Building a Safety Architecture Pattern System

CHRISTOPHER PRESCHERN, NERMIN KAJTAZOVIC and CHRISTIAN KREINER,
Institute for Technical Informatics, Graz University of Technology

Safety architecture patterns provide knowledge about large scale design decisions for safety-critical systems. They provide good ways to avoid, detect, and handle faults in software or hardware. In this paper we revise existing architectural safety patterns and organize them to build up a pattern system. We add Goal Structuring Notation diagrams to the patterns to provide a structured overview of their architectural decisions. Based on these diagrams we analyze and present relationships between the patterns. The diagrams can also be used to argue about a systems's safety, which we show with an example.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architecture—*Patterns*; K.4.1 [**Public Policy Issues**] Human Safety; K.6.5 [**Management of computing and information systems**] Security and Protection
ACM Reference Format:

Preschern, C., Kajtazovic, N. and Kreiner, C. 2015. Building a Safety Architecture Pattern System. EuroPLoP '13: Proceedings of the 18th European Conference on Pattern Languages of Program, Article 17 (July 2013) , 55 pages. ACM.

1. INTRODUCTION

Safety-critical systems can directly harm humans or machinery if they malfunction. To ensure that these systems operate properly, they often have to be certified and developed according to safety standards. Safety standards usually provide a big pool of requirements and techniques to achieve system safety. For system architects which are new to the safety domain, it is often difficult to chose which of the provided techniques or which overall system architecture should be used to achieve a safety goal.

To provide safety architects with knowledge about good solutions, we construct a system of architectural safety patterns[1]. We present a structured way how we build up this pattern system from existing safety patterns found in literature. Additionally, we extend these patterns with Goal Structuring Notation (GSN) diagrams, which present the main architectural decisions of the patterns. These diagrams provide a safety architect with a structured approach to argue for the overall system safety. We show with an example how the GSN diagrams can even be used to relate architectural design decisions in the patterns to requirements and techniques of the IEC 61508 safety standard. This approach can as well aid safety architects for arguing for the system safety in particular in the context of safety certification.

This paper is structured as follows: Section 2 gives an overview of existing safety patterns and approaches to build safety pattern systems/languages. Section 3 presents the pattern format we use for our pattern system and Section 4 shows how we bring the patterns into this format by the example of the TRIPLE MODULAR REDUN-

[1] A “pattern system” is similar to a “pattern language”, but compared to a pattern language it does not claim to be complete [Buschmann et al., 1996]. Precise definitions about the difference between pattern collections/systems/languages can be found in [Schumacher, 2003]

EuroPLoP '13, July 10 - 14, 2013, Irsee, Germany
 ACM 978-1-4503-3465-5/15/07...$15.00 DOI: http://dx.doi.org/10.1145/2739011.2739028

DANCY pattern. Section 5 presents how our patterns are connected to a pattern system and Section 6 shows the application of the patterns to highlight the benefits of the introduced GSN diagrams. Section 7 concludes this work. In Appendix A, all the patterns of the pattern system are presented, Appendix B shows a collection of safety tactics, and Appendix C shows how we analyzed our patterns to obtain the tactics they use.

2. RELATED WORK

In this section we give an overview of related work that introduces safety patterns (see Table I) and we present related work that collects and structures existing safety patterns.

Table I. Literature which introduces safety-related patterns

Title	**Description**
[Daniels et al., 1997] *"The Reliable Hybrid Pattern - A Generalized Software Fault Tolerant Design Pattern"*	A pattern which includes software fault tolerance techniques (e.g. N-version programming, voting, acceptance test) is presented. The pattern is presented as a generic architecture which explicitly states alternatives in the pattern (e.g. use voting instead of an acceptance test).
[Douglass, 1998] *"Safety-Critical Systems Design"*	The article covers safety architecture patterns and discusses how they can be implemented.
[Saridakis, 2002] *"A System of Patterns for Fault Tolerance"*	This paper introduces several architectural fault-tolerance patterns and discusses how to group them.
[Douglass, 2002] *"Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems"*	Besides other patterns, this book covers safety-related architecture patterns and also includes the patterns from [Douglass, 1998].
[Grunske, 2003] *"Transformational Patterns for the Improvement of Safety Properties in Architectural Specification"*	This paper presents patterns for architecture transformations to increase the overall system safety. Some of the patterns are related to the patterns from [Douglass, 2002].
[Hanmer, 2007] *"Patterns for Fault Tolerant Software"*	The book provides a pattern language of fault-tolerance patterns grouped as error detection, error processing, error mitigation, fault treatment, and architectural patterns.
[Douglass, 2010] *"Design Patterns for Embedded Systems in C"*	The book presents design patterns implemented in C. Some of the presented safety-related patterns come from [Douglass, 2002].
[Armoush, 2010] *"Design Patterns for Safety-critical Embedded Systems"*	This PhD thesis introduces new safety patterns and provides and collects existing safety patterns for embedded systems (mostly [Douglass, 2002] for hardware patterns and software fault tolerance techniques from [Pullum, 2001] brought into pattern notation for software patterns).
[Hampton, 2012] *"Survey of Safety Architectural Patterns"*	This survey presents the application of the patterns from [Armoush, 2010] within a company. Furthermore, some new and rather domain-specific safety patterns are introduced.
[Rauhamäki et al., 2012] *"Architectural Patterns for Functional Safety"*	The paper presents 4 patterns related to separating the safety functionality from non-critical functionality.
[Rauhamäki et al., 2013] *"Patterns for Safety and Control System Cooperation"*	The paper presents 3 safety patterns related to the control systems domain.
[Rauhamäki and Kuikka, 2013] *"Patterns for Controlling System Safety"*	The paper presents 4 safety patterns related to the control systems domain.

[Saridakis, 2002] presents several fault-tolerance patterns in detail and discusses how they can be related to each other. The patterns are classified according to several criteria: pattern complexity, space requirements, time requirements, failure types which are handled by the pattern, and the pattern aim (error detection, recovery, or masking). [Hanmer, 2007] also describes fault-tolerance patterns and presents the patterns and their relationships as a pattern language.

[Armoush, 2010] provides in his PhD thesis a comprehensive collection of safety architecture patterns for embedded systems. Most of the patterns are taken from literature and all are presented in a common pattern format. However, the relationships between the patterns are not described in detail. Armoush provides a tool which lists the patterns and provides detailed information about them (e.g. reliability calculations) when selected.

To bridge the gap between the high-level safety pattern descriptions and their actual implementation, [Gawand et al., 2011] represent safety patterns in UML notation. This is also done by [Sarma et al., 2013] with the pattern catalog of [Armoush, 2010]. This idea was taken further by [Antonino et al., 2012] who introduce a safety-related UML profile to capture architectural safety pattern elements (e.g. voter) and to define rules for them. Based on this idea [Olivera, 2012] implements a repository for safety patterns in UML notation.

The TERESA project applies a model-based approach coupled with a repository of safety/security patterns for embedded systems engineering. A generic metamodel for safety/security patterns is defined which can be used to model domain-specific patterns in the repository [Desnos et al., 2012]. The pattern repository can be accessed with an Eclipse plugin as described on the TERESA project homepage (www.teresa-project.org).

3. APPLIED PATTERN FORMAT

We use the pattern format presented by [Babar, 2007] for all our safety architecture patterns. The pattern format of Babar explicitly provides architectural information with the aim to aid architecture design and evaluation processes. Table II shows which sections this pattern format contains and where we got the information for these sections from. Most of our patterns are based on [Armoush, 2010] and are further elaborated by using other literature on similar safety patterns. For example, Armoush describes the WATCHDOG pattern, which is also described by [Grunske, 2003]. We take Armoush's Watchdog pattern as a starting point and enhance it with information from Grunske. In particular, in this case, we add Grunske's forces, because they are better elaborated.

Table II. Pattern format for our safety architecture patterns

Section	What it contains and where the contents comes from
Pattern Name	The pattern name is taken from the existing pattern - most of which come from [Armoush, 2010].
Pattern Type	Classification into hardware/software and fail-safe/fail-over. This classification comes from the pattern types which we classify during the process of building up the pattern language.
Also Known As	Other names for the pattern used in literature.
Context	The contents of this section comes from existing patterns and was structurally adapted to fit our pattern system.
Problem	The contents of this section comes from existing patterns and was structurally adapted to fit our pattern system.
Forces	The contents of this section comes from existing patterns (mostly from [Grunske, 2003]) and was structurally adapted to fit our pattern system.
Solution	The solution is shortly described in a few sentences and the structure of the safety architecture is shown in a diagram. Most of the diagrams are based on [Armoush, 2010] and [Douglass, 2002].
GSN Diagram	This section contains a Goal Structuring Notation (GSN) diagram which relates the main aim of the pattern to the architectural design decisions which were taken to achieve this aim. GSN is a graphical notation which is often used in the safety domain to describe how a certain goal is achieved. The advantage of using this notation is that it is familiar to safety experts and the resulting pattern GSN diagram can be used to structurally argue about a system's safety. Figure 1 shows the basic elements of GSN and explains them. The GSN diagram is based on information about the usage of basic architectural design decisions (architectural tactics) which are applied in the pattern. We obtain these tactics from pattern descriptions according to a method presented by [Kumar and Prabhakar, 2010b] which we will cover in more detail in the next section.
Consequences	The consequences are split into a part containing general consequences and a part explicitly covering quality-attribute related consequences (e.g. consequences on safety or availability). The information about the consequences mostly comes from the safety patterns from [Armoush, 2010] and [Grunske, 2003].
General Scenarios	This section contains scenarios of the system which can, for example, be used during architecture evaluations. The scenarios are mined from patterns as suggested in [Babar, 2007] by manually searching the problem and solution statements for scenarios for relevant quality attributes (in our case focused on safety). Scenarios are included in the patterns because there are existing safety reasoning frameworks which are based on scenarios ([Wu, 2007]) and the information of the scenarios is also needed to build up our GSN diagrams.
Known Uses	This section presents known uses for the patterns. We added this information by searching for literature which applies the pattern. We just included patterns for which we could find at least three known uses.
Credits	References to previous work on the pattern.

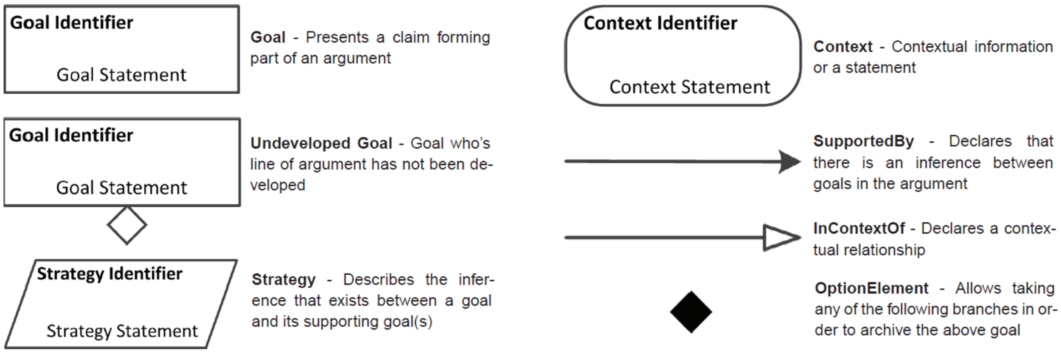


Fig. 1. Explanation of the basic GSN elements

4. SPECIFYING A PATTERN IN THE PROPOSED PATTERN FORMAT

In this section we show how we specify a pattern for our pattern system with the example of the TRIPLE MODULAR REDUNDANCY (TMR) pattern. We focus on how the **GSN Diagram** is built and also describe how we obtain the **General Scenarios** for a pattern.

4.1 Mining Tactics from the Pattern Descriptions

The TMR pattern is mentioned in literature by [Douglass, 2002] and [Armoush, 2010]. We studied both TMR patterns to find text passages which indicate the usage of general safety-related architectural design decisions (safety tactics). We do this as proposed by [Kumar and Prabhakar, 2010a], where architectural tactics are mined from GoF and POSA patterns to find relationships between patterns which use similar tactics. We apply the same method in order to find relationships between safety architecture patterns.

As proposed by [Kumar and Prabhakar, 2010a], we construct a table which includes text passages of the pattern and we give the corresponding tactic that this text passage relates to. For example, the TMR pattern in [Armoush, 2010] says: "*The voter plays a main role in this pattern by applying the voting policy to take the majority from the results which represents the correct actual result.*" This indicates that the pattern applies the *Voting* safety tactic[2].

Table III shows which tactics were mined for the TMR pattern.

4.2 Building the Tactic Topology Model

With the gathered tactics we construct a *Tactic Topology Model* with is also part of the method described by [Kumar and Prabhakar, 2010a]. First, one has to think about the main goal of the pattern (usually found in the patterns' **Intent** section). According to [Kumar and Prabhakar, 2010a], the main tactics which achieve this goal are usually related to the **Intent** or the **Problem** section of the pattern. In the Tactic Topology Model, these main tactics are connected to the patterns' goal with arrows. Further explanation about this connection is given in textual form next to the arrow. The tactics can bring up new goals which have to be achieved by additional tactics - these are also added with arrows and a textual description. In that way, a structured graph containing the patterns' tactics is constructed.

Figure 2 shows the Tactic Topology Model for the TMR pattern. We use the Tactic Topology Models to structurally establish relationships in our pattern system (this is explained in Section 5). Apart from that, the Tactic Topology Models are just intermediate results used to build GSN diagrams and are not included in the patterns.

[2] A list of all safety tactics is available in Appendix B

Table III. Determining the tactics used by the TMR patterns described in [Douglass, 2002] and [Armoush, 2010]

Abstract Section		
Core Intent		**Tactic**
This pattern consists of three identical modules operating in parallel to produce three results that are compared using a voting system to produce a common result		Voting Replication Redundancy
Problem Section		
Problem	**Elaboration of Problem (Scenario)**	**Achieved through Tactic**
How to deal with random faults and single-point of failure in order to increase the safety and reliability of the system without losing the input data in the presence of faults.	The system is fully operational even in case of a single channel failure. A single channel random fault does not lead to a system failure.	Voting
Solution Section		
Solution Description		**Tactic**
The system contains three identical modules or channels operating in parallel. *Test by redundant hardware*		Replication Redundancy
The voter plays a main role in this pattern by applying the voting policy to take the majority from the results which represents the correct actual result. *Fault detection and diagnosis (Voting)*		Voting
Consequences Section		
Consequence Description		**Tactic**
This pattern has a high recurring cost due to the using of three parallel modules. So, the recurring cost is 300% comparing to the basic system.		Replication Redundancy
The cost of voter which is normally a simple hardware circuit that depends on the type of the output control signal and the implementation method.		Voting
Implementation Section		
Implementation Description		**Tactic**
To implement this pattern, the designer should replicate the channel which includes the replication of the hardware as well as software.		Replication Redundancy

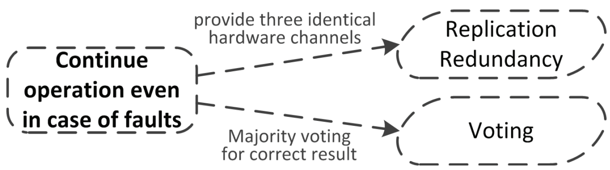


Fig. 2. Tactic Topology Model for the TMR pattern

4.3 Building the Goal Structuring Notation Diagram

Based on the Tactic Topology Model, we construct the GSN diagrams for the patterns. The GSN diagrams contain the tactics from the Tactic Topology Model and they additionally contain general scenarios which are mined from the pattern descriptions. This scenario mining is done as proposed in [Babar, 2007] by searching the problem and solution statements for safety-related scenarios. The scenarios found for the TMR pattern are shown in Table III under *Problem Section*. All our GSNs start with the main goal to maintain system safety. This main goal is split up into subgoals with the scenarios which we obtained from the patterns. If the scenarios are independent from each other, then they are put on the same level in the GSN. If a scenario depends on another scenario (as it is the case for the TMR pattern), then it is modeled as a subgoal of the scenario it depends on. The tactics which are necessary to achieve a GSN goal are put below this (sub-)goal as a GSN strategy which has the title of the tactic and which contains additional information (taken from the textual description of the Tactic Topology Model arrow connections) as GSN strategy description. GSN context elements are added to the GSN diagram if information of the pattern's context section is relevant for the GSN goals.

Figure 3 shows the GSN diagram of the TMR pattern. We can see that it consists of the tactics of the TMR pattern Tactic Topology Model from Figure 2 and of the scenarios of the TMR pattern from Table III. The complete TMR pattern including the here constructed GSN diagram is shown in Appendix A on page 17.

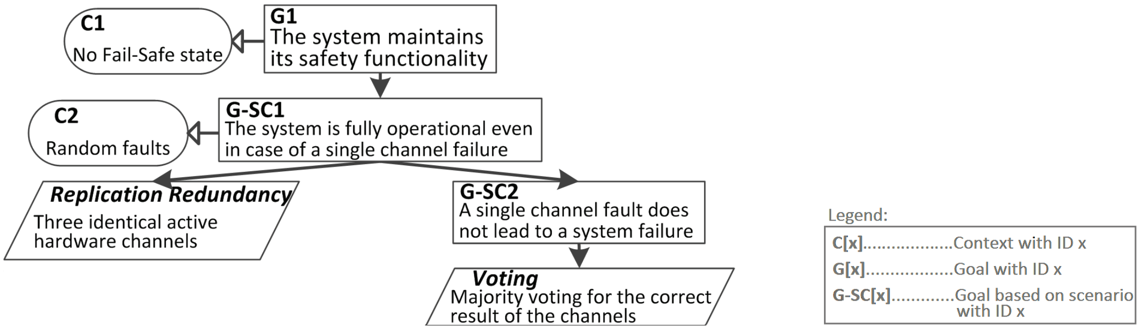


Fig. 3. GSN diagram of the TMR pattern

5. ORGANIZING SAFETY PATTERNS TO A PATTERN SYSTEM

To obtain the relationships between the patterns, we use the approach presented by [Kumar and Prabhakar, 2010b]. They compare Tactic Topology Models of patterns and define a mapping between Tactic Topology Model predicates and pattern relationships. For example, if the Tactic Topology Models of two different patterns are equal, then Kumar and Prabhakar say that these patterns are similar. Table IV shows all kinds of relationships defined by Kumar and Prabhakar. To find all relationships in a pattern system, every patterns' Tactic Topology Model has to be compared to the Tactic Topology Models of all other patterns and every such Tactic Topology Model pair has to be checked for all the predicates described in Table IV.

Table IV. Description of pattern relationships (slightly modified from [Kumar and Prabhakar, 2010a])

Relationship	Description	Tactic Topology Model predicate
is an alternative	Patterns A and B solve the same problem, but propose different solutions.	$SourceNode(A) = SourceNode(B)$ **AND** $Graph(A) \neq Graph(B)$
uses	A sub-problem of pattern A is similar to the problem addressed by pattern B.	$Graph(A) \supset Graph(B)$
refines	Pattern B provides a more detailed solution than pattern A.	$SourceNode(A) = SourceNode(B)$ **AND** $Graph(A) \subset Graph(B)$
specializes	The solution of pattern B is a special case of the solution of pattern A. *Example: Pattern B specializes pattern A if they have the same graph structure, but pattern B uses a refined tactic where pattern A uses a more general tactic (e.g. B uses Replication Redundancy where A uses Redundancy).*	$Graph(A) \subset generalizedGraph(B)$
is similar	Patterns A and B provide the same solution to a similar problem *Example: Pattern B is similar to pattern A if they have the same graph structure and they use two related refined tactics. E.g. A uses Replication Redundancy and B uses Diverse Redundancy*	$generalizedGraph(A) \equiv generalizedGraph(B)$

We applied this approach to our safety patterns to structurally build the relationships in our pattern system. We built the Tactic Topology Models as described in Section 4 for all our patterns. Then we compared each Tactic Topology Model with one another and checked for the predicates defined in Table IV. This delivers us the relationships between all the patterns for our pattern system.

