ORNL/TM-2017/745

Resilience Design Patterns

A Structured Approach to Resilience at Extreme Scale - version 1.2

Saurabh Hukerikar
Christian Engelmann

August 2017

Approved for public release.
Distribution is unlimited.

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

ORNL/TM-2017/745

Computer Science and Mathematics Division

Resilience Design Patterns
A Structured Approach to Resilience at Extreme Scale
version 1.2

Saurabh Hukerikar
Christian Engelmann

Date Published: August 2017

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

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 this 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.

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 [53].

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 [136]. 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. [25]. 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 the near-threshold voltage (NTV) operation [55] (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.

- Sections 11., 12., 13. present 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. Resilience Terminology and Concepts

The terminology is largely based on prior work that established the definitions for HPC Reliability, Availability & Serviceability (RAS) [142, 98, 150, 63, 65, 124].

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 the 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, these terms are associated with distinct formal concepts, which are defined as follows [18]:

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 line 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 SDC, when the read value is used as an index in a memory address calculation. A *detected correctable error* 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) [115]. 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 silent data corruption (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.

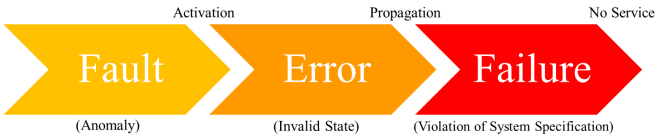

Figure 1. Relationship between fault, error and failure

- **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 [142]. 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 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 entails *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*.

In practice, a resilience mechanism may merge the implementation of two or even all three of the capabilities to deliver a complete solution.

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 $0 \leq t$, $F(t)$.

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

- **PDF:** The relative likelihood of an error or failure, $f(t)$.
- **Error** or **failure rate:** The frequency of errors or failures in a system, $\lambda(t)$.

$$\lambda(t) = \frac{f(t)}{R(t)} \tag{3}$$

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

$$MTTE\ or\ MTTF = \int_0^\infty R(t)dt \tag{4}$$

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

$$FIT = \frac{10^9}{MTTF} \tag{5}$$

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

$$R(n,t)_s = \prod_{i=1}^{n} R_i(t) \tag{6}$$

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

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

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

$$R(n,t)_{is} = R(t)^n \tag{8}$$

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

$$R(n,t)_{ip} = 1 - (1 - R(t))^n \tag{9}$$

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.

$$A = \frac{t_{pu}}{t_{pu} + t_{ud} + t_{sd}} \tag{10}$$
$$= \frac{MTTF}{MTTF + MTTR} \tag{11}$$
$$= \frac{MTTF}{MTBF} \tag{12}$$
$$\tag{13}$$

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

$$MTBF = MTTF + MTTR \tag{14}$$

- **Serial availability:** The availability of a system with n dependent components, A_s.

$$A_s = \prod_{i=1}^{n} A_i \tag{15}$$

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

$$A_p = 1 - \prod_{i=1}^{n} (1 - A_i) \tag{16}$$

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

$$A_{is} = A^n \tag{17}$$

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

$$A_{ip} = 1 - (1 - A)^n \tag{18}$$

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 unscheduled 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.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives} = \frac{True\ Positives}{Indicated\ Errors\ or\ Failures} \tag{19}$$

$$= 1 - \frac{False\ Positives}{True\ Positives + False\ Positives} = 1 - \frac{False\ Positives}{Indicated\ Errors\ or\ Failures} \tag{20}$$

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives} = \frac{True\ Positives}{Errors\ or\ Failures} \tag{21}$$

$$= 1 - \frac{False\ Negatives}{True\ Positives + False\ Negatives} = 1 - \frac{False\ Negatives}{Errors\ or\ Failures} \tag{22}$$

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

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) [142]. Both perspectives are quite different [44]. 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 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 [15]. 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 [43]. 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 [96]. 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 [137]. 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 [49]. 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 [43]. 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 checkpoint 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 single bit-error correction and double bit-error detection (SECDED) ECC for main memory, caches, registers and architectural state is typical in most modern HPC installations. Error correction codes (ECC) use 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 [90].

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 [88]. 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 [99]. The RAS log information may be used for post-hoc analysis [103] to understand system's resilience behavior, or even in online analysis to discover trends in the failure events [158].

The Cray XE6 architecture was designed to tolerate the failure of compute nodes or services nodes [89]. The Cray Gemini High Speed Network (HSN) is designed to dynamically route around the failed nodes [152]. 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 [14] also contains similar features.

In the IBM Blue Gene/Q (BG/Q), many mechanisms are deployed to protect the chip against errors. The compute node of a Blue Gene/Q contains an 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 [79]. 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 [37].

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 [81]. 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 [45].

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 [111] and the HP NonStop [22] that use two redundant processors running in locked step. The IBM G5 [139] 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-bit error correction and triple-bit error detection (DECTED) [117] for multi-bit detection and correction. Chipkill [51] is a stronger memory protection scheme that is widely used in production HPC systems. The scheme accommodates single 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 and 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 [105] was developed to provide checkpointing for UNIX processes, while the Berkeley Labs C/R library [57] was developed as an extension to the Linux OS. The libckpt [125] 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 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 [11] [34] [101].

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 [148]. The scalable checkpoint/restart (SCR) library [114] proposes multilevel checkpointing where checkpoints are written to storage that use 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 [122]. Incremental

checkpointing dynamically identifies the changed blocks of memory since the last checkpoint through a hash function [10] 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 [91]. Plank *et al.* eliminate the overhead of writing checkpoints to disk altogether with a diskless in-memory checkpointing approach [127].

Process-level redundancy (PLR) [138] creates a set of redundant application processes whose output values are compared. The scheduling of the redundant processes is left to the operating system (OS). The RedThreads API [85] 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 [61]. Therefore, several MPI libraries have been enabled with capabilities for C/R [102]. The CoCheck MPI [144], 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 [66], Open MPI [86], MPICH-V [26] and LAM/MPI [133] 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 [69], 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 [64] [70] 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 [116] [153]. 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

SWIFT [129] 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 [159] 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 SRMT [154] 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, EDDI [121] duplicates all instructions and inserts “compare" instructions to validate the program correctness at appropriate locations in the program code. The ROSE::FTTransform [104] 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 [61]. User-level failure mitigation (ULFM) [23] 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.* [39] 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.* [48]. Their original proposal [47], 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 [157].

Complementary to approaches that focus on resiliency of computational blocks, the Global View Resilience (GVR) project [38] 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.* [28] 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 [52]. Rolex [84] 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 [83] for a range of basic matrix operations including addition, multiplication, scalar product, transposition. Such techniques were also proven effective for LU factorization [46], Cholesky factorization [77] and QR factorization [93]. 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 [130] [126] [131]. The checksum-based detection and correction methods tend to incur very high overheads to performance in sparse matrix-based applications. Sloan *et al.* [140] 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 [141] [35].

Various studies have evaluated the fault resilience of solvers of linear algebra problems [30]. 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.* [112], 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.* [128]. 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 [155].

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 [156]. 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 [149]. The fault-tolerant version of the 3D-protein reconstruction algorithm (FT-COMAR) proposed by Vassura *et al.* [151] 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 [113] 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 [33]. The CLEAR framework [36] 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 [76]. 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 [29]. The ARGO OS/R system provides publish-subscribe framework called BEACON that enables applications and system services with access to reporting and resource monitoring [123].

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, 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 [21]. 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 [67]. The ACR is a checkpoint/restart library-based solution integrated with Charm++ that accounts for failures due to hard errors as well as undetected silent data corruptions [119]. 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 [59].

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 [13].

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 [41] and further refined [42].

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 [74]. 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 [32] to capture repeatedly used methodologies in software engineering practice. Pattern systems have also been developed for cataloging concurrent and networked object-oriented environments [135], resource management [95], and distributed software systems [31].

In the pursuit of quality and scalable parallel software, patterns for programming paradigms were developed [108] as well as a pattern language, called Our Pattern Language (OPL) [94]. 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) [109]. For the design of parallel algorithms, deterministic patterns support the development of systems that automatically avoid unsafe race conditions and deadlock [110].

Design patterns have been in a variety of other domains for codifying the best-known solutions to common problems, including natural language processing [147], user interface design [24], web design [58], visualization [80], and software security [54]. Patterns have also been defined for enterprise applications that involve data processing in support or automation of business processes [72] 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 [17]. 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 [78], distributed systems [134] and enterprise data processing systems [73]. 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) [75] defines a set of standard services and protocols defined by distributed object computing middleware. The fault tolerant version of CORBA [118] 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

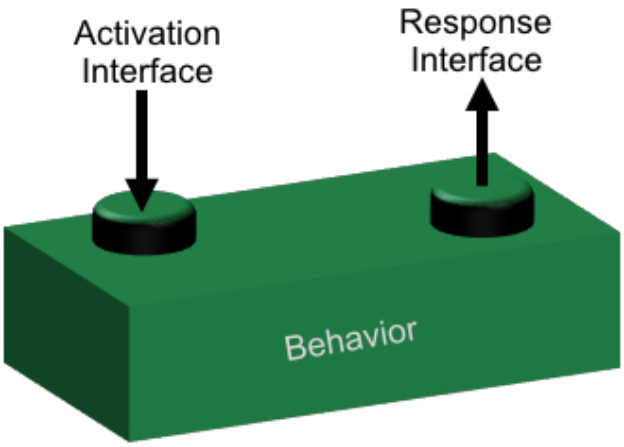

Figure 2. Anatomy of a Resilient Design Pattern

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

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 level in the hierarchy also addresses one or more of the design considerations, including the type of event

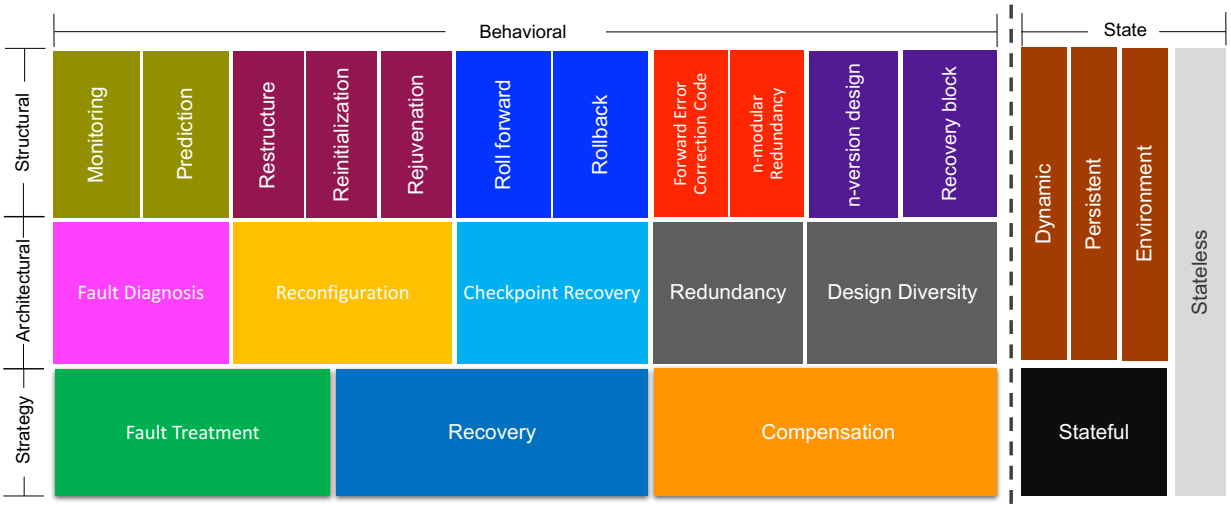

Figure 3. Classification of resilience design patterns

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.

State Patterns

The notion of state for an HPC application may be classified into three aspects [100]:

- *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.

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:

- **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.

- **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.

- **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.

Implementation 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. The Resilience Pattern Catalog

The resilience solutions presented in this specification document have been extensively studied and/or used. The primary objective here is to capture them in a standardized and accessible format. The resilience design patterns are based on practical solutions that have been 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 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 [74]:

- 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.
- 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 space and time trade-offs of applying the pattern in terms of performance and power overheads.

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.

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 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 make 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 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.

Terminology: The aim of defining a catalog of resilience design patterns is to provide 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 operating systems; 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 sub-systems or components. For a 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.

7.2 Strategy Patterns

7.2.1 Fault Treatment Pattern

Name: Fault Treatment Pattern

Problem: The presence of defects or anomalies in a system have the potential to activate, which may potentially lead to an error or a partial/complete failure of a HPC (sub-) system.

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 anomalies in the behavior of a monitored system.
- The interaction between the monitored and monitoring systems is bounded in terms of time.
- The monitoring system may optionally have the capability to modify the parameters of the monitored system, which enables the anomaly or defect to be removed before it results in an error or a failure.

Forces:

- The interactions of the monitoring and monitored systems may interfere with the operation of the monitored system. The frequency of these interactions must be minimized to reduce the overhead of the monitoring; yet, the interactions must be frequent enough to detect every anomaly in the monitored system.
- The interval for the monitoring system to gather data about the monitored system, and infer the presence of an anomaly or a defect must be rapid to prevent the activation of an error/failure. Also, this delay incurs an overhead to the operation of the monitored system.

Solution: The `Fault Treatment` pattern enables the discovery and treatment of the presence of defects or anomalies in the system that have the potential to activate, which may potentially lead to an error or a failure in the system. The pattern supports methods that attempt to recognize the presence of an anomaly or a defect within a system, and creates conditions that prevents the activation of the fault into an error or failed state. The solution requires a monitoring system, which may be a sub-system of the monitored

system or an external independent system, that observes the key parameters of the monitored system. These components are illustrated in Figure 4.

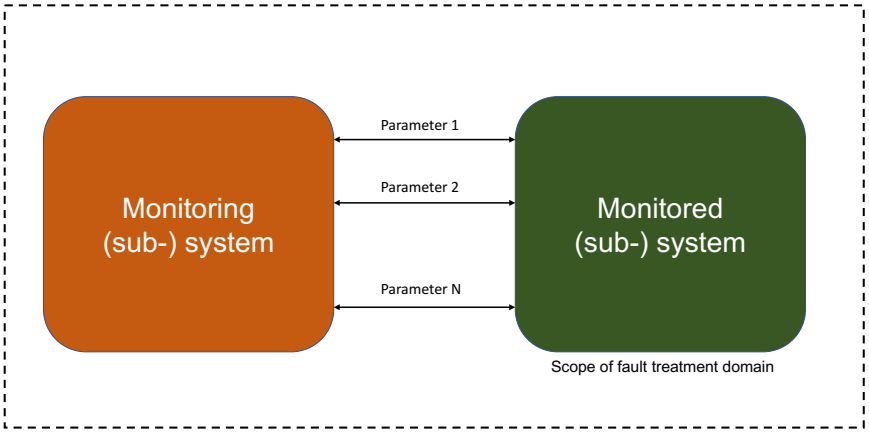

Figure 4. Fault Treatment Pattern

Capability: The pattern supports fault treatment, based on the following capabilities:

- **Fault detection**, to detect anomalies during operation before they impact the correctness of the system state, and cause errors and failures.
- **Fault mitigation**, which includes methods to enable an imminent error or failure to be prevented, or the defect to be removed.

The `Fault Treatment` pattern may support either one, or both capabilities. Figure 5 illustrates these capabilities.

Protection Domain: The protection domain extends to the scope of the monitored system, for which the monitoring system gathers data for discovering anomalies/defects. Since the pattern seeks to detect and alleviate a fault before activation, the protection domain implicitly extends to other systems that are interfaced to the monitored system.

Resulting Context: The `Fault Treatment` pattern requires the designer to identify parameters that indicate the presence of faults. The system design must include a monitoring system, which introduces design complexity in the overall system design. When the monitoring system is extrinsic to the monitored system, the design effort may be simplified, but the interfaces between the systems must be well-defined. However, when the monitoring system is intrinsic to the design of the monitored system, the complexity of the design process increases due to need to interface the sub-systems. The interference between the monitoring and monitored systems must be minimized.

Examples: Various hardware-based solutions for fault detection observe the attributes of a system, such as thermal state, timing violations to determine the presence of a defect in the behavior of the system that may potentially cause an error or failure. Similarly, software-based solutions detect the anomalies in the behavior of a system's data variables or control flow attributes to determine the presence of a fault.

Rationale: The key benefit of incorporating fault treatment patterns in a design, or deploying it during system operation is to preemptively recognize faults in the system, before they are activated and result in errors or partial/complete failures of the system. The preventive actions avoid the need for expensive recovery and/or compensation actions.

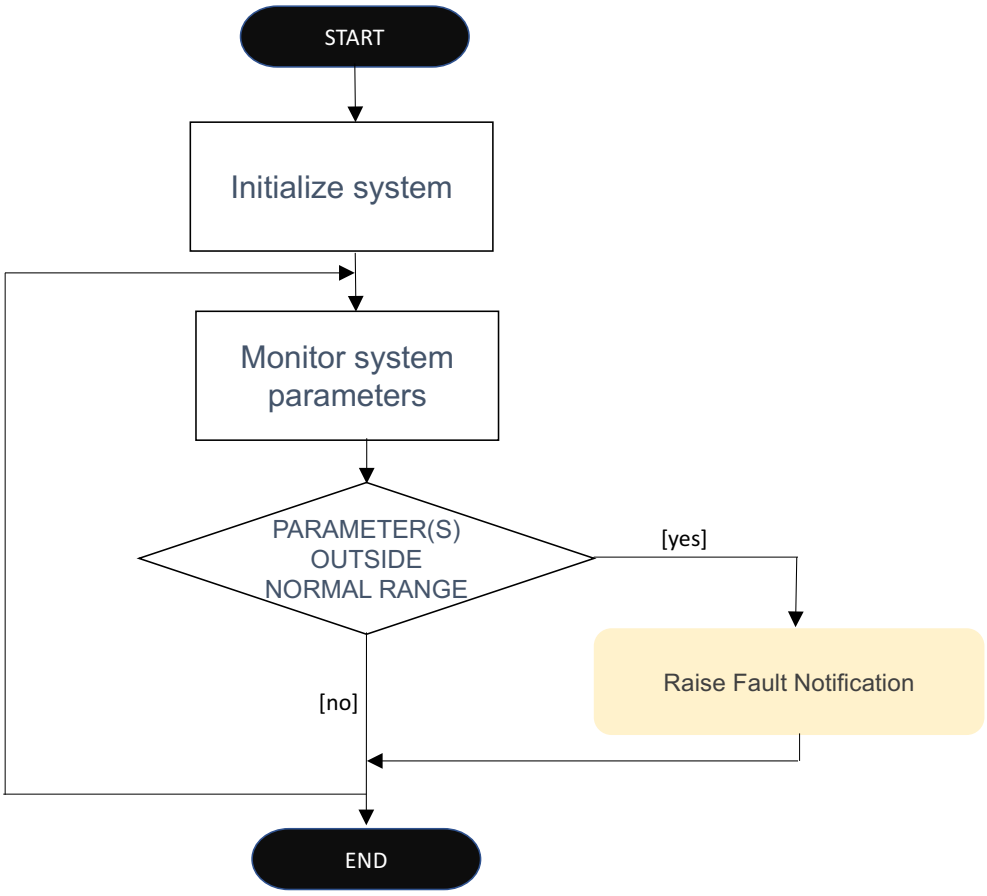

Figure 5. Fault Treatment Pattern Flowchart

Related Patterns: The recovery and compensation patterns are complementary to the fault treatment patterns. Those patterns are necessary only when the fault has been activated, and when an error or failure state exists in the system.

Known Uses:

- Processor chips used in HPC systems contain thermal sensors that detect anomalous thermal conditions in the processor cores. When the temperature reaches a preset level, the sensor trips and processor execution is halted to prevent failure of the chip. The IBM Power 8 [81] and Intel Xeon [45] series of processors include these features.
- Software-based heartbeat monitoring for liveness checking of MPI processes [19] enables detection of faults in a processor rank, before it may activate to result in failure of the MPI communicator.
- The Cray RAS and Management System (CRMS) supports real-time fault monitoring of the status of Cray system components, including the cabinets, blades, CPUs, network processors and network links in the Cray XT3/XT4/XT5 systems [20].

7.2.2 Recovery Pattern

Name: Recovery Pattern

Problem: The occurrence of errors or partial/complete failures in an HPC environment prevents applications from running correctly.

Context: The pattern applies to a system that 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 must also have the following characteristics:

- The error or failure in the system that the pattern handles must be detected; the pattern offers no implicit error/failure detection.
- The system operation has well-defined intervals that enables the pattern to transition the system to a known correct interval in response to an error/failure.
- The system can compartmentalize its state that is accurately representative of the progress of the system since initialization at the time such state is captured.

Forces:

- The pattern requires stable storage to capture system state, which increase overhead in terms of resources required by the system.
- The process of compartmentalizing and capturing system state interferes with system operation. The error/failure-free overhead penalty must be minimized.
- The amount of state captured during each creation of a recovery point incurs space and time overheads.
- The frequency of creation of system state snapshots determines overhead: more frequent creation of recovery points increases system execution time, but reduces amount of lost work upon occurrence of an error/failure.
- The post-recovery state of the system must be as close as possible to an error/failure-free system state.
- The time interval required for the recovery of a system from a stable recovery point as well the time interval to create and commit a checkpoint to a stable storage system must be less than the mean time between errors/failures in the system to enable the system to make forward progress.

Solution: The `Recovery` pattern enables the resumption of correct operation of a system impacted by the occurrence of errors in the system, or by the failure of the system. In an HPC environment, such events in the system typically result in incorrect results, and in some cases, catastrophic application crashes. The pattern supports resilient operation by restoring a system to a known correct state in the event of an error or failure. The solution suggested by the pattern, shown in Figure 6, is based on the creation of snapshots of the system state and the maintenance of these checkpoints on a stable storage system. Upon detection of an error or a failure, the checkpoints are used to recreate 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 recovery may be either forward or backward with reference to the time when the error/failure event occurred.

Capability: The `Recovery` pattern periodically preserves the essential aspects of the system state that may be subsequently used to resume operation from a known stable and correct version. The pattern handles an error or a failure by substituting an error-free state in place of the erroneous state. The pattern enables a system to tolerate errors/failures by resuming operation from a stable checkpointed version of the system

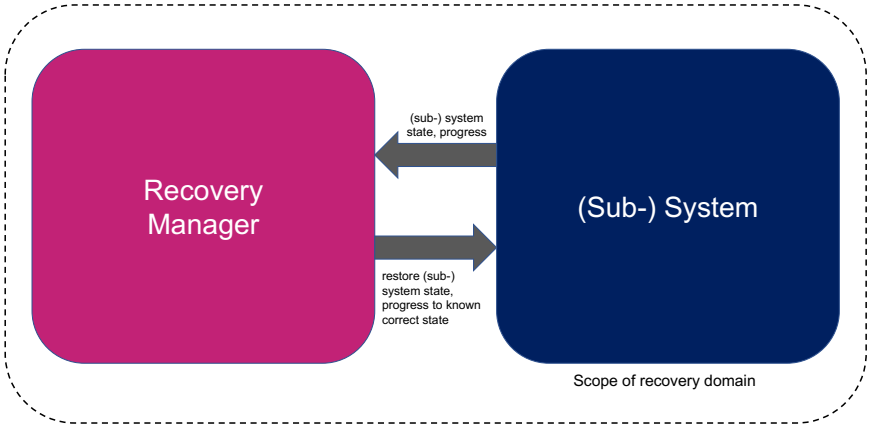

Figure 6. Recovery Pattern

that is free of the effects of the error/failure. The solution offered by this pattern is not dependent on the precise semantics of the error/failure propagation. The flowchart for this pattern is shown in Figure 7.

Protection Domain: The protection domain for a `Recovery` pattern is determined by the extent of state that is captured during checkpoint creation operation, which accurately represents the complete execution state of the system. The broader the scope of the system state that is preserved, the larger is the scope of the system state that may be protected from an error/failure event.

