Reliability Patterns: A Survey

Ingrid A. Buckley, Ph.D[1], and Eduardo B. Fernandez, Ph.D[2]
[1]Florida Gulf Coast University, USA, ibuckley@fgcu.edu
[2]Florida Atlantic University, USA, fernande@fau.edu

***Abstract*–This study presents an enumeration and evaluation of reliability patterns. We first catalog these patterns to make them accessible and useful to software developers and system designers. The objective here is to identify areas for which there are no patterns or the existing patterns are not complete enough to be useful to software designers. The patterns are classified based on the reliability properties they provide. As examples of our methodology we evaluate some of the patterns discussed in this survey.**

Keywords-- reliability, reliability patterns, software patterns, fault tolerance

I. Introduction

Reliability is especially important for safety-critical systems, which control applications important in a variety of areas, including government, transportation, power generation, and others. The need and use of critical systems has increased over time, especially with the pervasive and increasing use of distributed and cloud-based systems. Reliability is a property which allows some function, task or service to behave as intended when required and measures continuity of correct service. Reliability is particularly important in critical infrastructures and is a fundamental prerequisite of safety and fault tolerance. Fault tolerance masks faults that may occur during execution.

A pattern is an encapsulated solution to a recurrent problem in a given context and can be tailored to fit different situations [14], [26]. Patterns have proven to be useful in the development of good quality systems because they provide reusability and a systematic approach for designing, implementing, and evaluating complex software systems. Reliability patterns are useful for system designers and developers who need help in implementing reliable or high availability systems. Reliability patterns offer system designers and programmers who have little knowledge and experience in implementing software reliability, best practices and guidelines on how to achieve different levels of reliability.

In this survey we enumerate patterns intended to build reliable systems. We analyze some of these patterns to determine if they conform to the standard pattern definition, if there is another pattern with the same objectives but a different name, and if it has a complete description. We evaluate each pattern using a set of quality criteria, mostly taken from [63]. As part of this enumeration, we identify patterns that are not sufficiently elaborated and need to be completed before they can be useful for developers.

This paper is organized as follows: Section 2 provides background information about patterns and reliability. Section 3 presents a variety of reliability patterns. Section 4 evaluates some of the reliability patterns along two dimensions: reliability properties and quality indicators. Section 5 presents some conclusions and discusses future work.

II. Background

As indicated above, a pattern is a solution to a recurring problem in a specific context. Software patterns are categorized as analysis [26], design [26], architecture [14], and security patterns [22]. 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., “overhead must be reasonable”. 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). UML diagrams may include class, sequence, state, and activity diagrams. A set of consequences indicate 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. A section on “known uses” lists real systems where this solution has been used previously, i.e., a pattern is an abstraction of a good practice. A section on related patterns indicates patterns that complement or provide alternative solutions to the one in the pattern.

A pattern embodies the knowledge and experience of software developers and can be reused in new applications; carefully designed patterns implicitly apply good design principles. Patterns are also good for communication between designers and to evaluate or reengineer existing systems. While initially developed for software, patterns can describe hardware, physical entities, 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.

A fault is a defective value in the state of a component or in the design of a system. A fault can be classified by its duration, nature and degree [5]. 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. A system failure occurs when there is a deviation from the system specification; a failure is the manifestation of an error. A system that can detect, mask and recover from the effects of a fault and continue operating correctly is said to be reliable. Failures in a system can cause

Digital Object Identifier (DOI):
http://dx.doi.org/10.18687/LACCEI2019.1.1.53
ISBN: 978-0-9993443-6-1 ISSN: 2414-6390

harm and thus affect the safety of the system. Similarly, a failure can affect the availability of a system, because a system may become unresponsive when it experiences a failure. Reliability patterns describe a solution to avoid or mitigate different types of software failures.

III. Reliability Patterns

In this section we discuss a variety of approaches to achieve reliability which include reliability, fault tolerance, and availability patterns. Many patterns solve the same problem, however, they have different names, we discuss some of them here.