Figure 4 shows our safety patterns and their relationships which we obtained with the described approach. We can see that the approach to find pattern relationships worked out pretty well for our safety patterns. All the relationships between the patterns seem to be comprehensible. For example, according to the relationships obtained through the Tactic Topology Model comparison, the TRIPLE MODULAR REDUNDANCY pattern is a specialization of the M-OUT-OF-N pattern and is similar to the N-VERSION PROGRAMMING pattern which is both reasonable.

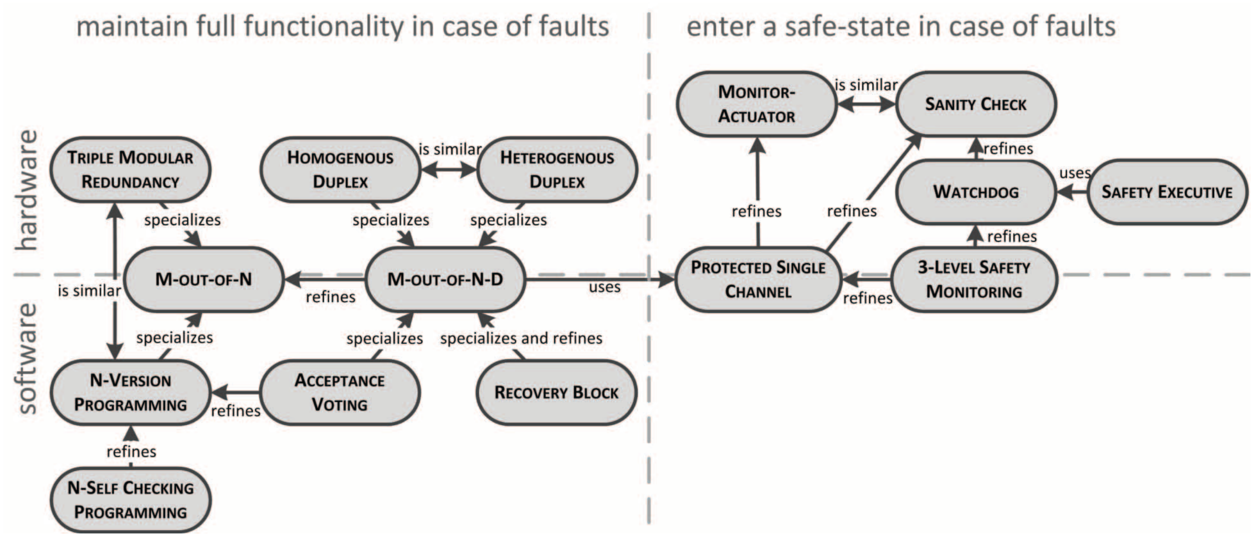


Fig. 4. Safety Architecture Pattern System

To not overload the the pattern-relationship representation, we did not explicitly annotate the *is alternative* relationships, but instead grouped patterns which are alternatives to one another into the group of patterns trying to maintain a safe-state in case of faults and the group of patterns providing full system functionality in case of faults. Additionally, we divided the patterns into software and hardware patterns as already suggested by [Armoush, 2010]. However, the classification of software and hardware patterns is not very strict. Some of the patterns are intended for either software or hardware, but could also be implemented for the other. For example, the WATCHDOG pattern is a hardware pattern, but could also be realized in software by a timer which watches the execution of another program.

The patterns in our safety pattern system are mostly taken from [Armoush, 2010], because these patterns already provide a good collection of other patterns in literature and they focus on rather large-scale architectural design decisions which is the main focus of our pattern system. We included all but one of Armoush's patterns. We excluded one pattern (RECOVERY BLOCK WITH BACKUP VOTING), because we could not find any known uses for it. Additionally to Armoush's patterns we included the M-OUT-OF-N and the M-OUT-OF-N-D pattern, which are based on architectures described in the IEC 61508 safety standard. For each of the patterns from Figure 4, we present the full pattern in Appendix A. Additionally, we provide the tables which show how we related the architectural tactics to the safety patterns as well as the Tactic Topology Models in Appendix C.

6. APPLYING THE TMR PATTERN TO AN EXAMPLE

In this section we show how one of the safety patterns can be applied to an example system. We describe a system found in literature and show which additional benefits could be gained if the architect used our patterns.

The system described in [Alvarez et al., 2005] is a Programmable Logic Device (PLD) safety architecture to be used in control system applications. The basic system (which does not yet fulfill the system safety requirements) consists of a component to handle sensor values (Safe Input), a processing unit which computes output values (CPU), and an interface element for actuators (Safe Output). The basic system is shown in Figure 5.

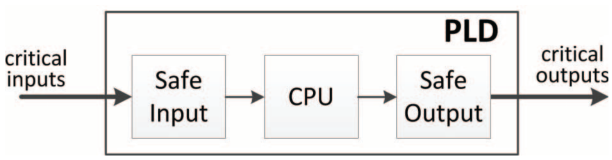


Fig. 5. Basic PLD control system architecture

The proposed safety architecture described in [Alvarez et al., 2005] applies (but not explicitly mentions) the M-OUT-OF-N-D PATTERN (the full pattern is presented in Appendix A - page 20). The architecture uses three identical redundant versions of the basic system architecture and the correct output of these three channels is decided by a majority voter. The three channels are diagnosed with self-tests and if the diagnosis fails, the corresponding channel informs the voter that it does not function properly. The voter then excludes this channel from the vote. The overall safety architecture is shown in Figure 6.

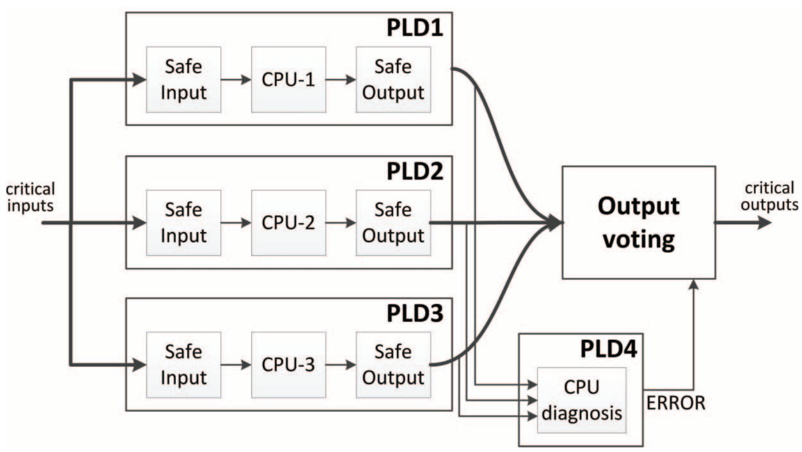


Fig. 6. Safe PLD control system architecture [Alvarez et al., 2005]

In the GSN diagram of the M-OUT-OF-N-D PATTERN, for an architecture several decisions have to be made. For example, we can see that the pattern ether uses the *Replication Redundancy* or the *Diverse Redundancy* tactic. The presented architecture uses *Replication Redundancy* (identical hardware channels), therefore we just consider this redundancy tactic and omit *Diverse Redundancy* from the GSN. This already shows how the GSN diagrams can flexibly be used to describe alternatives for a pattern. Furthermore, the architecture uses *Condition Monitoring* (checks if CPU outputs relate to a reference value) and *Voting* (majority voting). Both of these tactics were also chosen from the set of tactic options presented in the patterns' GSN diagram. Table V lists all the tactics that the architecture uses and presents the IEC 61508 methods which are related to these tactics (taken from

Appendix B). With this table, a safety architect gets a quick overview of methods presented in the safety standard which are relevant for the specific system architecture.

Table V. IEC 61508 methods suitable for the M-OUT-OF-N-D PATTERN

Tactic	IEC 61508 method
Replication Redundancy	A.2.1 Tests by redundant hardware A.2.5 Monitored redundancy A.3.5 Reciprocal comparison by software A.4.5 Block replication A.6.3 Multi-channel output A.6.5 Input comparison/voting A.7.3 Complete hardware redundancy A.7.5 Transmission redundancy
Condition Monitoring	A.1.1 Failure detection by online monitoring A.6.4 Monitored outputs A.9 Temporal and logical program monitoring A.13.1 Monitoring
Voting	A.1.4 Majority voter

From the list of IEC 61508 methods, a safety architect can now choose methods which are appropriate for the specific system. For the the specific design decisions taken in [Alvarez et al., 2005], the IEC 61508 methods that are eligible and are actually used are the following:

—A.1.1 Failure detection by online monitoring

—A.1.4 Majority voter

—A.2.1 Tests by redundant hardware

—A.6.4 Monitored output

—A.13.1 Monitoring

In the GSN diagram of the M-OUT-OF-N-D PATTERN, the tactics can now be replaced with the methods that are actually used in the architecture. Figure 7 shows the resulting GSN diagram which can be used by safety architects to reason about the overall system safety by structurally referring to methods suggested by the safety standard. This gives a structured connection between the goal to maintain the overall system safety down to the actually applied methods. Such a connection can be used during the system certification to argue how the safety goals are achieved by a specific system architecture.

Reasoning about the safety of a system by constructing GSN diagrams based on scenarios was already suggested in [Wu, 2007], where the argument is made that a system which covers all its goals mentioned in relevant scenarios is reasonably safe.

The safety standard describes in detail how to implement the methods which are now present in the GSN. Table VI shows additional information about the applied safety methods taken from the IEC 61508 safety standard. With this information the system architect gets guidance of how to realize the safety methods. Now, the safety architect just has to think about the remaining undeveloped goals (G2, G5, G6 in Figure 7 of the GSN diagram to obtain a complete safety argumentation for the architecture.

We saw, that when applying the suggested safety patterns, additionally to the solution description and the described consequences, a safety architect gets:

—A GSN diagram for the architecture which can be taken as a starting point to develop a structured argument about the system's safety.

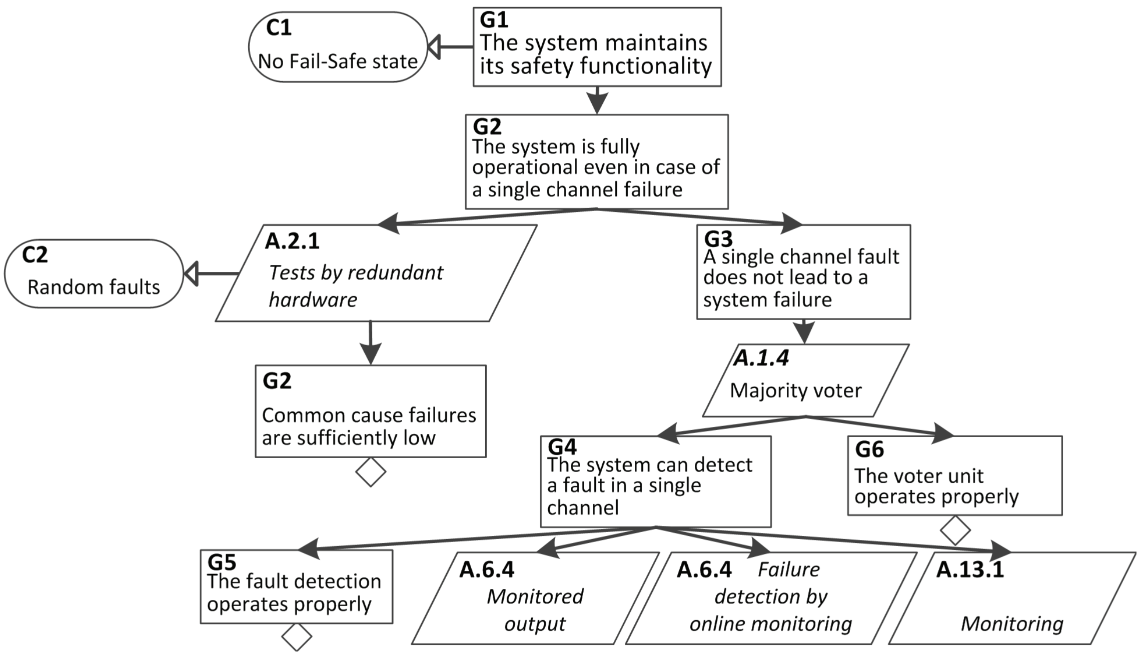


Fig. 7. GSN for the safe PLD control system architecture

Table VI. Safety methods used for the PLD control system architecture (taken form IEC 61508)

Method	Aim	Description
Failure detection by online monitoring	To detect failures by monitoring the behaviour of the E/E/PE safety-related system in response to the normal (on-line) operation of the equipment under control (EUC).	Under certain conditions, failures can be detected using information about (for example) the time behaviour of the EUC. For example, if a switch, which is part of the E/E/PE safety-related system, is normally actuated by the EUC, then if the switch does not change state at the expected time, a failure will have been detected. It is not usually possible to localise the failure.
Majority Voter	To detect and mask failures in one of at least three hardware channels.	A voting unit using the majority principle (2 out of 3, 3 out of 3, or m out of n) is used to detect and mask failures. The voter may itself be externally tested, or it may use selfmonitoring technology.
Tests by redundant hardware	To detect failures using hardware redundancy, i.e. using additional hardware not required to implement the process functions.	Redundant hardware can be used to test at an appropriate frequency the specified safety functions.
Monitored output	To detect individual failures, failures caused by external influences, timing failures, addressing failures, drift failures (for analogue signals) and transient failures	This is a dataflow-dependent comparison of outputs with independent inputs to ensure compliance with a defined tolerance range (time, value). A detected failure cannot always be related to the defective output. This measure is only effective if the dataflow changes during the diagnostic test interval.
Monitoring	To detect the incorrect operation of an actuator.	The operation of the actuator is monitored. The redundancy introduced by this monitoring can be used to trigger emergency action.

—A list of IEC 61508 methods which are related to the overall architecture. A safety architect gets a pool of methods which could be relevant for the chosen architecture. Furthermore, the standard provides additional information about how to implement the methods.
—A connection between the safety goals of the overall architecture and IEC 61508 methods which fulfill these goals. This allows a safety architect to structurally present a safety certification authority how the applied methods which are suggested by the standard are combined to achieve a safe system.

7. CONCLUSION

We presented a system of safety patterns and described their relationships to each other. The patterns include a GSN diagram for safety reasoning.

This pattern system allows safety engineers to easily get an overview of commonly used system architectures and their safety-related consequences. Additionally, when using a pattern, the safety engineer can construct a GSN diagram for his architecture based on the GSN diagrams in the patterns. The GSN representation for the patterns is very suitable, because many of the patterns have alternatives which just differ in changing a single design decision. For example, each of the patterns addressing random faults by using *Replication Redundancy* can easily be used to handle systematic faults as well if *Diverse Redundancy* is used instead. With the GSN representation such alternatives can easily be modeled by simply exchanging a tactic of the pattern (see the M-OUT-OF-N-D PATTERN for example). Similarly, variants of a pattern can be modeled by refining the pattern by an additional tactic. The systematic GSN notation allows to easily integrate additional safety patterns into our pattern system and it allows to reason about safety-specific consequences of these patterns by having a look at the consequences of the added tactic.

We think that the presented system for architectural safety patterns provides safety engineers a good overview of safety architectures and it allows to connect IEC 61508 methods to high level architectures. This is particularly important during safety certification and offers safety engineers a new way to argue about how their architecture achieves safety goals.

ACKNOWLEDGMENTS

We would like to thank our shepherd Brahim Hamid who gave us valuable feedback on this paper. He provided us good improvement suggestion to make this paper easier to read.

REFERENCES

ALVAREZ, Jacobo et al. (2005). Safe PLD-based programmable controllers. In: *International Conference on Field Programmable Logic and Applications*. IEEE, 559–562.

ANTONINO, Pablo Oliveira, Thorsten KEULER, and Pablo ANTONINO (2012). Towards an Approach to Represent Safety Patterns. In: *The Seventh International Conference on Software Engineering Advances (ICSEA)*. c, 228–237.

ARMOUSH, Ashraf (2010). Design patterns for safety-critical embedded systems. PhD thesis. RWTH Aachen University.

BABAR, M.A. (2007). Improving the Reuse of Pattern-Based Knowledge in Software Architecting. In: *EuroPLoP*. Lero, Ireland, 7–11.

BUSCHMANN, Frank et al. (1996). *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons.

DANIELS, Fonda, Kalhee KIM, and Mladen A VOUK (1997). The Reliable Hybrid Pattern A Generalized Software Fault Tolerant Design Pattern. In: *European Conference on Pattern Language of Programs (EuroPLoP)*, 1–9.

DESNOS, Nicolas et al. (2012). Towards a Security and Dependability Pattern Development Technique for Resource Constrained Embedded Systems. In: *4th International Conference on Software Quality, Process Automation in Software Development*. Springer, Vienna, Austria, 193–204.

DOUGLASS, Bruce Powel (1998). Safety-Critical Systems Design. *Electronic Engineering* 70, 862.

DOUGLASS, Bruce Powel (2002). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Pearson.

DOUGLASS, Bruce Powel (2010). *Design Patterns for Embedded Systems in C*. Elsevier.

GAWAND, Hemangi, RS MUNDADA, and P. SWAMINATHAN (2011). Design Patterns to Implement Safety and Fault Tolerance. *International Journal of Computer Applications* 18, 2, 6–13.

GRUNSKE, Lars (2003). Transformational Patterns for the Improvement of Safety Properties in Architectural Specification. In: *Proceedings of The Second Nordic Conference on Pattern Languages of Programs (VikingPLoP)*.

HAMPTON, Paul (2012). Survey of safety Architectural Patterns. In: *Achieving Systems Safety*. February 2012. Springer London, London, 7–9.

HANMER, Robert S. (2007). *Patterns for Fault Tolerant Software*. Wiley.

KUMAR, Kiran and T.V. PRABHAKAR (2010a). Design Decision Topology Model for Pattern Relationship Analysis. In: *1st Asian Conference on Pattern Languages of Programs (AsianPLoP 2010)*.

KUMAR, Kiran and T.V. PRABHAKAR (2010b). Pattern-oriented Knowledge Model for Architecture Design. In: *17th Conference on Pattern Languages of Programs (PLoP)*.

OLIVERA, Andre Rodrigues (2012). Taim : A Safety Pattern Repository, BsC thesis. Federal University of Rio Grande do sul.

PULLUM, L. (2001). *Software fault tolerance techniques and implementation*. Artech House.

RAUHAMÄKI, Jari and Seppo KUIKKA (2013). Patterns for control system safety. In: *18th European Conference on Pattern Languages of Programs (VikingPLoP)*.

RAUHAMÄKI, Jari, Timo VEPSÄLÄINEN, and Seppo KUIKKA (2012). Architectural patterns for functional safety. In: *Nordic Conference on Pattern Languages of Programs (VikingPLoP)*.

RAUHAMÄKI, Jari, Timo VEPSÄLÄINEN, and Seppo KUIKKA (2013). Patterns for safety and control system cooperation. In: *Nordic Conference on Pattern Languages of Programs (VikingPLoP)*.

SARIDAKIS, Titos (2002). A System of Patterns for Fault Tolerance. In: *EuroPLoP*.

