, and is a component of a larger system, i.e., a system is composed of multiple subsystems or components. For an HPC system architect, the scope of system may include compute nodes, I/O nodes, network interfaces, disks, etc., while an application developer may refer to a library interface, a function, or even a single variable as a system. The instantiation of the pattern interprets the notion of *system* to refer to any of these hardware or software-level components. A *full system* refers to the HPC system as a whole, or to a collection of nodes that is capable of running a parallel application. The resilience design pattern description format is as follows:

Name: Identifies the pattern and provides a convenient way to refer to it, typically using a short phrase.

Problem: A description of the problem indicating the intent behind applying the pattern. This describes the intended goals and objectives that will be accomplished with the use of this specific pattern.

Context: The preconditions under which the pattern is relevant, including a description of the system before the pattern is applied.

Forces: A description of the relevant forces and constraints, and how they interact or conflict with each other and with the intended goals and objectives. This description clarifies the intricacies of the problem and makes explicit the trade-offs that must be considered.

Solution: A description of the solution that includes specifics of how to achieve the intended goals and objectives. This description identifies the core structure of the solution and its dynamic behavior, including its collaborations with other patterns. The description may include guidelines for implementing the solution as well as descriptions of variants or specializations of the solution.

Capability: The resilience management capabilities provided by this pattern, which may include detection, containment, mitigation, or a combination of these capabilities. The listing of capabilities enables designers to determine whether other patterns must be employed to compose a complete resilience solution.

Protection Domain: The resiliency behavior provided by the pattern extends over a certain scope, which may not always be explicit. Also, a solution may be suitable for a specific fault model. The description of scope and nature of fault model that is supported by the pattern enables designers to reason about the coverage scope in terms of the complete system.

Resulting Context: A brief description of the post-conditions arising from the application of the pattern. There may be trade-offs between competing optimization parameters that arise due to the implementation of a solution using this pattern. The resulting context describes what aspects of the systems have been provided with protection, and which remain unprotected. This indicates other patterns that may be applied for supplementing the protection domain.

Examples: One or more sample applications of the pattern, which illustrate the use of the pattern for a specific problem, the context, and set of forces. This also includes a description of how the pattern is applied and the resulting context.

Rationale: An explanation of the pattern as a whole, with an elaborate description of how the pattern actually works for specific situations. This provides insight into the internal workings of a resilience pattern, including details on how the pattern accomplishes the desired goals and objectives.

Related Patterns: The relationships between this pattern and other relevant patterns. These patterns may be predecessor or successor patterns in the hierarchical classification. The pattern may collaborate to complement or enhance the resilience capabilities of other patterns. There may also be dependency relationships between patterns, which necessitate the use of co-dependent patterns in order to develop complete resilience solutions.

Known Uses: Known applications of the pattern in existing HPC systems, including any practical considerations and limitations that arise due to the use of the pattern at scale in production HPC environments.

7.2 STRATEGY PATTERNS

7.2.1 Fault Treatment

Name: Fault Treatment

Problem: An incomplete understanding of the presence, root cause and impact of a defect or anomaly in the system causes lack of corrective action at design or run time and may eventually result in an error or failure.

Context: The pattern applies to a system that has the following characteristics:

- The system has well-defined parameters that enable a monitoring system to discover the presence of a defect or anomaly in the behavior of the monitored system.
- The interaction between the monitored and the monitoring systems is bounded in terms of time.
- The monitoring system has the capability to notify the monitored system, which enables the anomaly or defect to be removed before or after it results in an error or a failure.

Forces:

- The interactions between the monitoring and monitored systems may interfere with the operation of the monitored system, specifically its performance.
- While the frequency and duration of these interactions must be minimized to reduce the monitoring overhead, the interactions must be able to detect/infer a defect or anomaly in the monitored system.
- The monitoring system's data gathering and defect/anomaly detection/inference must be in time to prevent the activation of an error and a subsequent failure.

Solution: The `Fault Treatment` strategy pattern enables the discovery and treatment of a defect or anomaly in the system, which either has the potential to activate or already has activated to an error and potentially a subsequent failure. The pattern supports methods that attempt to recognize the presence of a defect or anomaly within a monitored system. It prevents its activation or enables its containment and mitigation by notifying the monitored system, such that the anomaly or defect is removed before or after it results in an error or a failure. The solution requires a monitoring system, which may be a subsystem of the monitored system or an external independent system, to observe key parameters of the monitored system and to notify the monitored system when these parameters deviate. The components of this pattern are illustrated in Figure 4.

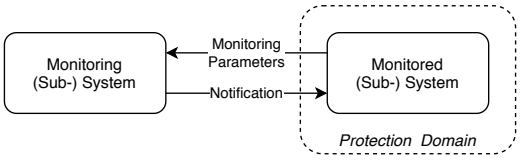
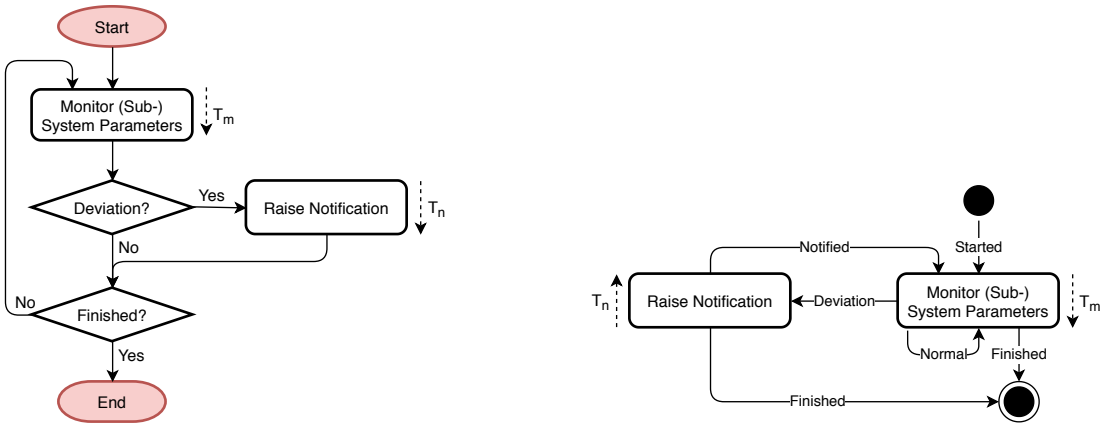


Figure 4. Fault Treatment pattern components

Capability: This pattern provides fault detection in the monitored system at design or run time, before it eventually results in an error or failure or after activation, by identifying deviations in monitored parameters. This pattern enables containment and mitigation of a future, imminent or present error or failure in the monitored system through detection and by notifying the monitored system. The flowchart and state diagram of the pattern is shown in Figure 5 and its parameters in Table 2.

Protection Domain: The protection domain extends to the monitored system.



(a) Flowchart **(b) State diagram**

Figure 5. Fault Treatment pattern flowchart and state diagram

Table 2. Fault Treatment pattern parameters

Parameter	Definition
T_m	Time to monitor (sub-) system parameters, including wait and probe times
T_n	Time to raise notification

Resulting Context: The discovery and treatment of a defect or anomaly in the system that has the potential to activate or has activated is enabled, which can be used to prevent, contain and mitigate an error or a failure in the system. The `Fault Treatment` strategy pattern requires identifying system parameters that indicate the presence of a fault. The overall system design must include a monitoring system, which introduces design complexity. When the monitoring system is extrinsic to the monitored system, the design effort may be simplified, but the interfaces between the monitoring and monitored systems must be well-defined. When the monitoring system is intrinsic to the design of the monitored system, design complexity increases due to the need to interface the monitoring and monitored subsystems.

A trade-off exists between interference with the operation of the monitored system caused by the frequency and duration of the interactions between the monitoring and monitored systems, and the ability to detect/infer a defect or anomaly in the monitored system. A high performing solution reduces the interference, while maintaining a reasonable ability to detect/infer a defect or anomaly.

The `Fault Treatment` strategy pattern may be used in conjunction with other strategy patterns that provide containment and mitigation and require or can make use of fault detection. It detects the presence of a defect or anomaly and reports it, but does act to remedy the fault.

Examples:

- Hardware sensors in the central processing units (CPUs), on the board, and in the power supply provide real-time operational data about temperature, fan speeds, and voltages. Software sensors in the Linux OS offer real-time operational data about CPU and memory load. The data is made available by sensor interface tools, such as Intelligent Platform Management Interface (IPMI) [10], such that a monitoring tool, like the Ganglia Monitoring System [123], can aggregate it with time stamps in a central database. The monitoring tool can be configured to send a notification when a certain threshold value for a sensor is exceeded.
- A heartbeat monitoring tool checks the liveness of an MPI process in the system by regularly sending a

message and waiting for a reply [147]. If no message is received back within a certain time period, the MPI process is considered failed and the MPI programming model runtime software is informed. This heartbeat monitoring is performed between all MPI processes, where all MPI processes watch each other. A message sent by the application can be used as a liveness request or reply message.
- An MPI tracing tool, like Vampir [8], is used to capture detailed time-stamped information about an application's MPI messages at execution time. The created MPI message trace contains time-stamped information about when an MPI function was called and when it was completed, such that it can be analyzed for inefficiencies in the MPI application's messaging behavior. A common root cause for MPI application scaling limitations is the inefficient use of MPI collectives. The MPI application developer is informed by the tool about such situations.
- In proactive fault tolerance, an observe, orient, decide, and act (OODA) loop control is employed that utilizes monitoring tools for collecting sensor data (e.g., temperature, fan speeds, voltages, computational load, memory and storage usage, etc.). It leverages the warning thresholds of these sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72]. The mitigation may use process-level [187] or virtual machine (VM) level [134] migration.
- Monitoring tools collecting event data (e.g., anomalous, error and failure events, debug messages, etc.) are used in conjunction with tools for temporal and spatial filtering to identify event correlations and to predict failures [116].
- Probabilistic networks are utilized for establishing correlations between event collected with monitoring tools to predict failures [159].

Rationale: The `Fault Treatment` strategy pattern enables a system to discover and treat a defect or anomaly in the system that has the potential to activate and become an error or failure or already has activated. It relies on a monitoring system to observe the monitored system, identifying deviations in monitored parameters. A discovered defect or anomaly is treated by raising a notification, permitting the monitored system to provide containment and mitigation. A key benefit of this pattern is preemptive discovery of faults in the system, before they are activated and result in errors or failures. Preventive actions taken upon such discovery avoid the need for expensive error/failure recovery, compensation, or correction actions.

Related Patterns: Unlike the other strategy patterns, the `Fault Treatment` strategy pattern does not provide error or failure containment or mitigation. However, it enables error and failure containment and mitigation. Since the `Recovery` strategy pattern does not offer error or failure detection, it may be used in conjunction with the `Fault Treatment` strategy for a complete solution. The `Compensation` and `Self-Stabilization` strategy patterns do offer error and failure detection, but their capabilities can be enhanced by additionally employing the `Fault Treatment` strategy pattern.

Known Uses: HPC system installations use monitoring tools for collecting operational data, such as sensor data (e.g., heat, fan speeds, voltages, computational load, memory and storage usage, etc.), performance data (e.g., application execution, message and file system access times, etc.) and event data (e.g., anomalous, error and failure events, debug messages, etc.). Popular solutions include:

- Hard- and software sensors in CPUs, on the board, in the power supply, in the OS, and in system services that provide real-time operational data to sensor interface tools. Examples are the hardware sensors in the IBM Power 8 [94] and Intel Xeon [54] processors, and the Linux OS monitoring capabilities.
- Sensor interface tools that gather real-time operational data from hard- and software sensors, such that they are available for monitoring tools. Examples are self-monitoring and reporting technology (SMART) [173] and IPMI [10].
- Monitoring tools that gather time-stamped information from sensor interface tools in a distributed computing system at a central location, such that they can be analyzed. Examples are the Supermon

High-Speed Cluster Monitoring System [171], the Ganglia Monitoring System [123], Nagios [4], and the OVIS Lightweight Distributed Monitoring System [14].
- Heartbeat monitoring tools for liveness checking of individual nodes in the system [147] or MPI processes [26].
- System event loggers that gather time-stamped information from the OS or system services in a distributed computing system, such that they can be analyzed. The most prominent example is Linux's syslog [6].
- MPI tracing tools gather detailed time-stamped information about an application's MPI messages, such as when a particular message was sent and when it was received. Examples are Vampir [8] and SCALASCA [7].
- Comprehensive RAS management tools by vendors that support real-time monitoring of an entire HPC system, such as the Cray RAS and Management Subsystem (CRMS) [28].

The collected operational data may be analyzed by advanced tools for predicting future behavior based on experiences with past behavior and/or based on current behavior, such as for predicting faults, errors and failures. Known uses include:

- Utilizing a wide-variety of methods to analyze monitoring and log data to predict failures [105]
- Using machine learning to predict general-purpose computing graphics processing unit (GPGPU) memory errors [139]
- Leveraging warning thresholds of sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72]
- Characterizing the correlation between temperature, power consumption and memory or logic errors using monitoring and log data [138]
- Using probabilistic networks for establishing event correlations for prediction algorithms [159]
- Employing temporal and spatial filtering for failure prediction [116]

7.2.2 Recovery

Name: Recovery

Problem: A hardware or software error or subsystem failure due to a physical fault (e.g., wear-out or destruction) or a design fault (e.g., resource underprovisioning) in an HPC environment causes a software, such as an application, to experience an error and potentially a subsequent failure.

Context: The pattern applies to a system with the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The error or failure in the system that the pattern handles must be detected, as the pattern offers no error or failure detection.
- The system is capable of compartmentalizing its state that is accurately representative of the progress of the system since initialization at the time such a state is captured.

Forces:

- The pattern requires additional persistent storage to capture system state, which increases overhead in terms of resources required by the system.
- The amount of state captured during each creation of a recovery point incurs storage space and execution time overheads.
- The creation frequency of system state snapshots determines overhead. More frequent snapshot creation increases system execution time, but reduces the amount of lost work upon an error or failure.
- The time interval for the recovery of a system from a snapshot as well as the time interval to create a snapshot must be less than the system's MTBF to guarantee forward progress.

Solution: The `Recovery` strategy pattern enables the resumption of correct operation of a system impacted by an error or failure. It supports resilient operation by restoring the system to a known correct state in the event of an error or failure. The pattern relies on the creation of system state snapshots and the maintenance of such snapshots on a persistent storage system that is not affected by the error or failure. Upon detection of an error or failure, the most recent snapshot is used to recreate the last known error/failure-free state of the system. When the system state is recovered, the operation of the system is resumed. Based on a temporal view of the system's progress, the error/failure recovery may be either backward to the time when the snapshot occurred (rollback) or forward to the time when the error/failure event occurred (rollforward).

Undetected (latent) errors that are either detected later or result in a different detected error or failure later represent a problem, as the most recent or even more snapshots may contain an illegal system state. In this case, the most recent correct snapshot may be used to recreate the last known error/failure-free state of the system, skipping snapshots containing illegal state and going further back in time in terms of when the snapshot was made. The components of this pattern are illustrated in Figure 6.

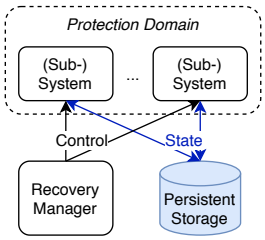
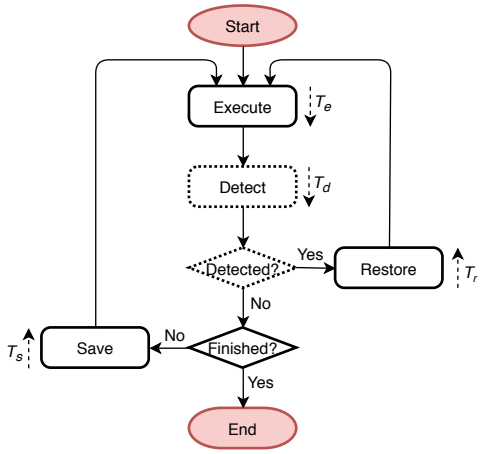
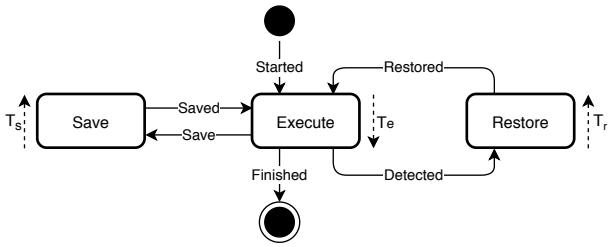


Figure 6. Recovery pattern components

Capability: This pattern provides mitigation of an error or failure in the system by preserving system state before such an event, often in a periodic fashion, and restoring the previously preserved system state after such an event to resume operation from a known correct state. The flowchart and state diagram of the pattern is shown in Figure 7 and its parameters in Table 3.



(a) Flowchart



(b) State diagram

Figure 7. Recovery pattern flowchart and state diagram

Table 3. Recovery pattern parameters

Parameter	Definition
T_e	Time to execute (sub-) system progress
T_d	Time to detect an error/failure (not part of this pattern, but shown for completeness)
T_r	Time to restore (sub-) system state and progress (initial and/or previously saved)
T_s	Time to save (sub-) system state and progress

Protection Domain: The protection domain extends to the system state that can be restored using the system state captured by snapshots stored on persistent storage.

Resulting Context: Correct operation is resumed after an error or failure impacted the system. Progress in the system is lost after an error or failure if the recovery is only able to recreate the system state to the time when the snapshot occurred. Progress is not lost if the recovery is able to recreate the system state to the time when the error/failure occurred. The system is interrupted during error-/failure-free operation for creating system state snapshots and maintaining them on a persistent storage system. The system is interrupted upon an error or failure for recreating the last known error/failure-free state of the system.

A trade-off exists between the creation frequency of system state snapshots and its corresponding execution time overhead during error-/failure-free operation vs. the amount of lost progress in case of a rollback or

the amount of work required to recreate the state in the case of a rollforward. The optimal solution of this trade-off depends on the MTBF, the time it takes to save and load a system state snapshot and the time it takes to recreate the system state.

The `Recovery` strategy pattern does not provide error or failure detection and may be used in conjunction with the `Fault Treatment` strategy pattern to be fully functional. The `Recovery` strategy pattern may be used in conjunction with other strategy patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other strategy pattern(s) and the `Recovery` strategy pattern covers for the remaining error/failure types.

Examples: Many HPC applications implement application-level checkpoint recovery by regularly saving intermediate results to stable storage as checkpoints and supporting the capability to load such a checkpoint in addition to some or all of the original data upon restart to recreate the last known correct state. Library-based solutions, such as Fault Tolerance Interface (FTI) [27], permit tracking of state that needs to be saved and restored. System-level solutions, such asDistributed MultiThreaded CheckPointing (DMTCP) [18], support transparent state saving and restoration using OS support. GVR [47] is a runtime system that provides fault tolerance to applications by versioning distributed arrays for checkpoint recovery, while the checkpoint-on-failure protocol [18] for MPI applications leverages the features of a high-quality fault-tolerant MPI implementation. In either case, algorithm-specific knowledge is needed to perform checkpoint recovery, Some ABFT solutions [122] can utilize the original or previously saved data as a replacement for lost or erroneous data and recover their state to the point at which the error/failure event occurred.

Rationale: The `Recovery` strategy pattern enables a system to tolerate an error or failure through resumption of correct operation after impact. It relies on the capability to preserve system state before an error or failure, often in a periodic fashion, and restore the previously preserved system state upon such an event to resume operation from a known correct state. The pattern performs proactive actions, such as preserving system state, but mostly relies on reactive actions after an error or failure impacted the system.

Error or failure detection is not part of the pattern. The preserved system state is managed on persistent storage, which is not part of the protection domain. The containment and mitigation offered by this pattern are independent from the type of error or failure. The pattern has very little design complexity and has low dependence on a system's architecture, which makes it appealing as a general and portable solution.

Related Patterns: The `Recovery`, `Compensation`, and `Self-Stabilization` strategy patterns provide containment and mitigation using different approaches. The `Recovery` strategy pattern relies on system state snapshots and may experience a loss of progress, but has low design complexity. The `Compensation` strategy pattern relies on redundancy of system state and optionally on system resources and experiences no loss of progress, but has high design complexity. The `Self-Stabilization` strategy pattern relies on the capability of reaching a correct system state from an illegal system state after a finite number of execution steps and guarantees forward progress, but has high design complexity.

Known Uses:

- The Mini-Ckpts framework [79] enables HPC applications to survive OS errors and failures. The OS is effectively rebooted, while the application and OS state directly relevant to the application is preserved in persistent memory. This rejuvenates all OS state that is not directly relevant to the application.
- Individual services, such as the parallel file system metadata service (MDS) or the system's resource manager, may be rebooted to fix erroneous behavior or failures. The correct service state is maintained on persistent storage to allow for resumption after reboot.
- Many HPC applications implement application-level checkpoint recovery by regularly saving

intermediate results to stable storage as checkpoints and supporting the capability to load such a checkpoint in addition to some or all of the original data upon restart to recreate the last known correct state.

- FTI [27] is a library-based checkpoint recovery solution, tracking, saving and restoring data structures in multilevel checkpoint storage.
- The CoCheck supports checkpoint recovery for an MPI implementation; for the coordination of the checkpoints, CoCheck uses a special process [172]. The Condor checkpoint/restart facility is enabled by the user by linking the program source code with the condor library [121].
- The libckpt [148] is a user-level, library-based checkpointing solution that supports explicit directives to determine the scope of an application's checkpointed state.
- The SCR [130] library uses multilevel checkpointing; it creates frequent inexpensive checkpoints that can recover the loss of a few nodes to the local node-level storage and writes complete checkpoints that can withstand an entire system failure to the parallel file system.
- BLCR is a process-level checkpoint recovery module for the Linus OS. It has been used in conjunction with LAM/MPI to permit OS-level checkpoint recovery for MPI applications [184].
- Improvements to system-level checkpoint recovery solutions, such as BLCR, include incremental checkpoint recovery, where only changes between two checkpoints are being saved and the recovery performs a reconstruction of correct state from multiple incremental checkpoints [185].
- DMTCP [18] is a transparent checkpoint recovery solution for multi-threaded and MPI applications.
- The local failure local recovery protocol uses checkpoint recovery of only some processes with locally stored checkpoints to create correct state [177].
- Message logging protocols have been implemented in OpenMPI to support faster failure recovery with message replay [35].
- GVR [47] is a runtime system that provides fault tolerance to applications by versioning distributed arrays. It supports checkpoint recovery based on application-specified mechanisms. Past versions of lost or erroneous data can be retrieved and used for computing the state at which the error/failure event occurred.
- The checkpoint-on-failure protocol [18] for MPI applications leverages the features of a high-quality fault-tolerant MPI implementation and algorithm-based checkpoint recovery. It provides the ability for all healthy processes to continue to operate and perform checkpoint recovery using message passing.
- Some ABFT solutions [122] can utilize the original or previously saved data as a replacement for lost or erroneous data and recover their state to the point at which the error/failure event occurred.

7.2.3 Compensation

Name: Compensation

Problem: A hardware or software error or subsystem failure due to a physical fault (e.g., wear-out or destruction) or a design fault (e.g., human mistake or defective design tool) in an HPC environment causes a software, such as a system service or an application, to experience an error and potentially a subsequent failure.

Context: The pattern applies to a system with the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has a modular design that has a well-defined scope and a set of inputs and outputs.

Forces:

- The pattern introduces an execution time and/or resource requirement (storage space, computational capability, etc.) penalty independent of whether an error or failure occurs during system operation or not.
- The scope and strength of the redundancy employed by the pattern determines its execution time and resource requirement overhead.

Solution: The `Compensation` strategy pattern enables the correct operation of a system impacted by an error or failure. It supports resilient operation by applying redundancy to system state and optionally to system resources. This redundancy may be in the form of encoded system state, functionally identical replicas, or functionally equivalent alternate system implementations. The pattern requires very well defined input and output to permit encoding/decoding of system state, input encoding/replication, and output decoding/comparison/validation.

Redundancy can be in time, meaning the same system resources are used for redundancy. Redundancy can also be in space, meaning additional (redundant) system resources are used. Redundancy in time saves system resources, while redundancy in space offers more error/failure coverage. A mix between redundancy in time and space is possible as well, where there is more encoded system state or there are more functionally identical replicas or functionally equivalent alternate system implementations than additional (redundant) system resources. The components of this pattern are illustrated in Figure 8.

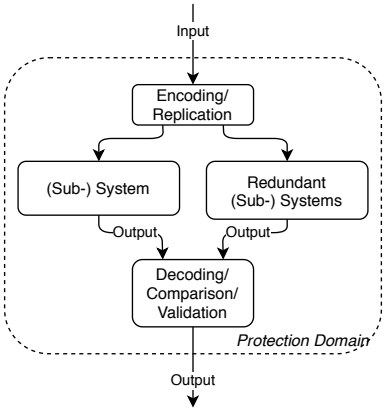
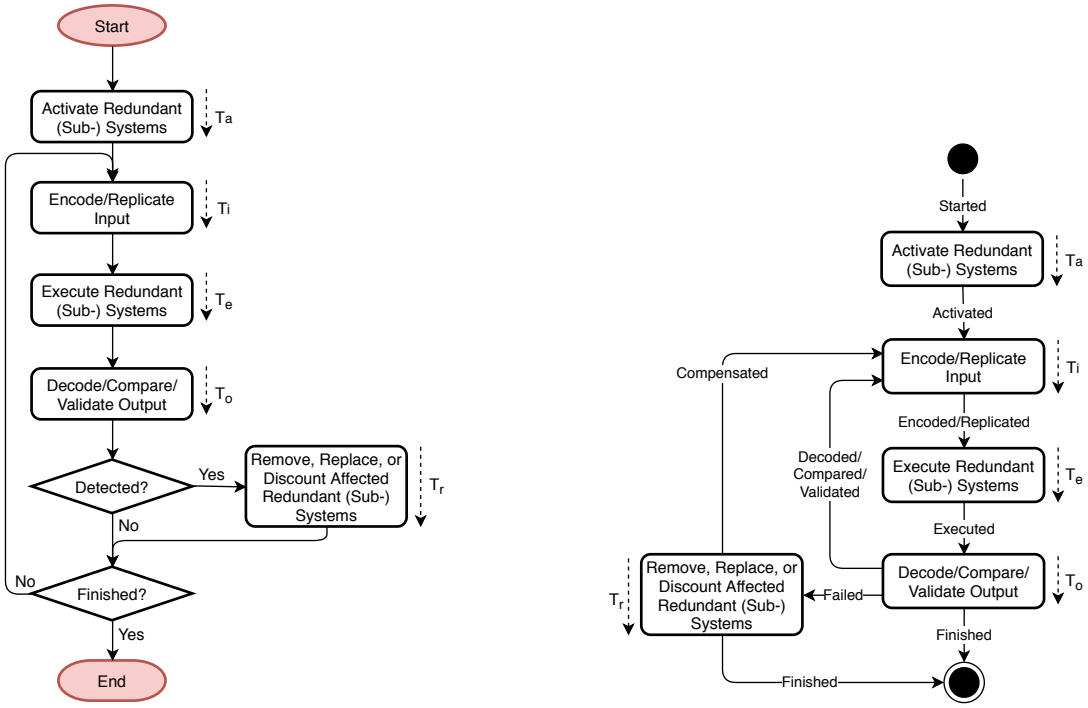


Figure 8. Compensation pattern components

Capability: This pattern provides error and/or failure detection in the system by applying redundancy to system state in the form of encoded system state, functionally identical replicas, or functionally equivalent alternate system implementations. The pattern provides mitigation of an error or failure in the system by applying redundancy to system state and optionally to system resources, such that the system continues to operate correctly in the presence of such an event. The flowchart and state diagram of the pattern is shown in Figure 9 and its parameters in Table 4.



(a) Flowchart **(b) State diagram**

Figure 9. Compensation pattern flowchart and state diagram

Table 4. Compensation pattern parameters

Parameter	Definition
T_a	Time to activate the redundant (sub-) systems
T_i	Time to encode/replicate the input to the redundant (sub-) systems
T_e	Time to execute redundant (sub-) system progress
T_o	Time to decode/compare/validate the output from the redundant (sub-) systems
T_r	Time to remove, replace or discount the affected redundant (sub-) systems

Protection Domain: The protection domain extends to (1) the encoded system state and to the system resources processing it, (2) the system state and the system resources that implement the N functionally identical replica systems, and (3) the system state and the system resources described by the design specification that implement the functionally equivalent alternate systems.

Resulting Context: Correct operation is performed despite an error or failure impacting the system. Progress in the system is not lost due to an error or failure. The system is not interrupted during error-/failure-free operation. The system may be interrupted when encountering an error or failure. Resource usage in time or space is increased according to the amount of redundancy employed in the form

of encoded system state, functionally identical replicas, or functionally equivalent alternate system implementations.

A trade-off exists between the amount of redundancy employed and the number of errors and/or failures that can be tolerated at the same time and/or in time. More redundancy generally tolerates more errors and/or failures, but requires either more resources or more execution time.

The `Compensation` strategy pattern may be used in conjunction with other strategy patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other strategy pattern(s) and the `Compensation` strategy pattern covers for the remaining error/failure types.

Examples: There are various schemes that enable forward error correction redundancy in memory devices, storage systems as well as communication channels. Based on time and space overhead constraints, schemes of different detection and correction capabilities are used. Popular examples include parity bits, checksums, Hamming codes, hash function codes. More elaborate schemes such as systematic cyclic block codes include binary BCH, Reed-Solomon and CRC. Forward error correction can be found in HPC storage systems with RAID, the InfiniBand interconnect [9], the memory hierarchy [131, 59], ABFT solutions [95] and coded computing [106]. Active/Standby redundancy is typically used for critical hardware or software systems in HPC environments. For example, power supplies, voltage regulators, the parallel file system MDS in Lustre [194] and the Simple Linux Utility for Resource Management (SLURM) [192] job and resource manager are often implemented in an active/standby fashion. Dual-modular redundancy for error detection and failure compensation and triple-modular redundancy for error detection and correction and failure compensation are used n HPC environments as well. Examples include dual-redundant cooling fans, dual- and triple–modular redundant MPI implementations [78], dual-redundant parallel file system MDS solutions [92] and dual-redundant mission-critical HPC systems (e.g., weather forecast).

Various versions of the same software are used in HPC systems for the detection of implementation errors. This applies to completely different implementations of the MPI standard and to numerical libraries as well as to different versions of the same implementation. Regression and comparison tests are performed to identify incorrect behavior, missing features and performance problems. Containment Domains [48] and the SWIFT library [97] provide language-based approaches for recovery blocks with design diversity. Applications also often contain verification routines that check for the validity of a computation and correct any detected errors using application-specific knowledge and design diversity.

Rationale: The `Compensation` strategy pattern enables a system to tolerate an error or failure through continuation of correct operation after impact. It relies on system state redundancy in the form of encoded system state, functionally identical replicas, or functionally equivalent alternate system implementations. The pattern performs mostly proactive actions, such as maintaining redundancy. Error or failure detection is part of the pattern in the form of output decoding, comparison, or validation. The pattern has high design complexity, as input needs to be encoded/replicated, processed encoded, and output needs to be decoded/compared. The need for functionally equivalent alternate system implementations additionally increases the design complexity.

Related Patterns: The `Recovery`, `Compensation`, and `Self-Stabilization` strategy patterns provide containment and mitigation using different approaches. The `Recovery` strategy pattern relies on system state snapshots and may experience a loss of progress, but has low design complexity. The `Compensation` strategy pattern relies on redundancy of system state and optionally on system resources and experiences no loss of progress, but has high design complexity. The `Self-Stabilization` strategy pattern relies on the capability of reaching a correct system state from an illegal system state after a finite number of execution steps and guarantees forward progress, but has high design complexity.

Known Uses:

- HPC storage systems use forward error correction redundancy in the form of RAID.
- InfiniBand, which is among the most widely deployed high-speed interconnect employs forward error correction redundancy in the InfiniBand devices, including adapters and switches, to fix bit errors throughout the network [9]. The forward error correction allows reduction in data re-transmission between the end-nodes.
- HPC systems use memory DIMMs that employ ECC, which is based on forward error correction redundancy by maintaining redundant bits per memory line to support SECDED for bit flips [131].
- Chipkill [59] uses forward error correction redundancy across memory chips atop ECC to perform SECDED at a symbol-level granularity.
- Some ABFT methods use schemes, such as checksums, that perform forward error correction redundancy at the application-level [95].
- Coded computing uses algorithmic forward error correction redundancy. A recent solution used coded computing for parallel matrix-matrix multiplication [106].
- Production HPC systems such as the Cray XC40 series [103] include active/standby redundant power supplies and voltage regulator modules to ensure continuous operation in the event that one of these units experiences malfunction or failure.
- Active/standby (cold, warm and hot) redundancy using shared storage between active and standby systems is a common technique. The shared storage may be a dual-hosted hard drive, a networked storage using RAID, or a distributed replicated block device (DRBD) [154]. Such solutions have been extensively used in HPC environments for critical system services, such as the job and resource manager (e.g., SLURM [192] and Sun Grid Engine (SGE) [175]) and the parallel file system MDS (e.g., Parallel Virtual File System (PVFS) [146] and Lustre [194]).
- Active/hot-standby redundancy using a commit protocol for state replication has been implemented for some HPC job and resource managers as part of high availability cluster solutions, such as HA-OSCAR [119] with its commit protocol for OpenPBS [16].
- High availability clustering is a form of active/hot-standby redundancy with n active systems and m standby systems that are set up in an $n + m$ configuration. It targets high throughput processing of a large number of small service requests with no or minimal state changes, where the active systems respond to service requests. High availability clustering may use shared storage, state replication or state separation. In contrast to shared storage and state replication, where systems have the same state, state separation splits the state space among active systems, such as two parallel file system MDSs serving two different file system directories. An implementation of HA-OSCAR supported high availability clustering for two job and resource managers, OpenPBS [16] and SGE [175]). Parallel file system MDSs, such as Lustre [194], support high availability clustering as well.
- Active/standby redundancy also plays a role in resilience for parallel applications in HPC environments. Starting a parallel application with additional spare compute nodes enables its reconfiguration and the replacement of a failed compute node without completely restarting the application [21].
- Production HPC systems such as the Cray XC40 series [103] include n-modular redundant cooling fans to ensure continuous operation in the event that one of these units experiences malfunction or failure.
- Some implementations of the MPI standard use n-modular redundancy for detection and correction of errors by replicating the MPI messages, or even by replicating MPI processes. The MR-MPI [70], rMPI [76], and RedMPI [78] prototypes are known examples for this n-modular redundancy approach. N-modular redundancy for compute nodes in a system have been evaluated and shown to improve the overall availability of a HPC system [71].
- Some n-modular redundancy implementations for service nodes in HPC systems exist as well, such as for the MDS of PVFS [92] and for HPC job and resource managers that are compliant with the Portable

Batch System (PBS) [178].

- N-modular redundancy at the whole system level is often implemented for mission critical HPC systems, such as for weather forecast. In this case, two completely redundant HPC systems perform exactly the same computation.
- HPC centers often provide various MPI library implementations, such as the MVAPICH2 [3], OpenMPI [5], MPICH2 [2], and Intel MPI [1], all of which are based on the MPI standard. Running large-scale applications with these separate implementations of MPI potentially exposes implementation errors in the MPI libraries. Similarly, different versions of numerical libraries are often provided as well. Regression and comparison tests are performed to identify incorrect behavior, missing features and performance problems.
- The DIVA processor architecture [23] includes an out-of-order core as well as a simple in-order pipelined core. The in-order pipeline is functionally equivalent to the primary processor core (it implements the same instruction set architecture) and is used to detect errors in the design of the out-of-order processor core.
- Containment Domains [48] provide a recover routine with design diversity that is initiated upon detection of an error in the execution of the block of code encapsulated by the domain. This enables the containment domain to constrain the detection and correction of errors to the boundary of the domain.
- The SWIFT library [97] provides language-based implementation of a recovery block with design diversity for use in C language programs.
- Applications often contain verification routines that check for the validity of a computation. Application-specific knowledge is used in a recovery block to correct any detected errors.

7.2.4 Self-Stabilization

Name: Self-Stabilization

Problem: A hardware or software error or subsystem failure in an HPC environment causes a software, such as a numerical library or application, to experience an error and potentially a subsequent failure.

Context: The pattern applies to a system with the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.

Forces:

- System performance in the presence of errors and failures is determined by the efficiency in reaching a correct system state from an illegal system state, which may differ for different illegal system states.
- There is no inherent impact on system performance in error/failure-free conditions. However, the needed capability to reach a correct state from an illegal state may reduce error/failure-free performance.
- A correct state may not be reached from all illegal system states, which limits the scope of this pattern.

Solution: The `Self-Stabilization` strategy pattern relies on the capability of reaching a correct system state from an illegal system state after a finite number of execution steps. This can be achieved in multiple ways, such as by self-masking or self-correction. The illegal system state is identified by the pattern either implicitly and the incorrect state is self-masked, or explicitly and the incorrect state is self-corrected. In both cases, forward progress is guaranteed by design as a correct system state can be reached in a finite number of steps due to self-masking or self-correction to a correct state.

Self-masking may be as simple as an approximation of a correct state. Self-correction may be as simple as discarding, recomputing, or estimating a wrong value in the system or a wrong or missing output from a subsystem. The correct state reached after self-masking or self-correction is not necessarily the same state that would have been reached without an error or failure. The scope of this pattern extends to the system state space that is either correct or incorrect but self-maskable or self-correctable to a correct state. The components of the pattern are illustrated in Figure 10.

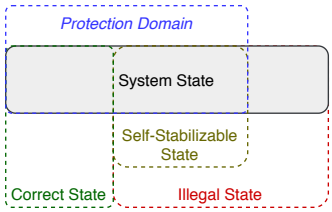
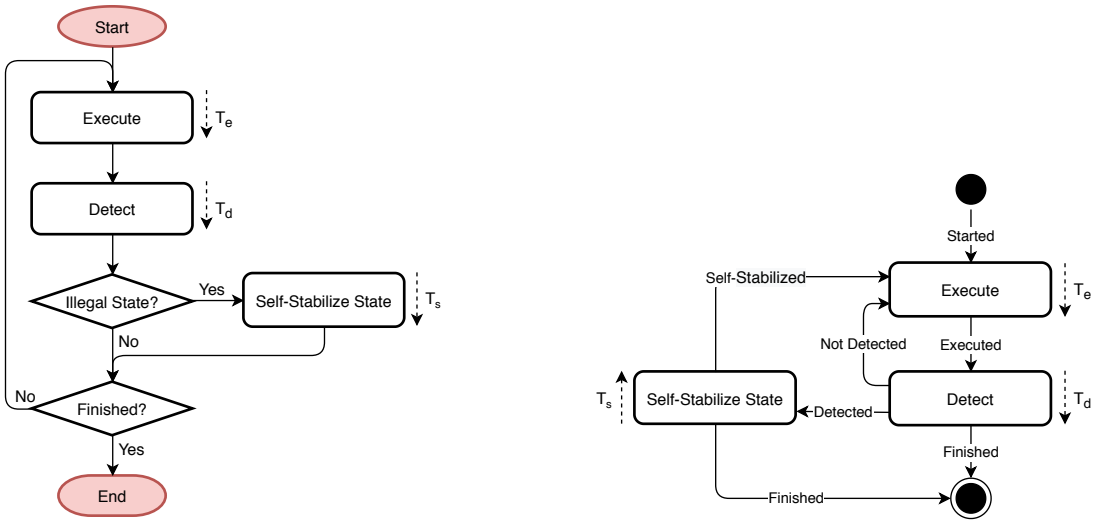


Figure 10. Self-Stabilization pattern components

Capability: This pattern supports the handling of errors and failures with detection, containment, and mitigation using self-stabilization, i.e., dynamic adaptation. Detection is either implicit by providing the capability to self-mask illegal system state or explicit by offering the ability to self-correct an illegal system state. Both, self-masking and self-correction are architectural features of the self-stabilization capability. The flowchart and state diagram of the pattern is shown in Figure 11 and its parameters in Table 5.

Protection Domain: The protection domain of this pattern extends to the system state space that is either correct or illegal but self-stabilizing to a correct state.



(a) Flowchart **(b) State diagram**

Figure 11. Self-Stabilization pattern flowchart and state diagram

Table 5. Self-Stabilization pattern parameters

Parameter	Definition
T_e	Time to execute system progress
T_d	Time to detect illegal system state
T_s	Time to self-stabilize illegal system state

Resulting Context: An error or failure is tolerated through adaptation. The `Self-Stabilization` strategy pattern requires the capability of reaching a correct system state from an illegal system state, which may reduce error/failure-free performance. The efficiency of self-stabilization determines performance in the presence of errors and failures. A high performing solution reduces the impact on error/failure-free performance, while at the same time providing efficient self-stabilization.

A trade-off exists between both, performance in the presence of errors and failures and error/failure-free performance, especially in systems with high error/failure rates. Self-stabilization may be performed by self-masking, a passive approach, or self-correction, an active approach. In both cases, implicit or explicit detection may reduce error/failure-free performance. The pattern's protection domain does not cover illegal system state that is not self-stabilizing.

The `Compensation` strategy pattern may be applied to extend the pattern's protection domain with redundant or diverse system state for self-correction. The `Self-Stabilization` strategy pattern may be used in conjunction with other strategy patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other strategy pattern(s) and the `Self-Stabilization` strategy pattern covers for the remaining error/failure types.

Examples:

- Gossip-based algorithms tolerate errors and failures using the `Self-Stabilization` strategy pattern. Data is communicated to multiple parts of the system participating in the Gossip-based algorithm and errors or failures are self-masked. The efficiency and performance of self-masking in the presence of errors and failures and the error/failure-free performance depend on the communication pattern (e.g.,

reach and frequency).

- Iterative solvers or solvers using sampling tolerate errors and failures using the `Self-Stabilization` strategy pattern using either an iterative process that progresses toward a correct solution in the presence of errors or a probabilistic sampling process that generates enough correct samples over incorrect samples to calculate a correct solution. The efficiency and performance of self-masking in the presence of errors and failures and the error/failure-free performance depend on the algorithmic capabilities of the iterative process (e.g., step size) or the probabilistic sampling process (e.g., oversampling).
- Self-stabilizing solvers may produce an illegal state in the presence of errors or failures and tolerate their impact using the `Self-Stabilization` strategy pattern to reach a correct state. The efficiency and performance of self-correction in the presence of errors and failures and the error/failure-free performance depend on the methods used for explicit error/failure detection (e.g., error check after every iteration) and for reaching the correct state (e.g., successive approximation toward a correct state).
- In proactive fault tolerance, an OODA loop control is employed that utilizes monitoring tools for collecting sensor data (e.g., temperature, fan speeds, voltages, computational load, memory and storage usage, etc.). It leverages the warning thresholds of these sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72]. The mitigation may use process-level [187] or VM level [134] migration.

Rationale: The `Self-Stabilization` strategy pattern enables a system to tolerate errors or subsystem errors or failures through adaptation. It relies on the capability of reaching a correct system system state from an illegal system state in a finite number of execution steps. The transition from an illegal to a correct system state uses self-masking or self-correction. Self-masking uses implicit error/failure detection and passively performs this transition. Self-correction uses explicit detection and performs the transition actively by correcting the illegal state. Self-masking or self-correction are intrinsic to the design of the system when this pattern is applied. The pattern has high design complexity and has low dependence on a system's architecture.

Related Patterns: The `Recovery`, `Compensation`, and `Self-Stabilization` strategy patterns provide containment and mitigation using different approaches. The `Recovery` strategy pattern relies on system state snapshots and may experience a loss of progress, but has low design complexity. The `Compensation` strategy pattern relies on redundancy of system state and optionally on system resources and experiences no loss of progress, but has high design complexity. The `Self-Stabilization` strategy pattern relies on the capability of reaching a correct system state from an illegal system state after a finite number of execution steps and guarantees forward progress, but has high design complexity.

Like the `Compensation` and `Recovery` strategy patterns, the `Self-Stabilization` strategy pattern seeks to mask or correct system state when using self-correction. However, it does not rely on state saving or redundancy for masking or correction. The `Compensation` strategy pattern may be applied to extend the `Self-Stabilization` strategy pattern's protection domain.

Known Uses:

- Gossip-based algorithms tolerate errors and failures using the `Self-Stabilization` pattern. This includes gossip-based aggregation and reduction algorithms [43, 140, 83], orthogonalization methods [84, 83], eigensolvers [174], and least squares solvers [151].
- Fixed point methods that converge globally when certain conditions are satisfied are able to tolerate certain errors using the `Self-Stabilization` pattern [20, 19], such as a Jacobi iterative scheme that will converge for any initial guess if the matrix is diagonally dominant.
- Certain domain decomposition methods are tolerant to errors using the `Self-Stabilization` pattern as they converge in the presence of errors, such as the classical overlapping Schwarz algorithm [86],

asynchronous domain decomposition methods [85] and task-based domain decomposition approaches that use sampling and a regression-based solution update [157, 156, 132].

- A self-stabilizing label-propagation algorithm that computes the connected components in a graph while being tolerant to errors [161].
- Two self-stabilizing iterative linear solvers [161], one for the steepest descent and one for conjugate gradient, are tolerant to errors.
- Leveraging warning thresholds of sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72].

7.3 ARCHITECTURAL PATTERNS

7.3.1 Fault Diagnosis

Name: Fault Diagnosis

Problem: An incomplete understanding of the presence, root cause and impact of a defect or anomaly in the system causes lack of corrective action at design or run time and may eventually result in an error or failure.

