J. Software Engineering & Applications, 2009, 2:1-12
Published Online April 2009 in SciRes (www.SciRP.org/journal/jsea)

Design Pattern Representation for Safety-Critical Embedded Systems

Ashraf Armoush[*], Falk Salewski[*], Stefan Kowalewski[*]

[*]Embedded Software Laboratory, RWTH Aachen University, Aachen, Germany
Email: {armoush, salewski, kowalewski}@embedded.rwth-aachen.de

Received January 9th, 2009; revised February 12th, 2009; accepted March 25th, 2009.

ABSTRACT

Design Patterns, which give abstract solutions to commonly recurring design problems, have been widely used in the software and hardware domain. As non-functional requirements are an important aspect in the design of safety-critical embedded systems, this work focuses on the integration of non-functional implications in an existing design pattern concept. We propose a pattern representation for safety-critical embedded application design methods by including fields for the implications and side effects of the represented design pattern on the non-functional requirements of the overall systems. The considered requirements include safety, reliability, modifiability, cost, and execution time.

***Keywords*:** *Design Pattern*, *Embedded Systems*, *Non-Functional Requirements*, *Safety-Critical Systems*

1. Introduction

Design pattern, originally proposed in [1] by the architect Christopher Alexander, is a universal approach to describe common solutions to widely recurring design problems. Ever since, this concept has been applied in several domains of hardware design (electronics) and also became popular in the software domain after the success of the book Design Patterns: Element of Reusable Object-Oriented Software by Gama *et al*. [2].

As the concept of design pattern aims at supporting designers and system architects in their choice of suitable solutions for commonly recurring design problems, this concept might also be useful to support the design of safety-critical embedded systems. The design of these systems is considered to be a complex process, as hardware and software components have to be considered during the design as well as potential interactions between hardware and/or software components. Moreover, not only functional requirements[1] have to be fulfilled by these systems. Failures in safety-critical systems could result in critical situations that may lead to serious injury or loss of life or unacceptable damage to the environment. Therefore, also the non-functional requirement *safety* has to be considered in these systems to assure that the risk of hazards is acceptable low in the considered system. To support the design of safe devices, safety measures are given by international safety standards as the IEC61508 [3]. Beside life cycle and process requirements, also different measures for the design of software and hardware components are recommended. These safety measures have typically an impact on the cost, the reliability, the real-time behavior of the system, and on the modifiability of the resulting system. Depending on the application domain of the later embedded system, these non-functional requirements are of great importance. For this reason, non-functional requirements should be considered during the design of any safety-critical system.

While current concepts of design pattern exist for many different application domains, they typically lack a consideration of potential side effects on non-functional requirements. In order to integrate these side effects into the pattern concept, we propose an extended template for an effective design pattern representation for safety-critical applications. This pattern representation includes the traditional pattern concept in combination with an extension describing the implications and side effects with respect to the non-functional requirements. While this concept has been described briefly in [4] before, this work focuses on the application of our approach. Thus, two example patterns are included to illustrate the proposed representation of design patterns for software and hardware components in safety-critical applications.

2. Related Work

The field of design pattern is large and still rapidly growing. Many researches have focused on the use of design pattern in the software domain [2,5,6,7,8,9], but further research is still needed in the domain of safety-critical embedded systems to integrate the non-functional requirements in design patterns. In his books [10] and [11], Bruce Douglass proposed several design patterns

[1]Note: functional requirements (FR) represent the required functionality itself while non-functional requirements (NFR) describe additional qualities that have to be achieved by the developed system (e.g. safety, reliability, modifiability, cost and execution time)

for safety-critical systems based on well known fault tolerant design methods and by integrating some modification to increase the safety level on these patterns. Gross and Yu [12] discuss the relationship between non-functional requirements and design patterns, and propose a systematic approach for evaluating design patterns with respect to non-functional requirements. They propose the use of design patterns for establishing traces between non-functional goals in a goal tree such as a soft goal interdependency graph (SIG) and the system design. Cleland-Huang *et al*. [13,14] enhance the patterns defined by Gross and Yu [12] through defining a model for establishing traceability between certain types of non-functional requirements and design and code artifacts, through the use of design patterns as intermediary objects. Xu *et al*. [15] classified the dependability needs into three types of requirements and proposed an architectural pattern that allows requirements engineers and architects to map dependability requirements into three corresponding types of architectural components. Konord *et al*. [16,17] describe a research of how the principle of design pattern can be applied to requirements specifications, which they term requirements patterns for embedded systems. They include a constraints field in the pattern template to show the functional and non-functional restrictions that are applied to the system.

In comparison to our work, none of the aforementioned approaches show clearly the implications on the non-functional requirements as part of the pattern. These patterns and the other developed patterns focus on the traditional structure of the pattern that includes: context, problem and solution. The use of non-functional requirements in these approaches is restricted to the requirement analysis phase of the design process. In these approaches, neither a relative measure nor an indication for the implications of the patterns on the non-functional requirements, were given. To improve these approaches, we propose a new template representation in Section 4 to show the implications of the represented patterns on the non-functional requirements.

3. Design Pattern Template

In this section, the template pattern we propose for the representation of design patterns for safety-critical embedded applications is described. As depicted in Figure 1, the upper part of the template includes the traditional representation of a design pattern while a listing of the pattern implications on the non-functional requirements is given in the *Implication* section. Moreover, further support is given by stating implementation issues, summarizing the consequences and side effects as well as a listing of related patterns.

