computers

Article

Dependability Patterns: A Survey [†]

Ingrid A. Buckley [1] and Eduardo B. Fernandez [2,*]

[1] Department of Software Engineering, Florida Gulf Coast University, Fort Myers, FL 33965, USA; ibuckley@fgcu.edu
[2] Department of Electrical Engineering and Computer Science, Florida Atlantic University, Boca Raton, FL 33431, USA
[*] Correspondence: fernande@fau.edu

[†] This paper is a highly extended version of our paper published in 17th LACCEI: Proceedings of International Multi-Conference for Engineering, Education, and Technology Conference, 24–26 July 2019, Jamaica.

Abstract: Patterns embody the experience and knowledge of designers and are effective ways to improve nonfunctional aspects of software systems. Although there are several catalogs and surveys of security patterns, there is no catalog or general survey about dependability patterns. Our survey presented an enumeration of dependability patterns, which include fault tolerance, reliability, safety, and availability patterns. After defining classification groups and showing basic pattern relationships, we showed the references to the publications where these patterns were introduced and enumerated their intents. Another objective was evaluating these patterns to see if their descriptions are appropriate for a possible catalog, which would make them useful to developers and researchers. We found that most of them need remodeling because they use ad hoc templates or no templates. We considered some models from which we can derive patterns and methodologies that incorporate the use of patterns to build dependable software systems. We also provided directions for research.

Keywords: dependability; software patterns; reliability; safety; recovery; dependability patterns

Citation: Buckley, I.A.; Fernandez, E.B. Dependability Patterns: A Survey. *Computers* **2023**, *12*, 214. https://doi.org/10.3390/computers12100214

Academic Editor: Paolo Bellavista

Received: 13 September 2023
Revised: 11 October 2023
Accepted: 16 October 2023
Published: 21 October 2023

1. Introduction

Patterns can be effective for designing secure systems. Accordingly, security patterns have been studied extensively, and many publications about them exist, including surveys and catalogs [1,2]. However, there is relatively little work on the use of patterns for building dependable systems and no catalogs or general surveys, although they can have a similar design value. By dependability, we understand the ability to deliver service that can justifiably be trusted [3]. In this definition of dependability, we include reliability, safety, availability, fault tolerance, and maintainability. Sometimes security is also included as part of dependability [3,4], but we leave it out because, as indicated, it has been studied extensively on its own, e.g., [1,2,5–8]; Avizienis et al. in [3] do the same. Reliability is a property that allows some function, task, or service to behave as intended when required and measures the continuity of correct service. Reliability is particularly important in critical infrastructures and is a fundamental prerequisite of safety. Fault tolerance masks execution faults and is an important mechanism for obtaining reliability. Availability implies that a function, task, or service should be ready for service when needed. Safety is a property that allows some function or service to operate in a manner that does not cause harm to humans or damage to valuable systems. Maintainability measures the ability of a system to be modified or repaired (see more detailed definitions in Section 2.1).

We survey patterns that can be used to build or understand dependable systems. A pattern is an encapsulated solution to a recurrent problem that solves a specific problem within a given context [9,10]. Patterns have proven to be useful in the development of suitable-quality systems because they provide reusability of ideas and a systematic approach for designing, implementing, and evaluating complex software systems. In this survey, we enumerate the patterns intended to build dependable systems we have been able to find, but this is not a systematic survey. We emphasize the types and use of patterns, not striving for completeness or counts of papers. Note that we have identified patterns, not articles about patterns, as in most systematic surveys. For these patterns, we analyze the quality of their descriptions using a set of quality criteria [2]. We identify patterns that are not sufficiently elaborated and need to be completed before they can be useful for developers. We also consider methodologies that use these patterns as artifacts to build dependable solutions and some metamodels or architectures that can be useful for those using patterns and from which we can derive new patterns. Finally, we mention related tactics and arguments. Our contributions are as follows:

- Survey and enumerate dependability patterns. We tried to see what is available and detect if they cover a broad set of topics;
- Classify these patterns in groups and relate them using pattern diagrams. This classification is important to see the coverage of topics and to understand how these patterns relate to each other;
- Evaluate the suitability of existing patterns to build dependable systems. We found that many patterns need completion or refactoring to have a practical catalog;
- Survey methodologies for systematic development of dependable systems. These are important to use the patterns in building dependable systems;
- Provide a list of directions for research. These are intended to guide future work and indicate aspects that need more work.

This article is organized as follows: Section 2 provides background on patterns and dependability. Section 3 classifies patterns into groups for their convenient enumeration and shows pattern diagrams to relate the main patterns within a group. Section 4 surveys reliability, fault tolerance, and availability patterns, while Section 5 surveys safety patterns. Section 6 enumerates specialized patterns, including hybrid approaches, auxiliary, recovery, and audit patterns, as well as patterns for web services and clouds; Section 7 is about metamodels, architectures, and reliable system development methodologies. Section 8 provides a list with brief descriptions of the patterns in the survey. Section 9 evaluates the most significant patterns found in our survey along two dimensions: dependability properties and quality indicators. Section 10 provides ideas for research, and Section 11 presents our results, which are evaluated in Section 12. Section 13 considers related work, and we finish with conclusions in Section 14.

2. Background

2.1. Patterns

As indicated above, a pattern is a solution to a recurring problem in a specific context. Software patterns are categorized as analysis [11], design [10], architecture [9], and security patterns [1]. All patterns that describe solutions to design problems can be called design patterns as opposed to patterns that describe problems, such as how to teach a course; however, the books on design patterns only consider code problems. *Abstract Security patterns* (ASPs) describe a conceptual security mechanism that realizes one or more security policies able to control (stop or mitigate) a threat or comply with a security-related regulation or institutional policy [5]. Some of the published security patterns are ASPs. Patterns are described using a template composed of a set of sections. A problem section describes a problem and the forces that constrain and define guidelines for its solution, e.g., “the solution must be transparent to the user”. Pattern solutions are usually described using modeling languages such as the Unified Modeling Language (UML), maybe combined with formal languages such as the Object Constraint Language (OCL) [12]. UML diagrams used in pattern solutions may include class, sequence, state, and activity diagrams. A set of consequences indicates how the pattern solved the specific problem and what are the advantages and disadvantages of using it, i.e., how well the forces were satisfied by the solution. An implementation section provides hints on how to use the pattern in an application or how it has been used in industrial products. A section on “known uses” lists real systems where this solution has been used previously, i.e., patterns are abstractions of good practices. A section on related patterns indicates patterns that complement or provide alternative solutions to the one in this pattern. A *pattern diagram* shows relationships between patterns; an arc from pattern A to pattern B is labeled with the indication of the contribution from pattern A to B. We use UML operations generalization and composition to relate patterns.

A pattern embodies the knowledge and experience of software developers and can be reused in new applications; well-designed patterns implicitly apply suitable design principles. Patterns are also useful for communication between designers and to evaluate or reengineer existing systems. Patterns can describe hardware, physical entities, organizational processes, and combinations of these. Pattern solutions are suggestions, not plug-ins or software components. A *compound* pattern is composed of two or more simpler patterns. Patterns are just ways to represent design constructs in a system that correspond to the best uses of a mechanism or procedure. They are not theoretical constructs; they are often described using UML. Patterns can be added to the architectures of systems under construction according to their textual specifications, but the large number of patterns (security and dependability) is confusing for developers; a better approach is to follow the stages of a secure development methodology [2,8].

A *Reference Architecture* (RA) is an abstract software architecture based on one or more domains with no implementation aspects [13]. An RA can define the fundamental concepts of a system expressed as patterns and the relations between them. An RA should be reusable, extendable, and configurable; that is, it is a kind of composite pattern for whole architectures, and it can be instantiated into a concrete software architecture by adding platform aspects. In addition to class and sequence diagrams, an RA may include a set of use cases (UC) and a set of Roles (R) corresponding to its stakeholders (actors). After adding security patterns to mitigate identified threats in an RA, we have a *Security Reference Architecture* (SRA).

A formal specification is a mathematical description of software or hardware that may be used to prove that a system is correct with respect to its specifications or to define or evaluate its properties. UML is considered a semi-formal notation, while OCL is a formal notation or language. Tactics are reusable design strategies that help achieve a particular quality attribute, e.g., reliability [14]. Tactics do not provide detailed realizations but can be realized using patterns [6,15]. Arguments are useful to assure systems; security arguments are used to demonstrate how someone can reasonably conclude that a system is acceptably secure from the evidence available [16]. They can be described in free form or as patterns.

2.2. Dependability

We use the definitions of [3] for the common terms used in this field. A *system* is an entity that interacts with other systems; *functions* indicate what a system is intended to do or what services it provides. A system has *structure* and *components*. A system that depends on another is said to *trust* that system. Systems have internal and external states. Reference [17] is a clear presentation of the typical structures of hardware and software fault-tolerant architectures. A *fault* is a defective value in the state of a component or a flaw in the design of a system. A fault can be classified by its duration, nature, and degree and can be external or internal. An *error* is a defective value in the state of a component or in the design of a system, which is the manifestation of a fault. Typically, local faults affect single components, whereas global faults affect multiple components. A *failure* is a situation that occurs when the delivered system service deviates from its correct value and is produced as a result of an error. Dependability is typically quantified probabilistically by measuring the mean time a system is operational or failure-free [18]. As indicated above, reliability is a property that allows some function, task, or service to behave as intended when required and measures the continuity of correct service. Safety is the absence of catastrophic consequences on the users and the environment. Reliability is a fundamental condition for safety. As we indicated in the Introduction, security has been included in dependability because it has some common aspects (integrity, availability), but the fact that security defends against the intentional actions of a determined adversary makes it different.

Policies are high-level guidelines defining how an institution conducts its activities in its business, professional, economic, social, and legal environment. Policies are common in security but are not generally used for dependability. For our purposes, a policy is a guideline or direction about how we can handle a dependability problem or how we want the system to be designed. A policy is realized by some *mechanism*. Possible policies include:

Error Detection—This implies constant checking of the state of a system to ensure that specifications are being met. This is a fundamental function because we cannot handle an error if we do not know that it has happened.

Error Masking—Concealing the effect of errors to prevent a failure of the system if a fault occurs.

Fault Containment—Faults should be contained within some specific execution domain, which prevents error propagation across components or system boundaries.

Fault Diagnosis—Determining where a detected error has occurred and possibly the type of error.

Fault Repair—Implies the elimination, replacement, or bypassing of a component.

Graceful degradation—This policy is essential in systems where, in the event of a failure, some functionality should remain. If the system's operating quality decreases, the decrease should be proportional to the severity of the failure.

Safety assertions provide policies that indicate what conditions should be true to achieve safety (avoid hazards).

A policy is realized by one or more dependability patterns. Conversely, dependability patterns are designed to realize at least one or more of these policies.

3. Classification of Dependability Patterns

Figure 1 shows a UML class diagram showing the relationships between dependability problems and the types of patterns that handle them. Note that a fault can produce several errors [19], and the same error can be the result of different faults. Figure 1 also shows that Policies handle Failures and Hazards (the asterisks indicate a many-to-many relationship, read in the direction of the arrow), that dependability patterns realize policies, that auxiliary patterns are parts of dependability patterns (the dark rhomboid indicates strong aggregation), and that hybrid patterns are included in the group of dependability patterns but also concern security. The figure also classifies (indicated by the generalization symbol of a white triangle) dependability patterns in the following categories:

Reliability Patterns—They describe solutions to improve the general reliability of systems.

Fault Tolerance Patterns—They are a subset of the reliability patterns, and they describe solutions that use redundancy to mask the effects of errors.

Availability Patterns—They describe solutions that avoid or mitigate an absence of service. Some of them are also security patterns.

Safety Patterns—They describe a solution to avoid or mitigate different hazards that may result in catastrophic failures.

Hybrid Patterns—They describe combined solutions, e.g., security with reliability.

Auxiliary Patterns—They are used as a part of the solution to a particular dependability problem.

Web Service and Cloud Patterns—They describe a solution to avoid or mitigate different types of failures in clouds or web services.

Recovery and auditing patterns—These can also be included in maintainability patterns. They describe ways to handle errors.

Methodologies, metamodels, and architectures—Describe a systematic process to build dependable systems or a metamodel or architecture from which it is possible to derive patterns.