N-version programming is a common approach which provides software fault tolerance, it has been discussed in detail by R. Hanmer [33]. A pattern describing a generalized version of this pattern is given in [46], which also includes Recovery Blocks, Consensus Recovery Blocks, Acceptance Voting, and N-Self Checking Programming. An Acceptance Voting pattern is described in [3]. Liu [47] presented several software versions of some of these algorithms. Saridakis presented a system of 13 patterns including most of the common mechanisms for reliability, including Fail-Stop Processor, Acknowledgement, Heartbeat, Passive Replication, and others [56]; his patterns are illustrated using a pattern diagram. Later, he described patterns for Fault Containment [57], Checkpoint-based Recovery [58], and Graceful Degradation [59].

Fault containment patterns include Input Guard, Output Guard, and Fault Container. Cecilia Rubira and his collaborators produced several papers on fault-tolerant architectures using patterns [23], [24], [46], [49]. Her paper [26] describes a reflective state pattern from which other several varieties are derived. The reflexive State pattern and others are discussed in [28]. Lemme et al. [46] describes the Fault Injector, Injector, and Monitor patterns.

Buckley presented Acknowledgement (Heartbeat), Active Replication, and Result Evaluator patterns [9]. Mwelwa and Pont [51] describe the Heartbeat pattern to estimate the health of a node, and wich provides error reporting. An Error Handler pattern is used together with a Heartbeat in [43]. Kim et al. [41] describe another version of the Heartbeat, Active Redundancy and Checkpoint patterns as well as some combinations thereof. Buckley [13] introduced a pattern to describe the sequence of actions needed to describe failures. Hoeller et al. [35] combine two patterns to obtain software diversity: Static and Dynamic randomization. These two approaches are low-cost alternatives to N-version programming. Adams et al. [2] describe two sets of availability patterns intended for fault tolerance and fault management. Jimenez-Peris et al. [38], presented a system of architectural patterns for highly available service oriented systems, including several varieties of database and session replication in addition to a pattern for multi-tier coordination.

A formal proof based on Petri nets for a pattern designed with fault-tolerant execution of parallel programs is given in [42]. Lopatkin et al. [48] shows seven patterns for representing FMEA concepts. Dyson and Longshaw described several availability patterns for Internet systems that include a variety of patterns mentioned earlier [21]. New patterns include Data Replication and Session Fail-over. Some policies for fault-tolerant telecommunication systems are given in [1]. These include Riding Over Transients, Leaky bucket counters, and several others.

A. Trad and C. Trad [62] discussed a set of patterns for Autonomous Robust Systems (ARS) that include data storage, video recognition, decision making, human robot interaction, and others. Islam et al. [37] describe the Recoverable Distributor, a pattern for fault-tolerant and state-sharing of distributed programs. The paper also shows the Distributed Observer pattern. A Fault Handler (and a sensor-actuator) are given in [43]. B.P. Douglas's book [19] includes some reliability patterns such as Watchdog, Monitor-Actuator and others. A set of patterns for software health monitoring is given in [44]; he defines a three layer software monitoring architecture and provides 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. W. Halang and his group produced several papers on UML profiles and models for fault tolerant systems: [31], [32] shows a safety shell pattern based on a reconfiguration management pattern.

The University of Newcastle PRIME project has several papers with models and patterns for fault tolerance; including a proposal for a holistic fault tolerant architecture, based on centralized fault tolerance management, with redundant functionality distributed across the entire system [29]. Rytter and Jorgensen [55] use a meta-level architecture to build fault containers which used uses the Lookout pattern as a component pattern. [61] introduced the Backup pattern that switches to a backup mode of operation. This provides redundancy in software to offer various alternatives for a function and to switch between them dynamically in response to failure. Iliasov and Romanovsky [36, 54] defined refinement patterns for fault tolerance, and added formal definitions to the Recovery Blocks and N-Version programming patterns. The paper uses the formal language B to prove correctness using automatic model transformations.