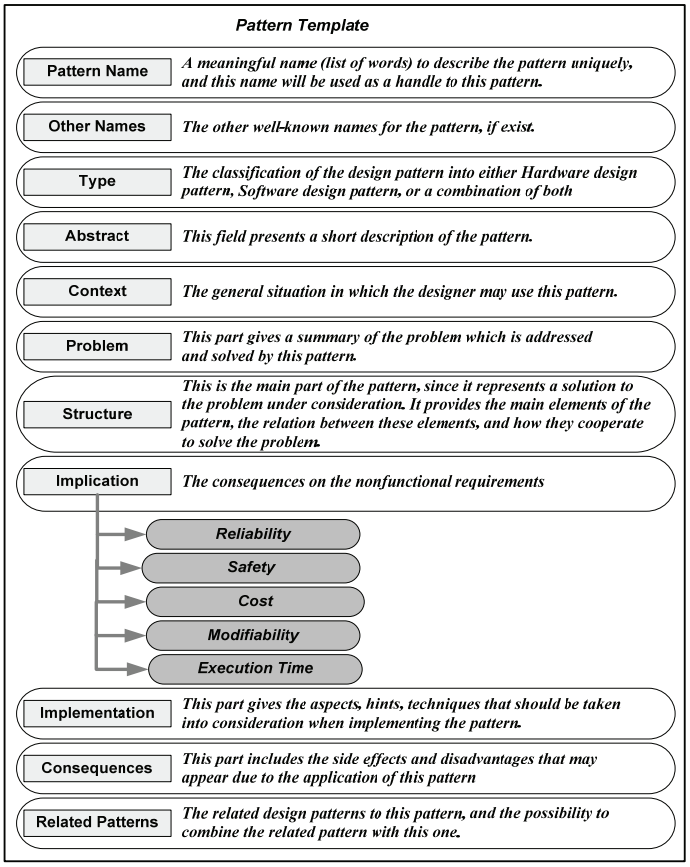

Figure 1. The design pattern template

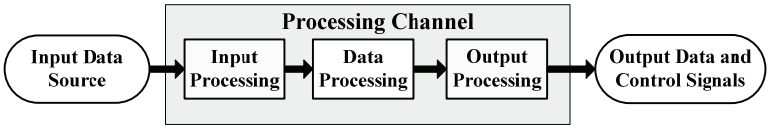

Figure 2. Basic system without specfic safety requirements

The proposed design template includes a part for pattern implications on the non-functional requirements reliability, safety, cost, modifiability and execution time. To allow a suitable description of these implications, the changes/improvements of using the corresponding design pattern are represented relative to a basic simple system (see Figure 2). This basic system has a given reliability (R_{old}), a given cost, a given modifiability and is resulting in a given executing time. Moreover, this basic system has no specific safety measurements.

4. The Implication On Non-Functional Requirements

While the main part of the design pattern proposed does not differ from well known approaches [18,19,20,21], the part for the implications on the non-functional requirements is described in this section. As mentioned above, the implications are stated relative to the basic system without any specific safety method. In the following, the determination of the five implications on non-functional requirements is described:

Reliability: In this context, reliability is defined as the probability that of a system or component to perform its required functions correctly under stated conditions for a specified period of time. This part of implications describes the relative improvement in the system's reliability relative to the maximum possible improvement[2] in reliability, which is defined in the following equation:

$$\text{Reliability Improvement} = \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% \qquad (1)$$

R_{new}: The reliability after using this pattern.

R_{old}: The reliability of the basic system.

Safety: The safety of a system is usually determined by the residual risk of operating this system (see e.g. [3]). Therefore, the notion of risk can be used as a measure for the assessment of safety-critical systems. The problem concerning design patterns is that they describe an abstract solution to a commonly recurring design problem. As it is not related to a specific application or to a specific case, it is difficult to determine an actual value for the possible residual risk without considering a concrete application. To allow an indication of the safety that can be achieved by the application of a specific design pattern, existing recommendations given in safety standards are used. In detail, it is stated to which Safety Integrity Level (SIL) the pattern is recommended in a given safety standard. The safety integrity levels used here include the levels SIL1 to SIL4 as they are defined in the standard IEC61508 [3]. Additionally, the notation SIL0 is used in this template to describe a system without specific safety requirements. If measures are described in design patterns that are not included in current safety standards, these measures have to be assessed in an appropriate manner, e.g. by comparing them to measures with known recommendations.

***Cost*:** The implications on costs include:

- The recurring cost per unit, which reflects the additional costs resulting from additional or specific hardware components required by the design pattern.
- The development cost of applying this pattern.

***Modifiability*:** This implication describes the degree to which the system developed according to this design pattern can be modified and changed.

***Impact on execution time*:** With this implication, the effect of the pattern on the total time of execution at runtime is indicated. It shows the execution time overhead that is resulting from the application of this pattern in the worst and the average cases.

The application of the design pattern proposed, especially the use of the implication part introduced briefly in this section, is described in form of two example patterns in the following section.

5. Example Patterns

Two example patterns are presented in this section to illustrate the application of the proposed approach: The first pattern is a hardware and software pattern that is expected to be suitable for complex and highly safety-critical systems (Safety Executive Pattern). The second pattern is a hybrid[3] software fault tolerance method intended to increase the reliability of the standard N-version programming approach (Acceptance Voting Pattern).

5.1 EXAMPLE 1

In this example pattern, the pattern originally described in [10] is presented in our extended pattern representation including also implications on non-functional requirements.

Pattern Name
Safety Executive Pattern (SEP)

Other Names
Safety Kernel Pattern

[2]Note: the maximum possible improvement in reliability is the difference between the reliability of the basic system and the maximum value for reliability which is equal to 1 (Ideal case without failures).

[3]Note: A pattern is called *hybrid* if the pattern is composed of at least two other patterns.

Type
Hardware and Software

Abstract

The Safety Executive Pattern can be considered as an extension of the Watchdog Pattern[4] targeting the problem that a shutdown of the system by the actuation channel itself might be critical in the presence of faults (shutdown might fail or take too long). This problem occurs especially in those systems in which a complicated series of steps involving several components is necessary to reach a fail-safe state. Therefore, the Safety Executive Pattern uses a watchdog in combination with an additional safety executive component, which is responsible for the shutdown of the system as soon as the watchdog sends a shutdown signal (see also Figure 3. The safety executive pattern). If the system has a safe state, the actuation channel is shutdown via the safety executive component. Otherwise, the safety executive component has to delegate all actuations necessary to an additional fail-safe processing channel.

Context

The application of this pattern is suitable in the following context:

• The considered actuation channel requires a risk reduction by safety measures.

• The considered system has at least one safe state. If this is not the case, an additional fail-safe processing channel has to be applied to overtake necessary actions.

• A shutdown of the actuation channel is complex. As an example, this is the case if several safety-related system actions have to be controlled simultaneously.

• A sufficient determination of failures in the actuation channel can be achieved by a watchdog.

Problem

Provision of a centralized and consistent method for monitoring and controlling the execution of a complex safety measure (shutdown or switch over to redundant unit in case of failures).