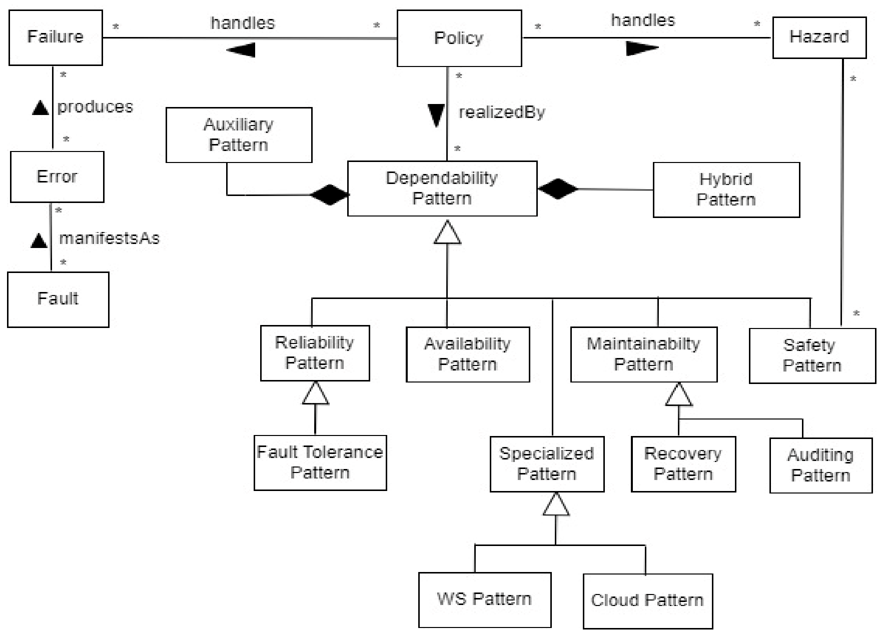

Figure 1. Classification metamodel of dependability patterns.

In the following survey, we show the authors of the patterns and where the patterns have been presented; more details of them are shown in Section 8. Some of the safety and availability patterns could also be classified as fault tolerance or reliability patterns, and vice versa; that is, the classification may have overlaps. Patterns in groups 1 to 4 are general dependability patterns; the rest are specialized patterns that apply to one type of system and maybe to any of the first categories.

4. Reliability, Fault Tolerance, and Availability Patterns

Figures 2 and 3 show pattern diagrams classifying the main patterns of this section. Figure 2 shows hardware dependability patterns, while Figure 3 shows software reliability patterns.

Saridakis presented a system of 13 patterns, including most of the common mechanisms for hardware reliability, including Fail-Stop Processor, Acknowledgment, Are You Alive (Heartbeat), Passive Replication, and others [20]. He then described patterns for fault containment [21], which include Input Guard, Output Guard, and Fault Container. Later, he produced patterns for Checkpoint-based Recovery [22] and Graceful Degradation [23]. Cecilia Rubira and collaborators produced several papers on fault-tolerant architectures using patterns [24–27]. Their paper [24] describes a Reflective State pattern from which several varieties are derived. The Reflexive State pattern, as well as several others, are also discussed in [25]. Leme et al. [26] describe the Fault Injector, Injector, and Monitor patterns. Buckley and Fernandez presented Acknowledgment, Active Replication, and Evaluator patterns [28]. Mwelwa and Pont [29] describe the Heartbeat pattern to estimate the health of a node and the error display that provides error reporting. An Error Handler pattern that can be used together with the Heartbeat pattern is shown in [30]. Kim et al. [31] had another version of the Heartbeat, Active Redundancy, and Checkpoint patterns, as well as some combinations thereof. Buckley and Fernandez [32] introduced a model to represent the sequence of actions needed to describe failures. Ahluwalia and Jain [33] described two sets of availability patterns intended for fault tolerance and fault management. Jimenez-Peris et al. [34] presented a system of architectural patterns for highly available service-oriented systems, including several varieties of database and session replication, as well as a pattern for multi-tier coordination. Several of these patterns are sketchily presented in [35], which erroneously calls them “safety” patterns; however, their paper is useful because it shows a reliable design for a robotic hand. There are few detailed examples of the application of reliability patterns.

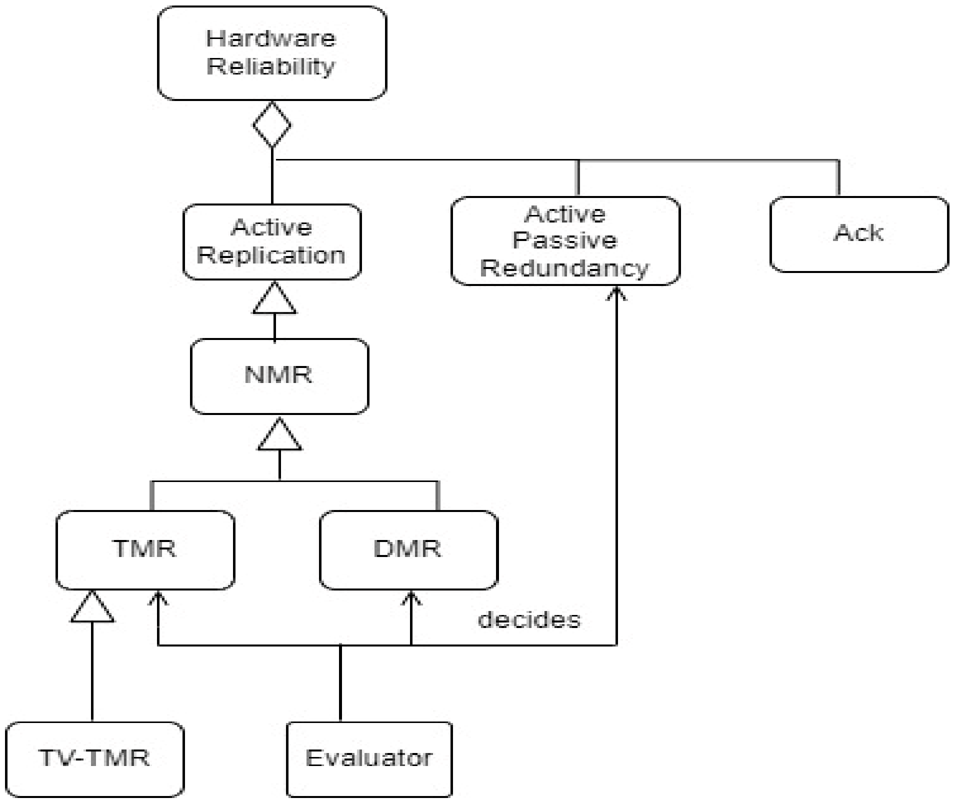

Figure 2. Pattern diagram for hardware reliability.

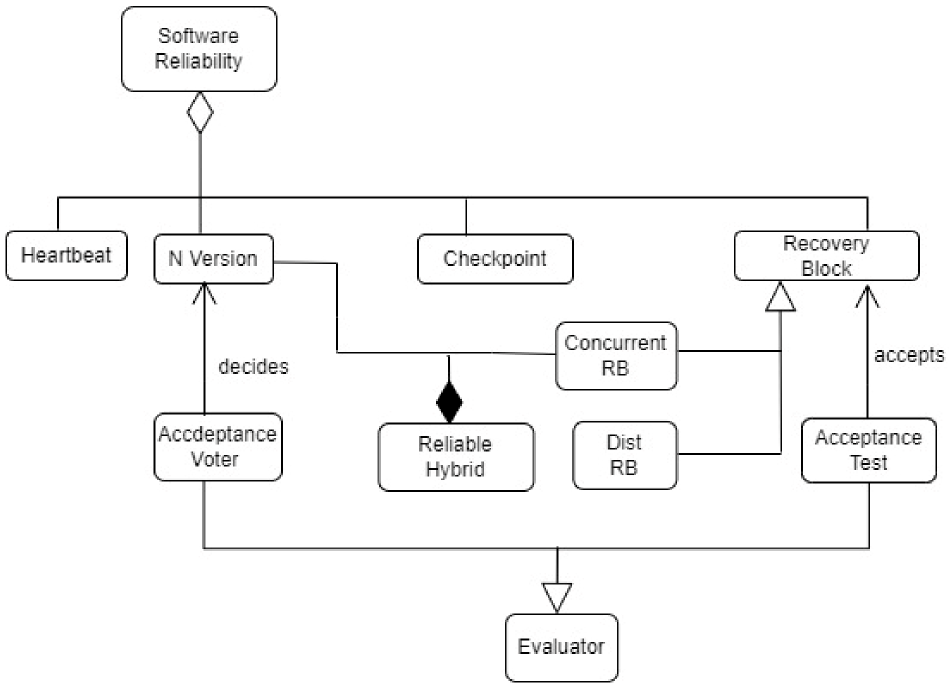

Figure 3. Pattern diagram for software reliability.

The standard mechanism to achieve dependable software uses diversity. The most common approaches, e.g., N-Version Programming, have been extensively discussed in the general literature on dependability. As patterns, they have been discussed by R. Hanmer [36–38]. A pattern describing a generalization of software methods for fault tolerance is provided in [39], which includes N-Version Programming, Recovery Blocks, Consensus Recovery Blocks, Acceptance Voting, and N-Self-Checking Programming. An Acceptance Voting pattern is described in [40]. Liu [41] showed several software versions of some of these algorithms. Ding et al. [42] proposed the Distributed Recovery Block pattern, which concurrently executes diverse versions of a program on different processors. Hoeller et al. [43] provide two patterns to obtain software diversity: Static and Dynamic Randomization; these two approaches are lightweight alternatives to N-Version Programming.

Dyson and Longshaw [44] described several availability patterns for Internet systems that include some of the others mentioned above; their new patterns included Data Replication and Session Failover. Patterns for fault-tolerant telecommunication systems are provided in [45]; these include, among others, Minimize Human Intervention, Riding Over Transients, and Leaky Bucket Counters.

Islam et al. [46] described the Recoverable Distributor, a pattern for fault-tolerant, state-sharing distributed programs. Their paper also showed the Distributed Observer pattern. A Fault Handler and a Sensor-Actuator are provided in [47]. B.P. Douglass' books [48,49] included some reliability patterns such as Watchdog, Monitor-Actuator, and others. Lau presented a set of patterns for software health monitoring in [50]; this paper defined a three-layer software monitoring architecture and provided patterns for the lower two layers, which are implemented using Aspect Oriented Programming. These include Generic and Specific Sensor, Generic and Specific Indicator, State Histogram Sensor, and Histogram Analysis Indicator. Halang et al. produced several papers on UML profiles and models for fault-tolerant systems [51,52] that included a safety shell pattern based on a Reconfiguration Management pattern.

Subramanian and Tsai [53] introduced the Backup pattern, which switches to a backup mode of operation. This provides redundancy in software when you want to offer various alternatives for a function and to switch between them dynamically. Kang and Jackson proposed the concept of Trusted Base (in the form of a pattern) [54]. Their Trusted Base pattern provided a solution to construct a system so that the most critical requirements depend on only small, reliable subsets of the system's parts, called "trusted bases". This is analogous to the concept of Trusted Computing Base (TCB) in security [7]. Their paper had several related patterns, including End-to-End Check and Trusted Kernel.

5. Safety Patterns

Safety requires reliability, and patterns for reliability are often considered safety patterns [55]; we include here only true safety patterns. Figure 4 shows a pattern diagram relating safety patterns. E. B. Fernandez and B. Hamid [56] presented two safety patterns: Safety Assertion, which describes the structure of safety assertions, and Safety Assertion Enforcer, which shows how to enforce assertions to avoid hazards. Rauhamaki and Kuikka [57,58] presented some patterns for implementing protective measures to avoid risks, including Isolate Hazard, Safety Risk Identification, and others. Rauhamaki [58,59] discussed a Shared Message Bus and a Separate Message Bus to connect safety systems to control systems. Safety systems are usually distributed and must collaborate with the control system to avoid catastrophic failures. Other patterns in [59] included Output interlocking, Safety Limits, and Shared Safety Actuator. The identification of safety risks leads to safety assertions, which are then enforced by the Safety Enforcer through the Safety Executive.

Hansen and Gullesen [60] showed how to relate safety properties to patterns that can be realized by components. A. Hauge and K. Stolen have written several papers about safety patterns. Their reference [61] proposed a pattern-based approach called Safe

Control Systems (SaCS) that included a pattern language and provided guidance on the development of design concepts for safety-critical systems. It provided three types of basic patterns: Requirements, Design, and Safety, with an example of each one: State Space Subset, Trusted Backup, and Adapting within a Constrained State Space [62], respectively. Each safety pattern is justified using arguments. These patterns are used in an example of a railway interlock. This work is extended in [63], where the patterns are applied to a nuclear plant system, and in [64], where the patterns are applied to the railway domain. These authors in [64] presented an analytical evaluation of the SaCS pattern language using appropriateness factors and requirements.

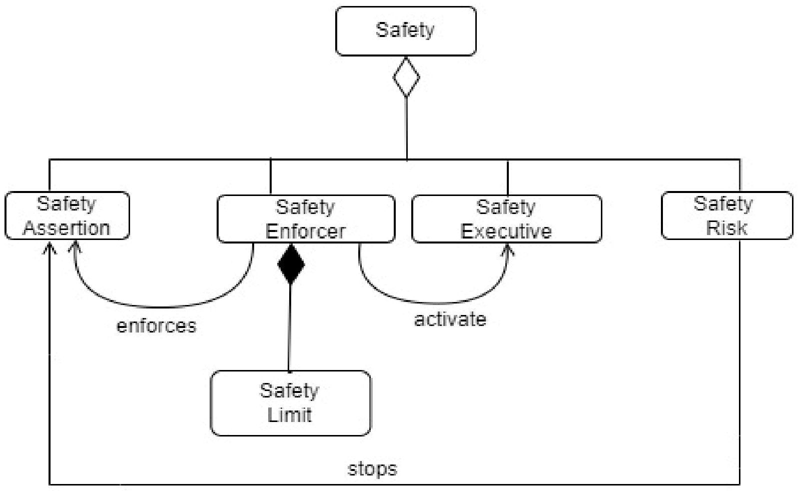

Figure 4. Pattern diagram relating safety patterns.