Kang and Jackson [40] proposes the concept of Trusted Base (in the form of a pattern). It provides a solution to construct a system such that the most critical requirements depend on only a small reliable subset of the system's parts, called "trusted bases". This is analogous to the concept of Trusted Computing Base (TCB) in security. The paper has several related patterns including End-to-End Check and Trusted Kernel. Harrison and Avgeriou [34] consider the use of tactics for fault tolerance in software architectures. Tactics can be considered as complements to patterns; they are "measures" or "decisions" taken to improve some quality factor. Scott and Katzman [60] show the use of a catalog of availability tactics in a real-world application. They consider tactics such as Active and Passive Redundancy, Spare, Exception Handling, Rollback, and others. The ADD method is an approach to defining a software architecture in which the design process is based on the quality 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. They illustrate its function by showing a practical application of the ADD method to a client-server system [65]. In particular, this example focuses on selecting patterns to satisfy typical availability requirements for fault tolerance. We assess and describe some reliability patterns in the next section.

IV. Evaluation of reliability patterns

In this section we provide a brief description and analysis of some fundamental reliability, fault tolerant and availability patterns. Some reliability patterns have several published descriptions. Some patterns are composite patterns which provide several reliability features in one pattern, while, others provide one distinct reliability feature. Redundancy, diversity, error detection, error masking and containment are basic properties of reliability and fault tolerance and several patterns incorporate them in different ways.

A. Reliability Patterns Descriptions

The results of applying redundancy and diversity can be evaluated with Acceptance Voting (AV), which is a hybrid pattern that incorporates N-version programming (NVP) with an acceptance test. This pattern includes N programs running in parallel to perform the same task on the same input to produce N outputs. The output of each version is presented to an acceptance test to check it for correctness [3].

The redundancy and diversity achieved using NVP at the software level can be applied to the hardware level using patterns like Triple Modular Redundancy (TMR) the Active Replication, Active-Passive Redundancy, Dual Modular Redundancy (DMR), Dynamic Dual Modular Redundancy (DDMR), Homogeneous Redundancy, Heterogeneous Redundancy, and N-Modular Redundancy (NMR) provide redundancy and sometimes diversity in addition to other reliability features. 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. If any one of the three systems fail, the other two systems can correct and mask the fault. If the voter fails then the complete system will fail [64]. This pattern provides error detection and masking. Another variation of TMR is the Triplicated Voters Triple Modular Redundancy (TV-TMR), which provides redundancy by using three voters/comparators instead of one to vote on the input provided by three identical modules to produce one output. The voter in the TMR pattern represents a single-failure point and this pattern avoids this weakness. Having three voters enables voting to take place even if one voter fails [18].This pattern provides error detection, error masking and uses redundancy.

The Active Replication pattern is another name for TMR. The outputs from all replicas are compared to determine the correct output. Only one processor error can be masked at a time [9]. N-Modular Redundancy (NMR) pattern is an abstraction of TMR and uses N instances of the same module to perform the same computation and then a majority vote of the output(s) is taken. As long as N/2 modules compute the output properly, the system output is correct [19]. This pattern uses redundancy, and provides error detection and masking.

The Active-Passive Redundancy pattern is used to provide redundancy in a system where performance cannot be compromised. Redundancy is added to the critical part of the system which may potentially act as a single point of failure in the system. This critical part of the system is provided with a standby replica which will be activated in case failure of the former occurs. The client to the failed part should be informed about the passive part's activation [2]. Dual Modular Redundancy (DMR) pattern uses two replications working in parallel to carry out a process [8]. This is a special case of Active Replication and NMR. The Dynamic Dual Modular Redundancy (DDMR) pattern allows an operating system to schedule redundant threads on any two cores/processors within a group of cores/processors. DDMR is a scalable dynamic DMR approach and may be used in symmetric shared-memory architectures as well as in distributed shared-memory architectures.