Context: The pattern is a derivative pattern of the `Fault Treatment` strategy pattern and applies to a system that has the following characteristics:

- The system has well-defined parameters that enable a monitoring system to discover the presence of a defect or anomaly in the behavior of the monitored system.
- The interaction between the monitored and monitoring systems is bounded in terms of time.
- The monitoring system has the capability to analyze the behavior of the monitored system.

Forces:

- The interactions between the monitoring and monitored systems may interfere with the operation of the monitored system, specifically its performance.
- While the frequency and duration of these interactions must be minimized to reduce the monitoring overhead, the interactions must be able to detect/infer a defect or anomaly in the monitored system.
- The monitoring system's data gathering and defect/anomaly detection/inference must be in time to prevent the activation of an error and a subsequent failure.
- The diagnostic resolution, i.e., the degree of accuracy of the fault diagnosis, must be high to be effective.

Solution: The `Fault Diagnosis` architectural pattern enables the discovery and treatment of a defect or anomaly in the system, which either has the potential to activate or already has activated to an error and potentially a subsequent failure. The pattern supports methods that attempt to recognize the presence of a defect or anomaly within a monitored system. It prevents its activation or enables its containment and mitigation by notifying the monitored system about a fault with location and type information, such that the anomaly or defect is removed before or after it results in an error or a failure. The solution requires a monitoring system, which may be a subsystem of the monitored system or an external independent system, to observe key parameters of the monitored system and to notify the monitored system when these parameters deviate.

The `Fault Diagnosis` architectural pattern makes an assessment on what may potentially or actually be wrong with a system. The diagnosis entails making inferences based on the observed behavior of a system and narrowing the search for the root cause of the fault. The assessment is based on the operating behavior of the monitored system. The components of this pattern are shown in Figure 12.

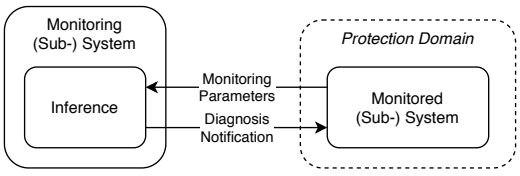
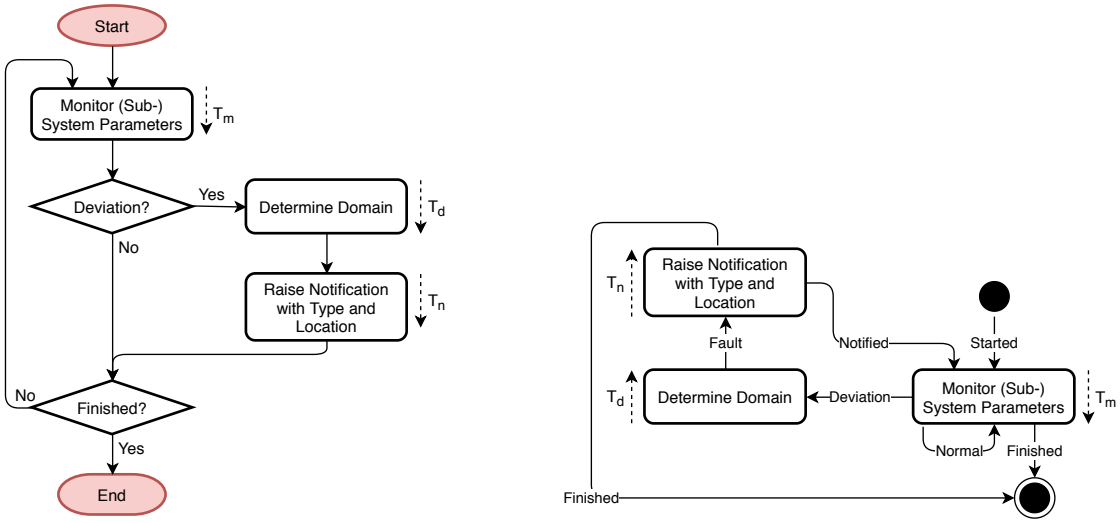


Figure 12. Fault Diagnosis pattern components

Capability: This pattern provides fault detection in the monitored system at design or run time, before it eventually results in an error or failure or after activation, by identifying deviations in monitored parameters. This pattern enables containment and mitigation of a future, imminent or present error or failure in the monitored system through detection and by notifying the monitored system about a fault with location and type information. The flowchart and state diagram of the pattern is shown in Figure 13 and its parameters in Table 6.



(a) Flowchart **(b) State diagram**

Figure 13. Fault Diagnosis pattern flowchart and state diagram

Table 6. Fault Diagnosis pattern parameters

Parameter	Definition
T_m	Time to monitor (sub-) system parameters, including wait and probe times
T_d	Time to determine the domain
T_n	Time to raise notification with type and location

Protection Domain: The protection domain extends to the monitored system.

Resulting Context: The discovery and treatment of a defect or anomaly in the system that has the potential to activate or has activated is enabled, which can be used to prevent, contain and mitigate an error or a failure in the system. The `Fault Diagnosis` architectural pattern requires identifying system parameters that indicate the presence of a fault. The overall system design must include a monitoring system, which introduces design complexity. When the monitoring system is extrinsic to the monitored system, the design effort may be simplified, but the interfaces between the monitoring and monitored systems must be well-defined. When the monitoring system is intrinsic to the design of the monitored system, design complexity increases due to the need to interface the monitoring and monitored subsystems.

A trade-off exists between interference with the operation of the monitored system caused by the frequency and duration of the interactions between the monitoring and monitored systems, and the ability to detect/infer a defect or anomaly in the monitored system. A high performing solution reduces the interference, while maintaining a reasonable ability to detect/infer a defect or anomaly.

The `Fault Diagnosis` architectural pattern may be used in conjunction with other architectural patterns

that provide containment and mitigation and require or can make use of fault detection. It detects the presence of a defect or anomaly and reports it, but does act to remedy the fault.

Examples:

- Hardware sensors in the CPUs, on the board, and in the power supply provide real-time operational data about temperature, fan speeds, and voltages. Software sensors in the Linux OS offer real-time operational data about CPU and memory load. The data is made available by sensor interface tools, such as IPMI [10], such that a monitoring tool, like the Ganglia Monitoring System [123], can aggregate it with time stamps in a central database. The monitoring tool can be configured to send a notification when a certain threshold value for a sensor is exceeded.
- A heartbeat monitoring tool checks the liveness of an MPI process in the system by regularly sending a message and waiting for a reply [147]. If no message is received back within a certain time period, the MPI process is considered failed and the MPI programming model runtime software is informed. This heartbeat monitoring is performed between all MPI processes, where all MPI processes watch each other. A message sent by the application can be used as a liveness request or reply message.
- An MPI tracing tool, like Vampir [8], is used to capture detailed time-stamped information about an application's MPI messages at execution time. The created MPI message trace contains time-stamped information about when an MPI function was called and when it was completed, such that it can be analyzed for inefficiencies in the MPI application's messaging behavior. A common root cause for MPI application scaling limitations is the inefficient use of MPI collectives. The MPI application developer is informed by the tool about such situations.
- In proactive fault tolerance, an OODA loop control is employed that utilizes monitoring tools for collecting sensor data (e.g., temperature, fan speeds, voltages, computational load, memory and storage usage, etc.). It leverages the warning thresholds of these sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72]. The mitigation may use process-level [187] or VM level [134] migration.
- Monitoring tools collecting event data (e.g., anomalous, error and failure events, debug messages, etc.) are used in conjunction with tools for temporal and spatial filtering to identify event correlations and to predict failures [116].
- Probabilistic networks are utilized for establishing correlations between event collected with monitoring tools to predict failures [159].

Rationale: The `Fault Diagnosis` architectural pattern enables a system to discover and treat a defect or anomaly in the system that has the potential to activate and become an error or failure or already has activated. It relies on a monitoring system to observe the monitored system, identifying deviations in monitored parameters. A discovered defect or anomaly is treated by raising a notification about a fault with location and type information, permitting the monitored system to provide containment and mitigation. A key benefit of this pattern is preemptive discovery of faults in the system, before they are activated and result in errors or failures. Preventive actions taken upon such discovery avoid the need for expensive error/failure recovery, compensation, or correction actions.

Related Patterns: In contrast to the `Fault Diagnosis` architectural pattern, the `Reconfiguration` architectural pattern additionally offers containment and mitigation.

Known Uses: HPC system installations use monitoring tools for collecting operational data, such as sensor data (e.g., heat, fan speeds, voltages, computational load, memory and storage usage, etc.), performance data (e.g., application execution, message and file system access times, etc.) and event data (e.g., anomalous, error and failure events, debug messages, etc.). Popular solutions include:

- Hard- and software sensors in CPUs, on the board, in the power supply, in the OS, and in system services

that provide real-time operational data to sensor interface tools. Examples are the hardware sensors in the IBM Power 8 [94] and Intel Xeon [54] processors, and the Linux OS monitoring capabilities.
- Sensor interface tools that gather real-time operational data from hard- and software sensors, such that they are available for monitoring tools. Examples are SMART [173] and IPMI [10].
- Monitoring tools that gather time-stamped information from sensor interface tools in a distributed computing system at a central location, such that they can be analyzed. Examples are the Supermon High-Speed Cluster Monitoring System [171], the Ganglia Monitoring System [123], Nagios [4], and the OVIS Lightweight Distributed Monitoring System [14].
- Heartbeat monitoring tools for liveness checking of individual nodes in the system [147] or MPI processes [26].
- System event loggers that gather time-stamped information from the OS or system services in a distributed computing system, such that they can be analyzed. The most prominent example is Linux's syslog [6].
- MPI tracing tools gather detailed time-stamped information about an application's MPI messages, such as when a particular message was sent and when it was received. Examples are Vampir [8] and SCALASCA [7].
- Comprehensive RAS management tools by vendors that support real-time monitoring of an entire HPC system, such as the CRMS [28].

The collected operational data may be analyzed by advanced tools for predicting future behavior based on experiences with past behavior and/or based on current behavior, such as for predicting faults, errors and failures. Known uses include:

- Utilizing a wide-variety of methods to analyze monitoring and log data to predict failures [105]
- Using machine learning to predict GPGPU memory errors [139]
- Leveraging warning thresholds of sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72]
- Characterizing the correlation between temperature, power consumption and memory or logic errors using monitoring and log data [138]
- Using probabilistic networks for establishing event correlations for prediction algorithms [159]
- Employing temporal and spatial filtering for failure prediction [116]

7.3.2 Reconfiguration

Name: Reconfiguration

Problem: A hardware or software error or subsystem failure due to a physical fault (e.g., wear-out or destruction) or a design fault (e.g., resource underprovisioning) in an HPC environment causes a software, such as an application, to experience an error or failure.

Context: The pattern is a derivative of the `Fault Treatment` and `Recovery` strategy patterns and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has well-defined parameters that enable a monitoring system to discover the presence of an existing or future fault, error, or failure in the behavior of the monitored system.
- The interaction between the monitored and monitoring systems is bounded in terms of time.
- The monitoring system has the capability to readily analyze the behavior of the monitored system to detect or predict a fault, error, or failure.
- The system is able to be reconfigured, where altering the configuration enables the system to remain operational.

Forces:

- The interactions between the monitoring and monitored systems may interfere with the operation of the monitored system, specifically its performance.
- While the frequency and duration of these interactions must be minimized to reduce the monitoring overhead, the interactions must be able to detect or predict a fault, error, or failure in the monitored system.
- The diagnostic resolution, i.e., the degree of accuracy of the fault diagnosis, must be high to be effective.
- The ability of the system to change its configuration must permit system operation that is functionally equivalent to the fault, error, or failure-free operation of the system.
- The reconfiguration may force the system to operate at a lower level of performance.

Solution: The `Reconfiguration` architectural pattern alleviates the impact of a fault, error, or failure on system operation by changing the system's configuration. It has a detection component that is derived from the `Fault Treatment` strategy pattern and an additional containment and mitigation component that is derived from the `Recovery` strategy pattern and acts upon the notification from the detection component.

The detection component enables the discovery of an existing or future fault, error, or failure in the system. It enables its containment and mitigation by notifying the system about it with location and type information. The solution requires a monitoring system, which may be a subsystem of the monitored system or an external independent system, to observe key parameters of the monitored system and to notify the monitored system when these parameters deviate. This pattern does not specify the detailed method of detection or prediction, which may be similar to the `Fault Diagnosis` architectural pattern.

The containment and mitigation component acts upon the notification from the detection component by (1) modifying the interconnection between *N* subsystems in the overall system, (2) restoring the affected subsystem or system to a known correct state (similar to the `Checkpoint Recovery` architectural pattern), or (3) restoring the affected subsystem or system to its initial state. When modifying the interconnection between *N* subsystems in the overall system, the affected subsystem is permanently isolated (excluded) and the system retains functional equivalency. When restoring the affected subsystem or system to a known correct state, previously preserved system state is restored and operation is resumed from a known correct

state. When restoring the affected subsystem or system to its initial state, the subsystem or system is reset. The components of this pattern are illustrated in Figure 14.

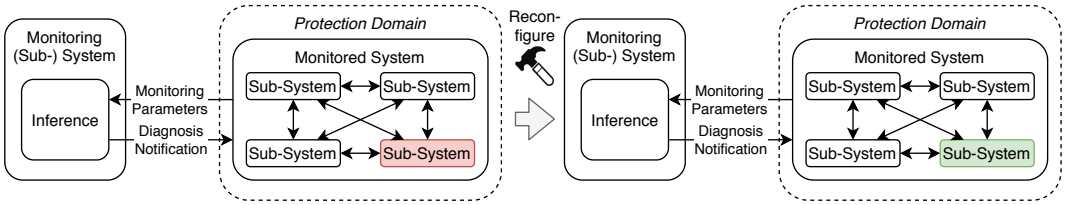
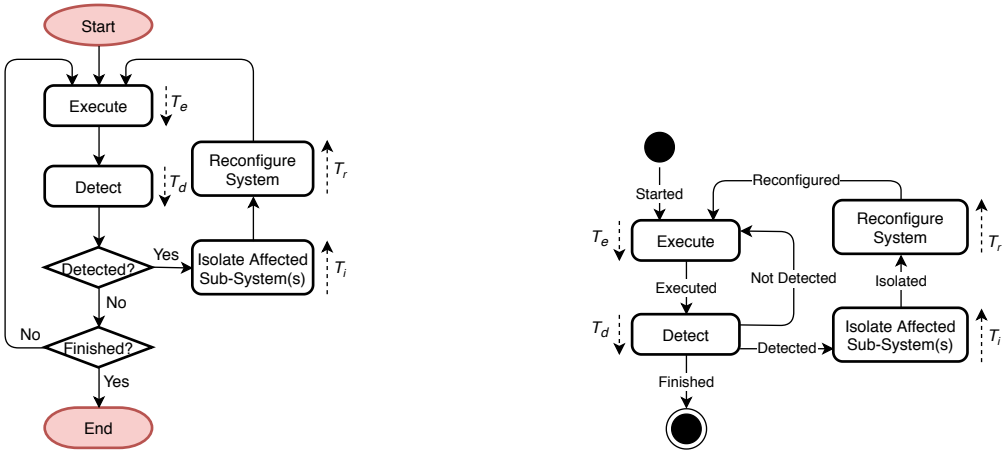


Figure 14. Reconfiguration pattern components

Capability: A system using this pattern is able to continue to operate in the presence of a non-permanent fault, error, or failure with some to significant interruption and some to no loss of progress. This pattern provides detection/prediction, containment, and mitigation of a future or existing fault, error, or failure in the system by changing the system's configuration. The flowchart and state diagram of the pattern is shown in Figure 15 and its parameters in Table 7.



(a) Flowchart **(b) State diagram**

Figure 15. Reconfiguration pattern flowchart and state diagram

Table 7. Reconfiguration pattern parameters

Parameter	Definition
T_e	Time to execute system progress
T_d	Time to detect or predict a fault, error or failure
T_i	Time to isolate the affected subsystem(s)
T_r	Time to reconfigure the system (may include or exclude the affected/isolated subsystem(s))

Protection Domain: The protection domain extends to the monitored system's state that is not lost due to a subsystem or system reconfiguration and to all of its resources that are able to be operate correctly after a reconfiguration.

Resulting Context: A fault, error, or failure is prevented from affecting the correct operation of the system. While functional equivalency to the original system configuration is maintained through reconfiguration, progress in the system may be lost. Reconfiguration may force the system to operate at a lower level of

performance. The system is interrupted during reconfiguration in response to a detected or predicted fault, error, or failure. It is interrupted during fault-, error- and failure-free operation when preserving system state. After reconfiguration, the system's ability to prevent a fault, error, or failure from affecting the correct operation of the system remains the same if it retains the ability to reconfigure to a functionally equivalent configuration.

The `Reconfiguration` architectural pattern may be used in conjunction with other architectural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other architectural pattern(s) and the `Reconfiguration` architectural pattern covers for the remaining error/failure types.

Examples: Unhealthy, erroneous, or failed system resources, such as compute nodes or NVIDIA GPGPU memory pages [141], are removed from the system's pool of resources. A programming model, such as the proposed ULFM extension to the MPI standard [31], permits an application to reconfigure the programming environment to handle resource failures. The targeted rejuvenation of data structures in system software, such as OS data structures, permits avoidance of and recovery from errors or failures without the need to reinitialize the affected compute node or the complete HPC system. The individual rejuvenation of HPC system services, such as the parallel file system MDS or the system's resource manager, allows dealing with errors or failures without the need to reinitialize the entire HPC system. A malfunctioning compute node may be rebooted.

Rationale: The `Reconfiguration` architectural pattern prevents an existing or future fault, error, or failure from affecting the correct operation of the system through changing the system's configuration. It relies on the ability to assume a configuration in response to a detected or predicted fault, error, or failure that retains functional equivalency with the original system configuration. The pattern may perform proactive actions, such as reconfiguring the system or subsystem before a fault, error, or failure impacts its operation, or reactive actions, such as reconfiguring the system or subsystem upon such an event.

Progress in the system may be lost. Reconfiguration may force the system to operate at a lower level of performance. Fault, error, or failure detection/prediction is part of the pattern and similar to the `Fault Diagnosis` architectural pattern. The containment and mitigation offered by this pattern can be similar to the `Checkpoint Recovery` architectural pattern and is independent from the type of fault, error, or failure. The pattern has very little to some design complexity and has low dependence on a system's architecture.

Related Patterns: In contrast to the `Fault Diagnosis` architectural pattern, the `Reconfiguration` architectural pattern additionally offers containment and mitigation. The `Reconfiguration` architectural pattern provides containment and mitigation capabilities that are, in part, similar to the `Checkpoint Recovery` architectural pattern, such as restoring the affected subsystem or system to a known correct state using previously preserved system state. In contrast to the `Checkpoint Recovery` architectural pattern, the `Reconfiguration` architectural pattern additionally offers detection and its containment and mitigation capabilities include changing the system's configuration.

Known Uses:

- HPC systems often automatically take unhealthy, erroneous, or failed compute nodes out of the system's resource manager's pool of available computing resources to avoid scheduling jobs on them and to permit maintenance (e.g., the the NodeKARE module in Cray Linux Environment (CLE) [102]).
- The recurrence of memory errors in a specific cell or region of the chip often indicates degradation of the memory module or region. NVIDIA GPGPU drivers support dynamic page retirement, which removes a memory page from the pool of available physical memory resources [141].
- The proposed ULFM extension to the MPI standard [31] allows parallel applications to get notifications

of process failures and to restructure the MPI environment. ULFM permits revoking an MPI communicator to prevent any further usage. It also allows shrinking an MPI communicator to exclude failed MPI processes from future communication.

- The Charm++ adaptive runtime system interacts with the resource manager to dynamically reconfigure the number of nodes assigned to a job based on the resilience, power, and performance characteristics of the system [11].
- The Mini-Ckpts framework [79] enables HPC applications to survive OS errors and failures. The OS is effectively rebooted, while the application and OS state directly relevant to the application is preserved in persistent memory. This rejuvenates all OS state that is not directly relevant to the application.
- Individual services, such as the parallel file system MDS or the system's resource manager, may be rebooted to fix erroneous behavior or failures. The correct service state is maintained on persistent storage to allow for resumption after reboot.
- OS structures may be regularly verified for correctness and rejuvenated if the correctness test fails. Inherent redundancy, such as with doubly linked lists, or other methods, such as using the known size of OS structures, are used to rejuvenate state, such as pointers.
- Various cluster management software systems, such as the Cray HSS [102], enable malfunctioning nodes in the cluster to be reset. The HSS initiates a reboot sequence for a failing node without disrupting the remaining nodes in the system.

7.3.3 Checkpoint Recovery

Name: Checkpoint Recovery

Problem: A hardware or software error or subsystem failure due to a physical fault (e.g., wear-out or destruction) or a design fault (e.g., resource underprovisioning) in an HPC environment causes a software, such as an application, to experience an error due to loss of or corrupted state, and potentially a subsequent failure.

Context: The pattern is a derivative of the `Recovery` strategy pattern. It applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The error or failure in the system that the pattern handles must be detected, as the pattern offers no error or failure detection.
- The system is capable of compartmentalizing its state that is accurately representative of the progress of the system since initialization at the time such state is captured.
- The system operation has well-defined intervals that enable the pattern to transition the system to a known correct interval in response to an error or failure.

Forces:

- The pattern requires additional storage to capture system state or to log messages or events, which increases overhead in terms of resources required by the system.
- The amount of state captured during each creation of a recovery point incurs storage space and execution time overheads.
- The creation frequency of system state snapshots determines overhead. More frequent snapshot creation increases system execution time, but reduces the amount of lost work upon an error or failure.
- The time interval for the recovery of a system from a snapshot as well as the time interval to create a snapshot must be less than the system's MTBF to guarantee forward progress.

Solution: The `Checkpoint Recovery` architectural pattern enables the resumption of correct operation of a system impacted by an error or failure. It supports resilient operation by restoring the system to a known correct state in the event of an error or failure. Checkpoint-recovery solutions are classified into checkpoint-based and log-based strategies.

The checkpoint-based strategy relies on the creation of system state snapshots and the maintenance of such checkpoints on a persistent storage system that is not affected by the error or failure. Upon detection of an error or failure, the most recent snapshot is used to recreate the last known error/failure-free state of the system. The log-based strategy relies on logging, i.e., storing the information of, events, such as messages sent between different parts of the system or to the system as input, on a persistent storage system that is not affected by the error or failure. Upon detection of an error or failure, the log is replayed to recreate the last known error/failure-free state of the system. In contrast to the checkpoint-based strategy, the log-based strategy is able to offer resilience in the presence of non-deterministic events and for non-deterministic systems, as replaying the event log deterministically recreates system state.

The checkpoint and log-based strategies may be used together, where events are logged in-between snapshots. With both strategies, the operation of the system is resumed when the system state is recovered. Based on a temporal view of the system's progress, the error/failure recovery may be either backward to the time when the snapshot occurred (rollback) or forward to the time when the error/failure event occurred (rollforward).

Undetected (latent) errors that are either detected later or result in a different detected error or failure later represent a problem, as the most recent or even more snapshots may contain an illegal system state. In this case, the most recent correct snapshot may be used to recreate the last known error/failure-free state of the system, skipping snapshots containing illegal state and going further back in time in terms of when the snapshot was made. The components of this pattern are illustrated in Figure 16.

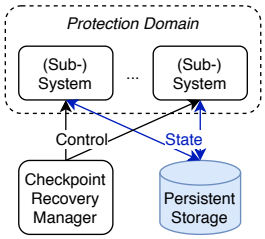
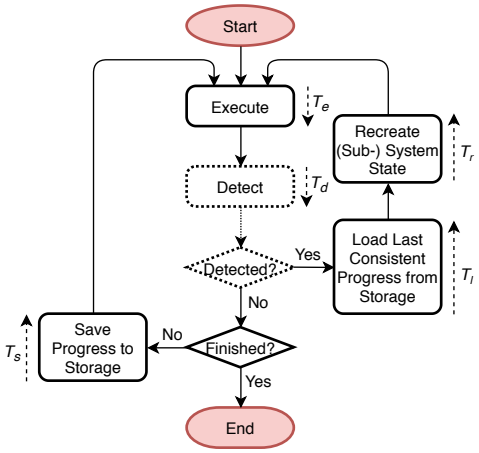
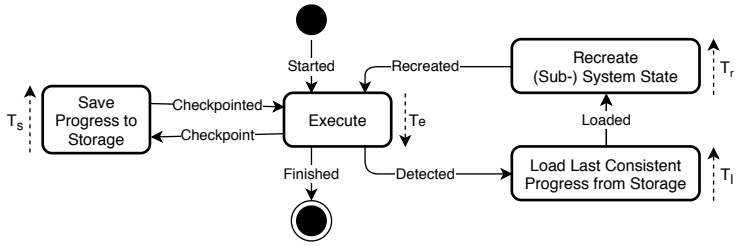


Figure 16. Checkpoint Recovery pattern components

Capability: This pattern provides mitigation of an error or failure in the system by preserving system state on a persistent storage system before an error or failure, using a checkpoint-based and or log-based strategy, and restoring the previously preserved system state upon such an event to resume operation from a known correct state. The flowchart and state diagram of the pattern is shown in Figure 17 and its parameters in Table 8.



(a) Flowchart



(b) State diagram

Figure 17. Checkpoint Recovery pattern flowchart and state diagram

Protection Domain: The protection domain extends to the system state that can be recreated using the system state captured by snapshots stored on persistent storage.

Table 8. Checkpoint Recovery pattern parameters

Parameter	Definition
T_e	Time to execute (sub-) system progress
T_d	Time to detect an error/failure (not part of this pattern, but shown for completeness)
T_l	Time to load (sub-) system state and progress from storage
T_r	Time to recreate (sub-) system state (initial and/or previously saved)
T_s	Time to save (sub-) system state and progress to storage

Resulting Context: Correct operation is resumed after an error or failure impacted the system. Progress in the system is lost after an error or failure if the recovery is only able to recreate the system state to the time when the snapshot occurred. Progress is not lost if the recovery is able to recreate the system state to the time when the error/failure event occurred. The system is interrupted during error-/failure-free operation for creating system state snapshots and maintaining them on a persistent storage system. The system is interrupted upon an error or failure for recreating the last known error/failure-free state of the system.

A trade-off exists between the creation frequency of system state snapshots and/or event logging and the corresponding execution time overhead during error-/failure-free operation vs. the amount of lost progress in case of a rollback or the amount of work required to recreate the state in the case of a rollforward. The optimal solution of this trade-off depends on the MTBF, the time it takes to save and load a system state snapshot, the time it takes to log events, and the time it takes to recreate the system state.

The `Checkpoint Recovery` architectural pattern does not provide error or failure detection and may be used in conjunction with the `Fault Diagnosis` architectural pattern to be fully functional. The `Checkpoint Recovery` architectural pattern may be used in conjunction with other architectural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other architectural pattern(s) and the `Checkpoint Recovery` architectural pattern covers for the remaining error/failure types.

Examples: Many HPC applications implement application-level checkpoint recovery by regularly saving intermediate results to stable storage as checkpoints and supporting the capability to load such a checkpoint in addition to some or all of the original data upon restart to recreate the last known correct state. Library-based solutions, such as FTI [27], permit tracking of state that needs to be saved and restored. System-level solutions, such asDMTCP [18], support transparent state saving and restoration using OS support. GVR [47] is a runtime system that provides fault tolerance to applications by versioning distributed arrays for checkpoint recovery, while the checkpoint-on-failure protocol [18] for MPI applications leverages the features of a high-quality fault-tolerant MPI implementation. In either case, algorithm-specific knowledge is needed to perform checkpoint recovery, Some ABFT solutions [122] can utilize the original or previously saved data as a replacement for lost or erroneous data and recover their state to the point at which the error/failure event occurred.

Rationale: The `Checkpoint Recovery` architectural pattern enables a system to tolerate an error or failure through resumption of correct operation after impact. It relies on the capability to preserve system state before an error or failure, often in a periodic fashion, and restore the previously preserved system state upon such an event to resume operation from a known correct state. The pattern performs proactive actions, such as preserving system state, but mostly relies on reactive actions after an error or failure impacted the system.

Error or failure detection is not part of the pattern. The preserved system state is managed on persistent storage, which is not part of the protection domain. The containment and mitigation offered by this pattern

are independent from the type of error or failure. The pattern has very little design complexity and has low dependence on a system's architecture, which makes it appealing as a general and portable solution.

Related Patterns: The `Reconfiguration` architectural pattern provides containment and mitigation capabilities that are, in part, similar to the `Checkpoint Recovery` architectural pattern, such as restoring the affected subsystem or system to a known correct state using previously preserved system state. In contrast to the `Checkpoint Recovery` architectural pattern, the `Reconfiguration` architectural pattern additionally offers detection and its containment and mitigation capabilities include changing the system's configuration.

Known Uses:

- Many HPC applications implement application-level checkpoint recovery by regularly saving intermediate results to stable storage as checkpoints and supporting the capability to load such a checkpoint in addition to some or all of the original data upon restart to recreate the last known correct state.
- FTI [27] is a library-based checkpoint recovery solution, tracking, saving and restoring data structures in multilevel checkpoint storage.
- The CoCheck supports checkpoint recovery for an MPI implementation; for the coordination of the checkpoints, CoCheck uses a special process [172]. The Condor checkpoint/restart facility is enabled by the user by linking the program source code with the condor library [121].
- The libckpt [148] is a user-level, library-based checkpointing solution that supports explicit directives to determine the scope of an application's checkpointed state.
- The SCR [130] library uses multilevel checkpointing; it creates frequent inexpensive checkpoints that can recover the loss of a few nodes to the local node-level storage and writes complete checkpoints that can withstand an entire system failure to the parallel file system.
- BLCR is a process-level checkpoint recovery module for the Linus OS. It has been used in conjunction with LAM/MPI to permit OS-level checkpoint recovery for MPI applications [184].
- Improvements to system-level checkpoint recovery solutions, such as BLCR, include incremental checkpoint recovery, where only changes between two checkpoints are being saved and the recovery performs a reconstruction of correct state from multiple incremental checkpoints [185].
- DMTCP [18] is a transparent checkpoint recovery solution for multi-threaded and MPI applications.
- The local failure local recovery protocol uses checkpoint recovery of only some processes with locally stored checkpoints to create correct state [177].
- Message logging protocols have been implemented in OpenMPI to support faster failure recovery with message replay [35].
- GVR [47] is a runtime system that provides fault tolerance to applications by versioning distributed arrays. It supports checkpoint recovery based on application-specified mechanisms. Past versions of lost or erroneous data can be retrieved and used for computing the state at which the error/failure event occurred.
- The checkpoint-on-failure protocol [18] for MPI applications leverages the features of a high-quality fault-tolerant MPI implementation and algorithm-based checkpoint recovery. It provides the ability for all healthy processes to continue to operate and perform checkpoint recovery using message passing.
- Some ABFT solutions [122] can utilize the original or previously saved data as a replacement for lost or erroneous data and recover their state to the point at which the error/failure event occurred.

7.3.4 Redundancy

Name: Redundancy

Problem: A hardware error or subsystem failure due to a physical fault (e.g., wear-out or destruction) in an HPC environment causes a software, such as a system service or an application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Compensation` strategy pattern and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has a modular design that has a well-defined scope and a set of inputs and outputs.

Forces:

- The pattern introduces an execution time and/or resource requirement (storage space, computational capability, etc.) penalty independent of whether an error or failure occurs during system operation or not.
- The scope and strength of the redundancy employed by the pattern determine its execution time and resource requirement overhead.

Solution: The `Redundancy` architectural pattern enables the continuous correct operation of a system impacted by an error or failure. It supports resilient operation by applying redundancy to system state and optionally to system resources. This redundancy may be in the form of encoded system state or *N* functionally identical replicas. The pattern requires very well defined input and output to permit encoding/decoding of system state, input encoding/replication, and output decoding/comparison.

For redundancy through encoded system state, input is encoded, processed redundantly in an encoded fashion by the system, and the output is then decoded. The decoding corrects an error or failure. The scope and strength of the redundancy are defined by the encoding/decoding. For redundancy through *N* functionally identical replicas, input is replicated to identical instances of the system, processed by each replica system, and the output is then compared. The comparison corrects an error or failure of a replica system. The scope and strength of the redundancy are defined by the number of functionally identical replicas *N*.

Redundancy can be in time, meaning the same system resources process the encoded input or execute the *N* functionally identical replicas in time. Redundancy can also be in space, meaning additional (redundant) system resources are used, such that the different system resources process the encoded input or execute the *N* functionally identical replicas in space. Redundancy in time saves system resources, while redundancy in space offers more error/failure coverage. A mix between redundancy in time and space is possible as well, where there is more encoded system state or there are more functionally identical replicas than additional (redundant) system resources. The components of this pattern are illustrated in Figure 18.

Capability: This pattern provides error and/or failure detection in the system by applying redundancy to system state in the form of encoded system state or *N* functionally identical replicas. The pattern provides mitigation of an error or failure in the system by applying redundancy to system state and optionally to system resources, such that the system continues to operate correctly in the presence of such an event. The flowchart and state diagram of the pattern is shown in Figure 19 and its parameters in Table 9.

Protection Domain: The protection domain extends to the encoded system state and to the system resources processing it, and to the system state and the system resources that implement the *N* functionally identical replica systems.

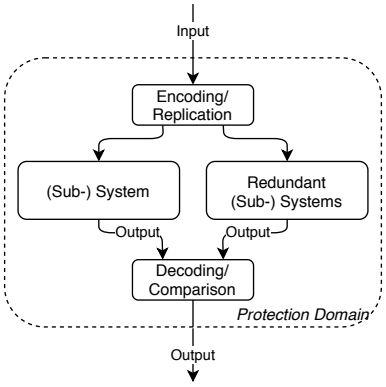


Figure 18. Redundancy pattern components

Table 9. Redundancy pattern parameters

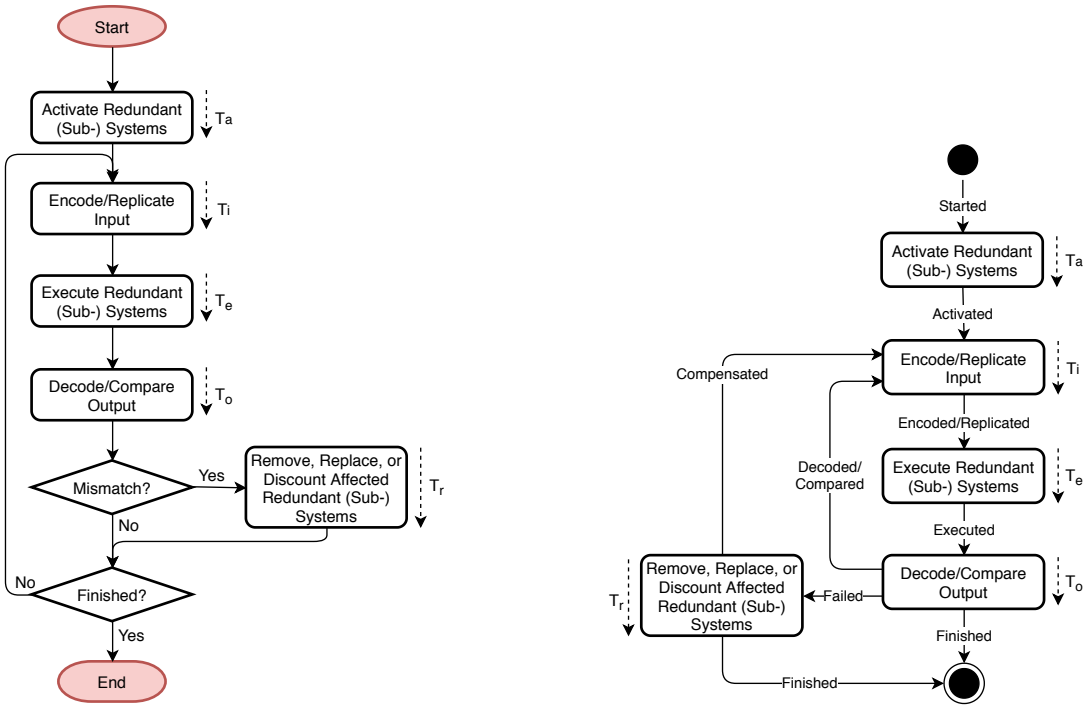
Parameter	Definition
T_a	Time to activate the redundant (sub-) systems
T_i	Time to encode/replicate the input to the redundant (sub-) systems
T_e	Time to execute redundant (sub-) system progress
T_o	Time to decode/compare the output from the redundant (sub-) systems
T_r	Time to remove, replace or discount the affected redundant (sub-) systems

Resulting Context: Correct operation is performed despite an error or failure impacting the system. Progress in the system is not lost due to an error or failure. The system is not interrupted during error-/failure-free operation. The system may be interrupted when encountering an error or failure. Resource usage in time or space is increased according to the amount of redundancy employed in the form of encoded system state or N functionally identical replicas.

A trade-off exists between the amount of redundancy employed and the number of errors and/or failures that can be tolerated at the same time and/or in time. More redundancy tolerates generally more errors and/or failures, but requires either more resources or more execution time.

The `Redundancy` architectural pattern may be used in conjunction with other architectural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other architectural pattern(s) and the `Redundancy` architectural pattern covers for the remaining error/failure types.

Examples: There are various schemes that enable forward error correction redundancy in memory devices, storage systems as well as communication channels. Based on time and space overhead constraints, schemes of different detection and correction capabilities are used. Popular examples include parity bits, checksums, Hamming codes, hash function codes. More elaborate schemes such as systematic cyclic block codes include binary BCH, Reed-Solomon and CRC. Forward error correction can be found in HPC storage systems with RAID, the InfiniBand interconnect [9], the memory hierarchy [131, 59], ABFT solutions [95] and coded computing [106]. Active/Standby redundancy is typically used for critical hardware or software systems in HPC environments. For example, power supplies, voltage regulators, the parallel file system MDS in Lustre [194] and the SLURM [192] job and resource manager are often implemented in an active/standby fashion. Dual-modular redundancy for error detection and failure compensation and triple-modular redundancy for error detection and correction and failure compensation



(a) Flowchart **(b) State diagram**

Figure 19. Redundancy pattern flowchart and state diagram

are used n HPC environments as well. Examples include dual-redundant cooling fans, dual- and triple–modular redundant MPI implementations [78], dual-redundant parallel file system MDS solutions [92] and dual-redundant mission-critical HPC systems (e.g., weather forecast).

Rationale: The `Redundancy` architectural pattern enables a system to tolerate an error or failure through continuation of correct operation after impact. It relies on system state redundancy in the form of encoded system state or functionally identical replicas. The pattern performs mostly proactive actions, such as maintaining redundancy. Error or failure detection is part of the pattern in the form of output decoding or comparison. The pattern has low to high design complexity, as input needs to be encoded/replicated, processed encoded, and output needs to be decoded/compared.

Related Patterns: The `Redundancy` and `Design Diversity` architectural patterns are based on inclusion of redundancy to compensate for errors or failures. The diversity in the `Redundancy` architectural pattern stems from the replication of the system's state, unlike the `Design Diversity` architectural pattern, which uses independently implemented versions of the system's design to tolerate errors or failures.

Known Uses:

- HPC storage systems use forward error correction redundancy in the form of RAID.
- InfiniBand, which is among the most widely deployed high-speed interconnect employs forward error correction redundancy in the InfiniBand devices, including adapters and switches, to fix bit errors throughout the network [9]. The forward error correction allows reduction in data re-transmission between the end-nodes.
- HPC systems use memory DIMMs that employ ECC, which is based on forward error correction redundancy by maintaining redundant bits per memory line to support SECDED for bit flips [131].
- Chipkill [59] uses forward error correction redundancy across memory chips atop ECC to perform

SECDED at a symbol-level granularity.

- Some ABFT methods use schemes, such as checksums, that perform forward error correction redundancy at the application-level [95].
- Coded computing uses algorithmic forward error correction redundancy. A recent solution used coded computing for parallel matrix-matrix multiplication [106].
- Production HPC systems such as the Cray XC40 series [103] include active/standby redundant power supplies and voltage regulator modules to ensure continuous operation in the event that one of these units experiences malfunction or failure.
- Active/standby (cold, warm and hot) redundancy using shared storage between active and standby systems is a common technique. The shared storage may be a dual-hosted hard drive, a networked storage using RAID, or a DRBD [154]. Such solutions have been extensively used in HPC environments for critical system services, such as the job and resource manager (e.g., SLURM [192] and SGE [175]) and the parallel file system MDS (e.g., PVFS [146] and Lustre [194]).
- Active/hot-standby redundancy using a commit protocol for state replication has been implemented for some HPC job and resource managers as part of high availability cluster solutions, such as HA-OSCAR [119] with its commit protocol for OpenPBS [16].
- High availability clustering is a form of active/hot-standby redundancy with n active systems and m standby systems that are set up in an $n + m$ configuration. It targets high throughput processing of a large number of small service requests with no or minimal state changes, where the active systems respond to service requests. High availability clustering may use shared storage, state replication or state separation. In contrast to shared storage and state replication, where systems have the same state, state separation splits the state space among active systems, such as two parallel file system MDSs serving two different file system directories. An implementation of HA-OSCAR supported high availability clustering for two job and resource managers, OpenPBS [16] and SGE [175]). Parallel file system MDSs, such as Lustre [194], support high availability clustering as well.
- Active/standby redundancy also plays a role in resilience for parallel applications in HPC environments. Starting a parallel application with additional spare compute nodes enables its reconfiguration and the replacement of a failed compute node without completely restarting the application [21].
- Production HPC systems such as the Cray XC40 series [103] include n-modular redundant cooling fans to ensure continuous operation in the event that one of these units experiences malfunction or failure.
- Some implementations of the MPI standard use n-modular redundancy for detection and correction of errors by replicating the MPI messages, or even by replicating MPI processes. The MR-MPI [70], rMPI [76], and RedMPI [78] prototypes are known examples for this n-modular redundancy approach. N-modular redundancy for compute nodes in a system have been evaluated and shown to improve the overall availability of a HPC system [71].
- Some n-modular redundancy implementations for service nodes in HPC systems exist as well, such as for the MDS of PVFS [92] and for HPC job and resource managers that are compliant with the PBS [178].
- N-modular redundancy at the whole system level is often implemented for mission critical HPC systems, such as for weather forecast. In this case, two completely redundant HPC systems perform exactly the same computation.

7.3.5 Design Diversity

Name: : Design Diversity

Problem: A hardware or software error or subsystem failure due to a design fault (e.g., human mistake or defective design tool) in an HPC environment causes a software, such as a system service or an application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Compensation` strategy pattern and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has a well-defined specification for which multiple implementation variants may be created.
- There is an implicit assumption of independence between multiple variants of the implementation.

Forces:

- The pattern introduces an execution time and/or resource requirement (storage space, computational capability, etc.) penalty independent of whether an error or failure occurs during system operation or not.
- The scope and strength of the diversity employed by the pattern determine its execution time and resource requirement overhead.
- The pattern requires distinct implementations of the same design specification, which may need to be created by different individuals.
- The pattern increases design complexity due to the need of additional design and verification effort required to create multiple implementations.
- The pattern may introduce a performance penalty during error/failure-free operation due to disparity in the implementation variants.

Solution: The `Design Diversity` architectural pattern enables the continuous correct operation of a system impacted by an error or failure. It supports resilient operation by applying redundancy to system state and optionally to system resources. This redundancy is in the form of *N* functionally equivalent alternate system implementations. This pattern designs different implementations of the system that are functionally equivalent to enable error and failure resilience through design diversity. Different implementations of the system are less likely to experience the same error or failure.

The pattern requires very well defined input and output to permit input replication and output validation. Input is replicated to functionally equivalent alternate system implementations, processed by each implementation of the system, and then compared. The validation corrects an error or failure of a system implementation. The scope and strength of the redundancy are defined by the number of functionally equivalent alternate system implementations, *N*, and by their implementation design diversity.

Redundancy can be in time, meaning the same system resources execute the *N* functionally equivalent alternate system implementations in time. Redundancy can also be in space, meaning additional (redundant/diverse) system resources execute the *N* functionally equivalent alternate system implementations in space. Redundancy in time saves system resources, while redundancy in space offers more error/failure coverage. A mix between redundancy in time and space is possible as well, where there are more functionally equivalent alternate system implementations than additional (redundant/diverse) system resources. The components of this pattern are illustrated in Figure 20.

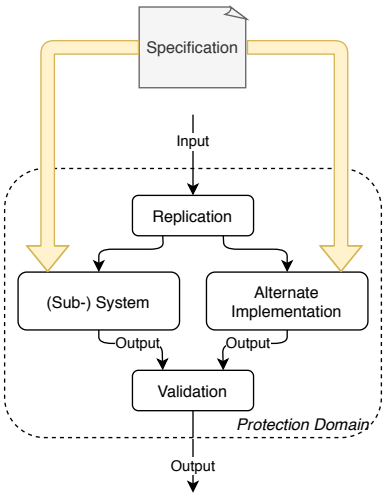


Figure 20. Design Diversity pattern components

Capability: This pattern provides error and/or failure detection in the system by applying redundancy to system state in the form of N functionally equivalent alternate system implementations. The pattern provides mitigation of an error or failure in the system by applying redundancy to system state and optionally to system resources, such that the system continues to operate correctly in the presence of such an event. The flowchart and state diagram of the pattern is shown in Figure 21 and its parameters in Table 10.