Armoush et al. [40] proposed extensions and modifications to the pattern templates of safety-critical systems to make explicit the effect of a pattern on nonfunctional requirements (NFRs); they illustrate their template with patterns for Safety Executive and Acceptance Voting. B.F. Douglass' book [48] included several safety patterns, such as Sanity Check and Monitor-Actuator. His later book [49] included some safety patterns, such as the Safety Executive and others. Olivera's [65] thesis presented a metamodel for safety patterns as a basis for building a safety pattern repository; the thesis presents details of the design and implementation of this repository.

Ljosland [66], in his thesis, described a controlled experiment that evaluated the difficulty of building safety-critical systems as opposed to functional designs. Gleirscher and Kugele [67] studied some safety patterns in detail. Mahemoff et al. [68] emphasized usability patterns for designing safety-critical systems, apparently the only work on this important aspect of the design of critical systems; their paper contains 18 patterns, an example of a pattern, and an example of the use of all the patterns. Daramola et al. [69] proposed an ontology-based approach that uses pattern templates to describe security requirements that can also be applied to safety.

Delmas [70] proposed hardening an architecture using "safety" patterns, although these were really reliable patterns. Pont et al. [71,72] described a safety architecture, but all the patterns in it, except one, are really reliability patterns. Luo et al. [73] developed an architecture pattern for autonomous driving applications called the Safety Channel, which intends to ensure safety using fail-safe strategies in safety-critical automated driving functions. Khalil et al. [74] defined a structured approach for the reuse of safety mechanisms in the Automotive Domain, including a pattern library; this was followed by guidance for its use and an evaluation of this approach [75].

6. Specialized Patterns and Related Artifacts

6.1. Hybrid Patterns

We consider as “hybrid” patterns that combine dependability with other nonfunctional requirements (NFRs), e.g., security, or that apply to both dependability and security. Buckley and Fernandez combined reliability with security in two patterns [4]. Secure Reliability describes how to secure reliable systems, while Reliable Security shows Reliable Security patterns. A Safety Reference Monitor that can enforce safety assertions as well as security policies is shown in [76]. Amorim et al. [77] developed patterns for combining safety and security in the Automotive Domain. K. Birman’s group has produced several papers on related topics. Reiter et al. [78] proposed a security architecture for distributed fault-tolerant systems. This architecture is based on process groups, a collection of processes with a common group address. Process groups have been described as patterns in [79], and other aspects of these architectures can also become patterns. Kreiner [80] presented a three-view architecture inspired by Kruchten’s 4+1 views model [81], including views for functions, elements, and composition and considering reliability, safety, and security requirements; the paper presented views and patterns for dependable systems design, including security aspects. M. Hafiz published a pattern called Unique Atomic Chunks that assures a cleaner locking for access to shared files; however, it is not clear why it is also a reliability or security pattern, as he implies in [82]. Montesi and Weber [83] described the Circuit Breaker as a pattern for preventing cascading failures by guarding service calls; this pattern is implemented as a Decorator [10] applied to the interface of the service. Asnar et al. developed organizational patterns for security and dependability [84]; these patterns consider the interaction of humans and systems and, as such, are particularly interesting and different from most of the patterns discussed here. Castellanos et al. [85] developed model transformations, preconditions, and postconditions to automate security and dependability design patterns. Gawand et al. [86] combined safety and fault tolerance.

6.2. Webservices and Cloud Patterns

Some of these patterns may be more general than just for web services or clouds. We do not include patterns oriented to specific products. Patterns for web services reliability standards are presented in [87]. This work includes WS-Reliability, WS-Reliable Messaging, and a comparison of both standards. Buckley et al. [88] demonstrated the value of using patterns for reliability and security certification of services. Faridoon and Pantea [89] considered a fault-tolerant composition of web services using WS-BPEL. Thaisongsuwan and Senivongse [90] used fault tolerance patterns in WS-BPEL processes. Hanmer [91] presented the first cloud fault tolerance patterns; the paper includes three patterns: No Hardware to Preserve, Virtual Machines are Cheap, and Loose Affiliation. There is ample room here for new patterns. Shunmugasundaram [92] described a variety of availability patterns oriented to cloud systems in a three-part sequence: Part 1 defines basic concepts, Part 2 shows availability and resilience patterns (this is the only place where we have found resilience patterns), and Part 3 shows an example using AWS EC2. T.B. Souza et al. in [93] showed some patterns for automated recovery of containers in clouds. There are several varieties of process isolation for error containment; the Bulkhead pattern (used by Microsoft’s Azure cloud [93,94]) described one of them as resembling the submarine principle used in more general systems. The Circuit Breaker is used in Azure to recover from faults that take a variable time [93]; this is also a more general pattern. Reliable Acquisition [95] is used in clouds for the reliable delivery of messages [96]. Azure uses the Retry pattern [97] and Health Endpoint Monitoring [98]. Grand [99] described some dependability patterns using Java.

6.3. Recovery and Audit Patterns

System recovery implies restoring past states by establishing checkpoints that contain past states. Audit requires recording the history of a process in the form of a Log. A pattern for a Security Logger/Auditor is shown in [1].

C.T. Davies [100] proposed the concept of *spheres of control*, which define logical boundaries for recovery and auditing and, as such, have an important significance in controlling the propagation of faults. This is a suitable direction to build new patterns. A recovery design pattern is presented in Fayad et al. [101], which applies to any domain where recovery of something (a signal, a state) is needed. Memento (a design pattern) [10] is another type of recovery pattern where a previous system state can be recovered. Saridakis presented a Checkpoint-based Recovery pattern [22].

6.4. Auxiliary Patterns

They include Voting, Comparator, Acceptance test, Sanity Check, Strategy, Mediator, and Memento. Strategy, Mediator, Observer, and Memento are design patterns described in [10]. Zghurskyi proposed patterns to protect systems after failure by balancing loads [102].

7. Methodologies, Metamodels, Architectures, Tactics, and Arguments

A catalog of patterns is not very useful by itself; the designer needs help to decide where in the system model a pattern is needed and which pattern would be most effective for a specific purpose. Having many patterns to choose from and deciding where to apply them is confusing for developers. A systematic methodology is required to build dependable systems where guidance is provided to the designer in the use of patterns. There are few methodologies that intend to build reliable systems. This is in contrast with security, where there are many methodologies to design secure systems [8]. To define a methodology, we need process patterns that indicate a sequence of steps where specific actions must be accomplished in each stage.

M. Tichy [103] considered the design of self-managing, dependable systems using patterns, including not only design aspects but also deployment aspects, something rarely discussed in most works. He also studied the pattern-based synthesis of fault-tolerant embedded systems. His objective was to automate the application of fault tolerance into embedded systems. For this purpose, he developed a formal model based on Petri nets. To define the points where to apply fault tolerance mechanisms, he considered the possible faults of the system. As an example, he used a TMR system applied to correct a system's weakness. Grunske focused on the evaluation of safety problems: If a software architecture does not satisfy its requirements, it is modified by adding patterns that improve its safety properties [104]. He applies several patterns for reliability and safety, which include Recovery Blocks, Watchdog, and others. Insertion of a pattern is considered a model transformation that effectively performs model refactoring [105]. Tichy and Grunske's methods are similar to the addition of security patterns to control threats and thus improve the security of a system [1,8].

M. J. Pont and his group wrote several papers on improving the reliability of embedded systems using patterns [29,71,72,106], where patterns are used to support the transition between event-triggered system architectures and time-triggered architectures. Gribov and Voos [107] proposed a safety-oriented process to design autonomous robots. The authors provided a metamodel for safety concepts from which some patterns can be derived. The process was demonstrated using a robot controlled by ROS (Robot Operating System). In a follow-up paper [108], they defined a multilayer architecture for safe robots using UML models that can easily become ideas for patterns. A set of patterns for robust robot systems was proposed in Trad and Trad [109] and included patterns such as Data Storage Design, Vision Recognition, Generic, and Infrastructure, as well as some related architectures.

Bernardi et al. [110] extended MARTE with dependability analysis and modeling. MARTE is an Object Management Group (OMG) profile that extends UML with real-time concepts. A survey of dependability using UML was produced later by this group [111]. Garbinat and Guerraoui [112] proposed an object-oriented framework for reliable distributed computing to build protocols such as atomic commitment and total order broadcasts to support the development of reliable distributed applications.

Y. Choi [113] described a methodology for building safe systems going from use cases to components. The design is model checked using RSML^{-e}, a formal dataflow language. While the methodology does not use patterns, its structure diagrams can easily be converted into patterns. Webel et al. [114] presented a System Design Language (SDL) methodology for the development of reliable systems using SDL design patterns that describe reliability patterns such as Heartbeat and Watchdog. They adopted an augmented system reliability process and defined reliability patterns to design SDL components. Their aim was to integrate these solutions into an existing system design that protects against certain types of system failures.

We presented the idea of a methodology that includes a failure enumeration method that shows how each activity in the system is analyzed using an activity diagram that maps actions to possible failures of the system [32]; identifying failures allows for countering failures using reliability policies. A software development life cycle (SDLC) and the different levels of the system architecture are integrated into the methodology to identify what is required given the specification at each stage of the SDLC and level (application, OS, etc.) of the system to counter failures. These policies are then described in the form of patterns.

Attribute-Driven Design (ADD) is an approach that defines a software architecture in which the design process is based on the quality attribute requirements the software must fulfill. ADD follows a recursive process that decomposes a system or system element by applying architectural tactics and patterns that satisfy its driving quality attribute requirements. The example in the report shows a practical application of the ADD method to a client–server system [115]. This example focuses on selecting patterns to satisfy typical availability requirements for fault tolerance.

Macher et al. [116] described patterns for the design of automotive embedded systems and for the migration of single-core-oriented software to multi-core systems. L. Yuan et al. [117] proposed a generic architecture for fault-tolerant software using Object-Z. The components of their architecture could become patterns, and the generic architecture could become a Dependability Reference Architecture.

J. L. De la Vara et al. [118] produced a metamodel for the specification and management of safety compliance for critical systems. Some of its components could be made into patterns. Ancona et al. [119] described a metamodel for an architecture for fault tolerance in concurrent systems. This work was extended in the form of a meta-object for fault tolerance in Clematis et al. [120]. The University of Newcastle PRIME project produced several papers with models and patterns for fault tolerance, including a proposal for a Holistic Fault Tolerance architecture based on centralized fault tolerance management, with related functionality distributed across the whole system [121]. Rytter and Jørgensen [122] used a metalevel architecture to build fault containers. Their architecture is a compound pattern and uses the Lookout pattern as a component pattern.

Iliasov and Romanovsky [47] provided formal descriptions of Recovery Blocks and N-Version Programming intended to assist in the correct application of these patterns; the paper uses the formal language B to prove correctness with the final objective of automatic model transformations. A formal proof based on Petri nets for a pattern for the fault-tolerant execution of parallel programs is provided in Kindler and Shasha [123]. Lopatkin et al. [124] showed seven patterns for representing Failure Modes and Effects Analysis (FMEA) concepts. FMEA is a technique used for inductive reliability and safety analysis.

Harrison and Avgeriou [15] considered the use of tactics for fault tolerance in software architectures. Tactics may also be considered as specialized design policies that can be realized by patterns. Scott and Kazman [125] showed the use of a catalog of availability tactics in a real-world application. They considered tactics such as Active and Passive Redundancy, Spare, Exception Handling, Rollback, and others. Harrison and Avgeriou [15] studied how the patterns (styles) used to build an architecture influence the use of fault tolerance tactics. Their findings showed that certain architecture patterns were better suited to support fault tolerance mechanisms than others. They believe that their approach can be used by system architects to help them select architecture patterns and mechanisms to build reliable systems. This work was further extended by Harrison et al. [126], who studied how information about fault tolerance tactics and software patterns can be used to help architects make better decisions about system design. They found that implementing these tactics affects patterns by modifying their components, such as adding or replicating components and connectors. An important aspect of the study was to identify the interactions of several of the most common fault tolerance tactics with several of the most common architecture patterns (styles). They built a case study to see how architects design with tactic–pattern interaction information against design without that information. While useful, they observed that the architects considered the approach challenging because they found it difficult to select the necessary tactics required to provide the level of fault tolerance needed in the system, given the constraints introduced by the architectural patterns. The case study also showed that architects tended to overuse tactics as well as show the shortcomings of tactics.