The Heterogeneous Redundancy pattern detects and handles systematic errors and random failures in a system. It provides fault safety in the same way as the Homogeneous Redundancy pattern, that is, when the primary channel detects a fault, the secondary channel takes over [8]. This pattern provides error detection, error masking and uses redundancy. Homogeneous Redundancy uses multiple channels which operate in sequence, much like the Switch to Backup Pattern (another alias for this pattern), or in parallel, as in the TMR pattern [64]. Since the redundancy is homogeneous, by definition any systematic fault in one copy of the system is replicated in its clones, so it provides no protection against systematic faults [7].

Some patterns offer multiple features to achieve reliability, while other patterns offer one or two primary reliability features. For example error detection or monitoring is typically achieved with the use of the Acknowledgement, Watchdog, Fault Injection, and Riding Over Transients patterns. The acknowledgement pattern detects errors in a system by acknowledging the reception of an input within a specified time interval. It acknowledges receipt of an input within a specified time interval without increasing the time overhead significantly.

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 [9].

Similarly, the Fault Injection pattern is used to evaluate the behaviour of computing systems in the presence of faults. It adopts a technique that tries to produce or simulate faults during an execution of the system under test, to observe the system's behaviour. This allows for the injection of faults, to monitor the system under test; system activation, and inform the user about the test results, as well to receive user requests [6]. Riding Over Transients pattern detects temporally dense events by allowing a system to roll through problems without users noticing them and without the aid of the machine operator intervening. This pattern resolves errors with minimal effort by first determining whether a problem actually exists [1] .

The Failover Cluster, Leaky Bucket, Protected, Process Pairs, Single Channel, Recovery Blocks, Recoverable Distributor, and Reliable Hybrid patterns are used to achieve high availability in a system. The Failover Cluster pattern provides protection against loss of service of a single server (single point of failure) in high available application infrastructures. 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. When a failover occurs, users continue to use the application and are unaware that a different server is providing it [17].

Another variation of this pattern is the Leaky Bucket Counters pattern which handles isolated errors by taking devices out of service. A counter is initialized to a predetermined value and 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 [12]. This pattern provides error masking and error containment. Similarly, the Process Pairs pattern passes information about its new consistent state to a backup server. As such, when the primary server successfully completes an entire transaction, both the primary and backup servers record these data in their persistent mass-storage devices. In this way, the backup server is kept current about completed transactions. While the primary server is available to clients, it sends regular heartbeat messages to the backup server. If the backup server detects that the stream of heartbeat messages has stopped, it understands that the primary server is dead or unavailable, and it will take over as a new primary server [15]. This pattern provides error detection and uses redundancy. The Protected Single Channel pattern uses a single channel to handle sensing and actuation in a system. Reliability is enhanced through the addition of checks at key points in the channel, which may require some additional hardware. The Protected Single Channel Pattern will not be able to continue to function in the presence of persistent faults, but it detects and may be able to handle transient faults [7].

The Recovery Blocks pattern provides error detection and error masking by performing an acceptance test after every processing alternative is tried. In this way, processing alternatives are run until a processing alternative succeeds in delivering results that pass the acceptance test [15]. The Recoverable Distributor pattern is a composite pattern for distributed systems that combines fault detection, containment and recovery. It has two important properties, one is masking processor failures; that is, it must be able to preserve the state of the system in spite of such failures. It also hides network latency while providing consistent access to the shared state of all processors in the system [14]. It has a state management section (local and global) and a fault detection and recovery section and enables creation of local views of shared data.

The Reliable Hybrid pattern provides a general object-oriented framework for fault tolerance which can range from basic approaches. It is a combination of several fault tolerance patterns to support development of applications based on classical fault tolerant strategies. This pattern provides error detection, error masking, recovery and uses diversity.