Resulting Context: With a `Recovery` pattern, a system is capable of tolerating failures by substitution of erroneous/failed system state to a known previous stable state (rollback recovery), or to an inferred future stable state (forward recovery). The system operation must be interrupted to restore system state from a checkpoint. The frequency of creation of checkpoints determines the overhead to system operation; frequent checkpointing incurs proportionally greater overheads during error/failure-free operation, but reduces the amount of lost work when an error/failure event does occur. The latency of saving and restoring state determine the overhead during error/failure-free operation, and the overhead of recovering from an error/failure respectively.

Examples: Various checkpoint and rollback protocols enable HPC applications and systems to capture state and commit the checkpoint files to parallel file systems. These are instances of the recovery pattern that support recovery of an application or the complete system upon detection of an error/failure [60].

Rationale: Since the solution offered by this pattern is not dependent on either the type of error/failure, or the precise semantics of the error/failure propagation, the design effort and complexity in using this pattern in any system design in low.

Related Patterns: The `Compensation` pattern is complementary to the recovery pattern, although they both seek to create conditions to recreate correct state. The key difference between the `Recovery` and the `Compensation` patterns is the method used to maintain any additional state that is used for error/failure processing. While the compensation pattern use replication of the system, a recovery pattern relies on committing error/failure-free versions of the system to stable storage.

Known Uses:

- The Berkeley Labs C/R library (BLCR) [57] is an extension to the Linux OS that supports creation checkpoint & restart capabilities for Linux processes and also provides an interface for programmers to checkpoint application program state.

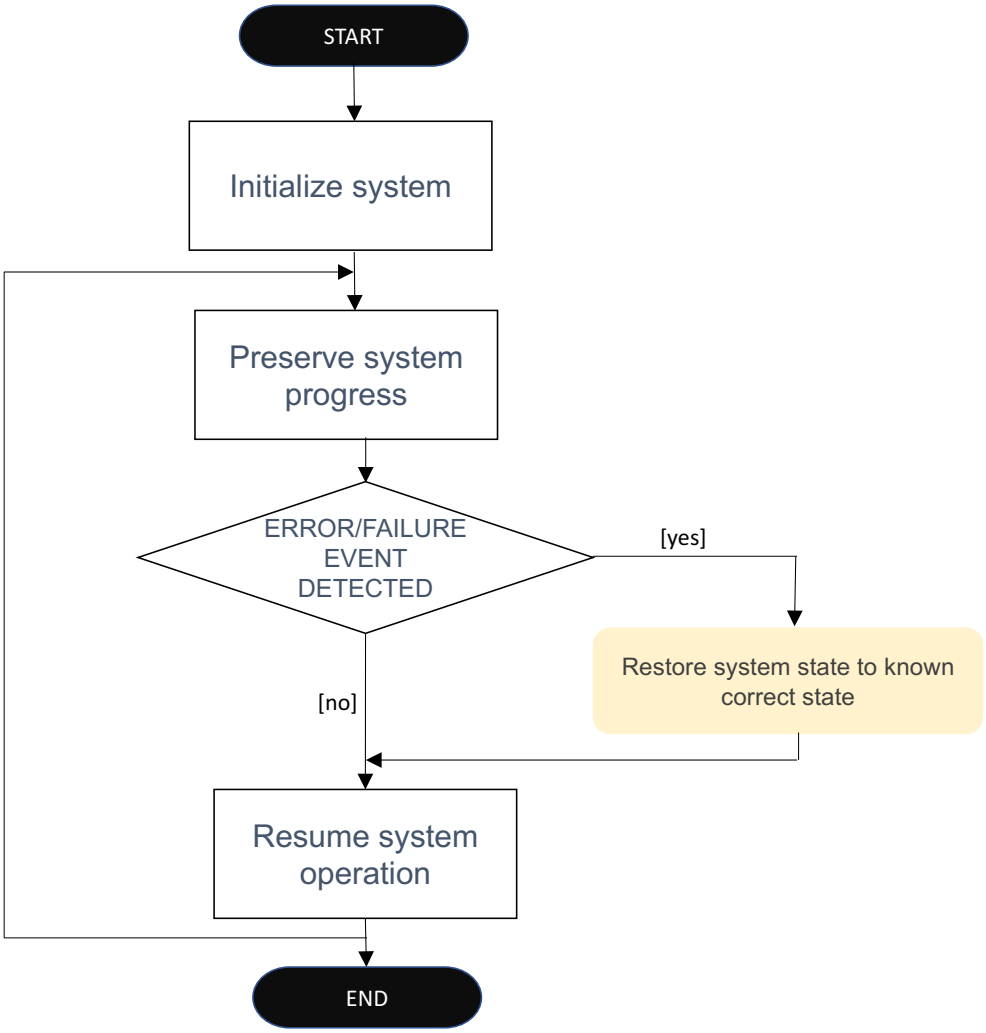

Figure 7. Recovery Pattern Flowchart

- The libckpt [125] is a user-level, library-based checkpointing solution that supports explicit directives to determine the scope of an application's checkpointed state.
- The Scalable Checkpoint/Restart (SCR) [114] 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.

7.2.3 Compensation Pattern

Name: Compensation Pattern

Problem: Errors or partial/complete failures in an HPC environment cause applications to experience errors or fail.

Context: The pattern applies to a system that 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 must also have the following characteristics:

- The error or failure in the system that the pattern handles must be detected, although the pattern may offer implicit error/failure detection.
- The system must allow for modular design with well-defined inputs and outputs for each module.
- The error/failure must not be in the inputs provided to the system.

Forces:

- The scope and strength of the redundancy employed by the pattern determines the overhead to the system performance.
- The pattern introduces a penalty in terms of time (increase in execution time), or space (increase in resources required) independent of whether errors or failure occurs during system operation.
- The error/failure-free overhead penalty must be minimized.

Solution: The `Compensation` pattern accounts for certain errors in the system, or the failure of a (sub-) system. The pattern is based on the definition of modules (with well-defined inputs and outputs), about which redundant information is maintained. The scope of the module may include a sub-system, or cover the full system. The pattern provides error/failure resiliency operation by applying redundancy to the module. The redundancy may be in the form of maintaining a group of replicas of the module, referred to as N-modular redundancy. The module replicas are functionally identical and each replica receives identical inputs. Alternatively, the redundancy may be in the form of maintaining encoded information about the module. The key components of the pattern are illustrated in Figure 8.

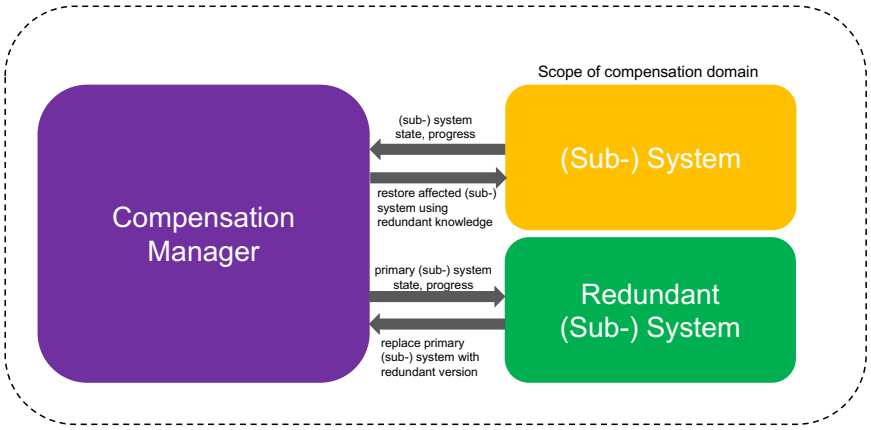

Figure 8. Compensation Pattern

Capability: The pattern supports error/failure detection and in some cases error/failure correction by using the redundant information about a module to recompense for the presence of an error/failure. When the redundancy is in the form of modular replication, an error or failure in one of the modules may be tolerated by substituting the module with a replica. The replicas of the modules permit the system to continue operation even in the presence of a fail-stop failure. In order to recover from 2N errors/failures in the system, there must be 2N + 1 distinct replicas. For the detection of errors, the outputs of the replicas of the system are compared by a monitoring system. For a system to tolerate an error/failure, the number of replicas must be greater than two, in which case the monitor performs majority voting on the outputs produced by the replicas. This enables incorrect outputs from replicas in error/failed state to be filtered out. The working of the pattern is shown in Figure 9.

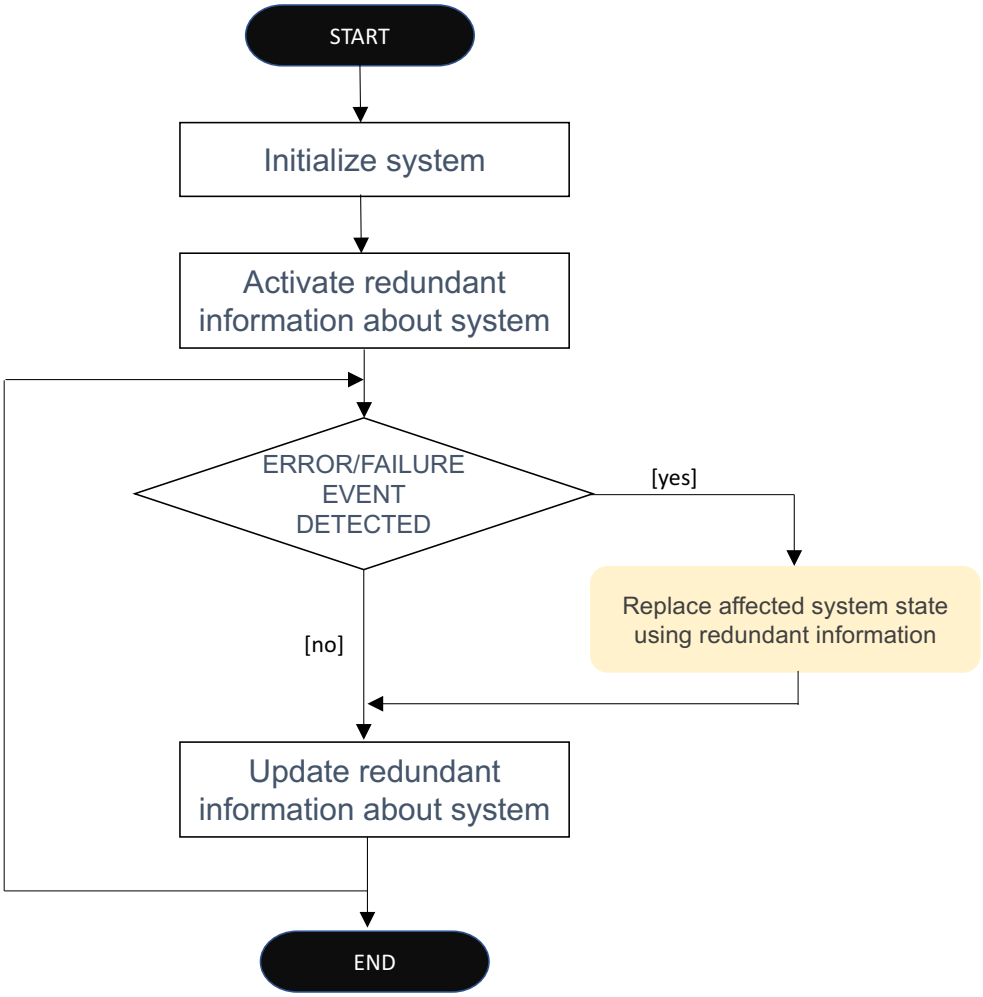

Figure 9. Compensation Pattern Flowchart

Protection Domain: The protection domain of the `Compensation` pattern extends to the scope of the module on which redundancy is applied.

Resulting Context: The `Compensation` pattern requires the replication of the system and its inputs. The design effort and complexity of replication of the system depends on the replication method. A naive replication requires low design effort; the design of functionally identical but independently designed versions of the replicas requires higher design effort. The preparedness of the replica during system operation to compensate for the error/failure state determines the level of overhead: The replica may be *active* or *passive*; the replica state may be classified *hot*, *warm* and *cold* which are the forms of replication configurations based on the levels of intervention required for compensating for an error/failure.

Examples: The N-modular redundancy approach is used at the hardware and software levels in various HPC components. Dual-modular redundancy (DMR) is typically used for error detection while triple-modular redundancy (TMR) is used for error detection and correction [97]. Redundant information in the form of error correction codes is also used at the hardware-level and at the application-level for application data structures [82] for compensation of data errors.

Rationale: The `Compensation` patterns enable systems to tolerate errors/failures by relying on the replicated versions of the system to substitute a failed system, or to infer and compensate for errors/failures by comparing the outputs of the replicas.

Related Patterns: The `Recovery` pattern is complementary to the compensation pattern, although they both seek to create conditions to recreate correct state. The key difference between the `Recovery` and the `Compensation` patterns is the method used to maintain any additional state that is used for error/failure processing. Unlike `Recovery` pattern, which uses a temporally forward or backward error/failure-free version of the system, a compensation pattern utilizes some form of redundancy to tolerate errors/failures in the system.

Known Uses:

- The MR-MPI [64], rMPI [68] and their successor RedMPI [70] are implementations of the message passing interface that transparently replicate MPI messages. They detect and correct errors through active comparison between redundant copies across MPI ranks.
- Production HPC systems use memory modules that provide ECC, which maintain redundant bits per memory line. These redundant bits compensate for bit flip errors within the memory lines and enables detection and correction of certain errors. The standard ECC supports single-bit error correction and double-bit error detection (SECDED) [115]. Chipkill [51] permits single 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.

7.3 Architectural Patterns

7.3.1 Fault Diagnosis Pattern

Pattern Name: Fault Diagnosis Pattern

Problem: An incomplete understanding of the cause and impact of a fault in an HPC system design makes design process of remedial actions difficult.

Context: The pattern is a derivative pattern of the `Fault Treatment` strategy pattern and it 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 anomalies in the behavior of a 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 of the monitoring and monitored systems may interfere with the operation of the monitored system. The frequency of these interactions must be bounded.
- The time interval for the monitoring system to infer the presence of an anomaly or a defect must be minimized in order to report the fault in a timely manner (before the activation of an error or failure). This time delay also incurs an overhead to the operation of the monitored system.
- The diagnostic resolution, i.e., the degree of accuracy of the fault diagnosis, must be high.

Solution: The `Fault Diagnosis` pattern identifies the presence of a defect or anomaly that has the potential to activate, leading to the occurrence of an error or a failure. The pattern uses a monitoring system that observes specific parameters of a monitored system. These components are shown in Figure 10. Since the presence of a fault that has not activated into an error does not affect the correct operation, the `Fault Diagnosis` pattern makes an assessment on what may potentially be wrong with a system. The diagnosis entails making inferences based on the observed behavior of a system and it also help narrow the search for the root cause of the fault. The assessment is based on the standard operating behavior of the monitored system.

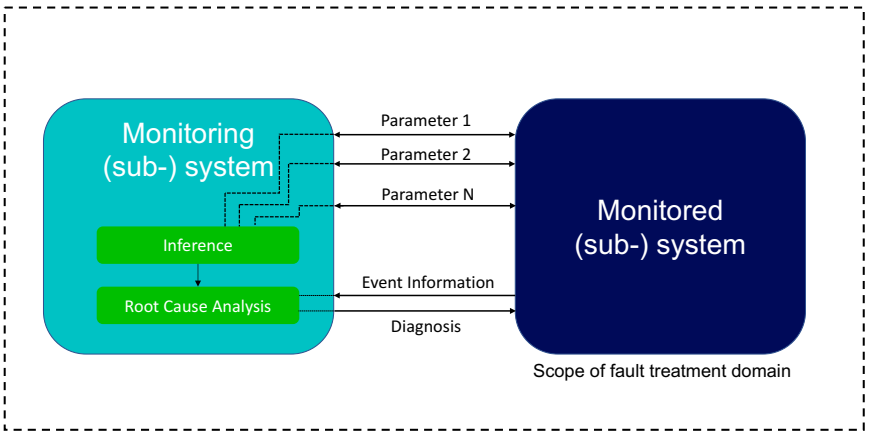

Figure 10. Fault Diagnosis Pattern

Capability: The pattern provides a method that attempts to recognize the presence of an anomaly or a defect within a system and identifies the fault location and type. The monitoring system observes deviations in the parameters to determine the presence of a fault and seeks to identify the location of the fault. The flowchart for this pattern is shown in Figure 11.

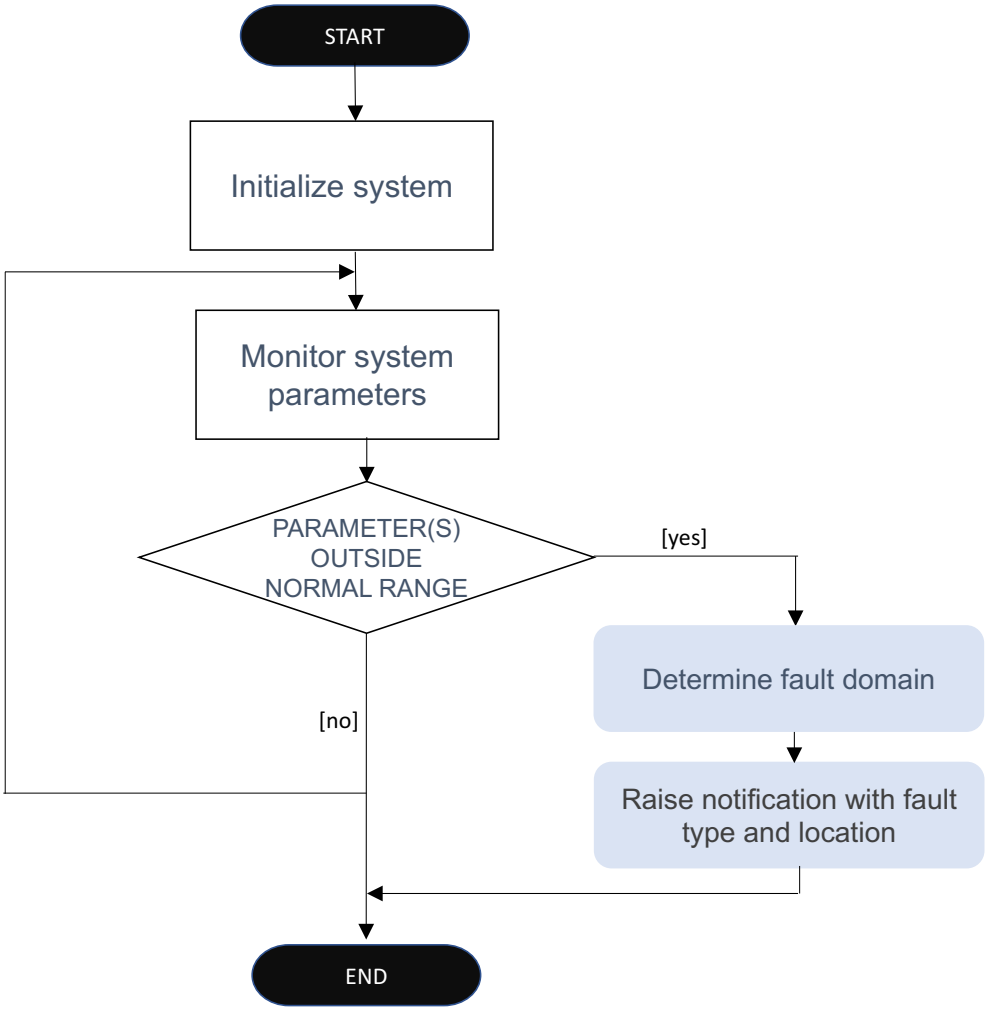

Figure 11. Fault Diagnosis Pattern Flowchart

Protection Domain: The protection domain extends to the scope of the monitored system, about which the monitoring system gathers data for discovering anomalies/defects. Since the pattern seeks to detect and alleviate a fault before activation, the protection domain implicitly extends to other systems that are interfaced to the monitored system.

Resulting Context: The `Fault Diagnosis` pattern requires the designer to identify parameters that indicate the presence of faults. The system design must include a monitoring system, which introduces design complexity in the overall system design. When the monitoring system is extrinsic to the monitored system, the design effort may be simplified, but the interfaces between the systems must be well-defined. However, when the monitoring system is intrinsic to the design of the monitored system, the complexity of the design process increases due to need to interface the sub-systems. The `Fault Diagnosis` pattern only

infers the presence of a defect 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 of defect mechanism.

Rationale: The `Fault diagnosis` pattern enables error/failure avoidance by detecting an anomaly in the system before it results in an error/failure. The pattern may also provide insights into the root cause and the location of the fault.

Examples: Various hardware faults are detected through analysis of software symptoms, such as observation of latency of operations. In the context of HPC systems, faults may be detected and diagnosed by accumulating empirical data on the characteristics and the behavior of system components and using this inferred knowledge to discover fault patterns.

Related Patterns: In contrast to the error/detection recovery and compensation patterns, the fault diagnosis pattern is a passive pattern that observes system behavior and infers the presence of a fault based on the deviation from specified *normal* behavior of the system.

Known Uses:

- Self-Monitoring and Reporting Technology (SMART) [146] is used in HDD and solid state disk systems. It detects and reports on various indicators of the disk's reliability.
- The Intelligent Platform Management Interface (IPMI) [50] is a set of specifications that provide standardized interfaces for monitoring hardware health information such as the system temperatures, fans, power supplies, etc. and the logging of abnormal system states in the System Event Log (SEL), Sensor Data Record (SDR). Using these interfaces, software tools may monitor the health of system resources and infer the presence of anomalies in behavior.

7.3.2 Reconfiguration Pattern

Pattern Name: Reconfiguration Pattern

Problem: The presence of a fault, error or failure event may affect configuration of the system components, preventing its correct operation.

Context: The pattern derives from the `Fault Treatment` and `Error/Failure Recovery` strategy patterns and is intended to manage the effects of a fault, error or failure event. It applies to a system that has the following characteristics:

- The system that 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 fault, error or failure in the system that the pattern handles must be detected; the pattern offers no implicit fault monitoring, prediction, or error/failure detection capability.
- The system should be able to be partitioned into logical modules; altering the interconnection between the modules in the system enables the system to remain operational.

Forces:

- The system design must allow for encapsulation of system functions into a set of well-defined modules; the ability of a system to alter the interconnection among modules must permit system operation, which is functionally equivalent to the fault, error, or failure-free version of the system.
- The time to reconfigure the system must be minimized; the interval required to reconfigure incurs an overhead to the operation of the system.

- The reconfiguration may force the system to operate at a lower level of performance. The performance degradation because of the reconfiguration of the system must be minimized.

Solution: The `Reconfiguration` pattern prevents a fault, error or failure event from affecting the correct operation of the system. The pattern entails modification of the interconnection between modules or sub-systems in the overall system. The system may assume several configurations in response to a fault, error or failure event, each of which is characterized by its own topology of interconnections, but retains functional equivalency with the original system configuration. The pattern makes sub-systems out of larger systems by isolating of the module of the system affected by the event. The sub-system consists of an altered organization of the system modules that is functionally equivalent to the system configuration before the occurrence of the fault, error or failure event. Figure 12 provides a graphical description of this pattern solution.

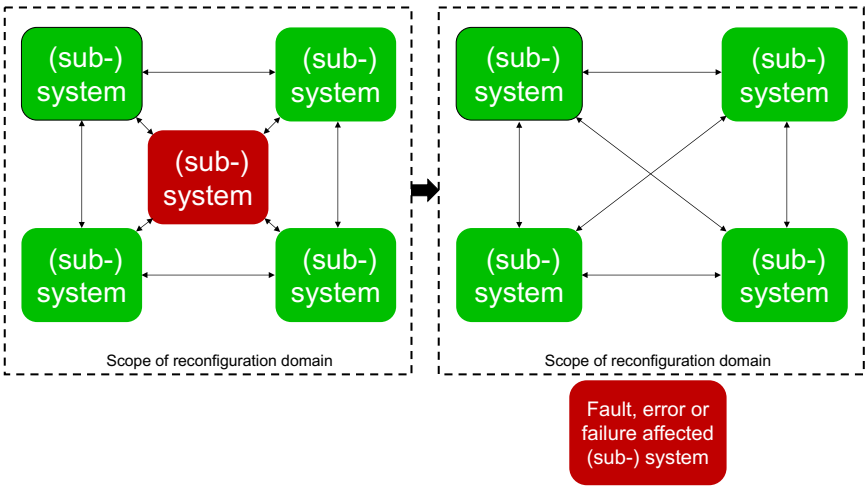

Figure 12. Reconfiguration Pattern

Capability: The pattern enables systems to tolerate the impact of a fault, error or failure by isolating the module affected by the event. The altered configuration prevents the module that experienced the fault, error or failure condition from affecting the correctness of the overall system operation, and enables the system to continue operation. The working of this pattern is shown in Figure 13.