Pattern Structure

The Safety Executive Pattern is based on an actuation channel to perform the required functionality and an optional fail-safe processing channel that is dedicated to the execution and control of the fail-safe processing (see also Figure 3). The central part of this pattern is the existence of a centralized safety executive component coordinating all safety-measures required to shut down the system or to switch over to the fail-safe processing channel. The safety executive component can also be used to control multiple actuation channels in the system that each may have multiple channels.

The components of the pattern depicted in Figure 3 are described below:

• Input Data Source: This component represents the source of information that is used as input to the system under consideration. Typically, this data comes either from the system user or from external sensors that are used to monitor environmental variables such as: temperature, pressure, speed, light, etc...

• Actuator(s): Actuators are the physical devices that perform the action of the channel like: motors, switches, heaters, signals, or any other device that performs a specific action. Often, there are more than one actuator in a single channel.

• Actuation Channel: This channel represents a subsystem that performs dedicated tasks in the overall system by taking an input data from the input data source, performs some transformation on this data, and then uses the results to generate suitable commands to drive the actuators.

• Fail-Safe Processing Channel: This is an optional channel; it is dedicated to the execution and control of the fail-safe processing. In the presence of a fault in the actuation channel, the safety executive turns off the actuation channel, and the fail-safe channel takes over. If the System doesn't have a fail-Safe Channel, then the actuation channels must have at least one reachable safe states.

• Data Acquisition (Input Processing): This part of the channel collects the raw data from the input data source and may perform some data formatting or transfor- mations.

• Data Processing (Transformation): This part may contain multiple data transformation components; where each one performs a single transformation or processing on the received data to execute the desired algorithm in order to generate the required control signals. The final component of this part sends the computed output to the output processing unit.

• Output Processing: This unit takes the computed data from the data transformation unit and generates the final data and the control signals to drive the actuators. It can be considered as a device driver for the actuator.

• Integrity Check (Optional): This is an optional component that 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.

• Time Base: This is an independent timing source (timing circuit) that is used to drive the watchdog. This time source has to be separate from the one used to drive the actuation channel.

• Watchdog (WD): The watchdog receives liveness messages (strokes) from the components of the actuation channel in a predefined timeframe. If a stroke comes too late or out of sequence, the watchdog considers this situation as a fault in the actuation channel and it issues a shutdown signal to the actuation channel or initiates a

[4] Note: The *Watchdog Pattern* is based on two components, the actuation channel and a supervisor, called watchdog. The actuation channel typically triggers the watchdog at defined time intervals to demonstrate that the actuation channel is still active. More advanced approaches include more sophisticated interactions between the actuation channel and the watchdog to allow a higher degree of fault coverage (see e.g. [11]).

corrective action through sending a command signal to the optional integrity check. If the system contains multiple actuation channels, then it may contain multiple watchdogs, one per actuation channel.

• Safety Executive: This is the main component in this safety executive pattern. It tracks and coordinates all safety monitoring to ensure the execution of safety action when appropriate. It consists of a safety coordinator that controls safety measures and safety policies. The safety executive component captures the shutdown signal from the watchdog in the case of failure in the actuation channel.

• Safety Coordinator: The safety coordinator is used to control and coordinate the safety processing that is managed by the safety measures. It also executes the control algorithms that are specified by the safety policies.

• Safety Measures: Include the detailed description of the safety measures. The safety coordinator may control multiple safety measures.

• Safety Policies: Each safety policy specifies a strategy or control algorithm for the safety coordinator. It involves a complicated sequence of steps that involve multiple safety measures. The reason for the separation of the coordinator from the safety policies is to make the process of changing and adapting a safety policy easier.

Implication

This section describes the implication of this pattern relative to the basic system without a specific safety method.

• **Reliability**

Let us have the following notations:

R_{AC}: the reliability of the actuation channel. ($R_{old} = R_{AC}$)

R_{SC}: the reliability of the fail-safe processing channel.

R_{SE}: the reliability of the safety executive component.

C: the coverage factor which is defined as: the probability that a fault in an actuation channel will be identified by the safety executive and the fail-safe processing channel will be activated.

Assume that the watchdogs are carefully designed with reliability≈1.

The safety executive pattern will continue to work without system failure as long as one of the following two conditions holds:

◎ There is no fault in the *actuation channel*.

◎ There is a fault in the *actuation channel* and the watchdog detects this fault and the *safety executive* initiates a shutdown or activates the fail-safe processing channel.

The new reliability after using this pattern (R_{new}) is equal to:

$$R = R_{AC} + CR_{SE}(1 - R_{AC})R_{SC} \quad (2)$$

In this equation, the first term represents the reliability of the actuation channel while the second term represents the reliability of the remaining parts in the case of failure in the actuation channel.