A complement to patterns or tactics is the use of arguments that attempt to prove that a system follows its requirements. Safety cases are structured arguments to argue the safety of a system. Kelly and his group at the University of York produced several papers on this subject [16]. They proposed a template to describe safety tactics where one of its sections (Rationale) presents arguments [127]. The templates can be used to build architectural patterns such as a control/monitor. More safety patterns were developed by them later, and examples of their use are shown in [16,128]. A pattern for arguing the compliance of system safety requirements decomposition is provided in [129]. Recurrent solutions can be described by patterns that allow for reusing successful arguments. Ayoub et al. [130] presented a safety case pattern to argue about the correctness of implementations developed using model-based approaches. Gleischer and Kugele wrote an extensive survey of safety and arguments [67,131], where they also considered related aspects that may affect safety, including security and reliability. L. Yuan et al. proposed a heterogeneous software architecture, GFTSA (Generic Fault-Tolerant Software Architecture), that can guide the development of safety-critical distributed systems [117].

8. Dependability Patterns

In this section, we show descriptions of the intents of the most significant dependability or auxiliary patterns listed in alphabetic order. Not all the patterns mentioned in the survey are here, but the most useful or common are included. By useful, we mean patterns that describe solutions to important and frequent problems, such as producing reliable results from software computations or avoiding failures of critical services, e.g., cargo port cranes. 'Common' means patterns that describe solutions for commonly occurring situations, e.g., availability of services in a building.

The **Acceptance test** decides if the result of a process execution is correct based on a predefined criterion [39]. It is used by the Recovery Block and other methods. This pattern provides error detection.

The **Acceptance Voting** pattern combines the NVP pattern with the Acceptance test used by the Recovery Block pattern [39–41]. Like the NVP, this pattern is based on the independent generation of N > = 2 functionally equivalent software modules from the same initial specification. The output of each version is presented to an Acceptance test to check its correctness. The outputs that pass the Acceptance test are used as inputs to a dynamic voter, which is executed to produce the correct output. This pattern provides error detection and masking.

The **Acknowledgment** pattern detects errors in a system by acknowledging the reception of an input within a specified time interval [20,28]. If a response is sent before the timeout, the system is considered to function correctly; otherwise, it is assumed that an error has occurred in the system. This pattern provides error detection.

The **Active Replication** pattern uses a set of replicated processors that take in the same input and conduct independent and concurrent processing of that input. The outputs from all replicas are compared to ascertain the correct output [20,28]. This pattern is a generalization of DMR and TMR and is used to mask hardware errors.

The **Active–Passive Replication (APR)** pattern provides a backup for an important process by replicating a standby, which is activated in case of failure of the former process [20,33]. The client of the failed part should be informed about the passive part's activation. This pattern produces error masking.

Active–Passive Conflict Resolution. This pattern works with the APR pattern to avoid conflict among competing redundant parts that are waiting to become active when the primary part fails [33].

Adapting Within a Constrained State Space provides an augmentation structure that demonstrates that an adaptable control system is safe [62]. This is achieved by arming the adaptable controller region of the system with a set of system conditions that involve constraints and privileges.

The **Automated Recovery** pattern defines checks to periodically evaluate the health of containers in a cloud and restart them automatically if unhealthy [93].

Backpressures are operational patterns designed to equalize the traffic characteristics that can protect systems from overload [102]. Their goal is to reduce the adverse effects of intermittent systemic failure.

The **Balking** method will only execute an action on an object when the object is in a particular state [53]. It returns control immediately, with a warning if an object is invoked when it is not in an appropriate state to execute the method. This pattern provides error detection.

Basic Runtime represents an initial step to provide availability for a website by performing software and node configuration scenarios within an internal and external network [62].

Bulkhead is a type of application design that is tolerant of failure [93,94]. In a bulkhead architecture, elements of an application are isolated into pools such that if one fails, the others will continue to function.

The **Checkpoint-Rollback** pattern takes a snapshot of the system just as it is about to begin the first step of the next transaction [20,31,132]. The snapshot is only taken if the previous transaction was completed in a fault-free state. When an error is detected during a transaction, the transaction is "Fail Stopped", and then the system is taken back to its latest saved checkpoint. This pattern provides error detection.

A Circuit Breaker handles faults that might take a variable amount of time (such as transient faults, slow network connections, timeouts, etc.) to recover from when connecting to a remote service or resource [83,93,133].

The **Data Replication** pattern ensures that the system state is consistent between servers in an architecture that uses multiple identical servers if the system maintains dynamic data [129].

The **Distributed Recovery Block** pattern concurrently executes diverse versions of a program on different processors [42]. This pattern extends the Recovery Block pattern by providing both hardware and software fault tolerance.

The **Dual Modular Redundancy (DMR)** pattern uses two hardware copies that work in parallel to carry out a process [134]. This pattern is a special case of the Active Replication pattern and provides error detection.

The **Dynamic Dual Modular Redundancy (DDMR)** pattern provides an approach for dynamic linking of redundant processors using a Dynamic DMR [135]. By supporting dynamic pairing of processors, DDMR allows the operating system to schedule redundant threads on any two processors within a group. DDMR is a scalable Dynamic DMR approach and may be used in symmetric shared-memory architectures as well as in distributed shared-memory architectures. This pattern provides error detection and masking.

The **End-to-End Check** pattern checks that every component carries out its task correctly [54]. The requirement that the actual output matches the expected value ("Output OK") depends on every component in the system behaving as expected and satisfying the property that is assigned to it. This pattern verifies that the outcome of a computation is correct.

The **Evaluator** pattern **s**elects a correct output from a set of independent output sources using a selection method [28]. Given a set of independent outputs, it performs a comparison using an appropriate algorithm for determining the correct output based on the number of outputs in the set. This pattern is a generalization of the voter and comparator. The number of outputs must be two or greater.

The ***Failover Cluster*** [44,136] pattern uses a highly available application infrastructure tier that protects against loss of service due to the failure of a single server or the software that it hosts. In a Failover Cluster, if one of the servers becomes unavailable, another server takes over and continues to provide the service to the end user. This is an alternative version of APR.

Fault Injection tries to produce or simulate faults during an execution of the system under test, and the behavior of the system is observed [26,93].

The **Fault-Tolerant Reflective State** pattern combines a Reflective State pattern with different approaches to fault tolerance [24,25]. It implements a Finite State Machine that can reconfigure a system to be fault tolerant.

The **Generic Indicator** pattern utilizes a unit that retrieves and analyzes relevant data from one or more sensors and derives a state value representing the state of a particular phase of the execution [50]. This pattern provides error detection.

The **Guarded Call** pattern provides timely response to service requests across multiple threads. It allows timely synchronization or data exchange between threads and handles mutual exclusion issues for thread-safe rendezvous [48]. This pattern provides error detection.

The **Health Endpoint Monitoring** pattern is used to verify that applications and services are performing correctly [98]. This pattern specifies the use of functional checks in an application. External tools may access these checks at regular intervals through exposed endpoints.

The **Heartbeat** pattern uploads messages to a special "heartbeat service" [20,29,31]. The Heartbeat service then updates the user's last Heartbeat record. Heartbeat is closely related to Timeout and Acknowledgement patterns, but it works in a different way. This pattern provides error detection.

The **Heterogeneous Redundancy** pattern. When the primary channel detects a fault, the secondary channel takes over [48]. This pattern is an alternative version of the APR pattern.

The **Homogeneous Redundancy** pattern is another name for Active Replication, i.e., it is used to increase reliability in a system by offering multiple channels [48].

The **Immutable Classes** technique bypasses safety issues by not changing an object's state after creation [137].

The **Independent Checkpoint** pattern **p**reserves process autonomy by allowing components to take checkpoints asynchronously [57]. It uses a communication-induced checkpoint coordination system for the progression of recovery using a recovery database that helps rollback propagation. When the recovery data are identified, a recovery mechanism instructs each component to roll back to its latest checkpoint and resumes execution from that point onward. It requires a recovery database consistent at all times with the latest checkpoint of any process.

The **Leaky Bucket Counters** pattern provides reliability in a system where errors are isolated and handled by taking devices out of service but where transient errors do not cause unnecessary out-of-service action [45,138]. A failure group with a counter is initialized to a predetermined value. The counter is decremented for each fault or event (usually faults) and incremented on a periodic basis. When the counter reaches its limit, i.e., when the last fault occurs within the timing window, the faulty unit is identified and taken out of service. This pattern provides error detection and containment.

The **Load Balancer Hot Standby** pattern removes the single point of failure from the load balancer node by adding a second node with the same function. The first node is the primary load balancer; the second one keeps track of the balancing information and is ready to take over the work of the primary node at any moment [41]. This pattern is a special case of APR and provides error detection.

The **Maximize Human Participation** pattern provides the capability for knowledgeable people to guide the system's error processing [36].

The **Mediator** pattern defines an object that encapsulates how a set of objects interact. It promotes loose coupling by keeping objects from referring to each other explicitly, and it allows for the varying of their interaction independently [10]. Colleagues (or peers) are not coupled to one another; each talks to the Mediator, which, in turn, knows and conducts the necessary operation of the others. This pattern provides error propagation control.

The **Memento** pattern. Without violating encapsulation, it captures and externalizes an object's internal state so that the object can be restored later to this state [10].

The **Minimize Human Intervention** pattern helps make a system less susceptible to human error by reducing human intervention and is used in processing high-reliability, continuous-running digital systems [36,45]. Human intervention can be reduced by building strategies that counter human tendencies to act impulsively, thereby causing further problems in the system. This pattern provides error avoidance.

The **N-Modular Redundancy (NMR)** pattern executes N instances (N odd) of the same module performing the same computation, and then a majority vote of the output(s) is taken. As long as a majority of modules compute the output properly, the system output is correct. This voting process enables masking of errors [21]. This pattern is a generalization of TMR and DMR and provides error detection and masking.

N-Version Programming (NVP) is based on the independent execution of N>=2 functionally equivalent software modules [36,37,39]. It uses multiple versions of the same process implemented in different languages and different hardware. The output of each version is presented to an Acceptance Test (voter) to check its correctness. The outputs that pass the Acceptance Test are used as inputs to a dynamic voter, which is executed to produce the correct output according to a voting method.

The **Object Group** pattern allows the implementation of replicated objects for load sharing and for efficient multicast communication in distributed systems [79]. A replicated state remains consistent despite objects entering and leaving the group dynamically and failures]. This pattern provides error masking.

The **Observer** pattern defines a one-to-many dependency between a subject object and any number of observer objects so that when the subject object changes state, all its observer objects are notified and updated [10]. This pattern may be used for error detection by monitoring changes in the state of a system resource.

The **Process Pairs** pattern passes information about its new consistent state to a backup server when the primary server successfully completes an entire transaction [139]. It is the same as the APR pattern.

The **Protected Single Channel** pattern uses a single channel to handle sensing and actuation [70]. Reliability is enhanced through the addition of checks at key points in the channel, which may require some additional hardware.

The **Recoverable Distributor** pattern is a compound pattern for distributed systems that combines fault detection, containment, and recovery [46]. It has two important properties: one is masking processor failures; that is, it must be able to preserve the state of the system despite such failures. It should also hide network latency as much as possible while providing consistent access to the shared state of all processors in the system. It has a state management section (local and global) and a fault detection and recovery section; this enables the creation of local views of shared data.

The **Recovery Block** pattern performs **a**n Acceptance test after every processing alternative is completed [139]. In this way, processing alternatives are run until a processing alternative succeeds in delivering results that pass the Acceptance test. This pattern provides error detection.

The **Reliable Acquisition** pattern delivers a message reliably from a cloud event source that does not implement transactions to an outbound operation of a connector that implements transactions [96]. This pattern provides error detection.

The **Reliable Hybrid** pattern provides a general object-oriented framework for fault tolerance [39]. It is a combination of several fault-tolerance patterns to support the development of applications based on classical fault-tolerant strategies. The strategies include N-Version Programming and Recovery Block, as well as those based on advanced hybrid techniques such as Consensus Recovery Block, Acceptance Voting, and N-Self Checking Programming. This pattern provides error detection, error masking, and recovery.

The **Reliable Security** pattern performs reliable authorization enforcement by applying reliability to a reference monitor and to its authorization rules [4]. User requests must be authorized based on the user's rights. A reference monitor is used to enforce authorization.

The **Retry** pattern enables an application to handle transient failures when it tries to connect to a service or network resource by transparently retrying a failed operation [97].

The **Riding Over Transients** pattern detects temporally dense events [45]. It allows a system to roll through problems without its users noticing them and without the machine operator intervening. This pattern resolves errors with minimal effort by first determining whether a problem actually exists.

Safety Assertion describes a state of the system that must not happen because it may lead to a mishap [56].

Safety Assertion Enforcer evaluates safety assertions when there is an incoming event that can change the state of the system, and it prevents the change if it violates an assertion [56].

The **Safety Executive** pattern utilizes a Watchdog pattern in combination with an additional Safety Executive component, which is responsible for the shutdown of the system as soon as the Watchdog sends a shutdown signal [40,49].

