Safely Embedded Software for State Machines in Automotive Applications

Juergen Mottok[1], Frank Schiller[2] and Thomas Zeitler[3]
[1]*Regensburg University of Applied Sciences*
[2]*Beckhoff Automation GmbH*
[3]*Continental Automotive GmbH*
Germany

1. Introduction

Currently, both fail safe and fail operational architectures are based on hardware redundancy in automotive embedded systems. In contrast to this approach, safety is either a result of diverse software channels or of one channel of specifically coded software within the framework of Safely Embedded Software. Product costs are reduced and flexibility is increased. The overall concept is inspired by the well-known Vital Coded Processor approach. There the transformation of variables constitutes an (AN+B)-code with prime factor A and offset B, where B contains a static signature for each variable and a dynamic signature for each program cycle. Operations are transformed accordingly.

Mealy state machines are frequently used in embedded automotive systems. The given Safely Embedded Software approach generates the safety of the overall system in the level of the application software, is realized in the high level programming language C, and is evaluated for Mealy state machines with acceptable overhead. An outline of the comprehensive safety architecture is given.

The importance of the non-functional requirement safety is more and more recognized in the automotive industry and therewith in the automotive embedded systems area. There are two safety categories to be distinguished in automotive systems:

- The goal of *active safety* is to prevent accidents. Typical examples are Electronic Stability Control (ESC), Lane Departure Warning System (LDWS), Adaptive Cruise Control (ACC), and Anti-lock Braking System (ABS).
- If an accident cannot be prevented, measures of *passive safety* will react. They act jointly in order to minimize human damage. For instance, the collaboration of safety means such as front, side, curtain, and knee airbags reduce the risk tremendously.

Each safety system is usually controlled by the so called Electronic Control Unit (ECU). In contrast to functions without a relation to safety, the execution of safety-related functions on an ECU-like device necessitates additional considerations and efforts.

The normative regulations of the generic industrial safety standard IEC 61508 (IEC61508, 1998) can be applied to automotive safety functions as well. Independently of its official present and future status in automotive industry, it provides helpful advice for design and development.

In the future, the automotive safety standard ISO/WD 26262 will be available. In general, based on the safety standards, a hazard and risk graph analysis (cf. e. g. (Braband, 2005)) of a given system determines the safety integrity level of the considered system functions. The detailed safety analysis is supported by tools and graphical representations as in the domain of Fault Tree Analysis (FTA) (Meyna, 2003) and Failure Modes, Effects, and Diagnosis Analysis (FMEDA) (Boersoek, 2007; Meyna, 2003).

The required hardware and software architectures depend on the required safety integrity level. At present, safety systems are mainly realized by means of hardware redundant elements in automotive embedded systems (Schaueffele, 2004).

In this chapter, the concept of Safely Embedded Software (SES) is proposed. This concept is capable to reduce redundancy in hardware by adding diverse redundancy in software, i.e. by specific coding of data and instructions. Safely Embedded Software enables the proof of safety properties and fulfills the condition of single fault detection (Douglass, 2011; Ehrenberger, 2002). The specific coding avoids non-detectable common-cause failures in the software components. Safely Embedded Software does not restrict capabilities but can supplement multi-version software fault tolerance techniques (Torres-Pomales, 2000) like N version programming, consensus recovery block techniques, or N self-checking programming. The new contribution of the Safely Embedded Software approaches the constitution of safety in the layer of application software, that it is realized in the high level programming language C and that it is evaluated for Mealy state machines with acceptable overhead.

In a recently published generic safety architecture approach for automotive embedded systems (Mottok, 2006), safety-critical and safety-related software components are encapsulated in the application software layer. There the overall open system architecture consists of an application software, a middleware referred to as Runtime-Environment, a basic software, and an operating system according to e. g. AUTOSAR (AUTOSAR, 2011; Tarabbia, 2005). A safety certification of the safety-critical and the safety-related components based on the Safely Embedded Software approach is possible independently of the type of underlying layers. Therefore, a sufficiently safe fault detection for data and operations is necessary in this layer. It is efficiently realized by means of Safely Embedded Software, developed by the authors.

The chapter is organized as follows: An overview of related work is described in Section 2. In Section 3, the Safely Embedded Software Approach is explained. Coding of data, arithmetic operations and logical operations is derived and presented. Safety code weaving applies these coding techniques in the high level programming language C as described in Section 4. A case study with a *Simplified Sensor Actuator State Machine* is discussed in Section 5. Conclusions and statements about necessary future work are given in Section 6.

2. Related work

In 1989, the Vital Coded Processor (Forin, 1989) was published as an approach to design typically used operators and to process and compute vital data with non-redundant hardware and software. One of the first realizations of this technique has been applied to trains for the metro A line in Paris. The Vital technique proposes a data mapping transformation also referred to in this chapter. The Vital transformation for generating diverse coded data x_c can be roughly described by multiplication of a date x_f with a prime factor A such that $x_c = A * x_f$ holds. The prime A determines the error detection probability, or residual error probability, respectively, of the system. Furthermore, an additive modification by a static signature for each variable B_x and a dynamic signature for each program cycle D lead finally to the code of the type $x_c = A * x_f + B_x + D$. The hardware consists of a single microprocessor, the so called Coded Monoprocessor, an additional dynamic controller, and a logical input/output interface. The dynamic controller includes a clock generator and a comparator function. Further on, a logical output interface is connected to the microprocessor and the dynamic controller. In particular, the Vital Coded Processor approach cannot be handled as standard embedded hardware and the comparator function is separated from the microprocessor in the dynamic controller.