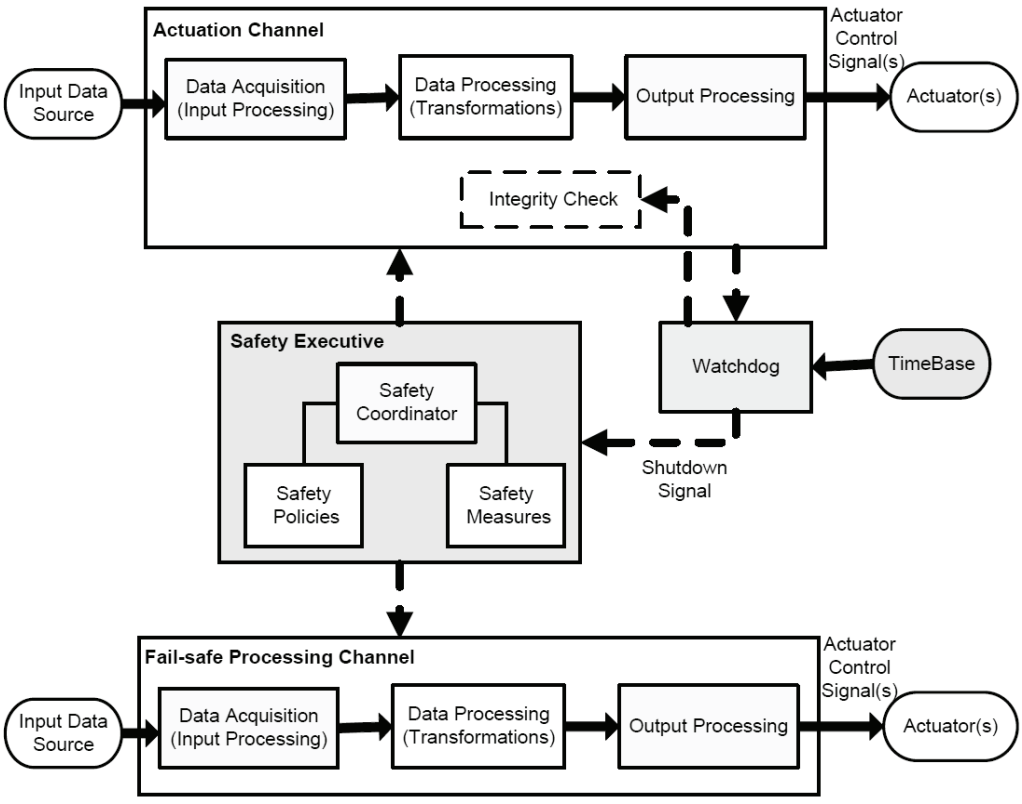

Figure 3. The safety executive pattern

The percentage improvement in reliability relative to the maximum possible improvement is equal to:

$$= \frac{R_{AC} + CR_{SE}(1 - R_{AC})R_{SC} - R_{AC}}{1 - R_{AC}} \times 100\% \quad (3)$$

$$= CR_{SE}R_{SC} \times 100\% \quad (4)$$

• Safety**:**

The safety executive pattern includes the following four design techniques: program sequence monitoring with a watchdog, test by redundant hardware (the watchdog that initiates the integrity check and BITs), safety bag techniques[5], and graceful degradation[6]. According to the standard IEC 61508 [3], the recommendation for these techniques is shown Table 1.

In general, we think that the combination of these techniques and the development cost makes the safety executive pattern suitable and highly recommended only for very high critical applications with high safety integrity levels (SIL4 and SIL3) and recommended for lower levels (SIL2 and SIL 1).

• **Cost:**

This pattern is an expensive pattern with very high cost since it consists of different components that involve high recurring and development cost.

○ Recurring cost: It includes the cost of the following:

◆ The actuation channel.

◆ The fail-safe processing channel (if present).

◆ The safety executive component.

◆ Watchdogs and their independent timing source.

○ Development cost: In general, the development cost for this pattern is very high since it includes a development of three different systems (channels) that include different architectures and different designs.

• **Modifiability:**

There are two types of possible modifications:

1) Actuation Channel: It is very simple to modify this pattern by adding extra functionality to the actuation channel. The only things that should be done: is to know whether the new components need to send stroke messages to the watchdog.

2) Safety policy: One of the main features of this pattern is the centralized safety processing which is performed by the Safety Executive Component. The Safety Executive separates the coordinator from the safety policies to simplify the change and modification of the safety policy and to make it easier.

[5]Note: A safety bag is an external monitor, implemented on an independent hardware component to ensure that the system performs safe actions [3].

[6]Note: The safety degradation is a technique that gives priorities to the various functions to be carried out by the system. The design ensures that if there are insufficient resources to carry out all the system functions, the higher priority functions (the safety functions in our case) are carried out in preference to the lower once [3].

[7]Note: a processing unit is a (programmable) electronic device executing a certain function. Typical examples are programmable logic controllers, microcontrollers, and FPGAs. Moreover, an electronic device might include more than one processing unit (e.g. multi-core architectures). In this case, the analysis of the independence between these units requires special care.

Table 1. Recommendations for safety integrity levels

Techniques	SIL1	SIL2	SIL3	SIL4
Program sequence monitoring (WD)	HR	HR	HR	HR
Test by redundant hardware	R	R	R	R
Safety bag techniques	—	R	R	R
Graceful degradation	R	R	HR	HR

• **Impact on execution time:**

The actuation channel and the safety executive have different CPUs and different memories, and they run simultaneously in parallel. Thus, there is no effect for the safety executive component on the actuation channel during the normal operation of the system except the execution of the periodic built in tests (BITs).

Implementation

The following points should be taken into consideration during the implementation of this pattern:

• The actuation channel, the safety executive, and the fail-safe processing channel run separately in parallel, therefore each channel will run on its own processing unit[7] and own memory.

• The safety-critical information must be protected against data corruption, e.g. by using CRCs or any other method to detect data errors.

• The watchdog component is simple and often implemented as a separate hardware device. It is capable of detecting a variety of hardware and software fault. However, its actual diagnostic coverage depends on the *i*ntegrity check implemented in the actuation channel.

• To provide protection from faults in a common time base, separate timing sources must be used for the watchdog, the safety executive and the actuation channel.

Consequences and Side Effects

The main drawback of this pattern is the high complexity of this pattern for implementation. Therefore it is used for complex and highly safety-critical systems.

Related Patterns

The safety executive pattern is used for complex safety-critical applications and it covers a large set of features, provided by of the other patterns, such as sequence monitoring provide by watchdog, switch-to-backup as in the fail-safe channel. For simpler systems with simpler safety requirements, other simpler patterns, such as Watchdog pattern, Sanity Check pattern and Monitor Actuator pattern [11], can be used.

5.2 EXAMPLE 2

This example pattern describes the pattern originally described in [22] including the standard pattern components as well as implications on non-functional requirements.

Pattern Name

Acceptance Voting Pattern

Other Names

Type
Software

Abstract
The Acceptance Voting pattern is a hybrid pattern that incorporates the *N-Version Programming Pattern* with *Acceptance Tests* (AT) used by the *Recovery Block Pattern*. Similar to the normal N-version programming approach, this pattern is based on two or more diverse software versions. Traditionally, these versions are functionally equivalent and are developed by independent teams from the same initial specification [23]. Moreover, approaches to increase the diversity of the resulting software versions could be applied already in the specification (functional diversity, see e.g. [40]). In case of this pattern, the output of each version is presented to an acceptance test to determine if the output is reasonable. The outputs that pass the acceptance test are used as inputs to a dynamic voter, which is executed to produce the correct output according to a specific voting scheme. Depending on the applications, the diverse software versions can be executed on parallel hardware or sequentially on a single hardware device.