Table 10. Design Diversity pattern parameters

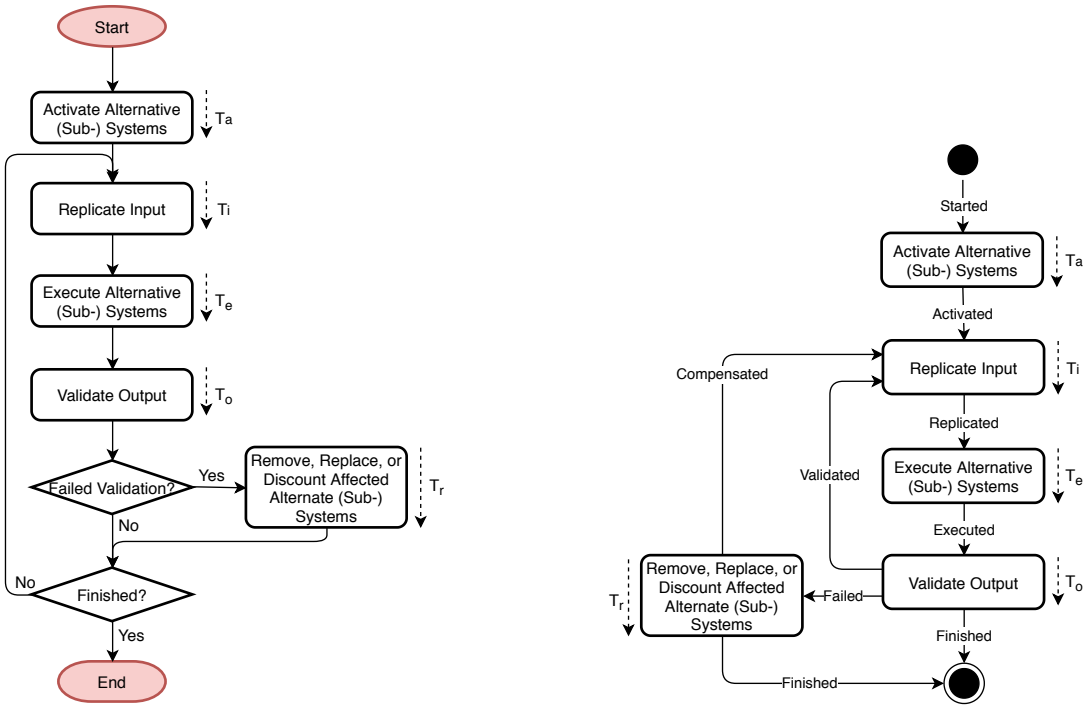
Parameter	Definition
T_a	Time to activate the alternative (sub-) systems
T_i	Time to replicate the input to the alternative (sub-) systems
T_e	Time to execute redundant/alternative (sub-) system progress
T_o	Time to validate the output from the alternative (sub-) systems
T_r	Time to remove, replace or discount the affected alternative (sub-) systems

Protection Domain: The protection domain extends to the system state and the system resources described by the design specification that implement the functionally equivalent alternate systems.

Resulting Context: Correct operation is performed despite an error or failure impacting the system. Progress in the system is not lost due to an error or failure. The system is not interrupted during error-/failure-free operation. The system may be interrupted when encountering an error or failure. Resource usage in time or space is increased according to the amount of redundancy employed in the form of N functionally equivalent alternate system implementations.

A trade-off exists between the amount of redundancy employed and the number of errors and/or failures that can be tolerated at the same time and/or in time. More redundancy tolerates generally more errors and/or failures, but requires either more resources or more execution time.

The `Design Diversity` architectural pattern may be used in conjunction with other architectural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other architectural pattern(s) and the `Design Diversity` architectural pattern covers for the remaining error/failure types.



(a) Flowchart **(b) State diagram**

Figure 21. Design Diversity pattern flowchart and state diagram

Examples: In HPC environments, various versions of the same software are used for the detection of implementation errors. This applies to completely different implementations of the MPI standard and to numerical libraries as well as to different versions of the same implementation. Regression and comparison tests are performed to identify incorrect behavior, missing features and performance problems. Containment Domains [48] and the SWIFT library [97] provide language-based approaches for recovery blocks with design diversity. Applications also often contain verification routines that check for the validity of a computation and correct any detected errors using application-specific knowledge and design diversity.

Rationale: The `Design Diversity` architectural pattern enables a system to tolerate an error or failure through continuation of correct operation after impact. It relies on system state redundancy in the form of functionally equivalent alternate system implementations. The pattern performs mostly proactive actions, such as maintaining redundancy. Error or failure detection is part of the pattern in the form of output validation. The pattern has high design complexity due to the need for functionally equivalent alternate system implementations.

Related Patterns: The `Redundancy` and `Design Diversity` architectural patterns are based on inclusion of redundancy to compensate for errors or failures. The diversity in the `Redundancy` architectural pattern stems from the replication of the system's state, unlike the `Design Diversity` architectural pattern, which uses independently implemented versions of the system's design to tolerate errors or failures.

Known Uses:

- HPC centers often provide various MPI library implementations, such as the MVAPICH2 [3], OpenMPI [5], MPICH2 [2], and Intel MPI [1], all of which are based on the MPI standard. Running large-scale applications with these separate implementations of MPI potentially exposes implementation errors in the MPI libraries. Similarly, different versions of numerical libraries are often provided as well.

Regression and comparison tests are performed to identify incorrect behavior, missing features and performance problems.

- The DIVA processor architecture [23] includes an out-of-order core as well as a simple in-order pipelined core. The in-order pipeline is functionally equivalent to the primary processor core (it implements the same instruction set architecture) and is used to detect errors in the design of the out-of-order processor core.
- Containment Domains [48] provide a recover routine with design diversity that is initiated upon detection of an error in the execution of the block of code encapsulated by the domain. This enables the containment domain to constrain the detection and correction of errors to the boundary of the domain.
- The SWIFT library [97] provides language-based implementation of a recovery block with design diversity for use in C language programs.
- Applications often contain verification routines that check for the validity of a computation. Application-specific knowledge is used in a recovery block to correct any detected errors.

7.3.6 Self-Masking

Name: Self-Masking

Problem: A hardware or software error or subsystem failure in an HPC environment causes a software, such as a numerical library or application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Self-Stabilization` strategy pattern and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.

Forces:

- System performance in the presence of errors and failures is determined by the efficiency in reaching a correct system state from an illegal system state, which may differ for different illegal system states.
- There is no inherent impact on system performance in error/failure-free conditions. However, the needed capability to reach a correct state from an illegal state may reduce error/failure-free performance.
- A correct state may not be reached from all illegal system states, which limits the scope of this pattern.
- The process of self-masking requires implicit error and/or failure detection of the illegal system state to be masked and self-maskable system state, which may require a system redesign to apply the pattern.

Solution: The `Self-Masking` pattern relies on the capability of reaching a correct system state from an illegal system state after a finite number of execution steps using implicit error/failure detection and self-masking. Forward progress is guaranteed by design as a correct system state can be reached in a finite number of steps. Self-masking may be as simple as an approximation of a correct state. The correct state reached after self-masking is not necessarily the same state that would have been reached without an error or failure. The scope of this pattern extends to the system state space that is either correct or incorrect but self-maskable to a correct state. The components of the pattern are illustrated in Figure 10.

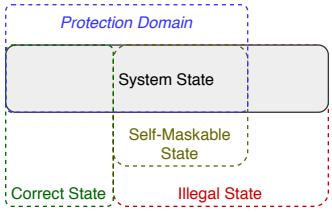
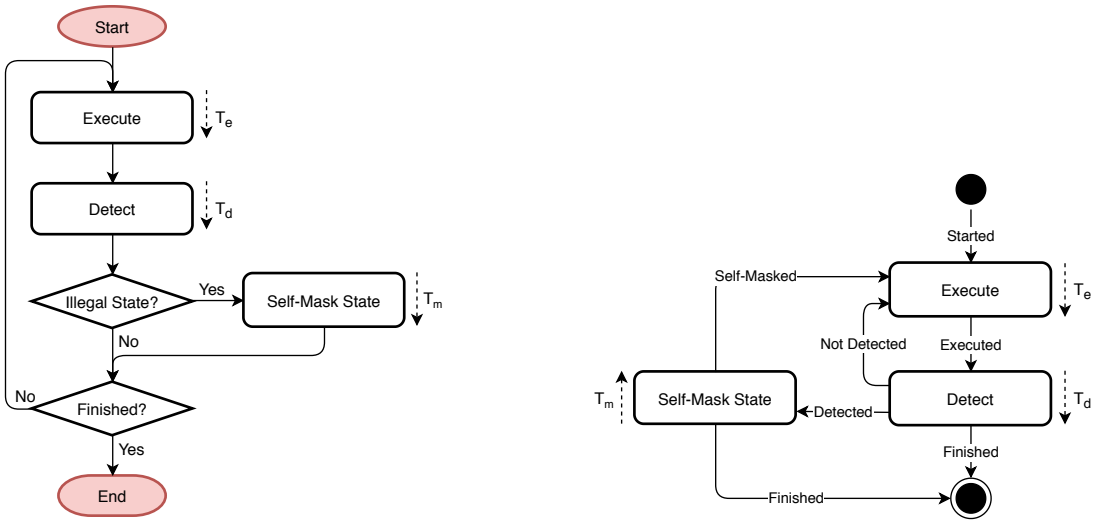


Figure 22. Self-Masking pattern components

Capability: This pattern supports the handling of errors and failures with detection, containment, and mitigation using self-masking, i.e., dynamic adaptation through masking of errors/failures. Error/failure detection is implicit by providing the capability to self-mask an illegal system state. Self-masking is an architectural feature of self-stabilization. The flowchart and state diagram of the pattern is shown in Figure 23 and its parameters in Table 11.

Table 11. Self-Masking pattern parameters

Parameter	Definition
T_e	Time to execute system progress
T_d	Time to detect illegal system state
T_m	Time to self-mask illegal system state



(a) Flowchart **(b) State diagram**

Figure 23. Self-Masking pattern flowchart and state diagram

Protection Domain: The protection domain of this pattern extends to the system state space that is either correct or incorrect but self-maskable to a correct state.

Resulting Context: An error or failure is tolerated through self-masking. The `Self-Masking` pattern requires the capability of reaching a correct system state from an illegal system state, which may reduce error/failure-free performance. The efficiency of self-masking determines performance in the presence of errors and failures. A high performing solution reduces the impact on error/failure-free performance, while at the same time providing efficient self-masking.

A trade-off exists between both, performance in the presence of errors and failures and error/failure-free performance, especially in systems with high error/failure rates. Self-masking is a passive approach, where the system is designed with self-maskable illegal system state. Self-masking becomes an intrinsic property of the system. The implicit detection may reduce error/failure-free performance. The pattern's protection domain does not cover illegal system state that is not self-maskable.

The `Redundancy` architectural pattern may be applied to extend the pattern's protection domain with redundant or diverse system state for self-masking. The `Self-Masking` pattern may be used in conjunction with other architectural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other architectural pattern(s) and the `Self-Masking` pattern covers for the remaining error/failure types.

Examples:

- Gossip-based algorithms tolerate errors and failures using the `Self-Masking` pattern. Data is communicated to multiple parts of the system participating in the Gossip-based algorithm and errors or failures are self-masked. The efficiency and performance of self-masking in the presence of errors and failures and the error/failure-free performance depend on the communication pattern (e.g., reach and frequency).
- Iterative solvers or solvers using sampling tolerate errors and failures using the `Self-Masking` pattern using either an iterative process that progresses toward a correct solution in the presence of errors or a probabilistic sampling process that generates enough correct samples over incorrect samples to calculate

a correct solution. The efficiency and performance of self-masking in the presence of errors and failures and the error/failure-free performance depend on the algorithmic capabilities of the iterative process (e.g., step size) or the probabilistic sampling process (e.g., oversampling).

Rationale: The `Self-Masking` pattern enables a system to tolerate errors or subsystem errors or failures through adaptation. It relies on the capability of reaching a correct system state from an illegal system state in a finite number of execution steps. It uses implicit error/failure detection and self-masking to passively transition from an illegal to a correct system state. Self-masking is intrinsic to the design of the system when this pattern is applied. The pattern has high design complexity and has low dependence on a system's architecture.

Related Patterns: The `Self-Correction` architectural pattern uses the same `Self-Stabilization` strategy pattern, but employs explicit detection and self-correction as architectural features. Like the `Checkpoint Recovery`, `Redundancy` and `Design Diversity` architectural patterns, the `Self-Masking` architectural pattern seeks to mask system state when using self-masking. However, it does not rely on state saving, redundancy, or design diversity for masking. The `Redundancy` architectural pattern may be applied to extend the `Self-Masking` architectural pattern's protection domain.

Known Uses:

- Gossip-based algorithms tolerate errors and failures using the `Self-Masking` pattern. This includes gossip-based aggregation and reduction algorithms [43, 140, 83], orthogonalization methods [84, 83], eigensolvers [174], and least squares solvers [151].
- Fixed point methods that converge globally when certain conditions are satisfied are able to tolerate certain errors using the `Self-Masking` pattern [20, 19], such as a Jacobi iterative scheme that will converge for any initial guess if the matrix is diagonally dominant.
- Certain domain decomposition methods are tolerant to errors using the `Self-Masking` pattern as they converge in the presence of errors, such as the classical overlapping Schwarz algorithm [86], asynchronous domain decomposition methods [85] and task-based domain decomposition approaches that use sampling and a regression-based solution update [157, 156, 132].

7.3.7 Self-Correction

Name: Self-Correction

Problem: A hardware or software error or subsystem failure in an HPC environment causes a software, such as a numerical library or application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Self-Stabilization` strategy pattern and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.

Forces:

- System performance in the presence of errors and failures is determined by the efficiency in reaching a correct system state from an illegal system state, which may differ for different illegal system states.
- There is no inherent impact on system performance in error/failure-free conditions. However, the needed capability to reach a correct state from an illegal state may reduce error/failure-free performance.
- A correct state may not be reached from all illegal system states, which limits the scope of this pattern.
- The process of self-correction requires an explicit error and/or failure detection of the illegal system state to be masked and self-correctable system state, which may require a system redesign to apply the pattern.

Solution: The `Self-Correction` pattern relies on the capability of reaching a correct system state from an illegal system state after a finite number of execution steps using explicit error/failure detection and self-correction. Forward progress is guaranteed by design as a correct system state can be reached in a finite number of steps. Self-correction may be as simple as discarding, recomputing, or estimating a wrong value in the system or a wrong or missing output from a subsystem. The correct state reached after self-correction is not necessarily the same state that would have been reached without an error or failure. The scope of this pattern extends to the system state space that is either correct or incorrect but self-correctable to a correct state. The components of the pattern are illustrated in Figure 10.

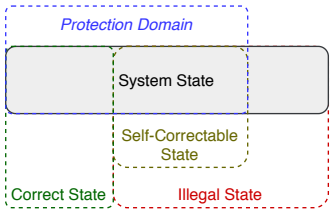
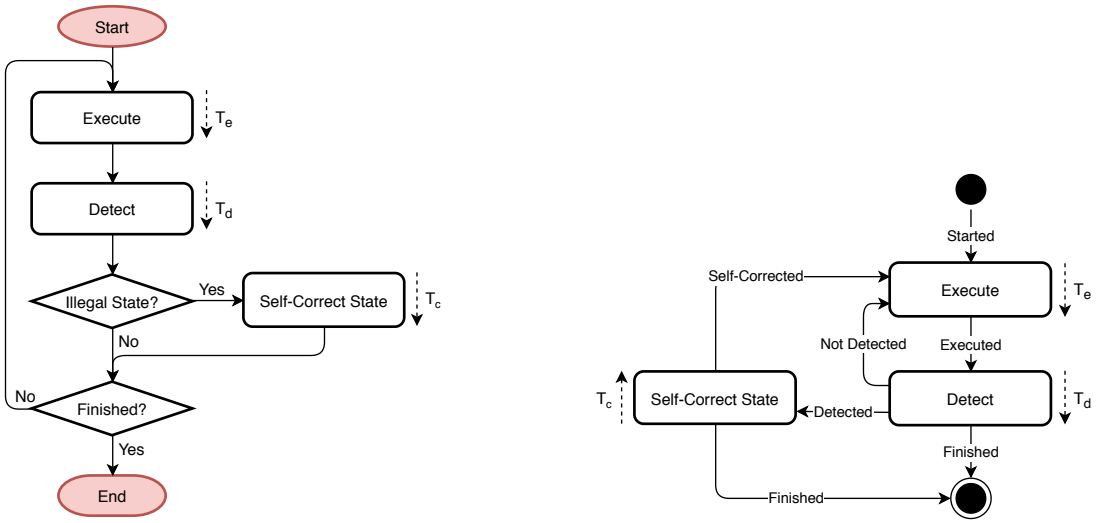


Figure 24. Self-Correction pattern components

Capability: This pattern supports the handling of errors and failures with detection, containment, and mitigation using self-correction, i.e., dynamic adaptation through correction of errors/failures. Error/failure detection is explicit by providing the capability to self-correct an illegal system state. Self-correction is an architectural feature of self-stabilization. The flowchart and state diagram of the pattern is shown in Figure 25 and its parameters in Table 12.

Protection Domain: The protection domain of this pattern extends to the system state space that is either correct or incorrect but self-correctable to a correct state.

Resulting Context: An error or failure is tolerated through self-correction. The `Self-Correction` pattern requires the capability of reaching a correct system state from an illegal system state, which may reduce error/failure-free performance. The efficiency of self-correction determines performance in the presence of



(a) Flowchart **(b) State diagram**

Figure 25. Self-Correction pattern flowchart and state diagram

Table 12. Self-Correction pattern parameters

Parameter	Definition
T_e	Time to execute system progress
T_d	Time to detect illegal system state
T_c	Time to self-correct illegal system state

errors and failures. A high performing solution reduces the impact on error/failure-free performance, while at the same time providing efficient self-correction.

A trade-off exists between both, performance in the presence of errors and failures and error/failure-free performance, especially in systems with high error/failure rates. Self-correction is an active approach, where the system is designed with self-correctable illegal system state. Self-correction becomes an intrinsic property of the system. The explicit detection may reduce error/failure-free performance. The pattern's protection domain does not cover illegal system state that is not self-correctable.

The `Redundancy` architectural pattern may be applied to extend the pattern's protection domain with redundant or diverse system state for self-correction. The `Self-Correction` pattern may be used in conjunction with other architectural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other architectural pattern(s) and the `Self-Correction` pattern covers for the remaining error/failure types.

Examples:

- Self-stabilizing solvers may produce an illegal state in the presence of errors or failures and tolerate their impact using the `Self-Correction` pattern to reach a correct state. The efficiency and performance of self-correction in the presence of errors and failures and the error/failure-free performance depend on the methods used for explicit error/failure detection (e.g., error check after every iteration) and for reaching the correct state (e.g., successive approximation toward a correct state).
- In proactive fault tolerance, an OODA loop control is employed that utilizes monitoring tools for collecting sensor data (e.g., temperature, fan speeds, voltages, computational load, memory and storage

usage, etc.). It leverages the warning thresholds of these sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72]. The mitigation may use process-level [187] or VM level [134] migration.

Rationale: The `Self-Correction` pattern enables a system to tolerate errors or subsystem errors or failures through adaptation. It relies on the capability of reaching a correct system state from an illegal system state in a finite number of execution steps. It uses explicit error/failure detection and self-correction to actively transition from an illegal to a correct system state. Self-correction is intrinsic to the design of the system when this pattern is applied. The pattern has high design complexity and has low dependence on a system's architecture.

Related Patterns: The `Self-Masking` architectural pattern uses the same `Self-Stabilization` strategy pattern, but employs implicit detection and self-masking as architectural features. Like the `Checkpoint Recovery`, `Redundancy` and `Design Diversity` architectural patterns, the `Self-Correction` architectural pattern seeks to correct system state when using self-correction. However, it does not rely on state saving, redundancy, or design diversity for correction. The `Redundancy` architectural pattern may be applied to extend the `Self-Correction` architectural pattern's protection domain. The `Fault Diagnosis` architectural pattern may be applied for explicit error and/or failure detection.

Known Uses:

- A self-stabilizing label-propagation algorithm that computes the connected components in a graph while being tolerant to errors [161].
- Two self-stabilizing iterative linear solvers [161], one for the steepest descent and one for conjugate gradient, are tolerant to errors.
- Leveraging warning thresholds of sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72].

7.4 STRUCTURAL PATTERNS

7.4.1 Monitoring

Name: Monitoring

Problem: Not knowing the presence, root cause, and impact of a defect or anomaly in the system causes a lack of corrective action at design or run time and may eventually result in an error or failure.

Context: The pattern is a derivative pattern of the `Fault Diagnosis` architectural pattern and applies to a system that has the following characteristics:

- The system has well-defined parameters that enable a monitoring system to discover the presence of a defect or anomaly in the behavior of the monitored system.
- The interaction between the monitored and monitoring systems is bounded in terms of time.
- The monitoring system has the capability to readily analyze the behavior of the monitored system to identify a defect or anomaly.

Forces:

- The interactions between the monitoring and monitored systems may interfere with the operation of the monitored system, specifically its performance.
- While the frequency and duration of these interactions must be minimized to reduce the monitoring overhead, the interactions must be able to detect/infer a defect or anomaly in the monitored system.
- The monitoring system's data gathering and defect/anomaly detection/inference must be in time to prevent the activation of an error and a subsequent failure.
- The diagnostic resolution, i.e., the degree of accuracy of the fault diagnosis, must be high to be effective.

Solution: The `Monitoring` structural pattern enables the discovery and treatment of a defect or anomaly in the system, which either has the potential to activate or already has activated to an error and potentially a subsequent failure. The pattern supports methods that attempt to recognize the presence of a defect or anomaly within a monitored system. It prevents its activation or enables its containment and mitigation by notifying the monitored system about a fault with location and type information, such that the anomaly or defect is removed before or after it results in an error or a failure. The solution requires a monitoring system, which may be a subsystem of the monitored system or an external independent system, to observe key parameters of the monitored system and to notify the monitored system when these parameters deviate.

The `Monitoring` structural pattern makes an assessment on what may potentially or actually be wrong with a system. The diagnosis entails making inferences based on the observed behavior of a system and narrowing the search for the root cause of the fault. The assessment is based on the operating behavior of the monitored system. The monitoring system may approach the problem using two strategies:

- *Effect-Cause Fault Diagnosis:* This approach entails observation of the parameters of the overall system for anomalies. When a parameter deviates from a range of values that may be considered as fault-free operation, the monitoring system attempts to determine the root cause. The monitoring system logically partitions the system into subsystems and progressively eliminates the subsystems known to be fault-free. Through this process, it narrows the search for the fault in the system.
- *Cause-Effect Fault Diagnosis:* This approach is based on a set of known fault models and the monitoring system comparing the system parameters with a model developed using fault-free system operation, or using simulations. When observed parameters deviate from the fault-free system operation model, the presence, location, and the type of fault may be inferred.

The components of this pattern are shown in Figure 26.

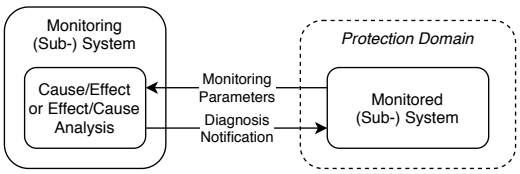
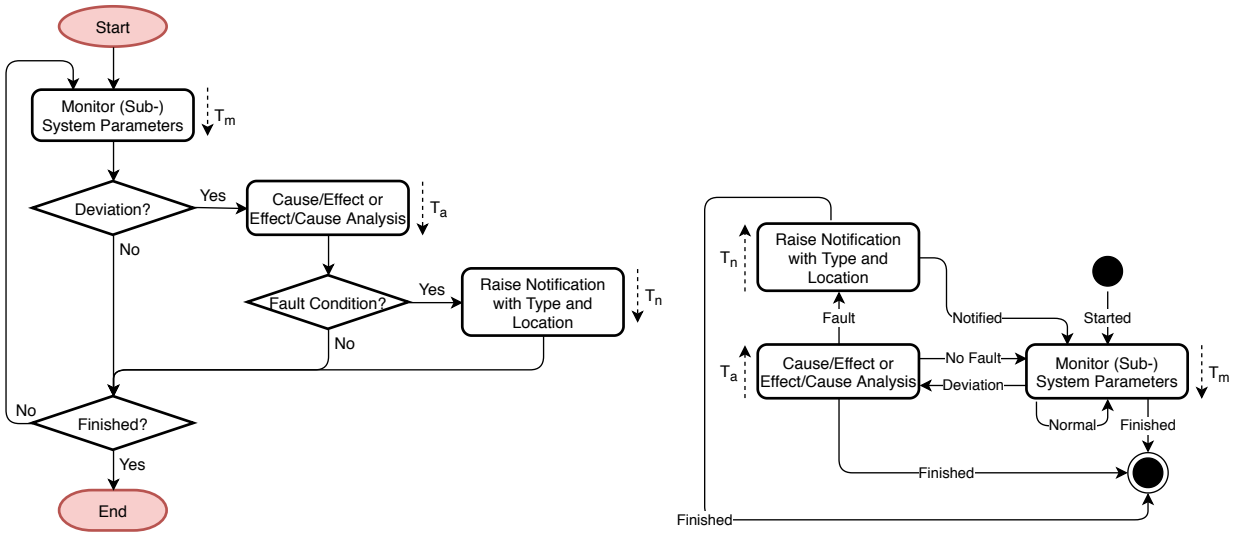


Figure 26. Monitoring pattern components

Capability: This pattern provides fault detection in the monitored system at design or run time, before it eventually results in an error or failure, or after activation, by identifying deviations in monitored parameters. This pattern enables containment and mitigation of a future, imminent or present error or failure in the monitored system through detection and by notifying the monitored system about a fault with location and type information. A system using this pattern is able to detect faults and take corrective action at design and run time. This pattern provides fault detection in the monitored system at design or run time, before it eventually results in an error or failure or after activation, by identifying deviations in monitored parameters and performing effect-cause or cause-effect fault diagnosis.

This pattern enables containment and mitigation of a future, imminent or present error or failure in the monitored system through detection and by notifying the monitored system about a fault with location and type information based on effect-cause or cause-effect inferences. The flowchart and state diagram of the pattern is shown in Figure 27 and its parameters in Table 13.



(a) Flowchart **(b) State diagram**

Figure 27. Monitoring pattern flowchart and state diagram

Table 13. Monitoring pattern parameters

Parameter	Definition
T_m	Time to monitor (sub-) system parameters, including wait and probe times
T_a	Time to perform the cause/effect or effect/cause analysis
T_n	Time to raise notification with type and location

Protection Domain: The protection domain extends to the monitored system.

Resulting Context: The discovery and treatment of a defect or anomaly in the system that has the potential to activate are enabled, which can be used to prevent, contain and mitigate an error or a failure in the system. The `Monitoring` structural pattern requires identifying system parameters that indicate the presence of a fault. The overall system design must include a monitoring system, which introduces design complexity. When the monitoring system is extrinsic to the monitored system, the design effort may be simplified, but the interfaces between the monitoring and monitored systems must be well-defined. When the monitoring system is intrinsic to the design of the monitored system, design complexity increases due to the need to interface the monitoring and monitored subsystems.

A trade-off exists between interference with the operation of the monitored system caused by the frequency and duration of the interactions between the monitoring and monitored systems, and the ability to detect/infer a defect or anomaly in the monitored system. A high performing solution reduces the interference, while maintaining a reasonable ability to detect/infer a defect or anomaly.

The `Monitoring` structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation and require or can make use of fault detection. It detects the presence of a defect or anomaly and reports it, but does act to remedy the fault. Based on the monitored system design and accessibility of the parameters selected for observation, the diagnosis may not be very precise and may sometimes give no indication. An efficient implementation performs effect-cause or cause-effect analysis with a reasonable degree of precision and recall.

Performance: In case when monitoring system is not a part of monitored system, monitoring system doesn't impact the task total execution time T_E. If monitoring system is a part of monitored system, it can impact the task total execution time by δ, the ratio of available resources utilized by monitored system. The performance T when monitoring system is a part of monitored system is defined by Equation 24.

$$T = \frac{T_E}{1 - \delta} \tag{24}$$

Reliability: As `Monitoring` pattern is not impacted by error or failure, the reliability remain same as per Equation 25.

$$R(t) = e^{-\lambda t} \tag{25}$$

Availability: The availability of `Monitoring` pattern can be calculated using the task's total execution time without `Monitoring` pattern T_E and the performance with `Monitoring` pattern T (Equation 26). T_E is PU and T is PU, SD and UD.

$$A = \frac{t_{pu}}{t_{pu} + t_{ud} + t_{sd}} \tag{26}$$

Examples:

- Hardware sensors in the CPUs, on the board, and in the power supply provide real-time operational data about temperature, fan speeds, and voltages. Software sensors in the Linux OS offer real-time operational data about CPU and memory load. The data is made available by sensor interface tools, such

as IPMI [10], such that a monitoring tool, like the Ganglia Monitoring System [123], can aggregate it with time stamps in a central database. The monitoring tool can be configured to send a notification when a certain threshold value for a sensor is exceeded.
- A heartbeat monitoring tool checks the liveness of an MPI process in the system by regularly sending a message and waiting for a reply [147]. If no message is received back within a certain time period, the MPI process is considered failed and the MPI programming model runtime software is informed. This heartbeat monitoring is performed between all MPI processes, where all MPI processes watch each other. A message sent by the application can be used as a liveness request or reply message.
- An MPI tracing tool, like Vampir [8], is used to capture detailed time-stamped information about an application's MPI messages at execution time. The created MPI message trace contains time-stamped information about when an MPI function was called and when it was completed, such that it can be analyzed for inefficiencies in the MPI application's messaging behavior. A common root cause for MPI application scaling limitations is the inefficient use of MPI collectives. The MPI application developer is informed by the tool about such situations.

Rationale: The `Monitoring` structural pattern enables a system to discover and treat a defect or anomaly in the system that has the potential to activate and become an error or failure or already has activated. It relies on a monitoring system to observe the monitored system, identifying deviations in monitored parameters. A discovered defect or anomaly is treated by raising a notification about a fault with location and type information, permitting the monitored system to provide containment and mitigation. A key benefit of this pattern is preemptive discovery of faults in the system, before they are activated and result in errors or failures. Preventive actions taken upon such discovery avoid the need for expensive error/failure recovery, compensation, or correction actions.

Related Patterns: The `Monitoring` and `Prediction` structural pattern are both derivatives of the `Fault Diagnosis` architectural pattern. While the `Monitoring` structural pattern detects existing faults in the system that already have or have not been activated to an error, the `Prediction` structural pattern primarily predicts future faults that have not been activated based on the idea that such future faults have detectable precursors. The `Prediction` structural pattern may also detect existing faults in the system that already have or have not been activated to an error. Both patterns use different methods to achieve their goals.

Known Uses: HPC system installations use monitoring tools for collecting operational data, such as sensor data (e.g., temperature, fan speeds, voltages, computational load, memory and storage usage, etc.), performance data (e.g., application execution, message and file system access times, etc.) and event data (e.g., anomalous, error and failure events, debug messages, etc.). Popular solutions include:

- Hard- and software sensors in the CPUs, on the board, in the power supply, in the OS, and in system services that provide real-time operational data to sensor interface tools. Examples are the hardware sensors in the IBM Power 8 [94] and Intel Xeon [54] processors, and the Linux OS monitoring capabilities.
- Sensor interface tools that gather real-time operational data from hard- and software sensors, such that they are available for monitoring tools. Examples are SMART [173] and IPMI [10].
- Monitoring tools that gather time-stamped information from sensor interface tools in a distributed computing system at a central location, such that they can be analyzed. Examples are the Supermon High-Speed Cluster Monitoring System [171], the Ganglia Monitoring System [123], Nagios [4], and the OVIS Lightweight Distributed Monitoring System [14].
- Heartbeat monitoring tools for liveness checking of individual nodes in the system [147] or MPI processes [26].
- System event loggers that gather time-stamped information from the OS or system services in a distributed computing system, such that they can be analyzed. The most prominent example is Linux's

syslog [6].

- MPI tracing tools gather detailed time-stamped information about an application's MPI messages, such as when a particular message was sent and when it was received. Examples are Vampir [8] and SCALASCA [7].
- Comprehensive RAS management tools by vendors that support real-time monitoring of an entire HPC system, such as the CRMS [28].

7.4.2 Prediction

Name: Prediction

Problem: Not anticipating the presence, root cause, and impact of a defect or anomaly in the system causes lack of corrective action at design or run time and may eventually result in an error or failure.

Context: The pattern is a derivative pattern of the `Fault Diagnosis` architectural pattern and applies to a system that has the following characteristics:

- The system has well-defined parameters that enable a monitoring system to discover the presence of a defect or anomaly in the behavior of the monitored system.
- The interaction between the monitored and monitoring systems is bounded in terms of time.
- The monitoring system has the capability to store historical data about the behavior of the monitored system to analyze it for defect or anomaly occurrences.

Forces:

- The interactions between the monitoring and monitored systems may interfere with the operation of the monitored system, specifically its performance.
- While the frequency and duration of these interactions must be minimized to reduce the monitoring overhead, the interactions must be able to detect/infer a defect or anomaly in the monitored system.
- The monitoring system's data gathering and defect/anomaly detection/inference must be in time to prevent the activation of an error and a subsequent failure.
- The diagnostic resolution, i.e., the degree of accuracy of the fault diagnosis, must be high to be effective.

Solution: The `Prediction` structural pattern enables the discovery and treatment of a defect or anomaly in the system, which has the potential to activate an error and potentially a subsequent failure. The pattern supports methods that attempt to recognize the potential of a future defect or anomaly within a monitored system. It prevents its activation or enables its containment and mitigation by notifying the monitored system about a future fault with location and type information, such that the anomaly or defect is removed before or after it results in an error or a failure.

The solution requires a monitoring system, which may be a subsystem of the monitored system or an external independent system, to observe key parameters of the monitored system and to notify the monitored system when these parameters deviate. The pattern identifies anomalous behavior that indicates the potential for a future fault, which may result in an error or failure in the system. The monitoring system may approach the problem using two strategies:

- *Statistical method:* The monitoring system discovers probabilistic characteristics of a potential fault in the system using statistical inference techniques that examine correlations with previous events.
- *Rule-based method:* The monitoring system builds rules of association to capture the causal correlations between system parameter values and faults.

The monitoring system contains the following components to predict faults in the monitored system:

- *Filter/Preprocessor*: This component removes incomplete monitoring data and duplicates and produces a consistent monitoring data format for analysis.
- *Regression*: This component analyzes the monitoring parameter values and establishes relationships between them.
- *Knowledge Base*: This storage component maintains the rules or statistical properties and models to be used for online prediction of faults using real-time monitoring data captured from the monitored system.

The components of this pattern are shown in Figure 28.

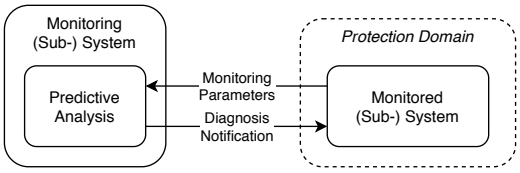
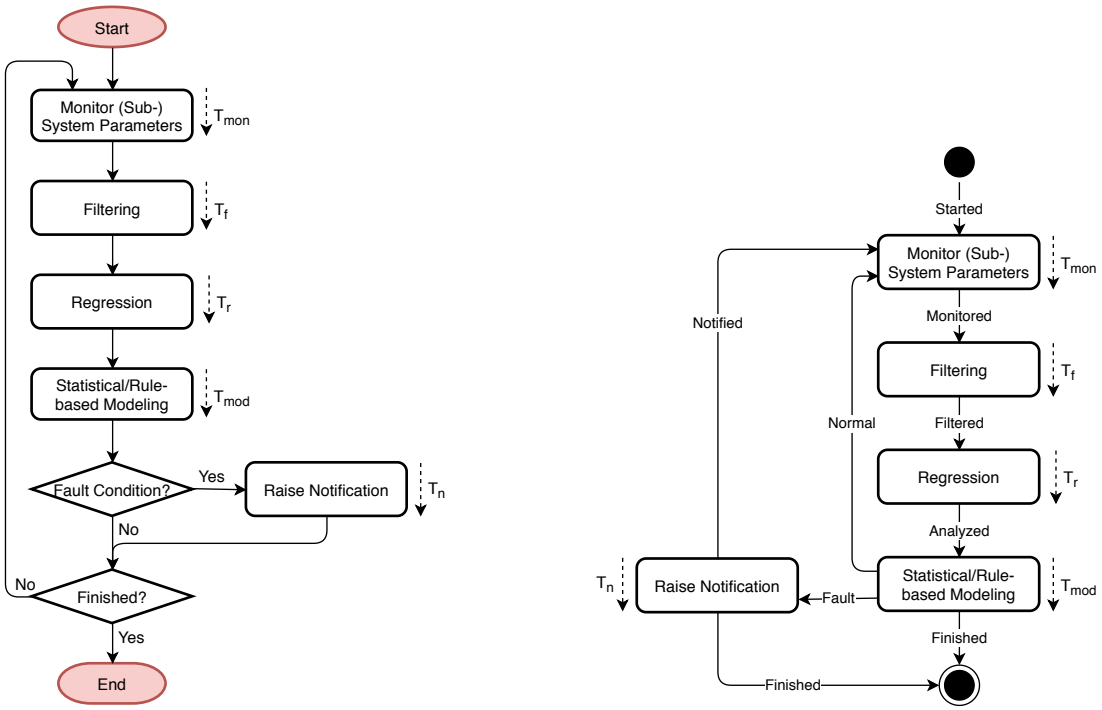


Figure 28. Prediction pattern components

Capability: This pattern provides fault anticipation in the monitored system at design or run time, before it eventually results in an error or failure or after activation, by identifying deviations in monitored parameters. This pattern enables containment and mitigation of a future, imminent or present error or failure in the monitored system through prediction and by notifying the monitored system about a future fault with location and type information. A system using this pattern is able to predict faults and take corrective action at design and run time. This pattern provides fault prediction/detection in the monitored system at design or run time, before it eventually results in an error or failure or after activation, by identifying deviations in monitored parameters and performing regression and statistical/rule-based modeling.

This pattern enables containment and mitigation of a future, imminent or present error or failure in the monitored system through detection and by notifying the monitored system about a fault with location and type information based on regression and statistical/rule-based modeling. The flowchart and state diagram of the pattern is shown in Figure 29 and its parameters in Table 14.



(a) Flowchart **(b) State diagram**

Figure 29. Prediction pattern flowchart and state diagram

Protection Domain: The protection domain extends to the monitored system.

Table 14. Prediction pattern parameters

Parameter	Definition
T_{mon}	Time to monitor (sub-) system parameters, including wait and probe times
T_f	Time to perform the filtering
T_r	Time to perform the regression
T_{mod}	Time to perform the statistical/rule-based modeling
T_n	Time to raise notification

Resulting Context: The discovery and treatment of a defect or anomaly in the system that has the potential to activate are enabled, which can be used to prevent, contain, and mitigate an error or a failure in the system. The `Prediction` structural pattern requires identifying system parameters that indicate the potential for a fault. The overall system design must include a monitoring system, which introduces design complexity. When the monitoring system is extrinsic to the monitored system, the design effort may be simplified, but the interfaces between the monitoring and monitored systems must be well-defined. When the monitoring system is intrinsic to the design of the monitored system, design complexity increases due to the need to interface the monitoring and monitored subsystems.

A trade-off exists between interference with the operation of the monitored system caused by the frequency and duration of the interactions between the monitoring and monitored systems, and the ability to detect/infer a defect or anomaly in the monitored system. A high performing solution reduces the interference, while maintaining a reasonable ability to detect/infer a defect or anomaly.

The `Prediction` structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation and require or can make use of fault prediction/detection. It detects the presence of a defect or anomaly and reports it, but does act to remedy the future fault. Based on the monitored system design and accessibility of the parameters selected for observation, the diagnosis may not be very precise and may sometimes give no indication. An efficient implementation performs regression and statistical/rule-based modeling with a reasonable degree of precision and recall.

Performance: When the monitoring system is a part of the monitored system, the failure-free performance $T_{f=0}$ of the `Prediction` pattern is defined by the task's total execution time without any resilience strategy T_E, the time to monitor sub-system parameters, including wait and probe times T_{mon}, the time to perform the filtering T_f, the time to perform the regression T_r, and the time to perform the statistical/rule-based modeling T_{mod} with the total number of input-execute-output cycles P. The performance under failure T is defined by $T_{f=0}$, plus the time T_n to raise notification with type and location, where the total time to raise notification with type and location is number of faults time T_n. Assuming constant times T_{mon} (t_{mon}), T_f (t_f), T_r (t_r), T_{mod} (t_{mod}), and T_n, T can be defined by Eq. 27.

$$T = T_E + P(t_{mon} + t_f + t_r + t_{mod}) + \frac{T_E}{M}(T_n) \tag{27}$$

Reliability: As `Prediction` pattern is not impacted by error or failure, the reliability remain same as per Equation 28.

$$R(t) = e^{-\lambda t} \tag{28}$$

Availability: The availability of `Prediction` pattern can be calculated using the task's total execution time without `Prediction` pattern T_E and the performance with `Prediction` pattern T (Equation 29). T_E is PU and T is PU, SD and UD.

$$A = \frac{t_{pu}}{t_{pu} + t_{ud} + t_{sd}} \tag{29}$$

Examples:

- In proactive fault tolerance, an OODA loop control is employed that utilizes monitoring tools for collecting sensor data (e.g., temperature, fan speeds, voltages, computational load, memory and storage usage, etc.). It leverages the warning thresholds of these sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72]. The mitigation may use process-level [187] or VM level [134] migration.
- Monitoring tools collecting event data (e.g., anomalous, error and failure events, debug messages, etc.) are used in conjunction with tools for temporal and spatial filtering to identify event correlations and to predict failures [116].
- Probabilistic networks are utilized for establishing correlations between event collected with monitoring tools to predict failures [159].

Rationale: The `Prediction` structural pattern enables a system to discover and treat a defect or anomaly in the system that has the potential to activate and become an error or failure. It relies on a monitoring system to observe the monitored system, identifying deviations in monitored parameters. A discovered defect or anomaly is treated by raising a notification about a future fault with location and type information, permitting the monitored system to provide containment and mitigation. A key benefit of this pattern is prediction of faults in the system, before they are activated and result in errors or failures. Preventive actions taken upon such discovery avoid the need for expensive error/failure recovery, compensation, or correction actions.

Related Patterns: The `Monitoring` and `Prediction` structural pattern are both derivatives of the `Fault Diagnosis` architectural pattern. While the `Monitoring` structural pattern detects existing faults in the system that already have or have not been activated to an error, the `Prediction` structural pattern primarily predicts future faults that have not been activated based on the idea that such future faults have detectable precursors. The `Prediction` structural pattern may also detect existing faults in the system that already have or have not been activated to an error. Both patterns use different methods to achieve their goals.

Known Uses: Prediction is mostly a research area and less used in practice. Most work focuses on proof-of-concept prototypes that utilize an offline learning approach based on system monitoring and failure logs. Known uses include:

- Utilizing a wide-variety of methods to analyze monitoring and log data to predict failures [105]
- Using machine learning to predict GPGPU memory errors [139]
- Leveraging warning thresholds of sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72]
- Characterizing the correlation between temperature, power consumption and memory or logic errors using monitoring and log data [138]
- Using probabilistic networks for establishing event correlations for prediction algorithms [159]
- Employing temporal and spatial filtering for failure prediction [116]

7.4.3 Restructure

Name: Restructure

Problem: A hardware or software error or subsystem failure due to a physical fault (e.g., wear-out or destruction) or a design fault (e.g., resource underprovisioning) in an HPC environment causes a software, such as an application, to experience an error or failure.

Context: The pattern is a derivative of the `Reconfiguration` architectural pattern and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has well-defined parameters that enable a monitoring system to discover the presence of an existing or future fault, error, or failure in the behavior of the monitored system.
- The interaction between the monitored and monitoring systems is bounded in terms of time.
- The monitoring system has the capability to readily analyze the behavior of the monitored system to detect or predict a fault, error, or failure.
- The system is able to be partitioned into logical subsystems, where altering the interconnection between the subsystems enables the system to remain operational.

Forces:

- The interactions between the monitoring and monitored systems may interfere with the operation of the monitored system, specifically its performance.
- While the frequency and duration of these interactions must be minimized to reduce the monitoring overhead, the interactions must be able to detect or predict a fault, error, or failure in the monitored system.
- The diagnostic resolution, i.e., the degree of accuracy of the fault diagnosis, must be high to be effective.
- The ability of the system to alter the interconnection among subsystems must permit system operation that is functionally equivalent to the fault, error, or failure-free operation of the system.
- The restructuring may force the system to operate at a lower level of performance.

Solution: The `Restructure` structural pattern alleviates the impact of a fault, error, or failure on system operation by changing the interconnection between the subsystems in the overall system. It has a detection component that is similar to the `Monitoring` or `Prediction` structural patterns and an additional containment and mitigation component that acts upon the notification from the detection component.

The detection component enables the discovery of an existing or future fault, error, or failure in the system. It enables its containment and mitigation by notifying the system about it with location and type information. The solution requires a monitoring system, which may be a subsystem of the monitored system or an external independent system, to observe key parameters of the monitored system and to notify the monitored system when these parameters deviate. This pattern does not specify the detailed method of detection or prediction, which may be similar to the `Monitoring` or `Prediction` structural patterns.