The ED^{4}I approach (Oh, 2002) applies a commercial off-the-shelf processor. Error detection by means of diverse data and duplicated instructions is based on the SIHFT technique that detects both temporary and permanent faults by executing two programs with the same functionality but different data sets and comparing their outputs. An original program is transformed into a new program. The transformation consists of a multiplication of all variables and constants by a diversity factor k. The two programs use different parts of the underlying hardware and propagate faults in different ways. The fault detection probability was examined to determine an adequate multiplier value k. A technique for adding commands to check the correct execution of the logical program flow has been published in (Rebaudengo, 2003). These treated program flow faults occur when a processor fetches and executes an incorrect instruction during the program execution. The effectiveness of the proposed approach is assessed by several fault injection sessions for different example algorithms.

Different classical software fail safe techniques in automotive applications are, amongst others, program flow monitoring methods that are discussed in a survey paper (Leaphart, 2005).

A demonstration of a fail safe electronic accelerator safety concept of electronic control units for automotive engine control can be found in (Schaueffele, 2004). The electronic accelerator concept is a three-level safety architecture with classical fail safe techniques and asymmetric hardware redundancy.

Currently, research is done on the Safely Embedded Software approach. Further results were published in (Mottok, 2007; Steindl, 2009;?; Mottok, 2009; Steindl, 2010; Raab, 2011; Laumer, 2011). Contemporaneous Software Encoded Processing was published (Wappler, 2007). This approach is based on the Vital transformation. In contrast to the Safely Embedded Software approach it provides the execution of arbitrary programs given as binaries on commodity hardware.

3. The safely embedded software approach

3.1 Overview

Safely Embedded Software (SES) can establish safety independently of a specific processing unit or memory. It is possible to detect permanent errors, e. g. errors in the Arithmetic Logical Unit (ALU) as well as temporary errors, e. g. bit-flips and their impact on data and control flow. SES runs on the application software layer as depicted in Fig. 1. Several application tasks have to be safeguarded like e. g. the evaluation of diagnosis data and the check of the data from the sensors. Because of the underlying principles, SES is independent not only of the hardware but also of the operating system.

Fig. 2 shows the method of Safety Code Weaving as a basic principle of SES. Safety Code Weaving is the procedure of adding a second software channel to an existing software channel.

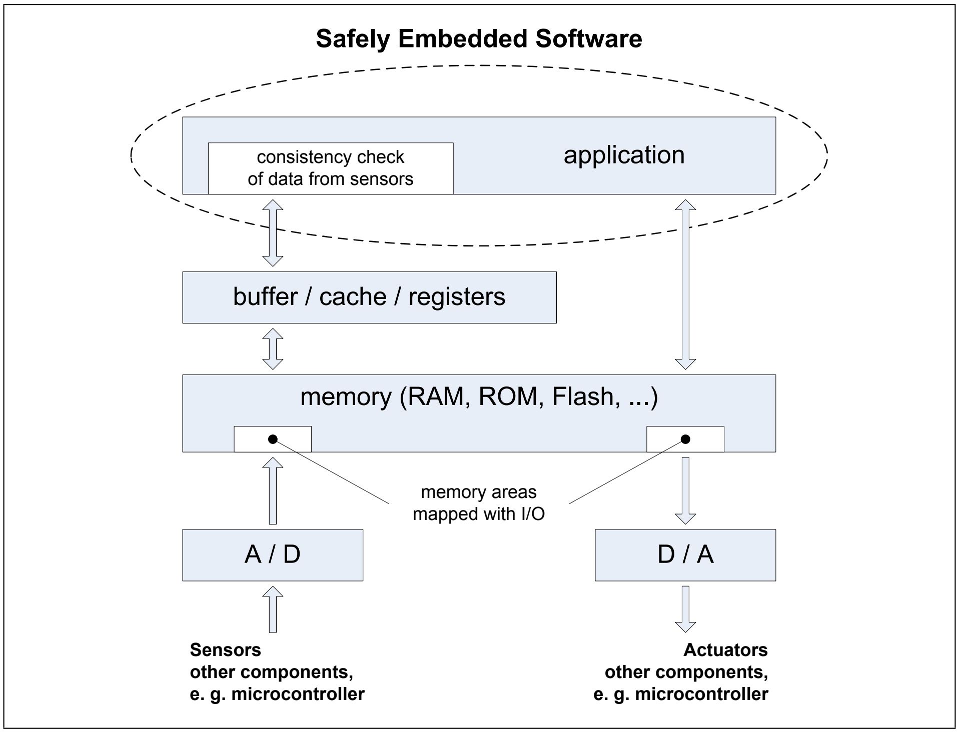

Safely Embedded Software
consistency check of data from sensors
application
buffer / cache / registers
memory (RAM, ROM, Flash, ...)
memory areas mapped with I/O
A / D
D / A
Sensors other components, e. g. microcontroller
Actuators other components, e. g. microcontroller

Fig. 1. The Safely Embedded Software approach.