Context
The application of this pattern is suitable in the following context:

• Tolerance of software faults is required for safety reasons (acceptance test)

• High reliability of the system's output is required (several software versions)

• The correctness of the results delivered by the diverse software versions can be checked by an acceptance test.

• The development of diverse software versions is possible (additional development costs, additional organizational effort, and sufficient number of developers for the development of each version).

Problem
Enable the tolerance of software faults that may remain after the software development to target safety and reliability requirements.

Pattern Structure
The Acceptance Voting Pattern (AV) is a hybrid pattern, which represents an extension of the N-version programming approach by incorporating this approach with the acceptance test used in the recovery block approach. This pattern includes N diverse software versions that are typically executed in parallel to perform the required task. The output of each version is tested for correctness using an acceptance test and only those results that pass the acceptance test are used by the voting algorithm to generate the final result. The goal of the Acceptance Voting pattern is to increase the system's reliability through a combination of a fault detection scheme provided by the acceptance test and a fault masking scheme provided by N-version programming with voting.

The structure of this pattern is shown in Figure 4 and the function of each component is described below:

• Input Data Source: This instance represents the source of information that is used as input to the designed system. Typically, this data comes either from a system user or from external sensors used to monitor environmental variables such as temperature, pressure, speed, or light.

• Output Data and Control Signals: The output data of the voter module represents the final output data of the designed system. This data may contain control signals to activate actuators as motors, switches, heaters, or messages for other components outside the system.

• Version 1, 2...N: These are diverse software versions implemented to fulfill the specified functionality. Typically, these versions result from an independent implementation by independent teams of software developers, based on the same initial specification. Thus, these versions are performing roughly the same functionality on the input data to produce the final result. Further aspects of generating these diverse software versions can be found in the *implementation section* below. Usually, these diverse versions are executed in parallel on different hardware devices to generate ***N*** outputs and each of these outputs is presented to an acceptance test to check them for correctness. Those results that pass the acceptance test are processed by the dynamic voter module to determine the output data by applying a specific voting strategy.

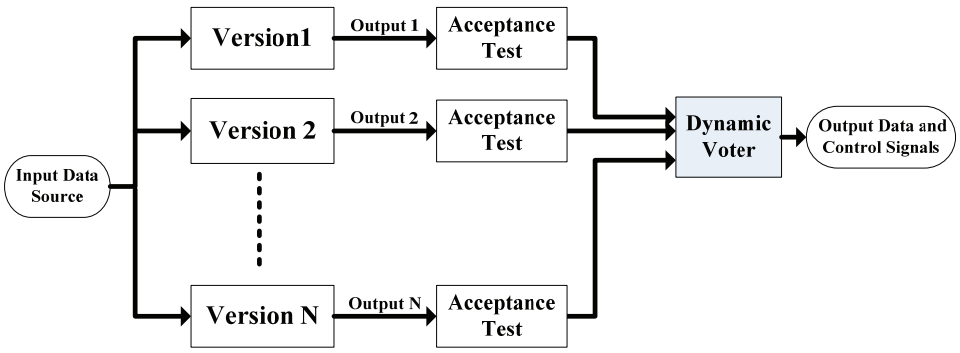

Figure 4. The acceptance voting pattern

• Acceptance Test (AT): This part of the software is executed on the outcome of each version to confirm that the result is reasonable and fulfills defined requirements given in the software specification. The acceptance test returns either true or false and may have several components. Moreover, it may include checks for runtime errors [24] and mechanisms for implicit error detection. Various implementations of the acceptance test are possible ranging from simple reasonableness checks to complex high-coverage validators [25].

The reliability of the system's output depends greatly on the quality of the acceptance test (especially an acceptance test that detects faults in a correct output reduces the system's reliability). Thus, this test should be carefully designed and it is desirable that the acceptance test is simple as well as easy to verify.

• Dynamic Voter: The voter reads the outputs that pass the acceptance test and uses these results as inputs to the voting algorithm in order to determine the final output and control signals. The voter that is used in this pattern should be dynamic[8] due to the variable number of inputs that ranges from 0 to N. Depending on the number of outputs that pass the acceptance test, the voter may include the following different actions:

– In the case when no output passes the acceptance test, it reports an overall system failure.

– In the case when one output passes the acceptance test, it just forwards this output.

– In the case when two outputs pass the acceptance test, the signal is only forwarded if both are equal (or the difference is within a defined tolerance). In the case of inequality, the action depends on the required level of safety and reliability, either an output is selected according to a predefined order or an exception is raised to indicate a failure.

– When the number of outputs that pass the acceptance test is more than two, a voting technique is executed to generate the final result.

Several voting techniques exist that can be used for voting as majority voting (most commonly used technique), consensus voting [26], and maximum likelihood voting [27]. The selection of these techniques depends on the type of data, the deviation in the outputs of the versions, the type of agreement [28], the output space cardinality size[9], the functionality of the voter [29], the reliability of the different versions, and perhaps even further factors.

[8]Note: A voter is considered as *dynamic* if it accepts varying numbers of input signals.

[9]Note: The cardinality size of an output space is the number of possible different values for an output.

[10]Note: While the assumption of failure independence is not realistic for practical software implementations, this assumption eases the calculation presented. The simplified calculation presented here already allows certain reliability evaluations. Moreover, a dependency term could be included into the calculations if an explicit consideration of dependencies between the software versions should be considered. Further aspects of software diversity can be found in the *implementation section* of this pattern description.

Implication

This section describes the implication of this pattern using a majority voting approach relative to the basic system without any specific safety method.

• **Reliability:**

Let us have the following notations:

f: the probability of failure in a version due to a bug in its implementation.

E: the event that the output of a version is erroneous. ($P\{E\} = f$)

T: the event that the acceptance test reports that the output is wrong.

N: the total number of different independent versions.

n: the number of versions that pass the acceptance test.

m: the agreement number which is equal to $\lceil (n+1)/2 \rceil$ for the majority voting.

P_{TP}: the probability that a version will pass the acceptance test, given that the outcome of a version is correct. (True Positive case) ($P_{TP} = P\{\overline{T} \mid \overline{E}\}$).

P_{FN}: the probability that a version will fail the acceptance test, given that the outcome of a version is correct. (False Negative case) ($P_{FN} = P\{T \mid \overline{E}\} = 1 - P_{TP}$).