The **Sanity Check** pattern ensures that the system is doing something reasonable, even if not quite correct [49,70,129]. This is useful in situations where the actuation is not critical, but it can do harm if it is performed incorrectly. It is a variant of the Monitor-Actuator pattern, and like that pattern, it assumes that a fail-safe state is available [70,129]. This pattern provides error masking.

The **Secure Reliability** pattern assures the use of reliable services in a system. It employs a strategy-based system that receives a request and selects the appropriate service to reliably process that request [4].

The **Session Failover** pattern ensures that user interaction is uninterrupted in the event of failure or the need to upgrade or maintain the system [44].

The **Single Load Balancer** pattern distributes incoming requests for an application among web servers on the web application server nodes [140]. It balances incoming traffic among a cluster of back-end servers. It can detect a failed back-end server and forward traffic to other servers within the same cluster of back-end servers. This pattern provides error detection.

The **State Space Subset** is used to address problem domain issues inherent in control systems by only allowing an adaptable controller to operate in a limited part of the operational state space [37]. This is achieved by dividing the controlled state space into a set of overlapping operational partitions and with the use of redundant and alternative means of control to reduce the risk related to adaptive controller failure. This pattern provides fault containment.

The **Strategy** pattern decouples an algorithm from its host and encapsulates the algorithm into a separate class [10]. This allows for switching the algorithm being used at any point in time. This pattern has been rewritten in [4], and it is used in the Secure Reliability pattern.

The **Timeout** pattern establishes a countdown to a timeout state. Each significant activity cancels the timer and starts a new one in its place [68].

The **Triple Modular Redundancy (TMR)** pattern utilizes three systems to perform a process, and the result is processed by a voting system to produce a single output [67,139,141]. If any of the three systems fails, the other two systems can mask the fault. If the voter fails, then the complete system will fail. This pattern is a special case of Active Replication and provides error detection and error masking.

The **Triplicated Voters Triple Modular Redundancy (TV-TMR)** pattern extends the TMR pattern discussed earlier. It provides redundancy using three voters instead of one to vote on inputs provided by three identical modules to produce one output [19,20]. This pattern provides error detection and error masking.

The **Trusted Base** pattern constructs a system so that the most critical requirements depend on only small, reliable subsets of the system's parts, called "trusted bases" [54]. Having isolated the trusted parts from the untrusted ones, we need to ensure only the reliability of the trusted bases and be confident that the system will satisfy its critical requirements.

The **WS-Reliability** pattern is a web services pattern that ensures that a notification is always sent in response to a failure. It also provides guaranteed message delivery, message ordering, and duplicate elimination whenever messages are sent from one entity to another [87]. This is achieved by establishing an enforceable contract between the sending and receiving parties and the use of sending and receiving reliable message processors (RMPs). This pattern provides error detection.

WS-Reliable Messaging is a web services pattern that ensures guaranteed receipt in response to each message sent; it also provides message state disposition, ordered delivery, and duplicate elimination whenever messages are sent between endpoints. This is achieved by first having an agreement that includes a policy exchange, endpoint resolution, and establishment of trust between endpoints [87]. This pattern provides error detection.

9. Evaluation of Selected Patterns

A problem with the identified patterns is that most of them are described in an ad hoc manner, not using a standard template. To make them useful for designers, we should represent them uniformly and collect them in a catalog. We now analyze the quality of their representation; we leave to others the construction of a catalog. We use the POSA template [9] as a reference; it is considered the most appropriate to describe architecture patterns [9]. The second author has written two books on security patterns and has found this template a convenient way to describe security and dependability patterns. We also use a notation based on [142]:

U, **under-specified or incomplete**. The pattern does not use an appropriate template, is missing whole sections, or its sections are not described in sufficient detail to be used by a designer.

O, **over-specified**. The pattern's description is overly detailed with additional unnecessary properties. The pattern may also include multiple solutions.

P, lack of precision. The solution is not presented using UML, SysUML, Modelica, or other precise notation or does not solve a specific well-defined problem. The structure and dynamics of the solution should be described as a guideline for its proper application. For example, some patterns are described using only words, but words can be misleading or vague. A pattern with missing sections can be completed, but an imprecise pattern needs to be completely redone.

G, lack of generality. The pattern's solution is only applicable to a narrow or specific problem or provides a solution that is unclear or impractical for reuse.

N, **unusual notation**. The pattern uses an unusual notation or template, or it is defined in an ad hoc way. This makes the pattern difficult to use together with patterns defined in more standard forms, and they need to be completely redone to be included in a catalog.

M, misrepresentation. The pattern name does not suit its intent or function, or it is misleading. For example, some pattern authors confuse safety with reliability.

Most of these are qualitative measures, and therefore, they may have a degree of subjectivity. Completeness is measurable only if we use a specific template as a reference. We prefer not to consider a pattern to be over-specified because the extra information may be useful for some users.

Table 1 shows some of the patterns discussed earlier, indicating their dependability properties and evaluating them using the notation above. In this table, patterns with no quality indicators imply they are reasonably correct or complete; that is, we could not find any clear defects with them. The result of this survey shows that clear and comprehensive information is necessary for any effective guideline, and a pattern should be a complete and helpful guideline.

Table 1. Evaluation of dependability patterns.

Pattern	Dependability Property	Quality Indicator
Reliability Patterns:		
1. Reliable Hybrid	Alerting Error Detection Redundancy Error Masking Fault Containment	■ P, U, N ■ Insufficient UML diagrams ■ Inadequate details given ■ Incorrect UML notation
2. Homogeneous Redundancy	Error Masking Redundancy	■ P, N ■ Inadequate details given ■ Incorrect UML notation
3. Heterogeneous Redundancy	Error Detection Redundancy Error Masking	■ P, N ■ Inadequate details given ■ Incorrect UML notation
4. Active–Passive Replication	Alerting Error Detection Redundancy Error Masking Fault Containment	■ P, U, N ■ Insufficient UML diagrams ■ Inadequate details given ■ Incorrect UML notation
5. Recoverable Distributor	Redundancy Error Masking	■ P, U, N, M ■ Incomplete pattern ■ Insufficient UML diagrams ■ Inadequate details given
6. Active–Passive Conflict Resolution	Redundancy Error Masking	■ P, U ■ Incomplete pattern ■ Insufficient UML diagrams ■ Inadequate details given
7. N-Modular Redundancy (NMR)	Redundancy Error Detection Error Masking	■ P, U ■ Incomplete pattern ■ No UML diagrams ■ Inadequate details given
8. Triplicated Voters Triple Modular Redundancy (TV-TMR)	Error Detection Redundancy Error Masking	■ P, U, N ■ Incomplete pattern ■ Incorrect UML notation ■ Inadequate details given
9. Recovery Blocks	Diversity	■ P, U ■ Incomplete pattern ■ No UML diagrams ■ Inadequate details given

Table 1. *Cont.*

Pattern	Dependability Property	Quality Indicator
Reliability Patterns:		
10. Leaky Bucket Counters	Error Detection Error Masking Fault Containment	▪ P, U ▪ Incomplete pattern ▪ Insufficient UML diagrams ▪ Inadequate details given
11. Acknowledgment	Error Detection Alert	▪ C ▪ Complete
12. Observer	Error Detection	▪ P, U ▪ Incomplete pattern ▪ No UML diagrams ▪ Inadequate details given
13. Object Group	Redundancy Error Masking	▪ P, U ▪ Incomplete pattern ▪ Insufficient UML diagrams
14. Heartbeat	Error Detection	▪ P, U ▪ Incomplete pattern ▪ Insufficient UML diagrams
15. Checkpoint-Rollback	Error Detection Redundancy Error Masking Recovery	▪ P, U ▪ Incomplete pattern ▪ No UML diagrams ▪ Inadequate details given
Safety Patterns:		
16. Immutable classes	Error Masking See definition	▪ P, U ▪ Incomplete pattern ▪ No UML diagrams ▪ Inadequate details given
17. Safety Assertion	Policy	▪ Complete
Web Service Patterns:		
18. WS-Reliability	Error Detection Alert Error Masking	▪ C ▪ Complete
19. WS-Reliable Messaging	Error Detection Alert Error Masking	▪ C ▪ Complete
Hybrid Reliability/Security Patterns:		
20. Secure Reliability	Redundancy Error Masking	▪ C ▪ Complete
21. Reliable Security	Redundancy Error Masking	▪ C ▪ Complete
Auxiliary Patterns:		
22. Mediator	Error Detection Fault Containment	▪ P, U ▪ Incomplete pattern ▪ Insufficient UML diagrams
23. Strategy	Diversity Error Masking	▪ P, U, N ▪ No UML diagrams ▪ Inadequate details given

10. Directions for Research

We mentioned some ideas for future work in the text; we summarize them here:

- Build a catalog of dependability patterns. This implies converting all of the patterns in Table 1 to POSA style, except those that already have this format. A template similar to that we used for security patterns in [1] would be appropriate, although some work should be performed first to see if this template can be improved.
- In Section 8, we ignored some patterns mentioned in the survey because their descriptions were insufficient or we did not have time to read these papers in detail; for example, Mahemoff presented 18 patterns [68]. Add them to Sections 8 and 9. Try to find new patterns in the related literature.
- The lists of tactics include some that have not been realized by patterns, e.g., the ones in the report of Scott and Kazman that contain several availability tactics [125].
- There are very few cloud reliability patterns; write more of them. Recovery and auditing are also areas where there are few patterns.
- Write patterns for robot reliability. Safety patterns are clearly important as complements. Start from Gribov and Voos [107,108] and Trad and Trad [109].
- Derive patterns from the models in MARTE [110] and in Choi [113]. These papers already have UML models, and it should be easy to discover patterns in them.
- Combinations of patterns are interesting ideas. They can be used to solve several related problems, as shown in Daniels et al. [39].
- Write patterns for automotive embedded systems. Safety and reliability patterns are needed for these applications. New areas that do not have any dependability patterns are machine learning, virtual networks, and multi-edge computing.
- Convert Tichy [103] and Grunske [104] reliable system development methodologies to a methodology in the style of Uzunov et al. [8]. Consider also the methodology of Buckley in [32].
- Several papers have models that, although not defined as patterns, are very close to being patterns, e.g., De la Vara [118], Choi [33], and Bernardi et al. [111]. These papers may contain architectures, metamodels, or a classification of approaches that could become patterns once transformed using a template.

11. Results

We consulted a total of 142 papers that provided background and contained dependability, reliability, or safety patterns, as well as related papers with UML models or reference architectures about these topics. After classifying and analyzing these papers, we found the following results:

- *Survey and enumerate dependability patterns*. We tried to see what is available and detect if they cover a broad set of topics. We identified a suitable number of patterns that cover the common problems of dependability. We identify how security can affect dependability.
- *Classify these patterns in groups and relate them using pattern diagrams*. This classification is important to see the coverage of topics and to understand how these patterns relate to each other. We defined a new classification and created a UML class model and three pattern diagrams that make explicit the relationships between these patterns, thus facilitating their use.
- *Evaluate the suitability of existing patterns to build dependable systems*. We found that many patterns need completion or refactoring to be used in a practical catalog. We made a table indicating the necessary modifications to make these patterns useful to practitioners.
- *Survey methodologies for systematic development of dependable systems*. These are important to use the patterns in building dependable systems. We identified several methodologies to build dependable systems.
- *Provide a list of directions for research*. We indicate 10 possible directions.

12. Discussion

In this section, we analyze, from a critical perspective, the set of results presented in Section 10.

From our survey, we found that there is a sufficient number of dependability patterns that cover a wide range of design problems. We found also that several patterns have been rediscovered (sometimes more than once), sometimes with different names. Most of the pattern papers use their own template or even no template. By completing the remodeling of selected patterns, we could build a catalog that designers could use to build dependable systems. Table 1 is a roadmap to indicate what patterns should be modified. We did not describe every pattern we found due to incomplete descriptions or because they were just variants of other patterns.

After 2019, we could not find any papers on dependability patterns. Apparently, all the most common problems already have patterns. There may be recent patterns in the gray (non-academic) literature, but they are usually shown without dates. New areas, such as machine learning, virtual networks, and multi-edge computing, should produce more patterns. On the other hand, we identified a suitable amount of current work on the general subject of dependability that shows the practical value of our results. It is clear that reliability is a fundamental requirement for IT systems; when systems are not reliable, they bring a lack of trust, loss of time, and economic losses. The importance of this topic is shown by the fact that there are several journals dedicated to it: IEEE Transactions on Dependable and Secure Computing, IEEE Transactions on Reliability, Reliability Engineering and System Safety, and the International Journal of Quality & Reliability Management.

The results of our survey should be useful to guide practitioners and researchers in the following use cases:

UC1: *Publish new mechanisms or security analysis of dependability patterns*. When researchers want to write and publish their “new” mechanisms or dependability analysis, they can be aware of existing related work so they can avoid rehashing old work. Deficiencies or limitations of existing solutions can also lead to new ideas. Our list of directions for research indicates areas where there is not enough work. This can be useful for selecting new work.