Some patterns offer multiple reliability features in one; however, patterns can be designed to combine reliability and security, safety and other dependability properties. We describe a few of them below. The Reliable Security pattern performs reliable authorization enforcement by applying reliability to a reference monitor and to a set of authorization rules that enforce security [4]. All user requests must be authorized based on the user's rights. A reference monitor is used to enforce authorization. This pattern provides security and error masking.

The Secure Reliability pattern controls the use of reliable services in a system. A strategy based system receives a request and selects the appropriate reliability service to process that request independently. All user requests are authorized based on the user's role. A role-based access control model manages a user's rights in the system. The response to the request is either completed or rejected [4]. The WS-Reliability pattern provides error detection and uses redundancy. It ensures that a notification is always sent in response to a failure, it provides guaranteed message delivery, message ordering, and duplicate elimination when messages are sent from one entity to another. 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) that send, deliver, order and eliminate duplicate messages [3]. This pattern provides error detection and redundancy in web services.

The WS-ReliableMessaging provides error detection and security. It helps to ensure 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. It uses a protocol that performs guaranteed receipt, ordered delivery, state disposition, and duplicate elimination of messages. This is achieved by first having an agreement which includes a policy exchange, endpoint resolution and establishment of trust between end points [11].

B. *Reliability Pattern Evaluation*

We analyze some reliability patterns using a notation based on [63], which is helpful to identify quality aspects of patterns:

U, **under-specified or incomplete**. The pattern does not use an appropriate template (we use as a guideline the POSA 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. Over specification may reduce the use of a pattern.

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 only using words but words can be misleading or vague. An incomplete 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 it is defined in an ad hoc way. This makes the pattern difficult to be used 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.

TABLE I

Reliability Pattern	Reliability Property	Quality Indicator
Reliable Hybrid [16]	• Alerting • Error Detection • Redundancy • Error Masking • Fault Containment	▪ P, U, N ▪ Insufficient UML diagrams ▪ Incomplete Pattern ▪ Incorrect UML notation
Recoverable Distributor[37]	▪ Redundancy ▪ Error Masking	▪ P, U, N, M ▪ Incomplete pattern ▪ Incorrect UML Notation
Acknowledgement [9]	▪ Error Detection ▪ Alerting	▪ Complete
WS-Reliability [11]	▪ Error Detection ▪ Alerting ▪ Error Masking	▪ Complete

Based on our analysis, we observed that many of the reliability patterns (including the variations discussed earlier) are incomplete and lack the necessary details to be helpful. In particular, two of the patterns illustrated in Table 1 do not conform to the POSA or GOF pattern templates. It was also noted that many patterns use different notations to describe their structures, which makes them more difficult to interpret and apply. It can also be noted that many of the patterns perform the same functions either atomically or using a composite approach. The analysis of these patterns showed the need to refine many of the existing patterns to make them more concise, useful, and to make them more complete in their description and representation with the use of UML diagrams and by using the POSA pattern template.

V. Conclusions and Future work

This initial survey and analysis has shown that many of the reliability patterns are incomplete, ambiguous and lack necessary details. This survey has highlighted the prospect of combining some reliability patterns with security, and safety to create more versatile composite patterns, as opposed to implementing these fundamental properties (safety, security, fault tolerance reliability) separately. This survey provides a basis for a catalog that will contain a unified set of reliability patterns.

References

[1] M.Adams, J. Coplien, R. Gamboke, R. Hanmer, F. Keeve and K. Nicodemus, "Fault-Tolerant Telecommunication System Patterns", in Pattern Languages of Program Design 2, 549 - 562 Addison-Wesley Longman Publishing Co, 1996.

[2] K. S. Ahluwalia, A. Jain, "High Availability Design Patterns", Proc. PLoP '06, October 21–23, 2006, Portland, OR, USA.

[3] A. Armoush, F. Salewski, and S. Kowalewski, "Design Pattern Representation for Safety-Critical Embedded Systems", Journal Software Engineering and Applications, vol. 2, 2009, 1-12.