The containment and mitigation component acts upon the notification from the detection component by modifying the interconnection between the N subsystems in the overall system to permanently isolate (exclude) the subsystem affected by the predicted or detected fault, error, or failure. The system assumes an $N-1$ subsystem configuration in response to a notification, which is characterized by its own interconnection topology and retains functional equivalency with the original system configuration. The components of this pattern are illustrated in Figure 30.

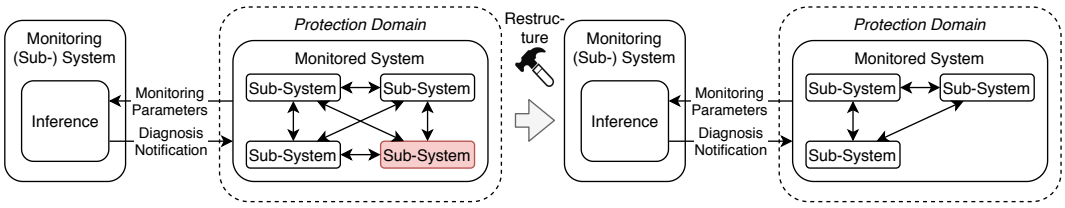
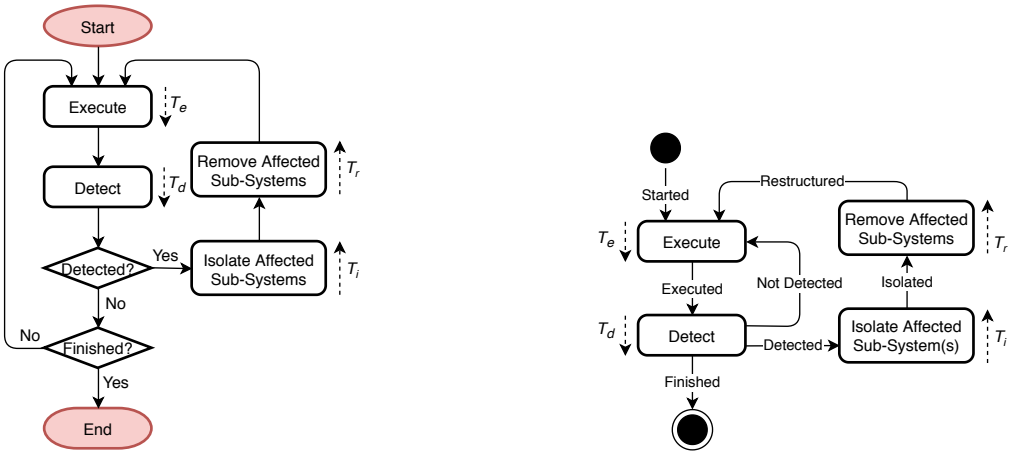


Figure 30. Restructure pattern components

Capability: A system using this pattern is able to continue to operate in the presence of a non-permanent fault, error, or failure with some interruption. This pattern provides detection/prediction, containment, and mitigation of a future or existing fault, error, or failure in the system by changing the interconnection between the subsystems in the overall system to a functionally equivalent configuration, permanently isolating (excluding) the affected subsystem. The flowchart and state diagram of the pattern is shown in Figure 31 and its parameters in Table 15.



(a) Flowchart **(b) State diagram**

Figure 31. Restructure pattern flowchart and state diagram

Table 15. Restructure pattern parameters

Parameter	Definition
T_e	Time to execute system progress
T_d	Time to detect or predict a fault, error or failure
T_i	Time to isolate the affected subsystem(s)
T_r	Time to remove the affected subsystem(s)

Protection Domain: The protection domain extends to the monitored system's state that is not lost due to a subsystem exclusion and to all of its resources that are able to be operate correctly after a restructuring.

Resulting Context: A fault, error, or failure is prevented from affecting the correct operation of the system. While functional equivalency to the original system configuration is maintained through reconfiguration, progress in the system may be lost. The system is interrupted during reconfiguration in response to a detected or predicted fault, error, or failure, but is not interrupted during fault-, error-, and failure-free operation. The restructuring may force the system to operate at a lower level of performance. After

reconfiguration, the system's ability to prevent a fault, error or failure from affecting the correct operation of the system remains the same if it retains the capability to assume an $N-1$ system configuration with functional equivalency (where N is the new number of subsystems after the previous reconfiguration).

The `Restructure` structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the `Restructure` structural pattern covers for the remaining error/failure types.

Performance: In case when monitoring system is a part of monitored system, the failure-free performance $T_{f=0}$ of the `Restructure` pattern is defined by the task's total execution time without any resilience strategy T_E and the time to detect or predict a fault, error or failure T_d with the total number of input-execute-output cycles P. The performance under failure T is defined by $T_{f=0}$, plus the time T_i to isolate the affected subsystem(s) and the time T_r to remove the affected subsystem(s), where the total time to isolate the affected subsystem(s) and to remove the affected subsystem(s) is number of faults, errors, or failures time T_i and T_r. Assuming constant times T_d (t_d), T_i, and T_r, T can be defined by Eq. 30.

$$T = T_E + P(t_d) + \frac{T_E}{M}(T_i + T_r) \tag{30}$$

Reliability: Given that the `Restructure` pattern enables the resumption of correct operation after an error or failure, the reliability of a system employing it is defined by errors and failures that are not handled by the pattern, such as failures of the persistent storage system. The reliability after applying the `Restructure` pattern $R(t)$ can be obtained using the performance under failure T and the failure rate λ_u (or MTTF M_u) of the unprotected part of the system (Eq. 31).

$$R(t) = e^{-\lambda_u T} = e^{-T/M_u} \tag{31}$$

Availability: The availability of `Restructure` pattern can be calculated using the task's total execution time without `Restructure` pattern T_E and the performance with `Restructure` pattern T (Equation 32). T_E is PU and T is PU, SD and UD.

$$A = \frac{t_{pu}}{t_{pu} + t_{ud} + t_{sd}} \tag{32}$$

Examples: Unhealthy, erroneous, or failed system resources, such as compute nodes or NVIDIA GPGPU memory pages [141], are removed from the system's pool of resources though restructuring. A programming model, such as the proposed ULFM extension to the MPI standard [31], permits an application to restructure the programming environment to handle resource failures.

Rationale: The `Restructure` structural pattern prevents an existing or future fault, error, or failure from affecting the correct operation of the system through changing the interconnection between N subsystems in the overall system. It relies on the ability to assume an $N-1$ functionally equivalent system configuration in response to a detected or predicted fault, error, or failure that retains functional equivalency with the original system configuration. The pattern may perform proactive actions, such as

restructuring the system or subsystem before a fault, error, or failure impacts its operation, or reactive actions, such as restructuring the system or subsystem upon such an event.

Progress in the system may be lost. Fault, error, or failure detection/prediction is part of the pattern, similar to the `Monitoring` or `Prediction` structural patterns. The containment and mitigation offered by this pattern are independent from the type of fault, error, or failure. The pattern has some design complexity due to the need to partition the system into logical subsystems. It has low dependence on a system's architecture.

Related Patterns: In contrast to the `Monitoring` and `Prediction` structural patterns, the `Restructure` structural pattern additionally offers containment and mitigation. In contrast to many other structural patterns providing containment and mitigation, the `Restructure` structural pattern does not restore previously preserved system state, employ redundancy, or self-mask or self-correct illegal system state.

The `Restructure`, `Rejuvenation`, and `Reinitialization` structural patterns are derived from the `Reconfiguration` architectural pattern. They provide the same detection/prediction capability, but offer different containment and mitigation capabilities. The `Restructure` structural pattern reconfigures a system of N subsystems to $N-1$ functionally equivalent subsystems. The `Rejuvenation` structural pattern restores the operation of the system or affected subsystem using previously preserves system state. The `Reinitialization` structural pattern resets the system or affected subsystem to its initial state.

Known Uses:

- HPC systems often automatically take unhealthy, erroneous, or failed compute nodes out of the system's resource manager's pool of available computing resources to avoid scheduling jobs on them and to permit maintenance (e.g., the the NodeKARE module in CLE [102]).
- The recurrence of memory errors in a specific cell or region of the chip often indicates degradation of the memory module or region. NVIDIA GPGPU drivers support dynamic page retirement, which removes a memory page from the pool of available physical memory resources [141].
- The proposed ULFM extension to the MPI standard [31] allows parallel applications to get notifications of process failures and to restructure the MPI environment. ULFM permits revoking an MPI communicator to prevent any further usage. It also allows shrinking an MPI communicator to exclude failed MPI processes from future communication.
- The Charm++ adaptive runtime system interacts with the resource manager to dynamically reconfigure the number of nodes assigned to a job based on the resilience, power, and performance characteristics of the system [11].

7.4.4 Rejuvenation

Name: Rejuvenation

Problem: A hardware or software error or subsystem failure due to a physical fault (e.g., wear-out or destruction) or a design fault (e.g., resource underprovisioning) in an HPC environment causes a software, such as an application, to experience an error or failure.

Context: The pattern is a derivative of the `Reconfiguration` architectural pattern and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has well-defined parameters that enable a monitoring system to discover the presence of an existing or future fault, error, or failure in the behavior of the monitored system.
- The interaction between the monitored and monitoring systems is bounded in terms of time.
- The monitoring system has the capability to readily analyze the behavior of the monitored system to detect or predict a fault, error, or failure.
- The experienced fault, error, or failure must not be persistent.
- The system is capable of compartmentalizing its state that is accurately representative of the progress of the system since initialization at the time such state is captured.
- The system operation has well-defined intervals that enable the pattern to transition the system to a known correct interval in response to an error or failure.
- The system is capable of saving its current state and loading a previously saved state.

Forces:

- The interactions between the monitoring and monitored systems may interfere with the operation of the monitored system, specifically its performance.
- While the frequency and duration of these interactions must be minimized to reduce the monitoring overhead, the interactions must be able to detect or predict a fault, error, or failure in the monitored system.
- The diagnostic resolution, i.e., the degree of accuracy of the fault diagnosis, must be high to be effective.
- The ability of the system to rejuvenate subsystems or the entire system must permit system operation that is functionally equivalent to the fault-, error-, and failure-free operation.
- The pattern requires additional persistent storage to capture system state, which increases overhead in terms of resources required by the system.
- The creation frequency of system state snapshots determines overhead. More frequent snapshot creation increases system execution time, but reduces the amount of lost work upon an error or failure.
- The time interval for the recovery of a system from a snapshot as well as the time interval to create a snapshot must be less than the system's MTBF to guarantee forward progress.

Solution: The `Rejuvenation` structural pattern alleviates the impact of a fault, error, or failure on system operation by restoring the affected subsystem or system to a known correct state. It has a detection component that is similar to the `Monitoring` or `Prediction` structural patterns and an additional containment and mitigation component that acts upon the notification from the detection component and is similar to the `Rollback` or `Rollforward` structural patterns.

The detection component enables the discovery of an existing or future fault, error, or failure in the system. It enables its containment and mitigation by notifying the system about it with location and type information. The solution requires a monitoring system, which may be a subsystem of the monitored

system or an external independent system, to observe key parameters of the monitored system and to notify the monitored system when these parameters deviate. This pattern does not specify the detailed method of detection or prediction, which may be similar to the `Monitoring` or `Prediction` structural patterns.

The containment and mitigation component acts upon the notification from the detection component by restoring the affected subsystem or system to a known correct state. The pattern relies on the creation of system state snapshots and the maintenance of such snapshots on a persistent storage system that is not affected by the fault, error, or failure, similar to the `Rollback` or `Rollforward` structural patterns. This pattern does not specify the detailed method of containment and mitigation.

Upon notification, the most recent snapshot is used to recreate the last known correct state of the affected subsystem or system. When the state is recovered, the operation of the system is resumed. Based on a temporal view of the system's progress, the error/failure recovery may be either backward to the time when the snapshot occurred (rollback) or forward to the time when the event was detected or predicted (rollforward).

Undetected (latent) errors that are either detected later or result in a different detected error or failure later represent a problem, as the most recent or even more snapshots may contain an illegal system state. In this case, the most recent correct snapshot may be used to recreate the last known error/failure-free state of the system, skipping snapshots containing illegal state and going further back in time in terms of when the snapshot was made.

The pattern may create subsystems out of the system to temporarily isolate, but not permanently exclude, the affected subsystem. The interconnection between the subsystems is restored after subsystem or system rejuvenation. The components of this pattern are illustrated in Figure 32.

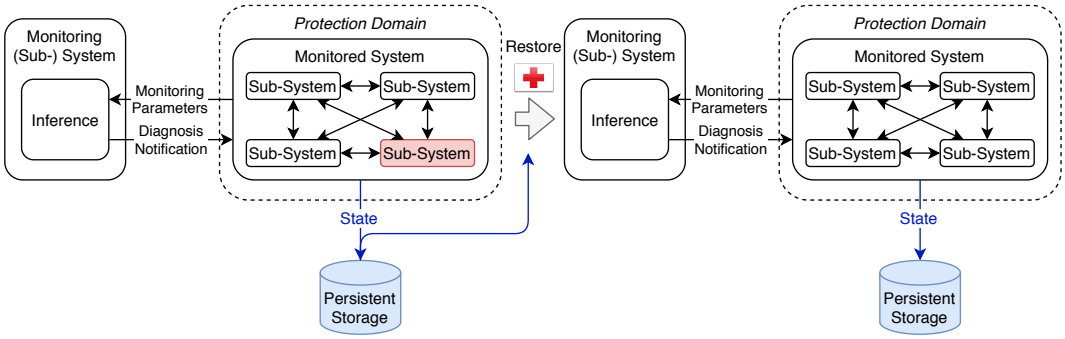
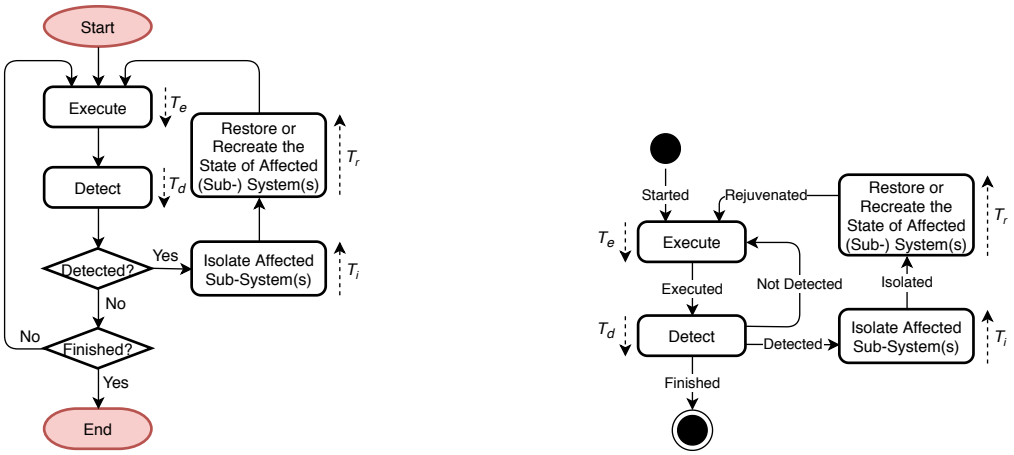


Figure 32. Rejuvenation pattern components

Capability: A system using this pattern is able to continue to operate in the presence of a non-permanent fault, error, or failure with some interruption and some or no loss of progress. This pattern provides detection/prediction, containment, and mitigation of a future or existing fault, error, or failure in the system by restoring the affected subsystem or system to a known correct state. The flowchart and state diagram of the pattern is shown in Figure 33 and its parameters in Table 16.

Protection Domain: The protection domain extends to the monitored system's state that is not lost due to a subsystem or system rejuvenation and to all of its resources that are able to be operate correctly after a rejuvenation.

Resulting Context: A fault, error, or failure is prevented from affecting the correct operation of the system. While functional equivalency to the original system configuration is maintained through rejuvenation, progress in the system may be partially lost. The system is interrupted during rejuvenation in response to a



(a) Flowchart **(b) State diagram**

Figure 33. Rejuvenation pattern flowchart and state diagram

Table 16. Rejuvenation pattern parameters

Parameter	Definition
T_e	Time to execute system progress
T_d	Time to detect or predict a fault, error or failure
T_i	Time to isolate the affected subsystem(s)
T_r	Time to restore or replace the state of the affected (sub-) system(s)

detected or predicted fault, error, or failure. It is also interrupted during fault-, error-, and failure-free operation for preserving system state. After rejuvenation, the system's ability to prevent a fault, error, or failure from affecting the correct operation of the system remains the same.

The `Rejuvenation` structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the `Rejuvenation` structural pattern covers for the remaining error/failure types.

Performance: `Rejuvenation` pattern detection component is same as the Monitoring pattern (Eq. 33). The containment and mitigation component impact the task total execution time same as in Rollback or Rollforward pattern (described later). We define performance using the Rollback pattern. We calculate performance under failure T by adding the time to detect or predict a fault, error, or failure T_d with the total number of input-execute-output cycles P in Eq. 34. T_l, T_r, and T_s represent T_i time to isolate the affected subsystem(s) and T_r time to restore or replace the state of the affected subsystem(s). Assuming constant times T_d (t_d), T_l, T_r, and T_s, T can be defined by Eq. 35.

$$T = \frac{T_E}{1-\delta} \tag{33}$$

$$T = T_E + \left(\frac{T_E}{\tau} - 1\right) T_s + \frac{T_E}{M} T_{e,f}(\tau + T_s) + \frac{T_E}{M}(T_l + T_r),$$

$$\tau = \sqrt{2MT_s} \tag{34}$$

$$T = T_E + P(t_d) + \left(\frac{T_E}{\tau} - 1\right) T_s + \frac{T_E}{M} T_{e,f}(\tau + T_s) + \frac{T_E}{M}(T_l + T_r) \tag{35}$$

Reliability: Given that the `Rejuvenation` pattern enables the resumption of correct operation after an error or failure, the reliability of a system employing it is defined by errors and failures that are not handled by the pattern, such as failures of the persistent storage system. The reliability after applying the `Rejuvenation` pattern $R(t)$ can be obtained using the performance under failure T and the failure rate λ_u (or MTTF M_u) of the unprotected part of the system (Eq. 36).

$$R(t) = e^{-\lambda_u T} = e^{-T/M_u} \tag{36}$$

Availability: The availability of `Rejuvenation` pattern can be calculated using the task's total execution time without `Rejuvenation` pattern T_E and the performance with `Rejuvenation` pattern T (Equation 37). T_E is PU and T is PU, SD and UD.

$$A = \frac{t_{pu}}{t_{pu} + t_{ud} + t_{sd}} \tag{37}$$

Examples: The targeted rejuvenation of data structures in system software, such as OS data structures, permits avoidance of and recovery from errors or failures without the need to reinitialize the affected compute node or the complete HPC system. The individual rejuvenation of HPC system services, such as the parallel file system MDS or the system's resource manager, allows dealing with errors or failures without the need to reinitialize the entire HPC system.

Rationale: The `Rejuvenation` structural pattern prevents an existing or future fault, error, or failure from affecting the correct operation of the system through restoring the operation of the system or the affected subsystem. It relies on the capability to preserve system state before a detected or predicted fault, error, or failure, often in a periodic fashion, and restore the previously preserved system state upon detection or prediction of such an event to resume operation from a known correct state.

The pattern performs proactive actions, such as preserving system state, but mostly relies on reactive actions after notification about a detected or predicted fault, error, or failure. Progress in the system may be partially lost. Fault, error, or failure detection/prediction is part of the pattern, similar to the `Monitoring` or `Prediction` structural patterns. Containment and mitigation is part of the pattern, similar to the `Rollback` or `Rollforward` structural patterns. The containment and mitigation offered by this pattern are independent from the type of fault, error, or failure. The pattern has very little to some design complexity and has low dependence on a system's architecture.

Related Patterns: In contrast to the `Monitoring` and `Prediction` structural patterns, the `Rejuvenation` structural pattern additionally offers containment and mitigation. In contrast to the `Rollback` and `Rollforward` structural patterns, the `Rejuvenation` structural pattern does not necessarily restore the preserved state of the entire system, as only the affected subsystem's preserved state may be restored. In contrast to the other structural patterns providing containment and mitigation, the `Rejuvenation` structural pattern does not employ redundancy, or self-mask or self-correct illegal system state.

The `Restructure`, `Rejuvenation`, and `Reinitialization` structural patterns are derived from the `Reconfiguration` architectural pattern. They provide the same detection/prediction capability, but offer different containment and mitigation capabilities. The `Restructure` structural pattern reconfigures a system of N subsystems to $N-1$ functionally equivalent subsystems. The `Rejuvenation` structural pattern restores the operation of the system or affected subsystem using previously preserves system state. The `Reinitialization` structural pattern resets the system or affected subsystem to its initial state.

Known Uses:

- The Mini-Ckpts framework [79] enables HPC applications to survive OS errors and failures. The OS is effectively rebooted, while the application and OS state directly relevant to the application is preserved in persistent memory. This rejuvenates all OS state that is not directly relevant to the application.
- Individual services, such as the parallel file system MDS or the system's resource manager, may be rebooted to fix erroneous behavior or failures. The correct service state is maintained on persistent storage to allow for resumption after reboot.
- OS structures may be regularly verified for correctness and rejuvenated if the correctness test fails. Inherent redundancy, such as with doubly linked lists, or other methods, such as using the known size of OS structures, are used to rejuvenate state, such as pointers.

7.4.5 Reinitialization

Name: Reinitialization

Problem: A hardware or software error or subsystem failure due to a physical fault (e.g., wear-out or destruction) or a design fault (e.g., resource underprovisioning) in an HPC environment causes a software, such as an application, to experience an error or failure.

Context: The pattern is a derivative of the `Reconfiguration` architectural pattern and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has well-defined parameters that enable a monitoring system to discover the presence of an existing or future fault, error, or failure in the behavior of the monitored system.
- The interaction between the monitored and monitoring systems is bounded in terms of time.
- The monitoring system has the capability to readily analyze the behavior of the monitored system to detect or predict a fault, error, or failure.
- The experienced fault, error, or failure must not be persistent.

Forces:

- The interactions between the monitoring and monitored systems may interfere with the operation of the monitored system, specifically its performance.
- While the frequency and duration of these interactions must be minimized to reduce the monitoring overhead, the interactions must be able to detect or predict a fault, error, or failure in the monitored system.
- The diagnostic resolution, i.e., the degree of accuracy of the fault diagnosis, must be high to be effective.
- The ability of the system to reinitialize subsystems or the entire system must permit system operation that is functionally equivalent to the fault-, error-, and failure-free operation.
- The reinitializing a subsystem or system is often a slow process and causes an interruption.

Solution: The `Reinitialization` structural pattern alleviates the impact of a fault, error, or failure on system operation by restoring the affected subsystem or system to its initial state. It has a detection component that is similar to the `Monitoring` or `Prediction` structural patterns and an additional containment and mitigation component that acts upon the notification from the detection component.

The detection component enables the discovery of an existing or future fault, error, or failure in the system. It enables its containment and mitigation by notifying the system about it with location and type information. The solution requires a monitoring system, which may be a subsystem of the monitored system or an external independent system, to observe key parameters of the monitored system and to notify the monitored system when these parameters deviate. This pattern does not specify the detailed method of detection or prediction, which may be similar to the `Monitoring` or `Prediction` structural patterns.

The containment and mitigation component acts upon the notification from the detection component by restoring the affected subsystem or system to its initial state, i.e., a reset. The pattern may create subsystems out of the system to temporarily isolate, but not permanently exclude, the affected subsystem. The interconnection between the subsystems is restored after subsystem or system reinitialization. The components of this pattern are illustrated in Figure 34.

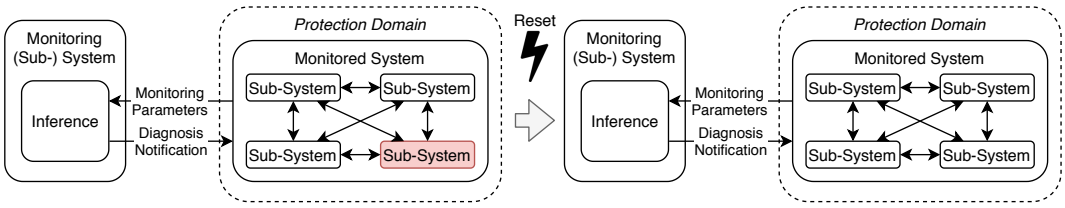
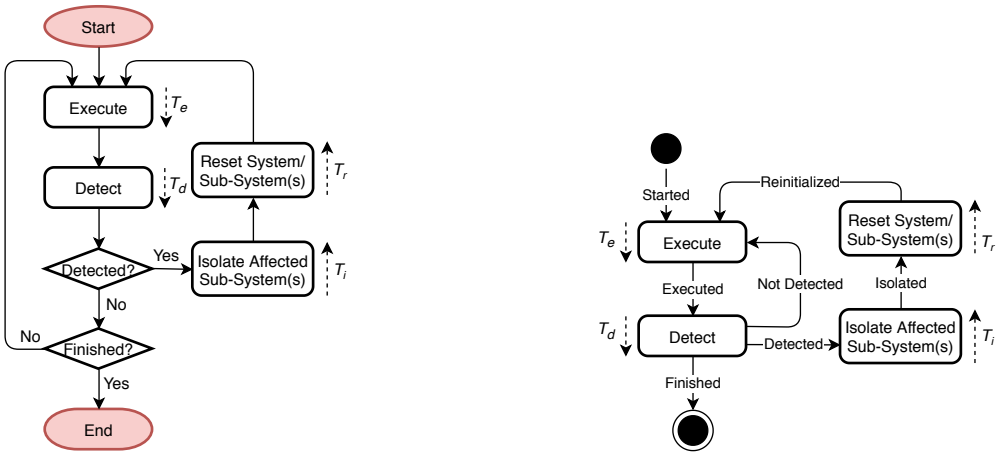


Figure 34. Reinitialization pattern components

Capability: A system using this pattern is able to continue to operate in the presence of a non-permanent fault, error, or failure with significant interruption and all or at least some loss of progress. This pattern provides detection/prediction, containment, and mitigation of a future or existing fault, error, or failure in the system by restoring the affected subsystem or system to its initial state. The flowchart and state diagram of the pattern is shown in Figure 35 and its parameters in Table 17.



(a) Flowchart **(b) State diagram**

Figure 35. Reinitialization pattern flowchart and state diagram

Table 17. Reinitialization pattern parameters

Parameter	Definition
T_e	Time to execute system progress
T_d	Time to detect or predict a fault, error or failure
T_i	Time to isolate the affected subsystem(s)
T_r	Time to reset the entire system or affected subsystem(s)

Protection Domain: The protection domain extends to the monitored system's state that is not lost due to a subsystem or system reinitialization and to all of its resources that are able to be operate correctly after a reinitialization.

Resulting Context: A fault, error, or failure is prevented from affecting the correct operation of the system. While functional equivalency to the original system configuration is maintained through reinitialization, progress in the system is partially or completely lost. The system is interrupted during reinitialization in response to a detected or predicted fault, error, or failure, but is not interrupted during fault-, error-, and failure-free operation. After reinitialization, the system's ability to prevent a fault, error, or failure from affecting the correct operation of the system remains the same.

The Reinitialization structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the Reinitialization structural pattern covers for the remaining error/failure types.

Performance: Reinitialization pattern failure-free performance $T_{f=0}$ is defined by the task's total execution time without any resilience strategy T_E and the time to detect or predict a fault, error, or failure T_d with the total number of input-execute-output cycles P. The performance under failure T is defined by $T_{f=0}$, plus the time T_i to isolate the affected subsystem(s), the time T_r to remove the affected subsystem(s), and the time for work lost (which is assumed to be half of T_E), where the total time to isolate the affected subsystem(s), to remove the affected subsystem(s), and the time for work lost is number of faults, errors, or failures time T_i, T_r, and half of T_E. Assuming constant times T_d (t_d), T_i, and, T_r, T can be defined by Eq. 38.

$$T = T_E + P(t_d) + \frac{T_E}{M}(T_i + T_r + T_E * 0.5) \tag{38}$$

Reliability: Given that the Reinitialization pattern enables the resumption of correct operation after an error or failure, the reliability of a system employing it is defined by errors and failures that are not handled by the pattern, such as failures of the persistent storage system. The reliability after applying the Reinitialization pattern $R(t)$ can be obtained using the performance under failure T and the failure rate λ_u (or MTTF M_u) of the unprotected part of the system (Eq. 39).

$$R(t) = e^{-\lambda_u T} = e^{-T/M_u} \tag{39}$$

Availability: The availability of Reinitialization pattern can be calculated using the task's total execution time without Reinitialization pattern T_E and the performance with Reinitialization pattern T (Equation 40). T_E is PU and T is PU, SD and UD.

$$A = \frac{t_{pu}}{t_{pu} + t_{ud} + t_{sd}} \tag{40}$$

Examples: A malfunctioning compute node may be rebooted.

Rationale: The Reinitialization structural pattern prevents an existing or future fault, error, or failure from affecting the correct operation of the system through restarting the operation of the system or the affected subsystem. It relies on the ability to assume a configuration in response to a detected or predicted fault, error, or failure that retains functional equivalency with the original system configuration. The pattern may perform proactive actions, such as restarting the system or subsystem before a fault, error, or failure impacts its operation, or reactive actions, such as restarting the system or subsystem upon such an event. Progress in the system is partially or completely lost. Fault, error, or failure detection/prediction is part of the pattern, similar to the Monitoring or Prediction structural patterns. The containment and mitigation offered by this pattern are independent from the type of fault, error, or failure. The pattern has very little design complexity and has low dependence on a system's architecture.

Related Patterns: In contrast to the `Monitoring` and `Prediction` structural patterns, the `Reinitialization` structural pattern additionally offers containment and mitigation. In contrast to many other structural patterns providing containment and mitigation, the `Reinitialization` structural pattern does not restore previously preserved system state, employ redundancy, or self-mask or self-correct illegal system state.

The `Restructure`, `Rejuvenation`, and `Reinitialization` structural patterns are derived from the `Reconfiguration` architectural pattern. They provide the same detection/prediction capability, but offer different containment and mitigation capabilities. The `Restructure` structural pattern reconfigures a system of N subsystems to $N - 1$ functionally equivalent subsystems. The `Rejuvenation` structural pattern restores the operation of the system or affected subsystem using previously preserves system state. The `Reinitialization` structural pattern resets the system or affected subsystem to its initial state.

Known Uses:

- Various cluster management software systems, such as the Cray HSS [102], enable malfunctioning nodes in the cluster to be reset. The HSS initiates a reboot sequence for a failing node without disrupting the remaining nodes in the system.

7.4.6 Rollback

Name: Rollback

Problem: A hardware or software error or subsystem failure due to a physical fault (e.g., wear-out or destruction) or a design fault (e.g., resource underprovisioning) in an HPC environment causes a software, such as an application, to experience an error due to loss of or corrupted state, and potentially a subsequent failure.

Context: The pattern is a derivative of the `Checkpoint Recovery` architectural pattern. It applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The error or failure in the system that the pattern handles must be detected, as the pattern offers no error or failure detection.
- The system is capable of compartmentalizing its state that is accurately representative of the progress of the system since initialization at the time such state is captured.
- The system operation has well-defined intervals that enable the pattern to transition the system to a known correct interval in response to an error or failure.
- The system is capable of saving its current state and loading a previously saved state.

Forces:

- The pattern requires additional storage to capture system state or to log messages or events, which increases overhead in terms of resources required by the system.
- The amount of state captured during each creation of a recovery point incurs storage space and execution time overheads.
- The creation frequency of system state snapshots determines overhead. More frequent snapshot creation increases system execution time, but reduces the amount of lost work upon an error or failure.
- The time interval for the recovery of a system from a snapshot as well as the time interval to create a snapshot must be less than the system's MTBF to guarantee forward progress.

Solution: The `Rollback` structural pattern enables the resumption of correct operation of a system impacted by an error or failure. It supports resilient operation by restoring the system to a known correct state in the event of an error or failure. Rollback solutions are classified into checkpoint-based and log-based strategies.

The checkpoint-based strategy relies on the creation of system state snapshots and the maintenance of such checkpoints on a persistent storage system that is not affected by the error or failure. Upon detection of an error or failure, the most recent snapshot is used to recreate the last known error/failure-free state of the system. The log-based strategy relies on logging, i.e., storing the information of, events, such as messages sent between different parts of the system or to the system as input, on a persistent storage system that is not affected by the error or failure. Upon detection of an error or failure, the log is replayed to recreate the last known error/failure-free state of the system. In contrast to the checkpoint-based strategy, the log-based strategy is able to offer resilience in the presence of non-deterministic events and for non-deterministic systems, as replaying the event log deterministically recreates system state.

The checkpoint and log-based strategies may be used together, where events are logged in-between snapshots. With both strategies, the operation of the system is resumed when the system state is recovered. Based on a temporal view of the system's progress, the error/failure recovery is backward to the time when the snapshot occurred. When the system is partitioned into several subsystems, the pattern must coordinate

the process of checkpointing. The pattern may use the following checkpointing methods:

- *Coordinated:* The subsystems to coordinate the process of creating checkpoints. The coordination enables a globally consistent checkpoint state, which simplifies the recovery.
- *Uncoordinated:* The subsystems each independently decides when to create their respective checkpoints. This has the potential to cause a domino effect, where rollbacks propagate among system components due to lack of consistency, potentially back to the initial state.
- *Communication-based:* Each subsystem creates local checkpoints, but periodically also enforces coordinated checkpoints between all subsystems as backstop for the domino effect.

The log-based recovery is based on a piecewise deterministic assumption, in which the system identifies nondeterministic events and the information necessary to replay the event during recovery is captured and logged. The following logging protocols for non-deterministic events may be used by the pattern:

- *Pessimistic:* The protocol assumes that an error or failure occurs after a nondeterministic event in the system. Therefore, the determinant of each nondeterministic event is immediately logged to persistent storage. The error/failure-free overhead of this approach is high.
- *Optimistic:* The determinants are held in a volatile storage and written to persistent storage asynchronously. This protocol makes the optimistic assumption that the logging is completed before the occurrence of an error or failure. The error/failure-free overhead of this approach is low.
- *Causal:* The protocol provides a balanced approach by avoiding immediate writing to persistent storage (much like the optimistic protocol to reduce error/failure-free overhead), but each subsystem commits output independently (like the pessimistic protocol to prevent creation of orphan subsystems).

Undetected (latent) errors that are either detected later or result in a different detected error or failure later represent a problem, as the most recent or even more snapshots may contain an illegal system state. In this case, the most recent correct snapshot may be used to recreate the last known error/failure-free state of the system, skipping snapshots containing illegal state and going further back in time in terms of when the snapshot was made. The components of this pattern are illustrated in Figure 36.

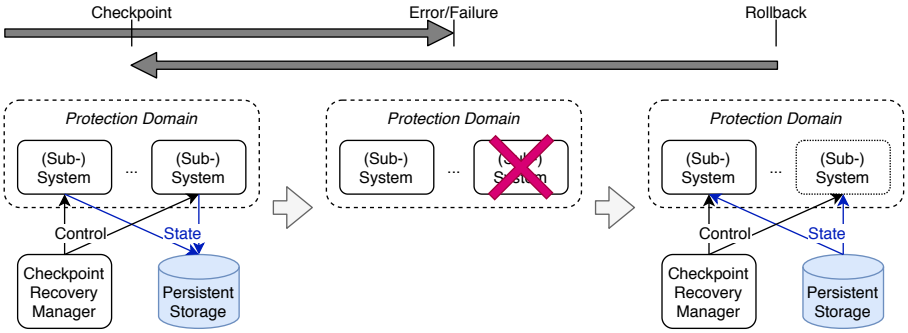
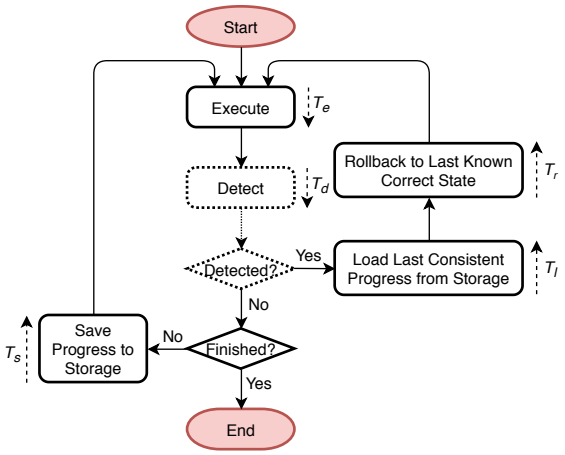
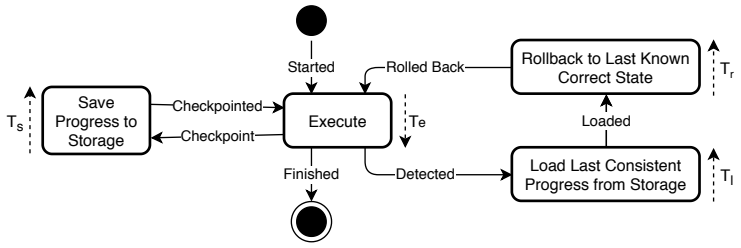


Figure 36. Rollback pattern components

Capability: A system using this pattern is able to continue to operate in the presence of an error or failure with some interruption and some loss of progress. This pattern provides mitigation of an error or failure in the system by preserving system state on a persistent storage system before an error or failure, using a checkpoint-based and or log-based strategy, and restoring the previously preserved system state upon such an event to resume operation from a previously preserved known correct state. The flowchart and state diagram of the pattern is shown in Figure 37 and its parameters in Table 18.



(a) Flowchart



(b) State diagram

Figure 37. Rollback pattern flowchart and state diagram

Table 18. Rollback pattern parameters

Parameter	Definition
T_e	Time to execute (sub-) system progress
T_d	Time to detect an error/failure (not part of this pattern, but shown for completeness)
T_l	Time to load consistent (sub-) system state and progress from storage
T_r	Time to rollback to the last known correct state (initial and/or previously saved)
T_s	Time to save (sub-) system state and progress to storage

Protection Domain: The protection domain extends to the previously preserved system state that can be recreated using the system state captured by snapshots stored on persistent storage.

Resulting Context: Correct operation is resumed after an error or failure impacted the system. Progress in the system is lost after an error or failure, since the recovery is only able to recreate the system state to the time when the snapshot occurred. The system is interrupted during error-/failure-free operation for creating system state snapshots and maintaining them on a persistent storage system. The system is interrupted upon an error or failure for recreating the last known error/failure-free state of the system.

A trade-off exists between the creation frequency of system state snapshots and/or event logging and the corresponding execution time overhead during error-/failure-free operation vs. the amount of lost progress. The optimal solution of this trade-off depends on the MTBF, the time it takes to save and load a system state snapshot, the time it takes to log events, and the time it takes to recreate the system state.

The `Rollback` structural pattern does not provide error or failure detection and may be used in conjunction

with the `Monitoring` and/or `Prediction` structural patterns to be fully functional. The `Rollback` structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the `Rollback` structural pattern covers for the remaining error/failure types.

Performance: The failure-free performance $T_{f=0}$ of the `Rollback` pattern is defined by the task's total execution time without any resilience strategy T_E and the time spent on saving system state and progress to storage T_s during task execution with a total number of checkpoints N. Assuming a constant checkpoint interval τ, the total number of checkpoints N is defined by the task's total execution time without any resilience strategy T_E divided by τ. T_d, time to detect an error/failure, is not part of this pattern.

The performance under failure T is defined by the failure-free performance $T_{f=0}$, plus the total lost time to execute system progress T_{EL} and the total time to load consistent system state and progress from storage and to rollback to the last known correct state T_R (Eq. 41). Assuming constant times T_s, T_l, and T_r, the performance under failure T can be further simplified with a total number of failures (Eq. 42). T can be calculated [52] using a first-order (Eq. 43) and a higher-order (Eq. 44) approximation for an optimal checkpoint interval τ.

$$T = T_E + T_S + T_{EL} + T_R \tag{41}$$

$$T = T_E + NT_s + T_{EL} + \frac{T_E}{M}(T_l + T_r) \tag{42}$$

$$T = T_E + \left(\frac{T_E}{\tau} - 1\right)T_s + \frac{T_E}{M}T_{e,f}(\tau + T_s) + \frac{T_E}{M}(T_l + T_r),$$

$$\tau = \sqrt{2MT_s} \tag{43}$$

$$T = Me^{(T_l+T_r)/M}\left(e^{(\tau+T_s)/M} - 1\right)\frac{T_E}{\tau},$$

$$\tau = \sqrt{2MT_s}\left[1 + \frac{1}{3}\left(\frac{T_s}{2M}\right)^{1/2} + \frac{1}{9}\left(\frac{T_s}{2M}\right)\right] - T_s \tag{44}$$

Reliability: Given that the `Rollback` pattern enables the resumption of correct operation after an error or failure, the reliability of a system employing it is defined by errors and failures that are not handled by the pattern, such as failures of the persistent storage system. The reliability after applying the `Rollback` pattern $R(t)$ can be obtained using the performance under failure T and the failure rate λ_u (or MTTF M_u) of the unprotected part of the system (Eq. 45).

$$R(t) = e^{-\lambda_u T} = e^{-T/M_u} \tag{45}$$

Availability: The availability of `Rollback` pattern can be calculated using the task's total execution time without `Rollback` pattern T_E and the performance with `Rollback` pattern T (Equation 46). T_E is PU and T is PU, SD and UD.

$$A = \frac{t_{pu}}{t_{pu} + t_{ud} + t_{sd}} \tag{46}$$

Examples: Many HPC applications implement application-level rollback by regularly saving intermediate results to stable storage as checkpoints and supporting the capability to load such a checkpoint in addition to some or all of the original data upon restart to recreate the last known correct state. Library-based solutions, such as FTI [27], permit tracking of state that needs to be saved and restored. System-level solutions, such asDMTCP [18], support transparent state saving and restoration using OS support.

Rationale: The `Rollback` structural pattern enables a system to tolerate an error or failure through resumption of correct operation after impact. It relies on the capability to preserve system state before an error or failure, often in a periodic fashion, and restore the previously preserved system state upon such an event to resume operation from a known correct state. The pattern performs proactive actions, such as preserving system state, but mostly relies on reactive actions after an error or failure impacted the system.

Progress in the system is lost after an error or failure, as the recovery is only able to recreate the system state to the time when the snapshot occurred. Error or failure detection is not part of the pattern. The preserved system state is managed on persistent storage, which is not part of the protection domain. The containment and mitigation offered by this pattern are independent from the type of error or failure. The pattern has very little design complexity and has low dependence on a system's architecture, which makes it appealing as a general and portable solution.

Related Patterns: The `Rollback` and `Rollforward` structural patterns are both derived from the `Checkpoint Recovery` architecture pattern. While the `Rollback` structural pattern's error/failure recovery is backward to the time when the snapshot occurred, the `Rollforward` structural pattern's error/failure recovery is forward to the time when the error/failure event occurred.

Known Uses:

- Many HPC applications implement application-level rollback by regularly saving intermediate results to stable storage as checkpoints and supporting the capability to load such a checkpoint in addition to some or all of the original data upon restart to recreate the last known correct state.
- FTI [27] is a library-based checkpoint/restart solution that supports rollback recovery by tracking, saving and restoring data structures in multilevel checkpoint storage.
- The libckpt [148] is a user-level, library-based checkpointing solution for rollback recovery that supports explicit directives to determine the scope of an application's checkpointed state.
- The SCR [130] library uses multilevel checkpointing for rollback recovery; it creates frequent inexpensive checkpoints that can recover the loss of a few nodes to the local node-level storage and writes complete checkpoints that can withstand an entire system failure to the parallel file system.
- The CoCheck checkpoint/restart supports rollback recovery for an MPI implementation; for the coordination of the checkpoints, CoCheck uses a special process [172]. The Condor checkpoint/restart facility is enabled by the user by linking the program source code with the condor library [121].
- BLCR is a process-level checkpoint/restart module for the Linus OS. It has been used in conjunction with LAM/MPI to permit OS-level checkpoint/restart for MPI applications [184].
- Improvements to system-level rollback solutions, such as BLCR, include incremental checkpoint/restart, where only changes between two checkpoints are being saved and the rollback performs a reconstruction of correct state from multiple incremental checkpoints [185].
- DMTCP [18] is a transparent checkpoint/restart solution for multi-threaded and MPI applications.
- The local failure local recovery protocol uses rollback of only some processes with locally stored checkpoints to create correct state [177].
- Message logging protocols have been implemented in OpenMPI to support faster failure recovery using rollback with message replay [35].

7.4.7 Rollforward

Name: Rollforward

Problem: A hardware or software error or subsystem failure due to a physical fault (e.g., wear-out or destruction) or a design fault (e.g., resource underprovisioning) in an HPC environment causes a software, such as an application, to experience an error due to loss of or corrupted state, and potentially a subsequent failure.

Context: The pattern is a derivative of the `Checkpoint Recovery` architectural pattern. It applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The error or failure in the system that the pattern handles must be detected, as the pattern offers no error or failure detection.
- The system is capable of compartmentalizing its state that is accurately representative of the progress of the system since initialization at the time such state is captured.
- The system operation has well-defined intervals that enable the pattern to transition the system to a known correct interval in response to an error or failure.
- The system is capable of saving its current state and loading a previously saved state.

Forces:

- The pattern requires additional storage to capture system state or to log messages or events, which increases overhead in terms of resources required by the system.
- The amount of state captured during each creation of a recovery point incurs storage space and execution time overheads.
- The creation frequency of system state snapshots determines overhead. More frequent snapshot creation increases system execution time, but reduces the amount of lost work upon an error or failure.
- The time interval for the recovery of a system from a snapshot as well as the time interval to create a snapshot must be less than the system's MTBF to guarantee forward progress.