P_{TN}: the probability that a version will fail the acceptance test, given that the outcome of a version is wrong. (True Negative case) ($P_{TN} = P\{T \mid E\}$).

P_{FP}: the probability that a version will pass the acceptance test, given that the outcome of a version is wrong. (False Positive case) ($P_{FP} = P\{\overline{T} \mid E\} = 1 - P_{TN}$).

R_{old}: the reliability of system software with single version ($R_{old} = R$).

R_{new}: the reliability of system software with acceptance voting pattern.

Assumptions:

– The voter is carefully designed and can be considered as fault free.

– The majority voting technique is used in the voter software component.

– The failures in the different versions are statically independent[10] and the different versions have the same probability of failure (f) and the same reliability ($R_i = R$). At any given time, if the number of versions' outputs that pass the acceptance test and participate the voting is n, then these outputs can be grouped into two parts:

– Correct Outputs (True Positive outputs that pass AT) with probability=$R*P_{TP}$.

– Incorrect Outputs(False Positive outputs that contain undetected faults) with probability = $(1-R)\cdot(1-P_{TN}) = (1-R)\cdot P_{FP}$.

The probability that an output passes the test is equal to:

$$P\{\overline{T}\} = RP_{TP} + (1-R)P_{FP} \quad (5)$$

The probability that an output does not pass the test is equal to:

$$p\{T\} = RP_{FN} + (1-R)P_{TN} \tag{6}$$

The Probability that the voter gives a correct output, given that n outputs passed the test, is equal to

$$= \sum_{i=m}^{n} \binom{n}{i} (RP_{TP})^{i} [(1-R)P_{FP}]^{n-i} \tag{7}$$

The probability that n outputs from the total number of outputs pass the acceptance test and give a correct result in the majority voting is:

$$= \left[\binom{N}{n} \left(\sum_{i=m}^{n} \binom{n}{i} (RP_{TP})^{i} [(1-R)P_{FP}]^{n-i} \right) (RP_{FN} + (1-R)P_{TN})^{N-n} \right] \tag{8}$$

The number of versions n that pass the acceptance to produce a correct result can be 1, 2…N. Therefore, the new reliability after using this pattern (R_{new}) is equal to:

$$R_{new} = \sum_{n=1}^{N} \left[\binom{N}{n} \left(\sum_{i=m}^{n} \binom{n}{i} (RP_{TP})^{i} [(1-R)P_{FP}]^{n-i} \right) (RP_{FN} + (1-R)P_{TN})^{N-n} \right] \tag{9}$$

Finally, the percentage improvement in software reliability relative to the maximum possible improvement is equal to:

$$= \frac{R_{new} - R_{old}}{1 - R_{old}} \times 100\% = \frac{R_{new} - R}{1 - R} \times 100\% \tag{10}$$

As shown in Equation (9) and (10), the reliability improvement in this pattern depends on the reliability and number of versions N, and on the performance and the effectiveness of the acceptance test used. The acceptance test should be carefully designed, reasonably simple, highly reliable, and with a high error detection coverage in order to mask the faulty outputs from participating in the voting step.

• **Safety:**

The presented pattern includes the concepts of diverse programming and fault detection with acceptance test and voting. According to the software requirements in the standard IEC 61508-3 [3], the recommendations for these techniques are shown in Table 2.

According to the last table, we think that this pattern is suitable and highly recommended only for very high critical applications with high safety integrity levels (SIL4 and SIL3), recommended for lower level (SIL2) and with no recommendation for SIL1.

Table 2. Recommendations for safety integrity levels

Techniques	SIL1	SIL2	SIL3	SIL4
Diverse programming	R	R	R	HR
Fault detection and diagnosis (Voting)	–	R	HR	HR
Fault detection and diagnosis (Acceptance Test)	–	R	HR	HR

• **Cost:**

In comparison to the basic system, this pattern is resulting in high additional costs. These costs can be divided into two parts.

– Recurring cost: includes the cost for the N different hardware units that are used for the parallel execution of the N-version software. So, the recurring cost will be N times (N*100%) comparing to the recurring cost for the basic system that includes a single version. In this pattern, the voter and the acceptance test are implemented in software. Therefore, this pattern includes additional recurring cost for the used memory.

– Development cost: The development of N diverse software versions will cost more than the development of single version software. An estimation of the development cost of N-version software has to consider the following aspects:

◆ The N versions have the same specification, and only one set of specification has to be developed [30].

◆ The cost for developing N versions prior to the verification and the validation phase is N times the cost for developing a single version [31].

◆ The management of an N-version project imposes overhead not found in traditional software development [30].

◆ The different versions can be used to validate each other [30]. While this approach could be used to decrease the cost for using verification and validation tools, it is not recommended as all versions implemented might contain a similar or even the same fault.

Exact information about the additional costs of creating N version instead of a single version is limited. The estimated practical cost of development of multiversion system showed that the costs increase sub-linearly with the number of components [32]. Moreover, it is stated in [33] that each additional version costs about 75-80% of a single version.

In addition to the previous costs, this pattern includes extra cost for developing and verifying an effective acceptance test.

• **Modifiability:**

The following possible modifications have to be considered:

1) Modification of a single version: It is possible to modify a single version either to remove a newly discovered fault or to improve a poorly programmed function [34]. In this case, the initial specification remains without any modification and the modification of this version is similar to the modification of single version software following a standard fault removal procedure.

2) Modification of all member versions: The reason for modification of all ***N*** versions is either to add a new functionality or to improve the overall performance of the N-version software [34]. In this case, the initial specification has to be modified and all ***N*** versions must be modified and tested independently by independent teams. In general, the modification of N-version software is re-

markably more difficult than the modification of single version software.

3) Modification of the acceptance test (AT): The acceptance test can be considered as an independent software module that is checking the output of each of the N versions. Thus, this acceptance test can be easily modified without any influence on the different versions.

4) Modification of the voter: The separation of the voting module from the N versions and the acceptance test allows easy modification or changes of the voting technique.

• **Impact on Execution Time:**

The diverse software versions in this pattern are executed in parallel, ideally on N independent hardware devices. As the execution times of these software versions might differ as they are implemented differently, the voter has to wait for the outputs of all software versions to be checked by the acceptance test before applying the voting algorithm. Thus, the total time of execution is determined by the slowest version in addition to the typically relatively small time to execute the acceptance test and the voting algorithm. In general, if we can neglect the execution time of the acceptance test and the voter, then the execution time of this pattern is slightly equal to single version software.