Protection Domain: The protection domain of the `Reconfiguration` pattern covers all the modules whose interconnections may be reconfigured in order to continue operating the system in a functionally equivalent operating state.

Resulting Context: The reconfiguration of the system may result in the operation at a degraded performance level. This incurs additional time overhead to the system. The design of the reconfiguration capability requires the partitioning of the system into modules that remain functionally correct in multiple different configurations; some of these configurations are sub-systems of the original system. The design complexity in defining the scope of these modules and validate their equivalency to alternative configuration is significant. The pattern also introduces runtime overhead in altering the interconnections of a system.

Examples: Various software-based solutions support the reconfiguration of the subsystems to isolate a fault, error or failure prone resource. This includes the configuration of processors, memory subsystems, storage subsystems and interconnect topologies. Cluster management systems often support the capabilities to dynamically adapt the cluster configuration based on the health of the compute nodes in the system.

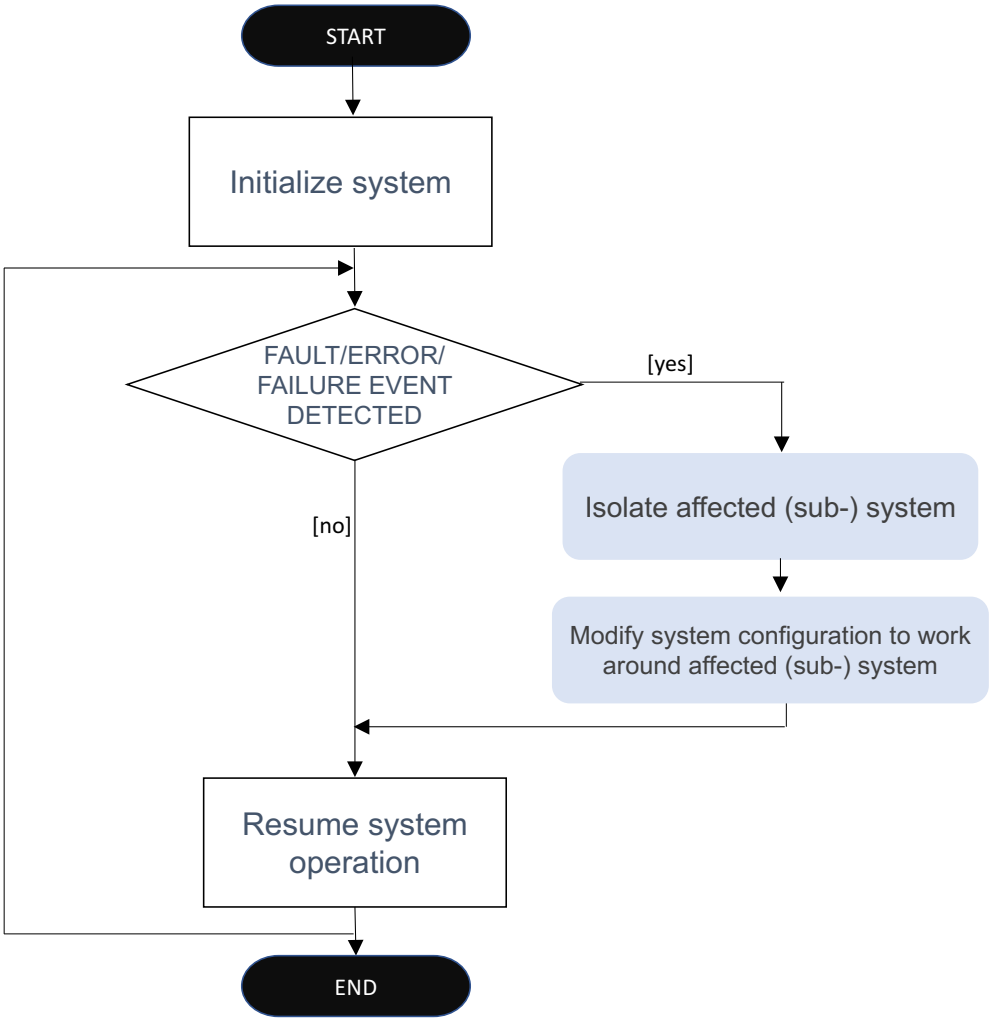

Figure 13. Reconfiguration Pattern Flowchart

Rationale: The `Reconfiguration` pattern enables a system to tolerate to a fault, error or failure by adapting itself to the impact of the event and continuing to operate. The pattern enables systems to make forward progress by relying on the reconfigured version of the system.

Related Patterns: Like the `Checkpoint-Recovery` pattern, the `Reconfiguration` pattern supports the isolation and removal of a fault in a system, and provides capabilities for recovery of the system from the impact of an error or failure event. In contrast to the `Checkpoint-Recovery` pattern, which maintains snapshots of the system to stable storage to perform forward or backward recovery, the `Reconfiguration` pattern isolates a module in error or failure state and adapts the interconnections among the remaining modules to recover the operation of the system.

Known Uses:

- 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 [9].

- In the Cray Linux Environment CLE, the NodeKARE module automatically runs diagnostics on all involved compute nodes in the cluster whenever a user's program terminates abnormally. It removes the failing nodes from the pool of available compute nodes so that subsequent jobs are allocated to healthy nodes and are able to run reliably to completion [89].

7.3.3 Checkpoint Recovery Pattern

Pattern Name: Checkpoint Recovery Pattern

Problem: An unrecoverable error or a failure event in an HPC environment prevents the execution of applications.

Context: The pattern is a derivative of the `Recovery` pattern and it applies to a system that 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 must also have the following characteristics:

- The error or failure in the system must be detected; the pattern offers no implicit error/failure detection capability.
- The system has well-defined intervals that enables the pattern to transition the system to a known correct interval in response to an error/failure.
- When the system encounters the same non-deterministic events with the same system state, and at the same location and timing, the system behavior is identical. The execution of the system between the intervals is deterministic.
- 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.

Forces:

- The pattern requires stable storage to capture system state or to log events, which increase overhead in terms of resources required by the system.
- The process of compartmentalizing and capturing system state or logging interferes with system operation. The error/failure-free overhead penalty must be minimized.
- The extent of system state captured during the creation of a checkpoint incurs space and time overheads.
- The post-recovery state of the system must be as close as possible to an error/failure-free system state.
- The frequency of creation of checkpoints determines overhead. An increase in the checkpointing frequency increases system execution time, but reduces amount of lost work upon occurrence of an error/failure.
- The time interval required for the recovery of a system from a stable recovery point as well the time interval to create and commit a checkpoint to a stable storage system must be less than the mean time between errors/failures in the system to enable the system to make forward progress.

Solution: The `Checkpoint-Recovery` pattern accounts for unrecoverable errors in the system, or supports the recovery from the failure of the system. In general, checkpoint-recovery solutions are classified into checkpoint-based and log-based strategies. In the case of the former, the solution is based on the creation of snapshots of the system state and maintenance of these checkpoints on a stable storage system during the error/failure-free operation of the system. In the case of log-based protocols, the pattern uses a combination of checkpointing and logging of non-deterministic events in the system. The pattern is

illustrated in Figure 14. Upon detection of an error or a failure, the checkpoints are used to recreate last known error/failure-free state of the system. The system operation is then restarted. Based on a temporal view of the system's progress, the error/failure recovery may be either forward or backward.

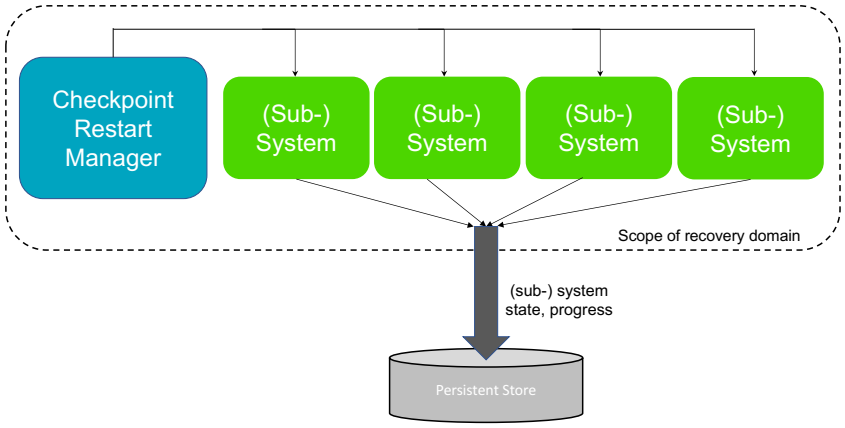

Figure 14. Checkpoint Recovery Pattern

Capability: The `Checkpoint-Recovery` pattern periodically preserves the essential aspects of the system state that may be subsequently used to recover the system from an error/failure. The pattern handles an error or a failure by retrieving an error/failure-free system state from a storage system, or by reconstructing such state, and substituting the erroneous/failed state with the error/failure-free state. Therefore, the system is able to resume operation with a version of the system state that is free of any effects of the error/failure event. These steps are shown in the flowchart in Figure 15. The solution offered by the pattern supports only recovery; the detection and containment of the error/failure is beyond the scope of the pattern's capabilities.

Protection Domain: The protection domain for a `Checkpoint-Recovery` pattern is determined by the extent of state captured during checkpoint operation and/or the number of system operations that can be recovered from the log of events. For failure recovery, the protection domain must include the state of the complete system. For error recovery, the protection domain includes the scope of system state captured at every checkpoint; the error recovery is possible only when the location of the error is within the state that is checkpointed.

Resulting Context: The use of the `Checkpoint-Recovery` pattern enables a system to tolerate errors or failures using substitution or inference to recreate an error/failure free version of the system state. The system is restarted from a known previous stable state (backward recovery), or to an inferred stable state (forward recovery), which potentially results in some loss of work. The pattern requires interruption of the system during error/failure-free operation of the system to record the checkpoint, which results in a time overhead. The frequency of creation of checkpoints and/or event logging determines the overhead to system operation; frequent checkpointing/logging incurs proportionally greater overheads during error/failure-free operation. However, more frequent checkpointing and logging reduces the amount of lost work when the system encounters an error/failure event. The checkpointing/logging latency affects the overhead during error/failure-free operation on account of the latency to write the checkpoint to a storage system. The scope of the system state captured during a checkpoint operation results in a space overhead due to the requirements of resources to retain the checkpoints. This scope also affects the time required for recovery from an error/failure due to the time required to retrieve a checkpoint, or recreate state from the log.

Rationale: The solution offered by this pattern is independent of the type of error/failure and its mode of

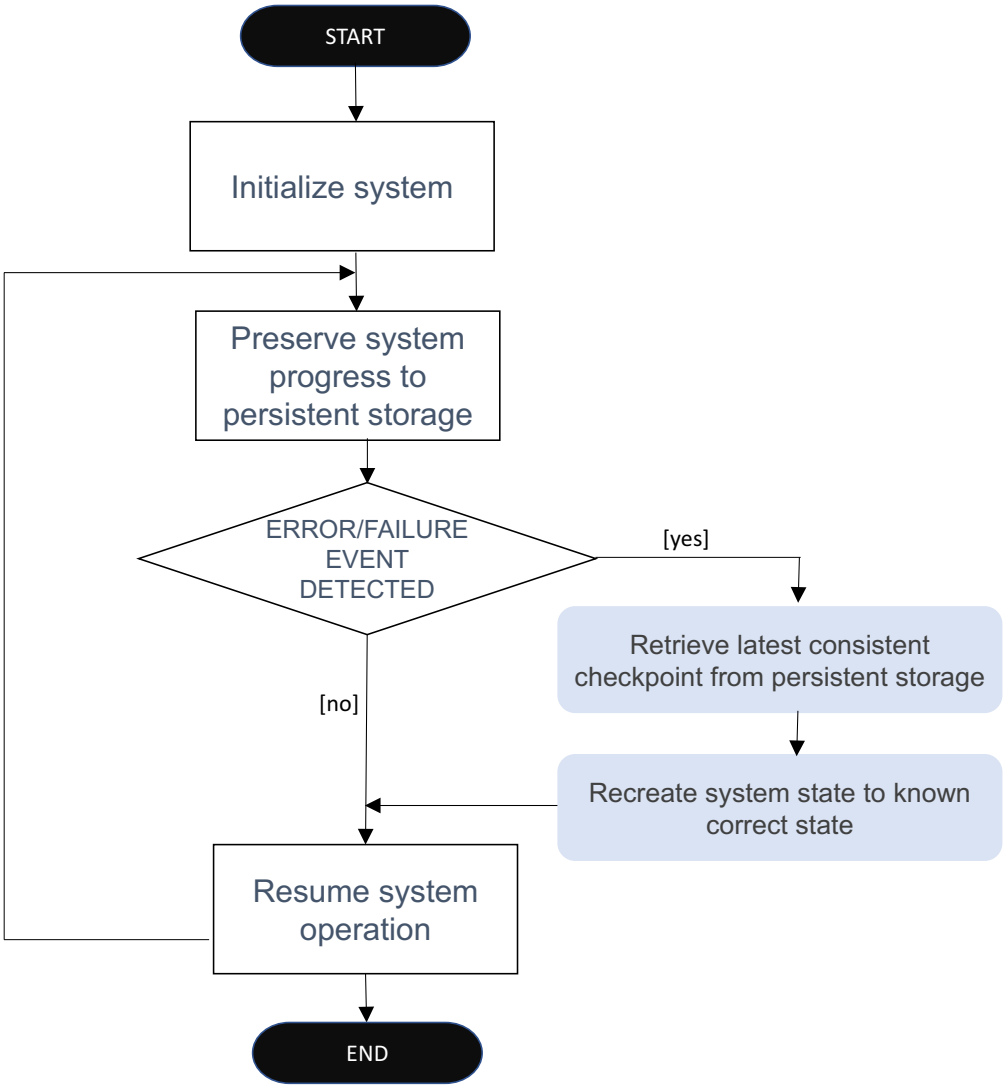

Figure 15. Checkpoint Recovery Pattern Flowchart

propagation in the system. Therefore, the design effort and complexity in using this pattern in any system design in low. The pattern also enables implementation of portable solutions due to the low dependence on system's architecture.

Examples: In the context of HPC systems, checkpoint and restart capabilities in the software layers, including various library-based and operating system-based solutions enable recovery from process errors/failures.

Related Patterns: The `Redundancy` and `Design Diversity` patterns are complementary to the `Checkpoint-Recovery` pattern, although they both seek to create conditions to recreate correct state. The key difference between these classes of patterns is the method used to maintain any additional state that is used for error/failure processing. While the diversity patterns use replication of the system, a recovery pattern relies on committing error/failure-free versions of the system to stable storage.

Known Uses:

- The Berkeley Labs C/R library (BLCR) [57] for Linux processes, the Scalable Checkpoint/Restart (SCR) [114] library for MPI-based applications are widely used instances of the checkpoint-restart pattern in HPC systems.
- Library implementations of the MPI standard, such as OpenMPI also support transparent checkpoint-restart capabilities [87].
- Log-based recovery based on message logging has been adopted by implementations of MPI [27].

7.3.4 Redundancy Pattern

Pattern Name: Redundancy Pattern

Problem: The occurrence of error or failure events caused by physical faults in an HPC environment often cause unrecoverable errors or failures of (sub-) systems, which prevent the execution of applications.

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

- The system must be 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 must allow for modular design with well-defined inputs and outputs for each module.
- The cause of errors or failures experienced by the system may not be due to errors in the inputs.

Forces:

- The pattern introduces penalty in terms of time (increase in execution time), or space (increase in resources required) independent of whether an errors or failure occurs.
- The error/failure-free overhead penalty introduced by the replication of state must be minimized.

Solution: The `Redundancy` pattern supports the detection of unrecoverable errors or failures in the system state, and accounts for the loss of correctness/functionality caused by such events. The pattern creates a group of N replicas of a system's state as shown in Figure 16. The redundancy may include replication of the system's operation and/or the inputs to the system. Each of the N copies of the system state exist simultaneously. The redundant versions of the system are provided with the identical inputs, and their respective outputs are compared to detect and potentially correct the impact of an error or a failure in either replica of the systems.

Capability: The availability of replicated versions of the system state enable the following capabilities:

- Fail-over, which entails substitution of a replica in error or failed state with another replica that is error/failure-free.
- Comparison, which entails observing the likeness of each replica's outputs as means to detect the presence of an error or failure in either replica.
- Majority voting on the outputs produced by each replica system enables the detection of errors and failures, and filtering out the outputs that fall outside the majority.

The working of this pattern is shown in the flowchart in Figure 17.

Protection Domain: The protection domain of the pattern extends to the scope of the system state that is replicated.

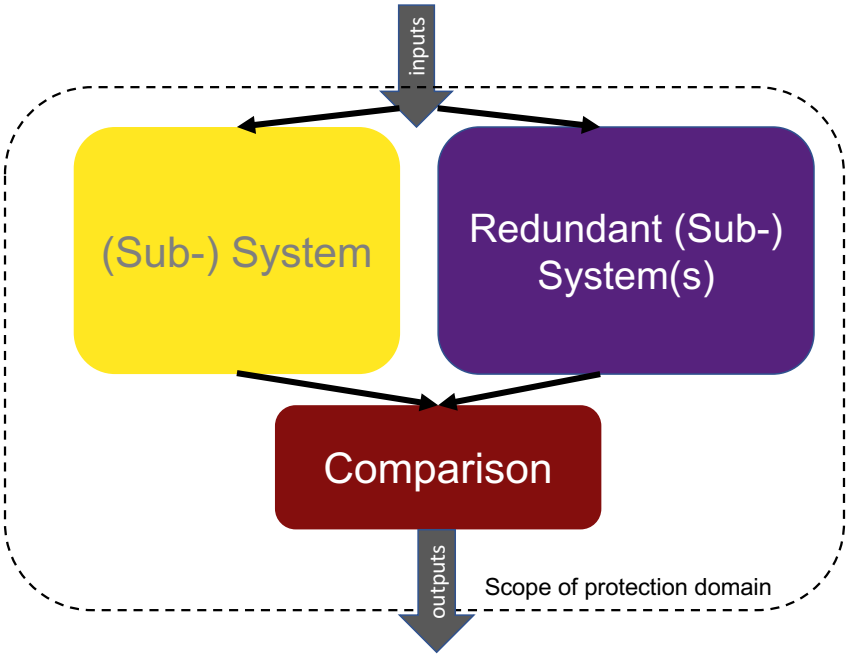

Figure 16. Redundancy Pattern

Resulting Context: The design effort and complexity of replication of the system state requires low design effort since the replication entails creation of identical copies of the system state.

Rationale: The `Redundancy` patterns enable systems to tolerate errors/failures by relying on the replicated versions of the system state to compensate for an error or failure in the system, or to infer and compensate for errors/failures by comparing the outputs of the replicas.

Examples: The use of dual-modular redundancy for error detection and triple-modular redundancy for error/failure detection and correction is common in various hardware and software-level modules. The use of redundant information based on encoding the state is also used to detect and correct errors in system state.

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

Known Uses:

- HPC systems contain service nodes that are responsible for system management tasks while the parallel computation is performed by a set of compute nodes. The tasks include user login, network file system, job and resource management, communication services. Various existing solutions provide hot-standby redundancy with transparent fail-over to manage the occurrence of failures, including the Simple Linux Utility for Resource Management (SLURM) [6], as well as the metadata servers of the Parallel Virtual File System (PVFS) [40] and the Lustre file system [7].
- Production HPC systems such as the Cray XC40 series [90] include redundant power supplies, voltage regulator modules and cooling fans to ensure continuous operation in the event that one of these units experiences malfunction or failure.

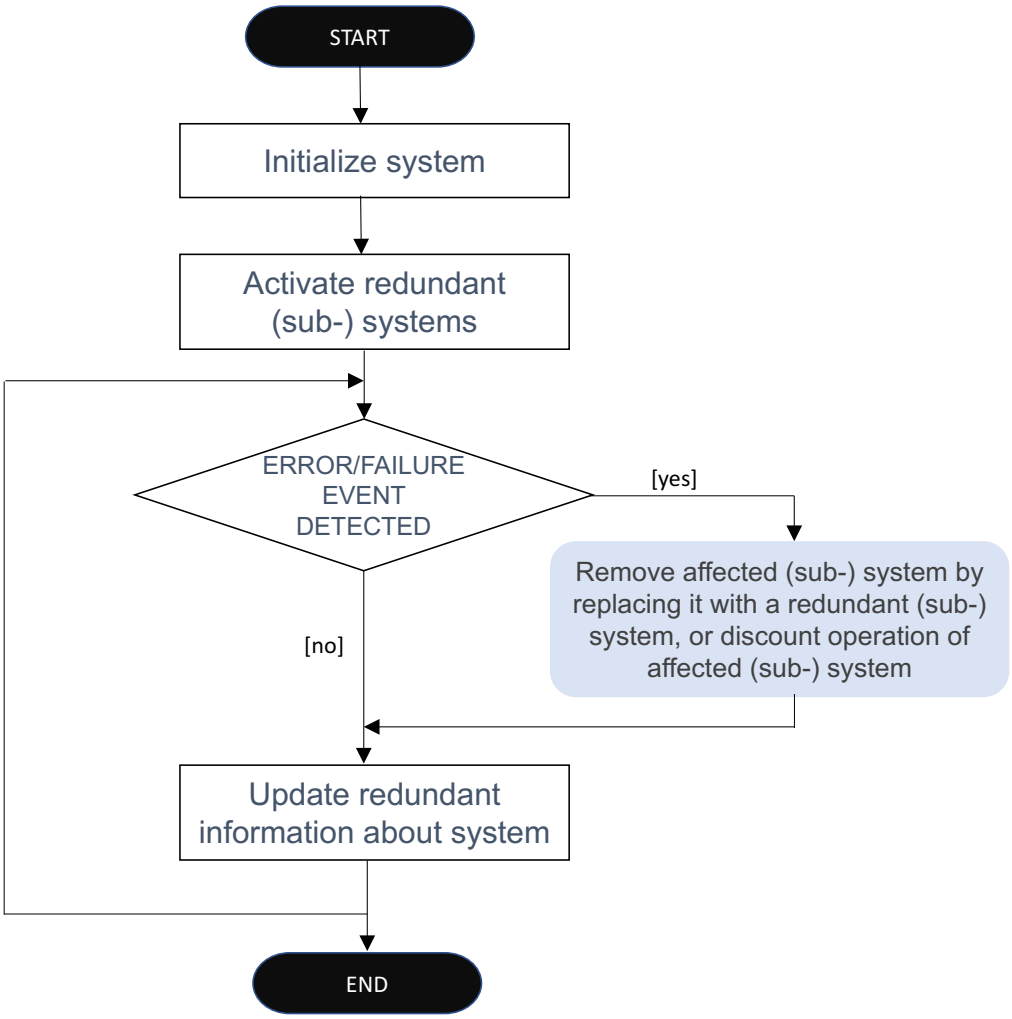

Figure 17. Redundancy Pattern Flowchart

7.3.5 Design Diversity Pattern

Pattern Name: Design Diversity Pattern

Problem: Design faults introduced by human mistakes or defective design tools cause systems to malfunction or fail.

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

- 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 requires distinct implementations of the same design specification, which are created by different individuals or teams.