Solution: The `Rollforward` structural pattern enables the resumption of correct operation of a system impacted by an error or failure. It supports resilient operation by restoring the system to a known correct state in the event of an error or failure. Rollback solutions are classified into checkpoint-based and log-based strategies.

The checkpoint-based strategy relies on the creation of system state snapshots and the maintenance of such checkpoints on a persistent storage system that is not affected by the error or failure. Upon detection of an error or failure, the most recent snapshot is used to recreate the last known error/failure-free state of the system. The log-based strategy relies on logging, i.e., storing the information of, events, such as messages sent between different parts of the system or to the system as input, on a persistent storage system that is not affected by the error or failure. Upon detection of an error or failure, the log is replayed to recreate the last known error/failure-free state of the system. In contrast to the checkpoint-based strategy, the log-based strategy is able to offer resilience in the presence of non-deterministic events and for non-deterministic systems, as replaying the event log deterministically recreates system state.

The checkpoint and log-based strategies may be used together, where events are logged in-between snapshots. With both strategies, the operation of the system is resumed when the system state is recovered. Based on a temporal view of the system's progress, the error/failure recovery is forward to the time when the error/failure occurred. The pattern must include a mechanism to facilitate restoring the system state

from the last snapshot up to the point of the error or failure. This mechanism may be based on either:

- Log-based protocols that are based on the piecewise deterministic assumption, in which the system identifies nondeterministic events and the information necessary to replay the event during recovery is captured and logged. The following logging protocols for non-deterministic events may be used:
  - *Pessimistic:* The protocol assumes that an error or failure occurs after a nondeterministic event in the system. Therefore, the determinant of each nondeterministic event is immediately logged to persistent storage. The error/failure-free overhead of this approach is high.
  - *Optimistic:* The determinants are held in a volatile storage and written to persistent storage asynchronously. This protocol makes the optimistic assumption that the logging is completed before the occurrence of an error or failure. The error/failure-free overhead of this approach is low.
  - *Causal:* The protocol provides a balanced approach by avoiding immediate writing to persistent storage (much like the optimistic protocol to reduce error/failure-free overhead), but each subsystem commits output independently (like the pessimistic protocol to prevent creation of orphan subsystems).
- Online recovery protocols, which do not rely on event logging for rollforward of the system. These protocols may rely on inference to recreate state or on self-correction.

Undetected (latent) errors that are either detected later or result in a different detected error or failure later represent a problem, as the most recent or even more snapshots may contain an illegal system state. In this case, the most recent correct snapshot may be used to recreate the last known error/failure-free state of the system, skipping snapshots containing illegal state and going further back in time in terms of when the snapshot was made. The components of this pattern are illustrated in Figure 38.

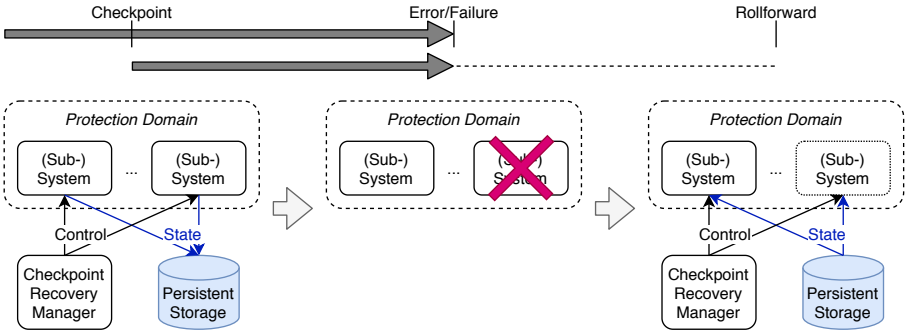
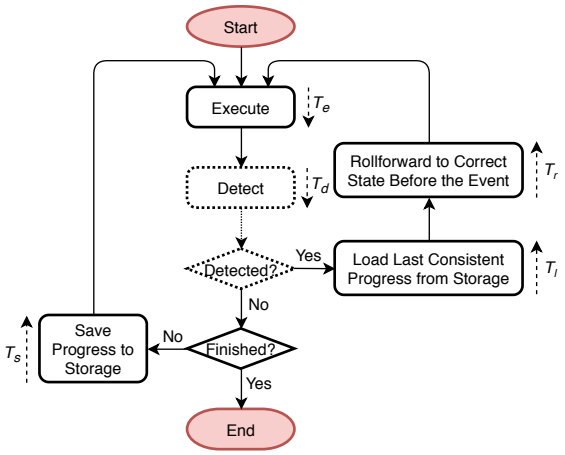


Figure 38. Rollforward pattern components

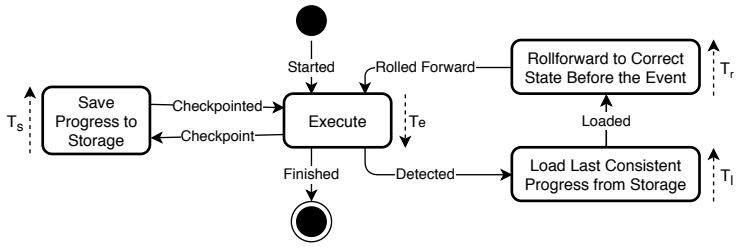
Capability: A system using this pattern is able to continue to operate in the presence of an error or failure with some interruption. This pattern provides mitigation of an error or failure in the system by preserving system state on a persistent storage system before an error or failure, using a checkpoint-based and or log-based strategy, and restoring the previously preserved system state upon such an event and rolling it forward to resume operation from the known correct state right before the event. The flowchart and state diagram of the pattern is shown in Figure 39 and its parameters in Table 19.

Protection Domain: The protection domain extends to the system state before an error or failure that can be recreated using the system state captured by snapshots stored on persistent storage.

Resulting Context: Correct operation is resumed after an error or failure impacted the system. Progress in the system is not lost, since the recovery is able to recreate the system state to the time when the error/failure occurred. The system is interrupted during error-/failure-free operation for creating system state snapshots and maintaining them on a persistent storage system. The system is interrupted upon an



(a) Flowchart



(b) State diagram

Figure 39. Rollforward pattern flowchart and state diagram

Table 19. Rollforward pattern parameters

Parameter	Definition
T_e	Time to execute (sub-) system progress
T_d	Time to detect an error/failure (not part of this pattern, but shown for completeness)
T_l	Time to load consistent (sub-) system state and progress from storage
T_r	Time to rollforward to the correct state before the event
T_s	Time to save (sub-) system state and progress to storage

error or failure for recreating the last known error/failure-free state of the system.

A trade-off exists between the creation frequency of system state snapshots and/or event logging and the corresponding execution time overhead during error-/failure-free operation vs. the amount of work required to recreate the state in the case. The optimal solution of this trade-off depends on the MTBF, the time it takes to save and load a system state snapshot, the time it takes to log events, and the time it takes to recreate the system state.

The `Rollforward` structural pattern does not provide error or failure detection and may be used in conjunction with the `Monitoring` and/or `Prediction` structural patterns to be fully functional. The `Rollforward` structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the `Rollforward` structural pattern covers for the remaining error/failure types.

Performance: The `Rollforward` pattern avoids losing any work as it recovers the system to stable state immediately before the error or failure event. Assuming constant times T_s, T_l, and T_r, the performance T can be calculated by getting rid of lost work T_{EL} in Eq. 47 (Eq. 48).

$$
\begin{aligned}
T &= T_E + \left(\frac{T_E}{\tau} - 1\right) T_s + \frac{T_E}{M} T_{e,f}(\tau + T_s) + \frac{T_E}{M}(T_l + T_r), \\
\tau &= \sqrt{2MT_s}
\end{aligned} \tag{47}
$$

$$
T = T_E + \left(\frac{T_E}{\tau} - 1\right) T_s + \frac{T_E}{M}(T_l + T_r), \tau = \sqrt{2MT_s} \tag{48}
$$

Reliability: Given that the `Rollforward` pattern enables the resumption of correct operation after an error or failure, the reliability of a system employing it is defined by errors and failures that are not handled by the pattern, such as failures of the persistent storage system. The reliability after applying the `Rollforward` pattern $R(t)$ can be obtained using the performance under failure T and the failure rate λ_u (or MTTF M_u) of the unprotected part of the system (Eq. 49).

$$
R(t) = e^{-\lambda_u T} = e^{-T/M_u} \tag{49}
$$

Availability: The availability of `Rollforward` pattern can be calculated using the task's total execution time without `Rollforward` pattern T_E and the performance with `Rollforward` pattern T (Equation 50). T_E is PU and T is PU, SD and UD.

$$
A = \frac{t_{pu}}{t_{pu} + t_{ud} + t_{sd}} \tag{50}
$$

Examples: GVR [47] is a runtime system that provides fault tolerance to applications by versioning distributed arrays for rollforward recovery, while the checkpoint-on-failure protocol [18] for MPI applications leverages the features of a high-quality fault-tolerant MPI implementation. In either case, algorithm-specific knowledge is needed to perform rollforward recovery, Some ABFT solutions [122] can utilize the original or previously saved data as a replacement for lost or erroneous data and rollforward recover their state to the point at which the error/failure event occurred.

Rationale: The `Rollforward` structural pattern enables a system to tolerate an error or failure through resumption of correct operation after impact. It relies on the capability to preserve system state before an error or failure, often in a periodic fashion, and restore the previously preserved system state upon such an event to resume operation from a known correct state. The pattern performs proactive actions, such as preserving system state, but mostly relies on reactive actions after an error or failure impacted the system.

Progress in the system is not lost, since the recovery is able to recreate the system state to the time when the error/failure occurred. Error or failure detection is not part of the pattern. The preserved system state is managed on persistent storage, which is not part of the protection domain. The containment and mitigation offered by this pattern are independent from the type of error or failure. The pattern has some design complexity, as a mechanism to restore the system state from the last snapshot up to the point of the error or failure is needed. The pattern has low dependence on a system's architecture.

Related Patterns: The `Rollback` and `Rollforward` structural patterns are both derived from the `Checkpoint Recovery` architecture pattern. While the `Rollback` structural pattern's error/failure recovery is backward to the time when the snapshot occurred, the `Rollforward` structural pattern's error/failure recovery is forward to the time when the error/failure event occurred.

Known Uses:

- GVR [47] is a runtime system that provides fault tolerance to applications by versioning distributed arrays. It supports rollforward recovery based on application-specified mechanisms. Past versions of lost or erroneous data can be retrieved and used for computing the state at which the error/failure event occurred.
- The checkpoint-on-failure protocol [18] for MPI applications leverages the features of a high-quality fault-tolerant MPI implementation and algorithm-based rollforward recovery. It provides the ability for all healthy processes to continue to operate and perform rollforward recovery using message passing.
- Some ABFT solutions [122] can utilize the original or previously saved data as a replacement for lost or erroneous data and rollforward recover their state to the point at which the error/failure event occurred. This is especially true for any reversible algorithms, where parts of the algorithm would temporarily perform rollback computation to allow the rollforward computation to catch up and maintain correctness.

7.4.8 Forward Error Correction Code

Name: Forward Error Correction Code

Problem: A hardware error or subsystem failure due to a physical fault (e.g., wear-out or destruction) in an HPC environment causes a software, such as a system service or an application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Redundancy` architectural pattern and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system state is represented using a sequence of symbols.

Forces:

- The pattern introduces an execution time and/or resource requirement (storage space, computational capability, etc.) penalty independent of whether an error or failure occurs during system operation or not.
- The scope and strength of the redundancy employed by the pattern determine its execution time and resource requirement overhead.
- The number of errors and failures that are detectable and correctable is limited by the amount of redundant information contained in the forward error correction code.

Solution: The `Forward Error Correction Code` structural pattern enables the continuous correct operation of a system impacted by an error or failure. It supports resilient operation by applying redundancy to system state and optionally to system resources. This redundancy is in the form of encoded system state. The pattern requires very well defined input and output to permit input encoding and output decoding. Input is encoded, processed redundantly in an encoded fashion by the system, and the output is then decoded. The decoding corrects an error or failure. The scope and strength of the redundancy are defined by the encoding/decoding.

Redundancy can be in time, meaning the same system resources process the encoded input in time. Redundancy can also be in space, meaning additional (redundant) system resources are used, such that the different system resources process the encoded input in space. Redundancy in time saves system resources, while redundancy in space offers more error/failure coverage. A mix between redundancy in time and space is possible as well, where there is more encoded system state than additional (redundant) system resources.

Encoding in its simplest form may be just repeating the input for redundancy in time, where the decoding just compares subsequent outputs. More involved encoding/processing/decoding schemes involve k information symbols and r redundant information symbols, where there may be less than, equal to or more than r symbols than k symbols. In the previously mentioned simplest form, the k and r symbols are the same and there may be 1 or more r symbols. The components of this pattern are illustrated in Figure 40.

Capability: A system using this pattern is able to continue to operate in the presence of an error or failure with no interruption. This pattern provides error and/or failure detection in the system by applying redundancy to system state in the form of encoded system state. The pattern provides mitigation of an error or failure in the system by applying redundancy to system state and optionally to system resources, such that the system continues to operate correctly in the presence of such an event. The flowchart and state diagram of the pattern is shown in Figure 41 and its parameters in Table 20.

Protection Domain: The protection domain extends to the encoded system state and to the system resources processing it.

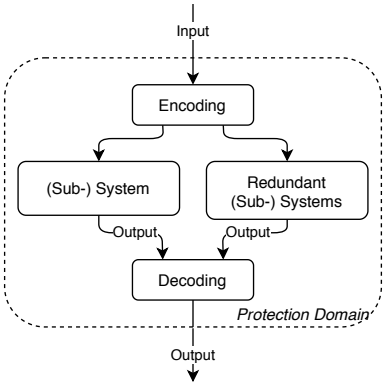


Figure 40. Forward Error Correction Code pattern

Table 20. Forward Error Correction Code pattern parameters

Parameter	Definition
T_a	Time to activate the redundant information storage
T_{en}	Time to encode the input for the (sub-) system
T_{ex}	Time to execute (sub-) system progress
T_d	Time to decode the output from the (sub-) system and detect
T_c	Time to correct using redundant information

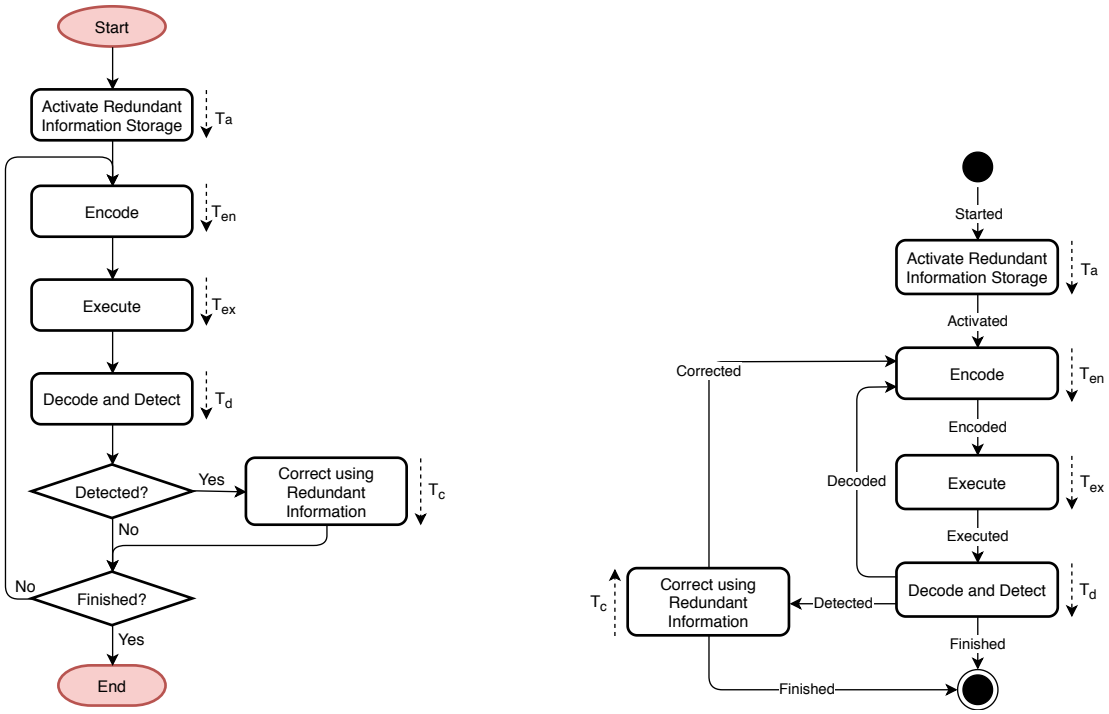
Resulting Context: Correct operation is performed despite an error or failure impacting the system. Progress in the system is not lost due to an error or failure. The system is not interrupted during error-/failure-free operation or when encountering an error or failure. Resource usage in time or space is increased according to the amount of redundancy employed in the form of encoded system state and due to the encoding of input and decoding and correction of output.

A trade-off exists between the amount of redundancy employed and the number of errors and/or failures that can be tolerated at the same time and/or in time. More redundancy tolerates generally more errors and/or failures, but requires either more resources or more execution time.

The `Forward Error Correction Code` structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the `Forward Error Correction Code` structural pattern covers for the remaining error/failure types.

Performance: The failure free performance $T_{f=0}$ of the `Forward Error Correction Code` pattern is defined by the task total execution time without any resilience strategy T_E, the total time to activate the redundant information storage T_a, the time to encode T_{en}, and the time to decode and detect T_d with the total number of input-execute-output cycles P. The performance under failure T is defined by $T_{f=0}$ plus the time T_c to correct using redundant information, where total time to correct using redundant information is number of error or failure times T_c. Assuming constant times T_a, T_{en} (t_{en}), T_d (t_d), and T_c, T can be defined by Eq. 51.

$$T = T_E + T_a + P(t_{en} + t_d) + \frac{T_E}{M}(T_c) \tag{51}$$



(a) Flowchart **(b) State diagram**

Figure 41. Forward Error Correction Code pattern flowchart and state diagram

Reliability: Given that the `Forward Error Correction Code` pattern enables the resumption of correct operation after an error or failure, the reliability of a system employing it is defined by errors and failures that are not handled by the pattern, such as failures of the persistent storage system. The reliability after applying the `Forward Error Correction Code` pattern $R(t)$ can be obtained using the performance under failure T and the failure rate λ_u (or MTTF M_u) of the unprotected part of the system (Eq. 52).

$$R(t) = e^{-\lambda_u T} = e^{-T/M_u} \tag{52}$$

Availability: The availability of `Forward Error Correction Code` pattern can be calculated using the task's total execution time without `Forward Error Correction Code` pattern T_E and the performance with `Forward Error Correction Code` pattern T (Equation 53). T_E is PU and T is PU, SD and UD.

$$A = \frac{t_{pu}}{t_{pu} + t_{ud} + t_{sd}} \tag{53}$$

Examples: There are various schemes that enable forward error correction in memory devices, storage systems as well as communication channels. Based on time and space overhead constraints, schemes of different detection and correction capabilities are used. Popular examples include parity bits, checksums, Hamming codes, hash function codes. More elaborate schemes such as systematic cyclic block codes include binary BCH, Reed-Solomon and CRC. Forward error correction can be found in HPC storage systems with RAID, the InfiniBand interconnect [9], the memory hierarchy [131, 59], ABFT solutions [95] and coded computing [106].

Rationale: The `Forward Error Correction Code` structural pattern enables a system to tolerate an error or failure through continuation of correct operation after impact. It relies on system state redundancy in the form of encoded system state. The pattern performs mostly proactive actions, such as maintaining redundancy. Error or failure detection is part of the pattern in the form of output decoding. The pattern has high design complexity due to the need for encoding input, decoding output, and processing encoded system state.

Related Patterns: The `Forward Error Correction Code`, `Active/Standby`, `N-modular Redundancy`, `N-version Design`, and `Recovery Block` structural patterns are based on similar inclusion of redundancy to compensate for errors or failures. The `Active/Standby` structural pattern offers a lower degree of redundancy using *N* functionally identical replicas. The `N-modular Redundancy` structural pattern offers a higher degree of redundancy than the `Active/Standby` structural pattern, also using *N* functionally identical replicas. It is sometimes referred to as Active/Active. The `N-version Design` structural pattern provides a different type of redundancy using *N* functionally equivalent alternate system implementations in an Active/Active fashion. The `Recovery Block` structural pattern offers a different type of redundancy with a functionally equivalent alternate system implementation encapsulated in a recovery block that is executed after an error or failure of the system.

Known Uses:

- HPC storage systems use forward error correction in the form of RAID.
- InfiniBand, which is among the most widely deployed high-speed interconnect employs forward error correction in the InfiniBand devices, including adapters and switches, to fix bit errors throughout the network [9]. The forward error correction allows reduction in data re-transmission between the end-nodes.
- HPC systems use memory DIMMs that employ ECC, which is based on forward error correction by maintaining redundant bits per memory line to support SECDED for bit flips [131].
- Chipkill [59] uses forward error correction across memory chips atop ECC to perform SECDED at a symbol-level granularity.
- Some ABFT methods use schemes, such as checksums, that perform forward error correction at the application-level [95].
- Coded computing uses algorithmic forward error correction. A recent solution used coded computing for parallel matrix-matrix multiplication [106].

7.4.9 Active/Standby

Name: Active/Standby

Problem: A hardware error or subsystem failure due to a physical fault (e.g., wear-out or destruction) in an HPC environment causes a software, such as a system service or an application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Redundancy` and `Fault Diagnosis` architectural patterns and it applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has a modular design that has a well-defined scope and a set of inputs and outputs.

Forces:

- The pattern introduces a resource requirement (storage space, computational capability, etc.) penalty independent of whether an error or failure occurs during system operation or not.
- The pattern may introduce an execution time penalty when an error or failure occurs during system operation.
- The scope and strength of the redundancy employed by the pattern determine its execution time and resource requirement overhead.

Solution: The `Active/Standby` structural pattern enables the continuous correct operation of a system impacted by an error or failure. It supports resilient operation by applying redundancy to system state and to system resources. This redundancy is in the form of *N* functionally identical replicas, using redundancy in space and potentially in time. The pattern requires very well defined input and output to permit replication. The pattern can operate in the following modes:

- *Active/Hot-Standby:* One active system performs the system's operations, while one or more standby replica systems perform the same operations or obtain an instant and consistent copy of the system state from the active system on change. If the standby systems perform the same operations, input is replicated to and processed by all standby systems. If the standby systems obtain an instant and consistent copy of the system state, the active system processes the input and system state is replicated upon change to all standby systems. This may be performed using a reliable communication protocol, such as a total order broadcast, or a shared stable storage, such as a shared hard disk or a replicated block device. The output is provided only by the active system. The standby systems monitor the active system for any error or failure, such as using a heartbeat. Upon a detected error or failure, a fail-over is performed from the active system to one of the standby systems, making that standby system to become the active system. The fail-over causes only very minimal interruption and no loss of system progress.
- *Active/Warm-Standby:* One active system performs the system's operations, while one or more standby replica systems obtain a consistent copy of the system state from the active system in regular intervals. The active system processes the input and system state is replicated in regular intervals to all standby systems. This may be performed using a reliable communication protocol, such as a total order broadcast, or a shared stable storage, such as a shared hard disk or a replicated block device. The output is provided only by the active system. The standby systems monitor the active system for any error or failure, such as using a heartbeat. Upon a detected error or failure, a fail-over is performed from the active system to one of the standby systems, making that standby system to become the active system. The fail-over causes only very minimal interruption. However, system progress between the last system state replication and the error or failure is lost.

- *Active/Cold-Standby:* One active system performs the system's operations, while one or more standby replica systems are not operating at all. The active system processes the input and provides the output. The active system is monitored for any error or failure bey either an external system, such as using a heartbeat, or by a human. Upon a detected error or failure, a fail-over is performed from the active system to one of the standby systems, making that standby system to become the active system. The fail-over can cause substantial interruption, as it may be performed by a human and not automatically by the external monitoring system. Since there is no state replication, all system progress is lost.

The components of this pattern are illustrated in Figure 42.

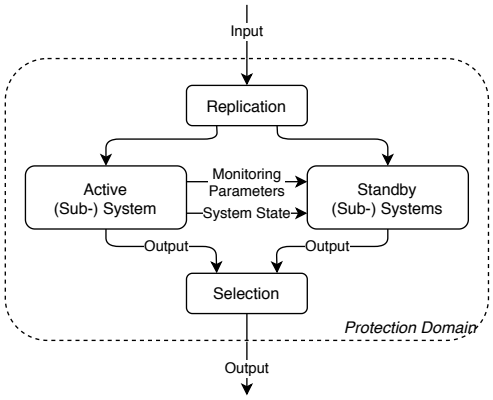


Figure 42. Active/Standby pattern components

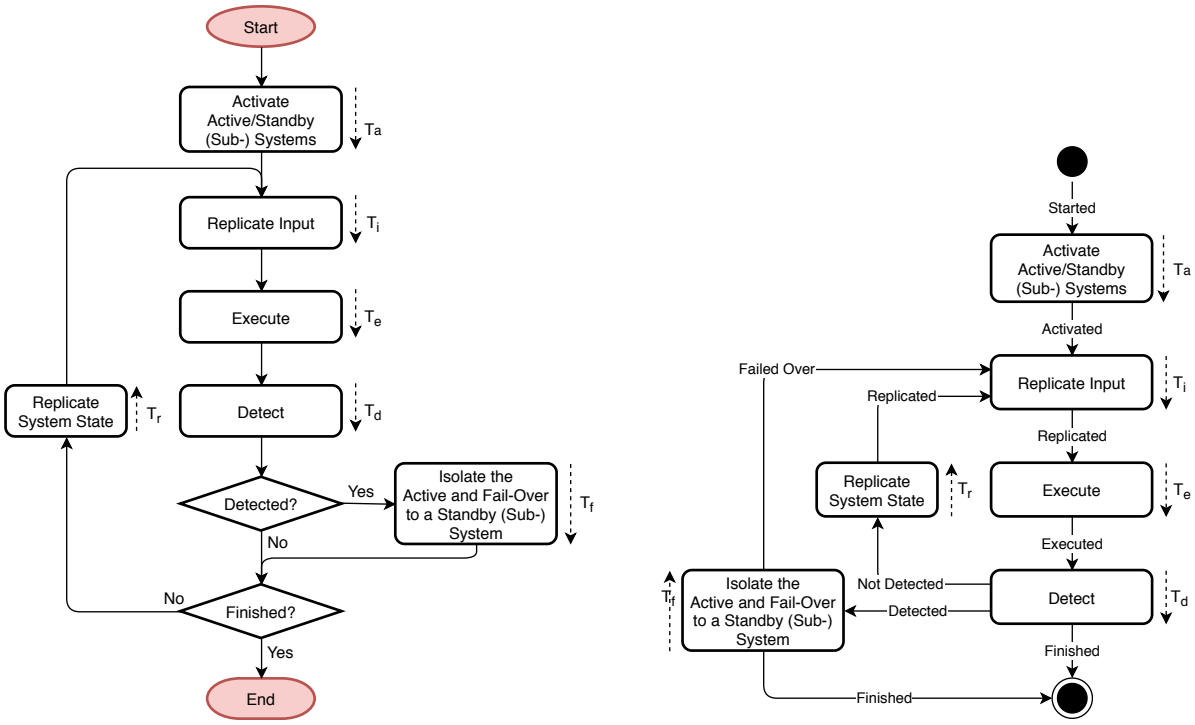
Capability: A system using this pattern is able to continue to operate in the presence of an error or failure with none to significant interruption and loss of progress, depending on the active/standby mode. This pattern provides error and/or failure detection and containment in the system by monitoring the active system. The pattern provides mitigation of an error or failure in the system by applying redundancy to system state and system resources, such that the system continues to operate correctly in the presence of such an event. The flowchart and state diagram of the pattern is shown in Figure 43 and its parameters in Table 21.

Table 21. Active/Standby pattern parameters

Parameter	Definition
T_a	Time to activate the active and standby (sub-) systems
T_i	Time to replicate the input to the active and standby (sub-) systems
T_e	Time to execute progress on the active (sub-) system
T_d	Time to detect an error in or failure of the active (sub-) system
T_f	Time to isolate the active (sub-) system and fail-over to a standby (sub-) system
T_r	Time to replicate system state from the active (sub-) system to the standby (sub-) systems

Protection Domain: The protection domain extends to the system state and the system resources that implement the N functionally identical replica systems.

Resulting Context: Correct operation is performed despite an error or failure impacting the system. Progress in the system may be lost due to an error or failure, depending on the active/standby mode. The system is not interrupted during error-/failure-free operation. It is interrupted when encountering an error or failure. Resource usage in space is increased according to the amount of redundancy employed in the form of N functionally identical replicas and due to the replication of input and or system state.



(a) Flowchart **(b) State diagram**

Figure 43. Active/Standby pattern flowchart and state diagram

A trade-off exists between the amount of redundancy employed and the number of errors and/or failures that can be tolerated at the same time and/or in time. More redundancy tolerates generally more errors and/or failures, but requires either more resources or more execution time.

The `Active/Standby` structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the `Active/Standby` structural pattern covers for the remaining error/failure types.

Performance: The failure-free performance $T_{f=0}$ of the `Active/Standby` pattern is defined by the task total execution time without any resilience strategy T_E, the total time to activate the active and (sub-) standby systems T_a, the time to replicate the input to the active and standby (sub-) systems T_i, the time to detect an error in or failure of the active (sub-) system T_d, and the time to replicate system state from the active (sub-) system to the standby (sub-) systems T_r with the total number of input-execute-output cycles P. The performance under failure T is defined by $T_{f=0}$ plus the time T_f to isolate the active (sub-) system and fail-over to a standby (sub-) system, where total time to isolate is number of error or failure times T_f. Assuming constant times T_a, T_i (t_i), T_d (t_d), T_r (t_r), and T_f, T can be defined by Eq. 54. When the redundancy is in space, using a ratio for replication in space vs. in time α, T (Eq. 55) can be reformulated.

$$T = T_E + T_a + P(t_i + t_d + t_r) + \frac{T_E}{M}\left(T_f\right) \tag{54}$$

$$T = \alpha T_E + (1-\alpha)NT_E + T_a + P(t_i + t_d + t_r) + \frac{T_E}{M}\left(T_f\right) \tag{55}$$

Reliability: Reliability is defined by the parallel reliability of the N-redundant execution and the performance under failure T (Eq. 56). It can be simplified for redundancy of identical systems (Eq. 57).

$$R(t) = 1 - \prod_{n=1}^{N}(1 - e^{-\lambda_n T}) \quad (56) \qquad R_i(t) = 1 - (1 - e^{-\lambda T})^N \quad (57)$$

Availability: The availability A of the `Active/Standby` pattern is defined by N-parallel availability and the performance under failure T (Eq. 58). It can be simplified for redundancy of identical systems (Eq. 59). If T_a, T_i, T_d, T_r, and T_f are small enough, non-identical and identical availability can be simplified further (Eqs. 60 and 61), where M_n (or M) is the MTTF and R_n (or R) is the MTTR of each individual system (T_f).

$$\begin{aligned} A &= 1 - \prod_{n=1}^{N}(1 - A_n) \\ &= 1 - \prod_{n=1}^{N}\left(1 - \frac{T_{E,n}}{T_n}\right) \end{aligned} \quad (58) \qquad \begin{aligned} A_i &= 1 - (1 - A)^N \\ &= 1 - \left(1 - \frac{T_E}{T}\right)^N \end{aligned} \quad (59)$$

$$A = 1 - \prod_{n=1}^{N}\left(1 - \frac{M_n}{M_n + R_n}\right) \quad (60) \qquad A_i = 1 - \left(1 - \frac{M}{M + R}\right)^N \quad (61)$$

Examples: The `Active/Standby` structural pattern is typically used for critical hardware or software systems in HPC environments. For example, power supplies, voltage regulators, the parallel file system MDS in Lustre [194] and the SLURM [192] job and resource manager are often implemented in an active/standby fashion.

Rationale: The `Active/Standby` structural pattern enables a system to tolerate an error or failure through continuation of correct operation after impact. It relies on system state and on system resource redundancy in the form of functionally identical replicas. The pattern performs mostly proactive actions, such as maintaining redundancy. Error or failure detection is part of the pattern in the form of monitoring. The pattern has some design complexity, as input or system state needs to be replicated and the fail-over needs to isolate the previous active system.

Related Patterns: The `Forward Error Correction Code`, `Active/Standby`, `N-modular Redundancy`, `N-version Design`, and `Recovery Block` structural patterns are based on similar inclusion of redundancy to compensate for errors or failures. The `Forward Error Correction Code` structural pattern provides redundancy using encoded system state. The `N-modular Redundancy` structural pattern offers a higher degree of redundancy also using N functionally identical replicas and is sometimes referred to as Active/Active. The `N-version Design` structural pattern provides a different type of redundancy using N functionally equivalent alternate system implementations in an Active/Active fashion. The `Recovery Block` structural pattern offers a different type of redundancy with a functionally equivalent alternate system implementation encapsulated in a recovery block that is executed after an error or failure of the system, somewhat similar to the Active/Hot-Standby mode of the `Active/Standby` structural pattern.

Known Uses:

- Production HPC systems such as the Cray XC40 series [103] include active/standby power supplies and voltage regulator modules to ensure continuous operation in the event that one of these units experiences malfunction or failure.

- Active/standby (cold, warm and hot) using shared storage between active and standby systems is a common technique. The shared storage may be a dual-hosted hard drive, a networked storage using RAID, or a DRBD [154]. Such solutions have been extensively used in HPC environments for critical system services, such as the job and resource manager (e.g., SLURM [192] and SGE [175]) and the parallel file system MDS (e.g., PVFS [146] and Lustre [194]).
- Active/hot-standby using a commit protocol for state replication has been implemented for some HPC job and resource managers as part of high availability cluster solutions, such as HA-OSCAR [119] with its commit protocol for OpenPBS [16].
- High availability clustering is a form of active/hot-standby with n active systems and m standby systems that are set up in an $n + m$ configuration. It targets high throughput processing of a large number of small service requests with no or minimal state changes, where the active systems respond to service requests. High availability clustering may use shared storage, state replication or state separation. In contrast to shared storage and state replication, where systems have the same state, state separation splits the state space among active systems, such as two parallel file system MDSs serving two different file system directories. An implementation of HA-OSCAR supported high availability clustering for two job and resource managers, OpenPBS [16] and SGE [175]). Parallel file system MDSs, such as Lustre [194], support high availability clustering as well.
- Active/standby also plays a role in resilience for parallel applications in HPC environments. Starting a parallel application with additional spare compute nodes enables its reconfiguration and the replacement of a failed compute node without completely restarting the application [21].

7.4.10 N-modular Redundancy

Name: N-modular Redundancy

Problem: A hardware error or subsystem failure due to a physical fault (e.g., wear-out or destruction) in an HPC environment causes a software, such as a system service or an application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Redundancy` architectural pattern and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has a modular design that has a well-defined scope and a set of inputs and outputs.

Forces:

- The pattern introduces an execution time and/or resource requirement (storage space, computational capability, etc.) penalty independent of whether an error or failure occurs during system operation or not.
- The scope and strength of the redundancy employed by the pattern determine its execution time and resource requirement overhead.

Solution: The `N-modular Redundancy` structural pattern enables the continuous correct operation of a system impacted by an error or failure. It supports resilient operation by applying redundancy to system state and optionally to system resources. This redundancy is in the form of *N* functionally identical replicas. The pattern requires very well defined input and output to permit input replication and output comparison. Input is replicated to identical instances of the system, processed by each replica system, and the output is then compared. The comparison corrects an error or failure of a replica system. The scope and strength of the redundancy are defined by the number of functionally identical replicas *N*.

Redundancy can be in time, meaning the same system resources are used for redundancy and execute the *N* functionally identical replicas in time. Redundancy can also be in space, meaning additional (redundant) system resources are used and execute the *N* functionally identical replicas in space. Redundancy in time saves system resources, while redundancy in space offers more error/failure coverage. A mix between redundancy in time and space is possible as well, where there are more functionally identical replicas than additional (redundant) system resources. The components of this pattern are illustrated in Figure 44.

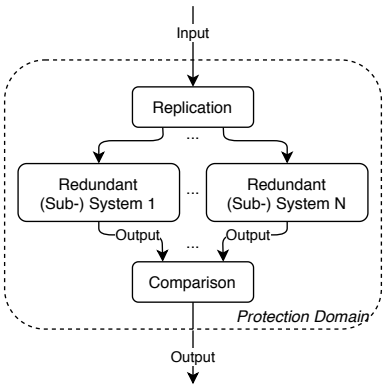
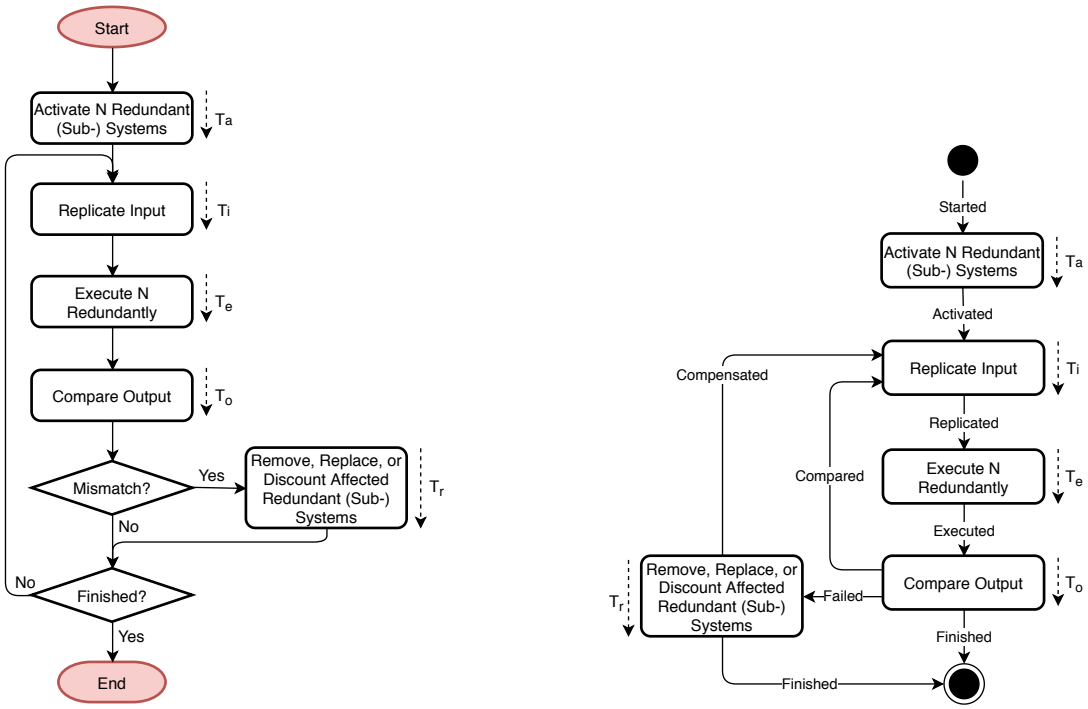


Figure 44. N-modular Redundancy pattern components

Capability: A system using this pattern is able to continue to operate in the presence of an error or failure with no interruption. This pattern provides error and/or failure detection in the system by applying redundancy to system state in the form of N functionally identical replicas. The pattern provides mitigation of an error or failure in the system by applying redundancy to system state and optionally to system resources, such that the system continues to operate correctly in the presence of such an event. The flowchart and state diagram of the pattern is shown in Figure 45 and its parameters in Table 22.



(a) Flowchart **(b) State diagram**

Figure 45. N-modular Redundancy pattern flowchart and state diagram

Table 22. N-modular Redundancy pattern parameters

Parameter	Definition
T_a	Time to activate N replicas of the (sub-) system
T_i	Time to replicate the input to the N replicas of the (sub-) system
T_e	Time to execute (sub-) system progress in the N replicas of the (sub-) system
T_o	Time to compare the outputs from the N replicas of the (sub-) system
T_r	Time to remove, replace, or discount the affected redundant (sub) system replica(s)

Protection Domain: The protection domain extends to the system state and the system resources that implement the N functionally identical replica systems.

Resulting Context: Correct operation is performed despite an error or failure impacting the system. Progress in the system is not lost due to an error or failure. The system is not interrupted during error-/failure-free operation or when encountering an error or failure. Resource usage in time or space is increased according to the amount of redundancy employed in the form of N functionally identical replicas and due to the replication of input and comparison and correction of output.

A trade-off exists between the amount of redundancy employed and the number of errors and/or failures

that can be tolerated at the same time and/or in time. More redundancy tolerates generally more errors and/or failures, but requires either more resources or more execution time.

The `N-modular Redundancy` structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the `N-modular Redundancy` structural pattern covers for the remaining error/failure types.

Performance: The failure-free performance $T_{f=0}$ of the `N-modular Redundancy` pattern is defined by the task's total execution time without any resilience strategy T_E, the total time to activate N replicas of the system T_a, the time to replicate the input T_i and the time to compare the outputs T_o with the total number of input-execute-output cycles P. The performance under failure T is defined by $T_{f=0}$, plus the total time T_r to remove, replace, or discount the replica(s) where total time to remove, replace, or discount is number of error or failure times T_f. Assuming constant times T_a, T_i (t_i), T_o (t_o), and T_r, T can be simplified (Eq. 62). Using a ratio for replication in space vs. in time α, T (Eq. 63) can be reformulated.

$$T = T_E + T_a + P(t_i + t_o) + \frac{T_E}{M}(T_r) \tag{62}$$

$$T = \alpha T_E + (1 - \alpha)NT_E + T_a + P(t_i + t_o) + \frac{T_E}{M}(T_r) \tag{63}$$

Reliability: Reliability is defined by the parallel reliability of the N-redundant execution and the performance under failure T (Eq. 64).

$$R_i(t) = 1 - (1 - e^{-\lambda T})^N \tag{64}$$

Availability: The availability A of N-parallel syaytems is defined by M_n (or M) and R_n (or R) where M is the MTTF and R is the MTTR of each individual system (Eq. 65).

$$A_i = 1 - \left(1 - \frac{M}{M + R}\right)^N \tag{65}$$

Examples: The use of the `N-modular redundancy` structural pattern in various hardware and software systems enables detection and correction of errors, or the compensation of failures. Dual-modular redundancy for error detection and failure compensation and triple-modular redundancy for error detection and correction and failure compensation are used forms of this pattern in HPC environments. Examples include dual-redundant cooling fans, dual- and triple–modular redundant MPI implementations [78], dual-redundant parallel file system MDS solutions [92] and dual-redundant mission-critical HPC systems (e.g., weather forecast).

Rationale: The `N-modular redundancy` structural pattern enables a system to tolerate an error or failure through continuation of correct operation after impact. It relies on system state redundancy in the form of functionally identical replicas. The pattern performs mostly proactive actions, such as maintaining redundancy. Error or failure detection is part of the pattern in the form of output comparison. The pattern has some design complexity, as input needs to be replicated and output needs to be compared.

Related Patterns: The `Forward Error Correction Code`, `Active/Standby`, `N-modular Redundancy`, `N-version Design`, and `Recovery Block` structural patterns are based on similar inclusion of redundancy to compensate for errors or failures. The `Forward Error Correction Code` structural pattern provides redundancy using encoded system state. The `Active/Standby` structural pattern offers a lower degree of redundancy also using *N* functionally identical replicas, where the `N-modular Redundancy` structural pattern is sometimes referred to as Active/Active. The `N-version Design` structural pattern provides a different type of redundancy using *N* functionally equivalent alternate system implementations. The `Recovery Block` structural pattern offers a different type of redundancy with a functionally equivalent alternate system implementation encapsulated in a recovery block that is executed after an error or failure of the system.

Known Uses:

- Production HPC systems such as the Cray XC40 series [103] include n-modular redundant cooling fans to ensure continuous operation in the event that one of these units experiences malfunction or failure.
- Some implementations of the MPI standard use n-modular redundancy for detection and correction of errors by replicating the MPI messages, or even by replicating MPI processes. The MR-MPI [70], rMPI [76], and RedMPI [78] prototypes are known examples for this n-modular redundancy approach. N-modular redundancy for compute nodes in a system have been evaluated and shown to improve the overall availability of a HPC system [71].
- Some n-modular redundancy implementations for service nodes in HPC systems exist as well, such as for the MDS of PVFS [92] and for HPC job and resource managers that are compliant with the PBS [178].
- N-modular redundancy at the whole system level is often implemented for mission critical HPC systems, such as for weather forecast. In this case, two completely redundant HPC systems perform exactly the same computation.

7.4.11 N-version Design

Name: N-version Design

Problem: A hardware or software error or subsystem failure due to a design fault (e.g., human mistake or defective design tool) in an HPC environment causes a software, such as a system service or an application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Design Diversity` architectural pattern and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has a well-defined specification for which multiple implementation variants may be created.
- There is an implicit assumption of independence between multiple variants of the implementation.

Forces:

- The pattern introduces an execution time and/or resource requirement (storage space, computational capability, etc.) penalty independent of whether an error or failure occurs during system operation or not.
- The scope and strength of the diversity employed by the pattern determine its execution time and resource requirement overhead.
- The pattern requires distinct implementations of the same design specification, which may need to be created by different individuals.
- The pattern increases design complexity due to the need of additional design and verification effort required to create multiple implementations.
- The pattern may introduce a performance penalty during error/failure-free operation due to disparity in the implementation variants.