[4] P. Avgeriou, "Describing, instantiating and evaluat-ing a reference architecture: A case study", Enterprise Archi-tecture Journal, June 2003.

[5] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. "Basic Concepts andTaxonomy of Dependable and Secure Computing", IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, pp. 11-33, Jan.-Mar. 2004.

[6] A. Ayoub, B. G. Kim, I. Lee, O. Sokolsky, "A safety case pattern for model-based development approach", NASA Formal Methods Symposium, April 2012.

[7] S. Bernardi, J. Merseguer, D. C. Petriu, "A dependability profile within

MARTE", Proc. Soft. Syst. Model., 313-336, 2011. doi: 10.1007/s10270-009-0128-1.

[8] S. Bernardi, J. Merseguer, D. C. Petriu, "Dependability modeling and analysis of software systems specified with UML", ACM Computing Surveys, Vol. 45, Issue 1, November 2012, doi>10.1145/2379776.2379778.

[9] I. A. Buckley and E. B. Fernandez, "Three patterns for fault tolerance", Proc. OOPSLA MiniPLoP, October 26, 2009.

[10] I. A. Buckley, E. B. Fernandez, G. Rossi, and M. Sadjadi, "Web services reliability patterns", Proc. 21st International Conference on Software Engineering and Knowledge Engineering (SEKE'2009), Boston, 4-9 July 1-3, 2009.

[11] I. Buckley and E. B. Fernandez, "Patterns Combing Reliability and Security", Proc. Third International Conferences on Pervasive Patterns and Applications, September 25-30, 2011.

[12] I. Buckley, E. B. Fernandez, M. Anisetti, C. A. Ardagna, M. Sadjadi, and E. Damiani, "Towards Pattern-based ReliabilityCertification of Services, Proc. 1st International Symposium on Secure Virtual Infrastructures (DOA-SVI'11), Vol. 7045, Springer Lecture Notes in Computer Science, 17-19 Oct. 2011.

[13] I. Buckley and E. B. Fernandez, "Failure patterns: A new way to analyze failures", Proc. First International Symposium on Software Architecture and Patterns in conjunction with the 10th Latin American and Caribbean Conference for Engineering and Technology, July 23-27, 2012.

[14] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, "A System of Patterns: Pattern-Oriented Software Architecture", John Wiley & Sons, 1996.

[15] Y. Choi, "Early Safety Analysis: from Use Cases to - Component-based Software Development", Journal of Object Technology, vol. 6, no. 8, 185-203, 2007. http://www.jot.fm/issues/issue_2007_09/article4.

[16] F. Daniels, K. Kim, and M.A. Vouk, "The Reliable Hybrid pattern- A generalized software fault tolerant design pattern",Proc. of PLoP'97,1997. http://hillside.net/plop/plop97/Proceedings/daniels.pdf

[17] O. Daramola, G. Sindre, T. Stålhane, "Pattern-based security requirements specification using ontologies and boilerplates", Proc. Second International Workshop on Requirements Patterns (RePa '12), 54-59, 2012.

[18] A. L. De Oliveira, R. T. V. Braga, P. C. Masiero, I. Habli, T. Kelly, "A pattern to argue the compliance of system safety requirements decomposition", Proc. 10th Conf. on Pattern Languages of Programs (SugarLoafPLoP) 2014, November 9-12, 2014.

[19] B. F. Douglass, "Doing Hard-Time: Using Object-Oriented Programming and Software Patterns in Real-Time Applications", Addison-Wesley, 1998.

[20] B. F. Douglass, "Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems", Addison-Wesley Professional, 2003.

[21] P. Dyson, A. Logshaw, "Patterns for high-availability Internet systems", Proc. EuroPLoP 2002.

[22] E. B. Fernandez, "Security patterns in practice: Building secure architectures using software patterns", Wiley Series on Software Design Patterns, 2013.

[23] L. L. Ferreira, C. M. F. Rubira , M. F. Rubira , "The Reflective State Pattern", Proc. PLoP'98, 1998.