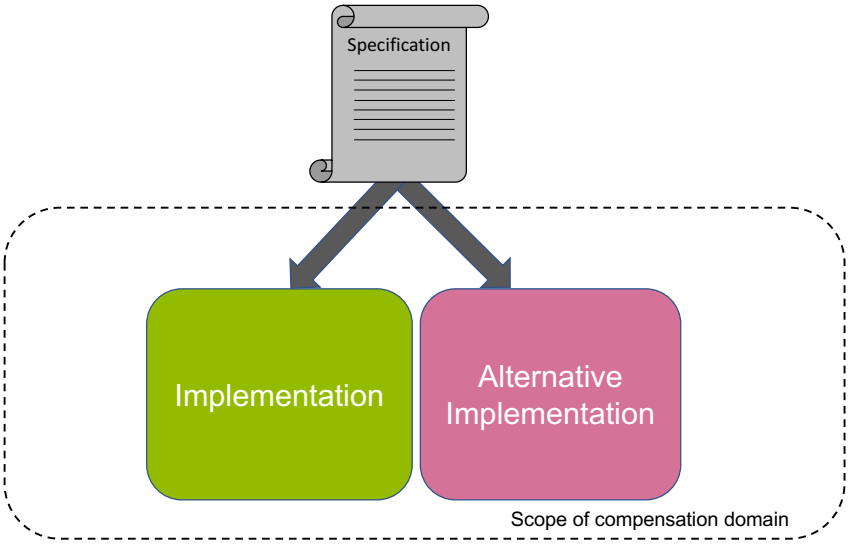

Figure 18. Design Diversity Pattern

- The pattern increases the design complexity due to the need of additional design and verification effort required to create multiple implementations.
- The performance penalty during error/failure-free operation due to disparity in the implementation variants must be minimized.

Solution: The `Design Diversity` pattern detects and corrects errors or failures in the system that may occur due to design flaws in the system. The prevention of design faults and their removal are not straightforward tasks due to the verification effort required to ensure that a system operates according to its specification for all possible inputs and operating states. The pattern enables systems to tolerate errors/failures due to design faults that may arise because of incorrect interpretation of the specifications, or due to mistakes during implementation. The `Design diversity` pattern entails partitioning the system into N replica systems. Each replica system is an implementation variant of a system's design specification, as illustrated in Figure 18. These design variants are applied in a time or space redundant manner. The redundant systems are provided with identical inputs and their respective outputs are compared to detect and potentially correct the impact of an error or a failure in either replica of the systems.

Capability: The pattern relies on independently created, but functionally equivalent versions of a system specification. The systems operate in parallel, in a time or space redundant manner. The pattern is based on the idea that the different implementations have different designs and therefore the same fault would not affect all the versions at once and in the same manner. This capability, which is shown in Figure 19, enables the systems to tolerate errors or failures caused by design flaws.

Protection Domain: The protection domain extends to the scope of the system that is described by the design specification, of which multiple implementation variants are created.

Resulting Context: The `Design Diversity` pattern enables systems to tolerate errors and failures due to design flaws given that the likelihood that each replica is affected by the same transient error or failure is small. Additionally, the pattern also supports resilience transient errors/failures because the replica systems are functionally identical and are deployed in a redundant manner. However, the need to implement multiple independent versions requires design and verification effort.

Rationale: The intent behind applying this pattern is to eliminate the impact of human error during the

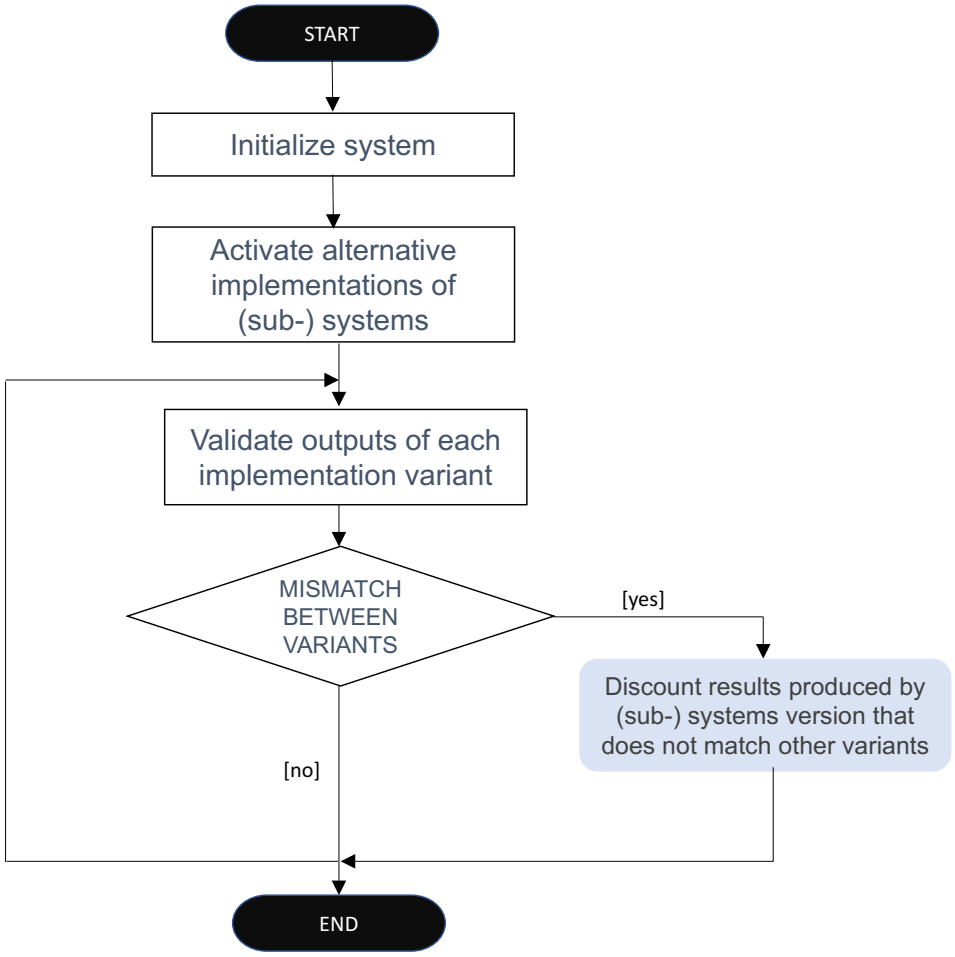

Figure 19. Design Diversity Pattern Flowchart

implementation of a system. Due to the low likelihood that different individuals or teams introduce identical bugs in their respective implementations, the pattern enables compensating for errors or failures caused by a bug in any one implementation of the same design.

Examples: The concept of design diversity is predominantly used in software design. N-version programming is used in software development, which entails the implementation of variants of a design by different teams, but to a common specification.

Related Patterns: The `Design Diversity` and `Redundancy` patterns are based on inclusion of redundancy to compensate for errors or failures. In contrast to the `Redundancy` pattern, which simply replicates the system state, the `Design Diversity` pattern typically uses multiple versions of the system that are only functionally equivalent.

Known Uses:

- The design diversity solution is widely used in the validation of the results produced by scientific applications, particularly those that require high-precision floating point arithmetic. Such applications may be compiled and executed using alternative implementations of compiler toolchains, message passing libraries, numerical analysis libraries to validate the application results.

7.4 Structural Patterns

7.4.1 Monitoring Pattern

Pattern Name: Monitoring Pattern

Problem: The presence of a defect or anomaly in the system 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 anomalies in the behavior of a 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 anomalies in the operation.

Forces:

- The interactions of the monitoring and monitored systems may interfere with the operation of the monitored system. The frequency of these interactions must be bounded.
- The time interval for the monitoring system to infer the presence of an anomaly or a defect must be minimized to report the fault in a timely manner (before the activation of an error or failure). This time delay also incurs an overhead to the operation of the monitored system.
- The degree of accuracy of the fault diagnosis must be high.

Solution: The `Monitoring` pattern analyzes the behavior of a system with the goal of discovering the presence of a defect or anomaly in the system that has the potential to cause errors or failures. The pattern, whose structure is outlined in Figure 20, contains a monitoring system that observes specific parameters of a monitored system. The monitoring system may approach the problem using two strategies:

- *Effect-Cause 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 *normal* operation, the monitoring system attempts to determine the root cause. The monitoring system logically partitions the system into modules and progressively eliminates the modules known to be fault-free. Through this process, it narrows the search for the fault in the system.
- *Cause-Effect Diagnosis*: This approach is based on a set of known fault models and the monitoring system compares the system parameters with a model developed using fault free system operation, or using simulations. When the observed set of parameters deviates from a model, the presence of and the type of fault may be inferred.

The pattern reports the presence of a fault and its location based on these inferences.

Capability: The pattern provides a method that attempts to recognize the presence of an anomaly or a defect within a system and identifies the fault location and type. Its working is shown by the flowchart in Figure 21.

Protection Domain: The protection domain extends to the scope of the monitored system, about which the monitoring system gathers data for discovering anomalies/defects. Since the pattern seeks to detect and alleviate a fault before activation, the protection domain implicitly extends to other systems that are interfaced to the monitored system.

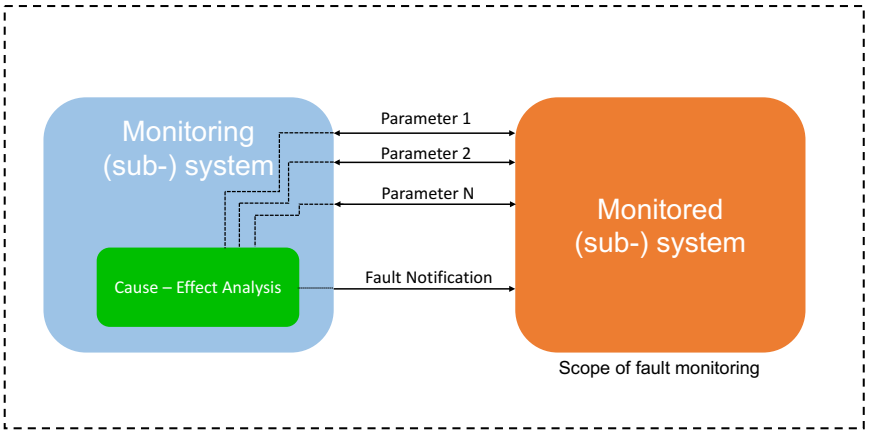

Figure 20. Monitoring Pattern

Resulting Context: The `Monitoring` pattern requires the designer to identify parameters that indicate the presence of faults. The system design must include a monitoring system, which introduces design complexity in the overall system design. When the monitoring system is extrinsic to the monitored system, the design effort may be simplified, but the interfaces between the systems must be well-defined. However, when the monitoring system is intrinsic to the design of the monitored system, the complexity of the design process increases due to need to interface the sub-systems. The `Monitoring` pattern only infers the presence of a defect 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 of defect mechanism.

Rationale: The pattern enables the monitored system to determine the presence of a fault and to analyze its root cause and location. The pattern enables the system to take precise corrective actions to prevent the activation of the fault to cause an error or failure event in the system.

Examples: Various platform-level interfaces, such as the IPMI [50], support the monitoring of a system's resources, including the memory modules, disk drive bays, network interfaces as well as various board-level sensors.

Related Patterns: The structure of the `Monitoring` pattern is closely related to the `Prediction` pattern since they both contain monitoring and monitored system entities. The key difference between these patterns is the amount of temporal information used by the patterns to assess the presence of a defect or anomaly in the system. The `Monitoring` pattern uses presently observed system parameters in contrast to the `Prediction` pattern, which uses historical trend information to forecast future fault events.

Known Uses: Various HPC system installations use monitoring tools for collecting performance- or health-related data about the system. Popular solutions include:

- Supermon High-Speed Cluster Monitoring System [143]
- Ganglia Monitoring System [107]
- Nagios [4]
- OVIS Lightweight Distributed Monitoring System [12]

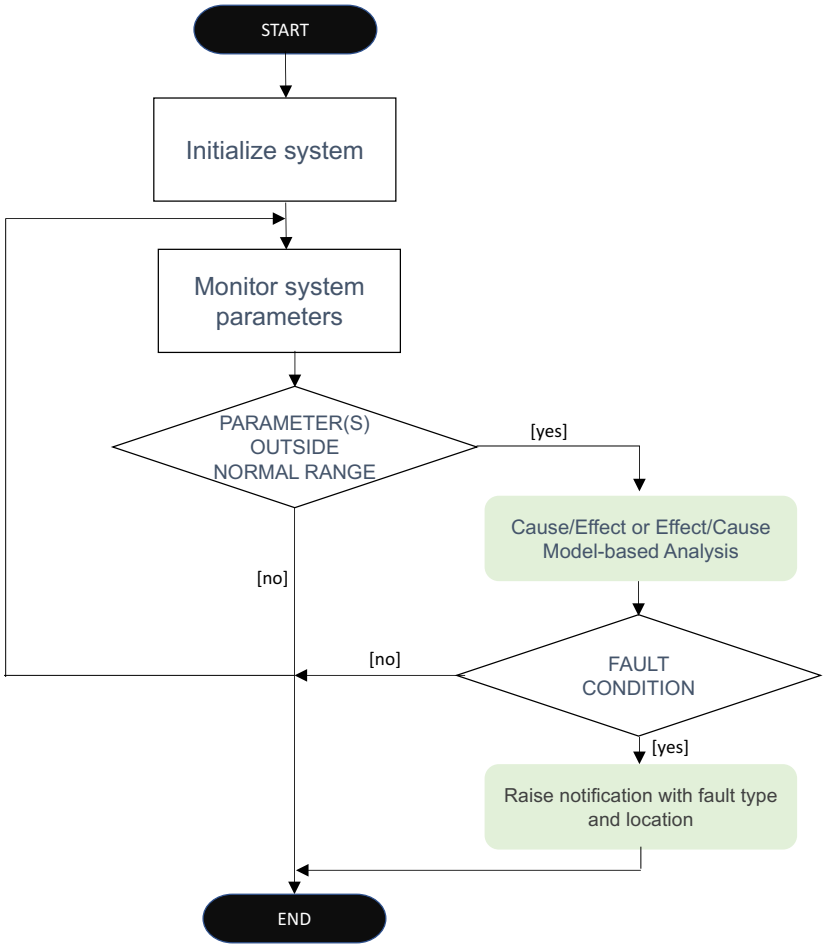

Figure 21. Monitoring Pattern Flowchart

7.4.2 Prediction Pattern

Pattern Name: Prediction Pattern

Problem: Recognizing system conditions that may cause future faults may help prevent an error or failure event in the system.

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 anomalies in the behavior of a 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 in order to analyze trends in fault occurrences.

Forces:

- The interactions of the monitoring and monitored systems may interfere with the operation of the monitored system. The frequency of these interactions must be bounded.
- The time interval for the monitoring system to infer the presence of an anomaly or a defect must be minimized in order to report the fault in a timely manner (before the activation of an error or failure). This time delay also incurs an overhead to the operation of the monitored system.
- The degree of accuracy of the fault prediction must be high.

Solution: The `Prediction` pattern identifies anomalous behavior that indicates the potential for future faults, which may result in errors or failures in the system. The pattern contains a monitoring system that observes specific parameters of a monitored system, as illustrated in Figure 22. The pattern may be based on:

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

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

- *Filter/Preprocessor*: removes incomplete fault data and duplicates and produces a consistent format for analysis.
- *Regression*: is a component that seeks to analyze the parameter values and establish relationships between them.
- *Knowledge Base*: is a storage component that maintains the rules or statistical properties and models, which may be used for online prediction of fault events using real-time data captured from the monitored system.

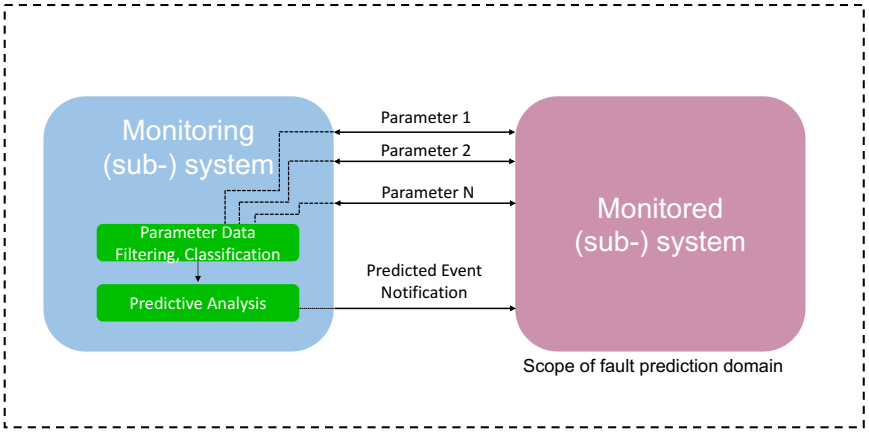

Figure 22. Prediction Pattern

Capability: The pattern provides a method that anticipates the occurrence of fault events based on patterns of behavior of the monitored system that attempts to recognize the potential for future occurrences of an anomaly or a defect within a system. Its operation is illustrated in the flowchart in Figure 23.

Protection Domain: The protection domain extends to the scope of the monitored system, about which the monitoring system gathers data for discovering anomalies/defects. Since the pattern seeks to detect and

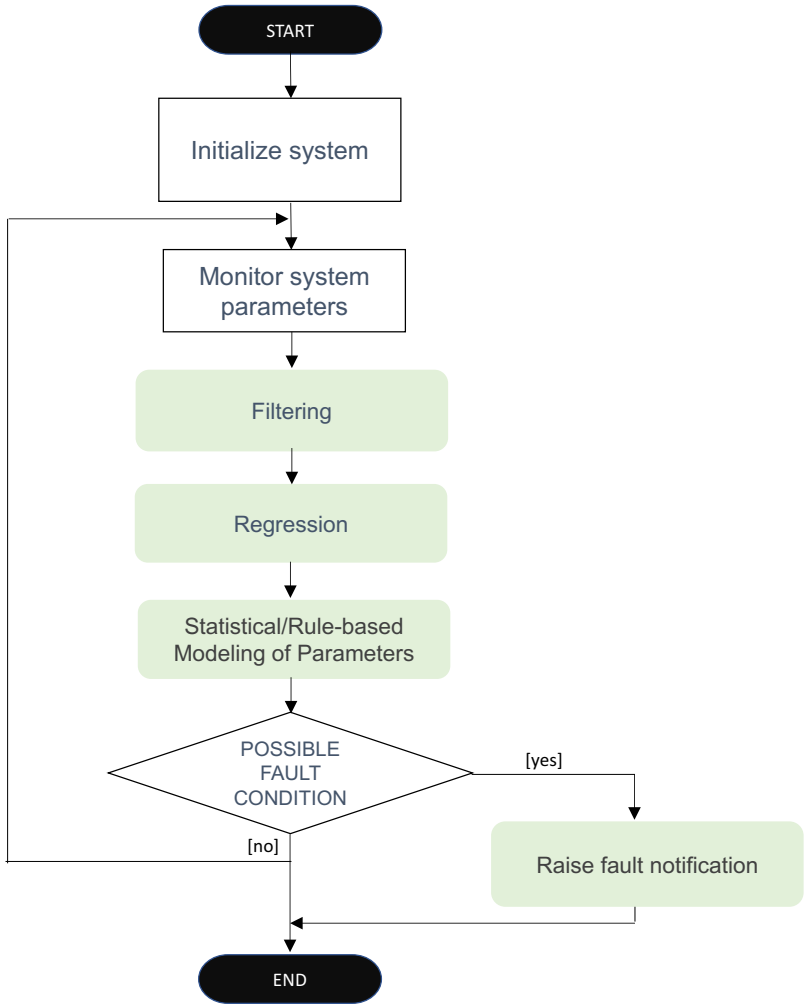

Figure 23. Prediction Pattern Flowchart

alleviate a fault before activation, the protection domain implicitly extends to other systems that are interfaced to the monitored system.

Resulting Context: The `Prediction` pattern requires the designer to identify parameters that indicate the presence of faults. The system design must include a monitoring system, which introduces design complexity in the overall system design. When the monitoring system is extrinsic to the monitored system, the design effort may be simplified, but the interfaces between the systems must be well-defined. However, when the monitoring system is intrinsic to the design of the monitored system, the complexity of the design process increases due to need to interface the sub-systems. The `Prediction` pattern only infers the presence of a defect 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 gives no indication of defect mechanism.

Rationale: The pattern enables the monitored system to use simple rule-based inference or historical trends in system behavior to predict future fault events. If future fault events are predicted with high precision, then avoidance or preventive actions may be used.

Examples: Hardware and software systems use correlations between past behaviors to predict the future occurrences of fault events, such as a memory device tends to show, for a given address, multiple repetitive correctable errors before showing an uncorrectable error.

Related Patterns: The structure of the `Prediction` pattern is closely related to the `Monitoring` pattern since they both contain monitoring and monitored sub-system entities. The key difference between these patterns is the amount of temporal information used by the patterns to assess the presence of a defect or anomaly in the system. The `Prediction` pattern uses historical trend information to forecast future fault events in contrast to the `Monitoring` pattern, which uses presently observed system parameters.

Known Uses:

- Prediction Models and Analysis for the IBM BlueGene/L [103]
- Event Prediction for Proactive Management in Large-scale Computer Clusters [132]

7.4.3 Restructure Pattern

Pattern Name: Restructure Pattern

Problem: The occurrence of a fault, or a resulting error or failure affects the configuration of a system such that correct system operation is not possible.

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

- The system that 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 fault, error or failure in the system that the pattern handles must be detected; the pattern offers no implicit fault monitoring, prediction, or error/failure detection capability.
- The system may be partitioned into N interconnected sub-systems.

Forces:

- The system must be able to be partitioned into a sub-set of sub-systems that is functionally equivalent to the fault, error, or failure-free version of the system.
- The restructuring may require the system to operate in degraded state using fewer than N sub-systems. The performance degradation of the system must be minimized.

Solution: The `Restructure` pattern solution is based on modifying the configuration between the N interconnected sub-systems to isolate the sub-system affected by a fault, error or failure. This reconfiguration seeks to exclude the affected sub-system from interaction with other sub-systems. The restructured system, as shown in Figure 24, includes N-1 sub-systems, and yet seeks to remain functionally equivalent to the system before the occurrence of the fault, error or failure event.

Capability: The pattern enables systems to tolerate the impact of a fault, error or failure by enabling the system to continue operation by preventing the affected part of the system from affecting the correctness of the system. The steps involved in this solution are shown in Figure 25.

Protection Domain: The protection domain of the `Restructure` pattern spans the part of system whose state may be reconfigured, and yet is able to continue operating in a functionally equivalent operating state.

Resulting Context:

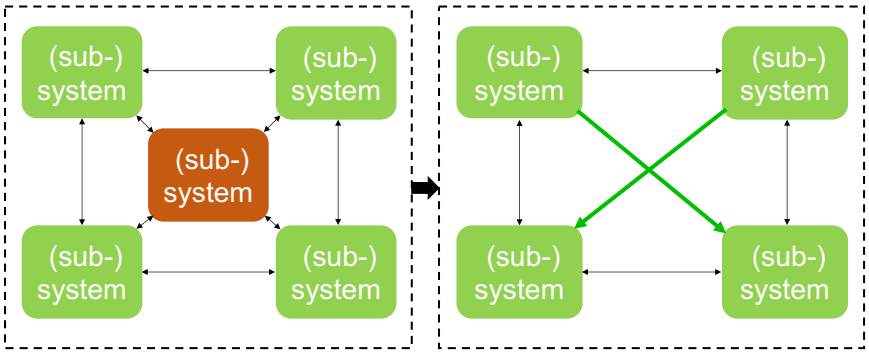

Figure 24. Restructure Pattern

- The reconfiguration of the system may result in the operation of the system in degraded condition. This incurs additional time overhead to the system.
- The pattern introduces additional design complexity since the system must remain functionally correct in multiple configurations.

Rationale: The `Restructure` pattern enables a system to tolerate to a fault, error or failure by adapting itself to the impact of the event and continuing to operate. The pattern enables systems to make forward progress by relying on the reconfigured version of the system.

Examples: Dynamic page retirement schemes are an example of the restructure pattern, in which pages that have an history of frequent memory errors are removed from the pool of available pages.

Related Patterns: The remaining reconfiguration patterns - the rejuvenation and reinitialization patterns - are closely related since they all seek to isolate the error/failed state of the system and prevent it from affecting the remaining error/failure-free part of the system.

Known Uses:

- The recurrence of memory errors in a specific cell or region of the chip often indicates degradation of the memory module. NVIDIA drivers support dynamic page retirement, which retires the page from the pool of available physical memory resources [120].
- The proposed user-level fault mitigation extension to the MPI standard [23] allows parallel applications to get notifications of process failures. ULFM provides a set of routines to revoke and restructure a MPI communicator consisting of the remaining processes.