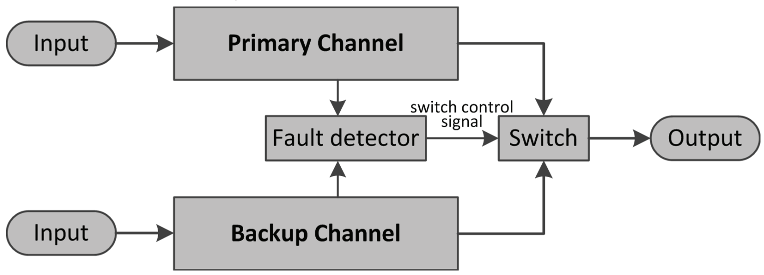
SARMA, U V R, Sahith RAMPELLI, and P PREMCHAND (2013). A Catalog of Architectural Design Patterns for Safety-Critical Real-Time Systems. *International Journal of Engineering Research and Applications* 3, 1, 125–131.

SCHUMACHER, Markus (2003). *Security Engineering with Patterns*. Springer.

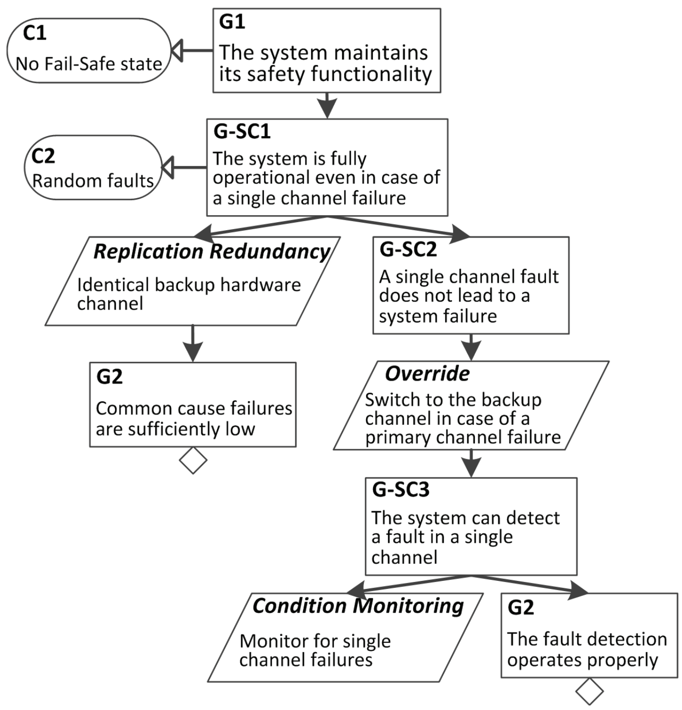
WU, Weihang (2007). Architectural Reasoning for Safety- Critical Software Applications. PhD thesis. University of York.

A. SAFETY ARCHITECTURE PATTERNS

<table>
<tr><td>Pattern Name</td><td>HOMOGENOUS DUPLEX PATTERN</td><td>Pattern Type</td><td>hardware, fail-over</td></tr>
<tr><td>Also Known As</td><td colspan="3">Homogeneous Redundancy Pattern, Standby-Spare Pattern, Dynamic Redundancy Pattern, Two-Channel Redundancy Pattern, 1oo2D Pattern</td></tr>
<tr><td>Context</td><td colspan="3">A safety-critical application without a fail-safe state has a high random error rate and a low systematic error rate.</td></tr>
<tr><td>Problem</td><td colspan="3">How to design a system which continues operating even in the presence of a fault in one of the system components</td></tr>
<tr><td>Forces</td><td colspan="3">- the system cannot shut down because it has no safe state
- development costs should not increase
- the safety standard requires high fault coverage for single-point of failure components
- high availability requires hardware platforms to be maintained at the runtime</td></tr>
<tr><td>Solution</td><td colspan="3">The system consists of a Primary Channel (active) and a Secondary Channel (backup) which are two identical hardware modules. A Fault detector monitors the channels and controls a Switch to select the Backup Channel in case of a Primary Channel failure.

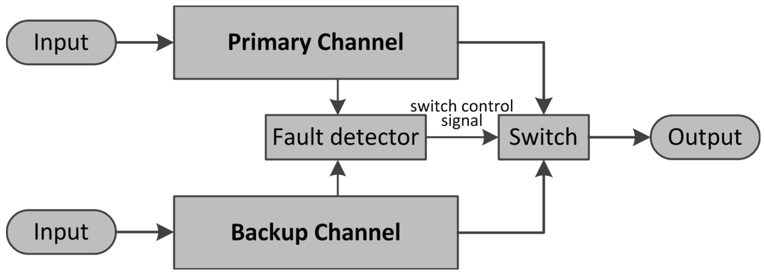
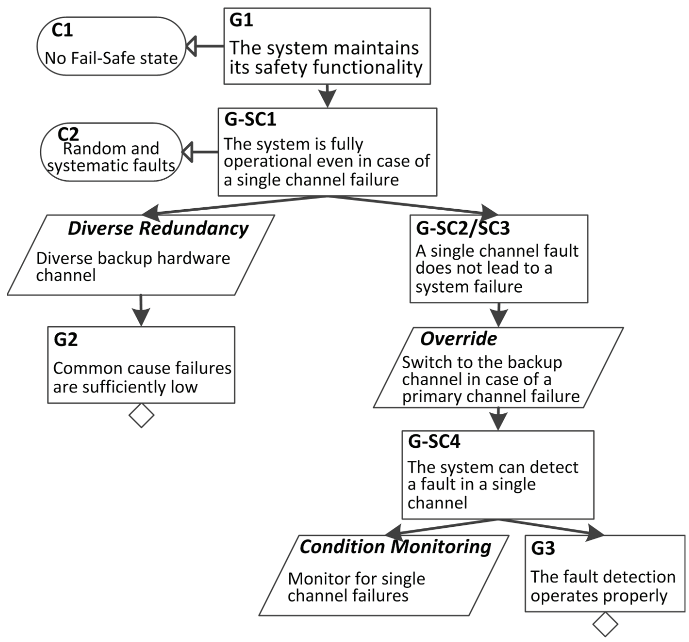


</td></tr>
<tr><td>GSN Diagram</td><td colspan="3">Used Tactics: Replication Redundancy, Override, Condition Monitoring

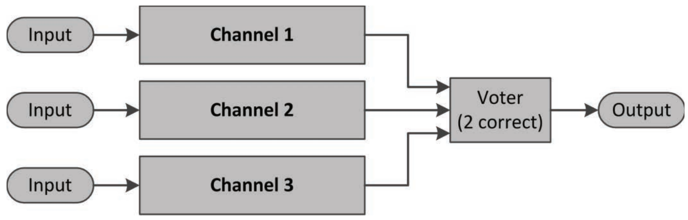
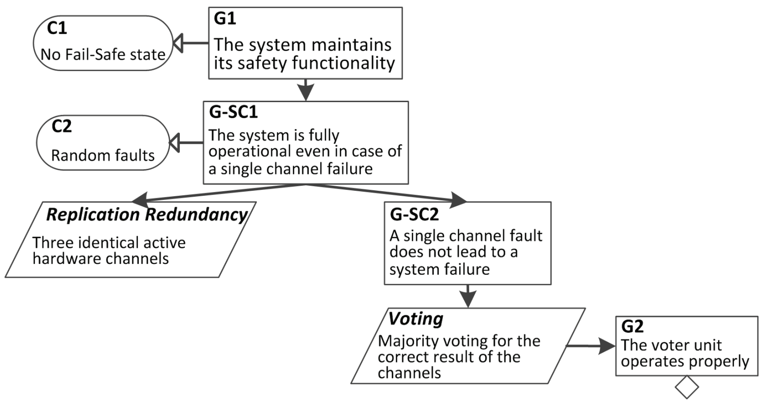


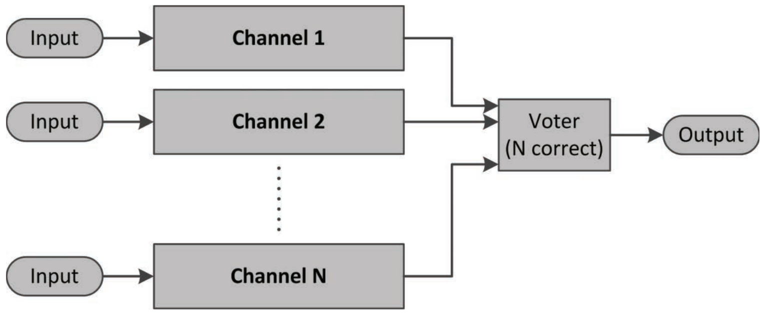
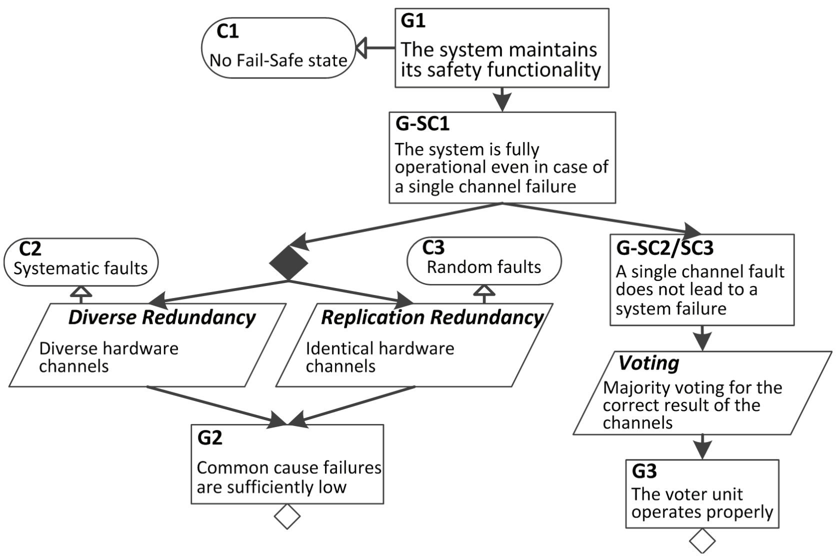
</td></tr>
</table>

Consequences		Systematic and random faults in a single channel are detected and masked. System reliability strongly depends on the fault coverage of the fault detection unit and on the proper functionality of the switch.	
		Affected Attributes	
		Positively	**Negatively**
		Safety: Random Errors in a single channel are handled *Availability:* The full system functionality is still available in case of a single random fault *Maintenance:* Hardware channels can be maintained at runtime	Double hardware costs for system replication
General Scenarios	**SC1**	The system is fully operational even in case of a single channel failure.	
	SC2	A single channel random fault does not lead to a system failure.	
	SC3	The system can detect a fault in a single channel.	
Known Uses	- TOYOPUC-PCS PLC [Miyawaki, 2008] - Navigation system safety [Ljosland, 2006] - Gebhardt GA DUPLEX-S 1oo2D PLC - http://www.gebhardt-automation.com		
Credits	[Douglass, 2002] introduces the pattern. [Grunske, 2003] presents a more general version of this pattern and [Armoush, 2010] adds detailed information about quality attribute related consequences.		

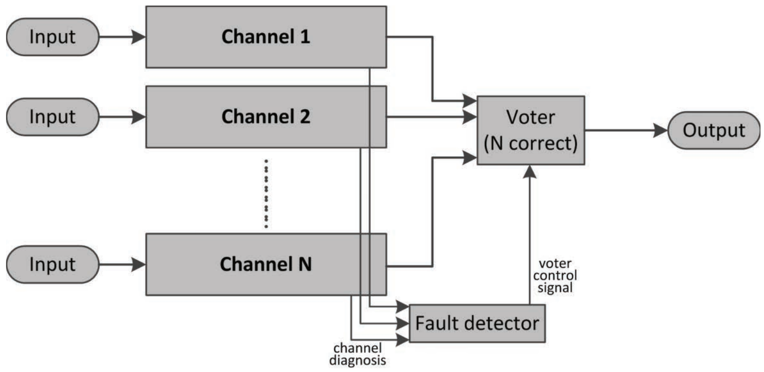
<table>
<tr><td>Pattern Name</td><td>HETEROGENOUS DUPLEX PATTERN</td><td>Pattern Type</td><td>hardware, fail-over</td></tr>
<tr><td>Also Known As</td><td colspan="3">Heterogenous Redundancy Pattern, Diverse Redundancy Pattern, 1oo2D Pattern</td></tr>
<tr><td>Context</td><td colspan="3">A safety-critical application without a fail-safe state has a high random and systematic error rate.</td></tr>
<tr><td>Problem</td><td colspan="3">How to design a system which continues operating even in the presence of a fault in one of the system components</td></tr>
<tr><td>Forces</td><td colspan="3">- the system cannot shut down because it has no safe state
- high safety certification levels require handling of systematic faults
- the safety standard requires high fault coverage for single-point of failure components
- high availability requires hardware platforms to be maintained at the runtime</td></tr>
<tr><td>Solution</td><td colspan="3">The system consists of a Primary Channel (active) and a Secondary Channel (backup) which are two diverse hardware modules. A Fault detector monitors the channels and controls a Switch to select the Backup Channel in case of a Primary Channel failure.

</td></tr>
<tr><td>GSN Diagram</td><td colspan="3">Used Tactics: Diverse Redundancy, Override, Condition Monitoring

</td></tr>
</table>

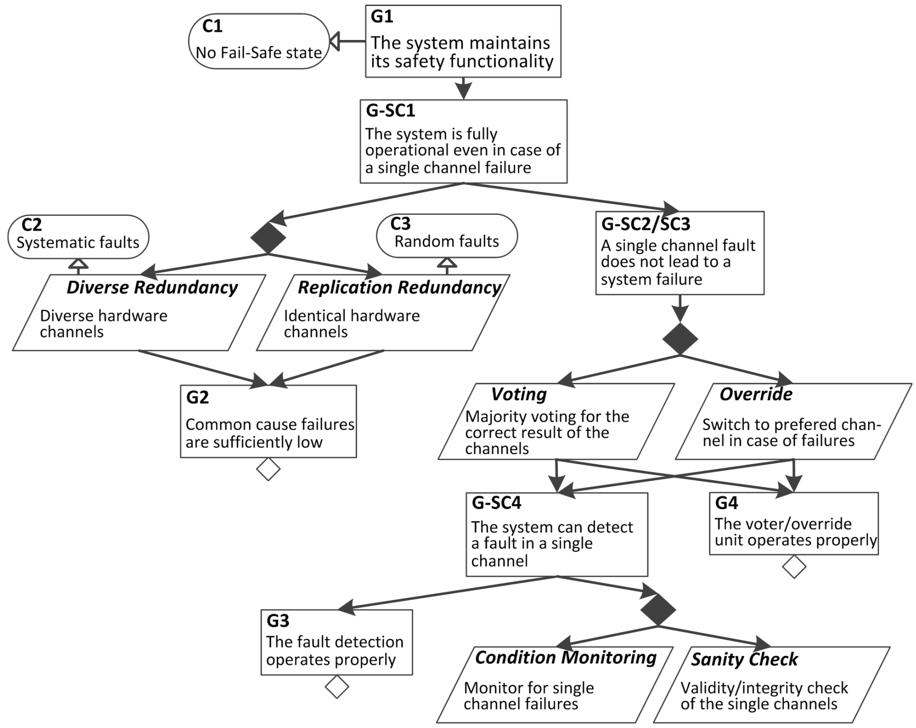
Consequences		Systematic and random faults in a single channel are detected and masked. System reliability strongly depends on the fault coverage of the fault detection unit, on the proper functionality of the switch, and on the level of diversity between the two channels	
		Affected Attributes	
		Positively	**Negatively**
		Safety: Random and systematic faults in a single channel are detected and handled *Availability:* The full system functionality is still available in case of a single random or systematic fault *Maintenance:* Hardware channels can be maintained at runtime	Double hardware costs for system replication, Double development costs due to diverse channels, Modifying the functionality of a channel requires double effort
General Scenarios	**SC1**	The system is fully operational even in case of a single channel failure.	
	SC2	A single channel random fault does not lead to a system failure.	
	SC3	A single channel systematic fault does not lead to a system failure.	
	SC4	The system can detect a fault in a single channel.	
Known Uses	- Turbine control system [Kohanawa et al., 2010] - Motor control software [Mutlu, 2004] - YOKOGAWA ProSafe PLCs - http://www.yokogawa.com		
Credits	[Douglass, 2002] introduces the pattern. [Grunske, 2003] presents a more general version of this pattern and [Armoush, 2010] adds detailed information about quality attribute related consequences.		

<table>
<tr><td>Pattern Name</td><td colspan="2">TRIPLE MODULAR REDUNDANCY PATTERN</td><td>Pattern Type</td><td>hardware, fail-over</td></tr>
<tr><td>Also Known As</td><td colspan="4">2oo3 Pattern, Homogeneous Triplex Pattern</td></tr>
<tr><td>Context</td><td colspan="4">A safety-critical application without a fail-safe state, a high random error and a low systematic error rate.</td></tr>
<tr><td>Problem</td><td colspan="4">How to design a system which continues operating even in the presence of a fault in one of the system components.</td></tr>
<tr><td>Forces</td><td colspan="4">- the system cannot shut down because it has no safe state
- the safety standard requires high fault coverage for single-point of failure components
- high availability requires hardware platforms to be maintained at the runtime</td></tr>
<tr><td>Solution</td><td colspan="4">Three identical hardware channels operate in parallel. If a single fault occurs in one channel then the other two channels still produce the correct output. A majority voter decides for the correct result.

</td></tr>
<tr><td>GSN Diagram</td><td colspan="4">Used Tactics: Replication Redundancy, Voting

</td></tr>
<tr><td rowspan="4">Consequences</td><td colspan="4">This pattern does not identify the type or the reason of the fault; it just determines the module that contains a fault without correcting the fault itself. The voter has to be very reliable.</td></tr>
<tr><td colspan="4">Affected Attributes</td></tr>
<tr><td colspan="2">Positively</td><td colspan="2">Negatively</td></tr>
<tr><td colspan="2">Safety: Random faults in a single channel are masked
Availability: The full system functionality is still available in case of a single random fault
Maintenance: Hardware channels can be maintained at runtime</td><td colspan="2">Triple hardware costs for system replication</td></tr>
<tr><td rowspan="2">General Scenarios</td><td>SC1</td><td colspan="3">The system is fully operational even in case of a single channel failure.</td></tr>
<tr><td>SC2</td><td colspan="3">A single channel random fault does not lead to a system failure.</td></tr>
<tr><td>Known Uses</td><td colspan="4">- Turbine control sensor input [Kohanawa et al., 2010]
- SRAM applying TMR [Kyriakoulakos and Pnevmatikatos, 2009]
- TMS-1000R Gas Turbine - http://www.turbinetech.com</td></tr>
<tr><td>Credits</td><td colspan="4">[Douglass, 2002] formulates this well-known architecture as a pattern. [Armoush, 2010] adds detailed information about quality attribute related consequences.</td></tr>
</table>

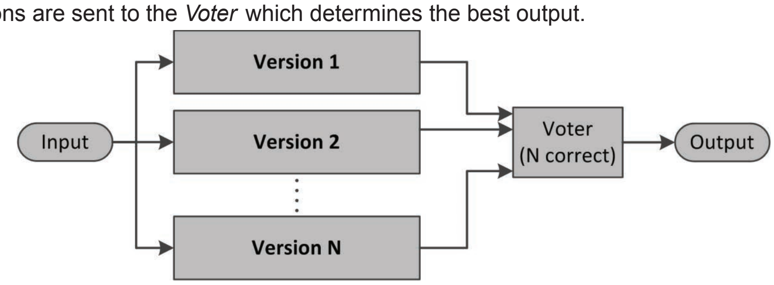
Pattern Name	M-OUT-OF-N PATTERN	**Pattern Type**	hardware/software, fail-over
Also Known As	M/N Parallel Redundancy Pattern, MooN Pattern		
Context	A safety-critical application without a fail-safe state has a high random error rate and a low or high systematic error rate.		
Problem	How to design a system which continues operating even in the presence of a fault in one of the system components.		
Forces	- the system cannot shut down because it has no safe state - high safety certification levels require handling of systematic faults - the safety standard requires high fault coverage for single-point of failure components - high availability requires hardware platforms to be maintained at the runtime		
Solution	N identical or diverse channels (software or hardware) operate in parallel. If a fault occurs in one channel then the other channels still produce the correct output. A voter decides for the result given by at least M channels.  		
GSN Diagram	*Used Tactics:* Replication Redundancy / Diverse Redundancy, Voting  		

Consequences		This pattern does not identify the type or the reason of the fault; it just determines the module that contains a fault without correcting the fault itself. To achieve high reliability, the voter has to be very reliable.	
		Affected Attributes	
		Positively	**Negatively**
		Safety: Single-channel random or systematic faults are masked *Availability:* The full system functionality is still available in case of a single fault *Maintenance:* Hardware channels can be maintained at runtime	Multiple hardware costs for system replication and multiple development costs if diverse channels are used
General Scenarios	**SC1**	The system is fully operational even in case of a single channel failure.	
	SC2	A single channel random fault does not lead to a system failure.	
	SC3	A single channel systematic fault does not lead to a system failure.	
Known Uses	- 1oo2 Architecture for LHC detectors [Fernandez and Denz, 2002] - Steering system controller [Börcsök et al., 2011] - Netherlocks safety lock - http://halmapr.com/news/netherlocks/tag/3oo4/		
Credits	[Grunske, 2003] describes this pattern and calls it Multi-Channel-Redundancy with Voting. [Armoush, 2010] adds detailed information about quality attribute related consequences.		