[24] L. L. Ferreira and C. M. F. Rubira, "Reflective design patterns to implement fault tolerance", Proc. OOPSLA Workshop on Reflective Programming 1998, http://www.csq.is.titech.ac.jp/~chiba/oopsla98/ferreira.pdf

[25] M. Fowler, "Analysis patterns -- Reusable object models", Addison-Wesley, 1997.

[26] E., Gamma, R. Helm , R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Reading Mass., Addison Wesley, 1994.

[27] R. Gamoke, "Pattern: Leaky bucket counters", Proc. Fault- Tolerant Telecommunication System Patterns, Report, AT&T Bell Laboratories, 1995.

[28] H. Gawand, R. S. Mundada, and P. Swaminathan, "Design Patterns to Implement Safety and Fault Tolerance", International Journal of Computer Applications (0975 – 8887), vol.18, No.2, March 2011.

[29] R. Gensh, A. Rafiev, A. B. Romanovsky, A. F. Garcia, F. Xia, A. Yakovlev, "Architecting Holistic Fault Tolerance", Proc. HASE, 5-8, 2017.

[30] A. Golander, S. Weiss, and R. Ronen, "DDMR: Dynamic and Scalable Dual Modular Redundancy with Short Validation Intervals", Proc. IEEE Computer Architecture Letters, vol. 7, no. 2, pp. 65–68, Feb. 2008, doi:10.1109/L-CA.2008.12.

[31] R. Gumzej, W. A. Halang, "A safety shell for UML-RT projects structure and methods of the corresponding UML pattern", Innovations in Systems and Software Engineering, Volume 5, Issue 2, 97–105, 2009.

[32] R. Gumzej, M. Colnaric, W. A. Halang, "A reconfiguration pattern for distributed embedded systems", Proc. SoSym, 145-161, 2009.

[33] R. S. Hanmer, "N-Version Programming", Proc. PLoP 2009.

[34] N. Harrison and P. Avgeriou, "Incorporating Fault Tolerance Tactics in Software Architecture Patterns", Proc. RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems, November 17-19, 2008.

[35] A. Hoeller, T. Rauter, J. Iber, C. Kreiner, "Patterns for automated software diversity", Proc. 20th European Conf. on Pattern Languages of Programs (EuroPLoP 2015), 2015.

[36] A. Iliasov and A. Romanovsky, "Refinement patterns for fault tolerant systems", Proc. EDCC 2008, 167-176, 2008.

[37] N. Islam and M. Devarakonda, "An essential design pattern for fault-tolerant distributed state sharing", Comm. of the ACM, vol. 39, No. 10, 65-74, 1996.

[38] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, F. Perez-Sorrosal, D. Serrano, "A System of Architectural Patterns for Scalable, Consistent and Highly Available Multi-Tier Service-Oriented Infrastructures", Proc. WADS2008, 1-23, 2009.

[39] D. Kalinsky Associates, "Design Patterns for High Availability",whitepaper,2002. http://www.kalinskyassociates.com/Wpaper6.html.

[40] E. Kang, D. Jackson, "Patterns for building dependable systems with trusted bases", Proc. 17th Conference on Pattern Languages of Programs (PLOP '10). ACM, 2010, http://dx.doi.org/10.1145/2493288.2493307.

[41] S. Kim, D.-K. Kim, L. Lu, S. Park, "Quality-driven architecture development using architectural tactics", Journal of Systems and Software, 2009.

[42] E. Kindler, and D. Shasha, “Verifying a design pattern for the fault-tolerant execution of parallel programs”, Technical Report, No. TR2000-803. New York University, 2000.

[43] S. Konrad, B.H.C.Cheng, “Requirements patterns for embedded systems”, Proc. IEEE Joint Int. Conf. on Reqs. Eng. (RE’02), 2002.

[44] A. Lau, R. E. Seviora, “Design Patterns for Software Health Monitoring”, Proc. 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05), 2005.