7.4.4 Rejuvenation Pattern

Pattern Name: Rejuvenation Pattern

Problem: A fault event, or a resulting error or failure may cause a sub-system to operate incorrectly or stop functioning, which prevents correct operation of the overall system.

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

- The system that 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.

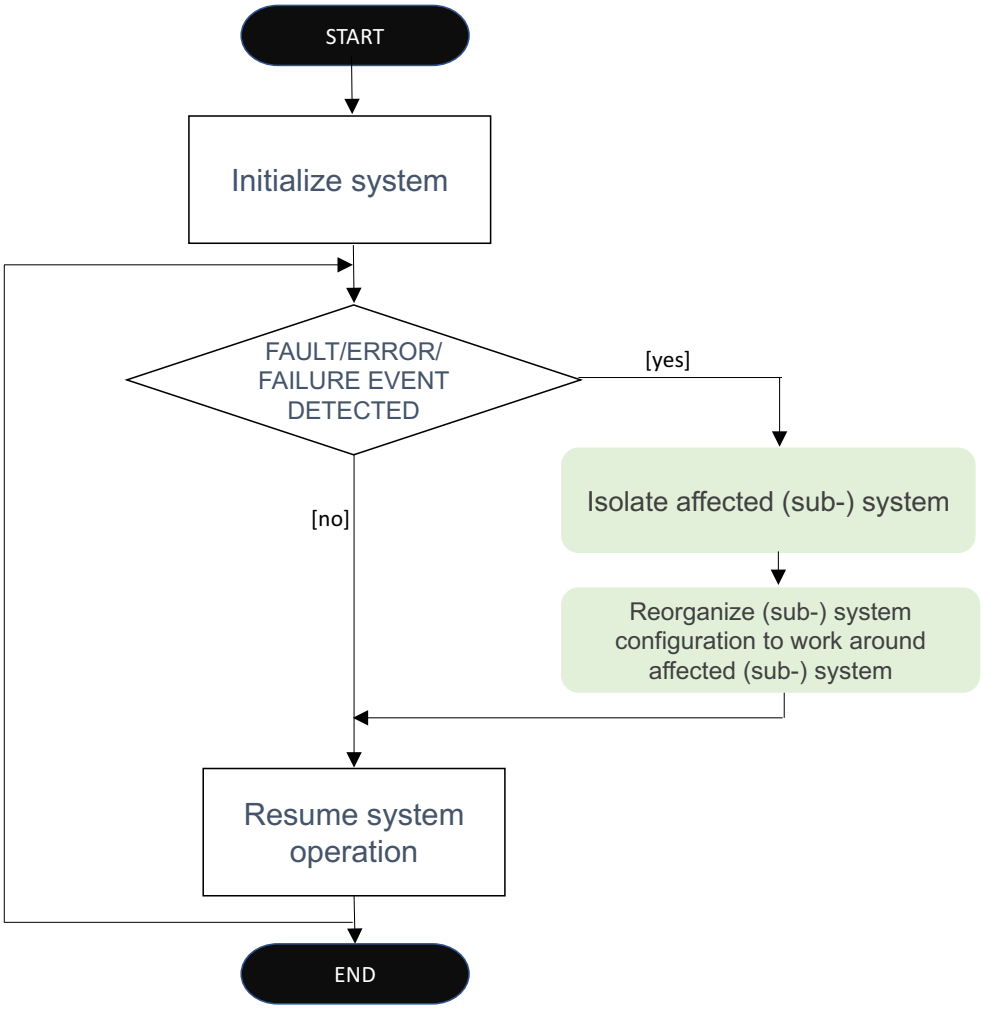

Figure 25. Restructure Pattern Flowchart

- The fault, error or failure in the system that the pattern handles must be detected; the pattern offers no implicit fault monitoring, prediction, or error/failure detection capability.
- The system may be partitioned into N interconnected sub-systems.
- The fault, error, or failure must not be persistent.

Forces:

- The system must be able to be partitioned into a sub-set of sub-systems that is functionally equivalent to the fault, error, or failure-free version of the system.
- The rejuvenation is often a slow process that requires substantial additional overhead to identify the part of the system affected by the fault, error or failure, and to selectively reinitialize the system, in addition to overhead incurred due to any lost work.

Solution: The `Rejuvenation` pattern provides isolation of the specific part of the system affected by an error/failure, and restoring or recreating the affected state such that the system may resume normal

operation. The pattern, whose structure is shown in Figure 26, enables the restoration of the correct operation of a (sub-) system impacted by a fault, error, or failure event.

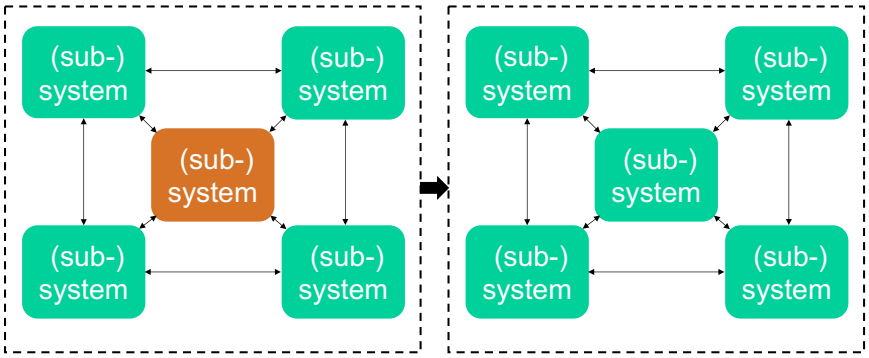

Figure 26. Rejuvenation Pattern

Capability: The pattern requires the system operation to be halted to identify the part of the system affected by the error/failure. As shown in the flowchart in Figure 27, only the affected part of the system is restored to ensure correct operation of the system.

Protection Domain: The protection domain of the `Rejuvenation` pattern spans the part of system whose state is restored.

Resulting Context:

- The rejuvenated system may not maintain the level of performance as before the occurrence of an event. Therefore, the pattern may result in the operation of the system in degraded condition, which incurs additional time overhead to the system.
- The overhead in terms of time to identify the specific state affected by the fault, error or failure, and restore it to known correct state may be considerable.

Rationale: The `Rejuvenation` pattern enables a system to tolerate a fault, error or failure by restoring the affected part of the system to known state that will ensure correct operation. Such targeted recovery prevents complete reset, or restructuring the system, both of which carry considerable overhead to the system operation.

Examples: The targeted recovery of data structures in system software, such as kernel modules, permits recovery without the need to reinitialize the complete system.

Related Patterns: The remaining reconfiguration patterns - the reinitialization and restructure patterns - are closely related since they all seek to isolate the error/failed state of the system and prevent it from affecting the remaining error/failure-free part of the system.

Known Uses:

- Algorithm-based recovery methods for data corruptions in structures used in numerical analysis problems use interpolation of neighboring data values to rejuvenate data values in error state. Such methods have been demonstrated in the context of the Hartree-Fock algorithm used in computational chemistry codes [149].
- The Mini-Ckpts framework enables HPC applications to survive fatal operating system failures/crashes by rejuvenating the OS using critical kernel structures, which are preserved in persistent memory [71].

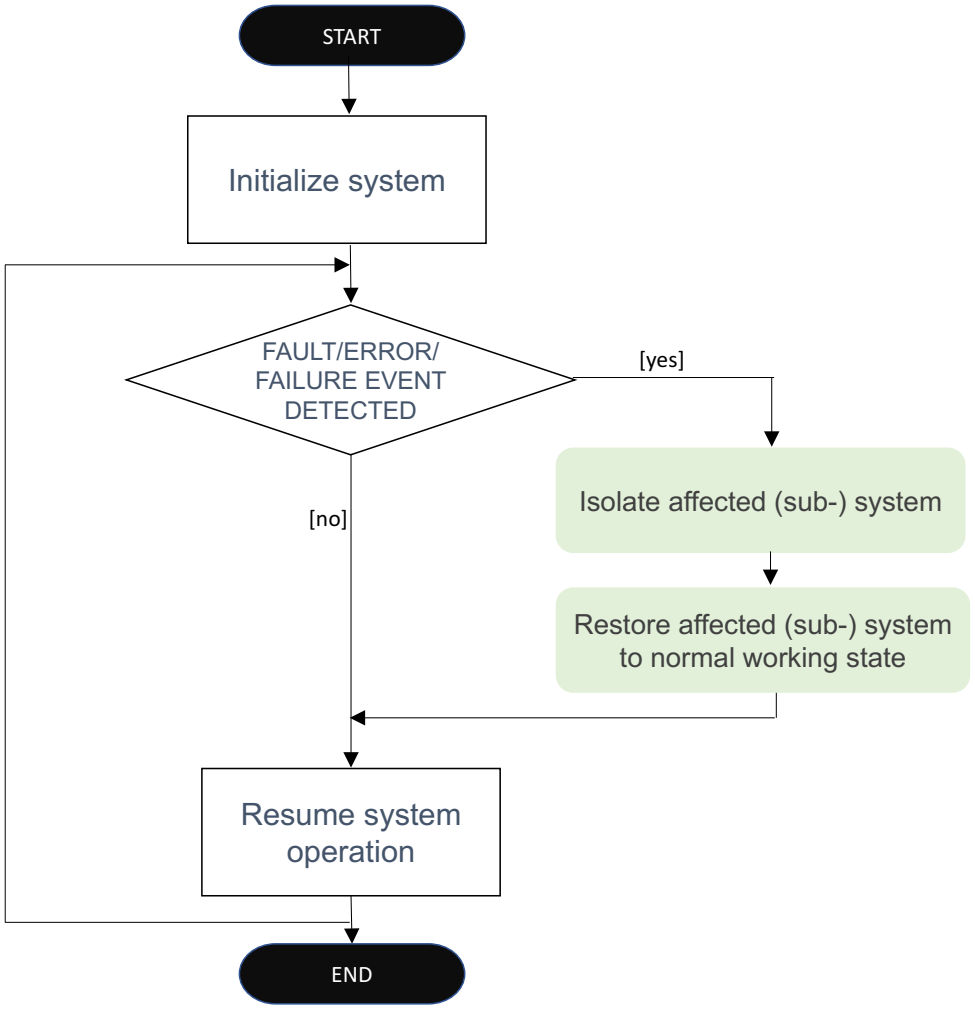

Figure 27. Rejuvenation Pattern Flowchart

7.4.5 Reinitialization Pattern

Pattern Name: Reinitialization Pattern

Problem: A fault, error or failure event affects a system to the extent that restoring correct operation is impossible.

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

- The system that 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 fault, error or failure in the system that the pattern handles must be detected; the pattern offers no implicit fault monitoring, prediction, or error/failure detection capability.
- The system may be partitioned into N interconnected sub-systems.
- The fault, error, or failure must not be persistent.

Forces:

- The system must be able to be partitioned into a subset of sub-systems that is functionally equivalent to the fault, error, or failure-free version of the system.
- The reinitialization is often a slow process that requires substantial additional overhead to reinitialize the system, in addition to overhead incurred due to lost work.

Solution: The `Reinitialization` pattern the alleviates the impact of the non-operation of a (sub-) system caused by the occurrence of a fault, error, or failure event. To recover from the error/failure, the pattern restores the system to its initial state. This causes system operation to *restart* and a pristine reset of state, which implicitly removes the effects of the error/failure. The pattern structure is illustrated in Figure 28.

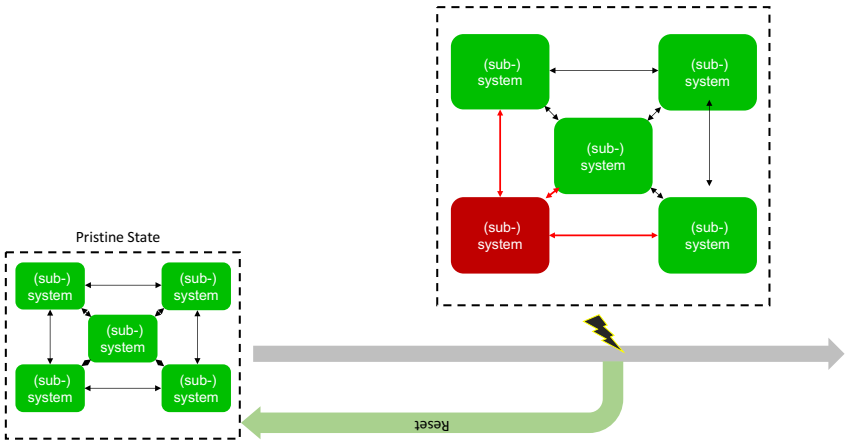

Figure 28. Reinitialization Pattern

Capability: The `Reinitialization` pattern performs a reset of the system state to restore pristine state before system operation is resumed. The flowchart in Figure 29 contains a graphical representation of these steps.

Protection Domain: Since the reinitialization causes reset of the system state, the protection domain of the `Reinitialization` pattern spans the complete system.

Resulting Context: The restoral of the system state to the initial state causes lost work, but guarantees the impact of the fault/error/failure is completely removed before service is resumed.

Rationale: The `Reinitialization` pattern is applied in conditions in which the recovery from the fault/error/failure instance is deemed impossible, or excessively expensive in terms of overhead to performance.

Examples: A node reboot is an instance of the `Reinitialization` pattern.

Related Patterns: The remaining reconfiguration patterns - the rejuvenation and restructure patterns - are closely related since they all seek to isolate the error/failed state of the system and prevent it from affecting the remaining error/failure-free part of the system.

Known Uses:

- Various cluster management software systems, such as the Cray Hardware Supervisory System (HSS) [89], 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.

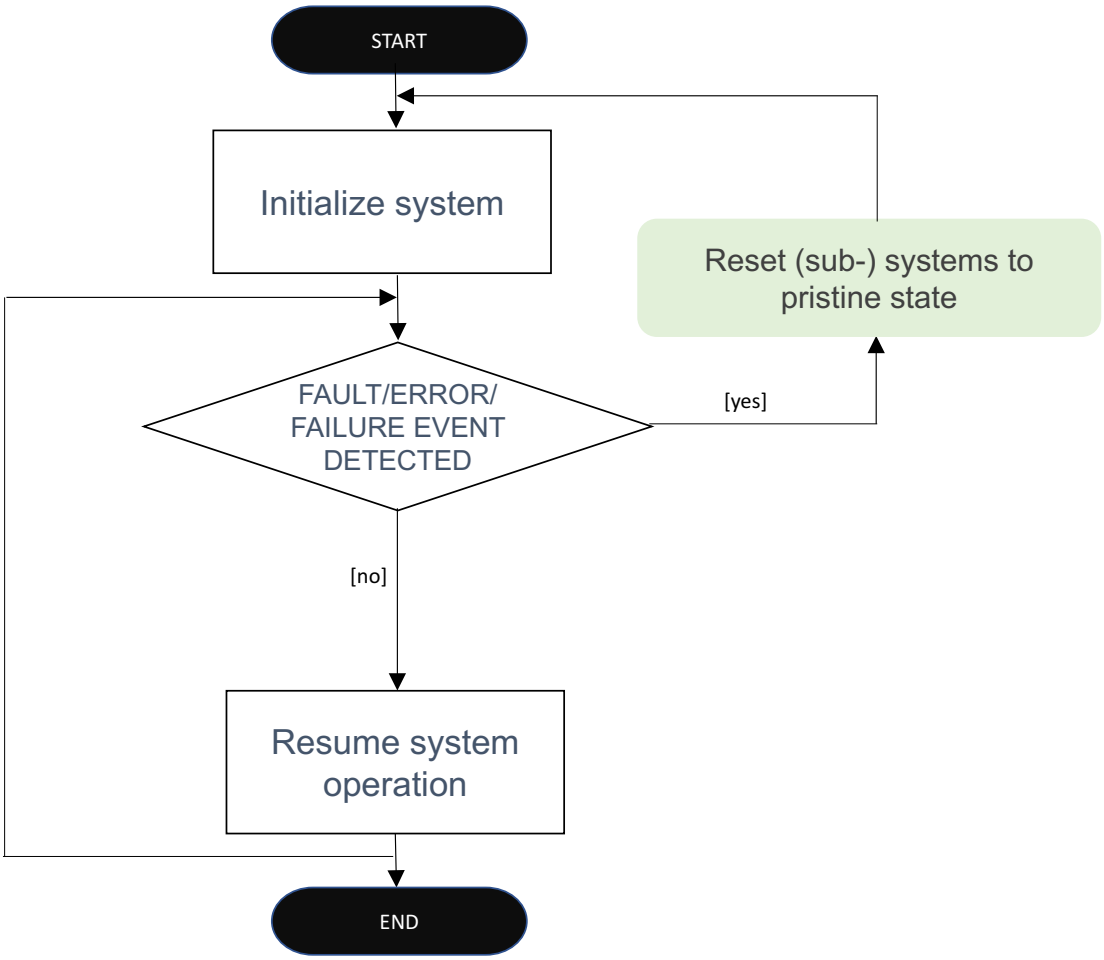

Figure 29. Reinitialization Pattern Flowchart

7.4.6 Rollback Pattern

Pattern Name: Rollback Pattern

Problem: The occurrence of an error or failure event prevents forward progress of a system.

Context: The pattern is a derivative of the `Checkpoint-Recovery` pattern. It applies to a system that 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 must also have the following characteristics:

- The error or failure in the system must be detected; the pattern offers no implicit error/failure detection capability.
- The system has well-defined intervals that enables the pattern to transition the system to a known correct interval in response to an error/failure.
- When the system encounters the same non-deterministic events with the same system state, and at the same location and timing, the system behavior is identical. The execution of the system between the intervals is deterministic.

- 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 is capable of exporting its current state and importing a new state.

Forces:

- The pattern requires stable storage to capture system state or to log events, which increase overhead in terms of resources required by the system.
- The process of compartmentalizing and capturing system state or logging interferes with system operation. The error/failure-free overhead penalty must be minimized.
- The extent of system state captured during the creation of a checkpoint incurs space and time overheads.
- The post-recovery state of the system must be as close as possible to an error/failure-free system state.
- The frequency of creation of checkpoints determines overhead. An increase in the checkpointing frequency increases system execution time, but reduces amount of lost work upon occurrence of an error/failure.
- The time interval required for the recovery of a system from a stable recovery point as well the time interval to create and commit a checkpoint to a stable storage system must be less than the mean time between errors/failures in the system to enable the system to make forward progress.

Solution: The `Rollback` pattern is based on the checkpoint-recovery solution, which entails the creation of snapshots of the system state and maintenance of these checkpoints on a stable storage system during the error/failure-free operation of the system. This pattern, whose solution is outlined in Figure 30, prevents fail-stop behavior in systems caused by errors in the system, or failure of (sub-) system(s) that eventually lead to catastrophic failure. The pattern may also be based on log-based protocols that use a combination of checkpointing and logging of non-deterministic events in the system. The defining feature of the `Rollback` solution is the backward recovery of the solution, i.e., based on a temporal view of the system's progress, the system state restored during the error/failure recovery process is a previous error/failure-free state of the system.

When the system design may be partitioned into of several sub-systems, the pattern must coordinate the process of checkpointing. The structure of the pattern is based on the coordination policy:

- *Coordinated rollback recovery protocol*: The approach requires the sub-systems to coordinate the process of creating checkpoints. The coordination enables a globally consistent checkpoint state, which simplifies the recovery.
- *Uncoordinated rollback recovery protocol*: The sub-systems each independently decide when to create their respective checkpoints. This approach has the potential to cause the full-system to propagate rollbacks to the initial system state to ensure that all dependencies are met; this condition is called the *domino effect*.
- *Communication-based rollback recovery protocol*: The protocol enables each sub-system to create *local* checkpoints, but periodically also enforces coordinated checkpoints between all sub-systems. Such a hybrid strategy helps avoid the *domino effect*.

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

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

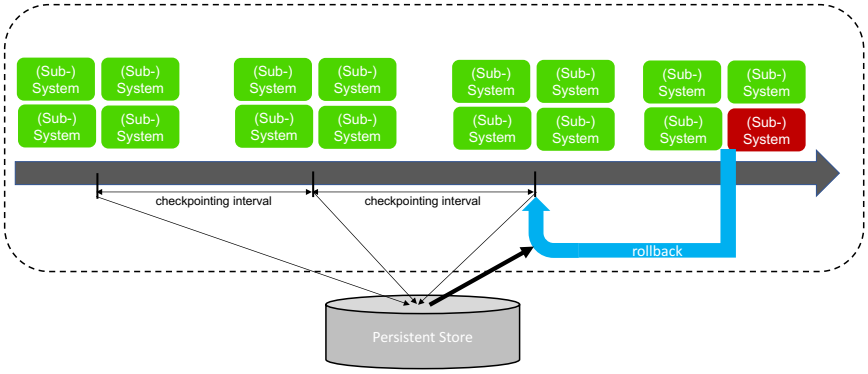

Figure 30. Rollback Pattern

Capability: The `Rollback` pattern enables the system to recover and resume operation from the point of occurrence of an error or a failure. The recovery of the system state typically leverages previously captured checkpointed state and/or logging information to recreate a stable version of the system state. The system is *rolled back* to the recovered state before resuming operation. These steps are shown in the flowchart in Figure 31.

Protection Domain: The protection domain for a `Rollback` pattern is determined by the extent of state captured during checkpoint operation and/or the number of system operations that can be recovered from the log of events. For failure recovery, the protection domain must include the state of the complete system. For error recovery, the protection domain includes the scope of system state captured at every checkpoint; error recovery is possible only when the location of the error is within the state that is checkpointed.

Resulting Context: The time overhead introduced by the application of the `Rollback` pattern during error-free operation depends on the frequency of taking checkpoints. The rollback leads to loss of work due to the need to recover the system from a previous version of the system state. The amount of work lost correlates with the frequency of the checkpoint operations. The worst-case scenario for recovery using this pattern is a rollback to the initial state of the system.

Rationale: The solution offered by this pattern is not dependent on either the type of error/failure, or the precise semantics of the error/failure propagation. The design effort and complexity in using this pattern in any system design is low. The impact of an error/failure on the system progress may be mitigated by restarting the system from a known version of the system state rather than endure complete loss of work performed until the occurrence of the error/failure event.

Examples: In the context of HPC systems, checkpoint and restart capabilities in the software layers, including various library-based and operating system-based solutions enable recovery from process errors/failures and rollback of the applications.

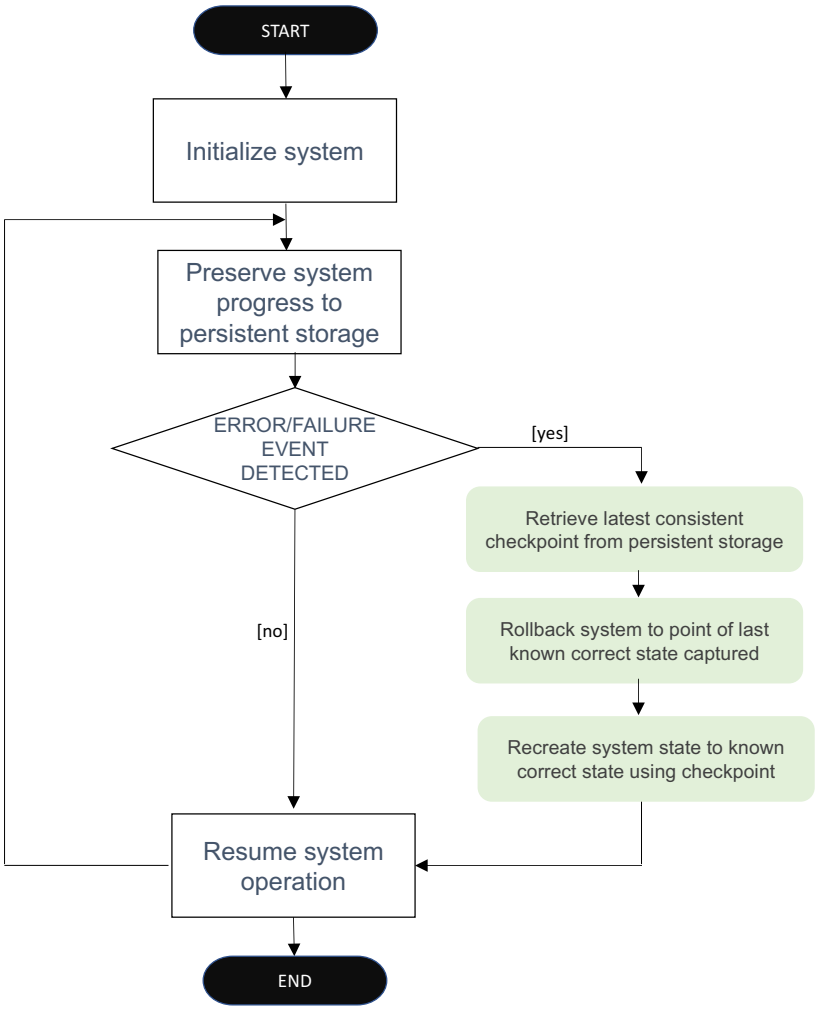

Figure 31. Rollback Pattern Flowchart

Related Patterns: The `Rollforward` pattern is closely related to the `Rollback` pattern. The key difference between the two patterns is the temporal relation between the recovered state and the error/failure state. In the `rollback` pattern the system is restarted from a recovered state that is based on a previous stable version of the system state.