Solution: The `N-version Design` structural pattern enables the continuous correct operation of a system impacted by an error or failure. It supports resilient operation by applying redundancy to system state and optionally to system resources. This redundancy is in the form of *N* functionally equivalent alternate system implementations. This pattern designs different implementations of the system that are functionally equivalent to enable error and failure resilience through design diversity. Different implementations of the system are less likely to experience the same error or failure.

The pattern requires very well defined input and output to permit input replication and output validation. Input is replicated to functionally equivalent alternate system implementations, processed by each implementation of the system, and the output is then validated. The validation corrects an error or failure of a system implementation. The scope and strength of the redundancy are defined by the number of functionally equivalent alternate system implementations *N* and by their implementation design diversity.

Redundancy can be in time, meaning the same system resources execute the *N* equivalent alternate system implementations in time. Redundancy can also be in space, meaning additional (redundant/diverse) system resources execute the *N* equivalent alternate system implementations in space. Redundancy in time saves system resources, while redundancy in space offers more error/failure coverage. A mix between redundancy in time and space is possible as well, where there are more equivalent alternate system implementations than additional (redundant/diverse) system resources. The components of this pattern are illustrated in Figure 46.

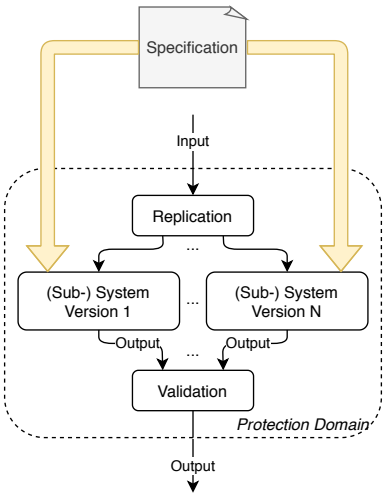


Figure 46. N-version Design pattern components

Capability: A system using this pattern is able to continue to operate in the presence of an error or failure with no or minimal interruption. This pattern provides error and/or failure detection in the system by applying redundancy to system state in the form of N functionally equivalent alternate system implementations. The pattern provides mitigation of an error or failure in the system by applying redundancy to system state and optionally to system resources, such that the system continues to operate correctly in the presence of such an event. The flowchart and state diagram of the pattern is shown in Figure 47 and its parameters in Table 23.

Table 23. N-version Design pattern parameters

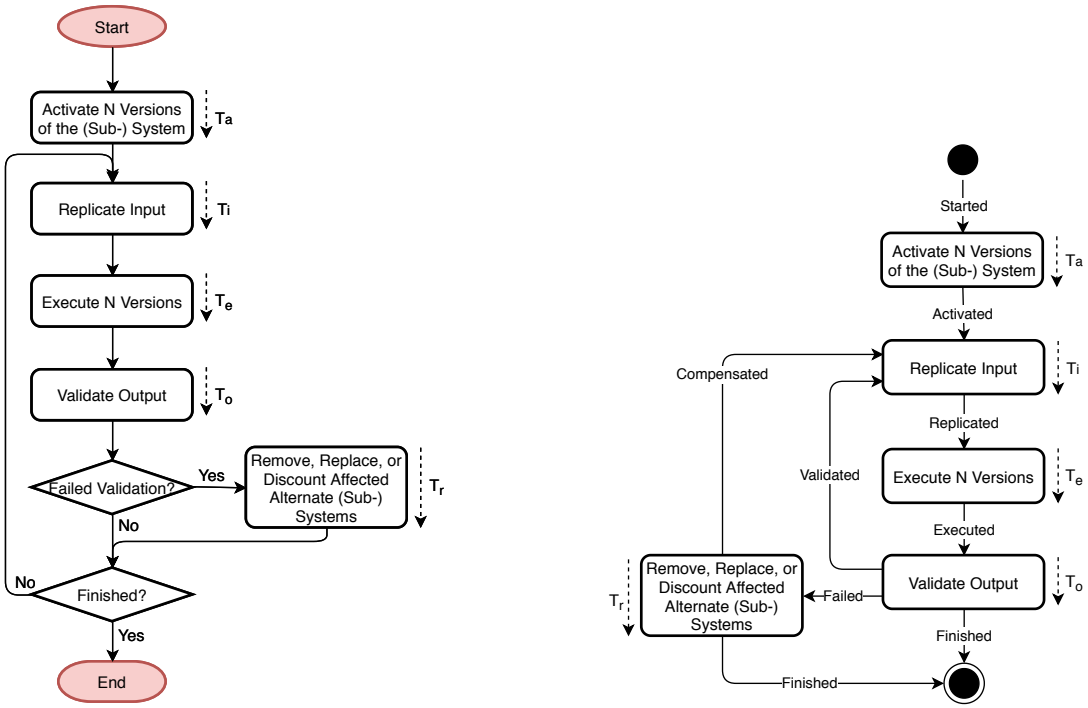
Parameter	Definition
T_a	Time to activate N versions of the (sub-) system
T_i	Time to replicate the input to the N versions of the (sub-) system
T_e	Time to execute (sub-) system progress in the N versions of the (sub-) system
T_o	Time to validate the output from the N versions of the (sub-) system
T_r	Time to remove, replace, or discount the affected redundant (sub) system version(s)

Protection Domain: The protection domain extends to the system state and the system resources described by the design specification that implement the N functionally equivalent alternate systems.

Resulting Context: Correct operation is performed despite an error or failure impacting the system. Progress in the system is not lost due to an error or failure. The system is not interrupted during error-/failure-free operation or when encountering an error or failure. Resource usage in time or space is increased according to the amount of redundancy employed in the form of N functionally equivalent alternate system implementations and due to the difference in resource usage and execution time of the N functionally equivalent alternate system implementations.

A trade-off exists between the amount of redundancy employed and the number of errors and/or failures that can be tolerated at the same time and/or in time. More redundancy tolerates generally more errors and/or failures, but requires either more resources or more execution time.

The `N-version Design` structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are



(a) Flowchart **(b) State diagram**

Figure 47. N-version Design pattern flowchart and state diagram

covered by the other structural pattern(s) and the `N-version Design` structural pattern covers for the remaining error/failure types.

Performance: The failure-free performance $T_{f=0}$ of the `N-version Design` pattern is defined by the task total execution time without any resilience strategy T_E (the worst case execution time of N versions of the (sub-) system), the total time to activate N versions of the (sub-) system T_a, the time to replicate the input to the N versions of the (sub-) system T_i, and the time to validate the output from the N versions of the (sub-) system T_o with the total number of input-execute-output cycles P. The performance under failure T is defined by $T_{f=0}$ plus the time T_r to remove, replace, or discount the affected redundant (sub-) system version(s), where total time to remove, replace, or discount is number of error or failure times T_r. Assuming constant times T_a, T_i (t_i), T_o (t_o), and T_r, T can be defined by Eq. 66. When the redundancy is in space, using a ratio for replication in space vs. in time α, T (Eq. 67) can be reformulated.

$$T = T_E + T_a + P(t_i + t_o) + \frac{T_E}{M}(T_r) \tag{66}$$

$$T = \alpha T_E + (1 - \alpha)NT_E + T_a + P(t_i + t_o) + \frac{T_E}{M}(T_r) \tag{67}$$

Reliability: Reliability is defined by the parallel reliability of the N-redundant execution and the performance under failure T (Eq. 68).

$$R_i(t) = 1 - (1 - e^{-\lambda T})^N \tag{68}$$

Availability: The availability A of N-parallel syaytems is defined by M_n (or M) and R_n (or R) where M is the MTTF and R is the MTTR of each individual system (Eq. 69).

$$A_i = 1 - \left(1 - \frac{M}{M+R}\right)^N \tag{69}$$

Examples: In HPC environments, various versions of the same software are used for the detection of implementation errors. This applies to completely different implementations of the MPI standard and to numerical libraries as well as to different versions of the same implementation. Regression and comparison tests are performed to identify incorrect behavior, missing features and performance problems.

Rationale: The `N-version Design` structural pattern enables a system to tolerate an error or failure through continuation of correct operation after impact. It relies on system state redundancy in the form of functionally equivalent alternate system implementations. The pattern performs mostly proactive actions, such as maintaining redundancy. Error or failure detection is part of the pattern in the form of output validation. The pattern has high design complexity due to the need for functionally equivalent alternate system implementations.

Related Patterns: The `Forward Error Correction Code`, `Active/Standby`, `N-modular Redundancy`, `N-version Design`, and `Recovery Block` structural patterns are based on similar inclusion of redundancy to compensate for errors or failures. The `Forward Error Correction Code` structural pattern provides redundancy using encoded system state. The `Active/Standby` structural pattern offers a lower degree of redundancy using N functionally identical replicas. The `N-modular Redundancy` structural pattern offers a higher degree of redundancy than the `Active/Standby` structural pattern, also using N functionally identical replicas. It is sometimes referred to as Active/Active. The `Recovery Block` structural pattern offers a similar type of redundancy with a functionally equivalent alternate system implementation encapsulated in a recovery block that is executed after an error or failure of the system.

Known Uses:

- HPC centers often provide various MPI library implementations, such as the MVAPICH2 [3], OpenMPI [5], MPICH2 [2], and Intel MPI [1], all of which are based on the MPI standard. Running large-scale applications with these separate implementations of MPI potentially exposes implementation errors in the MPI libraries. Similarly, different versions of numerical libraries are often provided as well. Regression and comparison tests are performed to identify incorrect behavior, missing features and performance problems.
- The DIVA processor architecture [23] includes an out-of-order core as well as a simple in-order pipelined core. The in-order pipeline is functionally equivalent to the primary processor core (it implements the same instruction set architecture) and is used to detect errors in the design of the out-of-order processor core.

7.4.12 Recovery Block

Name: Recovery Block

Problem: A hardware or software error or subsystem failure due to a design fault (e.g., human mistake or defective design tool) in an HPC environment causes a software, such as a system service or an application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Design Diversity` architectural pattern and applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has a well-defined specification for which multiple implementation variants may be created.
- There is an implicit assumption of independence between multiple variants of the implementation.

Forces:

- The pattern introduces an execution time and/or resource requirement (storage space, computational capability, etc.) penalty independent of whether an error or failure occurs during system operation or not.
- The scope and strength of the diversity employed by the pattern determine its execution time and resource requirement overhead.
- The pattern requires distinct implementations of the same design specification, which may need to be created by different individuals.
- The pattern increases design complexity due to the need of additional design and verification effort required to create multiple implementations.
- The pattern introduces a performance penalty upon an error or failure, as the recovery block processes the input and validates its output after error/failure discovery.

Solution: The `Recovery Block` structural pattern enables the continuous correct operation of a system impacted by an error or failure. It supports resilient operation by applying redundancy to system state and optionally to system resources. This redundancy is in the form of a functionally equivalent alternate system implementation encapsulated in a recovery block. This pattern designs a different implementation of the system that is functionally equivalent to enable error and failure resilience through design diversity. The two different implementations of the system are less likely to experience the same error or failure.

The pattern requires very well defined input and output to permit input replication and output validation. Input is replicated to the functionally equivalent alternate system implementations. The original system processes the input and validates its output. Upon discovery of an error or failure, the recovery block implementation processes the input, validates its output, and corrects the output of the original system. The scope and strength of the redundancy is defined by the implementation design diversity.

Redundancy is in time, as the recovery block processes the input and validates its output after error/failure discovery. It may be additionally in space, if the recovery block is executed on different resources than the system. The components of this pattern are illustrated in Figure 48.

Capability: A system using this pattern is able to continue to operate in the presence of an error or failure with some interruption for the execution of the recovery block. This pattern provides error and/or failure detection in the system by output validation. The pattern provides mitigation of an error or failure in the system by applying redundancy to system state and optionally to system resources, such that the system continues to operate correctly in the presence of such an event. The flowchart and state diagram of the pattern is shown in Figure 49 and its parameters in Table 24.

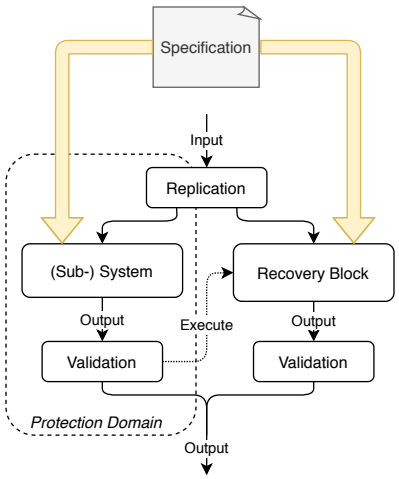


Figure 48. Recovery Block pattern components

Table 24. Recovery Block pattern parameters

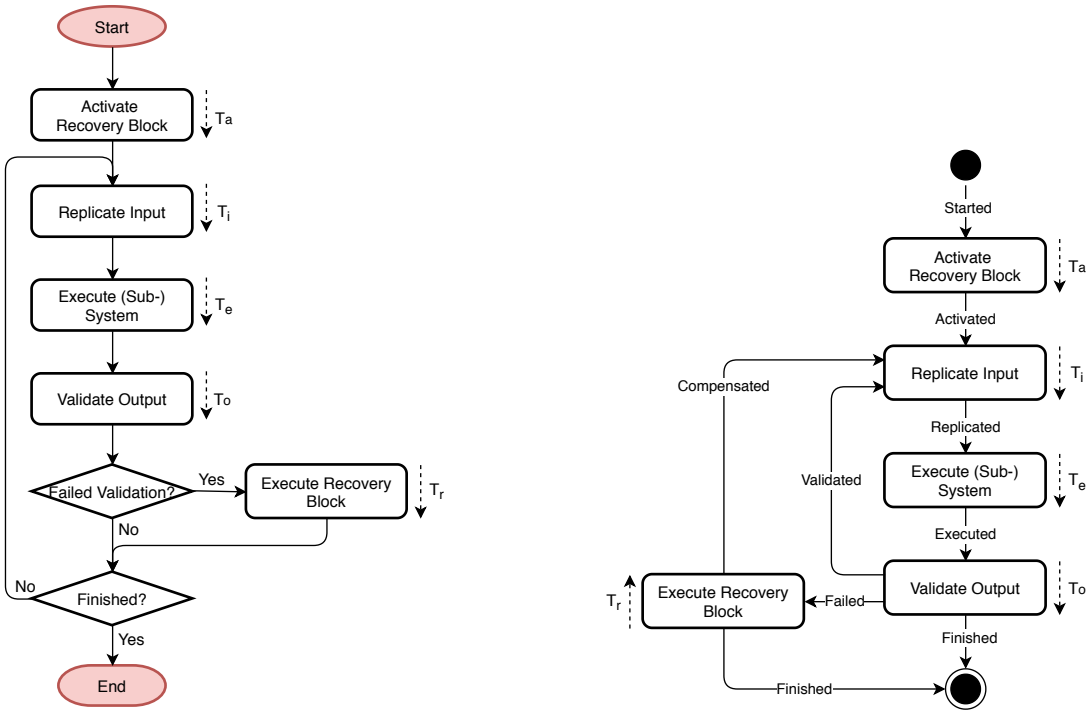
Parameter	Definition
T_a	Time to activate the recovery block of the (sub-) system
T_i	Time to replicate the input to the (sub-) system and the recovery block of the (sub-) system
T_e	Time to execute (sub-) system progress
T_o	Time to validate the output from the (sub-) system
T_r	Time to execute the recovery block of the (sub-) system

Protection Domain: The protection domain extends to the system state and the system resources described by the design specification that implement the recovery block.

Resulting Context: Correct operation is performed despite an error or failure impacting the system. Progress in the system is not lost due to an error or failure. The system is not interrupted during error-/failure-free operation. It is interrupted when encountering an error or failure for the execution of the recovery block. Resource usage in time or space is increased according to the additional resource usage and execution time of the recovery block that employs the redundancy in the form of the functionally equivalent alternate system implementation.

The `Recovery block` structural pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the `Recovery block` structural pattern covers for the remaining error/failure types.

Performance: The failure-free performance $T_{f=0}$ of the `Recovery block` pattern is defined by the task total execution time without any resilience strategy T_E, the total time to activate the recovery block of the (sub-) system T_a, the time to replicate the input to the (sub-) system and the recovery block of the (sub-) system T_i, and the time to validate the output from the (sub-) system T_o with the total number of input-execute-output cycles P. The performance under failure T is defined by $T_{f=0}$ plus the time T_r to execute the recovery block of the (sub-) system, where total time to execute the recovery block of the (sub-) system is number of error or failure times T_r. Assuming constant times T_a, T_i (t_i), T_o (t_o), and T_r, T can be defined by Eq. 70. When the redundancy is in space, using a ratio for replication in space vs. in time α, T (Eq. 71) can be reformulated.



(a) Flowchart **(b) State diagram**

Figure 49. Recovery Block pattern flowchart and state diagram

$$T = T_E + T_a + P(t_i + t_o) + \frac{T_E}{M}(T_r) \tag{70}$$

$$T = \alpha T_E + (1-\alpha)NT_E + T_a + P(t_i + t_o) + \frac{T_E}{M}(T_r) \tag{71}$$

Reliability: Reliability is defined by the parallel reliability of the N-redundant execution and the performance under failure T (Eq. 72).

$$R_i(t) = 1 - (1 - e^{-\lambda T})^N \tag{72}$$

Availability: The availability A of N-parallel syaytems is defined by M_n (or M) and R_n (or R) where M is the MTTF and R is the MTTR of each individual system (Eq. 73).

$$A_i = 1 - \left(1 - \frac{M}{M+R}\right)^N \tag{73}$$

Examples: Containment Domains [48] and the SWIFT library [97] provide language-based approaches for recovery blocks. Applications also often contain verification routines that check for the validity of a computation and correct any detected errors using application-specific knowledge.

Rationale: The `Recovery Block` structural pattern enables a system to tolerate an error or failure through continuation of correct operation after impact. It relies on system state redundancy in the form of a functionally equivalent alternate system implementation encapsulated in a recovery block. The pattern performs some proactive actions, such as maintaining redundancy, but mainly relies on reactive actions, such as the execution of a recovery block after an error or failure was detected. Error or failure detection is part of the pattern in the form of output validation. The pattern has high design complexity due to the need for a functionally equivalent alternate system implementation encapsulated in a recovery block.

Related Patterns: The `Forward Error Correction Code`, `Active/Standby`, `N-modular Redundancy`, `N-version Design`, and `Recovery Block` structural patterns are based on similar inclusion of redundancy to compensate for errors or failures. The `Forward Error Correction Code` structural pattern provides redundancy using encoded system state. The `Active/Standby` structural pattern offers redundancy using *N* functionally identical replicas. The `N-modular Redundancy` structural pattern offers a higher degree of redundancy than the `Active/Standby` structural pattern, also using *N* functionally identical replicas. It is sometimes referred to as Active/Active. The `N-version Design` structural pattern provides a similar type of redundancy using *N* functionally equivalent alternate system implementations.

Known Uses:

- Containment Domains [48] provide a recover routine that is initiated upon detection of an error in the execution of the block of code encapsulated by the domain. This enables the containment domain to constrain the detection and correction of errors to the boundary of the domain.
- The SWIFT library [97] provides language-based implementation of the recovery block for use in C language programs.
- Applications often contain verification routines that check for the validity of a computation. Application-specific knowledge is used in a recovery block to correct any detected errors.

7.4.13 Natural Tolerance

Name: Natural Tolerance

Problem: A hardware or software error or subsystem failure in an HPC environment causes a software, such as a numerical library or application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Self-Masking` architectural pattern and is often used in conjunction with the `Redundancy` architectural pattern. It applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.

Forces:

- System performance in the presence of errors and failures is determined by the efficiency in reaching a correct system state from an illegal system state, which may differ for different illegal system states.
- Self-masking through redundancy may reduce error/failure-free performance.
- A correct state may not be reached from all illegal system states, which limits the scope of this pattern.

Solution: The `Natural Tolerance` pattern relies on the capability of reaching a correct system state from an illegal system state after a finite number of execution steps using implicit error/failure detection and self-masking. Forward progress is guaranteed by design as a correct system state can be reached in a finite number of steps. Self-masking may be as simple as approximation of a correct state. The correct state reached after self-masking is not necessarily the same state that would have been reached without an error or failure. The `Redundancy` pattern is often employed to aid in the process of self-masking and to extend the pattern's protection domain. Self-masking is performed passively, i.e., without special treatment of the error or failure. However, self-masking may still impact performance, as the correct state reached after an error or failure may be different from an error/failure-free execution. The scope of this pattern extends to the system state space that is either correct or incorrect but self-maskable to a correct state. The components of the pattern are illustrated in Figure 50.

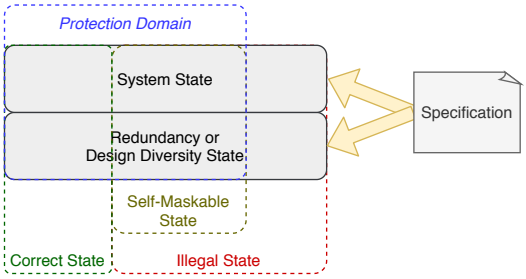
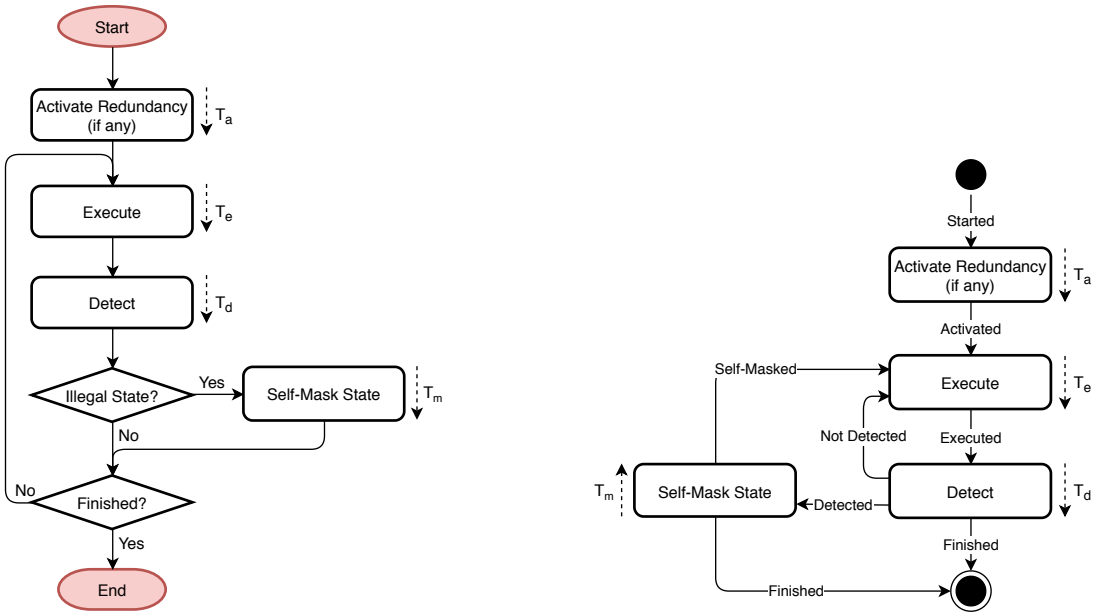


Figure 50. Natural Tolerance pattern components

Capability: A system using this pattern becomes naturally tolerant to an error or failure. The pattern supports the handling of such events with detection, containment, and mitigation using self-masking, i.e., dynamic adaptation through masking. Error/failure detection is implicit by providing the capability to self-mask an illegal system state. Self-masking is an architectural feature inherited from the `Self-Masking` architectural pattern and optionally aided by the application of the `Redundancy` architectural pattern. The flowchart and state diagram of the pattern is shown in Figure 51 and its parameters in Table 25.



(a) Flowchart **(b) State diagram**

Figure 51. Natural Tolerance pattern flowchart and state diagram

Table 25. Natural Tolerance pattern parameters

Parameter	Definition
T_a	Time to activate redundancy (if any)
T_e	Time to execute system progress
T_d	Time to detect illegal system state
T_m	Time to self-mask illegal system state

Protection Domain: The protection domain of this pattern extends to the system state space that is either correct or incorrect but self-maskable to a correct state.

Resulting Context: An error or failure is tolerated through natural tolerance. The `Natural Tolerance` pattern requires the capability of reaching a correct system state from an illegal system state by optionally employing the `Redundancy` architectural pattern, which may reduce error/failure-free performance. The efficiency of self-masking determines performance in the presence of errors and failures. A high performing solution reduces the impact on error/failure-free performance, while at the same time providing efficient self-masking.

A trade-off exists between both, performance in the presence of errors and failures and error/failure-free performance, especially in systems with high error/failure rates. Self-masking is a passive approach, where the system is designed with self-maskable illegal system state. Self-masking becomes a natural property of the system. The implicit detection may reduce error/failure-free performance. The pattern's protection domain does not cover illegal system state that is not self-maskable.

The `Natural Tolerance` pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the `Natural Tolerance` pattern covers for the remaining error/failure types.

Performance: The failure-free performance $T_{f=0}$ of the `Natural Tolerance` pattern is defined by the task total execution time without any resilience strategy T_E, the total time to activate redundancy T_a, and the time to detect illegal system state T_d with the total number of input-execute-output cycles P. The performance under failure T is defined by $T_{f=0}$ plus the time T_m to self-mask illegal system state, where total time to self-mask illegal system state is number of error or failure times T_m. Assuming constant times T_a, T_d (t_d) and T_m, T can be defined by Eq. 74. When the redundancy is in space, using a ratio for replication in space vs. in time α, T (Eq. 75) can be reformulated.

$$T = T_E + T_a + P(t_d) + \frac{T_E}{M}(T_m) \tag{74}$$

$$T = \alpha T_E + (1-\alpha)NT_E + T_a + P(t_d) + \frac{T_E}{M}(T_m) \tag{75}$$

Reliability: Reliability is defined by the parallel reliability of the N-redundant execution and the performance under failure T (Eq. 76).

$$R_i(t) = 1 - (1 - e^{-\lambda T})^N \tag{76}$$

Availability: The availability A of N-parallel syaytems is defined by M_n (or M) and R_n (or R) where M is the MTTF and R is the MTTR of each individual system (Eq. 77).

$$A_i = 1 - \left(1 - \frac{M}{M+R}\right)^N \tag{77}$$

Examples:

- Gossip-based algorithms have the natural ability to create data and algorithmic redundancy through communication, such that errors and failures are tolerated using self-masking and redundancy patterns. Data is communicated to multiple parts of the system participating in the Gossip-based algorithm, such that it is received redundantly and processed using self-masking. Errors or failures occurring before data has been communicated to multiple parts of the system may not necessarily be tolerated as the self-masking may be defeated. Too many errors or failures within a short period of time may also may not necessarily be tolerated for the same reason. The efficiency and performance of self-masking in the presence of errors and failures and the error/failure-free performance depend on the communication pattern (e.g., reach and frequency).
- Iterative solvers or solvers using sampling have the natural ability to create data and algorithmic redundancy through either an iterative process that progresses toward a correct solution in the presence of errors or a probabilistic sampling process that generates enough correct samples over incorrect samples to calculate a correct solution. In both cases, the employed error tolerance though convergence is highly dependent on the problem being solved and the amount of data and algorithmic redundancy. High error rates or high error magnitudes (in terms of impact) may not necessarily be tolerated. The efficiency and performance of self-masking in the presence of errors and failures and the error/failure-free performance depend on the algorithmic capabilities of the iterative process (e.g., step size) or the probabilistic sampling process (e.g., oversampling).

Rationale: The `Natural Tolerance` pattern enables a system to tolerate errors or subsystem errors or failures through adaptation. It relies on the capability of reaching a correct system state from an illegal system state in a finite number of execution steps. It uses implicit error/failure detection and self-masking to passively transition from an illegal to a correct system state. Self-masking is aided by redundancy and intrinsic to the design of the system when this pattern is applied. The pattern has high design complexity and has low dependence on a system's architecture.

Related Patterns: The `Self-Healing` structural pattern uses the same `Self-Stabilization` strategy pattern, but with a different architectural pattern (`Self-Correction`). It also optionally employs the `Redundancy` architectural pattern, but for self-correction.

The `Self-Aware` structural pattern uses the same `Self-Stabilization` strategy pattern, but with a different architectural pattern (`Self-Correction`). It additionally employs the `Fault Diagnosis` architectural pattern and an OODA loop control for error/failure detection and self-correction.

Patterns using the `Self-Stabilization` strategy pattern are a unique class, as self-masking or self-correction may result the system to be in a different correct state after an error or failure than in an error/failure-free execution.

Known Uses:

- Gossip-based algorithms create data and algorithmic redundancy through communication, such that errors and failures are tolerated using the `Natural Tolerance` pattern in conjunction with the `Redundancy` pattern. This includes gossip-based aggregation and reduction algorithms [43, 140, 83], orthogonalization methods [84, 83], eigensolvers [174], and least squares solvers [151].
- Fixed point methods that converge globally when certain conditions are satisfied are able to tolerate certain errors using the `Natural Tolerance` pattern with the `Redundancy` pattern [20, 19], such as a Jacobi iterative scheme that will converge for any initial guess if the matrix is diagonally dominant. Convergence in the presence of errors is aided by data and algorithmic redundancy.
- Certain domain decomposition methods are tolerant to errors using the `Natural Tolerance` pattern in conjunction with the `Redundancy` pattern as they converge in the presence of errors, such as the classical overlapping Schwarz algorithm [86], asynchronous domain decomposition methods [85] and task-based domain decomposition approaches that use sampling and a regression-based solution update [157, 156, 132]. Data and algorithmic redundancy are employed.

7.4.14 Self-Healing

Name: Self-Healing

Problem: A hardware or software error or subsystem failure in an HPC environment causes a software, such as a numerical library or application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Self-Correction` architectural pattern and is often used in conjunction with the `Redundancy` architectural pattern. It applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.

Forces:

- System performance in the presence of errors and failures is determined by the efficiency in reaching a correct system state from an illegal system state, which may differ for different illegal system states.
- Self-correction through redundancy may reduce error/failure-free performance.
- A correct state may not be reached from all illegal system states, which limits the scope of this pattern.

Solution: The `Self-Healing` pattern relies on the capability of reaching a correct system state from an illegal system state after a finite number of execution steps using explicit error/failure detection and self-correction. Forward progress is guaranteed by design as a correct system state can be reached in a finite number of steps. Self-correction may be as simple as discarding, recomputing, or estimating a wrong value in the system or a wrong or missing output from a subsystem. The correct state reached after self-correction is not necessarily the same state that would have been reached without an error or failure. The `Redundancy` architectural pattern is often employed to aid in the process of self-correction and to extend the pattern's protection domain. Self-correction is performed actively, i.e., through special treatment of the error or failure. The scope of this pattern extends to the system state space that is either correct or incorrect but self-correctable to a correct state. The components of the pattern are illustrated in Figure 52.

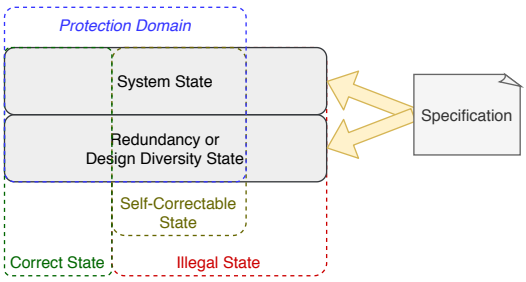
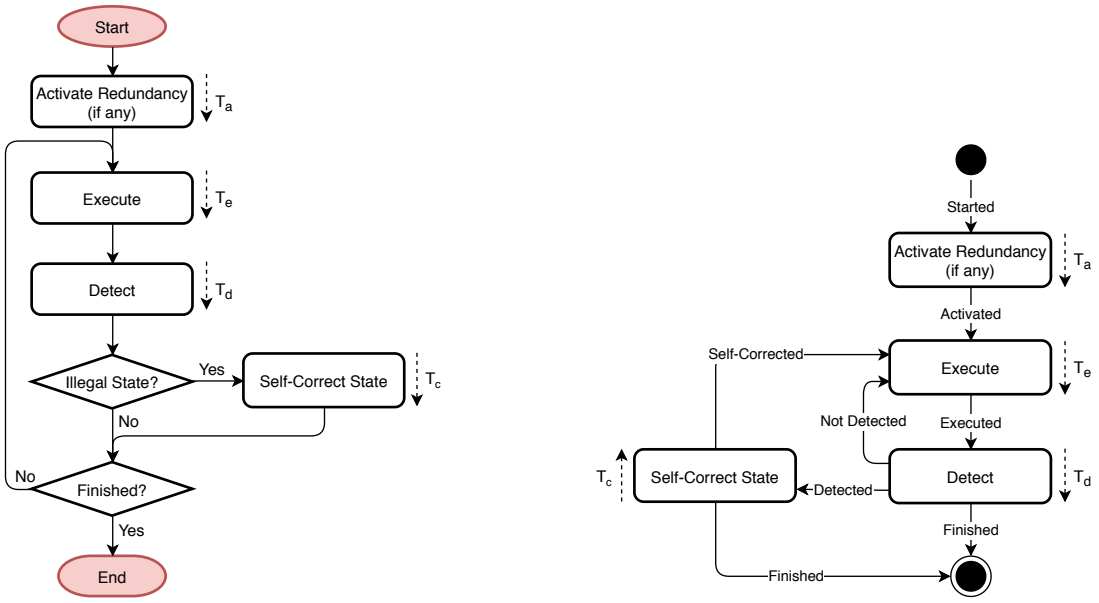


Figure 52. Self-Healing pattern components

Capability: A system using this pattern is able to self-heal an error or failure. This pattern supports the handling of such events with detection, containment, and mitigation using self-correction, i.e., dynamic adaptation through correction. Error/failure detection is explicit by providing the capability to self-correct an illegal system state. Self-correction is an architectural feature inherited from the `Self-Correction` architectural pattern and optionally aided by the application of the `Redundancy` architectural pattern. The flowchart and state diagram of the pattern is shown in Figure 53 and its parameters in Table 26.

Protection Domain: The protection domain of this pattern extends to the system state space that is either correct or incorrect but self-correctable to a correct state.



(a) Flowchart **(b) State diagram**

Figure 53. Self-Healing pattern flowchart and state diagram

Table 26. Self-Healing pattern parameters

Parameter	Definition
T_a	Time to activate redundancy (if any)
T_e	Time to execute system progress
T_d	Time to detect illegal system state
T_c	Time to self-correct illegal system state

Resulting Context: An error or failure is tolerated through self-healing. The `Self-Healing` pattern requires the capability of reaching a correct system state from an illegal system state by optionally employing the `Redundancy` architectural pattern, which may reduce error/failure-free performance. The efficiency of self-correction determines performance in the presence of errors and failures. A high performing solution reduces the impact on error/failure-free performance, while at the same time providing efficient self-correction.

A trade-off exists between both, performance in the presence of errors and failures and error/failure-free performance, especially in systems with high error/failure rates. Self-correction is an active approach, where the system is designed with self-correctable illegal system state. Self-correction becomes an intrinsic property of the system. The explicit detection may reduce error/failure-free performance. The pattern's protection domain does not cover illegal system state that is not self-correctable.

The `Self-Healing` pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the `Self-Healing` pattern covers for the remaining error/failure types.

Performance: The failure-free performance $T_{f=0}$ of the `Self-Healing` pattern is defined by the task total execution time without any resilience strategy T_E, the total time to activate redundancy T_a, and the total time to detect illegal system state T_d with the total number of input-execute-output cycles P. The

performance under failure T is defined by $T_{f=0}$ plus the time T_c to self-correct illegal system state, where total time to self-correct illegal system state is number of error or failure times T_c. Assuming constant times T_a, T_d (t_d) and T_c, T can be defined by Eq. 78. When the redundancy is in space, using a ratio for replication in space vs. in time α, T (Eq. 79) can be reformulated.

$$T = T_E + T_a + P(t_d) + \frac{T_E}{M}(T_c) \tag{78}$$

$$T = \alpha T_E + (1-\alpha)NT_E + T_a + P(t_d) + \frac{T_E}{M}(T_c) \tag{79}$$

Reliability: Reliability is defined by the parallel reliability of the N-redundant execution and the performance under failure T (Eq. 80).

$$R_i(t) = 1 - (1 - e^{-\lambda T})^N \tag{80}$$

Availability: The availability A of N-parallel syaytems is defined by M_n (or M) and R_n (or R) where M is the MTTF and R is the MTTR of each individual system (Eq. 81).

$$A_i = 1 - \left(1 - \frac{M}{M+R}\right)^N \tag{81}$$

Examples:

- Self-stabilizing solvers use selective reliability to perform progress in an unreliable mode and detection and self-correction in a reliable mode that uses redundancy. While the unreliable progress may produce an illegal state, the reliable detection and self-correction are able to reach a correct state using redundancy. The reliable detection and self-correction itself are not error tolerant. The protection domain of the redundancy only applies to the unreliable progress. Additional patterns may be employed error tolerance of the detection and self-correction. The efficiency and performance of self-correction in the presence of errors and failures and the error/failure-free performance depend on the overhead of the redundancy (e.g., redundant data structures) and the self-correction (e.g., successive approximation toward a correct state).

Rationale: The `Self-Healing` pattern enables a system to tolerate errors or subsystem errors or failures through adaptation. It relies on the capability of reaching a correct system state from an illegal system state in a finite number of execution steps. It uses explicit error/failure detection and self-correction to actively transition from an illegal to a correct system state. Self-correction is aided by redundancy and intrinsic to the design of the system when this pattern is applied. The pattern has high design complexity and has low dependence on a system's architecture.

Related Patterns: The `Natural Tolerance` structural pattern uses the same `Self-Stabilization` strategy pattern, but with a different architectural pattern (`Self-Masking`). It also optionally employs the `Redundancy` architectural pattern, but for self-masking.

The `Self-Aware` structural pattern uses the same `Self-Stabilization` strategy and `Self-Correction` architectural patterns. It additionally employs the `Fault Diagnosis` architectural pattern and an OODA loop control for error/failure detection and self-correction.

Patterns using the `Self-Stabilization` strategy pattern are a unique class, as self-masking or self-correction may result the system to be in a different correct state after an error or failure than in an error/failure-free execution.

Known Uses:

- A self-stabilizing label-propagation algorithm that computes the connected components in a graph while being tolerant to errors [161] uses the `Self-Healing` pattern, employing the `Self-Correction` and `Redundancy` patterns. This solution leverages selective reliability, where progress is performed in an unreliable mode, and detection and self-stabilization is performed in reliable mode using the `Redundancy` pattern. The reliable mode itself is not error tolerant, as the protection domain of the `Redundancy` pattern only extends to the unreliable mode.
- Two self-stabilizing iterative linear solvers [161], one for the steepest descent and one for conjugate gradient, are tolerant to errors and use the `Self-Healing` pattern with the `Self-Correction` and `Redundancy` patterns. These solvers also leverage selective reliability, with progress in an unreliable mode, and detection and self-stabilization in reliable mode using the `Redundancy` pattern. The reliable mode itself is not error tolerant, as the protection domain of the `Redundancy` pattern only extends to the unreliable mode.

7.4.15 Self-Aware

Name: Self-Aware

Problem: A hardware or software error or subsystem failure in an HPC environment causes a software, such as a numerical library or application, to experience an error and potentially a subsequent failure.

Context: The pattern is a derivative of the `Self-Correction` and `Fault Diagnosis` architectural patterns. It applies to a system that has the following characteristics:

- The system is deterministic, i.e. forward progress of the system is defined in terms of the input state to the system and the execution steps completed since system initialization.
- The system has well-defined parameters that enable a monitoring system to discover the presence of a defect or anomaly in the behavior of the monitored system.
- The interaction between the monitored and monitoring systems is bounded in terms of time.
- The monitoring system has the capability to analyze the behavior of the monitored system.

Forces:

- System performance in the presence of errors and failures is determined by the efficiency in reaching a correct system state from an illegal system state, which may differ for different illegal system states.
- Fault diagnosis for explicit error and/or failure detection may reduce error/failure-free performance.
- Self-correction through an OODA loop control may reduce performance in the presence of errors and failures.
- A correct state may not be reached from all illegal system states, which limits the scope of this pattern.

Solution: The `Self-Aware` pattern relies on the capability of reaching a correct system state from an illegal system state after a finite number of execution steps using explicit error/failure detection and self-correction. Forward progress is guaranteed by design as a correct system state can be reached in a finite number of steps. Self-correction may be as simple as discarding, recomputing, or estimating a wrong value in the system or a wrong or missing output from a subsystem. The correct state reached after self-correction is not necessarily the same state that would have been reached without an error or failure.

The `Fault Diagnosis` pattern is employed to aid in the process of error/failure detection and self-correction in the form of an OODA loop control. Self-correction is performed actively, i.e., through special treatment of the error or failure. The OODA loop control observes errors and failures through system state monitoring, reasons about them through root cause analysis, decides on how to handle them by weighing options and trade-offs, and handles them using self-correction. Since the monitored and the monitoring system may be the same, the OODA loop control may be part of the system state. The scope of this pattern extends to the system state space that is monitored and is either correct or incorrect but self-correctable to a correct state. The entire OODA loop control or parts of it may be a subsystem that does not monitor itself and therefore outside the protection domain. The components of the pattern are illustrated in Figure 54.

Capability: A system using this pattern is self-aware about an error or failure and self-corrects it. This pattern supports the handling of such events with detection, containment, and mitigation using fault diagnosis and self-correction, i.e., dynamic adaptation through observation, analysis, and correction. Error/failure detection is explicit by providing the capability to diagnose faults in the illegal system state. While fault diagnosis is an architectural feature inherited from the `Fault Diagnosis` architectural pattern, self-correction is an architectural feature inherited from the `Self-Correction` architectural pattern. The flowchart and state diagram of the pattern is shown in Figure 55 and its parameters in Table 27.

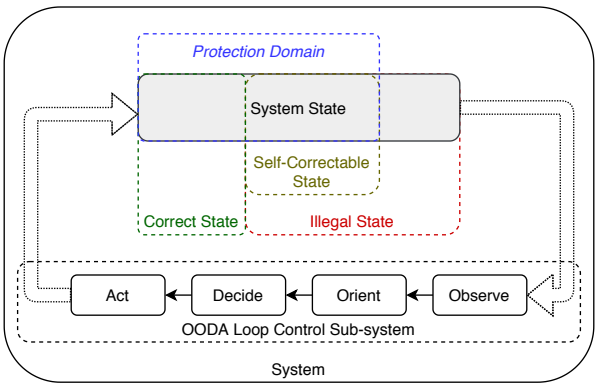


Figure 54. Self-Aware pattern components

Table 27. Self-Aware pattern parameters

Parameter	Definition
T_e	Time to execute system progress
T_m	Time to monitor (sub-) system parameters, including wait and probe times
T_a	Time to perform the cause/effect or effect/cause analysis
T_o	Time to perform the option/trade-off decision making
T_c	Time to self-correct illegal system state

Protection Domain: The protection domain of this pattern extends to the system state space that is monitored and is either correct or incorrect but self-correctable to a correct state. The entire OODA loop control or parts of it may be a subsystem that does not monitor itself and therefore outside the protection domain.

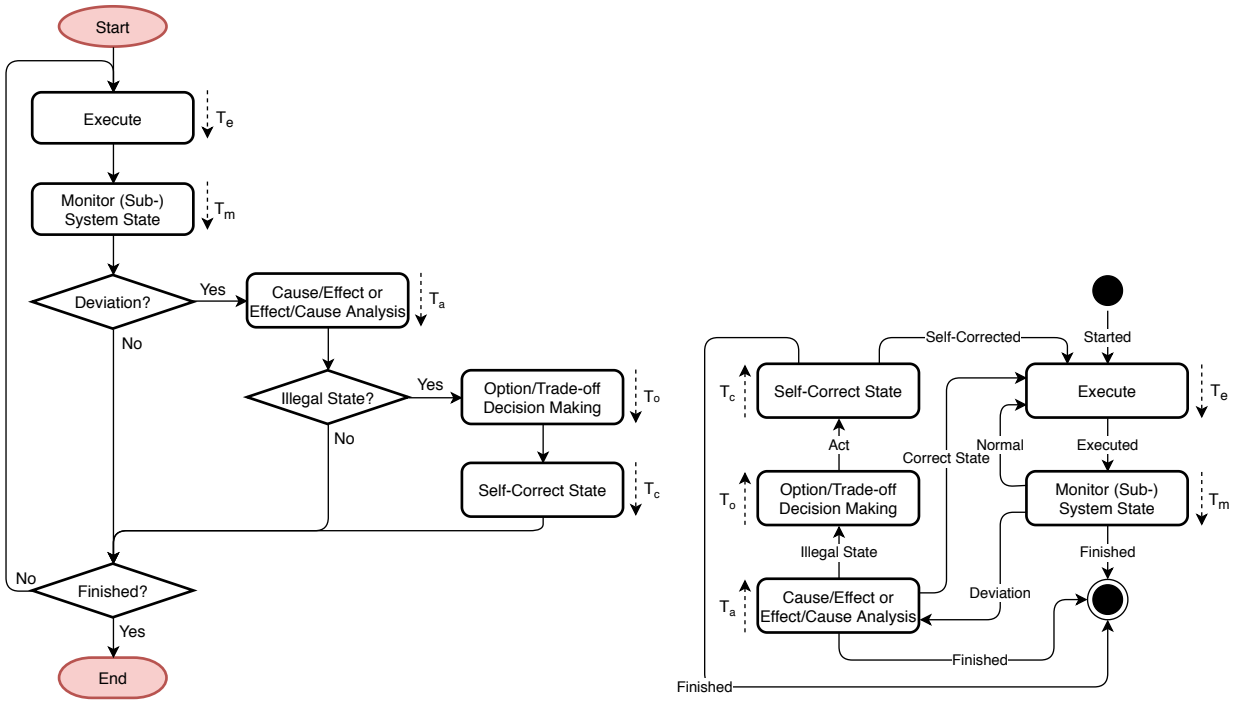
Resulting Context: An error or failure is tolerated through self-awareness. The `Self-Aware` pattern requires the capability of reaching a correct system state from an illegal system state by system state monitoring, reasoning about errors or failures through root cause analysis, deciding on how to handle them by weighing options and trade-offs, and handling them using self-correction.