[45] M. Laverdiere, A. Mourad, A. Hanna, M. Debbabi, “Security design patterns: Survey and evaluation”, Proc. Canadian Conference on Electrical and Computer Engineering(CCECE’06), IEEE, 1605–8, 2006.

[46] N. G. M. Leme, E. Martins, C.M. F. Rubira, “A Software Fault Injection Pattern System”, Proc. PLoP 2001.

[47] C. Liu, “A general framework for software fault tolerance”, Proc. Workshop on fault- tolerant parallel and distributed systems”, 1992.

[48] I. Lopatkin, A. Iliasov, A. B. Romanovsky, Y. Prokhorova, E. Troubitsyna, “Patterns for Representing FMEA in Formal Specification of Control Systems”, Proc. HASE’11: 146-151, 2011.

[49] E. Martins, C. M. F Rubira, N. G. M. Leme, “A reflective fault injection tool based on patterns”, Proc. Int. Dependable Systems and Networks (DSN’02), 2002.

[50] Microsoft, “Performance and Reliability Patterns - Failover Cluster”, Microsoft Patterns and Practices –Proven practices for predictable results, MSDN, 2014. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff650328(v=pandp.10)

[51] C. Mwelwa, M. J. Pont, “Two Simple Patterns to Support the Development of Reliable Embedded Systems”, Proc. of Viking PLoP 2013.

[52] V. P. Nelson, “Fault-Tolerant Computing: Fundamental Concepts”, IEEE, vol.23 n.7, pp.19-25, July 1990, doi:10.1109/2.56849.

[53] J. V. Neumann, “Probabilistic logics and the synthesis of reliable organism from unreliable components,” Proc. Automata Studies, Princeton Univ. Press, pp. 43–98, 1956.

[54] A. Romanovsky, and A. Iliasov, "Refinement Patterns for Fault Tolerant Systems", Proc. 2008 Seventh European Dependable Computing Conference, 167-176, 2008. doi:10.1109/EDCC-7.2008.18

[55] M. Rytter, B. N. Jørgensen, “Enhancing NetBeans with Transparent Fault Tolerance Using Meta-Level Architecture”, Journal of Object Technology, 9(5), 55-73, 2010.

[56] T. Saridakis, “A System of Patterns for Fault Tolerance”, Proc, EuroPLoP 2002.

[57] T. Saridakis, “Design Patterns for Fault Containment”, Proc. EuroPLoP, 2003.

[58] T. Saridakis, “Design Patterns for Checkpoint-Based Rollback Recovery”, Proc. EuroPLoP, 2003.

[59] T. Saridakis, “Design Patterns for Graceful Degradation”, Proc. Trans. Pattern Languages of Programs, 67-93, 2009.

[60] J. Scott, R. Kazman, “Realizing and refining architectural tactics: Availability”, Tech. Rept. CMU/SEI-2009-TR-006, August 2009.

[61] S. Subramanian, W. Tsai, “Backup Pattern: Designing Redundancy in Object-Oriented Software”, Pattern Languages of Program Design, Addison-Wesley 1996.

[62] A. Trad, C. Trad, “Audit, control and monitoring design patterns (ACMDP) for autonomous robust systems (ARS)”, Proc. Int. J. of Advanced Robotic Systems, vol. 2, No 1, 25-38, 2005.

[63] A.V. Uzunov, E. B. Fernandez, K. Falkner, "Securing distributed systems using patterns: A survey", Computers & Security, 31(5), 681–703, 2012. doi:10.1016/j.cose.2012.04.005

[64] J. F. Wakerly, “Microcomputer Reliability Improvement Using Triple Modular Redundancy”, IEEE Transactions on Computers, 889-895, 1976. doi: 10.1109/T-C.1975.224263.

[65] W. G. Wood, “A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0”, Tech. Rept. CMU/SEI-2007-TR-005ESC-TR-2007-005, 2007. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8319