UC2: *Solve dependability design problems*. When practitioners and researchers want to solve design problems in systems that need to be dependable, our classification results can help them compare existing mechanisms or analyses and then select and reuse the appropriate one according to their objectives.

UC3: *Communicate and search for ideas*. Our classification results can serve as a reference for the dependability engineering community, including practitioners and researchers. Our results can be extended by peers, providing the community with an important body of knowledge to guide future publications and research on this subject. Abstracting this knowledge in the form of a pattern catalog would provide a useful tool for future developers.

UC4: *Transition from software engineering to dependability or from dependability to software engineering*. If a software engineering expert (researcher or practitioner) wants to add dependability to her applications, in this survey, she can find knowledge to guide her to apply patterns in the right stages of the lifecycle and in the right places in the architecture. A dependability expert may want to see how the dependability mechanisms he knows can be applied in the design of new applications.

UC5: *Obtain a view of the state of the art*. Somebody preparing a presentation or publication of the current state of the art on system dependability would have a summary and repository of the relevant research.

13. Related Work

Several surveys about different aspects of dependability exist:

Waseem Ahmed, Yong Wei Wu, A survey on reliability in distributed systems. *Journal of Computer and System Sciences* **2013**, *79*, 1243–1255. https://doi.org/10.1016/j.jcss.2013.02.006.

Analyzes in detail existing reliability methodologies from the viewpoint of the reliability of individual components and explains why a comprehensive reliability model for applications running in the distributed system is needed.

Khaled Almakadmeh, Ghadeer Al Qahmouss, A Comprehensive Survey of System Dependability for Real Time Embedded Software, IPAC '15, Batna, Algeria, 22–25 November 2015. http://dx.doi.org/10.1145/2816839.2816917.

This is a survey of previous research studies of the system dependability of real-time embedded software in a variety of application domains and intends to suggest some dependability solutions for future research.

Andrea Bobbio, Dependability analysis of fault-tolerant systems: a literature survey. *Microprocessing and Microprogramming* **1990**, *29*, 11–13. https://doi.org/10.1016/0165-6074(90)90007-V.

This paper surveys the literature on dependability modeling, with particular emphasis on modeling techniques, computing methodologies, and software tools.

Massimo Felici, Taxonomy of Evolution and Dependability, Conference: Workshop on Unanticipated Software Evolution, USE 2003.

This paper reviews a taxonomy of evolution, identifying a conceptual framework to analyze the evolutionary phenomena of computer-based systems as well as models of evolution and their limits. The taxonomy of evolution points out different dependability aspects of computer-based systems.

Anne Immonen · Eila Niemelä, Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Softw. Syst. Model* **2008**, *7*, 49–65. https://doi.org/10.1007/s10270-006-0040-x.

A comparison of the existing analysis methods and techniques for reliability and availability prediction at the architectural level. The objective is to discover which methods are suitable for the reliability and availability prediction of today's complex systems, what are the shortcomings of the methods, and which research activities need to be conducted in order to overcome these identified shortcomings.

Raj Kamal Kaur, Babita Pandey, Lalit Kumar Singh, Dependability analysis of safety-critical systems: Issues and challenges. *Annals of Nuclear Energy* **2018**, *120*, 127–154.

A comprehensive literature survey that investigates different metrics, threats, means, techniques, and methodologies to ensure the dependability of computer-based critical systems. The limitations of these elements are also analyzed with respect to their applicability in SC systems.

Sparsh Mittal, Jeffrey S. Vetter, A Survey of Techniques for Modeling and Improving Reliability of Computing Systems. *IEEE Transactions on Parallel and Distributed Systems* **2016**, *27*, 1226–1238. https://doi.org/10.1109/TPDS.2015.2426179.

A survey of architectural techniques for improving resilience of computing systems. It especially focuses on techniques proposed for microarchitectural components, such as processor registers, functional units, cache and main memory, etc.

Mangey Ram, On system reliability approaches: a brief survey. *Int. J. Syst. Assur. Eng. Manag.* **2013**, *4*, 101–117. https://doi.org/10.1007/s13198-013-0165-6.

A survey of reliability approaches in various fields of engineering and physical sciences. The survey discusses past, current, and future trends in reliability methods and applications.

V.V.S. Sarma, A survey of software dependability. *Sddhand* **1987**, *11*, 23–48.

An overview of the issues in precisely defining, specifying, and evaluating the dependability of software, particularly in the context of computer-controlled process systems. The paper surveys the models and methods available for measuring and improving software reliability.

Ref. 12. Bernardi, S.; Merseguer, J.; Petriu D. C. Dependability modeling and analysis of software systems specified with UML. *ACM Computing Surveys* 2012, 45, 1–48. https://doi.org/10.1145/2379776.2379778.

Its goal is to survey dependability modeling and analysis of software and systems specified with UML, with a focus on reliability, availability, maintainability, and safety (RAMS). The survey shows that more works are devoted to reliability and safety, fewer to availability and maintainability, and none to integrity.

Ref. 50. Gleirscher, M.; Kugele, S. Assurance of system safety: A survey of design and argument patterns. 14 February 2019. https://doi.org/10.48550/arXiv.1902.05537.

This work summarizes applied research on safety with a focus on the last two decades and on the state-of-the-art patterns in safety-critical system design and assurance argumentation.

Ref. 140. Xie, Z.; Sun, H.; Saluja, K. A Survey of Fault Tolerance Techniques. Available online: http://www.pld.ttu.ee/IAF0030/Paper_4.pdf (accessed on 14 July 2023).

Considers only fault tolerance aspects but no patterns.

All these references are about surveys that either do not consider patterns or include only specific types of patterns. Ours is the only survey that includes all kinds of dependability patterns. Refs. 12 and 50, mentioned in the paper, are the closest to our objectives and can be considered complementary papers.

14. Conclusions

Our analysis has shown that there is a suitable amount and variety of dependability patterns that should cover most of the needs of designers; however, many dependability patterns that have been proposed are incomplete and lack necessary details. Many of the patterns illustrated in Table 1 do not conform to the POSA or any other template. It can also be seen that many patterns use different notations to describe their structures, which makes them more difficult to interpret and apply. This survey has also highlighted the prospect of combining some dependability patterns together to create a composite structure, as opposed to implementing each separately. This survey provides a basis for a catalog that would contain a unified set of dependability patterns. A catalog requires rewriting most of these patterns using a common template and the same degree of detail. We must indicate that not many dependability patterns have been published recently, although, as shown above, there are many models that could be the basis of suitable patterns and new areas where there are few or no patterns. We provided a list of directions for research that should be of value to researchers by indicating missing patterns and related possibilities. As far as we know, this is the only survey of dependability patterns.

Author Contributions: Conceptualization, E.B.F. and I.A.B.; methodology, E.B.F.; investigation, E.B.F. and I.A.B.; writing—original draft preparation, I.A.B.; writing—review and editing, E.B.F.; supervision, E.B.F. Handling of the two revisions: E.B.F. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Fernandez, E.B. *Security Patterns in Practice: Building Secure Architectures Using Software Patterns*; Wiley Series on Software Design Patterns, 2013; Available online: https://www.amazon.sg/Security-Patterns-Practice-Designing-Architectures/dp/1119998948 (accessed on 14 July 2023).
2. Uzunov, A.V.; Fernandez, E.B.; Falkner, K. Securing distributed systems using patterns: A survey. *Comput. Secur.* **2012**, *31*, 681–703. [CrossRef]

3. Avizienis, A.; Laprie, J.C.; Randell, B.; Landwehr, C. Basic Concepts and Taxonomy of Dependable and Secure Computing. *Proc. IEEE Trans. Dependable Secur. Comput.* **2004**, *1*, 11–33. [CrossRef]
4. Buckley, I.A.; Fernandez, E.B. Patterns Combing Reliability and Security. In Proceedings of the Third International Conferences on Pervasive Patterns and Applications, 25–30 September 2011.
5. Fernandez, E.B.; Yoshioka, N.; Washizaki, H.; Yoder, J. Abstract security patterns for requirements specification and analysis of secure systems. In Proceedings of the WER 2014 Conference, a Track of the 17th Ibero-American Conference on Software Engineering (CIbSE 2014), Pucon, Chile, 23–25 April 2014.
6. Fernandez, E.B.; Astudillo, H.; Pedraza-Garcia, G. Revisiting architectural tactics for security. In Proceedings of the 9th European Conference on Software Architecture (ECSA 2015), Cavtat, Croatia, 5–7 September 2015; pp. 55–69.
7. Gollmann, D. *Computer Security*, 3rd ed.; Wiley: New York, NY, USA, 2011; ISBN 978-0-470-74115-3.
8. Uzunov, A.V.; Fernandez, E.B.; Falkner, K. Engineering Security into Distributed Systems: A Survey of Methodologies. *J. Univers. Comput. Sci.* **2013**, *18*, 2920–3006. [CrossRef]
9. Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. *A System of Patterns: Pattern-Oriented Software Architecture*; John Wiley & Sons: Hoboken, NJ, USA, 1996; ISBN 978-0471958697.
10. Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*; Addison Wesley: Boston, MA, USA, 1994; ISBN 978-0201633610.
11. Fowler, M. *Analysis Patterns: Reusable Object Models*; Addison-Wesley: Upper Saddle River, NJ, USA, 1997; ISBN 978-0134186054.
12. Warmer, J.; Kleppe, A. *The Object Constraint Language*, 2nd ed.; Addison-Wesley: Upper Saddle River, NJ, USA, 2003; ISBN 978-0321179364.
13. Avgeriou, P. Describing, instantiating and evaluating a reference architecture: A case study. *Enterp. Archit. J.* **2003**, *342*, 1–24.
14. Bass, L.; Clements, P.; Kazman, R. *Software Architecture in Practice*, 3rd ed.; Addison-Wesley: Upper Saddle River, NJ, USA, 2012; ISBN 978-0321815736.
15. Harrison, N.; Avgeriou, P. Incorporating Fault Tolerance Tactics in Software Architecture Patterns. In Proceedings of the RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, Newcastle upon Tyne, UK, 17–19 November 2008. [CrossRef]
16. Kelly, T.P.; McDermid, J.A. Safety case construction and reuse using patterns. In Proceedings of the 16th Int. Conference on Computer Safety, Reliability and Security (SAFECOMP'97), York, UK, 7–10 September 1997; pp. 55–69. [CrossRef]
17. Laprie, J.C.; Arlat, J.; Beounes, C.; Kanoun, K. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer* **1990**, *23*, 39–51. [CrossRef]
18. von Neumann, J. Probabilistic logics and the synthesis of reliable organism from unreliable components. In *Automata Studies*; Princeton University Press: Princeton, NJ, USA, 1956; pp. 43–98.
19. Nelson, V.P. Fault-Tolerant Computing: Fundamental Concepts. *IEEE Comput.* **1990**, *3*, 19–25. [CrossRef]
20. Saridakis, T. A System of Patterns for Fault Tolerance. In Proceedings of the EuroPLoP, Irsee, Germany, 3–7 July 2002.
21. Saridakis, T. Design Patterns for Fault Containment. In Proceedings of the EuroPLoP, Irsee, Germany, 25–29 June 2003.
22. Saridakis, T. Design Patterns for Checkpoint-Based Rollback Recovery. In Proceedings of the EuroPLoP, Irsee, Germany, 25–29 June 2003.
23. Saridakis, T. Design Patterns for Graceful Degradation. In Proceedings of the Transactions on Pattern Languages of Programs, Chicago, IL, USA, 28–30 August 2009; pp. 67–93. [CrossRef]
24. Ferreira, L.L.; Rubira, C.M.F.; Rubira, M.F. The Reflective State Pattern. In Proceedings of the PLoP'98, Monticello, IL, USA, 11–14 August 1998.
25. Ferreira, L.L.; Rubira, C.M.F. Reflective design patterns to implement fault tolerance. In Proceedings of the OOPSLA Workshop on Reflective Programming, Vancouver, BC, Canada, 18 October 1998; Available online: https://www.csq.is.titech.ac.jp/~chiba/oopsla98/ferreira.pdf (accessed on 14 July 2023).
26. Leme, N.G.M.; Martins, E.; Rubira, C.M.F. A Software Fault Injection Pattern System. In Proceedings of the PLoP, Park Monticello, IL, USA, 11–15 September 2001.
27. Martins, E.; Rubira, C.M.F.; Leme, N.G.M. A reflective fault injection tool based on patterns. In Proceedings of the International Dependable Systems and Networks (DSN'02), Washington, DC, USA, 23–26 June 2002.
28. Buckley, I.A.; Fernandez, E.B. Three patterns for fault tolerance. In Proceedings of the OOPSLA MiniPLoP, Orlando, FL, USA, 26 October 2009.
29. Mwelwa, C.; Pont, M.J. Two Simple Patterns to Support the Development of Reliable Embedded Systems. In Proceedings of the of Viking PLoP, Ikaalinen, Finland, March 2013.
30. Konrad, S.; Cheng, B.H.C. Requirements patterns for embedded systems. In Proceedings of the IEEE Joint International Conference on Requirements Engineerinh (RE'02), Essen, Germany, 9–13 September 2002. [CrossRef]
31. Kim, S.; Kim, D.; Lu, L.; Park, S. Quality-driven architecture development using architectural tactics. *J. Syst. Softw.* **2009**, *82*, 1211–1231. [CrossRef]
32. Buckley, I.A.; Fernandez, E.B. Failure patterns: A new way to analyze failures. In Proceedings of the First International Symposium on Software Architecture and Patterns in Conjunction with the 10th Latin American and Caribbean Conference for Engineering and Technology, Panama City, Panama, 23–27 July 2012.