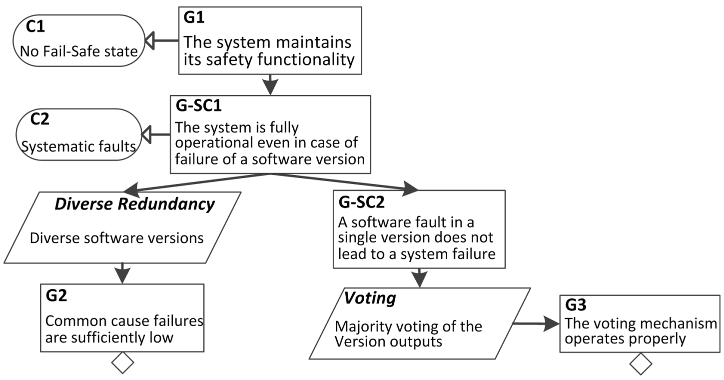
Pattern Name	M-OUT-OF-N-D PATTERN	**Pattern Type**	hardware/software, fail-over
Also Known As	MooN-D Pattern		
Context	A safety-critical application without a fail-safe state has a high random error rate and a low or high systematic error rate.		
Problem	How to design a system which continues operating even in the presence of a fault in one of the system components.		
Forces	- the system cannot shut down because it has no safe state - high safety certification levels require handling of systematic faults - the safety standard requires high fault coverage for single-point of failure components - due to these high availability requirements the hardware platforms must be maintained at the runtime of the system		
Solution	N identical or diverse channels operate in parallel. If a single fault occurs in one channel then the other channels still produce the correct output. A *Voter* decides for the result given by at least M channels. The *Voter* can be influenced by a diagnostic check implemented within the channels. For example, a channel could be excluded from the vote if its diagnostic check fails. 		

<table>
<tr><td colspan="2">GSN Diagram</td><td colspan="2">Used Tactics: Replication Redundancy / Diverse Redundancy, Voting / Override, Sanity Check / Condition Monitoring

</td></tr>
<tr><td colspan="2" rowspan="4">Consequences</td><td colspan="2">This pattern can identify the type or the reason of a fault. System reliability strongly depends on the voter unit, the diagnostic test, and the level of diversity between the two channels.</td></tr>
<tr><td colspan="2">Affected Attributes</td></tr>
<tr><td>Positively</td><td>Negatively</td></tr>
<tr><td>Safety: Random or systematic faults in a single channel are masked
Availability: The full system functionality is still available in case of a single fault
Maintenance: Hardware channels can be maintained at runtime</td><td>Multiple hardware costs for system replication and multiple development costs if diverse channels are used</td></tr>
<tr><td rowspan="4">General Scenarios</td><td>SC1</td><td colspan="2">The system is fully operational even in case of a single channel failure.</td></tr>
<tr><td>SC2</td><td colspan="2">A single channel random fault does not lead to a system failure.</td></tr>
<tr><td>SC3</td><td colspan="2">A single channel systematic fault does not lead to a system failure.</td></tr>
<tr><td>SC4</td><td colspan="2">The system can detect a fault in a single channel.</td></tr>
<tr><td>Known Uses</td><td colspan="3">- HIMA HiQuad 2oo4D architecture [Skambraks, 2006]
- 2oo3D architecture for PLC [Alvarez et al., 2005]
- RTP 3000 process heater controller - http://rtpcorp.com/documents/ProcessHeaters3000.pdf</td></tr>
<tr><td>Credits</td><td colspan="3">The MooN-D architecture is described by the IEC 61508 standard.</td></tr>
</table>

<table>
<tr><th>Pattern Name</th><td colspan="2">N-VERSION PROGRAMMING PATTERN</td><th>Pattern Type</th><td>software, fail-over</td></tr>
<tr><th>Also Known As</th><td colspan="4">-</td></tr>
<tr><th>Context</th><td colspan="4">A safety-critical software without a fail-safe state which probably contains software faults.</td></tr>
<tr><th>Problem</th><td colspan="4">How to design a system which continues operating even in the presence of software faults.</td></tr>
<tr><th>Forces</th><td colspan="4">- software often contains faults
- high safety certification levels require handling of systematic faults
- the safety standard requires high fault coverage for single-point of failure components</td></tr>
<tr><th>Solution</th><td colspan="4">N software versions are developed independently from the same initial specification. The outputs of these versions are sent to the Voter which determines the best output.

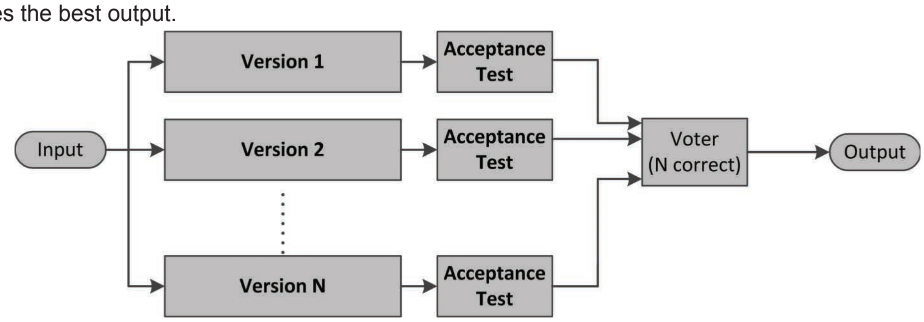


</td></tr>
<tr><th>GSN Diagram</th><td colspan="4">Used Tactics: Diverse Redundancy, Voting

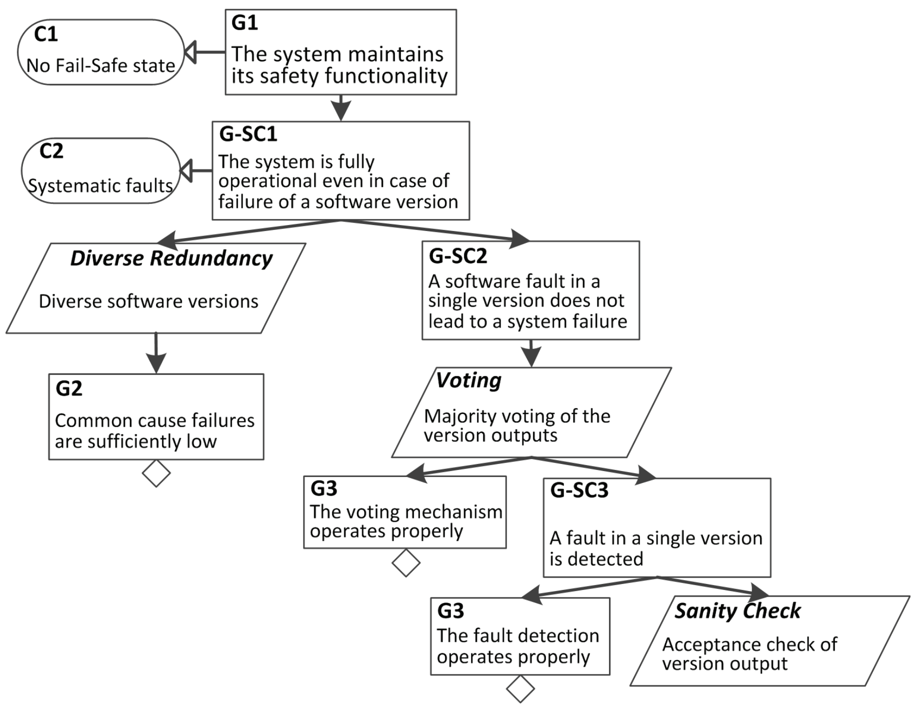


</td></tr>
<tr><th rowspan="4">Consequences</th><td colspan="4">This pattern can handle systematic faults in the software. A drawback is that the high dependency on the initial specification may lead to a propagation of dependent faults to all versions. The voter has to be highly reliable.</td></tr>
<tr><th colspan="4">Affected Attributes</th></tr>
<tr><th colspan="2">Positively</th><th colspan="2">Negatively</th></tr>
<tr><td colspan="2">Safety: Software faults are handled but not detected
Availability: The full system functionality is still available in case of faults</td><td colspan="2">Costs: Multiple development costs, multiple hardware costs if the software versions run on separate hardware
Modifications: Multiple effort for software modifications</td></tr>
<tr><th rowspan="2">General Scenarios</th><th>SC1</th><td colspan="3">The system is fully operational even in case of a failure of a software version.</td></tr>
<tr><th>SC2</th><td colspan="3">A software fault in a single version does not lead to a system failure.</td></tr>
<tr><th>Known Uses</th><td colspan="4">- Analysis of N-version programming [Brilliant et al., 1990]
- Train Control [Carr et al., 2005]
- Railway Interlocking System [Durmuù et al., 2011]</td></tr>
<tr><th>Credits</th><td colspan="4">[Armoush, 2010] presents this pattern with detailed information about quality attribute related consequences.</td></tr>
</table>

<table>
<tr><td>Pattern Name</td><td>Acceptance Voting Pattern</td><td>Pattern Type</td><td>software, fail-over</td></tr>
<tr><td>Also Known As</td><td colspan="3">-</td></tr>
<tr><td>Context</td><td colspan="3">A safety-critical software without a fail-safe state which probably contains software faults.</td></tr>
<tr><td>Problem</td><td colspan="3">How to design a system which continues operating even in the presence of software faults.</td></tr>
<tr><td>Forces</td><td colspan="3">- software often contains faults
- high safety certification levels require handling of systematic faults
- the safety standard requires high fault coverage for single-point of failure components</td></tr>
<tr><td>Solution</td><td colspan="3">N software versions are developed independently from the same initial specification. The outputs of these versions are checked by an Acceptance Test and valid outputs are sent to a Voter which determines the best output.



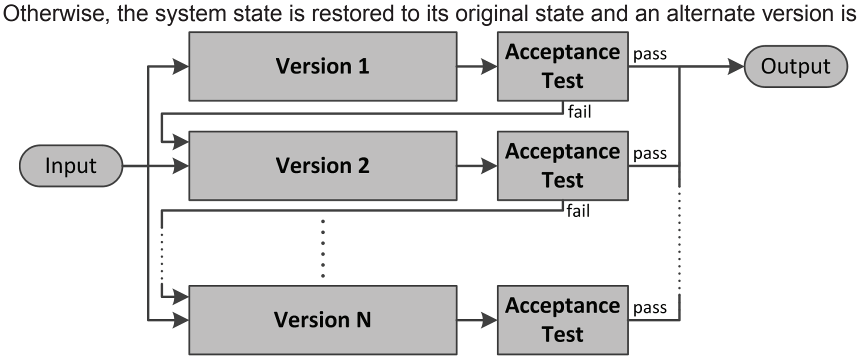
</td></tr>
<tr><td>GSN Diagram</td><td colspan="3">Used Tactics: Diverse Redundancy, Voting, Sanity Check



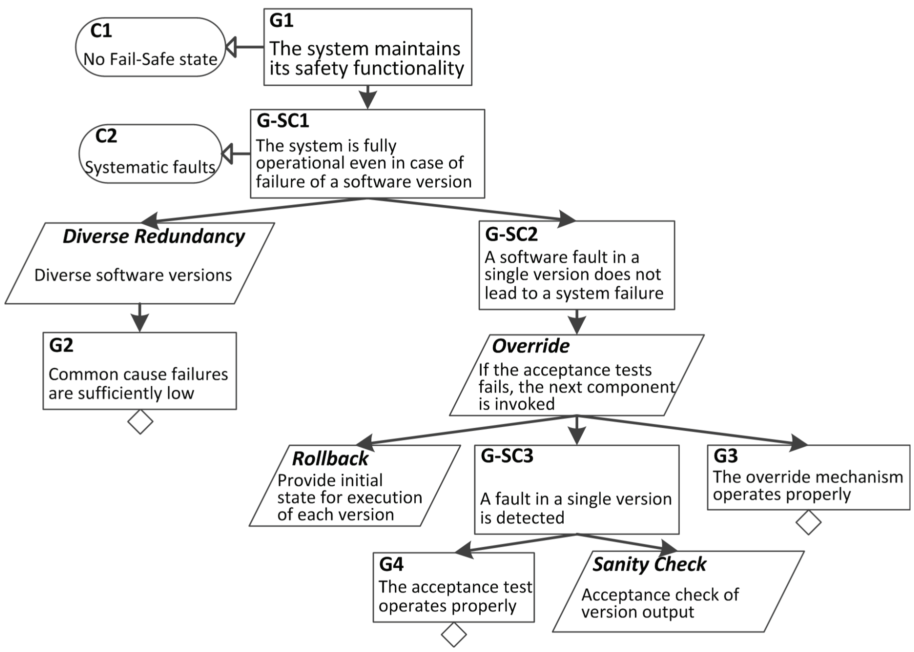
</td></tr>
</table>

<table>
<tr><td rowspan="4" colspan="2">Consequences</td><td colspan="2">This pattern can handle systematic faults in the software. A drawback is that the high dependency on the initial specification may lead to a propagation of dependent faults to all versions.</td></tr>
<tr><td colspan="2">Affected Attributes</td></tr>
<tr><td>Positively</td><td>Negatively</td></tr>
<tr><td>Safety: Software faults are handled and probably detected
Availability: The full system functionality is still available in case of faults</td><td>Costs: Multiple development costs, multiple hardware costs if the software versions run on separate hardware
Modifications: Multiple effort for software modifications</td></tr>
<tr><td rowspan="3">General Scenarios</td><td>SC1</td><td colspan="2">The system is fully operational even in case of a failure of a software version.</td></tr>
<tr><td>SC2</td><td colspan="2">A software fault in a single version does not lead to a system failure.</td></tr>
<tr><td>SC3</td><td colspan="2">A fault in a single software version is detected.</td></tr>
<tr><td>Known Uses</td><td colspan="3">- Dependable web services [Nourani and Azgomi, 2009]
- Protected C++ Dispatcher [Borchert et al., 2012]
- Fault-tolerant middleware [Kim, 1998]</td></tr>
<tr><td>Credits</td><td colspan="3">[Armoush, 2010] presents this pattern with detailed information about quality attribute related consequences.</td></tr>
</table>

<table>
<tr><td>Pattern Name</td><td>RECOVERY BLOCK PATTERN</td><td>Pattern Type</td><td>software, fail-over</td></tr>
<tr><td>Also Known As</td><td colspan="3">-</td></tr>
<tr><td>Context</td><td colspan="3">A safety-critical software without a fail-safe state which probably contains software faults.</td></tr>
<tr><td>Problem</td><td colspan="3">How to design a system which continues operating even in the presence of software faults.</td></tr>
<tr><td>Forces</td><td colspan="3">- software often contains faults
- high safety certification levels require handling of systematic faults
- the safety standard requires high fault coverage for single-point of failure components
- no additional processing hardware or processing time is available</td></tr>
<tr><td>Solution</td><td colspan="3">N software versions are developed independently from the same initial specification. Only a single version is executed at a time. After the execution of Version 1, an Acceptance Test is executed to check if the software output is reasonable. If the Acceptance Test is passed, then the outcome is considered as correct. Otherwise, the system state is restored to its original state and an alternate version is invoked.