Known Uses:

- The CoCheck checkpoint restart supports rollback recovery for a MPI implementation; for the coordination of the checkpoints, CoCheck uses a special process [145]. The Condor checkpoint/restart facility is enabled by the user by linking the program source code with the condor library [106].
- Message logging protocols have been implemented in OpenMPI to support faster failure recovery [27].

7.4.7 Rollforward Pattern

Pattern Name: Rollforward Pattern

Problem: The impact of an error or failure may prevent correct operation of a system, causing fail-stop behavior.

Context: The pattern is a derivative of the `Checkpoint-Recovery` pattern. It applies to a system that 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 must also have the following characteristics:

- The error or failure in the system must be detected; the pattern offers no implicit error/failure detection capability.
- The system has well-defined intervals that enables the pattern to transition the system to a known correct interval in response to an error/failure.
- When the system encounters the same non-deterministic events with the same system state, and at the same location and timing, the system behavior is identical. The execution of the system between the intervals is deterministic.
- 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 is capable of exporting its current state and importing a new state.

Forces:

- The pattern requires stable storage to capture system state or to log events, which increase overhead in terms of resources required by the system.
- The process of compartmentalizing and capturing system state or logging interferes with system operation. The error/failure-free overhead penalty must be minimized.
- The extent of system state captured during the creation of a checkpoint incurs space and time overheads.
- The post-recovery state of the system must be as close as possible to an error/failure-free system state.
- The frequency of creation of checkpoints determines overhead. An increase in the checkpointing frequency increases system execution time, but reduces amount of lost work upon occurrence of an error/failure.
- The time interval required for the recovery of a system from a stable recovery point as well the time interval to create and commit a checkpoint to a stable storage system must be less than the mean time between errors/failures in the system to enable the system to make forward progress.

Solution: The `Rollforward` pattern is also based on the checkpoint-recovery solution, which entails the creation of snapshots of the system state and maintenance of these checkpoints on a stable storage system during the error/failure-free operation of the system. The pattern structure is illustrated in Figure 32. The pattern must be able to create checkpoints of the system state and includes a stable storage system. When the system may be partitioned into modules, the checkpointing may be coordinated, uncoordinated, or communication-based. The pattern may employ log-based protocols use a combination of checkpointing and logging of non-deterministic events in the system. The forward recovery supported by the pattern enables the state of a system that experiences an error/failure to be recovered and restarted to the state it had reached right before the occurrence of the error/failure.

The `Rollforward` pattern must include a mechanism to facilitate restoral of the system from the last stable checkpoint up to the point of the error/failure event. 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. When the pattern uses logging protocols for the non-deterministic events, it may be based on *pessimistic*, *optimistic*, or *causal* protocols.
- 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 may permit the state to self-correct after restart.

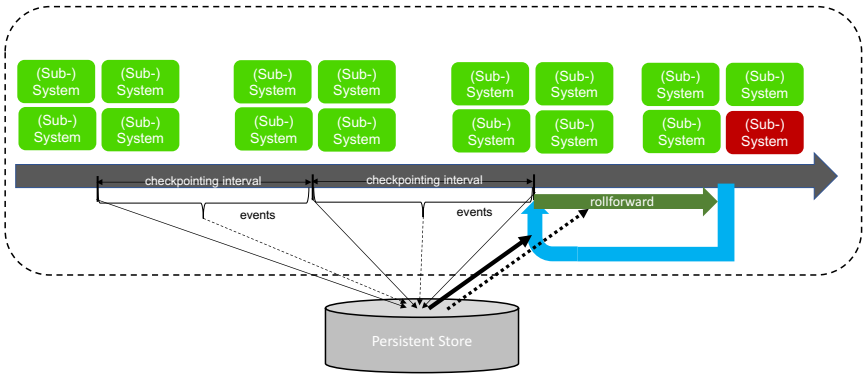

Figure 32. Rollforward Pattern

Capability: The `Rollforward` pattern enables the system to recover and resume operation from the point of occurrence of an error or a failure. The recovery of the system state typically leverages previously captured checkpointed state and/or logging information to recreate a stable version of the system state. The pattern uses the recovered stable version of the system to *roll forward* the system to the most recent state before the error/failure. The working of the pattern during error-/failure-free operation and during recovery is shown in Figure 33.

Protection Domain: The protection domain for a `Rollforward` pattern is determined by the extent of state captured during checkpoint operation and/or the number of system operations that can be recovered from the log of events. For failure recovery, the protection domain must include the state of the complete system. For error recovery, the protection domain includes the scope of system state captured at every checkpoint; the error recovery is possible only when the location of the error is within the state that is checkpointed.

Resulting Context: The `Rollforward` pattern enables the system to perform recovery and restart operation from the point of occurrence of an error or a failure, which prevents the loss of work due to the error/failure event. The rollforward capability may incur a time overhead to bring the system state to the most recent state before the error/failure.

Rationale: The solution offered by this pattern is not dependent on either the type of error/failure, or the precise semantics of the error/failure propagation, the design effort and complexity in using this pattern in any system design in low. The pattern also reduces the amount of lost progress due to the occurrence of an error/failure event.

Examples: In the context of HPC systems, checkpoint and restart capabilities in the software layers, including various library- based and operating system-based solutions that support replay of log events enable rollforward recovery. The use of algorithm-specific knowledge in scientific algorithms may also enable rollforward capabilities.

Related Patterns: The rollback pattern is closely related to the rollforward pattern. The key difference

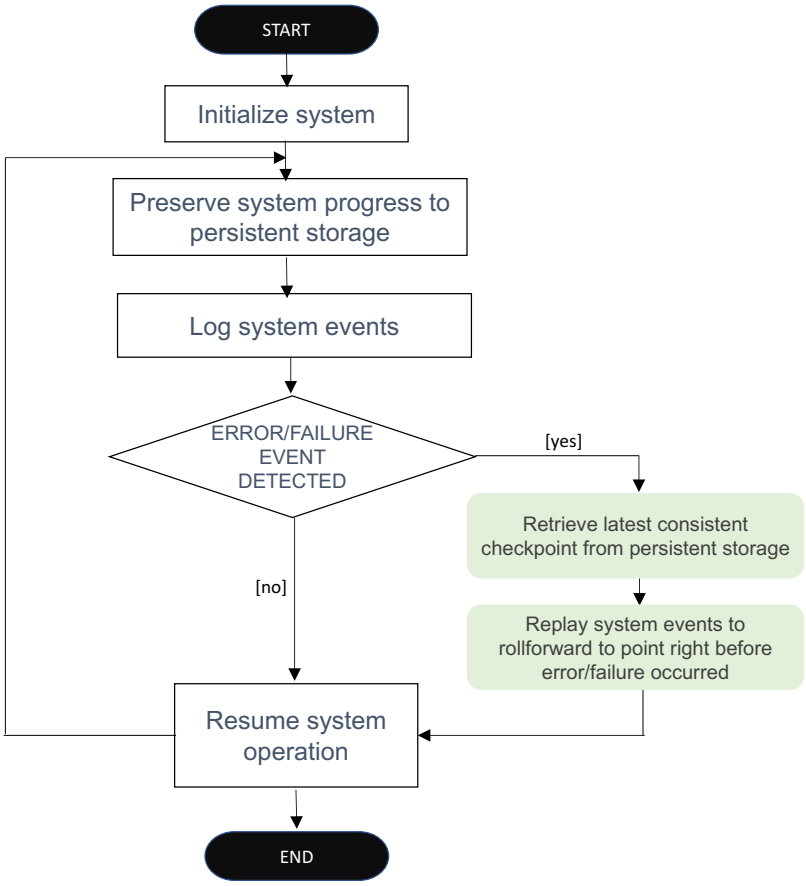

Figure 33. Rollforward Pattern Flowchart

between the two patterns is the temporal relation between the recovered state and the error/failure state. In the rollforward pattern the recovered state is based on a stable version of the system state immediately before the occurrence of the error/failure.

Known Uses:

- Global View of Resilience (GVR) [38] uses versioning of distributed arrays supports rollforward recovery based on application-specified mechanisms.

7.4.8 Forward Error Correction Code Pattern

Pattern Name: Forward Error Correction Code Pattern

Problem: The presence of information errors in a system's state affects its correct operation.

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

- The system must be 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 penalty in terms of time (increase in information access latency), and space (increase in resources required) independent of whether an error or failure occurs.
- The error/failure-free overhead penalty introduced by the replication of state must be minimized.
- The space overhead incurred by the additional redundant state information must be minimized.
- The number of errors 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` pattern detects and corrects information errors in the system state. The pattern consists of an encoder and a decoder module. In the simplest form (illustrated in Figure 34), the encoder repeats each symbol that represents the system state. The decoder module checks both instances of each symbol. In general, the encoder module accepts k information symbols that represent the system state and appends separately a set of r redundant symbols that are derived from the symbols representing system state. The output of the encoder module is a (n, k) code, in which n=k+r. The symbols that represent the system state are not modified by the encoder module. The decoder module permits the original state to be extracted from the FEC encoded state.

Capability: The availability of forward error correction codes in the system allows for recovery from corruption of some symbols that represent the system state. The FEC encoded redundant state information is a complex function of the original state. The pattern enables the recovery of the system from any errors in the representation of the system state. The working of the FEC pattern is shown in Figure 35.

Protection Domain: The protection domain of the pattern extends to the scope of the state that is encoded and decoded using the forward error correction code.

Resulting Context: The `Forward Error Correction Code` pattern permits the system to tolerate the corruption of some symbols in the system state. The design complexity of the pattern is based on the type and strength of the FEC. At runtime, every operation that affects the system state requires encoding/decoding operations, which adds time overhead to the operations. Additionally, the encoder/decoder modules must maintain the FEC symbols, which incurs a storage overhead for these symbols. The system state may be partitioned to mitigate the overhead of encoding and decoding the system state.

Rationale: The application of the FEC pattern allows the detection and correction of symbol errors in the system state. In many cases, the compensation of the symbol errors using the redundant information computed by the FEC enables reconstruction of the original state information.

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

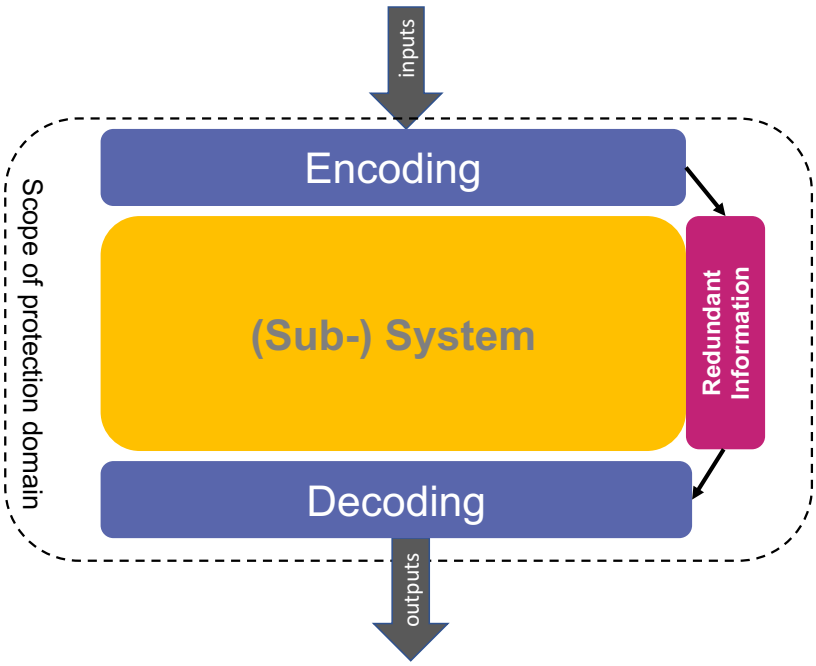

Figure 34. Forward Error Correction Code Pattern

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, Cyclic redundancy checks (CRC).

Related Patterns: The `Forward Error Correction Code` and the `N-modular redundancy` patterns are based on the inclusion of redundancy in the system to compensate for errors or failures. The `N-version` pattern is also based on inclusion of redundancy. However, the diversity in the `N-modular` pattern stems from the introduction of redundancy in the system's state unlike the `N-version` pattern, which uses independently implemented versions of the system's design to tolerate errors or failures.

Known Uses:

- 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 [8]. The FEC 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. These ECC bits support compensation of bit flip errors within the memory lines [115].
- Algorithm-based fault tolerance (ABFT) methods use schemes such as checksums that perform forward error correction at the application-level [82].

7.4.9 N-modular Redundancy Pattern

Pattern Name: N-modular Redundancy Pattern

Problem: The pattern solves the problem of dealing with errors, as well as partial or complete failures.

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

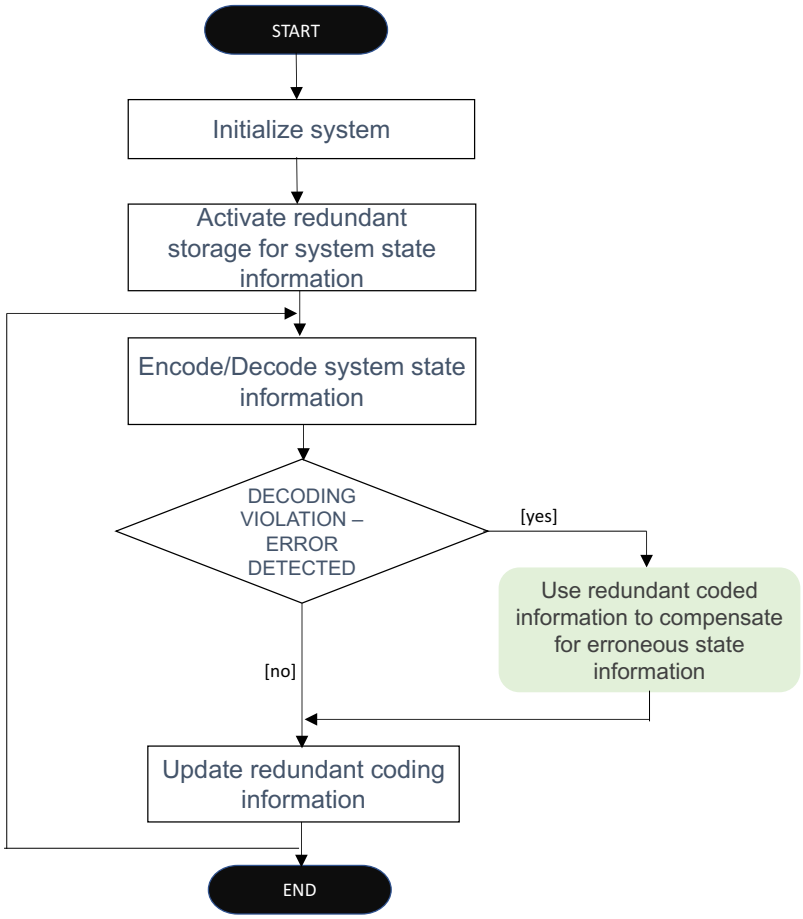

Figure 35. Forward Error Correction Code Pattern Flowchart

- The system must be 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.
- The cause of errors or failures experienced by the system may not be due to errors in the inputs.

Forces:

- The pattern introduces penalty in terms of time (increase in execution time), or space (increase in resources required) independent of whether an errors or failure occurs.
- The scope of the system that is replicated must be carefully selected; too large a scope incurs significant overhead, while too small a scope leads to incomplete protection for the system.
- The error/failure-free overhead in terms of time and space introduced by the replication of state must be minimized.

Solution: The `N-modular Redundancy` pattern enables the detection and correction of errors or failures in a system. The pattern defines modules for which it creates a group of N replicas as shown in Figure 36. Each module has a set of inputs, a well-defined scope of system operations, and a set of outputs. The n-modular redundancy entails replication of the module that requires protection. Each of the N modules of

the system exist simultaneously. The modules may be active at the same time (spatial replication), or may operate in succedent order (temporal replication), or may activate the redundant modules on-demand. The redundant modules in a system must be provided with the identical inputs. Therefore, the pattern may require replication of the input state.

When the modules are operated in spatial or temporal redundant mode, the pattern includes an independent module that compares their respective outputs to detect and potentially compensate for the presence of an error or a failure in one of the replica modules in the systems.

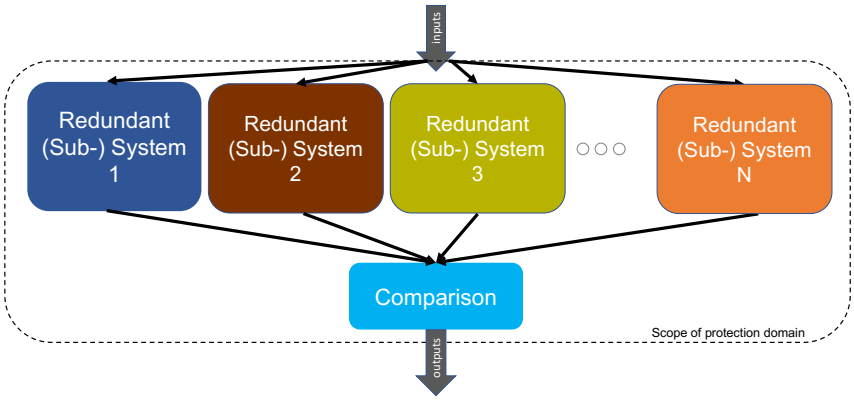

Figure 36. N-modular Redundancy Pattern

Capability: The working of the pattern is shown in the flowchart in Figure 37. The presence of replicated versions of the system state modules enable the following capabilities:

- *Fail-over*, which entails substitution of a replica in error or failed state with another replica that is error/failure-free.
- *Comparison*, which entails observing the likeness of each replica's outputs as means to detect the presence of an error or failure in either replica.
- *Majority voting* on the outputs produced by each replica system enables the detection of errors and failures, and filtering out the outputs that fall outside the majority.

Protection Domain: The protection domain of the pattern extends to the module that encapsulates the scope of the system state that is replicated.

Resulting Context: The design effort and complexity of replication of the system state requires low design effort since the replication entails creation of identical copies of the system state.

Rationale: The `N-modular redundancy` patterns enable systems to tolerate errors/failures by relying on the replicated versions of the system state to substitute a failed system, or to infer and compensate for errors/failures by comparing the outputs of the replicas.

Examples: The use of n-modular redundancy in various hardware and software components enables detection and correction of errors, or the compensation of failures in a system. The dual-modular redundancy for error detection and triple-modular redundancy for error detection and correction are widely used forms of this pattern in HPC environments.

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

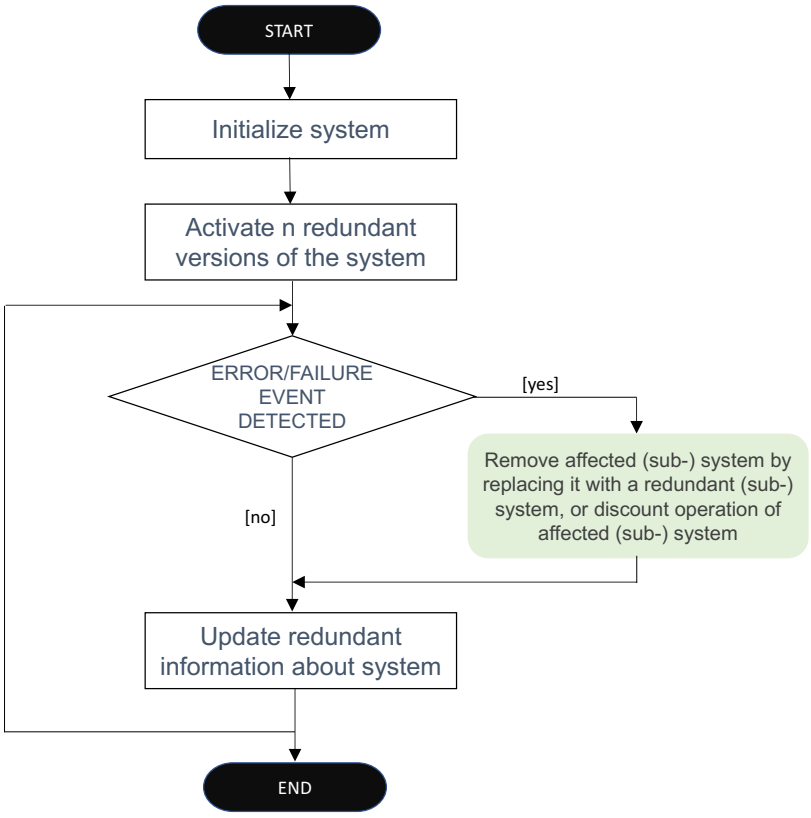

Figure 37. N-modular Redundancy Pattern Flowchart

Known Uses:

- 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 ranks. The MR-MPI [64], rMPI [68] and RedMPI [70] are known examples of the n-modular redundancy approach.
- Approaches that use n-modular redundancy for the compute nodes in a system have been evaluated and shown to improve the overall availability of a HPC system [62].

7.4.10 N-version Design Pattern

Pattern Name: N-version Design Pattern

Problem: Design bugs may manifest themselves during system operation causing incorrect operation or failure.

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

- The system has a well-defined specification for which multiple implementation variants may be designed.
- There is an implicit assumption of independence of between multiple variants of the implementation.

- The cause of errors or failures experienced by the system may not be due to errors in the inputs.

Forces:

- The pattern requires distinct implementations of the same design specification, which are created by different individuals or teams.
- The pattern increases the system complexity due to the need additional design and verification effort required to create multiple implementations.
- The error/failure-free overhead penalty due to disparity in the implementation variants must be minimized.

Solution: The `N-version Design` pattern supports the detection and correction of errors or failures in the behavior of the system that may occur due to design faults in the system. The pattern enables handling errors or failures due to Bohrbugs, although Heisenbugs may also be treated using this design pattern. The pattern entails creation of N independent versions of the system that are functionally identical, but designed independently. A majority voting logic is used to compare the results produced by each design version. Figure 38 illustrates the overall structure of this pattern solution.

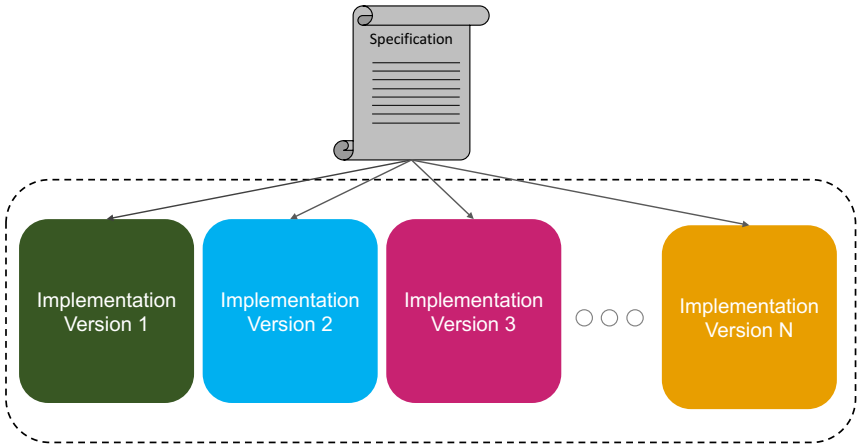

Figure 38. N-version Design Pattern

Capability: In this pattern, each of the N (N >= 2) versions of the designs are independently implemented, but the versions are functionally equivalent systems. The versions are operated independently and the critical aspects of the system state are compared to detect and correct errors/failures due to Bohrbugs or Heisenbugs. The flowchart in Figure 39 describes the working of this pattern.