It is also possible to execute the independent versions followed by the acceptance test and voting algorithm sequentially on a single hardware. However, the time of execution will increase by N times of a single version. This disadvantage[11] makes the sequential execution less attractive, especially for time critical applications.

Implementation

The acceptance voting pattern is a hybrid pattern that combines the idea of N-version programming and fault detection using an acceptance test. Therefore, the success of this pattern depends on three factors:

1) The quality of the acceptance test is an important factor. Thus, the acceptance test should be carefully designed to detect most of the possible software faults.

2) The N diverse software versions and especially the level of diversity between these versions to avoid common failures between different versions. In order to increase the level of diversity and the independence of the designed versions, the following have been recommended in [30]:

• The use of a complete, correct, and carefully documented specification to prevent an error in the specification from propagating to the different versions.

• The use of independent and isolated teams of programmers with diversity in their training and experience.

• The use of diverse algorithms and diverse implementation techniques.

• The use of diverse programming languages.

• The use of diverse compilers, development tools, and test methods.

With respect to N-version programming, it has to be noted that experiments have shown that developers which implement the same function independently tend to make the same faults. For this reason, the assumption of statistically independent failure behavior of the software versions does not hold [35,36]. Approaches modeling this dependency structure (e.g. [37]) and corresponding empirical studies [38,39] are known which allow certain (model-based) predictions of failure probabilities in systems built on N-version programming. The measures presented above in this section try to decrease the dependencies between the different software versions. The assumption is that different development methodologies lead to diversity in decision and thus diversity in the behavior of the resulting software. However, even with this increased effort, the absence of undesired dependencies between the diverse software versions cannot be guaranteed [36,40,41]. For this reason, it is recommended to apply N-version programming in combination with further software fault tolerance measures as the acceptance tests applied in this pattern.

3) The use of a suitable voting technique to implement the voting component such as:

• Majority Voting: It is the simplest and most common used method that is used to find the output, where at least $\lceil (n+1)/2 \rceil$ variant results agree.

• Plurality Voting (PV): It is a simple voter, that implements m-out-of-n voting, where m is less than a strict majority.

• Consensus Voting (CV) [26]: This voting method is used for multiversion software with a small output space. In this method, the result of the largest agreement number is chosen as correct output.

• Maximum Likelihood Voting (MLV) [27]: In this method, the voter uses the reliability of each software version to make a more accurate estimation of the most likely correct result.

• Adaptive Voting [42]: This technique introduces an individual weighting factor to each version which is later included in the voting procedure. These weighting factors are dynamically changeable to model and manage different quality levels of versions.

Consequences and Side Effects

Similar to the original N-version programming approach, the drawbacks of the *Acceptance Voting Pattern* are seen in the effort of developing N diverse software versions in addition to the high dependency on the initial specification which may propagate faults to all versions. With respect to safety, the problem of dependent faults in all N software versions is less critical in this pattern than in the original N-version programming approach, as the acceptance test included represents an additional measure to detect these faults.

[11]Note: Another disadvantage is that the execution on a single hardware can tolerate only few hardware faults (certain transient faults) while the approach on N different hardware devices can tolerate most transient and permanent hardware faults. However, even in this case faults have to be considered that could affect all N versions simultaneously.

Related Patterns
In comparison to the basic system, the Acceptance Voting Pattern allows improvements in the reliability and the safety of a software based system. As it is executed on different hardware devices, it is possible to combine this pattern with the Heterogeneous Design Pattern for the design of these diverse hardware units to deal with systematic hardware faults. This combination will improve the reliability and safety of the hardware as well as the software.

6. Conclusions

The design of safety-critical embedded applications requires an integration of the commonly used software and hardware design methods. Therefore, the use of design pattern is very promising in this application domain, if the specific properties of embedded systems are considered in the pattern representation. In this paper, we proposed an extended pattern representation for the design of safety-critical embedded applications. This representation focuses on the implications and side effects of the represented design method on the non-functional requirements of the safety-critical embedded system including safety, reliability, modifiability, cost and execution time. Two example patterns have been used to show the effectiveness of the proposed pattern representation. We expect that this extended representation will guide the selection of a suitable design as it allows evaluating alternative patterns with respect to their implications.

7. Future Work

For a successful application of the proposed representation of design patterns for safety-critical embedded systems, an integration of a higher number of design patterns is desirable. For this reason, we currently construct a pattern catalogue based on the proposed representation by collecting and classifying commonly used hardware and software design methods. Moreover, it is intended to construct the catalogue such that an automatic recommendation of suitable design methods for a given application can be achieved in the future.

8. Acknowledgments

This work was supported by the German Academic Exchange Service (DAAD) under the program: Research Grants for Doctoral Candidates and Young Academics and Scientists.