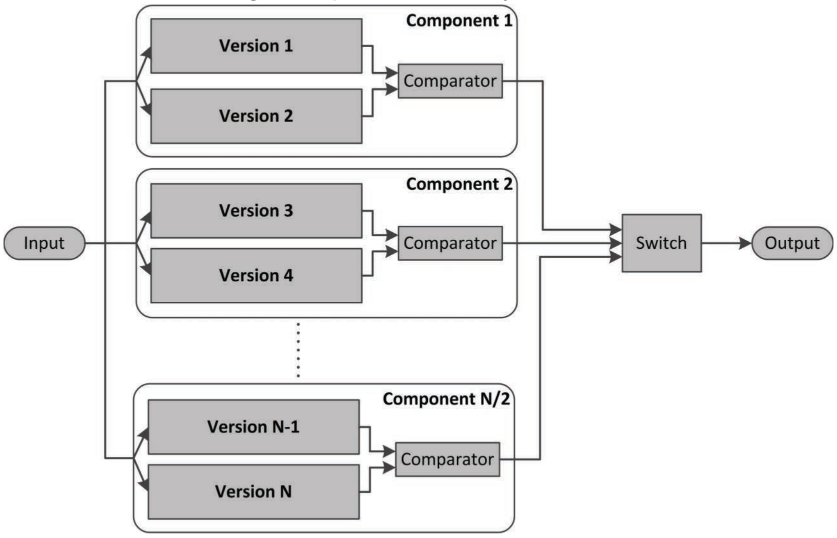


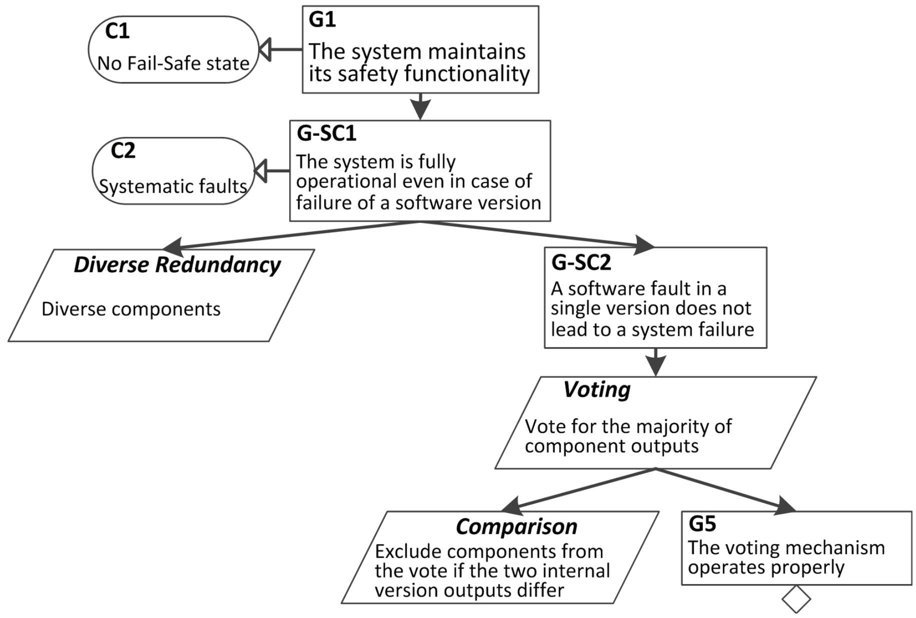
</td></tr>
<tr><td>GSN Diagram</td><td colspan="3">Used Tactics: Diverse Redundancy, Override, Rollback, Sanity Check

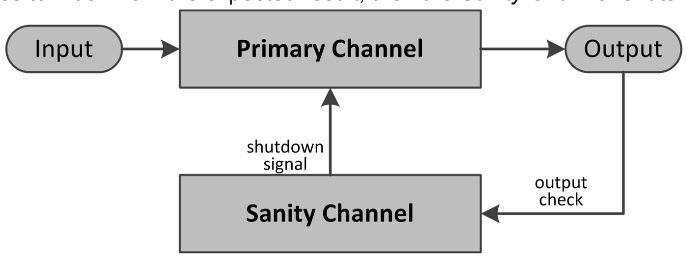
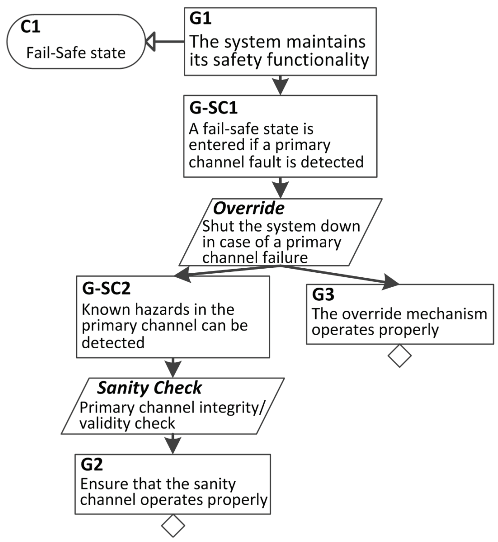


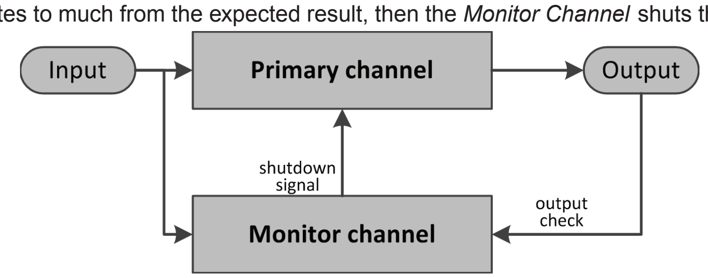
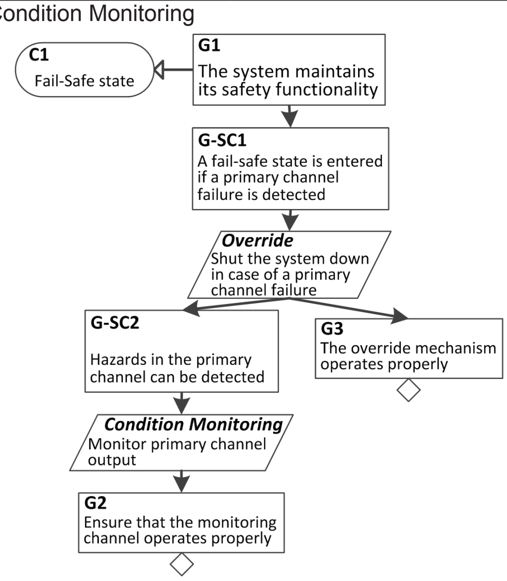
</td></tr>
</table>

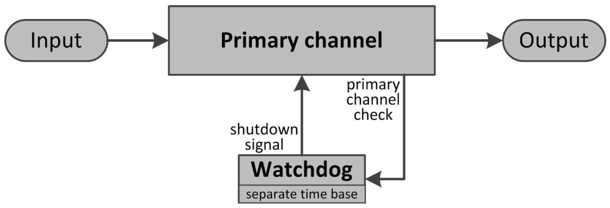
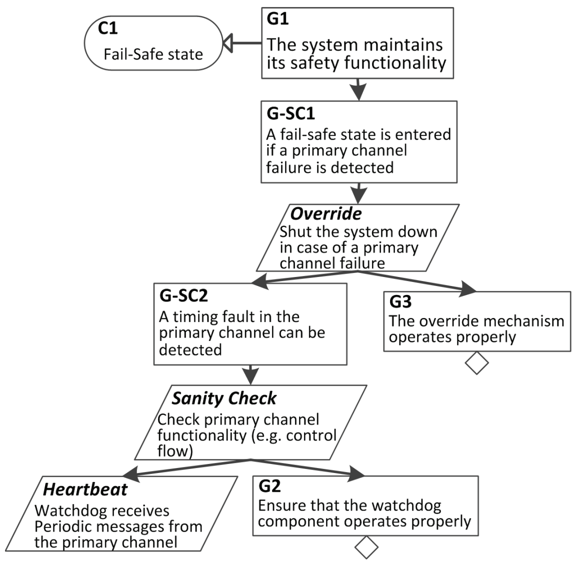
Consequences		This pattern can handle systematic faults in the software. A drawback is that the high dependency on the initial specification may lead to a propagation of dependent faults to all versions. Also the reliability highly depends on the quality of the acceptance test.	
		Affected Attributes	
		Positively	**Negatively**
		Safety: Software faults are handled and probably detected *Availability:* The full system functionality is still available in case of faults	*Costs:* Multiple development costs *Modifications:* Multiple effort for software modifications
General Scenarios	**SC1**	The system is fully operational even in case of a failure of a software version.	
	SC2	A software fault in a single version does not lead to a system failure.	
	SC3	A fault in a single software version is detected.	
Known Uses	- Mission-Critical Intrusion-Tolerant Architecture [Wang et al., 2001] - Fault-Tolerant WSN Framework [Beder et al., 2011] - Evaluation of the Recovery Block pattern in software projects [Anderson et al., 1985]		
Credits	[Armoush, 2010] presents this pattern with detailed information about quality attribute related consequences.		

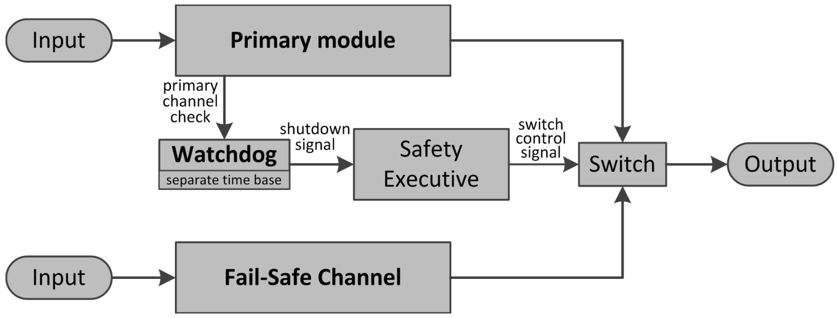
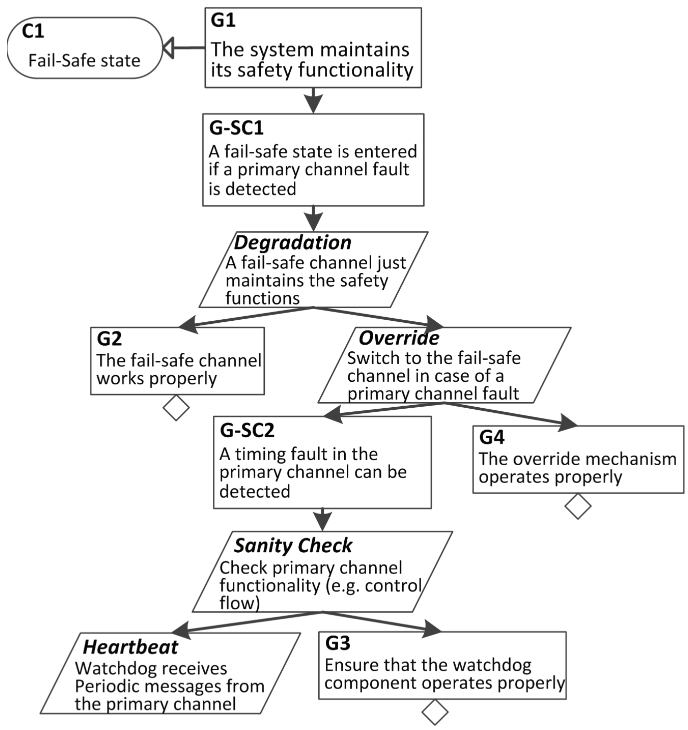
Pattern Name	N-Self Checking Programming Pattern	Pattern Type	software, fail-over
Also Known As	-		
Context	A safety-critical software without a fail-safe state which probably contains software faults.		
Problem	How to design a system which continues operating even in the presence of software faults.		
Forces	- software often contains faults - high safety certification levels require handling of systematic faults - the safety standard requires high fault coverage for single-point of failure components		
Solution	N>=4 software versions are developed independently from the same initial specification. The versions are arranged in pairs of two as components. Within a component, the results of the two versions are compared to detect errors. If a component fails due to different results from its versions, the next component is invoked to start delivering the required functionality.  		

GSN Diagram		*Used Tactics:* Diverse Redundancy, Voting, Comparison  	
Consequences		This pattern can handle systematic faults in the software. A drawback is that the high dependency on the initial specification may lead to a propagation of dependent faults to all versions.	
		Affected Attributes	
		Positively	**Negatively**
		Safety: Software faults are handled and probably detected at component level *Availability:* The full system functionality is still available in case of faults	*Costs:* Multiple development costs *Modifications:* Multiple effort for software modifications
General Scenarios	**SC1**	The system is fully operational even in case of a failure of a software version.	
	SC2	A software fault in a single version does not lead to a system failure.	
Known Uses		- High Available Web services using N-self checking programming [Parchas and Lemos, 2004] - Airbus uses N-self checking programming [Sghairi et al., 2008] - N-self Checking Programming in the avionics domain [Laprie et al., 1995]	
Credits		[Armoush, 2010] presents this pattern with detailed information about quality attribute related consequences.	

<table>
<tr><td>Pattern Name</td><td colspan="2">SANITY CHECK PATTERN</td><td>Pattern Type</td><td>hardware, fail-safe</td></tr>
<tr><td>Also Known As</td><td colspan="4">-</td></tr>
<tr><td>Context</td><td colspan="4">A safety-critical system with a fail-safe state and low availability requirements.</td></tr>
<tr><td>Problem</td><td colspan="4">Find an appropriate mechanism to detect failures or errors that can lead to known hazards.</td></tr>
<tr><td>Forces</td><td colspan="4">- The set of relevant hazards is often known for a specific application domain
- Full redundancy solutions are expensive</td></tr>
<tr><td>Solution</td><td colspan="4">A separate Sanity Channel monitors the correct operation of the Primary Channel. If the Primary Channel output deviates to much from the expected result, then the Sanity Channel shuts the system down.

</td></tr>
<tr><td>GSN Diagram</td><td colspan="4">Used Tactics: Override, Sanity Check

</td></tr>
<tr><td rowspan="4">Consequences</td><td colspan="4">The sanity channel is diverse from the primary channel which allows limited systematic fault as well as random fault detection.</td></tr>
<tr><td colspan="4">Affected Attributes</td></tr>
<tr><td colspan="2">Positively</td><td colspan="2">Negatively</td></tr>
<tr><td colspan="2">Safety: Known hazards can be handled</td><td colspan="2">Availability: Decreased if the system goes into its safe state</td></tr>
<tr><td rowspan="2">General Scenarios</td><td>SC1</td><td colspan="3">A fail-safe state is entered if a primary channel fault is detected.</td></tr>
<tr><td>SC2</td><td colspan="3">Known hazards in the primary channel can be detected.</td></tr>
<tr><td>Known Uses</td><td colspan="4">- Oxygen level software Sanity Channel - PISCAS fish farm automation system [Preschern, 2011]
- Automotive Distance Sensor [Zimmer, 2009]
- Sanity Check in Semiconductor Devices [Tong, 2007]</td></tr>
<tr><td>Credits</td><td colspan="4">[Douglass, 2002] introduces the pattern. [Grunske, 2003] presents a more general version of this pattern and [Armoush, 2010] adds detailed information about quality attribute related consequences.</td></tr>
</table>

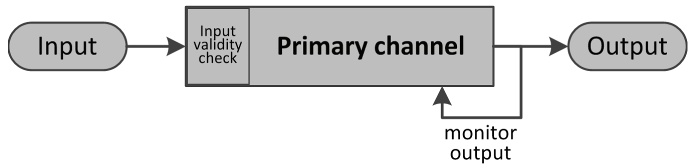
<table>
<tr><td colspan="2">Pattern Name</td><td>MONITOR-ACTUATOR PATTERN</td><td>Pattern Type</td><td>hardware, fail-safe</td></tr>
<tr><td colspan="2">Also Known As</td><td colspan="3">-</td></tr>
<tr><td colspan="2">Context</td><td colspan="3">A safety-critical system with a fail-safe state and with low availability requirements.</td></tr>
<tr><td colspan="2">Problem</td><td colspan="3">Find an appropriate mechanism to detect failures or errors</td></tr>
<tr><td colspan="2">Forces</td><td colspan="3">- Full redundancy solutions are expensive</td></tr>
<tr><td colspan="2">Solution</td><td colspan="3">A separate Monitor Channel monitors the correct operation of the primary channel. The Monitor Channel computes reference values from the inputs and compares them to the Primary Channel output. If the value deviates to much from the expected result, then the Monitor Channel shuts the system down.

</td></tr>
<tr><td colspan="2">GSN Diagram</td><td colspan="3">Used Tactics: Override, Condition Monitoring

</td></tr>
<tr><td colspan="2" rowspan="4">Consequences</td><td colspan="3">The monitor channel is diverse from the primary channel which allows limited systematic fault as well as random fault detection.</td></tr>
<tr><td colspan="3">Affected Attributes</td></tr>
<tr><td>Positively</td><td colspan="2">Negatively</td></tr>
<tr><td>Safety: Hazardous situations can be detected and handled</td><td colspan="2">Availability: Decreased if the system goes into its safe state</td></tr>
<tr><td rowspan="2">General Scenarios</td><td>SC1</td><td colspan="3">A fail-safe state is entered if a primary channel failure is detected.</td></tr>
<tr><td>SC2</td><td colspan="3">Hazards in the primary channel can be detected.</td></tr>
<tr><td>Known Uses</td><td colspan="4">- Fly-by-Wire System [Jacazio et al., 2008]
- Vehicle Simulator [Chao et al., 2004]
- WSN Testbed [Bapat et al., 2007]</td></tr>
<tr><td>Credits</td><td colspan="4">[Douglass, 2002] introduces the pattern. [Grunske, 2003] presents a more general version of this pattern and [Armoush, 2010] adds detailed information about quality attribute related consequences.</td></tr>
</table>

Pattern Name	WATCHDOG PATTERN	**Pattern Type**	hardware, fail-safe
Also Known As	Watchdog Timer, Watchdog Processor, Hardware Watchdog Pattern		
Context	A system provides a timing-critical safety functionality.		
Problem	How to make sure that the internal computational processing is proceeding properly and timely.		
Forces	- Full redundancy solutions are expensive - An unavailable component cannot tell that it is unavailable		
Solution	A separate *Watchdog* hardware component receives liveness messages from the *Primary Channel*. If the *Watchdog* does not receive the expected messages, it will initiate a corrective action such as a shutdown signal. 		
GSN Diagram	*Used Tactics:* Override, Heartbeat, Sanity Check 		
Consequences	The watchdog can detect failures of the primary channel if the failure affects the liveness messages. The watchdog is diverse from the primary channel which allows limited systematic fault detection.		
	Affected Attributes		
	Positively	**Negatively**	
	Safety: Timing faults are handled	*Availability:* Decreased if the system shuts down	
General Scenarios	**SC1**	A fail-safe state is entered if a primary channel failure is detected.	
	SC2	A timing fault in the primary channel can be detected.	
Known Uses	- Medical Robot [Guiochet and Vilchis, 2002] - Traffic Management System [Stögerer and Kastner, 2010] - Software implemented watchdog pattern [Chen et al., 2007]		
Credits	[Douglass, 2002] introduces the pattern. [Grunske, 2003] and [Hanmer, 2007] also present this pattern and [Armoush, 2010] adds detailed information about quality attribute related consequences.		