The employed `Fault Diagnosis` architectural pattern for explicit error and/or failure detection may reduce error/failure-free performance. Self-correction through an OODA loop control may reduce performance in the presence of errors and failures. The efficiency of fault diagnosis and self-correction determines performance in the presence of errors and failures. A high performing solution reduces the impact on error/failure-free performance, while at the same time providing efficient self-correction.

A trade-off exists between both, performance in the presence of errors and failures and error/failure-free performance, especially in systems with high error/failure rates. Self-correction is an active approach, where the system is designed with self-correctable illegal system state. Self-correction becomes an intrinsic property of the system. The pattern's protection domain does not cover illegal system state that is not monitored or not self-correcting.

The `Self-Aware` pattern may be used in conjunction with other structural patterns that provide containment and mitigation in a complementary fashion, where some error/failure types are covered by the other structural pattern(s) and the `Self-Aware` pattern covers for the remaining error/failure types.

Performance: The failure-free performance $T_{f=0}$ of the `Self-Aware` pattern is defined by the task total execution time without any resilience strategy T_E and the time to monitor (sub-) system parameters,



(a) Flowchart **(b) State diagram**

Figure 55. Self-Aware pattern flowchart and state diagram

including wait and probe times T_m with the total number of input-execute-output cycles P. The performance under failure T is defined by $T_{f=0}$ plus the time to perform the cause/effect or effect/cause analysis T_a, the time T_o to perform the option/trade-off decision making and the time T_c to self-correct illegal system state, where total time to perform the cause/effect or effect/cause analysis, to perform the option/trade-off decision making and to self-correct illegal system state is number of error or failure times T_a, T_o, and T_c. Assuming constant times T_m (t_m), T_a, T_o, and T_c, T can be defined by Eq. 82. When the redundancy is in space, using a ratio for replication in space vs. in time α, T (Eq.83) can be reformulated.

$$T = T_E + P(t_m) + \frac{T_E}{M}(T_a + T_o + T_c) \tag{82}$$

$$T = \alpha T_E + (1-\alpha)NT_E + P(t_m) + \frac{T_E}{M}(T_a + T_o + T_c) \tag{83}$$

Reliability: Reliability is defined by the parallel reliability of the N-redundant execution and the performance under failure T (Eq. 84).

$$R_i(t) = 1 - (1 - e^{-\lambda T})^N \tag{84}$$

Availability: The availability A of N-parallel syaytems is defined by M_n (or M) and R_n (or R) where M is the MTTF and R is the MTTR of each individual system (Eq. 85).

$$A_i = 1 - \left(1 - \frac{M}{M+R}\right)^N \tag{85}$$

Examples: In proactive fault tolerance, an OODA loop control is employed that utilizes monitoring tools for collecting sensor data (e.g., temperature, fan speeds, voltages, computational load, memory and storage usage, etc.). It leverages the warning thresholds of these sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72]. The mitigation may use process-level [187] or VM level [134] migration.

Rationale: The `Self-Aware` pattern enables a system to tolerate errors or subsystem errors or failures through adaptation. It relies on the capability of reaching a correct system state from an illegal system state in a finite number of execution steps. It uses explicit error/failure detection and self-correction to actively transition from an illegal to a correct system state. Self-correction is aided by fault diagnosis and intrinsic to the design of the system when this pattern is applied. The pattern has high design complexity and has low dependence on a system's architecture.

Related Patterns: The `Self-Healing` structural pattern uses the same `Self-Stabilization` strategy and `Self-Correction` architectural patterns, but does not employ the `Fault Diagnosis` architectural pattern or an OODA loop control. It optionally uses the `Redundancy` architectural pattern for self-correction.

The `Natural Tolerance` structural pattern uses the same `Self-Stabilization` strategy pattern, but with a different architectural pattern (`Self-Masking`). It optionally employs the `Redundancy` architectural pattern for self-masking.

Patterns using the `Self-Stabilization` strategy pattern are a unique class, as self-masking or self-correction may result the system to be in a different correct state after an error or failure than in an error/failure-free execution.

Known Uses: Self-awareness is heavily used in autonomous systems. In HPC, however, it is mostly a research area and less used in practice. Known uses include:

- Leveraging warning thresholds of sensors as early fault indicators to migrate computation away from compute nodes that are about to fail [72].

7.5 STATE PATTERNS

7.5.1 Static State

Name: Static State

Problem: The scope of the system state that remains unchanged for the entire duration of system operation has unique resilience properties and needs that are different from other aspects of the system state.

Context: The pattern applies to the state of the system that has the following characteristics:

- The overall state of the system is deterministic, i.e., the system output state is determined solely by the input state and the sequence of operations since initialization.
- The notion of lifetime can be associated with the various parts of a system's state.

Forces:

- The state patterns expose an intrinsic property of the system. The precise definition of aspects of the system state requires a detailed understanding of the system structure and operation.
- The facility in identifying the state patterns depends on the layer of system abstraction at which the behavioral patterns associated with the state pattern are instantiated.

Solution: The `Static State` pattern encapsulates the portion of the system state that remains unchanged for the entire duration of system operation. The static state refers to all aspects of a system's state that is computed when the system is initialized, but is not modified during the system operation. From the perspective of an HPC application, the static state is persistent; it includes program instructions and variable state that is computed upon application initialization. Figure 56 shows how the scope of the pattern relates to other state patterns. The encapsulation of such persistent state enables a behavioral resilience pattern to leverage the unchanging nature of the state when performing detection or recovery actions.

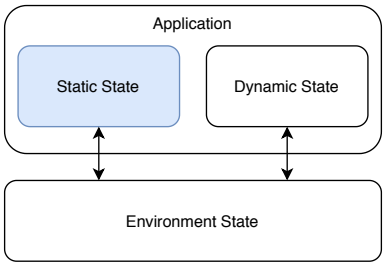


Figure 56. Static State pattern relations

Capability: The static state outlives the process that creates/initializes it. The identification of this pattern follows the steps outlined in Figure 57. In the context of HPC programs, the `Static` state pattern presents an application-centric view of a system. The correctness of the static state at all times is essential to the correct execution and outcome of a program.

Protection Domain: The `Static State` pattern defines the scope of the system state that remains unchanged after initialization.

Resulting Context: The persistent state pattern defines the scope of the static program state. The presence of any errors in the persistent state may not necessarily lead to immediate catastrophic failure of an application program's execution, but might lead the program on divergent paths that cause a failure at a future point in the program's execution. Therefore, the integrity of this aspect of a program's state is essential for correct execution.

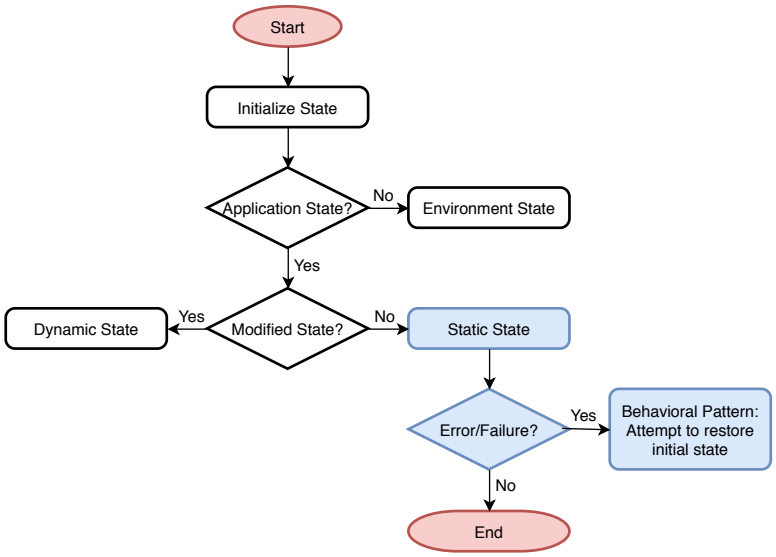


Figure 57. Static State pattern flowchart

Examples: Various algorithm-based fault tolerance methods leverage the property of invariance in the persistent state. These methods maintain redundant information about the application variables in the static state that enables recovery to their default data values at any time during application execution.

Rationale: The isolation of the state that is persistent throughout an HPC application program execution is supported by this pattern. The invariance property of the state encapsulated by this pattern enables the use of a resilience behavioral pattern that leverage this property to detect and recover errors/failure of such state.

Related Patterns: Together with the `Dynamic State` pattern and `Environment State` pattern, the `Static State` pattern defines the overall state of a system.

Known Uses:

- In the design of iterative methods, the static data structures such as the operand matrix A, the right-hand side B, or the preconditioner are computed once in the initialization phase of the application and are unchanged after. Errors in these structures are recovered by maintaining checksums [95].

7.5.2 Dynamic State

Name: Dynamic State

Problem: The scope of the system state that changes as a result of the system operation has unique resilience needs from other aspects of the system state.

Context: The pattern applies to the state of the system that has the following characteristics:

- The overall state of the system is deterministic, i.e., the system output state is determined solely by the input state and the sequence of operations since initialization.
- The notion of lifetime can be associated with the various parts of a system's state.

Forces:

- The state patterns expose an intrinsic property of the system. The precise definition of aspects of the system state requires a detailed understanding of the system structure and operation.
- The facility in identifying the state patterns depends on the layer of system abstraction at which the associated behavioral patterns are instantiated.

Solution: The `Dynamic State Pattern` encapsulates the system state that changes as the system makes forward progress. In an HPC application, the dynamic state includes the variables that are modified by the algorithm. This scope with respect to other state patterns is shown in Figure 58. The encapsulation of the dynamic state enables the identification of the appropriate behavioral resilience patterns to detect and correct fault/errors in such state. Often the behavioral pattern cannot rely on the current error-affected version of the state to perform recovery; the detection/recovery entails the use of a previously preserved version of the dynamic state, or repeating operations from a known stable point. Alternatively, the resilience solution must accept limited loss of information when the behavioral patterns do not completely recreate an error-free version of the dynamic state pattern.

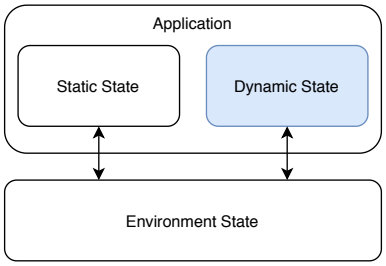


Figure 58. Dynamic State pattern relations

Capability: The state refers to all aspects of the program state that continuously changes as an application program executes. This includes the data values that are computed during system operation, or the control-flow variables that enable forward progress of the system. The correctness of the dynamic state is essential for a correct outcome of a program, but certain algorithms permit faults in the dynamic state to be tolerated. The identification of the dynamic state pattern is illustrated in Figure 59.

Protection Domain: The `Dynamic State` pattern defines the scope of the system state that changes during system operation as the system makes forward progress.

Resulting Context: From the perspective of an HPC program, the encapsulation of the dynamic state enables definition of resilience behaviors, i.e., detection and correction techniques, which must not rely on the specific structure or intrinsic properties of the data contained by the pattern.

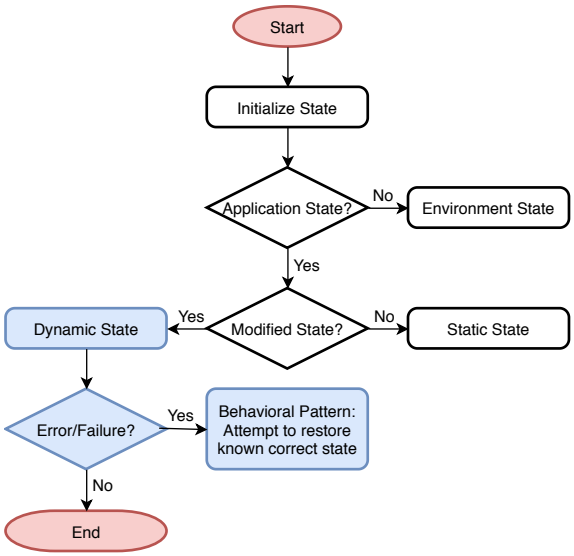


Figure 59. Dynamic State pattern flowchart

Examples: Algorithm-based fault tolerance strategies that guarantee resilience of the dynamic state actively track changes to state. Redundancy methods maintain copies of the change to the dynamic state to compensate for the version that is impacted by an error or failure.

Rationale: The isolation of the dynamic state that is updated throughout an application program execution is supported by this pattern. The *dynamic* feature of this state pattern implies that any errors/failure in such state amounts to lost work. Therefore, the isolation of the dynamic state enables the use of resilience behavioral patterns that explicitly seek to recover the dynamic state. Alternatively, a behavioral pattern may recover an error in the dynamic state without the need to abort and restart an application program using lossy techniques.

Related Patterns: Together with the `Dynamic State` pattern and `Environment State` pattern, the `Persistent State` pattern defines the overall state of a system.

Known Uses:

- The most widely used method for protecting dynamic state is using checkpointing-based roll-back recovery methods.
- Algorithm-specific techniques that support recovery of dynamic state without the need for recovery use lossy methods [113]

7.5.3 Environment State

Name: Environment State

Problem: The scope of the system state that provides a common set of services that support the primary function of the system has unique resilience properties and needs from other aspects of the system state.

Context: The pattern applies to the state of the system that has the following characteristics:

- The overall state of the system is deterministic, i.e., the system output state is determined solely by the input state and the sequence of operations since initialization.
- The complete system state may be described in terms of state relevant to the core function of the system, called the *primary state* and the system state that serves to support its function, called the *environment*.

Forces:

- The state patterns expose an intrinsic property of the system. The precise definition of aspects of the system state requires a detailed understanding of the system structure and operation.
- The facility in identifying the state patterns depends on the layer of system abstraction at which the associated behavioral patterns are instantiated.

Solution: The `Environment State Pattern` encapsulates the system state that supports the operation of the system. The pattern defines the scope of the system state that provides a common set of services in support of the primary function of the system. The environment also facilitates and coordinates the operation of various subsystems in a system. Figure 60 illustrates the relation between the state pattern and the static and dynamic patterns. In general, HPC systems navigate complexity through the definition of abstractions that hide the details of specific functions behind well-defined interfaces. From the perspective of an HPC application, the overall system state may be partitioned into the aspects that serve the primary function of the system and those that provide access to the system resources and services that enable the application to fulfill its function. The encapsulation of the environment state enables designers to instantiate behavioral patterns that are independent of the design of the HPC applications and their algorithms.

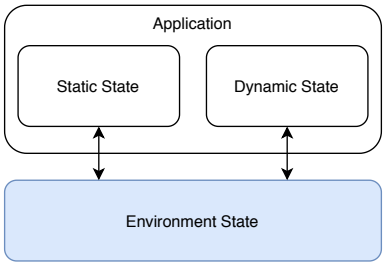


Figure 60. Environment State pattern relations

Capability: An error/failure in the environment state is often immediately catastrophic to the operation of the primary system. The encapsulation of the environment state follows the steps in Figure 61 and enables the development of resilience strategies for the system environment separately from the resilience capabilities of an application program.

Protection Domain: The `Environment State` pattern defines the scope of the components in the environment that support the operation of the primary system. For an HPC system, this scope includes productivity tools and libraries, the runtime system, the operating system, file systems, communication channels, etc.

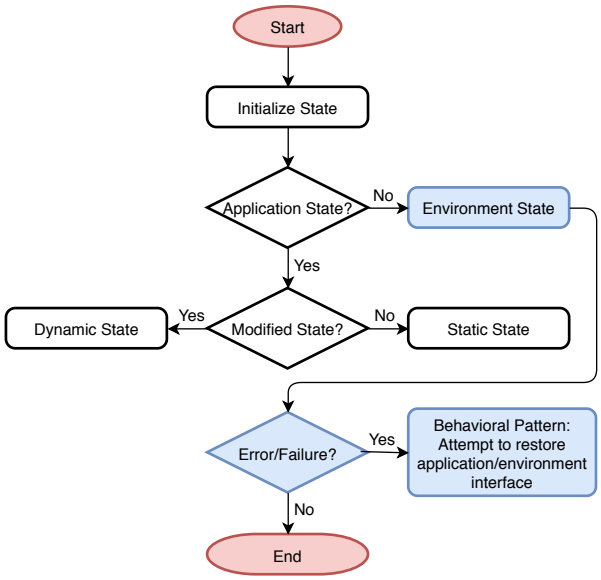


Figure 61. Environment State pattern flowchart

Resulting Context: The pattern defines the scope of the state that support resource sharing, coordination, and communication between the various subsystems. While an application program does not normally have complete control over its environment, it may exert partial control to affect the environment through well-defined interfaces.

Examples: Operating-system based resilience mechanisms focus on the correctness of the data structures within the kernel. These mechanisms are independent of the resilience features of the application program.

Rationale: Any changes in the environment due to an error or failure event directly affects the application program operating within the environment. The encapsulation of the environment enables the resilience behavior of the environment state to be reasoned about separately from the resilience behavior of the primary system state, i.e., an HPC application's state.

Related Patterns: Together with the `Persistent State` pattern and `Dynamic State` pattern, the `Environment State` pattern defines the overall state of a system.

Known Uses:

- The failure of the operating system environment on the compute node of HPC system is often fatal for the application. The Mini-Ckpts framework emphasizes the recovery of the OS environment by preserving kernel structures in persistent memory [79].
- The ULFM MPI provides recovery of the communication environment from the failure of processes by reconstructing the MPI communicator [31].

7.5.4 Stateless

Name: Stateless

Problem: Several resilience strategies operate without the need for defining a specific protection domain. However, behavioral patterns expect a complementary state pattern for completeness.

Context: The pattern applies to the state of the system that has the following characteristics:

- The overall state of the system is deterministic, i.e., the system output state is determined solely by the input state and the sequence of operations since initialization.

Forces:

- The state patterns expose an intrinsic property of the system. The precise definition of aspects of the system state requires a detailed understanding of the system structure and operation.
- The facility in identifying the stateless pattern depends on the layer of system abstraction at which the associated behavioral patterns are instantiated.

Solution: The `Stateless Pattern` facilitates the creation of resilience strategies that are independent of system state. It provides the construct of *null* state in order to create solutions that have a well-defined notion of behavior but need not define a scope for the behavior. From the perspective of an HPC application, the definition of the `Stateless` pattern permits defining the scope of operations that perform fault/error/failure detection or recovery without explicitly defining the variable state of the program that is affected by the operations.

Capability: In the context of HPC programs, solutions that are based on a `Stateless` pattern may include: (i) applications that consist of predominantly memory load operations and rarely contain state-modifying memory and I/O operations; these applications typically perform reductions operations over large number of data elements, and (ii) applications that yield imperfect results since their algorithms are based on approximation and iterative refinement, or use noisy input data to begin with.

Protection Domain: The `Stateless` pattern defines the notion of null system state when defining a resilience solution.

Resulting Context: The stateless pattern is utilized together with behavioral resilience patterns whose actions do not necessitate modifying any aspect of the system state during the detection or recovery. The resilience solution that uses a stateless pattern must select and instantiate a behavioral pattern that can deal with any additional side-effects due to the inclusion of the stateless pattern.

Examples: The use of the *transaction* model to provide resilient behavior is an example of the `Stateless` pattern. Transactions support execution of a sequence of operations that may complete as a unit, or fail; the notion of partial execution is not supported. While the transaction may entail performing computation on data variables, the resilience of the data is independently managed; the resilience solution may be defined with a `Stateless` pattern.

Rationale: The pattern is the equivalent of a *null* pattern that enables resilience solutions to be constructed without the requirement for the behavioral patterns to operate on the program state.

Related Patterns: While the `Persistent State` pattern, `Dynamic State` pattern, and the `Environment State` pattern define the complement of the overall state of a system, the `Stateless` pattern offers the notion of `Null` state.

Known Uses:

- The idempotence property guarantees that any region can be freely re-executed, even after partial execution, and still produce the same result. Language-level constructs as well as compiler-based techniques enable definition of idempotent regions of execution; the recovery of such regions is stateless [56].

8. BUILDING RESILIENCE SOLUTIONS USING RESILIENCE DESIGN PATTERNS

8.1 COMPONENTS OF RESILIENCE SOLUTIONS

The resilience design patterns presented in the catalog offer solutions to problems that repeatedly occur in the design of resilience capabilities for HPC. Each pattern in the catalog presents a solution to a specific problem in dealing with fault, error and failure events. Complete resilience solutions must seek to ensure that an HPC application executes to a correct solution despite the presence of events in the systems. However, architecting a HPC system and its software environment is a complex process. To incorporate resilience into the system design and software development efforts, the interaction between the resilience patterns and the distribution of responsibility between the patterns is as important as identifying the appropriate pattern for a solution based on its characteristics.

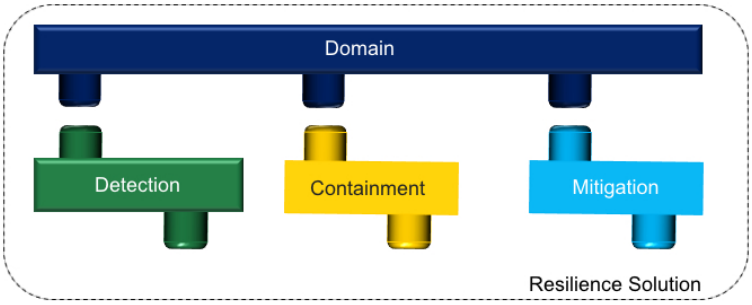


Figure 62. Elements of a resilience solution for HPC systems and applications

The artifacts of a design process that uses the resilience design patterns are complete resilience solutions that provide fault/error/failure detection, containment and mitigation capabilities for a specific fault model. Additionally, the resilience capability must protect a well-defined domain. These key constituents of a complete solutions are shown in Figure 62. The design patterns may be instantiated at multiple layers of system abstraction, and are relevant to various application and system scales. However, many of the patterns in the catalog individually provide partial solutions by supporting only one or two out of the detection, containment and mitigation solutions. For system and application designers to use these patterns in the construction of resilient versions of their designs, these patterns must be organized into a well-defined system of patterns.

8.2 DESIGN SPACES

During the design of a complete resilience solution, there are various factors that must be considered in the selection of patterns besides their detection, containment and mitigation capabilities, including the layer of abstraction for their implementation, scalability of the solution, portability to other architectures, dependencies on any hardware/software features, flexibility to adapt the solution to accelerated fault rates, capability to handle alternative fault and error events, the performance and performance overheads.

We define a framework that enables the composition of the resilience design patterns. A pattern framework enables the creation of the outline of the resilience solution that captures the dimensions and capabilities of the patterns, reveals and clarifies the relationships between the patterns. The combination of these patterns based on the guidelines offered by the hierarchical classification scheme enables the complete solutions for resilience to specific fault models in HPC systems. However, there is sufficient flexibility to adapt the solution to specific situations. The framework also enables the designer to navigate the various issues that must be addressed in the process of developing practical resilience solutions.

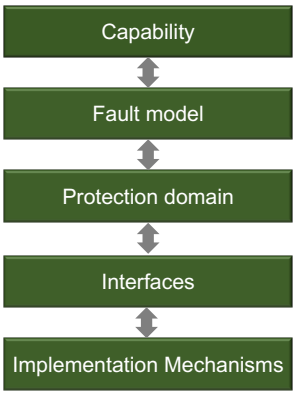


Figure 63. Design Spaces for construction of resilience solutions using patterns

In order to articulate a systematic method for customized designs, the framework is based on *design spaces* (Figure 63). These design spaces provide guidelines for the decision making in the design process, which consists of selection of the appropriate patterns based on the requirements of protection and the cost of using specific patterns. The framework consists of the following design spaces:

- **Capability**: The patterns must support capabilities that enable the detection, containment, mitigation of faults/errors/failure events.
- **Fault model**: The identification of the root causes of fault events and their impact and propagation through the system must be well-understood to provide effective solutions.
- **Protection domain**: The definition of the protection domain enables clear encapsulation of the system scope over which the resilience patterns operate.
- **Interfaces**: The identification and implementation of the activation and response interfaces for behavioral patterns affect the propagation of fault/error/failure event information.
- **Implementation mechanisms**: The implementation design space is concerned with constraints imposed by specific features of hardware, execution or programming models, software ecosystems.

These design spaces represent the important aspects of the design process that a designer must contemplate in order to create effective and efficient resilience solutions. The design spaces provide a structured flow to the design process since each design space presents one or more key decision points that shapes the resilience solution, as well as the overall design of the system. Design spaces also provide a framework to guide the creation of cross-layered resilience solutions that leverage capabilities from multiple layers of the system abstraction. With the use of resilience patterns in the context of the framework provided by the design spaces, HPC system designers, users and application developers may evaluate the feasibility and effectiveness of novel resilience techniques, as well as analyze and evaluate existing solutions.

The design spaces are ordered from the abstract to concrete, and they cover the important structural and behavioral design considerations. As a designer advances through these design spaces, they are able to develop a clearer understanding of the solution profile and the general constraints, which enables them to select the appropriate patterns from the catalog and decide on implementation alternatives. Designers may use various approaches to navigate the design spaces, including a strictly top-down approach, in which the design is driven by the event type and model that a system must be protected against, and the implementation of the system is adapted to enable the system to survive the different ways in which the event may impact the reliability of the system. Alternatively, in a bottom-up approach, the resilience

capability must be woven into the existing hardware or software component designs and interfaces, and additional components are included to enhance the protection coverage, or to handle specific fault model behaviors. Often, designers may be required to take a hybrid approach, in which the design spaces are revisited in an effort to refine a design, to optimize the features of a solution, and to enable designers to overcome constraints imposed by any hardware or software system features.

9. A PATTERN LANGUAGE FOR HPC RESILIENCE

A pattern language is considered as a system of patterns that are related with each other in a hierarchy or network. The structure of the network helps designers makes sense of the individual patterns, as well as helps anchor them in various combinations to provide complete solutions. Our pattern language for HPC resilience explains the discipline to use the various design patterns to create effective and efficient resilience solutions. The elements of the language are the patterns detailed in Section 7. The language guides a designer from the beginning of a design problem to the realization of its solution.

9.1 TYPES OF PATTERN RELATIONS

In general, a pattern language has the structure of a network such that patterns that are related by some measure of relevance are linked together. The definition of the linkage between patterns is the key for a set of patterns to become a language rather than be seen as a collection of isolated, standalone ideas for design.

In contrast to a pattern classification, which provides the means to group patterns based on a set of rules or pattern properties, a pattern language explicitly interweaves the patterns in the catalog based on every possible (but at least one) type of pattern interrelation. Based on the interrelations between the patterns, the complete set of the resilience patterns in the catalog forms a language. Therefore, making these relations explicit is essential to the process of developing a pattern language.

Table 28. Types of pattern relations

Relation	Description	Inverse Relation
abstraction	Pattern x describes an abstract form of pattern y	*specialization*
specialization	Pattern x provides specific details about pattern y	*abstraction*
used with	Pattern x is used to address different problem than y; may be used together	*conflict*
conflict	Pattern x and y are not suitable to be applied together for a specific problem	*used with*
similarity	Pattern x and y have some similar features, but address different problems	-
domain	Pattern x specifies the protection domain for the behavioral pattern y	-

Highlighting these relationships between patterns enables designers to grasp the entire collection of patterns. Therefore, the pattern language also serves as an index to the catalog of resilience design patterns. For the resilience design patterns, various types of pattern relations may be used to express kinds of relatedness between the patterns. Table 28 provides an overview of the types of relationships between the resilience patterns. These interrelations between the patterns form the links between patterns in the network, thereby defining the order in which the patterns should be applied to a HPC resilience design.

9.2 STRUCTURE OF THE PATTERN LANGUAGE

Forming a pattern language requires establishing rules for linking each of the patterns in the catalog. This is a particularly complex task for resilience design patterns due to their large number and the various design considerations and optimizations that must be accounted for. To enable designers to understand the language and for rapid analysis of the relationships between the various resilience design patterns, we have represented the pattern language using a graph. Each pattern is represented as a vertex and every relation between any two patterns is represented by an edge in the graph network. Based on the type of relation

between the patterns, the edges may be directed or undirected. This representation of our pattern language is shown in Figure 64.

The pattern graph represents the language since it captures all the interrelations between the resilience patterns. This representation of the language is intended to make these patterns useful for a broad target audience. System architects may use the language to understand the scope of the problem and develop a high-level layout of the pattern-based solution, while the designers of individual component may use the language to understand the pattern relationships that directly impacts their part of the design.

The use of the graph representation of the pattern language also enables structured analysis of resilience solutions. For example, a simulator may use the graph representation of the pattern language for design space exploration to evaluate alternative combinations of patterns that may have different complexity and performance characteristics. Similarly, the graph representation of the language may enable a runtime system or scheduler to make dynamic decisions about the suitability of instantiating a specific combination of patterns.

The graph representation of the language highlights the pattern relations (listed in Table 28) between all the resilience patterns in the catalog. The vertices representing the patterns are clustered to align with the classification scheme described in Section 6. The state patterns and the three categories of the strategy patterns are represented in different colors. The derivative patterns of each of these classes are represented in the same color as their parents. The patterns are ordered from abstract to concrete to enable designers to focus on the contours of a solution before delving into implementation specifics. Additionally, most of the relations are directed from one pattern to another, but they often also imply an inverse relation in the opposite direction. Therefore, every edge in the graph may be treated as a directed connection between patterns that highlights a specific relation between the two patterns. From the designers' perspective, this representation of the pattern language provides the methodology for selecting patterns from the catalog. The language outlines the ordering of the critical decisions that must be considered when designing and implementing a resilience solution.

9.3 USING THE PATTERN LANGUAGE

Our pattern language spans all the way from the initial architecture of a resilience solution down to the lowest level details of the implementation for a specific architecture and software environment. Defining which patterns to use and how to combine them is the very essence of the pattern language. However, an emphasis of a pattern language is often not represented in the inherent structure of the pattern language. Since our pattern language is in the form of a network, there is no one sequence that perfectly captures the pattern relationships. Therefore, when selecting a suitable combination of patterns for constructing a resilience solution, there are numerous ways in which the network of patterns may be traversed.

9.3.1 Structured Design of HPC Resilience Solutions

The pattern language outlines the intended flow of information when reading or browsing the pattern catalog. Using the pattern language, solutions are designed incrementally by exploring the links of the network that represents the pattern language. This yields an order in which the patterns should be applied to a design problem, which is called a pattern language sequence. However, the pattern sequence is not strictly linear. Various stakeholders, including system HPC system architects, hardware and software designers, application developers and users can construct solutions by discovering a sequence that fits their design objectives and constraints. For the following key aspects of a resilience design process, the pattern language enables the discovery of pattern-based solutions:

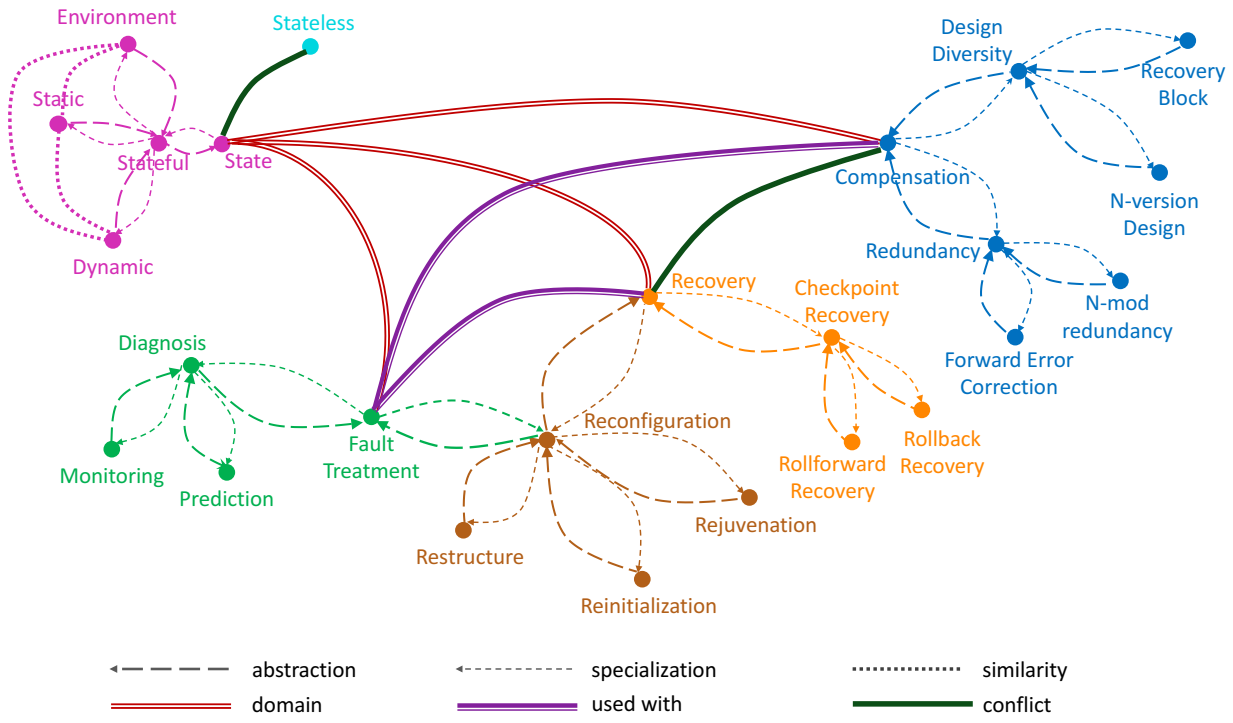


Figure 64. Resilience pattern language representation

- **Protection Domain:** Based on the scope of the system that the solution intends to protect, the language may traverse the network starting from the state pattern vertices, and then identify the behavioral patterns to protect the selected domain.
- **Fault Model:** The type of event that a solution is designed for forces the designer to consider one of the strategy pattern vertices, before exploring the network links that will enable the identification of derivative patterns that are capable of handling the consequences of a specific fault, error or failure type.
- **Fault Management Capabilities:** Based on whether the pattern offers detection, containment, recovery or masking semantics, or a combination of these capabilities, the traversal may commence at specific cluster in the graph representing the language.
- **Implementation-Driven:** Often the design of a resilience solution may be constrained by the idiosyncrasies of a hardware architecture or software environment, or by the availability of specific technologies for supporting a resilience solution. In this case, the pattern language may be used to identify the structural patterns first, and traverse the links of the network towards the more abstract behavioral patterns and the state patterns to evaluate the effective protection domain and capabilities.

9.3.2 Other Design Considerations for Resilience Solutions

While the pattern language for designing resilience solutions for HPC systems is intended to provide designers with a roadmap to create solutions, there are various other critical decisions that must be considered in addition to the fundamental choices of protection domain, fault model, capability and implementation mechanisms. These include:

- **Design complexity of the solution:** The effort necessary to incorporate the patterned solution in the overall design of a system.
- **Time overhead in the absence of fault, error, or failure events:** The impact of the pattern (in terms of time to solution) on the fault-free operation of a system.
- **Time overhead to manage fault, error, or failure events:** The impact on time to solution on account of the actions required to manage an event.
- **Space overhead of the solution:** The number of additional components or subsystems that the solution requires.
- **Power overhead of the solution:** The impact of applying the pattern on the system's power consumption.

For each optimization objective, the graph edges may be annotated with relations that express the implications of selecting a pattern when traversing the network. Using these additional relations, the pattern language may be used to discover an ordering of patterns that meets these design considerations as well as the functional requirements of a solution for confronting a specific type of fault, error or failure.

10. CASE STUDIES

10.1 CHECKPOINT AND ROLLBACK SOLUTION FOR PROCESS FAILURES

This section explores use cases for the application of resilience design patterns to the systematic design and analysis of resilience solutions. We use the pattern-based approach for understanding existing solutions with the view to adapt the solution to future generations of HPC systems. The case study describes the pattern-based design process for the fault model of process/task failure on a notional architecture and software environment of a HPC system.

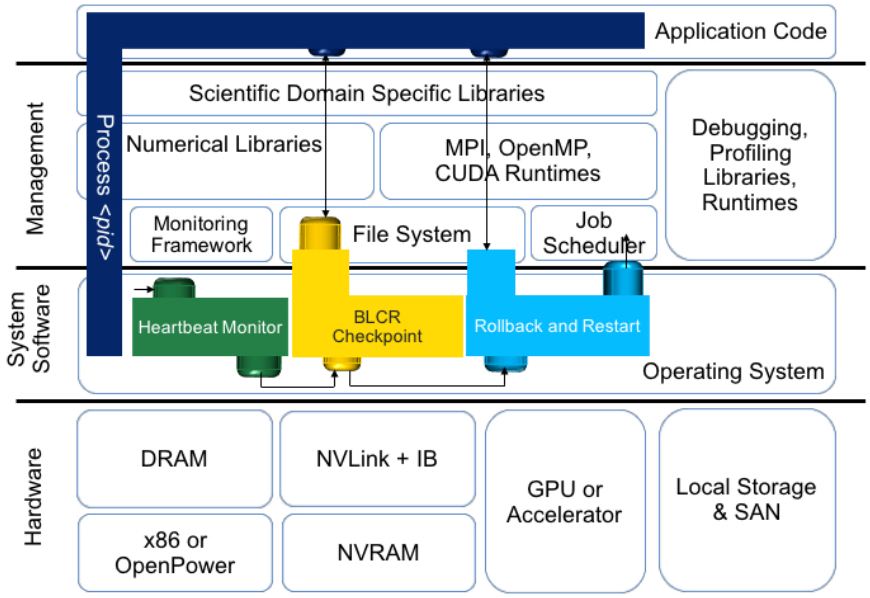


Figure 65. Resilience solution case study: Checkpoint & restart using BLCR

For this case study, we aim to develop a resilience solution that enables an HPC application to survive a process failure. In an HPC environment, the diagnosis of the precise root cause of these failures is difficult due to the lack of sufficient hardware-level debugging information. For designing a purely software-based solution, the fault model is a process crash or hang whose cause is unknown. This type of failure results from the presence of a fault in the processor or memory that activates to cause an error in the form of an illegal instruction or an invalid address in the program state. When the program execution encounters the address in the program state that is in error state, the process may crash or hang.

C/R solutions are the often used to support resilience to process failures in HPC systems. We reexamine this well-known software-based solution using the structured pattern-based approach to analyze composition of the constituent patterns needed to design this solution. Such analysis will be useful for adapting C/R solutions to future systems and evaluate their performance characteristics. The goal of a complete C/R solution is to recover a failed process such that the application may resume from an error-free state. This requires that the solution capture the image, or snapshot, of a running process and preserves it for later recovery. The checkpoint is typically committed to parallel file system on disk storage. For parallel applications, the C/R framework's coordination protocols produce a global snapshot of the application by combining the state of all the processes and their communications in a parallel application. Since most parallel applications using the message passing interface (MPI) define a MPI process to be a Portable Operating System Interface (POSIX) process, the protection domain of the solution must cover the complete POSIX process state and the point-to-point and collective messages exchanged between the

processes.

The protection domain for the solution must cover the global process state. Therefore, we fuse the Persistent and Dynamic and Environment state patterns, which extends the domain of our system-level checkpointing solution to the entire memory associated with a process; in a Linux-based environment, the protection domain extends over the virtual address space of a Linux process.

For the detection of a process failure, we require instantiation of the Fault Treatment strategy pattern. Specifically, our solution requires a Fault Diagnosis architecture pattern to discover the location of the failure and the type of event, which is enabled by a Monitoring structural pattern. The instantiation of the Monitoring pattern is a kernel-level heartbeat monitor, which is deployed in the system to detect whether the process is alive.

For the selection of a recovery pattern, there are key two considerations: (i) the frequency of node failures; and (ii) the performance and resource overhead of applying the pattern. The space overhead incurred by instantiating a Compensation strategy pattern for recovery is substantial due to the need to replicate the protection domain. For systems that experience process failures infrequently, the use of a compensation-based solution proves prohibitively expensive. Therefore, for the failure recovery we select the Recovery strategy pattern. The Checkpoint Recovery architectural pattern is appropriate since Linux provides the capability for a running process to be interrupted and its context to be written to disk. Also, the process state is deterministic and defined by the state of the program counter and the registers; therefore, the Rollback structure pattern is suitable for implementation at the operating system level. With the selection of this pattern, protection domain of the failure is to be limited to a single process context, which implicitly defines the containment pattern. The implementation of the recovery pattern requires a disk storage system, to which the checkpoint, i.e., the process state captured during failure-free operation is exported. The performance overhead of these patterns during failure-free operation and the recovery time are dependent on bandwidth available between memory and the disk system.

The implementation of the patterns, which is illustrated in Figure 65, is implemented using the BLCR [65] framework. Since BLCR does not provide a failure detection mechanism, the Monitoring pattern is implemented by a kernel-level module that uses heartbeat monitoring to check for process liveness. BLCR provides a completely transparent checkpoint of the process, which saves the current state of a Linux process. The framework uses a coarse-grain locking mechanism to momentarily interrupt the execution of all the threads of the process, giving them a global view of its current state. The entire state is saved, including the CPU registers, the virtual memory map as well as the function call stack. From the perspective of an application programmer, the checkpoint routine returns with a different error code, to let the caller know if this function call returns from a successful checkpoint or from a successful restart. The Rollback pattern handles recovery after the detection of a process failure by restoring the context file set from the stable storage, and recreating the process on the same hardware, with the same software environment. BLCR also provides an API for applications programmers to manage pattern behavior through hooks that allow the application to block off code sections where checkpoints are not permitted. These hooks also give applications a chance to respond to checkpoint/requests and take appropriate action, which provides an application programmer with explicit control over the pattern's activation and response interfaces.

10.2 PROACTIVE PROCESS MIGRATION FOR FAILURE AVOIDANCE

In HPC environments, various fault indicators indicate the imminence of error or failure events. The goal of this case study is to design and implement a proactive resilience solution using the structured design pattern-based approach. Various resilience strategies are inherently reactive, i.e., they respond to the occurrence of an error or a failure event and seek to prevent the event from affecting the correct execution of an HPC application. In contrast to a reactive solution that seeks to recover from an error or a failure event after the fact, the notion of proactive fault tolerance responds to faults in a system and seeks to prevent their activation into errors/failures. This analysis of an error or failure avoidance solution is intended to identify the patterns that must instantiated for a proactive design approach, and to articulate the protection domain, capabilities and implementation specifics of the solution. The case study describes the pattern-based design process for faults that have the potential to result in future failure events in an HPC environment.

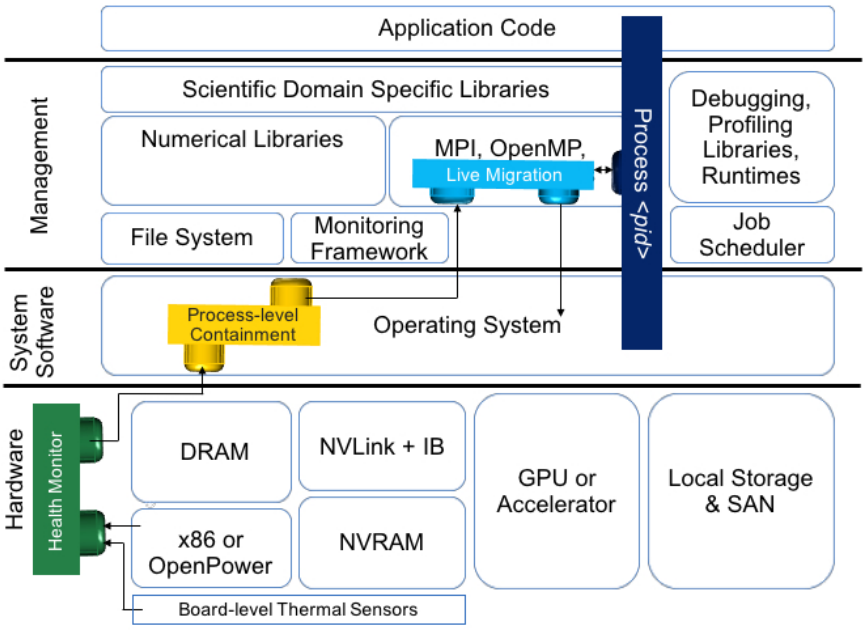


Figure 66. Resilience solution case study: Process migration

The key to designing a proactive strategy is the identification of fault indicators that can sufficiently predict the activation of an error or failure. The fault model for this case study is a defect in the system that has the potential to result in an error or failure. We consider faults that are known to cause errors, which result in application crashes. Using design patterns, we seek to develop a software-based solution that can preemptively migrate parts of an application away from system resources that are about to fail. In a HPC system, the failure of a compute node causes termination of the application processes running on that node. Since the presence of a fault does not impact the correctness of an application program until it activates, the solution supports proactive failure avoidance from the application's perspective. We select the protection domain by fusing the `Persistent` and `Dynamic` and `Environment` state patterns. Much like the C/R solution, the protection domain covered by these patterns includes the complete POSIX process state in a Linux environment. The ultimate objective of the solution is to preemptively migrate the application processes from compute nodes where a failure is likely to cause them to crash to another node in the system.