33. Ahluwalia, K.S.; Jain, A. High Availability Design Patterns. In Proceedings of the PLoP '06, Portland, OR, USA, 21–23 October 2006.
34. Jiménez-Peris, R.; Patiño-Martínez, M.; Kemme, B.; Perez-Sorrosal, F.; Serrano, D. A System of Architectural Patterns for Scalable, Consistent and Highly Available Multi-Tier Service-Oriented Infrastructures. In Proceedings of the WADS2008, Tokyo, Japan, 28 August–1 September 2023; pp. 1–23. [CrossRef]
35. Kumar, S.P.; Ramaiah, P.S.; Khanaa, V. Architectural patterns to design software safety-based safety-critical systems. In Proceedings of the ICCCS'11, Rourkela Odisha, India, 12–14 February 2011; pp. 620–623. [CrossRef]
36. Hanmer, R.S. Patterns for Fault Tolerant Software. Wiley Series in Software Design Patterns; 2007; ISBN 978-0470319796. Available online: https://www.oreilly.com/library/view/patterns-for-fault/9780470319796/ (accessed on 14 July 2023).
37. Hanmer, R.S.; Lane, L. N-Version Programming. In Proceedings of the PLoP, Chicago, IL, USA, 28–30 August 2009.
38. Hanmer, R.S. Software rejuvenation. In Proceedings of the PLoP, Salvador, Bahia, Brazil, 23–26 September 2010; pp. 1–13. [CrossRef]
39. Daniels, F.; Kim, K.; Vouk, M.A. The Reliable Hybrid pattern—A generalized software fault tolerant design pattern. In Proceedings of the PLoP'97, 1997; Available online: https://hillside.net/plop/plop97/Proceedings/daniels.pdf (accessed on 14 July 2023).
40. Armoush, A.; Salewski, F.; Kowalewski, S. Design Pattern Representation for Safety-Critical Embedded Systems. *J. Softw. Eng. Appl.* **2009**, *2*, 1–12. [CrossRef]
41. Liu, C. A general framework for software fault tolerance. In Proceedings of the Workshop on Fault-Tolerant Parallel and Distributed Systems, Amherst, MA, USA, 6 July 1992.
42. Ding, K.; Morozov, A.; Klaus, J. Classification of Hierarchical Fault-Tolerant Design Patterns. Available online: https://www.researchgate.net/publication/324179369_Classification_of_Hierarchical_Fault-Tolerant_Design_Patterns (accessed on 14 July 2023).
43. Hoeller, A.; Rauter, T.; Iber, J.; Kreiner, C. Patterns for automated software diversity. In Proceedings of the 20th European Conference on Pattern Languages of Programs (EuroPLoP 2015), Kaufbeuren, Germany, 8–12 July 2015. [CrossRef]
44. Dyson, P.; Logshaw, A. Patterns for high-availability Internet systems. In Proceedings of the EuroPLoP, Irsee, Germany, 3–7 July 2002.
45. Adams, M.; Coplien, J.; Gamboke, R.; Hanmer, R.; Keeve, F.; Nicodemus, K. Fault-Tolerant Telecommunication System Patterns. In Proceedings of the Pattern Languages of Program Design 2; Addison-Wesley Longman Publishing Co.: Upper Saddle River, NJ, USA, 1996; pp. 549–562.
46. Islam, N.; Devarakonda, M. An essential design pattern for fault-tolerant distributed state sharing. *Commun. ACM* **1996**, *39*, 65–74. [CrossRef]
47. Iliasov, A.; Romanovsky, A. Refinement Patterns for Fault Tolerant Systems. In Proceedings of the 2008 Seventh European Dependable Computing Conference EDCC, Kaunas, Lithuania, 7–9 May 2008; pp. 167–176. [CrossRef]
48. Douglass, B.F. *Doing Hard-Time: Using Object-Oriented Programming and Software Patterns in Real-Time Applications*; Addison-Wesley: Upper Saddle River, NJ, USA, 1998; ISBN 978-0321774934.
49. Douglass, B.F. *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*; Addison-Wesley Professional: Upper Saddle River, NJ, USA, 2003; ISBN 978-0201699562.
50. Lau, A.; Seviora, R.E. Design Patterns for Software Health Monitoring. In Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05), Shanghai, China, 16–20 June 2005. [CrossRef]
51. Gumzej, R.; Halang, W.A. A safety shell for UML-RT project structure and methods of the corresponding UML pattern. *Innov. Syst. Softw. Eng.* **2009**, *5*, 97–105. [CrossRef]
52. Gumzej, R.; Colnaric, M.; Halang, W.A. A reconfiguration pattern for distributed embedded systems. *Proc. Softw. Syst. Model. (Sym.)* **2009**, *8*, 145–161. [CrossRef]
53. Subramanian, S.; Tsai, W. Backup Pattern: Designing Redundancy in Object-Oriented Software. In *Pattern Languages of Program Design*; Addison-Wesley: Upper Saddle River, NJ, USA, 1996.
54. Kang, E.; Jackson, D. Patterns for building dependable systems with trusted bases. In Proceedings of the 17th Conference on Pattern Languages of Programs (PLOP '10), Reno, NE, USA, 16–18 October 2010. [CrossRef]
55. Preschern, C.; Kajtazovic, N.; Kreiner, C. Building a Safety Architecture PatternSystem. In Proceedings of the 18th European Conference on Pattern Languages of Program, Irsee, Germany, 10–14 July 2015. [CrossRef]
56. Fernandez, E.B.; Hamid, B. Two safety patterns: Safety Assertion and Safety Assertion Enforcer. In Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, Germany, 12–16 July 2017.
57. Rauhamaki, J.; Kuikka, S. A few patterns to implement protective measures. In Proceedings of the 20th European Conf. on Pattern Languages of Programs (EuroPLoP), Kaufbeuren, Germany, 8–12 July 2015; pp. 1–13. [CrossRef]
58. Rauhamaki, J. Patterns for safety system bus architecture. In Proceedings of the Viking PLoP, Leerdam, The Netherlands, 7–10 April 2016; pp. 1–8. [CrossRef]
59. Rauhamäki, J. Designing Functional Safety Systems: A Pattern Language Approach. *Trans. Pattern Lang. Program. IV* **2017**, *1478*, 100–138. [CrossRef]
60. Hansen, K.; Gullesen, I. Utilizing UML and Patterns for Safety Critical Systems. 2002. Available online: https://www.researchgate.net/publication/238477117_Utiliing_UML_and_patterns_for_safety_critical_systems (accessed on 14 July 2023).

61. Hauge, A.A.; Stølen, K. SACS: A pattern language for safe adaptive control software. In Proceedings of the 18th Conference on Pattern Languages of Programs (PLoP '11), Portland, OR, USA, 21–23 October 2011. [CrossRef]
62. Hauge, A.A.; Stølen, K. A pattern-based method for safe control systems exemplified within nuclear power production. In Proceedings of the SAFECOMP, Magdeburg, Germany, 25–28 September 2012; pp. 13–24. [CrossRef]
63. Hauge, A.A.; Stølen, K. Developing safe control systems using patterns for assurance. In Proceedings of the Third Int. Conf. on Performance, Safety and Robustness in Complex Systems and Applications (PESARO 2013), Opatija, Croatia, 20–24 May 2013.
64. Hauge, A.A.; Stølen, K. An analytic evaluation of the saCS pattern language—Including explanations of major design choices. In Proceedings of the PATTERNS, Moscow, Russia, 23–28 June 2014; pp. 79–88.
65. Olivera, A. Taim: A Safety Pattern Repository. Bachelor's Thesis, Federal University of Rio Grande do Sul, São Lourenço do Sul, Brazil, 2012.
66. Ljosland, I. BUCS: Patterns and Robustness. Master's Thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2006.
67. Gleirscher, M.; Kugele, S. *A Study of Safety Patterns: First Results*; Institut fur Informatik, Technical University of Munich: Munich, Germany, 2016. [CrossRef]
68. Mahemoff, M.; Hussey, A.; Johnston, L. Pattern-based reuse of successful designs: Usability of safety-critical systems. In Proceedings of the Australian Software Engineering Conference, Canberra, ACT, Australia, 27–28 August 2001. [CrossRef]
69. Daramola, O.; Sindre, G.; Stålhane, T. Pattern-based security requirements specification using ontologies and boilerplates. In Proceedings of the Second International Workshop on Requirements Patterns (RePa '12), Chicago, IL, USA, 24 September 2012; pp. 54–59. [CrossRef]
70. Delmas, K.; Delmas, R.; Pagetti, C. Automatic Architecture Hardening Using Safety Patterns. In Proceedings of the International Conference on Computer Safety, Reliability, and Security (Safecomp), Delft, The Netherlands, 23–25 September 2015; pp. 283–296. [CrossRef]
71. Pont, M.J. Designing and Implementing Reliable Embedded Systems Using Patterns. In Proceedings of the EuroPLoP, Irsee, Germany, 7–11 July 1999; pp. 257–290.
72. Pont, M.J.; Banner, M.P. Designing embedded systems using patterns: A case study. *J. Syst. Softw.* **2004**, *71*, 201–213. [CrossRef]
73. Luo, Y.; Saberi, A.K.; Bijlsma, T.; Lukkien, J.J.; van den Brand, M. An architecture pattern for safety critical automated driving applications: Design and analysis. In Proceedings of the Annual IEEE International Systems Conference (SysCon), Montreal, QC, Canada, 24–27 April 2017; pp. 1–7. [CrossRef]
74. Khalil, M.; Prieto, A.; Hölzl, F. A Pattern-Based Approach towards the Guided Reuse of Safety Mechanisms in the Automotive Domai. In Proceedings of the IMBSA 2014, Munich, Germany, 27–29 October 2014; pp. 137–151. [CrossRef]
75. Khalil, M. Design Patterns to the rescue: Guided model-based reuse for automotive solutions. In Proceedings of the PLoP, Irsee, Germany, 24–26 October 2018.
76. Fernandez, E.B.; VanHilst, M.; LaRed, D.; Mujica, S. An extended reference monitor for security and safety. In Proceedings of the 5th Iberoamerican Conference on Information Security (CIBSI 2009), Jakarta, Indonesia, 5–7 December 2017.
77. Amorim, T.; Martin, H.; Ma, Z.; Schmittner, C.; Schneider, D.; Macher, G.; Winkler, B.; Krammer, M.; Kreiner, C. Systematic Pattern Approach for Safety and Security Co-engineering in the Automotive Domain. In Proceedings of the SAFECOMP, Trento, Italy, 12–15 September 2017; pp. 329–342.
78. Reiter, M.K.; Birman, K.P.; van Renesse, R. A secure architecture for fault-tolerant systems. Proc. *ACM Trans. Comput. Syst.* **1994**, *12*, 340–371. [CrossRef]
79. Maffeis, S. The Object Group Design Pattern—An Object Behavioural Pattern for Fault-Tolerance and Group Communication in Distributed Systems. Available online: https://ecommons.cornell.edu/bitstream/handle/1813/7227/96-1570.pdf?sequence=1 (accessed on 13 July 2023).
80. Kreiner, C. Essential architectural views for dependable system design. In Proceedings of the 20th European Conference on Pattern Languages of Programs (EuroPLoP), Kaufbeuren, Germany, 8–12 July 2015.
81. Kruchten, P. The 4+1 view model of architecture. *IEEE Softw.* **1995**, *12*, 42–50. [CrossRef]
82. Hafiz, M. Unique atomic chunks—A pattern for security and reliability. In Proceedings of the PLoP, Allerton Park in Monticello, IL, USA, 8–12 September 2004.
83. Montesi, F.; Weber, J. From the decorator patterns to circuit breakers in microservices. In Proceedings of the 33rd Annual ACM Symposium (SAC 2018), Pau, France, 9–13 April 2018; pp. 1733–1735. [CrossRef]
84. Asnar, Y.; Massacci, F.; Saïdane, A.; Riccucci, C.; Felici, M.; Tedeschi, A.; ElKhoury, P.; Li, K.; Seguran, M.; Zannone, N. Organizational Patterns for Security and Dependabiity: FromDesign to Application. *Int. J. Secur. Softw. Eng.* **2011**, *2*, 22. [CrossRef]
85. Castellanos, C.; Vergnaud, T.; Borde, E.; Derive, T.; Pautet, L. Formalization of design patterns for security and dependability. In Proceedings of the 4th ACM Sigsoft International Symposium on Architecting Critical Systems (ISARCS), Vancouver, BC, Canada, 17–21 June 2013; pp. 17–26. [CrossRef]
86. Gawand, H.; Mundada, R.S.; Swaminathan, P. Design Patterns to Implement Safety and Fault Tolerance. *Int. J. Comput. Appl.* **2011**, *18*, 6–13. [CrossRef]