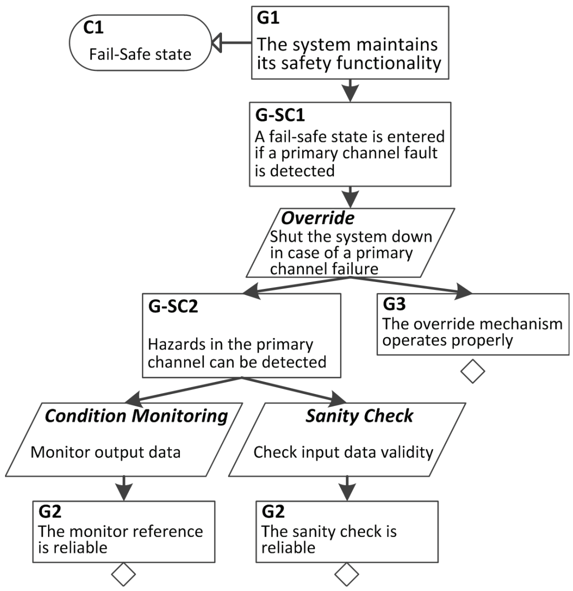
Pattern Name	Safety Executive Pattern	**Pattern Type**	hardware, fail-safe
Also Known As	Safety Kernel Pattern, Shadow-Pattern, Simplex-Pattern		
Context	A system with a complex fail-safe state should maintain its safety functionality even in case of faults.		
Problem	How to check if a fail-safe state should be entered and how to maintain it.		
Forces	- Full redundancy solutions are expensive - An unavailable component cannot tell that it is unavailable - Complex fail-safe state		
Solution	The *Primary Channel* performs all the required functionality. An optional *Fail-Safe Channel* executes just the safety-critical functionality. A centralized *Safety Executive* component coordinates all safety-measures required to shut down the system or to switch over to the *Fail-Safe* processing channel.  		
GSN Diagram	*Used Tactics:* Degradation, Override, Heartbeat, Sanity Check  		

<table>
<tr><td colspan="2" rowspan="4">Consequences</td><td colspan="2">The fail-safe processing channel is diverse from the primary channel which allows limited systematic fault detection.</td></tr>
<tr><td colspan="2">Affected Attributes</td></tr>
<tr><td>Positively</td><td>Negatively</td></tr>
<tr><td>Safety: A safe-state is entered if a fault occurs</td><td>Availability: Decreased if the system goes into its safe state</td></tr>
<tr><td rowspan="2">General Scenarios</td><td>SC1</td><td colspan="2">A fail-safe state is entered if a primary channel failure is detected.</td></tr>
<tr><td>SC2</td><td colspan="2">A timing fault in the primary channel can be detected.</td></tr>
<tr><td>Known Uses</td><td colspan="3">- Safety Kernel for Weapon System [Michael et al., 2006]
- Cardiac Pacemaker [Bak et al., 2009]
- Train Control System [Ghosh et al., 1995]</td></tr>
<tr><td>Credits</td><td colspan="3">[Douglass, 2002] introduces the pattern and [Armoush, 2010] adds detailed information about quality attribute related consequences.</td></tr>
</table>

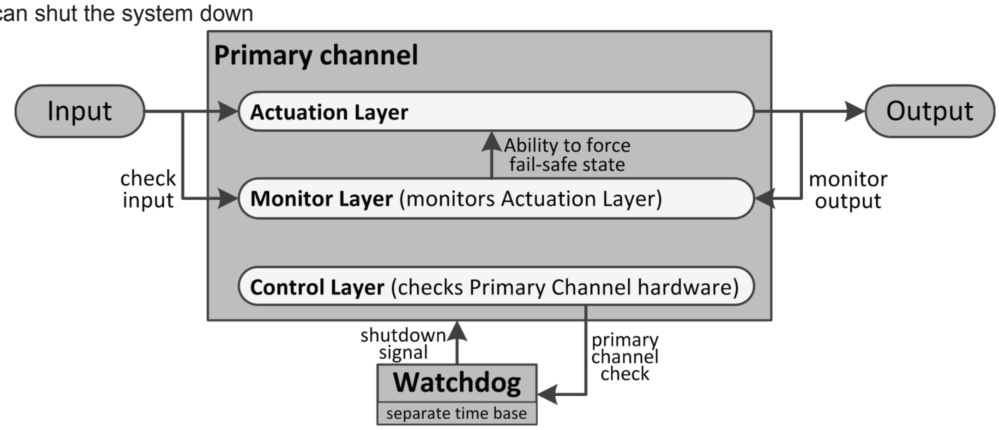
<table>
<tr><td>Pattern Name</td><td colspan="2">PROTECTED SINGLE CHANNEL</td><td>Pattern Type</td><td>hardware/software, fail-safe</td></tr>
<tr><td>Also Known As</td><td colspan="4">Safety Kernel Pattern, Shadow-Pattern, Simplex-Pattern</td></tr>
<tr><td>Context</td><td colspan="4">A system with a fail-safe state and with low availability requirements.</td></tr>
<tr><td>Problem</td><td colspan="4">Find an appropriate mechanism to handle failures or errors that can lead to known hazards.</td></tr>
<tr><td>Forces</td><td colspan="4">- Full redundancy solutions are expensive
- Components have become so complex that we cannot assume them to be error free
- Not any additional hardware components can be introduced</td></tr>
<tr><td>Solution</td><td colspan="4">The input and/or output data of the Primary Channel is monitored and checked regarding its validity or compared to reference data or expected data.



</td></tr>
<tr><td>GSN Diagram</td><td colspan="4">Used Tactics: Override, Condition Monitoring, Sanity Check

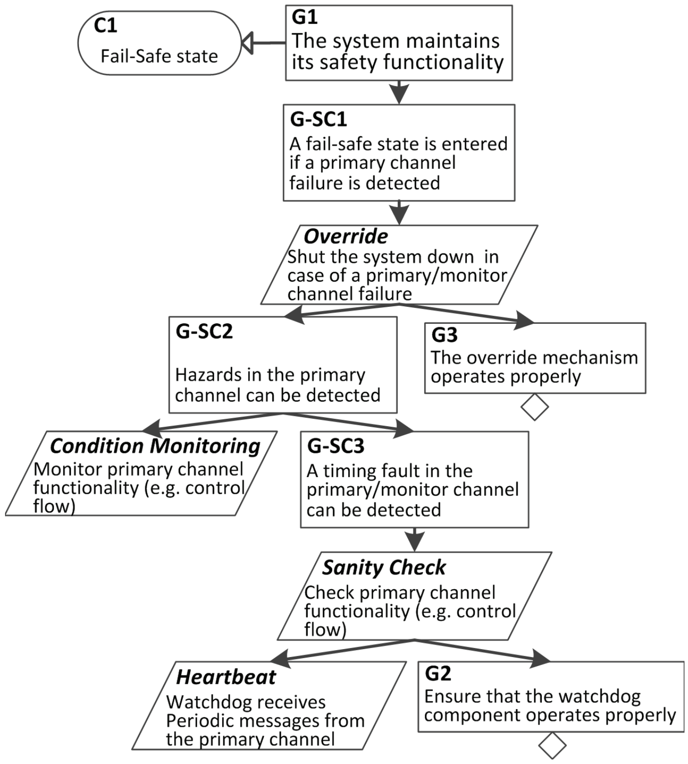


</td></tr>
<tr><td rowspan="4">Consequences</td><td colspan="4">The checks are diverse from the primary channel functionality which allows limited systematic fault detection.</td></tr>
<tr><td colspan="4">Affected Attributes</td></tr>
<tr><td colspan="2">Positively</td><td colspan="2">Negatively</td></tr>
<tr><td colspan="2">Safety: Known hazards can be handled</td><td colspan="2">Availability: Decreased if the system goes into its safe state</td></tr>
<tr><td rowspan="2">General Scenarios</td><td>SC1</td><td colspan="3">A fail-safe state is entered if a primary channel fault is detected.</td></tr>
<tr><td>SC2</td><td colspan="3">Hazards in the primary channel can be detected.</td></tr>
<tr><td>Known Uses</td><td colspan="4">- Robot hand [S. P. Kumar et al., 2011]
- Protected Single Channel implementation in C [Douglass, 2010]
- Real-time autonomic system architecture [Solomon et al., 2007]</td></tr>
<tr><td>Credits</td><td colspan="4">[Douglass, 2002] introduces the pattern. [Grunske, 2003] also presents this pattern and [Armoush, 2010] adds detailed information about quality attribute related consequences.</td></tr>
</table>

Pattern Name	3-Level Safety Monitoring	**Pattern Type**	hardware/software, fail-safe
Also Known As	Safety Kernel Pattern, Shadow-Pattern, Simplex-Pattern		
Context	A system with a fail-safe state and with low availability requirements		
Problem	Find an appropriate mechanism to handle failures or errors that can lead to known hazards.		
Forces	- Full redundancy solutions are expensive - Components have become so complex that we cannot assume them to be error free		
Solution	Divide the system into 3 layers: - The *Actuation Layer* performs the system functionality - The *Monitoring Layer* monitors the *Actuation Layer* and forces a fail-safe state if values deviate too much from references - The *Control Layer* checks the system hardware and sends messages to a *Watchdog* component which can shut the system down		
			

GSN Diagram

Used Tactics: Override, Sanity Check, Condition Monitoring, Heartbeat



Consequences		The pattern is not applicable for systems with high availability requirements. The checks are diverse from the primary channel which allows limited systematic fault detection.	
		Affected Attributes	
		Positively	**Negatively**
		Safety: Known hazards can be handled	*Availability:* Decreased if the system goes into its safe state
General Scenarios	**SC1**	A fail-safe state is entered if a primary channel failure is detected.	
	SC2	Hazards in the primary channel can be detected.	
	SC3	A timing fault in the primary/monitor channel can be detected.	
Known Uses	- E-Gas unit to control the drive power of a motor vehicle [Bederna and Zeller, 1999] - Standardized E-Gas concept [EGAS, Arbeitskreis, 2006] - Yokogawa ProSafe-RS PLC [Emori and Kawakami, 2005]		
Credits	[Armoush, 2010] presents this pattern with detailed information about quality attribute related consequences.		

B. SAFETY TACTICS

This section presents the full list of safety tactics from [Preschern et al., 2013]. These safety tactic are used for the Tactic Topology Models in Appendix C.

Tactics are architectural design decisions which influence and manipulate quality attributes [Bachmann et al., 2003]. Compared to design patterns, they rather describe general concepts or principles and not specific solutions for a problem in a given context. It is often hard to decide whether something is a tactic or a pattern, because there is no precise definition for tactics. However, in this work we simply take a given set of safety tactics from [Preschern et al., 2013] (where a more detailed explanation about safety tactics can be found) and consider every architecture which uses several of these tactics as architectural safety patterns.

Tactic	Aim	Description	IEC 61508 methods
Simplicity	Avoid failures through keeping the system as simple as possible.	*Simplicity* reduces the system complexity. It includes structuring methods or cutting unnecessary functionality and organizes system elements or reduces them to their core safety functionality, thus, eliminating hazards. An example for the application of the *Simplicity* tactic is an emergency stop switch system which is usually kept as simple as possible.	IEC 61508-7: B.2.1 structured specification, B.3.2 structured design, C.2.7 structured programming, E.3 structured description method, C.4.2 programming language subset, C.4.2 limit asynchronous constructs, E.5.13 software complexity controller
Substitution	Avoid failures though usage of more reliable components.	Components or methods are replaced by other components or methods one has higher confidence in. For hardware and software this can mean usage of existing components which are well-proven in the safety domain.	IEC 61508-7: B.3.3 usage of well-proven components, B.5.4 field experience, C.2.10 usage of well-proven/verified software elements, E.20 application of validated soft-cores, E.35 application of validated hard-cores, E.41 usage of well-tried circuits, C.4.3 certified tools and compilers, C.4.4 well-proven tools and compilers, E.4 well-proven tools, E.42 well-proven production process, E.28 application of well-proven synthesis tools, E.29 application of well-proven libraries
Sanity Check (Checking)	Detection of implausible system outputs or states.	The *Sanity Check* tactic checks whether a system state or value remains within a valid range which can be defined in the system specification or which is based on knowledge about the internal structure or nature of the system. An example for a *Sanity Check* is a stuck-at fault RAM-test which checks the proper functionality of the memory during system runtime. The test is based on the understanding of the memory behavior (if we write data to the memory, we should later on be able to read the same data). Faults are detected if the memory behaves differently.	A.1.2 monitoring relay contacts, A.2.7 analog signal monitoring, A.3.1-A.3.3 self-tests, A.4.1-A.4.4 checksums, A.5.1-A.5.5 RAM-Tests, A.6.1 test pattern, A.7.1 one-bit hardware redundancy, A.7.2 multi-bit hardware redundancy, A.7.4 inspection using test patterns, A.9 temporal and logical program monitoring, C.3.3 assertion programming, C.5.3 interface checking, C.4.1 strong typed programming language
Condition Monitoring (Checking)	Detect deviations from the intended system outputs or states.	*Condition Monitoring* checks whether a system value remains within a reasonable range compared to a more reliable, but usually less accurate, reference value. The reference value is computed at runtime by a redundant part in the implementation which can be based on system input values and is not pre-known from the specification (like it would be the case for *Sanity Check*). An example for *Condition Monitoring* is a system which has to be time-synchronized via the Internet and which checks if the synchronized time is feasible by comparing it to an internal clock.	IEC 61508-7: A.1.1 failure detection by online monitoring, A.6.4 monitored outputs, A.8.2 voltage control, A.9 temporal and logical program monitoring, A.12.1 reference sensor, A.13.1 monitoring
Comparison	Detection of discrepancies of redundant system outputs.	*Comparison* tests if the outputs of fully redundant subsystems are equal in order to detect failures. The *Comparison* tactic usually implies the usage of a redundancy tactic. An example for the application of the *Comparison* tactic is a dual-core processor running in lock-step mode. The processor runs the same software on both cores and compares their outputs after each cycle.	IEC 61508-7: A.1.3 comparator, A.6.5 input comparison/voting

Tactic	Aim	Description	IEC 61508 methods
Diverse Redundancy (Redundancy)	Introduction of a redundant system which allows detection or masking of failures in the specification or implementation as well as random hardware failures.	*Diverse Redundancy* can be applied to the specification or to the implementation level. In a system using *Diverse Redundancy* on the implementation level, redundant components use different implementations which were developed independently from the same specification. *Diverse Redundancy* on a specification level goes one step further and additionally requires that even the requirement specifications for the redundant components have to be set up by individual teams.	IEC 61508-7: A.7.6 information redundancy, A.13.2 cross-monitoring of multiple actuators, B.1.4 diverse hardware, C.4.4 diverse programming
Replication Redundancy (Redundancy)	Introduction of a redundant systems which allows detection or masking of random hardware failures (not systematic failures).	*Replication Redundancy* means introduction of a redundant system of the same implementation. The redundant systems maintain the same functionality, use identical hardware, and run the same software implementation. An example for *Replication Redundancy* is the RAID1 data storage technology.	IEC 61508-7: A.2.1 tests by redundant hardware, A.2.5 monitored redundancy, A.3.5 reciprocal comparison by software, A.4.5 block replication, A.6.3 multi-channel output, A.7.3 complete hardware redundancy, A.7.5 transmission redundancy
Repair (Recovery)	Bring a failed system back to a state of full functionality.	The full system functionality is manually or automatically restored if a system failure occurs.	IEC 61508-7: C.3.9 error correction, C.3.10 dynamic reconfiguration
Degradation (Recovery)	*Degradation* brings a system with an error into a state with reduced functionality in which the system still maintains the core safety functions.	*Degradation* systems define a core safety functionality. The systems maintain this safety functionality and additional non-critical functions. In case of an error, the system falls back into a degraded mode in which it just maintains the core safety functionality. An example where the *Degradation* tactic is often applied are automation systems. These systems control safety-critical processes and often visualize these processes in a GUI. If the system has too few resources (e.g. processing time), then the system stops the GUI service and just focuses on its core functionality to control the safety-critical processes.	IEC 61508-7: A.8 voltage supply error handling, C.3.8 degraded function limitation
Voting (Masking)	Mask the failure of a subsystem so that the failure does not propagate to other systems.	*Voting* makes a failure transparent. The tactic does not try to repair the failure, but it hides the failure through choosing a correct result from redundant subsystems. It decides for the majority of the output values.	IEC 61508-7: A.1.4 voter, A.6.5 input comparison/voting
Override (Masking)	Mask the failure of a subsystem so that the failure does not propagate to other systems.	The *Override* tactic forces the system output to a safe state. For example, if we have a system which is in a safe state when shut off, we can apply the *Override* tactic to shut off the system if we have doubt about the system output (e.g. if an output validity check fails). In this scenario overriding the system output with a safe output value decreases the availability of the system. Another form of the *Override* tactic, which does not decrease the availability and is closely related to the *Voting* tactic, chooses the output of redundant subsystems by preferring one subsystem or one output state over another.	IEC 61508: Fail-Safe Principle, A.1.3 comparator
Barrier	Protect a subsystem from influences or influencing other subsystems.	The *Barrier* tactic provides a mechanism to protect from unintentional influences between subsystems. To apply *Barrier*, the interfaces between subsystems have to be analyzed and specified. These interfaces are controlled at runtime by a trustworthy component (the *Barrier*) which often is an already existing reliable mechanism. An example for a *Barrier* is a memory protection unit which controls and restricts the communication between different tasks.	IEC 61508-7: A.11 separation of energy lines from information lines, B.1.3 separation of safety functions from non-safety functions, B.3.4 modularization, C.2.8 information hiding/ encapsulation, C.2.9 modular approach, E.12 modularization, C.3.11 time-triggered architecture

C. RETRIEVING TACTICS FROM SAFETY PATTERNS

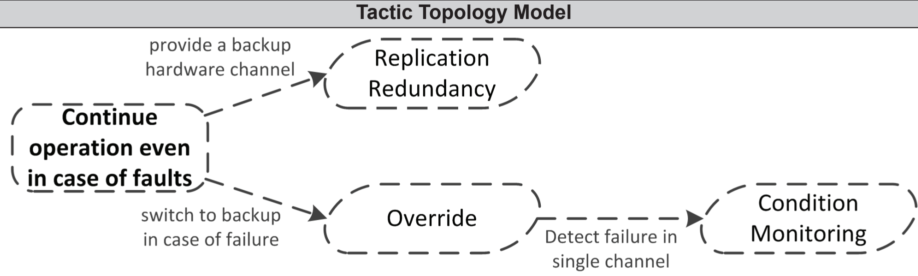
In this section we present the tactic mining process as proposed in Section 4 and its results. We analyzed the safety patterns regarding the tactics they use as described by [K. Kumar and Prabhakar, 2010] where all pattern sections are analyzed and compared to tactic descriptions. If a part of a pattern section is similar to a safety tactic, then this tactic is used in the pattern. An overview of the considered safety tactics is given in Appendix B.

In the left column of the following tables we give the original statements from the context, problem, solution, consequences, and implementation sections of the patterns. In the right column we give the tactic which is addressed by the description on the left. Additionally we elaborate the problem section as described by [Babar, 2007] to gather general scenarios for the patterns. For tactic mining we put special focus on the solution section as suggested by [Zhu et al., 2004].

With the gathered tactics we construct a *Tactic Topology Model* which was introduced by [K. Kumar and Prabhakar, 2010]. The model takes the main tactic to handle the pattern problem as root node. If the application of this tactic introduces new goals, then additional tactics to handle them are added as child nodes. The edges of the model give further explanation of the tactic application.