To anticipate the occurrence of a failure, the solution must observe critical indicators that will predict the likelihood of a failure. We apply the `Fault Treatment` strategy pattern, which is instantiated as a `Fault`

Diagnosis pattern in every node of the HPC system. This pattern is instantiated as a Prediction structural pattern, which enables estimating the possibility of an imminent error or failure event. Its activation interface reads health monitoring data for the various components in each compute node and its response interface signals the possibility of a node failure. The prediction pattern creates a control feedback-loop such that a mitigation pattern can take preventive action to avoid failure of the processes running on the node. Since the solution addresses faults in the computes nodes, it requires the instantiation of another Fault Treatment pattern for mitigation rather than a Recovery strategy pattern. For this solution, we assume that the number of nodes allocated for an application run are determined during startup and are fixed for the lifetime of the application run. If the application uses all nodes in the allocation at initialization and leaves no spare nodes, the inclusion of a Compensation strategy pattern is not a suitable alternative. The Reconfiguration architectural pattern is applied, which is instantiated in the form of a Restructure structural pattern that isolates a failing node and migrates the application processes to an alternative compute node in the system. The containment is implemented by a kernel level module provides containment for the fault by identifying the process that is executing on the node which the Prediction pattern has assessed vulnerable due to a specific set of changes in operating conditions of the node.

The overall structure of the pattern-based design is illustrated in Figure 66. The implementation of the Prediction pattern is realized as a per-node health monitoring mechanism that uses various platform-level indicators in the system. It uses platform data available through the IPMI interface, which relies on the baseboard management controller (BMC) to collect sensors readings for health monitoring, including the data on temperature, fan speed, and voltage. The response interface of the pattern notifies the scheduler when the sensors indicate deterioration of a node's health. Since the behavior of the Recovery strategy pattern used by this solution entails performing a live migration of a POSIX process in the context of the MPI execution environment, the implementation of the Restructure pattern is realized within the system's job scheduler. The pattern identifies healthy nodes in the system as potential destinations for the process migration. Once a destination node has been identified, the pattern initiates the migration of the process from source to destination node. It is imperative the entire context of a process be migrated when the presence of a fault is inferred on a compute node. Therefore, the migration entails transfer of the process image, which occurs by a page-by-page copy of the address space. The implementation then synchronizes all the MPI processes to a consistent state, after which the in-flight data in the MPI communication channels is drained. Once all the MPI processes reach a consistent global state, the remaining dirty pages, which includes the registers, signal information, pid, files, etc. to the destination node. Once the mapping of the processes to nodes in the system has been restructured, the communication channels and the previously saved in-flight messages are restored. The migrated processes resume execution on the destination node. The implementation of the patterns in this solution ensure the transparency of the proactive migration to the HPC application.

10.3 CROSS-LAYER HARDWARE/SOFTWARE SOLUTION FOR SOFT ERROR RESILIENCE

Among the major challenges for future hardware and software design is the requirement of solutions to balance the performance and power efficiency with the robustness offered by a solution. Resilience solutions that are implemented across multiple layers of the system stack, where partial solutions work together to achieve required degrees of resilience in a highly performance and power-efficient manner. While design patterns provide a systematic framework to re-evaluate and refine existing resilience solutions, they are also a valuable tool to design novel solutions. This section demonstrates the use of resilience design patterns for the exploration and assessment of such novel cross-layered solutions. The case study describes the pattern-based design process for the fault model of transient errors on a notional architecture and software environment of a HPC system.

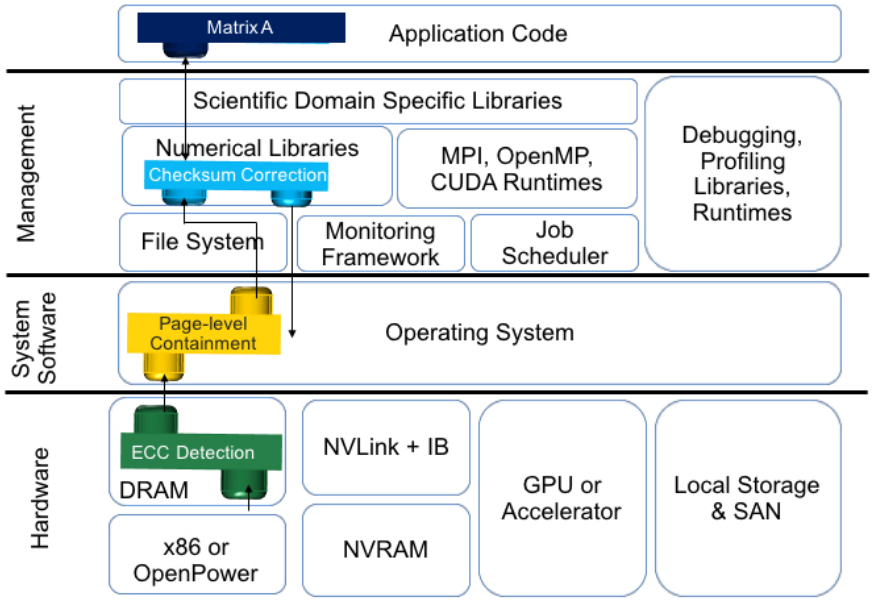


Figure 67. Resilience solution case study: Cross-layer design using ECC with ABFT

In this case study, we use design patterns as building blocks to explore novel resilience solutions that leverage capabilities from various layers of the system stack. By navigating the design spaces of the resilience design pattern framework, we can evaluate the effectiveness of instantiating a detection, containment or mitigation pattern at a specific level in the system stack and systematically construct a cross-layer resilience solution that connects patterns from multiple layers. The structured approach supported by the framework also enables refining the cross-layered solution. The aim of this case study is to develop a solution that provides soft error detection and correction for HPC application data structures. The fault model that we consider is transient errors in memory structures that cause multiple bit flips in the application's data or control variables, which may result in outcomes ranging from incorrect results to fatal program crashes.

The DRAM memory chips used in HPC systems use ECC to detect and correct bit flip errors. Similarly, algorithm-based fault tolerance techniques are available that maintain checksums for data structures to detect and correct data value errors at the application level. However, the lack of formal methods to combine these solutions often precludes cross-layer hardware-software designs that cooperative protect the application data. Our proposed solution is designed to support transient error resilience for a scientific application that uses an iterative linear solver method. In general, these methods solve a system of linear

equations represented as A.x = B, where x is the solution vector, A is the operand matrix and b is a known vector. The iterative algorithm begins with an initial approximation of the solution x, and refines this solution until the residual norm is below a certain error bound. Therefore, the matrix A and vector b are scoped within `Static` state patterns, the solution vector x in a `Dynamic` state pattern, and the remaining variable state is contained within an `Environment` pattern. While the solution vector is often tolerant to perturbations due to the iterative nature of the algorithm, any transient errors within the scope of the two `Static` state patterns affects the correctness of the solver. Therefore, we define the protection domain of our cross-layer solution to include only these static patterns.

For achieving error detection and correction in digital data, the general approach is to add redundant information to discover errors and reconstruct the original data. This approach fits the `Compensation` strategy pattern, which may be instantiated in the form of a `Forward Error Correction` pattern. For the detection of the transient errors, we assume that this pattern is implemented in the form of ECC in the DRAM modules, which supports single-bit error correction and double-bit error detection. Therefore, the instantiation of this structural pattern handles both detection and mitigation for single-bit errors. Double-bit errors result in an ECC violation on the memory line, which is asynchronously communicated by the `Forward Error Correction` pattern to the operating system via its response interface by raising a machine check exception. For the containment of the double-bit error, we deploy a `Fault Treatment` pattern in the operating system, since the OS views the double-bit corruption as a fault. Since the pattern must discover whether the double-bit corruption maps to the protection domain specified by the state patterns, it is instantiated as a `Fault Diagnosis` pattern, specifically as a `Monitoring` structural pattern. For recovery of variable state scoped by the `Static` state pattern, the solution instantiates the `Compensation` strategy pattern. It uses the `Redundancy` architecture pattern and structures the solution based on the `Forward Error Correction` pattern.

The instantiation of the patterns across the system stack is illustrated in Figure 67. The `Monitoring` pattern for containment is implemented as a kernel-level module that maps the physical address to the virtual address space to discover whether the fault may be contained within the `Static` state pattern. The pattern's response interface treats the presence of the fault in the state pattern as an application error and notifies the numerical library. When the error is outside the scope of the `Static` state pattern, the response interfaces indicates to the kernel module that the error is unrecoverable, which results in the OS killing the application. Besides the `Forward Error Correction` pattern in ECC for single-bit error recovery, another instance of this pattern type is implemented in the numerical library to handle double-bit errors. The implementation maintains a set of checksums for the matrix A and vector b. The checksums enable the identification of the element of the matrix affected by the error, and substitution of that element with a correct value using the remaining uncorrupted elements in the row/column and the checksum values. The instantiation of the `Forward Error Correction` pattern at the application library level provides context about the significance of the error to the overall application, and is able to employ an algorithm-specific fault tolerance detection and correction method, which is more cost effective for double-bit error mitigation than system-level bulk reliability provided by hardware-level solution such as an enhanced ECC that supports double-bit correction. Therefore, the cooperation between patterns across system layers supports a flexible memory protection mechanism to single and double-bit memory errors, which allows the application to resume operation towards completion rather than experience a fatal crash with higher performance and energy efficiency.

11. SUMMARY

In this document, we presented an updated and revised specification of the concept of resilience design patterns, which support a systematic approach to designing and implementing resilience solutions. The structured approach to designing and implementing HPC resilience solutions is useful to reduce the complexity of the design process, and is particularly relevant for the future generations of extreme-scale parallel systems and their applications. The patterns are based on well-known and well-understood solutions that have been applied in HPC systems and provide solutions to specific problems encountered in the management of resilience. We have formatted each of these solutions in the template of design patterns such that they may be used by designers as reusable templates when building and refining resilience solutions. The resilience design patterns presented in this document support detection, containment, masking and recovery capabilities. This revision contains additional patterns as well as improvements to the textual descriptions of the patterns from the previous version of the specification and the inclusion of graphical representations of each pattern. We modified the classification scheme, which organizes the resilience patterns in a layered hierarchy, to highlight the important relationship between the state and behavioral patterns, and to accommodate the new patterns introduced in this version. The new classification is designed to expose the relationships between the various patterns in the catalog and their capabilities, which enables system architects to approach the solution abstractly while allowing individual component designers and developers to restrict their work to the level that directly impacts their portion of the solution. In this version, we have also made minor changes to the design framework that simplifies the construction of complete resilience solutions through the composition of design patterns. The framework is intended to be useful in creating portable solutions, whose implementation may be customized to specific architectures and software stacks. The resilience patterns and the pattern-oriented framework facilitates the exploration of a variety of alternative solutions, the refinement and optimization of individual solutions, and the investigation of the effectiveness and efficiency of solutions. This structured approach aims to address the resilience challenge for extreme-scale HPC systems through a systematic design of solutions with an emphasis on optimizing the trade-off, at design time or runtime, between the key system design factors: performance, resilience, and power consumption.

12. REFERENCES

[1] Intel MPI library. https://software.intel.com/en-us/intel-mpi-library.

[2] MPICH: High-performance portable MPI. https://www.mpich.org/.

[3] MVAPICH project. http://mvapich.cse.ohio-state.edu/.

[4] Nagios monitoring system. https://www.nagios.org/.

[5] OpenMPI. https://www.open-mpi.org.

[6] Overview of syslog. https://www.gnu.org/software/libc/manual/html_node/Overview-of-Syslog.html.

[7] Scalasca. https://icl.utk.edu/scalasca.

[8] Vampir - performance optimization. https://vampir.eu.

[9] Mellanox Infiniband FDR 56Gb/s for server and storage interconnect solutions. http://www.mellanox.com/related-docs/whitepapers/WP_InfiniBand_FDR.pdf, 2011.

[10] Intelligent platform management interface (IPMI), v2.0 specification. http://www.intel.com/content/www/us/en/servers/ipmi/ipmi-home.html, 2015.

[11] B. Acun, A. Langer, E. Meneses, H. Menon, O. Sarood, E. Totoni, and L. V. Kale. Power, reliability, and performance: One system to rule them all. *Computer*, 49(10):30–37, October 2016.

[12] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual International Conference on Supercomputing*, ICS '04, pages 277–286, 2004.

[13] A.M. Agbaria and R. Friedman. Starfish: Fault-tolerant Dynamic MPI Programs on Clusters of Workstations. In *Proceedings of The Eighth International Symposium on High Performance Distributed Computing*, pages 167–176, 1999.

[14] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker. Lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. In *Proceedings of IEEE/ACM International Conference for High Performance Storage, Networking, and Analysis (SC14)*. IEEE/ACM, 2014.

[15] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, August 1977.

[16] Altair Engineering, Troy, MI, USA. OpenPBS documentation, 2007.

[17] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray XC series network, Cray Inc., 2014.

[18] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09)*, pages 1–12, Rome, Italy, 2009. IEEE.

[19] Hartwig Anzt, Jack Dongarra, and Enrique Quintana-Orti. Fine-grained bit-flip protection for relaxation methods. *Journal of Computational Science*, 36:100583, 2019.

[20] Hartwig Anzt, Jack Dongarra, and Enrique S. Quintana-Ortí. Tuning stationary iterative solvers for fault resilience. In *Proceedings of the 6th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, New York, NY, USA, 2015. Association for Computing Machinery.

[21] Rizwan Ashraf, Saurabh Hukerikar, and Christian Engelmann. Shrink or substitute: Handling process failures in HPC systems using in-situ recovery. In *Proceedings of the* 26^{th} *Euromicro International Conference on Parallel, Distributed, and network-based Processing (PDP) 2018*, pages 178–185, Cambridge, UK, March 21-23, 2018. IEEE Computer Society, Los Alamitos, CA, USA.

[22] T. Austin, V. Bertacco, S. Mahlke, and Yu Cao. Reliable systems on unreliable fabrics. *IEEE Design Test of Computers*, 25(4):322–332, 2008.

[23] T. M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 196–207, 1999.

[24] Algirdas Avižienis. Toward systematic design of fault-tolerant systems. *Computer*, 30(4):51–58, April 1997.

[25] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Secure Computing*, pages 11–33, January 2004.

[26] Rajanikanth Batchu, Yoginder S. Dandass, Anthony Skjellum, and Murali Beddhu. MPI/FT: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4):303–315, 2004.

[27] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. FTI: High performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 32:1–32:32, 2011.

[28] Jeffrey Becklehimer, Cathy Willis, Josh Lothian, Don Maxwell, and David Vasil. Real time health monitoring of the Cray XT3/XT4 using the simple event correlator (SEC). In *Cray Users Group*, 2007.

[29] Anne Benoit, Aurélien Cavelan, Yves Robert, and Hongyang Sun. Optimal resilience patterns to cope with fail-stop and silent errors. Research Report RR-8786, LIP - ENS Lyon, October 2015.

[30] D. Bernick, B. Bruckert, P.D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. NonStop Advanced Architecture. In *International Conference on Dependable Systems and Networks*, pages 12–21, 2005.

[31] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[32] Jan Borchers. *A Pattern Approach to Interaction Design*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[33] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, November 2005.

[34] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29, 2002.

[35] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211, 2010.

[36] Patrick G. Bridges, Mark Hoemmen, Kurt B. Ferreira, Michael A. Heroux, Philip Soltero, and Ron Brightwell. Cooperative application/OS DRAM fault recovery. In *4th Workshop on Resiliency in High Performance Computing in Clusters, Clouds, and Grids*, Bordeaux, France, September 2011.

[37] Ron Brightwell, Ron Oldfield, Arthur B. Maccabe, and David E. Bernholdt. Hobbes: Composition and virtualization as the foundations of an extreme-scale OS/R. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '13, pages 2:1–2:8, New York, NY, USA, 2013. ACM.

[38] Greg Bronevetsky and Bronis de Supinski. Soft error vulnerability of iterative linear algebra methods. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, pages 155–164, 2008.

[39] Frank Buschmann, Kevin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture - Volume 4: A Pattern Language for Distributed Computing*. Wiley Publishing, 2007.

[40] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[41] Nicholas P. Carter, Helia Naeimi, and Donald S. Gardner. Design techniques for cross-layer resilience. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 1023–1028, 2010.

[42] Jeremy Casas, Dan Clark, Phil Galbiati, Ravi Konuru, Steve Otto, Robert Prouty, and Jonathan Walpole. MIST: PVM with Transparent Migration and Checkpointing. In *In 3rd Annual PVM Users' Group Meeting*, 1995.

[43] Marc Casas, Wilfried N. Gansterer, and Elias Wimmer. Resilient gossip-inspired all-reduce algorithms for high-performance computing: Potential, limitations, and open questions. *IJHPCA*, 33(2), 2019.

[44] Zizhong Chen. Algorithm-based recovery for iterative methods without checkpointing. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 73–84, 2011.

[45] Eric Cheng, Shahrzad Mirkhani, Lukasz G. Szafaryn, Chen-Yong Cher, Hyungmin Cho, Kevin Skadron, Mircea R. Stan, Klas Lilja, Jacob A. Abraham, Pradip Bose, and Subhasish Mitra. CLEAR: Cross-layer exploration for architecting resilience - combining hardware and software techniques to tolerate soft errors in processor cores. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 68:1–68:6, 2016.

[46] Chen-Yong Cher, Meeta S. Gupta, Pradip Bose, and K. Paul Muller. Understanding soft error resiliency of Blue Gene/Q compute chip through hardware proton irradiation and software fault injection. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 587–596, 2014.

[47] A Chien, P Balaji, N Dun, A Fang, H Fujita, K Iskra, Z Rubenstein, Z Zheng, J Hammond, I Laguna, D Richards, A Dubey, B van Straalen, M Hoemmen, M Heroux, K Teranishi, and A Siegel. Exploring versioned distributed arrays for resilience in scientific applications: global view resilience. *The International Journal of High Performance Computing Applications*, 0(0):1094342016664796, 0.

[48] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. Containment domains: a scalable, efficient, and flexible resilience scheme for exascale systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 58:1–58:11, 2012.

[49] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[50] Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, March 2004.

[51] John Daly, Bill Harrod, Thuc Hoang, Lucy Nowell, Bob Adolf, Shekhar Borkar, Nathan DeBardeleben, Mootaz Elnozahy, Mike Heroux, David Rogers, Rob Ross, Vivek Sarkar, Martin Schulz, Mark Snir, Paul Woodward, Rob Aulwes, Marti Bancroft, Greg Bronevetsky, Bill Carlson, Al Geist, Mary Hall, Jeff Hollingsworth, Bob Lucas, Andrew Lumsdaine, Tina Macaluso, Dan Quinlan, Sonia Sachs, John Shalf, Tom Smith, Jon Stearley, Bert Still, and John Wu. Inter-agency workshop on HPC resilience at extreme scale. February 2012.

[52] John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, 2006.

[53] John T. Daly, Lori A. Pritchett-Sheats, and Sarah E. Michalak. Application MTTFE vs. platform MTTF: A fresh perspective on system reliability and application throughput for computations at scale. In *Proceedings of the 8th IEEE International Symposium on Cluster Computing and the Grid (CCGrid) 2008: Workshop on Resiliency in High Performance Computing (Resilience)*. IEEE Computer Society, May 2008.

[54] Intel Corporation Data Center Group. Intel xeon processor E7 family: Reliability, availability, and serviceability. 2011.

[55] Teresa Davies, Christer Karlsson, Hui Liu, Chong Ding, and Zizhong Chen. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 162–171, 2011.

[56] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 497–508, 2010.

[57] Marc A. de Kruijf, Karthikeyan Sankaralingam, and Somesh Jha. Static analysis and compiler design for idempotent processing. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 475–486, 2012.

[58] Nathan DeBardeleben, James Laros, John T Daly, Stephen L Scott, Christian Engelmann, and Bill Harrod. High-end computing resilience: Analysis of issues facing the HEC community and path-forward for research and development. *Whitepaper*, December 2009.

[59] Timothy J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. IBM Microelectronics Division, 1997.

[60] T.J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. Technical report, IBM Microelectronics Division Whitepaper, November 1997.

[61] J. Dinan, A. Singri, P. Sadayappan, and S. Krishnamoorthy. Selective recovery from failures in a task parallel programming model. In *IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 709–714, 2010.

[62] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfy Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E Papka, Dan Reed, Mitsuhisa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. The International Exascale Software Project Roadmap. *International Journal on High Performance Computing Applications*, pages 3–60, February 2011.

[63] Chad Dougherty, Kirk Sayre, Robert Seacord, David Svoboda, and Kazuya Togashi. Secure design patterns. Technical Report CMU/SEI-2009-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2009.

[64] R. G. Dreslinski, M. Wieckowski, D. Blaauw, D. Sylvester, and T. Mudge. Near-threshold computing: Reclaiming moore's law through energy efficient integrated circuits. *Proceedings of the IEEE*, 98(2):253–266, February 2010.

[65] J. Duell, P. Hargrove, and E. Roman. The design and implementation of berkeley lab's linux checkpoint/restart. Technical report, Lawrence Berkeley National Lab (LBNL), December 2002.

[66] Douglas K. Van Duyne, James Landay, and Jason I. Hong. *The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[67] James Elliott, Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Combining partial redundancy and checkpointing for HPC. In *Proceedings of the* 32^{nd} *International Conference on Distributed Computing Systems (ICDCS) 2012*, pages 615–626, Macau, SAR, China, June 18-21, 2012. IEEE Computer Society, Los Alamitos, CA, USA.

[68] E.N.Elnozahy, Ricardo Bianchini, Tarek El-Ghazawi, Armando Fox, Forest Godfrey, Adolfy Hoisie, Kathryn McKinley, Rami Melhem, James Plank, Partha Ranganathan, and Josh Simons. System Resilience at Extreme Scale. Technical report, DARPA, 2008.

[69] Christian Engelmann. *Symmetric Active/Active High Availability for High-Performance Computing System Services*. PhD thesis, Department of Computer Science, University of Reading, UK, December 8, 2008.

[70] Christian Engelmann and Swen Böhm. Redundant execution of HPC applications with MR-MPI. In *Proceedings of the* 10^{th} *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2011*, pages 31–38, Innsbruck, Austria, February 15-17, 2011. ACTA Press, Calgary, AB, Canada.

[71] Christian Engelmann, Hong H. Ong, and Stephen L. Scott. The case for modular redundancy in large-scale high performance computing systems. In *Proceedings of the* 8^{th} *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2009*, pages 189–194, Innsbruck, Austria, February 16-18, 2009. ACTA Press, Calgary, AB, Canada.

[72] Christian Engelmann, Geoffroy R. Vallée, Thomas Naughton, and Stephen L. Scott. Proactive fault tolerance using preemptive migration. In *Proceedings of the* 17^{th} *Euromicro International Conference on Parallel, Distributed, and network-based Processing (PDP) 2009*, pages 252–257, Weimar, Germany, February 18-20, 2009. IEEE Computer Society, Los Alamitos, CA, USA.

[73] Irene Eusgeld, Felix C. Freiling, and Ralf Reussner, editors. *Dependability Metrics: Advanced Lectures [Result from a Dagstuhl seminar, October 30 - November 1, 2005]*, volume 4909 of *Lecture Notes in Computer Science*. Springer, 2008.

[74] Graham E. Fagg and Jack Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, 2000.

[75] Massimiliano Fasi, Yves Robert, and Bora Uçar. Combining algorithm-based fault tolerance and checkpointing for iterative solvers. Research Report RR-8675, INRIA Grenoble - Rhône-Alpes ; INRIA, January 2015.

[76] Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, and Ron Brightwell. rMPI: increasing fault resiliency in a message-passing environment. Technical report, Sandia National Laboratories, Technical Report SAND2011-2488, 2011.

[77] Kurt Ferreira, Jon Stearley, James H. Laros, III, and et al. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[78] David Fiala, Frank Mueller, Christian Engelmann, Kurt Ferreira, Ron Brightwell, and Rolf Riesen. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the* 25^{th} *IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2012*, pages 78:1–78:12, Salt Lake City, UT, USA, November 10-16, 2012. ACM Press, New York, NY, USA.

[79] David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Mini-Ckpts: Surviving OS failures in persistent memory. In *Proceedings of the* 30^{th} *ACM International Conference on Supercomputing (ICS) 2016*, pages 7:1–7:14, Istanbul, Turkey, June 1-3, 2016. ACM Press, New York, NY, USA.

[80] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[81] Uwe Friedrichsen. No crash allowed - patterns for fault tolerance. In *The Conference for Java and Software Innovation (JAX)*, October 2012.

[82] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[83] Wilfried N. Gansterer, Gerhard Niederbrucker, Hana Straková, and Stefan Schulze Grotthoff. Robust distributed orthogonalization based on randomized aggregation. In Vassil N. Alexandrov, Al Geist,

and Jack J. Dongarra, editors, *Proceedings of the second workshop on Scalable algorithms for large-scale systems, ScalA@SC 2011, Seattle, WA, USA, November 14, 2011*, pages 7–10. ACM, 2011.

[84] Wilfried N. Gansterer, Gerhard Niederbrucker, Hana Straková, and Stefan Schulze Grotthoff. Scalable and fault tolerant orthogonalization based on randomized distributed data aggregation. *J. Comput. Sci.*, 4(6):480–488, 2013.

[85] Christian Glusa, Paritosh Ramanan, Erik G. Boman, Edmond Chow, and Sivasankaran Rajamanickam. Asynchronous one-level and two-level domain decomposition solvers. *CoRR*, abs/1808.08172, 2018.

[86] Michael Griebel and Peter Oswald. Stochastic subspace correction methods and fault tolerance. *Mathematics of Computation*, 89(321):279–312, 2020.

[87] Object Management Group. Common object request broker architecture (CORBA) specification, version 3.3. http://www.omg.org/spec/CORBA/3.3/, 2012.

[88] Rinku Gupta, Pete Beckman, Byung-Hoon Park, Ewing Lusk, Paul Hargrove, Al Geist, Dhabaleswar Panda, Andrew Lumsdaine, and Jack Dongarra. CIFTS: A coordinated infrastructure for fault-tolerant systems. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 237–245. IEEE Computer Society, 2009.

[89] D. Hakkarinen and Zizhong Chen. Algorithmic cholesky factorization fault recovery. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, 2010.

[90] Robert Hanmer. *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007.

[91] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, alan gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan Kim. The IBM Blue Gene/Q compute chip. *IEEE Micro*, 32(2):48–60, March 2012.

[92] Xubin (Ben) He, Li Ou, Christian Engelmann, Xin Chen, and Stephen L. Scott. Symmetric active/active metadata service for high availability parallel file systems. *Journal of Parallel and Distributed Computing (JPDC)*, 69(12):961–973, December 2009.

[93] Jeffrey Heer and Maneesh Agrawala. Software design patterns for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):853–860, September 2006.

[94] Daniel Henderson. POWER8 processor-based systems RAS. August 2016.

[95] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518–528, June 1984.

[96] Kuang-Hua Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, C-33(6):518 –528, june 1984.

[97] Y. Huang, C. M. Kintala, L. Bernstein, and Y. Wang. Components for software fault tolerance and rejuvenation. *AT&T Technical Journal*, pages 29–37, 1996.

[98] Saurabh Hukerikar and Robert F. Lucas. Rolex: Resilience-oriented language extensions for extreme-scale systems. *The Journal of Supercomputing*, pages 1–33, 2016.

[99] Saurabh Hukerikar, Keita Teranishi, Pedro C. Diniz, and Robert F. Lucas. Redthreads: An interface for application-level fault detection/correction through adaptive redundant multithreading. *International Journal of Parallel Programming*, pages 1–27, 2016.

[100] J. Hursey, J.M. Squyres, T.I. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for open MPI. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2007.

[101] Cray Inc. Cray XT6 computing platform. http://www.cray.com/Assets/PDF/products/xt/CrayXT6Brochure.pdf, 2009.

[102] Cray Inc. Cray XE6 computing platform. http://www.cray.com/sites/default/files/resources/CrayXE6Brochure.pdf, 2010.

[103] Cray Inc. Cray XC40 computing platform. http://www.cray.com/Assets/PDF/products/xc/CrayXC40Brochure.pdf, 2014.

[104] Tanzima Zerin Islam, Kathryn Mohror, Saurabh Bagchi, Adam Moody, Bronis R. de Supinski, and Rudolf Eigenmann. McrEngine: a scalable checkpointing system using data-aware aggregation and compression. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 17:1–17:11, 2012.

[105] David Jauk, Dai Yang, and Martin Schulz. Predicting faults in high performance computing systems: An in-depth survey of the state-of-the-practice. In *Proceedings of the 32nd IEEE/ACM International Conference on High Performance Computing, Networking, Storage and Analysis (SC) 2019*, Denver, CO, USA, November 17-22, 2019. ACM Press, New York, NY, USA.

[106] Haewon Jeong, Yaoqing Yang, Christian Engelmann, Vipul Gupta, Tze Meng Low, Pulkit Grover, Viveck Cadambe, and Kannan Ramchandran. 3D coded SUMMA: Communication-efficient and robust parallel matrix multiplication. In *Lecture Notes in Computer Science: Proceedings of the 26th European Conference on Parallel and Distributed Computing (Euro-Par) 2020*, volume 12247, pages 392–407, Warsaw, Poland, August 24-28, 2020. Springer Verlag, Berlin, Germany.

[107] Jing-Yang Jou and Jacob A. Abraham. Fault-tolerant matrix operations on multiple processor systems using weighted checksums. pages 94–101, 1984.

[108] K. Keutzer and T. Mattson. Our pattern language (OPL): A design pattern language for engineering (parallel) software. In *ParaPLoP Workshop on Parallel Programming Patterns*, 2009.

[109] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. John Wiley & Sons, Inc., New York, NY, USA, 2004.

[110] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dallya, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. Exascale Computing Study: Technology Challenges in Achieving Exascale systems. Technical report, DARPA, September 2008.

[111] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, Burlington, MA, USA, July 2007.

[112] Gary Lakner and Brant Knudson. IBM system Blue Gene solution: Blue Gene/Q system administration. *IBM Redbooks*, May 2013.

[113] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra. Recovery patterns for iterative methods in a parallel unstable environment. *SIAM Journal Scientific Computing*, 30:102–116, November 2007.

[114] Juan Leon, Allan L. Fisher, and Peter Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical report, 1993.

[115] Wei-Jih Li and Jyh-Jong Tsay. Checkpointing Message-Passing Interface (MPI) Parallel Programs. In *Proceedings of Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 147–152, 1997.

[116] Yinglung Liang, Yanyong Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo. Blue Gene/L failure analysis and prediction models. In *Proceedings of the* 36^{th} *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2006*, pages 425–434. IEEE Computer Society, Los Alamitos, CA, USA, June 2006.

[117] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra Sahoo. Bluegene/l failure analysis and prediction models. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 425–434. IEEE, 2006.

[118] J. Lidman, D.J. Quinlan, C. Liao, and S.A McKee. ROSE::FTTransform - a Source-to-Source Translation Framework for Exascale Fault-tolerance Research. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6, June 2012.

[119] Kshitij Limaye, Chokchai Leangsuksun, Zeno Greenwood, Stephen L. Scott, Christian Engelmann, Richard M. Libby, and Kasidit Chanchio. Job-site level fault tolerance for cluster and grid environments. In *Proceedings of the* 7^{th} *IEEE International Conference on Cluster Computing (Cluster) 2005*, pages 1–9, Boston, MA, USA, September 26-30, 2005.

[120] Michael Litzkow and Miron Livny. Supporting checkpointing and process migration outside the UNIX kernel. In *Proceedings of the Winter 1992 USENIX Conference*, pages 283–290, San Francisco, CA, January 1992.

[121] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical report, University of Wisconsin, April 1997.

[122] Hatem Ltaief, Edgar Gabriel, and Marc Garbey. Fault tolerant algorithms for heat transfer problems. *Journal of Parallel and Distributed Computing (JPDC)*, 68(5):663–677, 2008.

[123] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[124] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[125] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[126] Michael D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Parallelism*, HotPar'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[127] Dennis McEvoy. The architecture of tandem's nonstop system. In *Proceedings of the ACM '81 conference*, New York, NY, USA, 1981. ACM.

[128] A. Mishra and P. Banerjee. An algorithm-based error detection scheme for the multigrid method. *IEEE Transactions on Computers*, 52(9):1089–1099, 2003.

[129] S. Mitra, K. Brelsford, and P. N. Sanda. Cross-layer resilience challenges: Metrics and optimization. In *2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010)*, pages 1029–1034, March 2010.

[130] Kathryn Mohror, Adam Moody, Greg Bronevetsky, and Bronis R. de Supinski. Detailed modeling and evaluation of a scalable multilevel checkpointing system. *IEEE Transactions on Parallel and Distributed Systems*, 99:1, 2013.

[131] Todd K Moon. Error correction coding: Mathematical methods and algorithms. 2005.

[132] K. Morris, F. Rizzi, B. Cook, P. Mycek, O. LeMaitre, O. M. Knio, K. Sargsyan, K. Dahlgren, and B. J. Debusschere. Performance scaling variability and energy analysis for a resilient ulfm-based pde solver. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 41–48, Nov 2016.

[133] Arun B. Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *Proceedings of the* 21^{st} *ACM International Conference on Supercomputing (ICS) 2007*, pages 23–32, Seattle, WA, USA, June 16-20, 2007. ACM Press, New York, NY, USA.

[134] Arun B. Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *Proceedings of the* 21^{st} *ACM International Conference on Supercomputing (ICS) 2007*, pages 23–32, Seattle, WA, USA, June 16-20, 2007. ACM Press, New York, NY, USA.

[135] R. Naseer and J. Draper. Parallel double error correcting code design to mitigate multi-bit upsets in SRAMs. In *34th European Solid-State Circuits Conference*, pages 222–225, 2008.

[136] B. Natarajan, A. Gokhale, S. Yajnik, and D. C. Schmidt. DOORS: towards high-performance fault tolerant CORBA. In *Proceedings of the International Symposium on Distributed Objects and Applications*, pages 39–48, 2000.

[137] Xiang Ni, Esteban Meneses, Nikhil Jain, and Laxmikant V. Kalé. Acr: Automatic checkpoint/restart for soft and hard error protection. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 7:1–7:12, 2013.

[138] Bin Nie, Ji Xue, Saurabh Gupta, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities. In *Proceedings of the* 25^{th} *IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS) 2017*, pages 22–31, Banff, AB, Canada, September 20-22, 2017. IEEE Computer Society, Los Alamitos, CA, USA.

[139] Bin Nie, Ji Xue, Saurabh Gupta, Tirthak Patel, Christian Engelmann, Evgenia Smirni, and Devesh Tiwari. Machine learning models for GPU error prediction in a large scale HPC system. In *Proceedings of the* 48^{th} *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN) 2018*, pages 95–106, Luxembourg City, Luxembourg, June 25-28, 2018. IEEE Computer Society, Los Alamitos, CA, USA.

[140] Gerhard Niederbrucker and Wilfried N. Gansterer. Robust gossip-based aggregation: A practical point of view. In Peter Sanders and Norbert Zeh, editors, *Proceedings of the 15th Meeting on Algorithm Engineering and Experiments, ALENEX 2013, New Orleans, Louisiana, USA, January 7, 2013*, pages 133–147. SIAM, 2013.

[141] Nvidia. Dynamic page retirement, reference guide vr352. https://docs.nvidia.com/deploy/pdf/Dynamic_Page_Retirement.pdf, 2015.

[142] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error Detection by Duplicated Instructions in Super-scalar Processors. *IEEE Transactions on Reliability*, pages 63–75, 2002.

[143] Adam J. Oliner, Larry Rudolph, and Ramendra K Sahoo. Cooperative checkpointing: a robust approach to large-scale systems reliability. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 14–23, 2006.

[144] Swann Perarnau, Rinku Gupta, and Pete Beckman. Argo: An exascale operating system and runtime [extended abstract]. In *The International Conference for High Performance Computing, Networking, Storage and Analysis*, SC15, 2015.

[145] Hoang Pham. *System Software Reliability*. Springer Series in Reliability Engineering. Springer, 2007.

[146] Philip H. Carns. PVFS2 high-availability clustering using Heartbeat 2.0, 2007.

[147] Pedro Pla. Drbd in a heartbeat. *Linux Journal (LJ)*, September 2006.

[148] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[149] J.S. Plank, Youngbae Kim, and J.J. Dongarra. Algorithm-based diskless checkpointing for fault tolerant matrix operations. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, pages 351–360, 1995.

[150] J.S. Plank, K. Li, and M.A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[151] Karl E. Prikopa and Wilfried N. Gansterer. Fault-tolerant least squares solvers for wireless sensor networks based on gossiping. *J. Parallel Distributed Comput.*, 136:52–62, 2020.

[152] A.L.N. Reddy and P. Banerjee. Algorithm-based fault detection for signal processing applications. *IEEE Transactions on Computers*, 39(10):1304–1308, 1990.

[153] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization, 2005*, pages 243–254, 2005.

[154] Philipp Reisner and Lars Ellenberg. Drbd v8 – Replicated storage with shared disk semantics. In *Proceedings of the* 12^{th} *International Linux System Technology Conference (Linux-Kongress) 2005*, Hamburg, Germany, October 11-14, 2005.

[155] J. Rexford and N.K. Jha. Algorithm-based fault tolerance for floating-point operations in massively parallel systems. In *Proceedings of IEEE International Symposium on Circuits and Systems*, volume 2, pages 649–652 vol.2, 1992.

[156] F. Rizzi, K. Morris, K. Sargsyan, P. Mycek, C. Safta, O. Le LeMaitre, O.M. Knio, and B.J. Debusschere. Exploring the interplay of resilience and energy consumption for a task-based partial

differential equations preconditioner. *Parallel Computing*, 73:16 – 27, 2018. Parallel Programming for Resilience and Energy Efficiency.

[157] Francesco Rizzi, Karla Morris, Khachik Sargsyan, Paul Mycek, Cosmin Safta, Bert Debusschere, Olivier LeMaitre, and Omar Knio. Ulfm-mpi implementation of a resilient task-based partial differential equations preconditioner. In *Proceedings of the ACM Workshop on Fault-Tolerance for HPC at Extreme Scale*, New York, NY, USA, 2016. Association for Computing Machinery.

[158] A. Roy-Chowdhury and P. Banerjee. Algorithm-based fault location and recovery for matrix computations on multiprocessor systems. *IEEE Transactions on Computers*, 45(11):1239–1247, 1996.

[159] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings of the* 9^{th} *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD) 2003*, pages 426–435, Washington, DC, USA, 2003. ACM Press, New York, NY, USA.

[160] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In *In Proceedings of LACSI Symposium, Sante Fe*, pages 479–493, 2003.

[161] Piyush Sao, Christian Engelmann, Srinivas Eswar, Oded Green, and Richard Vuduc. Self-stabilizing connected components. In *Proceedings of the* 32^{nd} *International Conference on High Performance Computing, Networking, Storage and Analysis (SC) Workshops 2019:* 9^{th} *Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS) 2019*, pages 50–59, Denver, CO, USA, November 22, 2019. IEEE Computer Society, Los Alamitos, CA, USA.

[162] Titos Saridakis. A system of patterns for fault tolerance. In *Proceedings of 2002 European Conference on Pattern Languages of Programs (EuroPLoP)*, 2002.

[163] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2000.

[164] J. Shalf, D. Quinlan, and C. Janssen. Rethinking hardware-software codesign for exascale systems. *Computer*, 44(11):22–30, November 2011.

[165] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, pages 1–25. Springer-Verlag, 2011.

[166] A. Shye, J. Blomstedt, T. Moseley, V.J. Reddi, and D.A. Connors. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, pages 135–148, 2009.

[167] T.J. Slegel, III Averill, R.M., M.A. Check, and et. al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, pages 12–23, 1999.

[168] J. Sloan, R. Kumar, and G. Bronevetsky. Algorithmic approaches to low overhead fault detection for sparse linear algebra. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12, 2012.

[169] J. Sloan, R. Kumar, and G. Bronevetsky. An algorithmic approach to error localization and partial recomputation for low-overhead fault tolerance. In *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013.

[170] Marc Snir, Robert W. Wisniewski, Jacob A. Abraham, Sarita V. Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A. Chien, Paul Coteus, Nathan A. Debardeleben, Pedro Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, and Eric Van Hensbergen. Addressing failures in exascale computing. *International Journal of High Performance Computing Applications (IJHPCA)*, 28(2):127–171, May 2014.

[171] M. J. Sottile and R. G. Minnich. Supermon: a high-speed cluster monitoring system. In *Proceedings. IEEE International Conference on Cluster Computing*, pages 39–46, 2002.

[172] Georg Stellner. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the* 10^{th} *International Conference on Parallel Processing (IPPS) 1996*, pages 526–531, Honolulu, HI, USA, April 15-19, 1996. IEEE Computer Society, Los Alamitos, CA, USA.

[173] Curtis E. Stevens. AT attachment 8 - ATA/ATAPI command set (ATA8-ACS). http://www.t13.org/documents/uploadeddocuments/docs2008/d1699r6a-ata8-acs.pdf, 2008.

[174] Hana Straková, Gerhard Niederbrucker, and Wilfried N. Gansterer. Fault tolerance properties of gossip-based distributed orthogonal iteration methods. In Vassil N. Alexandrov, Michael Lees, Valeria V. Krzhizhanovskaya, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013*, volume 18 of *Procedia Computer Science*, pages 189–198. Elsevier, 2013.

[175] Sun Microsystems, Inc, Santa Clara, CA, USA. Sun Grid Engine (SGE) documentation, 2007.

[176] Jerry Talton, Lingfeng Yang, Ranjitha Kumar, Maxine Lim, Noah Goodman, and Radomír Měch. Learning design patterns with bayesian grammar induction. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology*, UIST '12, pages 63–74, New York, NY, USA, 2012. ACM.

[177] Keita Teranishi and Michael A. Heroux. Toward local failure local recovery resilience model using mpi-ulfm. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, page 51–56, New York, NY, USA, 2014. Association for Computing Machinery.

[178] Kai Uhlemann, Christian Engelmann, and Stephen L. Scott. JOSHUA: Symmetric active/active replication for highly available HPC job and resource management. In *Proceedings of the* 8^{th} *IEEE International Conference on Cluster Computing (Cluster) 2006*, pages 1–10, Barcelona, Spain, September 25-28, 2006. IEEE Computer Society, Los Alamitos, CA, USA.

[179] Nitin H. Vaidya. A case for two-level distributed recovery schemes. In *Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 64–73, 1995.

[180] Hubertus J. J. van Dam, Abhinav Vishnu, and Wibe A. de Jong. A case for soft error detection and correction in computational chemistry. *Journal of Chemical Theory and Computation*, 9(9):3995–4005, 2013.

[181] Enrique Vargas. High availability fundamentals. *Sun Blueprints*, November 2000.

[182] Marco Vassura, Luciano Margara, Pietro Di Lena, Filippo Medri, Piero Fariselli, and Rita Casadio. FT-COMAR: Fault tolerant three-dimensional structure reconstruction from protein contact maps. *Bioinformatics*, 24(10):1313–1315, 2008.

[183] Jeffrey S. Vetter. *Contemporary High Performance Computing: From Petascale Toward Exascale*. Chapman & Hall/CRC, 2013.

[184] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. In *Proceedings of the* 21^{st} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2007*, pages 1–10, Long Beach, CA, USA, March 26-30, 2007. ACM Press, New York, NY, USA.

[185] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Hybrid checkpointing for MPI jobs in HPC environments. In *Proceedings of the* 16^{th} *IEEE International Conference on Parallel and Distributed Systems (ICPADS) 2010*, pages 524–533, Shanghai, China, December 8-10, 2010. IEEE Computer Society, Los Alamitos, CA, USA.

[186] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration and back migration in HPC environments. *Journal of Parallel and Distributed Computing (JPDC)*, 72(2):254–267, February 2012.

[187] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration and back migration in HPC environments. *Journal of Parallel and Distributed Computing (JPDC)*, 72(2):254–267, February 2012.

[188] Cheng Wang, H. Kim, Y. Wu, and V. Ying. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In *International Symposium on Code Generation and Optimization, 2007*, pages 244–258, 2007.

[189] Sying-Jyan Wang and N.K. Jha. Algorithm-based fault tolerance for FFT networks. In *IEEE International Symposium on Circuits and Systems*, volume 1, pages 141–144 vol.1, 1992.

[190] S. Yajnik and N.K. Jha. Synthesis of fault tolerant architectures for molecular dynamics. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 4, pages 247–250 vol.4, 1994.

[191] Gulay Yalcin, Osman Unsal, Ibrahim Hur, Adrian Cristal, and Mateo Valero. FaulTM: Fault-Tolerance Using Hardware Transactional Memory. In *Workshop on Parallel Execution of Sequential Programs on Multi-core Architecture*, Saint Malo, France, 2010.

[192] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux utility for resource management. In *Lecture Notes in Computer Science: Proceedings of the* 9^{th} *International Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*, volume 2862, pages 44–60, Seattle, WA, USA, June 24, 2003.

[193] L. Yu, Z. Zheng, Z. Lan, and S. Coghlan. Practical online failure prediction for Blue Gene/P: Period-based vs event-driven. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 259–264, June 2011.

[194] Weikuan Yu, Ranjit Noronha, Shuang Liang, and Dhabaleswar K. Panda. Benefits of high speed interconnects to cluster file systems: A case study with Lustre. In *Proceedings of the* 20^{st} *IEEE International Parallel and Distributed Processing Symposium (IPDPS) 2006*, pages 8–15, Rhodes Island, Greece, April 25-29, 2006.

[195] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. DAFT: Decoupled Acyclic Fault Tolerance. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 87–98, 2010.