Protection Domain: The protection domain extends to the scope of the system that is described by the design specification, of which multiple implementation variants are created.

Resulting Context: The extent to which each of the n design versions are different affects the ability of the pattern to tolerate errors/failures in the system. The use of the n-version design pattern requires significant design overhead in implementing and testing independent versions of a specification. Differences in the design may cause differences in timing in generating output values for comparison and majority voting - these differences incur overhead to the overall system operation.

Rationale: The intent behind applying this pattern is to eliminate the impact of design faults during the implementation of a system. Due to the low likelihood that different individuals or teams introduce

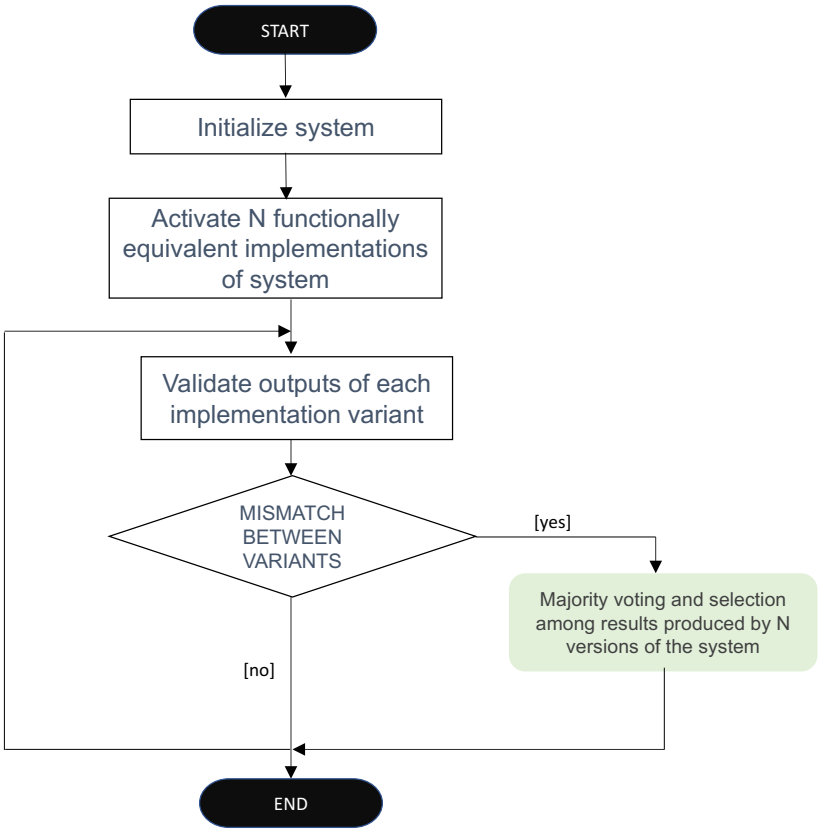

Figure 39. N-version Design Pattern Flowchart

identical bugs in their respective implementations, the pattern enables compensating for errors or failures caused by a bug in any one implementation of the same design.

Examples: Various versions of the same software are used for the detection of errors due to bugs in the implementation of either version.

Related Patterns: The pattern is similar to the n-modular redundancy pattern, which entails creating replica versions of the state associated with the pattern and accounting for the presence of errors/failures through majority voting. The key difference between the patterns is the independence of design between the replica versions 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. By testing large-scale applications with these separate implementations of MPI potentially exposes bugs in the MPI libraries.
- The DIVA processor architecture [16] 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.

7.4.11 Recovery Block Pattern

Pattern Name: Recovery Block Pattern

Recovery Block Problem: Flaws in the design due to human errors and/or faulty tools may cause errors or failures during system operation.

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

- The system has a well-defined specification for which multiple implementation variants may be designed.
- There is an implicit assumption of independence of between multiple variants of the implementation.
- The cause of errors or failures experienced by the system may not be due to errors in the inputs.

Forces:

- The pattern requires distinct implementations of the same design specification.
- The pattern increases the system complexity due to the need of additional design and verification effort required to create multiple implementations.
- The error/failure-free overhead penalty due to disparity in the implementation variants must be minimized.

Solution: The `Recovery Block` pattern supports the detection and correction errors or failures in the behavior of the system that may occur due to design faults in the system. The pattern is a flavor of the N-version design pattern, in which a recovery block is invoked when the result from the primary version of the system fails an acceptance test. The recovery block consists of an alternate implementation of the primary system, which is based on the same design specification. The relation between the primary system and its recovery block is shown in Figure 40.

With the use of the `Recovery block` pattern, the system is composed of functional blocks. Each block contains at least a primary design and exceptional case handler along with an adjudicator. If the adjudicator does not accept the results of the primary system, it invokes the exception handler subsystem. This indicates that the primary system could not perform the requested service operation.

Capability: The working of the `Recovery Block` pattern is illustrated in Figure 41. The pattern uses an acceptance test to validate the result produced by the primary system. If the result from the primary version passes the acceptance test, this result is reported and the test ends. If, on the other hand, the result from the primary version fails the acceptance test, another version from among the multiple versions is invoked and its result produced is checked by the acceptance test.

Protection Domain: The protection domain extends to the scope of the system that is described by the design specification, of which the recovery block implementation variant is created.

Resulting Context: The extent which the primary design and recovery block versions of the system specification are different affects the ability of the pattern to tolerate errors/failures in the system. The use of the `Recovery block` design pattern requires significant design overhead in implementing and testing independent versions of a specification.

Rationale: This pattern relies on multiple variants of a design which are functionally equivalent but designed independently. The secondary recovery block design is used to perform recovery, if the system implementation of the primary design produces an output that suggests the presence of an error/failure of the primary system. This determination is made by the adjudicator sub-system.

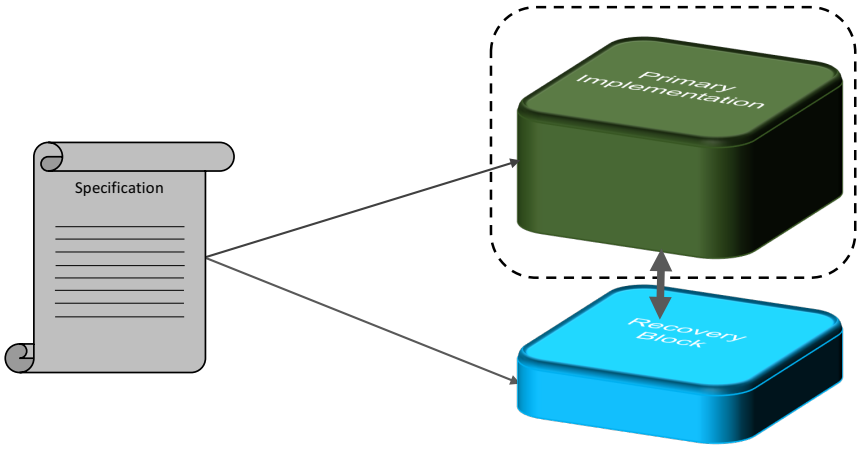

Figure 40. Recovery Block Pattern

Examples: Various application-based fault tolerance methods include verification routines that check for the validity of a computation.

Related Patterns: The significant differences in the recovery block approach from N-version programming are that only one version is executed at a time and the acceptability of results is decided by an adjudicator test rather than by majority voting.

Known Uses:

- Containment Domains [39] 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 provides language based implementation of the recovery block for use in C language programs.

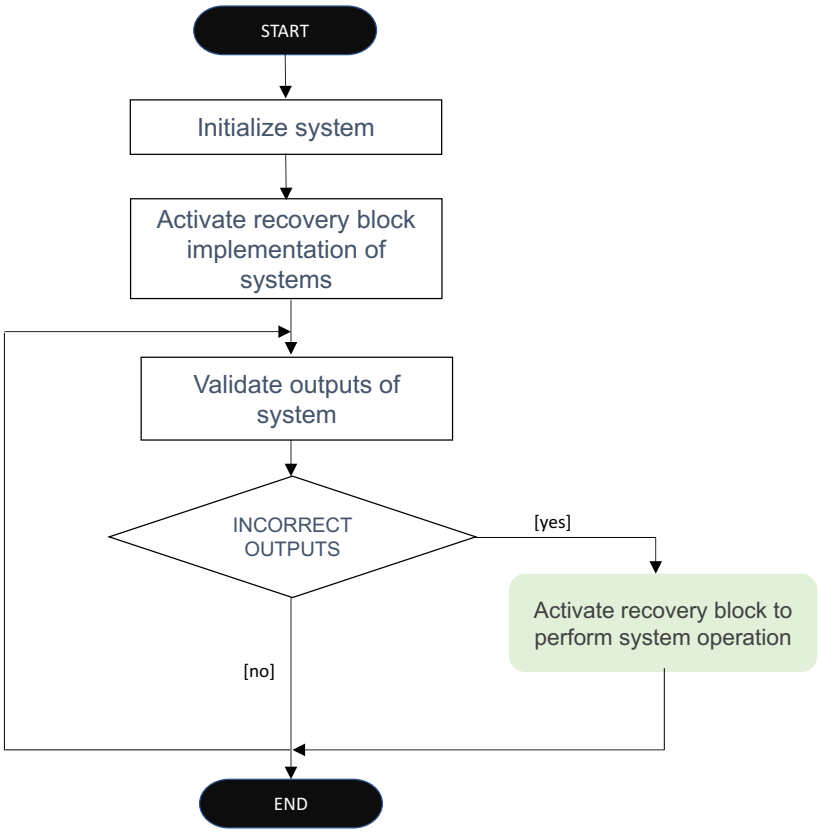

Figure 41. Recovery Block Pattern Flowchart

7.5 State Patterns

7.5.1 Static State Pattern

Pattern Name: Static State Pattern

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 42 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.

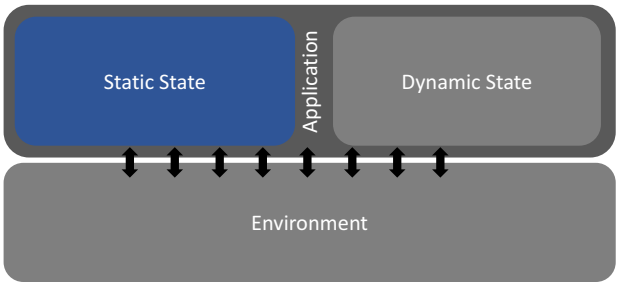

Figure 42. Static State Pattern

Capability: The static state outlives the process that creates/initializes it. The identification of this pattern follows the steps outlined in Figure 43. 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

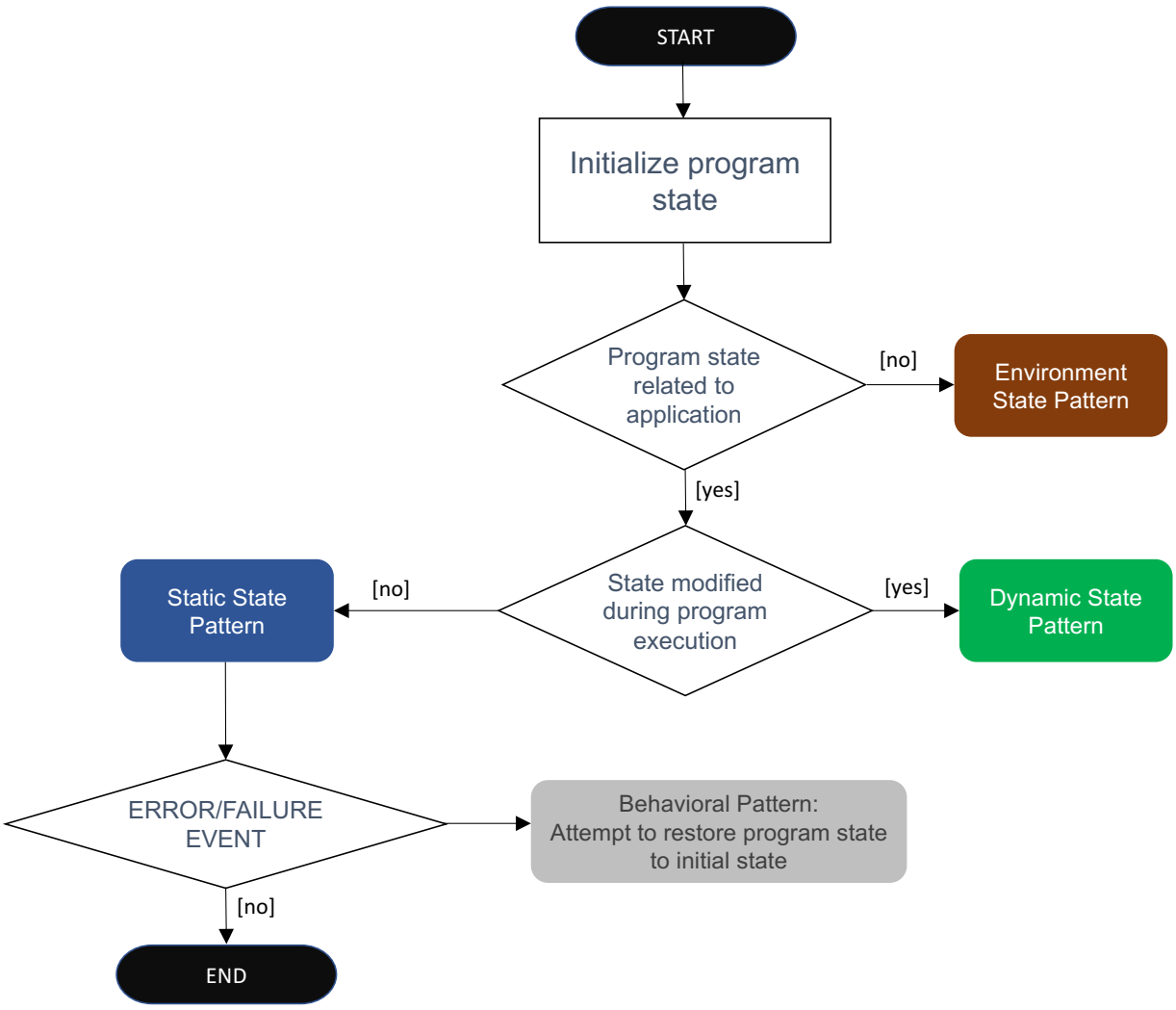

Figure 43. Static State Pattern Flowchart

future point in the program's execution. Therefore, the integrity of this aspect of a program's state is essential for correct execution.

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 [82].

7.5.2 Dynamic State Pattern

Pattern Name: Dynamic State Pattern

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 44. 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.

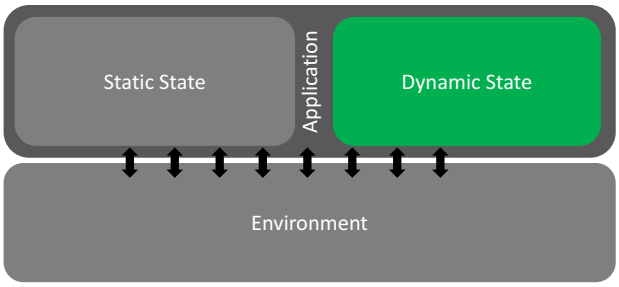

Figure 44. Dynamic State Pattern

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 45.

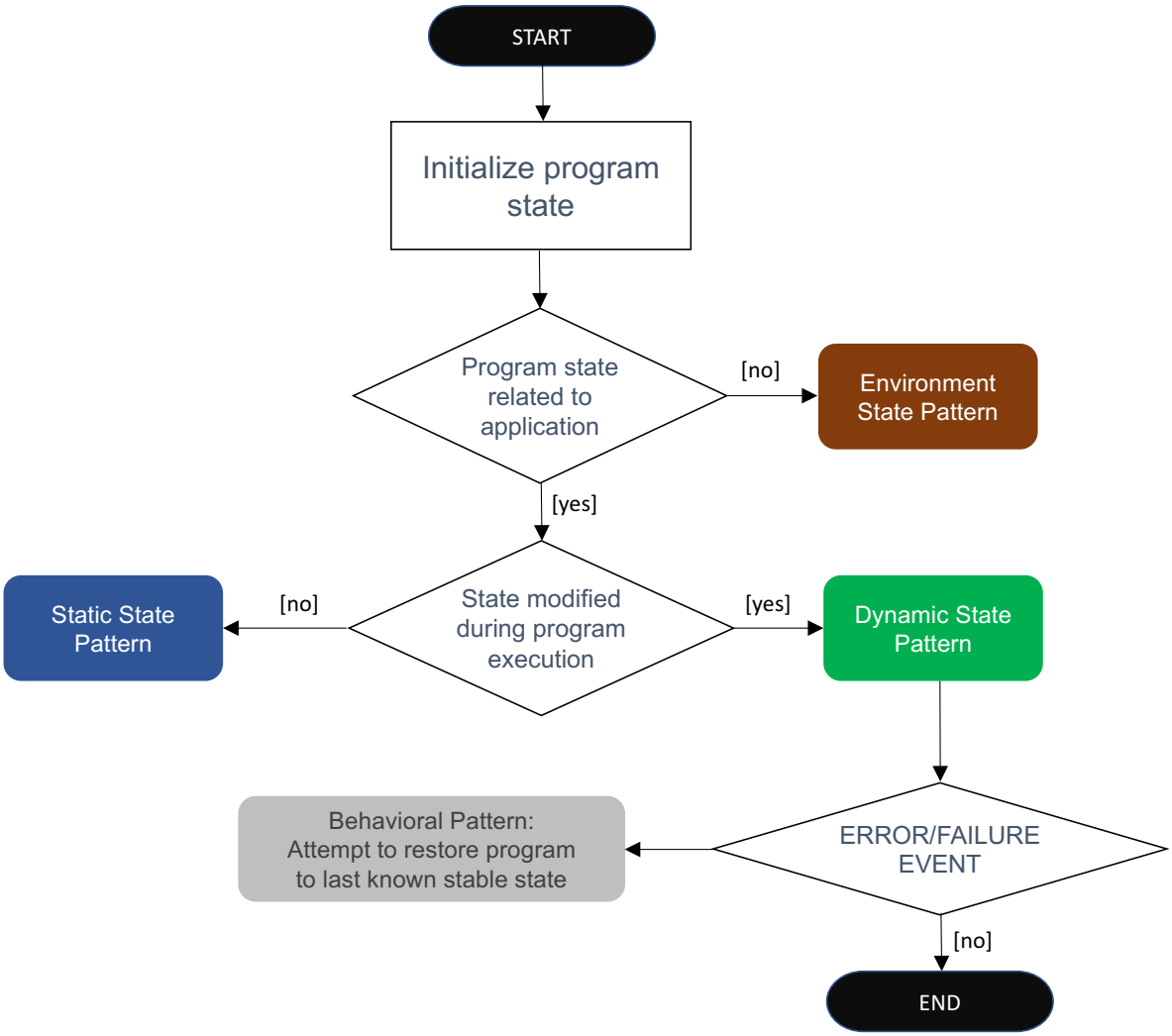

Figure 45. Dynamic State Pattern Flowchart

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.

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 [100]

7.5.3 Environment State Pattern

Pattern Name: Environment State Pattern

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 sub-systems in a system. Figure 46 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 for that are independent of the design of the HPC applications and their algorithms.

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 47 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.

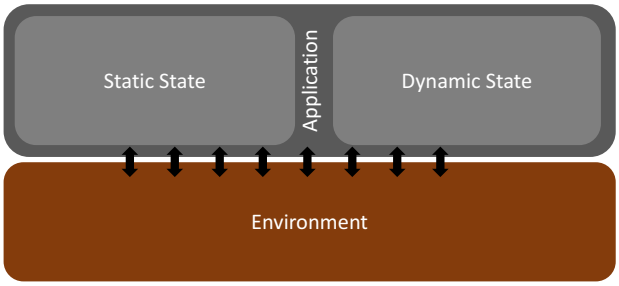

Figure 46. Environment State Pattern

Resulting Context: The pattern defines the scope of the state that support resource sharing, coordination and communication between the various sub-systems. 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 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 [71].
- The ULFM MPI provides recovery of the communication environment from the failure of processes by reconstructing the MPI communicator [23].

7.5.4 Stateless Pattern

Pattern Name: Stateless Pattern

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.

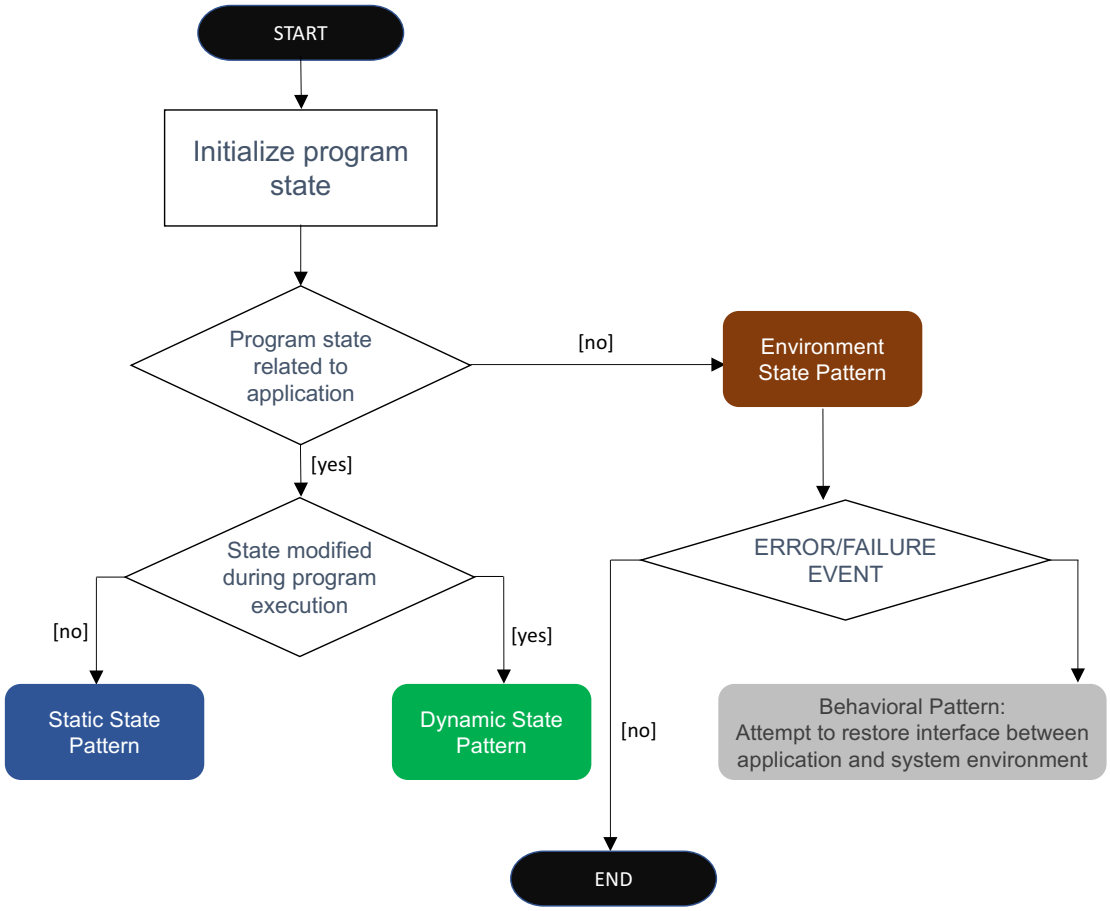

Figure 47. Environment State Pattern Flowchart

- 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 defines 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 are stateless [47].