HOMOGENOUS DUPLEX PATTERN

Abstract Section		
Core Intent		**Tactic**
It is a hardware pattern that is used to increase the safety and reliability of the system by providing a replication of the same module (Modular redundancy) to deal with the random faults.		Replication Redundancy
Problem Section		
Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
Make the system continue operating in the presence of a fault	The system is fully operational even in case of a single channel failure. A single channel random fault does not lead to a system failure. The system can detect a fault in a single channel.	Replication Redundancy Override
Solution Section		
Solution Description		**Tactic**
The system consists of two identical modules; a primary (active) module and secondary (standby) *Test by redundant hardware*		Replication Redundancy
There is a fault detection unit that monitors the primary module and switches to the secondary module when a fault appears in the primary. *Fault detection and diagnosis (Comparator and Acceptance Test)*		Override
This method performs a check on the two channels by checking for input valid data within a given range and by checking the output signals from the two modules whether they are valid or not.		Condition Monitoring
Consequences Section		
Consequence Description		**Tactic**
When a fault is detected in the primary channel, the switch circuit switches over to the secondary channel		Override
Implementation Section		
Implementation Description		**Tactic**
To implement this pattern, the computational channel should be duplicated		Replication Redundancy
Tactic Topology Model		



HETEROGENOUS DUPLEX PATTERN

Abstract Section

Core Intent	**Tactic**
Solution for embedded system with no fail-safe-state in a situation that includes high random failure and high systematic failure rate.	Diverse Redundancy

Problem Section

Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
How to deal with systematic faults as well as random faults in order to increase the safety and reliability of the system. How to make the system continue operating in the presence of a fault in one of the system components	The system is fully operational even in case of a single channel failure. A single channel random/systematic fault does not lead to a system failure. The system can detect a fault in a single channel.	Diverse Redundancy Override

Solution Section

Solution Description	**Tactic**
The system consists of two modules (channels) with the same functionality; a primary (active) module and secondary (standby). *Test by redundant hardware*	Redundancy
There is a fault detection unit that monitors the primary module and switches to the secondary module when a fault appears in the primary. *Fault detection and diagnosis (Comparator and Acceptance Test)*	Condition Monitoring
The two modules have independent designs or implementation methods, which gives this pattern the ability to handle systematic faults as well as random faults. *Diverse Hardware*	Diverse Redundancy
When there is a fault in the primary channel, the comparator has to detect and to identify the faulty channel, then it generates an instruction to the switch circuit to switch to the secondary channel	Condition Monitoring Override
This method performs a check on the two channels by checking for input valid data within a given range and by checking the output signals from the two modules whether they are valid or not.	Condition Monitoring

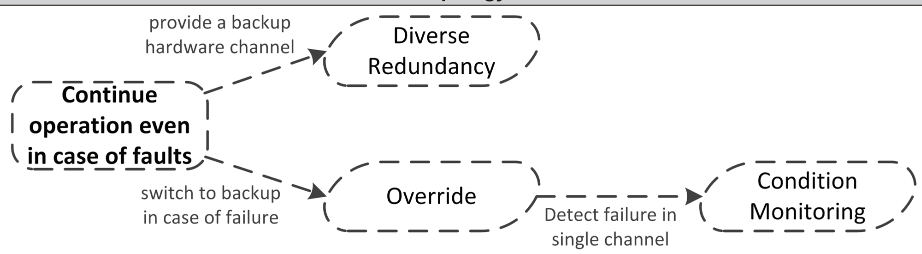
Consequences Section

Consequence Description	**Tactic**
This pattern includes two independent and diverse modules	Diverse Redundancy
When a fault is detected, the switch circuit switches over to the secondary channel	Redundancy Override

Implementation Section

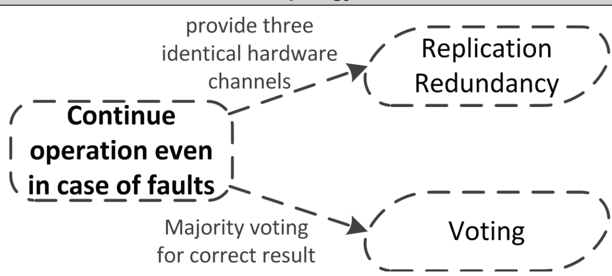
Implementation Description	**Tactic**
To implement this pattern, the computational channel should be duplicated	Redundancy
The duplicated modules should be implemented using independent designs or independent methods to avoid common systematic faults. It is more preferable to use different software versions that are designed by different teams and using different algorithms, when it is possible.	Diverse Redundancy

Tactic Topology Model



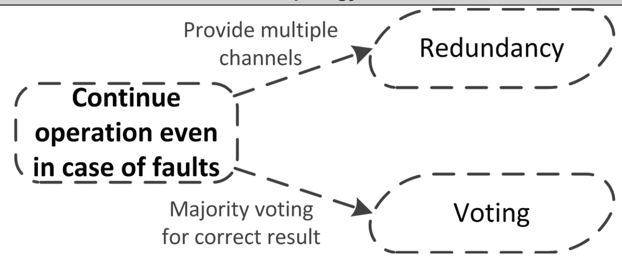
Triple Modular Redundancy Pattern

Abstract Section		
Core Intent		**Tactic**
This pattern consists of three identical modules operating in parallel to produce three results that are compared using a voting system to produce a common result		Voting Replication Redundancy
Problem Section		
Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
How to deal with random faults and single-point of failure in order to increase the safety and reliability of the system without losing the input data in the presence of faults.	The system is fully operational even in case of a single channel failure. A single channel random fault does not lead to a system failure.	Voting
Solution Section		
Solution Description		**Tactic**
The system contains three identical modules or channels operating in parallel. *Test by redundant hardware*		Replication Redundancy
The voter plays a main role in this pattern by applying the voting policy to take the majority from the results which represents the correct actual result. *Fault detection and diagnosis (Voting)*		Voting
Consequences Section		
Consequence Description		**Tactic**
This pattern has a high recurring cost due to the using of three parallel modules. So, the recurring cost is 300% comparing to the basic system.		Replication Redundancy
The cost of voter which is normally a simple hardware circuit that depends on the type of the output control signal and the implementation method.		Voting
Implementation Section		
Implementation Description		**Tactic**
To implement this pattern, the designer should replicate the channel which includes the replication of the hardware as well as software.		Replication Redundancy
Tactic Topology Model		



M-OUT-OF-N PATTERN

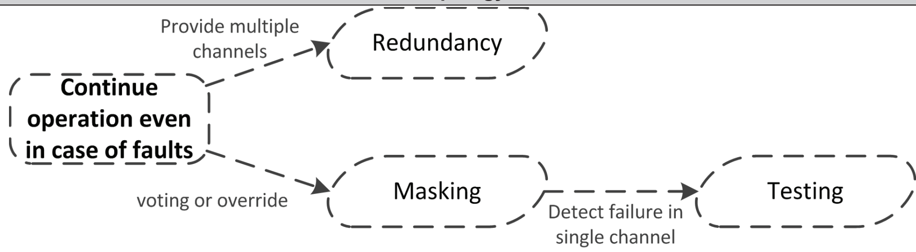
Abstract Section		
Core Intent		**Tactic**
The M-oo-N redundancy requires that at least M components succeed out of the total N parallel modules for the system to succeed.		Voting Redundancy
Problem Section		
Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
How to deal with random or systematic faults in order to increase the safety and reliability of the system without losing the input data.	The system is fully operational even in case of a single channel failure. A single channel random/systematic fault does not lead to a system failure.	Voting
Solution Section		
Solution Description		**Tactic**
The pattern structure contains N identical modules or channels *Test by redundant hardware*		Redundancy
The voting element plays the main role in this pattern since it is used to find the possible correct result by performing the M-oo-N voting strategy *Fault detection and diagnosis (Voting)*		Voting
Consequences Section		
Consequence Description		**Tactic**
This pattern has little influence on the executing time, since the N modules are running separately.		Redundancy
Implementation Section		
Implementation Description		**Tactic**
For homogeneous implementation of this pattern, the designer should use the same hardware as well as the software for all the channels.		Replication Redundancy
If the hardware diversity concept is used in the implementation to solve the problem of systematic faults, then the possible deviation in value or time between the correct outputs should be taken into consideration in the design of the voting system		Diverse Redundancy
Tactic Topology Model		



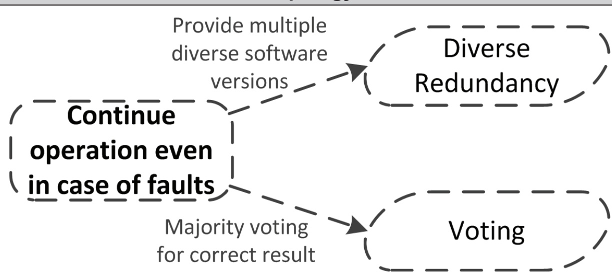
M-out-of-N-D Pattern

Papers about the MooND Architecture	
Citation	**Tactic**
In addition, if the diagnostic tests in either channel detect a fault then the output voting is adapted so that the overall output state then follows that given by the other channel [International Electrotechnical Commission, 2010]	Checking Override Voting
MooND means M out of N channel architecture with diagnostic [H. Yang and X. Yang, 2010]	Replication Redundancy Diverse Redundancy Voting Checking
The architecture consists of equipment with inputs and outputs wired in parallel [Goble, 1998]	Redundancy
Faulty sensors and actuators can also be detected and isolated from the function with proper diagnostics. This method is designated with an abbreviation MooND, where 'D' stands for diagnostics. [Varjoranta, 2012]	Checking

Tactic Topology Model

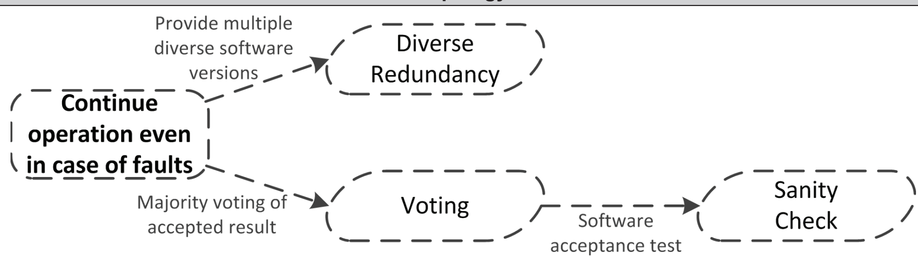


N-Version Programming Pattern

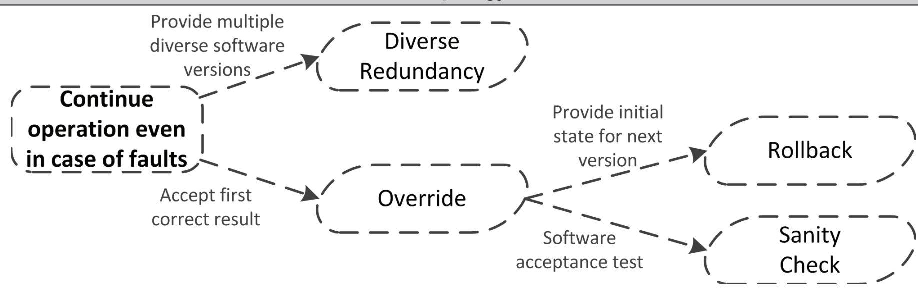
<table>
<tr><th colspan="3">Abstract Section</th></tr>
<tr><th colspan="2">Core Intent</th><th>Tactic</th></tr>
<tr><td colspan="2">independent generation of N>=2 functionally equivalent software modules called ’versions’ from the same initial specification</td><td>Diverse Redundancy</td></tr>
<tr><th colspan="3">Problem Section</th></tr>
<tr><th>Problem</th><th>Elaboration of Problem (scenario)</th><th>Achieved through Tactic</th></tr>
<tr><td>Overcome software faults, which may remain after the software development.</td><td>The system is fully operational even in case of a failure of a software version.
A software fault in a single version does not lead to a system failure</td><td>Diverse Redundancy
Voting</td></tr>
<tr><th colspan="3">Solution Section</th></tr>
<tr><th colspan="2">Solution Description</th><th>Tactic</th></tr>
<tr><td colspan="2">The N-Version Programming Pattern is based on the concept of independent generation of functionally equivalent N versions from the same initial specification.
Diverse programming</td><td>Diverse Redundancy</td></tr>
<tr><td colspan="2">The outputs of these versions are sent to the voter which executes a voting strategy to determine the best correct output. Fault detection with voting</td><td>Voting</td></tr>
<tr><th colspan="3">Consequences Section</th></tr>
<tr><th colspan="2">Consequence Description</th><th>Tactic</th></tr>
<tr><td colspan="2">The main drawbacks of the NVP Pattern are the complexity of developing independent N-versions</td><td>Diverse Redundancy</td></tr>
<tr><th colspan="3">Implementation Section</th></tr>
<tr><th colspan="2">Implementation Description</th><th>Tactic</th></tr>
<tr><td colspan="2">The success of the NVP Pattern depends on the independent development of the required N versions and the level of diversity in these versions to avoid the common failures</td><td>Diverse Redundancy</td></tr>
<tr><td colspan="2">There are several voting techniques that can be used in this pattern to implement the voter component</td><td>Voting</td></tr>
<tr><th colspan="3">Tactic Topology Model</th></tr>
<tr><td colspan="3">
</td></tr>
</table>

Acceptance Voting Pattern

Abstract Section		
Core Intent		**Tactic**
Acceptance Voting Pattern is based on the independent generation of N>=2 functionally equivalent software modules called ŞversionsŤ from the same initial specification		Diverse Redundancy
Problem Section		
Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
How to overcome the software faults, which may remain after the software development, in order to improve the software reliability and safety	The system is fully operational even in case of a failure of a software version. A software fault in a single version does not lead to a system failure. A fault in a single software version is detected.	Diverse Redundancy Voting
Solution Section		
Solution Description		**Tactic**
It includes N independent and functionally equivalent versions that are typically executed in parallel to perform the required task. *Diverse programming*		Diverse Redundancy
The output of each version is tested for correctness using an acceptance test.		Sanity Check
Those results that pass the acceptance test are then used by the voting algorithm to generate the final result.		Voting
Fault detection and diagnosis (Voting and Acceptance Test)		Sanity Check Voting
Consequences Section		
Consequence Description		**Tactic**
The voter has to wait for the outputs of all versions to be checked by the acceptance test before applying the voting algorithm		Voting Sanity Check
The development cost include the development of independent and functionally equivalent N versions.		Diverse Redundancy
-		-
Implementation Section		
Implementation Description		**Tactic**
The quality of the acceptance. Thus, it should be carefully designed to detected most of the possible software faults.		Sanity Check
The independent development of the required N versions and the level of diversity in these versions to avoid the common failures		Diverse Redundancy
The use of a suitable voting technique		Voting
Tactic Topology Model		

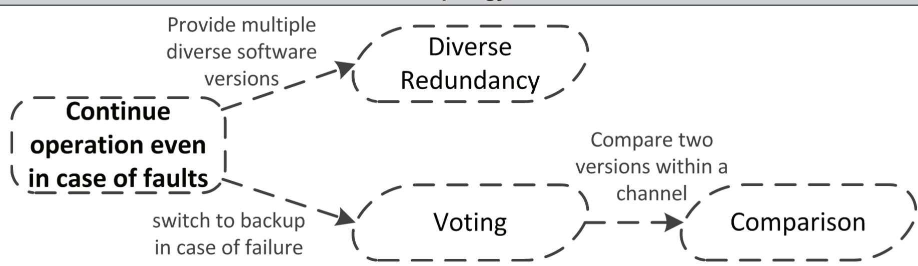


RECOVERY BLOCK PATTERN

<table>
<tr><th colspan="3">Abstract Section</th></tr>
<tr><th colspan="2">Core Intent</th><th>Tactic</th></tr>
<tr><td colspan="2">It includes N diverse, independent, and functionally equivalent software modules called ’versions’</td><td>Diverse Redundancy</td></tr>
<tr><th colspan="3">Problem Section</th></tr>
<tr><th>Problem</th><th>Elaboration of Problem (scenario)</th><th>Achieved through Tactic</th></tr>
<tr><td>How to overcome the software faults, which may remain after the software development, in order to improve the software reliability and safety</td><td>The system is fully operational even in case of a failure of a software version.
A software fault in a single version does not lead to a system failure.
A fault in a single software version is detected.</td><td>Diverse Redundancy
Override</td></tr>
<tr><th colspan="3">Solution Section</th></tr>
<tr><th colspan="2">Solution Description</th><th>Tactic</th></tr>
<tr><td colspan="2">After the execution of the primary version, the acceptance test is executed to check if the outcome is reasonable and to detect any possible erroneous result.
Fault detection and diagnosis</td><td>Sanity Check</td></tr>
<tr><td colspan="2">The system state should be restored to its original state and an alternate version will be invoked to repeat the same computations.
Recovery block</td><td>Rollback</td></tr>
<tr><td colspan="2">An overall system failure is reported to execute the available safety action such as switching the system into its fail-safe sate, or to shutdown the system.</td><td>Override</td></tr>
<tr><td colspan="2">Diverse programming</td><td>Diverse Redundancy</td></tr>
<tr><td colspan="2">The primary alternate is the one which is intended to be used normally to perform the desired operation</td><td>Override</td></tr>
<tr><th colspan="3">Consequences Section</th></tr>
<tr><th colspan="2">Consequence Description</th><th>Tactic</th></tr>
<tr><td colspan="2">The normal recovery block runs the independent ver- sions serially on a single hardware unit.</td><td>Diverse Redundancy</td></tr>
<tr><td colspan="2">The main drawbacks of the RB are the high dependency on the quality of the acceptance test.</td><td>Sanity Check</td></tr>
<tr><th colspan="3">Implementation Section</th></tr>
<tr><th colspan="2">Implementation Description</th><th>Tactic</th></tr>
<tr><td colspan="2">The acceptance test should be carefully designed to detected most of the possible software faults</td><td>Sanity Check</td></tr>
<tr><td colspan="2">The independent development of the required N versions and the level of diversity in these versions to avoid the common failures.</td><td>Diverse Redundancy</td></tr>
<tr><th colspan="3">Tactic Topology Model</th></tr>
<tr><td colspan="3"></td></tr>
</table>

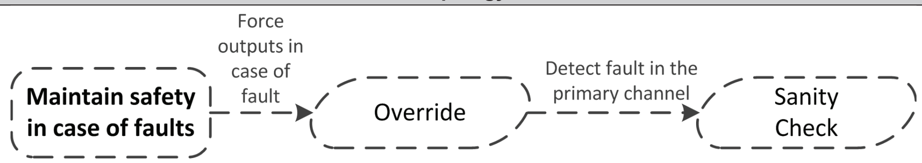
N-Self Checking Programming Pattern

Abstract Section		
Core Intent		**Tactic**
This pattern includes an independent generation of N>=4 functionally equivalent software modules called 'versions' from the same initial specification		Diverse Redundancy
Problem Section		
Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
How to overcome the software faults, which may remain after the software development, in order to improve the software reliability and safety	The system is fully operational even in case of a failure of a software version. A software fault in a single version does not lead to a system failure.	Diverse Redundancy Voting
Solution Section		
Solution Description		**Tactic**
NSCP uses software design diversity and error detection by self-checking programming diverse programming. *Diverse programming*		Diverse Redundancy
each component includes two independent and functionally equivalent versions that run in parallel and are self checked using a comparison algorithm. *Fault detection and diagnosis with a comparator*		Diverse Redundancy Voting
When the running component fails due to different results from its versions, a spare component is invoked to start delivering the required functionality		Diverse Redundancy
If there is no agreement between the two versions, then this component is discarded and a signal is generated to indicate a fault in this component and the selector switches to the next spare component.		Comparison
Consequences Section		
Consequence Description		**Tactic**
Development of independent and functionally equivalent N versions		Diverse Redundancy
developing the comparator and selector unit		Voting Comparison
Implementation Section		
Implementation Description		**Tactic**
The comparator and selector component should be carefully designed to provide an efficient comparison and fast switching.		Voting Comparison
The success of the NSCP Pattern depends on the independent development of the required N versions		Diverse Redundancy
-		-
Tactic Topology Model		



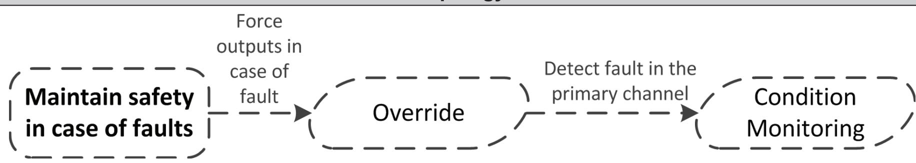
Sanity Check Pattern

Abstract Section		
Core Intent		**Tactic**
The sanity channel, which provides a monitoring to the actuation channel to ensure that the actuation output is approximately correct and within some fixed range.		Override
Problem Section		
Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
Improve the safety of an embedded system in the presence of single point of failure in a system that includes a fail-safe state and low availability requirement.	A fail-safe state is entered if a primary channel fault is detected. Known hazards in the primary channel can be detected.	Override
Solution Section		
Solution Description		**Tactic**
A safety monitoring method switches the system into its fail-safe state in the presence of failure.		Override
In the case of great difference between the set point and the measured value, the sanity channel forces the actuation channel entering the fail-safe state		Sanity Check
the monitor generates a shutdown signal to the actuation channel		Override
In the case of great difference between the set point and the measured value, the sanity channel forces the actuation channel entering the fail-safe state		Sanity Check Override
Consequences Section		
Consequence Description		**Tactic**
If the result of the comparison shows that the output is totally incorrect and may affect the safety of the system, the monitor generates a shutdown signal to the actuation channel		Override
Implementation Section		
Implementation Description		**Tactic**
The implementation of the monitor component is very simple since it is a very simple unit that includes a simple algorithm to perform the required broad range comparison.		Sanity Check
Tactic Topology Model		