87. Buckley, I.A.; Fernandez, E.B.; Rossi, G.; Sadjadi, M. Web services reliability patterns. In Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE'2009), Boston, MA, USA, 1–3 July 2009; pp. 4–9.
88. Buckley, I.A.; Fernandez, E.B.; Anisetti, M.; Ardagna, C.A.; Sadjadi, M.; Damiani, E. Towards Pattern-based Reliability Certification of Services. In Proceedings of the 1st International Symposium on Secure Virtual Infrastructures (DOA-SVI'11), Crete, Greece, 17–19 October 2011; Springer Lecture Notes in Computer Science; p. 7045. [CrossRef]
89. Faridoon, S.; Pantea, N. Propound Solutions for Increase Fault Tolerance in Web Services CompositionInt. *J. Syst. Soft.Eng.* **2013**, *1*, 17–22.
90. Thaisongsuwan, T.; Senivongse, T. Applying software fault tolerance patterns to WS-BPEL processes. In Proceedings of the Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE), Nakhonpathom, Thailand, 11–13 May 2011; pp. 269–274. [CrossRef]
91. Hanmer, R.S. Patterns for Fault Tolerant Cloud Software. In Proceedings of the PLoP, Monticello, IL, USA, 14–17 September 2014; Available online: https://hillside.net/plop/2014/papers/Security/hanmer.pdf (accessed on 14 July 2023).
92. Shunmugasundaram, S. Architecting for Reliability Series. Available online: https://medium.com/becloudy/architecting-for-reliability-part-1-concepts-17028343089 (accessed on 14 July 2023).
93. Sousa, T.B.; Ferreira, H.S.; Correia, F.F.; Aguiar, A. Engineering Software for the Cloud: External Monitoring and Failure Injection. In Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18), Irsee, Germany, 4–8 July 2018; Volume 7, pp. 1–8. [CrossRef]
94. Bulkhead Pattern. Available online: https://learn.microsoft.com/en-us/azure/architecture/patterns/bulkhead (accessed on 12 July 2023).
95. Reliability Patterns. Available online: https://docs.mulesoft.com/mule-runtime/4.4/reliability-patterns (accessed on 12 July 2023).
96. Xie, Z.; Sun, H.; Saluja, K. A Survey of Fault Tolerance Techniques. Available online: http://www.pld.ttu.ee/IAF0030/Paper_4.pdf (accessed on 12 July 2023).
97. Retry Pattern. Available online: https://learn.microsoft.com/en-us/azure/architecture/patterns/retry (accessed on 26 August 2023).
98. Health Endpoint Monitoring pattern. Available online: https://learn.microsoft.com/en-us/azure/architecture/patterns/health-endpoint-monitoring (accessed on 12 July 2023).
99. Grand, M. *Patterns in Java Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML*; Addison-Wesley: Upper Saddle River, NJ, USA, 1998; ISBN 0471258393.
100. Davies, C.T. Data processing spheres of control. *IBM Syst. J.* **1978**, *17*, 179–198. [CrossRef]
101. Fayad, M.; Rajagopalan, J.; Hamza, H. Recovery design pattern. In Proceedings of the IEEE International Conference on Info. Reuse and Integration (IRI 2003), Las Vegas, NV, USA, 27–29 October 2003. [CrossRef]
102. Zghurskyi, O. Backpressure Patterns in Practice. Available online: https://www.zghurskyi.com/backpressure/2019 (accessed on 12 July 2023).
103. Tichy, M. Pattern Based Synthesis of Fault Tolerant Embedded Systems. In Proceedings of the SIGSOFT, Portland, ON, USA, 5–11 November 2006.
104. Grunske, L. Transformational Patterns for the Improvement of Safety Properties in architectural Specifications. In Proceedings of the Viking PLoP, Ikaalinen, Finland, 21–24 March 2013.
105. Fowler, M. *Refactoring—Improving the Design of Existing Code*; Addison-Wesley: Upper Saddle River, NJ, USA, 1999; ISBN 978-0201485677.
106. Lakhani, F.N.; Pont, M.J. Applying Design Patterns to Improve the Reliability of Embedded Systems through a Process of Architecture Migration. In Proceedings of the Computing and Communication & IEEE International Conference on Embedded Software and Systems, Liverpool, UK, 25–27 June 2012; pp. 1563–1570. [CrossRef]
107. Gribov, V.; Voos, H. Safety oriented software engineering process for autonomous robots. In Proceedings of the 2013 IEEE 18th Conference on Emerging Technologies & Factory Automation (ETFA), Cagliari, Italy, 10–13 September 2013; pp. 1–8. [CrossRef]
108. Gribov, V.; Voos, H. A multilayer software architecture for safe autonomous robots. In Proceedings of the IEEE Emerging Tech. and Factory Automation, Barcelona, Spain, 16–19 September 2014. [CrossRef]
109. Trad, A.; Trad, C. Audit, control and monitoring design patterns (ACMDP) for autonomous robust systems (ARS). *Int. J. Adv. Robot. Syst.* **2005**, *2*, 25–38. [CrossRef]
110. Bernardi, S.; Merseguer, J.; Petriu, D.C. A dependability profile within MARTE. *Softw. Syst. Model* **2011**, *10*, 313–336. [CrossRef]
111. Bernardi, S.; Merseguer, J.; Petriu, D.C. Dependability modeling and analysis of software systems specified with UML. *ACM Comput. Surv.* **2012**, *45*, 1–48. [CrossRef]
112. Garbinat, B.; Guerraoui, R. An Open Framework for Reliable Distributed Computing. *ACM Comput. Surv.* **2000**, *32*, 22–26. [CrossRef]
113. Choi, Y. Early Safety Analysis: From Use Cases to—Component-based Software Development. *J. Object Technol.* **2007**, *6*, 185–203. Available online: https://www.jot.fm/issues/issue_2007_09/article4 (accessed on 25 March 2023).
114. Webel, C.; Fliege, I.; Geraldy, A.; Gotzhein, R. Developing Reliable Systems with SDL Design Patterns and Design Components. In Proceedings of the ISSRE04 Workshop on Integrated-Reliability with Telecommunications and UML Languages, 2004; Available online: https://www.sdl-forum.org/issre04witul/papers/witul04_developing_reliable_systems.pdf (accessed on 14 July 2023).

115. Wood, W.G. *A Practical Example of Applying Attribute-Driven Design (ADD)*, version 2.0; Software Engineering Institute, Carnegie Mellon University: Pittsburgh, PA, USA, 2007. [CrossRef]
116. Macher, G.; Hoeller, A.; Armengaud, E.; Kreiner, C. Safety-critical embedded system multi-core migration pattern. In Proceedings of the 20th European Conference on Pattern Languages of Programs (EuroPLoP'15), Kaufbeuren, Germany, 8–12 July 2015.
117. Yuan, L.; Dong, J.S.; Sun, J.; Basit, H.A. Generic Fault Tolerant Software Architecture Reasoning and Customization. *Proc. IEEE Trans. Reliab.* **2006**, *55*, 421–435. [CrossRef]
118. De la Vara, J.L.; Nair, S.; Verhulst, E.; Studzizba, J.; Pepek, P.; Lambourg, J.; Sabetzadeh, M. Towards a model-based evolutionary chain of evidence for compliance with safety standards. In Proceedings of the Workshop Next Generation Syst. Assurance Approaches Safety-Critical Syst. Workshop; 2012; pp. 64–78. Available online: https://people.svv.lu/sabetzadeh/pub/SASSUR12.pdf (accessed on 12 July 2023).
119. Ancona, M.; Clematis, A.; Dodero, G.; Fernandez, E.B.; Gianuzzi, V. System Architecture for Fault Tolerance in Concurrent Systems. *IEEE Comput.* **1990**, *23*, 23–32. [CrossRef]
120. Clematis, A.; Ancona, T.; Dodero, G.; Gianuzzi, V.; Lisbôa, M.L. An object-oriented approach to fault-tolerant software. In Proceedings of the Euromicro Workshop on Parallel and Distributed Processing, San Remo, Italy, 25–27 January 1995. [CrossRef]
121. Gensh, R.; Rafiev, A.; Romanovsky, A.B.; Garcia, A.F.; Xia, F.; Yakovlev, A. Architecting Holistic Fault Tolerance. In Proceedings of the HASE, Singapore, 12–14 January 2017; pp. 5–8. [CrossRef]
122. Rytter, M.; Jørgensen, B.N. Enhancing NetBeans with Transparent Fault Tolerance Using Meta-Level Architecture. *J. Object Technol.* **2010**, *9*, 55–73. [CrossRef]
123. Kindler, E.; Shasha, D. Verifying a design pattern for the fault-tolerant execution of parallel program. In *Technical Report*; New York University: New York, NY, USA, 2000.
124. Lopatkin, I.; Iliasov, A.; Romanovsky, A.B.; Prokhorova, Y.; Troubitsyna, E. Patterns for Representing FMEA in Formal Specification of Control Systems. In Proceedings of the HASE'11, Nanjing, China, 3–5 December 2008; pp. 146–151.
125. Scott, J.; Kazman, R. Realizing and refining architectural tactics: Availability. In *Technical Report*; Software Engineering Institute, Carnegie Mellon University: Pittsburgh, PA, USA, 2009. [CrossRef]
126. Harrison, N.; Avgeriou, P.; Zdun, U. On the Impact of Fault Tolerance Tactics on Architecture. In Proceedings of the ACM SERENE 2010, London, UK, 13–16 April 2010. [CrossRef]
127. Wu, W.; Kelly, T. Safety tactics for software architecture design. In Proceedings of the 28th Annual International Conference of Computer Software and Applications, Hong Kong, China, 28–30 September 2004; pp. 368–375. [CrossRef]
128. Hawkins, R.; Clegg, K.; Alexander, R.; Kelly, T. Using a Software Safety Argument Pattern Catalogue: Two Case Studies. In Proceedings of the SAFECOMP, Naples, Italy, 19–22 September 2011; pp. 185–198. [CrossRef]
129. De Oliveira, A.L.; Braga, R.T.V.; Masiero, P.C.; Habli, I.; Kelly, T. A pattern to argue the compliance of system safety requirements decomposition. In Proceedings of the 10th Conference on Pattern Languages of Programs (SugarLoafPLoP), Sagadi Manor, Estonia, 9–12 November 2014.
130. Ayoub, A.; Kim, B.G.I.; Sokolsky, O. A safety case pattern for model-based development approach. In Proceedings of the NASA Formal Methods Symposium, Norfolk, VA, USA, 3–5 April 2012.
131. Gleirscher, M.; Kugele, S. Assurance of system safety: A survey of design and argument patterns. *arXiv* **2019**, arXiv:1902.05537. [CrossRef]
132. Holzner, S. Design Patterns: Simply. Available online: http://sourcemaking.com/design_patterns/mediator (accessed on 13 July 2023).
133. Circuit Breaker Pattern. Available online: https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker (accessed on 12 July 2023).
134. Teifel, J. Self-voting dual-modular-redundancy circuits for single event transient mitigation. *Proc. IEEE Trans. Nucl. Sci.* **2008**, *55*, 3435–3439. [CrossRef]
135. Golander, A.; Weiss, S.; Ronen, R. DDMR: Dynamic and Scalable Dual Modular Redundancy with Short Validation Intervals. *Proc. IEEE Comput. Archit. Lett.* **2008**, *7*, 65–68. [CrossRef]
136. Microsoft, Failover Cluster. Available online: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff650328(v=pandp.10) (accessed on 13 July 2023).
137. Nierstrasz, O. Safety Patterns. Available online: https://sgb.unibe.ch/download/cp/04SafetyPatterns.pdf (accessed on 14 July 2023).
138. Gamoke, R. Pattern: Leaky Bucket Counters. 1995. Available online: https://www.researchgate.net/publication/242361940_Fault-tolerant_telecommunication_system_patterns (accessed on 14 July 2023).
139. Kalinsky, D. Design Patterns for High Availability. Available online: https://www.design-reuse.com/articles/3671/design-patterns-for-high-availability.html (accessed on 13 July 2023).
140. IBM, Non-Functional Requirements: High Availability: Runtime Patterns. Available online: http://www.ibm.com/developerworks/patterns/edge/at1-runtime.html. (accessed on 13 July 2023).

141. Wakerly, J.F. Microcomputer Reliability Improvement Using Triple Modular Redundancy. *IEEE Trans. Comput.* **1976**, *64*, 889–895. [CrossRef]
142. Laverdiere, M.; Mourad, A.; Hanna, A.; Debbabi, M. Security design patterns: Survey and evaluation. In Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE'06), Ottawa, ON, Canada, 7–10 May 2006; pp. 1605–1608. [CrossRef]