REFERENCES

[1] C. Alexander, "A Pattern Language: Towns, Buildings, Construction," New York: Oxford University Press, 1977.
[2] E. Gama, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: Element of reusable object-oriented software," New York: Addison-Wesley, 1997.
[3] IEC61508 Functional safety for electrical/electronic/ programmable electronic safety-related systems, International Electrotechnical Commission, 1998.
[4] A. Armoush, F. Salewski, and S. Kowalewski, "Effective pattern representation for safety critical embedded systems," International Conference on Computer Science and Software Engineering (CSSE 2008), pp. 91−97, 2008.
[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal "Pattern-oriented software architecture: A system of patterns," John Wiley & Sons, Inc., New York, NY, 1996.
[6] P. Coad, "Object-oriented patterns," Communications of the ACM, Vol. 35, pp. 152−159, 1992.
[7] K. Beck and W. Cunningham, "Using pattern languages for object-oriented programs," Presented at the OOPSLA-87 Workshop on Specification and Design for Object-Oriented Programming.
[8] J. Coplien, "Idioms and patterns as architectural literature," IEEE Software, Vol. 14, pp. 36−42, 1997.
[9] B. Appleton. "Patterns and software: Essential concept and terminology," available at <http://www.enteract.com /~bradapp/docs/patterns-intro.html>.
[10] B. P. Douglass, "Doing hard time: Developing real-time system with UML, objects, frameworks, and pattern," New York: Addison-Wesley, 1999.
[11] B. P. Douglass, "Real-time design patterns," New York: Addison-Wesley, 2003.
[12] D. Gross and E. Yu, "From non-functional requirements to design through patterns," Requirements Engineering, Vol. 6, No. 1, pp. 18−36, 2002.
[13] J. Cleland-Huang and D. Schmelzer, "Dynamically tracing non-functional requirements through design pattern invariants," Workshop on Traceability in Emerging Forms of Software Engineering, in conjunction with IEEE International Conference on Automated Software Engineering, 2003.
[14] J. Fletcher and J. Cleland-Huang, "Softgoal traceability patterns," in Proceedings of the 17th IEEE International Symposium on Software Reliability Engineering (ISSRE 2006), pp. 363−374, 2006.
[15] L. Xu, H. Ziv, T. A. Alspaugh, and D. J. Richardson, "An architectural pattern for non-functional dependability requirements," Journal of Systems and Software, Vol. 79, No. 10, pp. 1370−1378, 2006.
[16] S. Konrad and B. Cheng, "Requirements patterns for embedded systems," in Proceedings of the IEEE Joint International Requirements Engineering Conference (RE'02), pp. 127−136, 2002.
[17] S. Konrad, B. Cheng, and L. Campbell, "Object analysis patterns for embedded systems," IEEE Transactions on Software Engineering, Vol. 30, No. 12, pp. 970−992, 2004.
[18] K. Wolf and C. Liu, "New clients with old servers" A Pattern Language for Client/Server Frameworks," in Pattern Languages of Program Design, J. Coplien and D. Schmidt, Eds. Reading, MA: Addison Wesley, pp. 55−64, 1955.
[19] D. Riehle and H. Züllighoven, "A pattern language for tool construction and integration based on the tools and materials metaphor," in Pattern Languages of Program Design, J. Coplien and D. Schmidt, Eds. Reading, MA: Addison Wesley, pp. 55−64, 1955.
[20] S. Adams, "Functionality ala carte," in Pattern Languages of Program Design, J. Coplien and D. Schmidt, Eds. Reading, MA: Addison Wesley, pp. 55−64, 1955.
[21] R. Lajoie and R. K. Keller, "Design and reuse in object-oriented frameworks: Patterns, contracts and motifs in concert," in Object-Oriented Technology for Database and Software Systems, V. Alagar and R, Missaoui, Eds.

Singapore: World Scientific Publishing, pp. 295−312, 1995.
[22] A. Athavale, "Performance evaluation of hybrid voting schemes," M. S. thesis, North Carolina State University, Department of Computer Science, 1989.
[23] A. Avizienis and L. Chen, "On the implementation of N-version programming for software fault tolerance during execution," in Proceedings of IEEE COMPSAC 77, pp. 149−155, 1977.
[24] N. Storey, "Safety-Critical Computer Systems," Boston: Addison-Wesley, 1996.
[25] B. Prahami, "Design of reliable software via general combination of N-Version Programming and Acceptance Testing," in Proceedings of 7th International Symposium on Software Reliability Engineering ISSRE'96, pp. 104−109, 1996.
[26] D. F. McAllister, C. E. Sun, and M. A. Vouk, "Reliability of voting in fault-tolerant software systems for small output spaces," IEEE Transactions on Reliability, Vol. 39, No. 5, pp. 524−534, 1990.
[27] Y. W. Leung, "Maximum likelihood voting for fault-tolerant software with finite output space," IEEE Transactions on Reliability, Vol. 44, No. 3, 1995.
[28] G. Latif-Shabgahi, J. M. Bass, and S. Bennett, "A taxonomy for software voting algorithms used in safety-critical systems," IEEE Transactions, Reliability, Vol. 53, No. 3, pp. 319−328, 2004.
[29] B. Parhami, "Voting algorithms," IEEE Transactions on Reliability, Vol. 43, pp. 617−629, 1994.
[30] I. Koren and C. M. Krishna, "Fault-tolerant systems," Elsevier, 2007.
[31] A. Avizienis, "The N-version approach to fault-tolerant software," IEEE Transactions on Software Engineering, Vol. 11, No. 12, pp. 1491−1501, 1985.
[32] F. Daniels, K. Kim and M. A. Vouk, "The reliable hybrid pattern: a generalized software fault tolerant design pattern," in Conference PloP'97, pp. 1−9, 1997.
[33] M. Lyu, "Handbook of software reliability engineering," New York: McGraw-Hill and IEEE Computer Society Press, 1996.
[34] A. Avizienis, "The methodology of N-version programming," in Software Fault Tolerance, M. Lyu, Ed. New York: Wiley, pp. 23−46, 1995.
[35] J. C. Knight and N. G. Leveson, "An experimental evaluation of the assumption of independence in multiversion programming," IEEE Transactions on Software Engineering, Vol. 12, pp. 96−109, 1986.
[36] F. Salewski, D. Wilking, and S. Kowalewski, "The effect of diverse hardware platforms on n-version programming in embedded systems-an empirical evaluation," in 3rd International Workshop on Dependable Embedded Systems (WDES'06), 2006.
[37] B. Littlewood and D. R. Miller, "Conceptual modeling of coincident failures in multiversion software," IEEE Transactions on Software Engineering, 1989.
[38] J. G. W. Bentley, P. G. Bishop, and M. J. P. van der Meulen, "An empirical exploration of the difficulty function," in Computer Safety, Reliability and Security (Safecomp), 2004.
[39] X. Cai and M. R. Lyu, "An empirical study on reliability modeling for diverse software systems," 15th International Symposium on Software Reliability Engineering (ISSRE), 2004.
[40] B. Littlewood, P. Popov and L. Strigini, "A note on modeling functional diversity," in Reliability Engineering and System Safety, 1999.
[41] F. Salewski and S. Kowalewski, "Achieving highly reliable embedded software: An empirical evaluation of different approaches," in Proceeding of 26th International Conference on Computer Safety, Reliability and Security (SAFECOMP'07), pp. 270−275, 2007.
[42] K. Kanoun, M. Kaaniche, C. Beounes, J. C. Laprie, and J. Arlat, "Reliability growth of fault tolerant software," IEEE Transactions on Reliability, Vol. 42, No. 2, 1993.