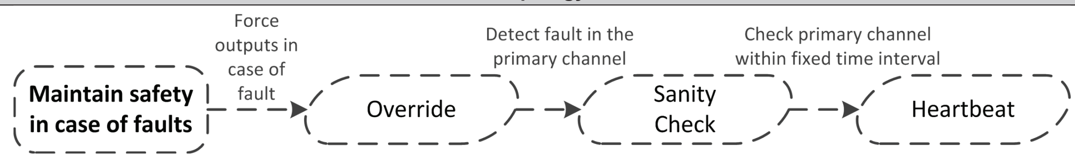
MONITOR-ACTUATOR PATTERN

Abstract Section		
Core Intent		**Tactic**
A monitoring channel monitors the actuation channel in order to detect and to identify the possible faults		Override
Problem Section		
Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
Improve the safety of a system that includes a fail-safe state and low availability requirements at reasonable cost.	A fail-safe state is entered if a primary channel failure is detected. Hazards in the primary channel can be detected.	Override
Solution Section		
Solution Description		**Tactic**
The Monitoring Channel monitors the actuation channel to check its proper operation. It takes the information from the set point source and the actuator sensors to detect possible faults in the actuation channel.		Condition Monitoring
In the case of improper operation, it forces the actuation channel to enter the fail-safe state.		Override
The monitor takes the information about the outputs of the actuators, which is collected by the actuator sensors and processed by the monitoring acquisition system, and compares it with the provided set points.		Condition Monitoring
If the result of the comparison shows improper operation in the actuation channel, the monitor generates a shutdown signal to the actuation channel.		Override
Consequences Section		
Consequence Description		**Tactic**
generate the shutdown signal to force the actuation channel entering its fail-safe state		Override
Implementation Section		
Implementation Description		**Tactic**
it is a good idea for the monitoring channel to store some historical information about the monitored value which could be helpful to determine whether the detected value represents a transient or persistent fault.		Condition Monitoring
Tactic Topology Model		



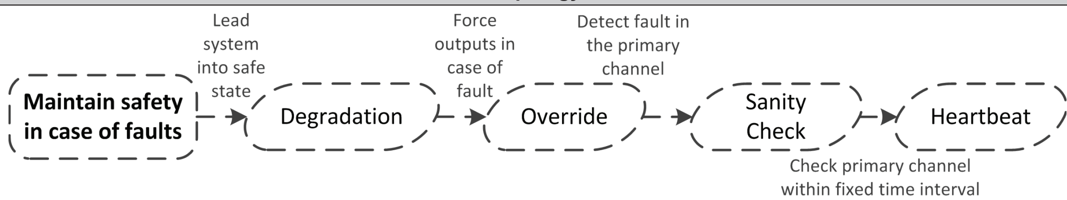
Watchdog Pattern

Abstract Section		
Core Intent		**Tactic**
The pattern widely used in the embedded systems to make sure that the time-dependent computational processing is proceeding properly as expected in a predefined order		Override
Problem Section		
Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
How to make sure that the internal computational processing of the actuation channel is proceeding properly and timely.	A fail-safe state is entered if a primary channel failure is detected. A timing fault in the primary channel can be detected.	Override
Solution Section		
Solution Description		**Tactic**
The watchdog receives liveness messages (Strokes) from the actuation channel on a periodic or in a predefined-sequence base. *Program sequence monitoring*		Heartbeat
The watchdog must be stroked within a specified period of time or it will initiate a corrective action such as a shutdown signal		Override
The Watchdog Pattern checks that the time-dependent computational processing is proceeding properly as expected in a predefined order		Sanity Check
Built In Test (BIT) verifies all or a portion of the internal functionality of the actuation channel.		Sanity Check
Consequently, it issues a shutdown or reset signal to the actuation channel or initiates a corrective action through sending a command signal		Override
Consequences Section		
Consequence Description		**Tactic**
execution of the built in tests that may be initiated by the watchdog.		Sanity Check
Implementation Section		
Implementation Description		**Tactic**
To increase the fault coverage, it is common to invoke a BIT, CRC, or stack overflow check when the watchdog is stroked to ensure that the computational processing of the actuation channel is proceeding properly.		Sanity Check
Tactic Topology Model		



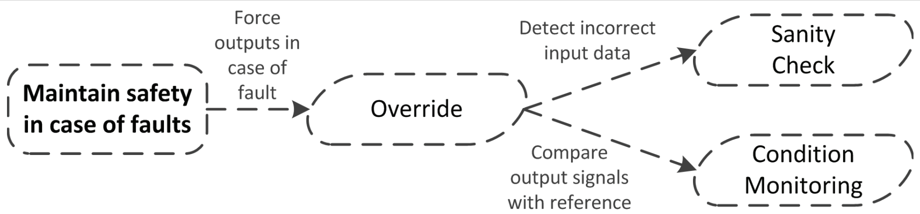
Safety Executive Pattern

Abstract Section		
Core Intent		**Tactic**
The safety executive component is responsible for the shutdown of the system as soon as the watchdog sends a shutdown signal		Degradation
Problem Section		
Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
How to provide a centralized and consistent method for monitoring and controlling the execution of a complex safety measure in case of failures.	A fail-safe state is entered if a primary channel failure is detected. A timing fault in the primary channel can be detected.	Degradation
Solution Section		
Solution Description		**Tactic**
A centralized safety executive component coordinates all safety-measures required to shut down the system or to switch over to the fail-safe processing channel. *Graceful degradation*		Override Degradation
It is an optional component, which is invoked by the watchdog to run a periodic Built In Test (BIT) to verify all or a portion of the internal functionality of the actuation channel. *Program sequence monitoring*		Sanity Check
The watchdog receives liveness messages (strokes) from the components of the actuation channel in a predefined time frame.		Heartbeat
The Safety Executive tracks and coordinates all safety monitoring to ensure the execution of safety actions.		Degradation
Consequently, it issues a shutdown signal to the safety executive component or initiates a corrective action.		Override
Consequences Section		
Consequence Description		**Tactic**
Execution of the periodic built in tests		Sanity Check
Implementation Section		
Implementation Description		**Tactic**
Graceful degradation		Degradation
The designer should determine whether the new components need to send stroke messages to the watchdog or not		Heartbeat
Tactic Topology Model		



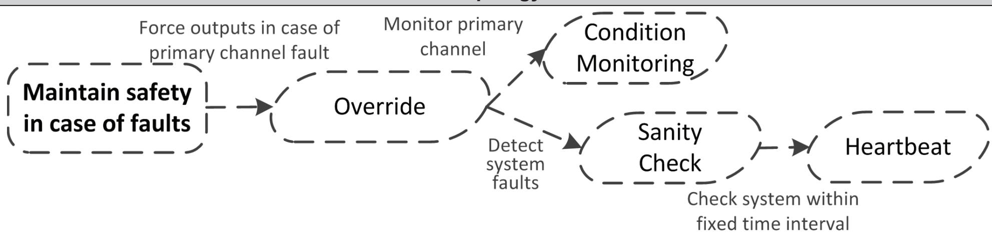
PROTECTED SINGLE CHANNEL PATTERN

Abstract Section		
Core Intent		**Tactic**
It should be integrated with another safety technique in the presence of immediate fail-safe state to be used for light safety-critical applications.		Override
Problem Section		
Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
How to deal with the transient faults to provide some level of safety and reliability to the embedded system in an inexpensive manner	A fail-safe state is entered if a primary channel fault is detected. Hazards in the primary channel can be detected.	Override
Solution Section		
Solution Description		**Tactic**
input data validation and one for actuation monitoring. *Failure detection by online monitoring*		Sanity Check Condition Monitoring
The actuator sensors are used to get feed back signals from the output of the actuators to be used for the actuation monitoring		Condition Monitoring
checks on the input data and the system itself		Sanity Check
Actuator Monitoring: It provides a monitoring to the output of the channel, such as checking the output commands for validity before delivering this command to the actuators. It can also check the output actuators using separate sensors by getting feedback values from the actuators and comparing these values with the previously generated control signals.		Sanity Check Condition Monitoring
Consequently, it can use this information to reconfigure the output processing component to overcome the transient faults when it is possible.		Override
Consequences Section		
Consequence Description		**Tactic**
If the validation checks are performed by hardware, then the extra components are working in parallel with the basic channel which does not affect the basic system in the normal execution		Sanity Check
Implementation Section		
Implementation Description		**Tactic**
In this pattern, there are two versions to be implemented: either (open loop) with only data integrity checking unit, or (close loop) that includes additional actuation monitoring unit.		Sanity Check Condition Monitoring
The existence of a fail-safe state gives the data integrity and the actuator monitoring components the capability to switch the system into the fail-safe state in the presence of persistent fault.		Override
Tactic Topology Model		



3-Level Safety Monitoring Pattern

Abstract Section		
Core Intent		**Tactic**
The monitoring level monitors the first level, and the control level controls the monitoring level and the entire hardware channel.		Condition Monitoring
Problem Section		
Problem	**Elaboration of Problem (scenario)**	**Achieved through Tactic**
How to continue providing the required safety level and to ensure that the system does no injure or harm, when there is any deviation in the output of the actuators from the commanded set point.	A fail-safe state is entered if a primary channel failure is detected. Hazards in the primary channel can be detected A timing fault in the primary/monitor channel can be detected	Condition Monitoring Override
Solution Section		
Solution Description		**Tactic**
The Monitoring Module monitors the actuation module through a comparison of the processing results, input data and the data from the actuatorŠs sensors.		Condition Monitoring
In the case of great difference between the desired value and the measured value, this module forces the actuation channel to switch into its fail-safe state.		Override
If the result of the comparison shows that the output of actuation channel is totally incorrect and may affect the system safety, the monitor will generate a correction command or shutdown signal		Override
Data Validation (Data Integrity Check): It provides a check on the input data during the executing of the desired algorithm to ensure that the input data is valid and in the safe boundaries. *Fault detection and diagnoses*		Sanity Check
A Watchdog is used to provide a sequence control to the monitoring level and to the entire actuation channel. *Program sequence monitoring*		Sanity Check Heartbeat Override
Consequences Section		
Consequence Description		**Tactic**
generate the shutdown or reset signal to force the actuation channel entering the fail- safe state		Override
Implementation Section		
Implementation Description		**Tactic**
If low-sensitive sensors are used for the monitoring level, then a simple algorithm should be used to perform the required broad range comparison.		Condition Monitoring
Tactic Topology Model		



Appendix References

ALVAREZ, Jacobo et al. (2005). Safe PLD-based programmable controllers. In: *International Conference on Field Programmable Logic and Applications*. IEEE, 559–562.

ANDERSON, T O M et al. (1985). Software Fault Tolerance: An Evaluation. *IEEE Transactions on Software Engineering* SE-11, 12, 1502–1510.

ARMOUSH, Ashraf (2010). Design patterns for safety-critical embedded systems. PhD thesis. RWTH Aachen University.

BABAR, M.A. (2007). Improving the Reuse of Pattern-Based Knowledge in Software Architecting. In: *EuroPLoP*. Lero, Ireland, 7–11.

BACHMANN, Felix, Len BASS, and Mark KLEIN (2003). *Deriving Architectural Tactics : A Step Toward Methodical Architectural Design*. Tech. rep. March. Carnegie Mellon Software Engineering Institute.

BAK, Stanley et al. (Apr. 2009). The System-Level Simplex Architecture for Improved Real-Time Embedded System Safety. In: *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 99–107.

BAPAT, Sandip et al. (2007). Chowkidar : A Health Monitor for Wireless Sensor Network Testbeds. In: *3rd International Conference on Testbeds and Research Infrastructure for the Development of Networks and Communities (TridentCom)*. IEEE.

BEDER, Delano M., Jo UEYAMA, and Marcos L. CHAIM (Dec. 2011). A generic policy-free framework for fault-tolerant systems: Experiments on WSNs. In: *2011 IEEE 2nd International Conference on Networked Embedded Systems for Enterprise Applications*. IEEE, 1–7.

BEDERNA, F. and T. ZELLER (1999). *Method and arrangement for controlling the drive unit of a vehicle*.

BORCHERT, Christoph, Horst SCHIRMEIER, and Olaf SPINCZYK (2012). Protecting the Dynamic Dispatch in C ++ by Dependability Aspects. In: *1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*.

BÖRCSÖK, J et al. (2011). High-Availability Controller Concept for Steering Systems : The Degradable Safety Controller. In: *Proceedings of the 2nd international conference on Circuits, Systems, Communications & Computers*, 220–228. ISBN: 9781618040565.

BRILLIANT, S.S., J.C. KNIGHT, and N.G. LEVESON (1990). Analysis of faults in an N-version software experiment. *IEEE Transactions on Software Engineering* 16, 2, 238–247.

CARR, D.W. et al. (2005). An Open On-Board CBTC Controller Based on N-Version Programming. In: *International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce (CIMCA-IAWTIC'06)*. Vol. 1. IEEE, 834–839.

CHAO, H C, T W PEARCE, and M.J.D. HAYES (2004). Use of the HLA in a Real-Time Multi-Vehicle Simulator. In: *The Canadian Society of Mechanical Engineering Forum*, 1–10.

CHEN, Xi et al. (2007). Application of Software Watchdog as a Dependability Software Service for Automotive Safety Relevant Systems. In: *37th International Conference on Dependable Systems and Networks (DSN)*. IEEE.

DOUGLASS, Bruce Powel (2002). *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Pearson.

DOUGLASS, Bruce Powel (2010). *Design Patterns for Embedded Systems in C*. Elsevier.

DURMUÙ, Mustafa Seçkin et al. (2011). A New Voting Strategy in Diverse Programming for Railway Interlocking Systems. In: *International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE)*. IEEE, 723–726.

EGAS, ARBEITSKREIS (2006). *Standardisiertes E-Gas-Ueberwachungskonzept fuer Motorsteuerungen von Otto- und Dieselmotoren*.

EMORI, Toshiyuki and Shigehito KAWAKAMI (2005). Safety technologies incorporated in the safety control system. *Yokogawa Technical Report* 40, 4, 43–46.

FERNANDEZ, A Vergara and R. DENZ (2002). Reliability Analysis for the quench detection in the LHC machine. In: *8th European Particle Accelerator Conference*, 2445–2447.

GHOSH, A.K. et al. (1995). A distributed safety-critical system for real-time train control. In: *21st Annual Conference on IEEE Industrial Electronics*. Vol. 2. IEEE, 760–767.

GOBLE, William M (1998). The Use and Development of Quantitative Reliability and Safety Analysis in New Product Design. PhD thesis. Technical University of Eindhoven.

GRUNSKE, Lars (2003). Transformational Patterns for the Improvement of Safety Properties in Architectural Specification. In: *Proceedings of The Second Nordic Conference on Pattern Languages of Programs (VikingPLoP)*.

GUIOCHET, J. and A. VILCHIS (2002). Safety Analysis of a Medical Robot for Tele-echography. In: *2nd IARP IEEE/RAS joint workshop on Technical Challenge for Dependable Robots in Human Environments*. IEEE, 217–227.

HANMER, Robert S. (2007). *Patterns for Fault Tolerant Software*. Wiley.

INTERNATIONAL ELECTROTECHNICAL COMMISSION (2010). *IEC 61508, Functional Safety of Electrical/ Electronic/ Programmable Electronic Safety Related Systems*.

JACAZIO, G, P Serena GUINZIO, and M SORLI (2008). A dual-duplex electrohydraulic system for the fly-by-wire control of a helicopter main rotor. In: *26th International Congress of the Aeronautical Sciences*, 1–9.

KIM, K H Kane (1998). ROAFTS : A Middleware Architecture for Real-time Object-oriented Adaptive Fault Tolerance Support. In: *3rd International High-Assurance Systems Engineering Symposium*. IEEE.

KOHANAWA, Akihiko, Masami HASEGAWA, and Shigeharu KANAMORI (2010). Safety Control Solutions Protecting Onsite Safety. *Fuji Electric Group* 56, 1.

KUMAR, Kiran and T.V. PRABHAKAR (2010). Design Decision Topology Model for Pattern Relationship Analysis. In: *1st Asian Conference on Pattern Languages of Programs (AsianPLoP 2010)*.

KUMAR, S Phani, P. Seetha RAMAIAH, and V. KHANAA (2011). Architectural patterns to design software safety based safety-critical systems. In: *Proceedings of the 2011 International Conference on Communication, Computing & Security - ICCCS '11*. ACM Press, New York, New York, USA, 620.

KYRIAKOULAKOS, Konstantinos and Dionisios N. PNEVMATIKATOS (2009). A novel SRAM-based FPGA architecture for efficient TMR fault tolerance support. In: *19th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.

LAPRIE, J.C. et al. (1995). Architectural Issues in Software Fault Tolerance. In: *Software Fault Tolerance*. Wiley, 47–80.

LJOSLAND, Ingvar (2006). BUCS : Patterns and Robustness A Navigation System Case Study.

MICHAEL, J Bret, Anil NERODE, and Duminda WIJESEKERA (2006). On the Provision of Safety Assurance via Safety Kernels for Modern Weapon Systems. In: *DTIC Science & Technology*, 102–105.

MIYAWAKI, N. (2008). Study of Machine Safety Control. *JTEKT Engineering Journal* 1004E, 119–124.

MUTLU, Ahmet (2004). DC Motor Speed Controller Software.

NOURANI, Esmaeil and Mohammad Abdollahi AZGOMI (Dec. 2009). A design pattern for dependable web services using design diversity techniques and WS-BPEL. In: *2009 International Conference on Innovations in Information Technology (IIT)*. IEEE, 325–329.

PARCHAS, E. and R. de LEMOS (2004). An architectural approach for improving availability in Web services. In: *Third Workshop on Architecting Dependable Systems (WADS)*. IET.

PRESCHERN, Christopher (2011). PISCAS: Pisciculture Automation System Product Line. MA thesis. Graz University of Technology.

PRESCHERN, Christopher, Nermin KAJTAZOVIC, and Christian KREINER (2013). Catalog of Safety Tactics in the light of the IEC 61508 Safety Lifecycle. In: *VikingPLoP*.

SGHAIRI, M et al. (2008). Challenges in Building Fault -Tolerant Flight Control System for a Civil Aircraft. *IAENG International Journal of Computer Science* 35, 4, 495–499.

SKAMBRAKS, Martin (Sept. 2006). An Architecture for Runtime State Restoration after Transient Hardware-Faults in Redundant Real-Time Systems. In: *Conference on Emerging Technologies and Factory Automation*. IEEE, 78–85.

SOLOMON, Bogdan et al. (May 2007). Towards a Real-Time Reference Architecture for Autonomic Systems. In: *International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '07)*. IEEE.

STÖGERER, Christoph and Wolfgang KASTNER (2010). Distributed Monitoring for Component-based Traffic Management Systems. In: *Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE.

TONG, Adams N. (2007). Fabrication of deep-submicron complementary metal-oxide semiconductor devices. PhD thesis. University of Notre Dame.

VARJORANTA, Velu (2012). Software safety issues in machine control system design process. PhD thesis. Tampere University of Technology.

WANG, Feiyi et al. (2001). SITAR : A Scalable Intrusion-Tolerant Architecture for Distributed Services. In: *Foundations of Intrusion Tolerant Systems (OASIS'03)*. June. IEEE, 5–6.

YANG, Hao and Xianhui YANG (Aug. 2010). Automatic Generation of Markov Models in Safety Instrumented Systems with Non-identical Channels. In: *2010 International Conference of Information Science and Management Engineering*. IEEE, 287–290.

ZHU, Liming, Muhammad Ali BABAR, and Ross JEFFERY (2004). Mining Patterns to Support Software Architecture Evaluation. In: *4th Working IEEE / IFIP Conference on Software Architecture (WICSA)*. IEEE.

ZIMMER, Marcel (2009). Prototypische Implementierung und Evaluation von Sicherheitsmustern in eingebetteten Systemen. MA thesis. Technische Universität Kaiserslautern.