8. Building Resilience Solutions using Resilience Design Patterns

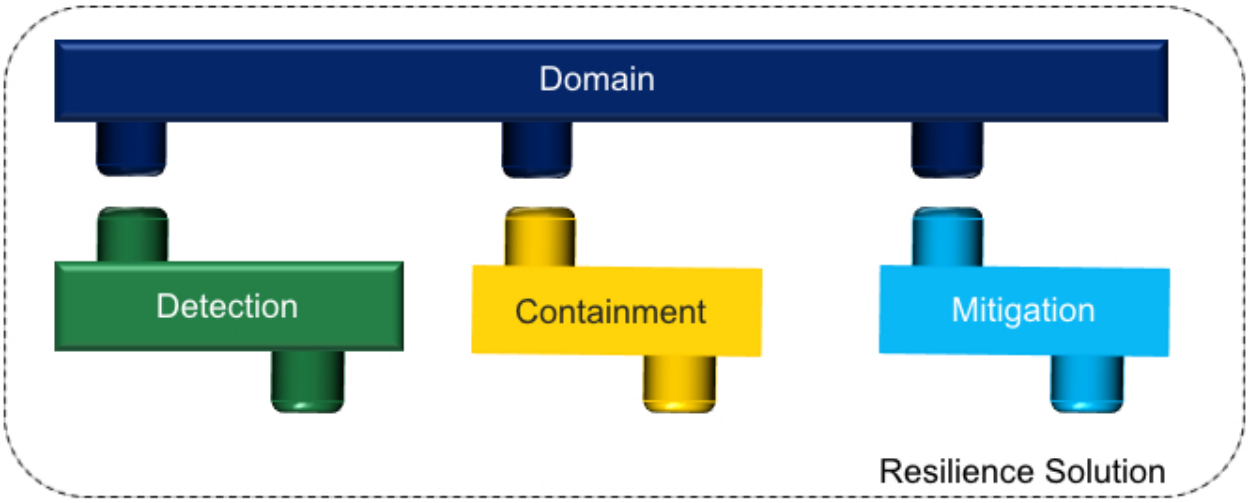

Figure 48. Elements of a resilience solution for HPC systems and applications

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.

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 48. 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.

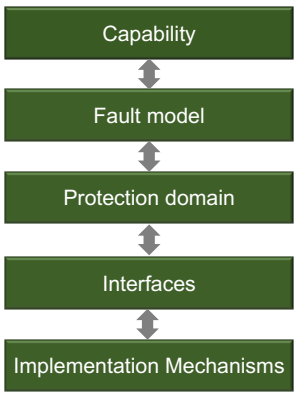

Figure 49. Design Spaces for construction of resilience solutions using patterns

In order to articulate a systematic method for customized designs, the framework is based on *design spaces* (Figure 49). 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 2. Types of Pattern Relations

Pattern 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 2 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 50.

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 2) 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.

10. 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.

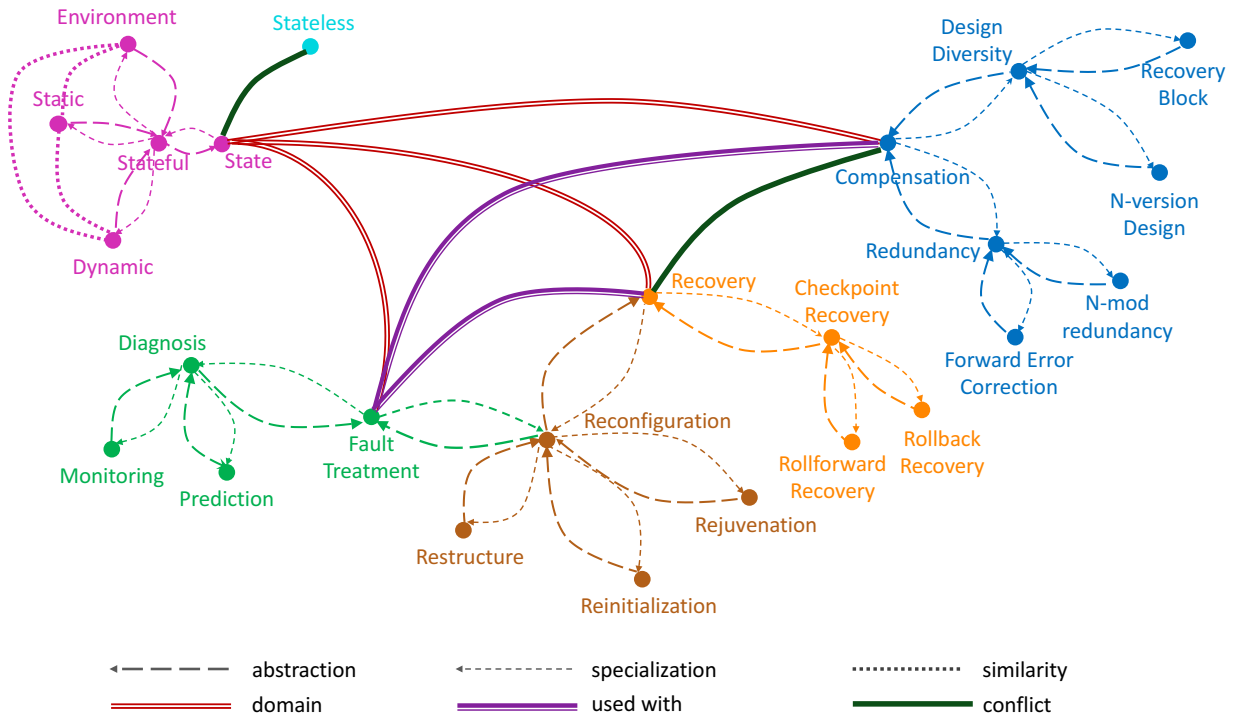

Figure 50. Resilience Pattern Language Representation

10.1 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:

- **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.

10.2 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.

11. Case Study: 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.

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.

Checkpoint and restart (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.

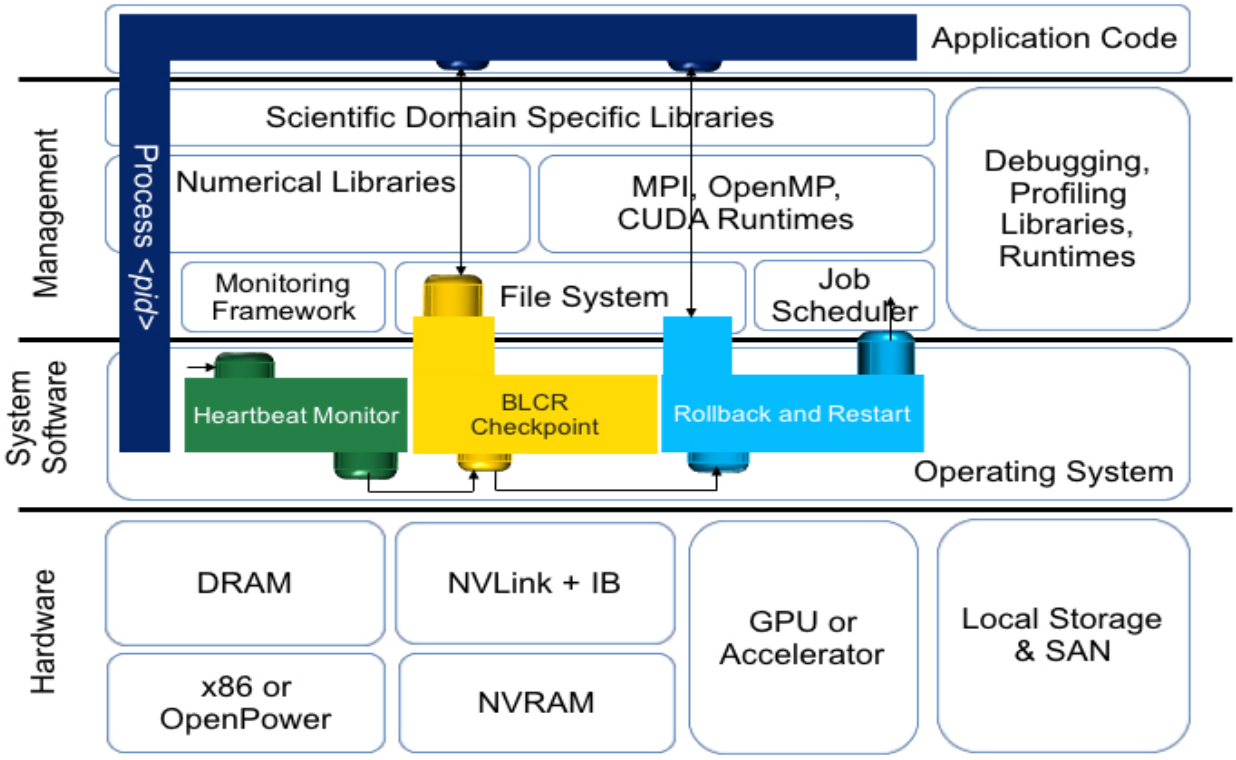

Figure 51. Resilience Solution Case Study: Checkpoint & Restart using BLCR

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 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 Roll-back 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 51, is implemented using the Berkley Lab's Checkpoint/Restart (BLCR) [57] 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 Roll-back 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.

12. Case Study: 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 a error or failuer 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.

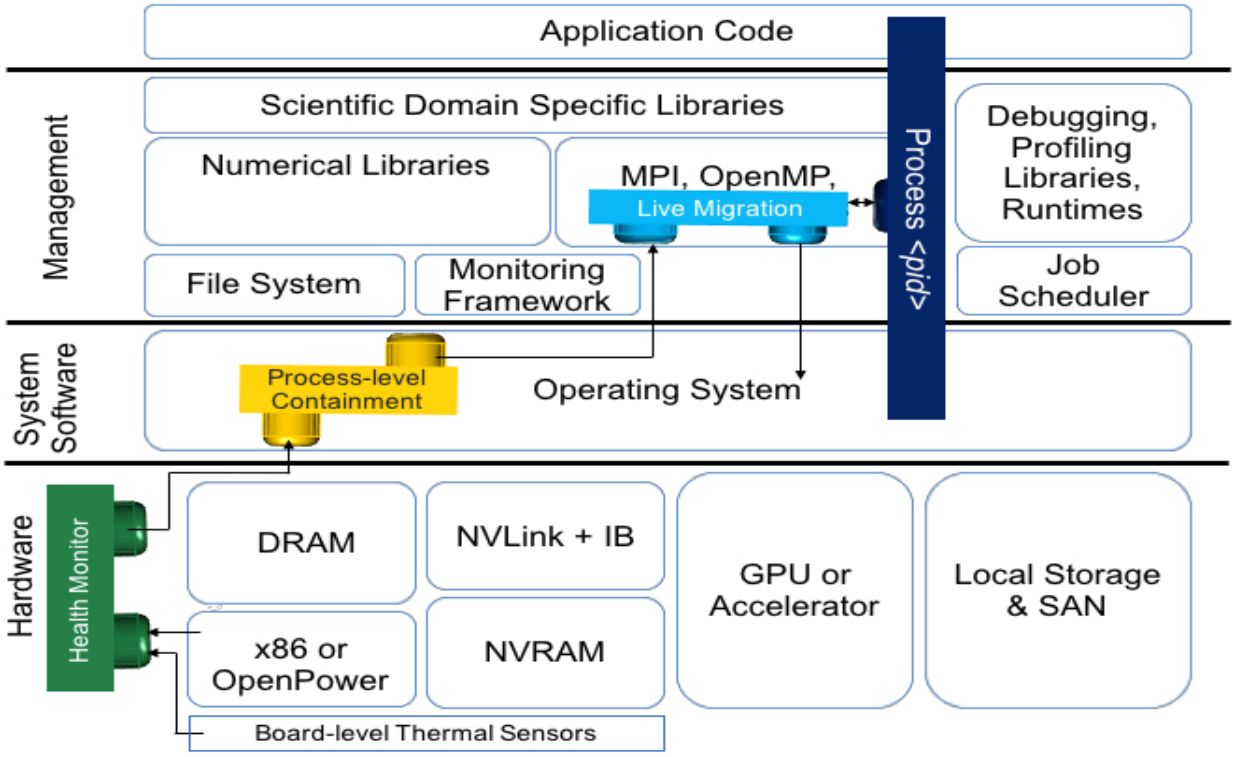

Figure 52. Resilience Solution Case Study: Process Migration

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 52. 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 Intelligent Platform Management Interface (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.

13. Case Study: 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.

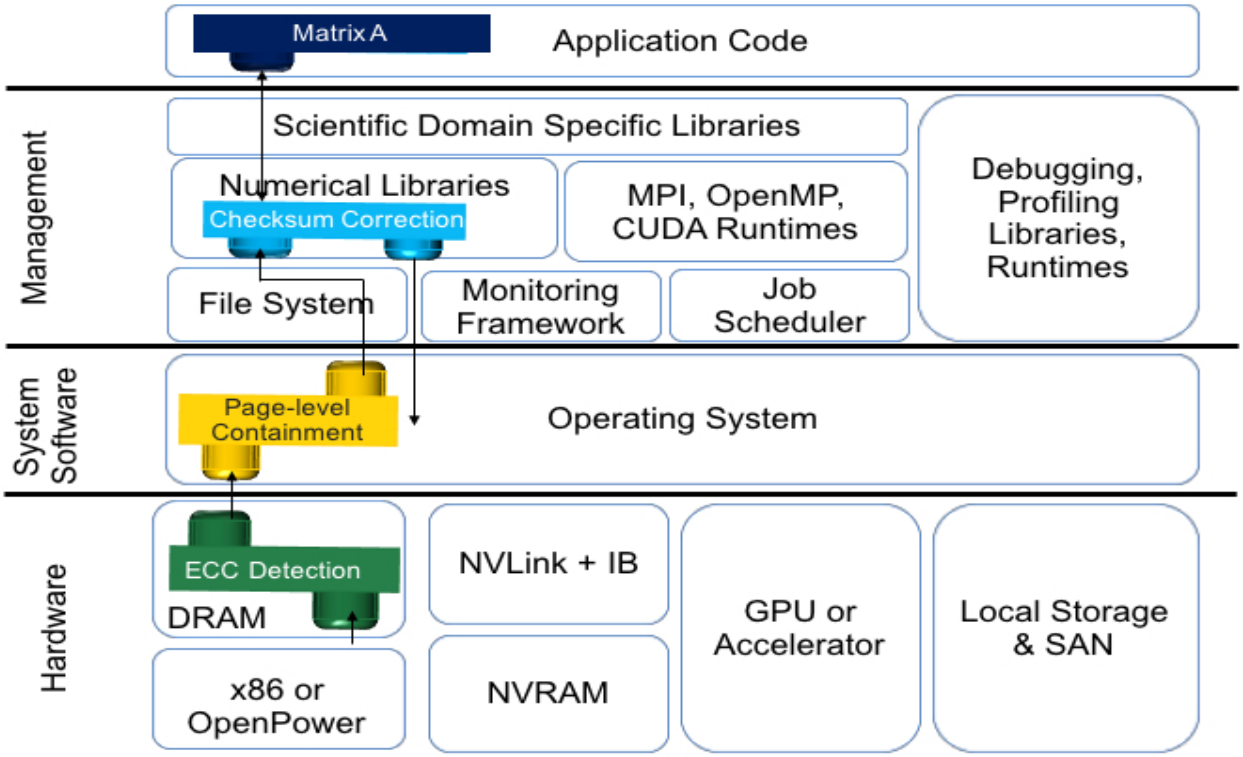

Figure 53. Resilience Solution Case Study: Cross-Layer Design using ECC with ABFT

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 error correcting codes (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 53. 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.

14. 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 accomodate 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.

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] Slurm workload manager. https://slurm.schedmd.com/.

[7] Lustre file system, high-performance storage architecture and scalable cluster file system, white paper. Technical report, Sun Microsystems, Inc., December 2007.

[8] Mellanox Infiniband FDR 56Gb/s for server and storage interconnect solutions. http://www.mellanox.com/related-docs/whitepapers/WP_InfiniBand_FDR.pdf, 2011.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, August 1977.

[14] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray XC series network, Cray Inc., 2014.

[15] T. Austin, V. Bertacco, S. Mahlke, and Yu Cao. Reliable systems on unreliable fabrics. *IEEE Design Test of Computers*, 25(4):322–332, 2008.

[16] 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.

[17] Algirdas Avižienis. Toward systematic design of fault-tolerant systems. *Computer*, 30(4):51–58, April 1997.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] Jan Borchers. *A Pattern Approach to Interaction Design*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[25] Shekhar Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, November 2005.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] Frank Buschmann, Kevin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture - Volume 4: A Pattern Language for Distributed Computing*. Wiley Publishing, 2007.

[32] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] PVFS2 High-Availability Clustering. PVFS2 development team. `www.pvfs.org/cvs/pvfs-2-8-branch-docs/doc/pvfs2/pvfs2-ha-heartbeat-v2.php`, June 2004.

[41] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[42] Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, March 2004.

[43] 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.

[44] 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.

[45] Intel Corporation Data Center Group. Intel xeon processor E7 family: Reliability, availability, and serviceability. 2011.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] Intel Hewlett-Packard NEC Dell. Intelligent platform management interface (IPMI), v2.0 specification. `http://www.intel.com/content/www/us/en/servers/ipmi/ipmi-home.html`, 2015.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] Peng Du, Aurelien Bouteiller, George Bosilca, Thomas Herault, and Jack Dongarra. Algorithm-based Fault Tolerance for Dense Matrix Factorizations. In *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 225–234, 2012.

[57] 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.

[58] 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.

[59] J. Elliott, K. Kharbas, D. Fiala, F. Mueller, K. Ferreira, and C. Engelmann. Combining partial redundancy and checkpointing for HPC. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 615–626, June 2012.

[60] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.

[61] 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.

[62] C. Engelmann, H. H. Ong, and S. L. Scott. The Case for Modular Redundancy in Large-scale High Performance Computing Systems. In *International Conference on Parallel and Distributed Computing and Networks*, pages 189–194, February 2009.

[63] Christian Engelmann. *Symmetric Active/Active High Availability for High-Performance Computing System Services*. PhD thesis, Department of Computer Science, University of Reading, UK, 2008. Thesis research performed at Oak Ridge National Laboratory. Advisor: Prof. Vassil N. Alexandrov (University of Reading).

[64] Christian Engelmann and Swen Böhm. Redundant execution of HPC applications with MR-MPI.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] 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.

[70] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, 2012.

[71] David Fiala, Frank Mueller, Kurt Ferreira, and Christian Engelmann. Mini-ckpts: Surviving OS failures in persistent memory. In *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, pages 7:1–7:14, 2016.

[72] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[73] Uwe Friedrichsen. No crash allowed - patterns for fault tolerance. In *The Conference for Java and Software Innovation (JAX)*, October 2012.

[74] 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.

[75] Object Management Group. Common object request broker architecture (CORBA) specification, version 3.3. `http://www.omg.org/spec/CORBA/3.3/`, 2012.

[76] 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.

[77] D. Hakkarinen and Zizhong Chen. Algorithmic cholesky factorization fault recovery. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–10, 2010.

[78] Robert Hanmer. *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007.

[79] 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.

[80] Jeffrey Heer and Maneesh Agrawala. Software design patterns for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):853–860, September 2006.

[81] Daniel Henderson. POWER8 processor-based systems RAS. August 2016.

[82] 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.

[83] 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.

[84] Saurabh Hukerikar and Robert F. Lucas. Rolex: Resilience-oriented language extensions for extreme-scale systems. *The Journal of Supercomputing*, pages 1–33, 2016.

[85] 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.

[86] 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.

[87] Joshua Hursey, Timothy I. Mattox, and Andrew Lumsdaine. Interconnect agnostic checkpoint/restart in open MPI. In *HPDC '09: Proceedings of the 18th ACM international symposium on High Performance Distributed Computing*, pages 49–58, New York, NY, USA, 2009. ACM.

[88] Cray Inc. Cray XT6 computing platform. `http://www.cray.com/Assets/PDF/products/xt/CrayXT6Brochure.pdf`, 2009.

[89] Cray Inc. Cray XE6 computing platform. `http://www.cray.com/sites/default/files/resources/CrayXE6Brochure.pdf`, 2010.

[90] Cray Inc. Cray XC40 computing platform. `http://www.cray.com/Assets/PDF/products/xc/CrayXC40Brochure.pdf`, 2014.

[91] 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.

[92] J.-Y. Jou and J.A. Abraham. Fault-tolerant FFT networks. *IEEE Transactions on Computers*, 37(5):548–561, 1988.

[93] Jing-Yang Jou and Jacob A. Abraham. Fault-tolerant matrix operations on multiple processor systems using weighted checksums. pages 94–101, 1984.

[94] 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.

[95] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*. John Wiley & Sons, Inc., New York, NY, USA, 2004.

[96] 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.

[97] I. Koren and S. Y. H. Su. Reliability analysis of n-modular redundancy systems with intermittent and permanent faults. *IEEE Transactions on Computers*, 28(7):514–520, July 1979.

[98] Israel Koren and C. Mani Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers, Burlington, MA, USA, July 2007.

[99] Gary Lakner and Brant Knudson. IBM system Blue Gene solution: Blue Gene/Q system administration. *IBM Redbooks*, May 2013.

[100] 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.

[101] Juan Leon, Allan L. Fisher, and Peter Steenkiste. Fail-safe PVM: A portable package for distributed programming with transparent recovery. Technical report, 1993.

[102] 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.

[103] Yinglung Liang, Yanyong Zhang, A. Sivasubramaniam, M. Jette, and R. Sahoo. Blue Gene/L failure analysis and prediction models. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 425–434, June 2006.

[104] 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.

[105] 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.

[106] 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.

[107] 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.

[108] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[109] 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.

[110] 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.

[111] Dennis McEvoy. The architecture of tandem's nonstop system. In *Proceedings of the ACM '81 conference*, New York, NY, USA, 1981. ACM.

[112] A. Mishra and P. Banerjee. An algorithm-based error detection scheme for the multigrid method. *IEEE Transactions on Computers*, 52(9):1089–1099, 2003.

[113] 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.

[114] 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.

[115] Todd K Moon. Error correction coding: Mathematical methods and algorithms. 2005.

[116] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for HPC with xen virtualization. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07, pages 23–32. ACM, 2007.

[117] 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.

[118] 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.

[119] 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.

[120] Nvidia. Dynamic page retirement, reference guide vr352. `https://docs.nvidia.com/deploy/pdf/Dynamic_Page_Retirement.pdf`, 2015.

[121] 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.

[122] 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.

[123] 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.

[124] Hoang Pham. *System Software Reliability*. Springer Series in Reliability Engineering. Springer, 2007.

[125] 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.

[126] 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.

[127] J.S. Plank, K. Li, and M.A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[128] A.L.N. Reddy and P. Banerjee. Algorithm-based fault detection for signal processing applications. *IEEE Transactions on Computers*, 39(10):1304–1308, 1990.

[129] 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.

[130] 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.

[131] 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.

[132] 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 Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 426–435, New York, NY, USA, 2003. ACM.

[133] 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.

[134] Titos Saridakis. A system of patterns for fault tolerance. In *Proceedings of 2002 European Conference on Pattern Languages of Programs (EuroPLoP)*, 2002.

[135] 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.

[136] J. Shalf, D. Quinlan, and C. Janssen. Rethinking hardware-software codesign for exascale systems. *Computer*, 44(11):22–30, November 2011.

[137] 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.

[138] 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.

[139] 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.

[140] 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.

[141] 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.

[142] 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 C 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*, 28(2):129–173, 2014.

[143] 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.

[144] G. Stellner. CoCheck: Checkpointing and Process migration for MPI. In *Proceedings of The Tenth International Parallel Processing Symposium*, pages 526–531, 1996.

[145] G. Stellner. CoCheck: checkpointing and process migration for MPI. In *Proceedings of International Conference on Parallel Processing*, pages 526–531, Apr 1996.

[146] Curtis E. Stevens. AT attachment 8 - ATA/ATAPI command set (ATA8-ACS). `http://www.t13.org/documents/uploadeddocuments/docs2008/d1699r6a-ata8-acs.pdf`, 2008.

[147] 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.

[148] 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.

[149] 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.

[150] Enrique Vargas. High availability fundamentals. *Sun Blueprints*, November 2000.

[151] 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.

[152] Jeffrey S. Vetter. *Contemporary High Performance Computing: From Petascale Toward Exascale*. Chapman & Hall/CRC, 2013.

[153] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in HPC environments. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 43:1–43:12, 2008.

[154] 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.

[155] 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.

[156] 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.

[157] 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.

[158] 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.

[159] 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.