In this way, SES adds a second channel of the transformed domain to the software channel of the original domain. In dedicated nodes of the control flow graph, comparator functionality is added. Though, the second channel comprises diverse data, diverse instructions, comparator and monitoring functionality. The comparator or voter, respectively, on the same ECU has to be safeguarded with voter diversity (Ehrenberger, 2002) or other additional diverse checks.

It is not possible to detect errors of software specification, software design, and software implementation by SES. Normally, this kind of errors has to be detected with software quality assurance methods in the software development process. Alternatively, software fault tolerance techniques (Torres-Pomales, 2000) like N version programming can be used with SES to detect software design errors during system runtime.

As mentioned above, SES is also a programming language independent approach. Its implementation is possible in assembler language as well as in an intermediate or a high programming language like C. When using an intermediate or higher implementation language, the compiler has to be used without code optimization. A code review has to assure, that neither a compiler code optimization nor removal of diverse instructions happened. Basically, the certification process is based on the assembler program or a similar machine language.

Since programming language C is the de facto implementation language in automotive industry, the C programming language is used in this study exclusively. C code quality can be

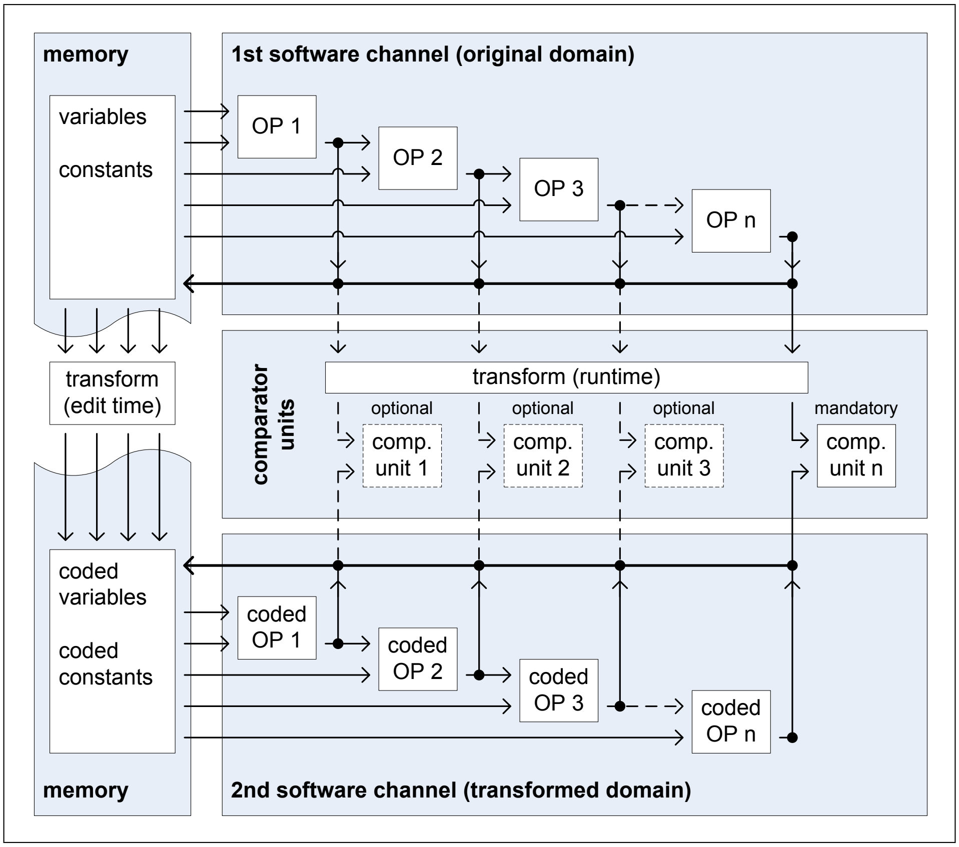


Fig. 2. Safety Code Weaving.

assured by application of e. g. the MISRA-2 (MISRA, 2004). A safety argument for dedicated deviation from MISRA-2 rules can be justified.

3.2 Detectable faults by means of safely embedded software

In this section, the kind of faults detectable by means of Safely Embedded Software is discussed. For this reason, the instruction layer model of a generalized computer architecture is presented in Fig. 3. Bit flips in different memory areas and in the central processing unit can be identified.

Table 1 illustrates the Failure Modes, Effects, and Diagnosis Analysis (FMEDA). Different faults are enumerated and the SES strategy for fault detection is related.

In Fig. 2 and in Table 1, the SES comparator function is introduced. There are two alternatives for the location of the SES comparator. If a local comparator is used on the same ECU, the comparator itself has also to be safeguarded. If an additional comparator on a remote receiving ECU is applied, hardware redundancy is used implicitely, but the inter-ECU communication has to be safeguarded by a safety protocol (Mottok, 2006). In a later system

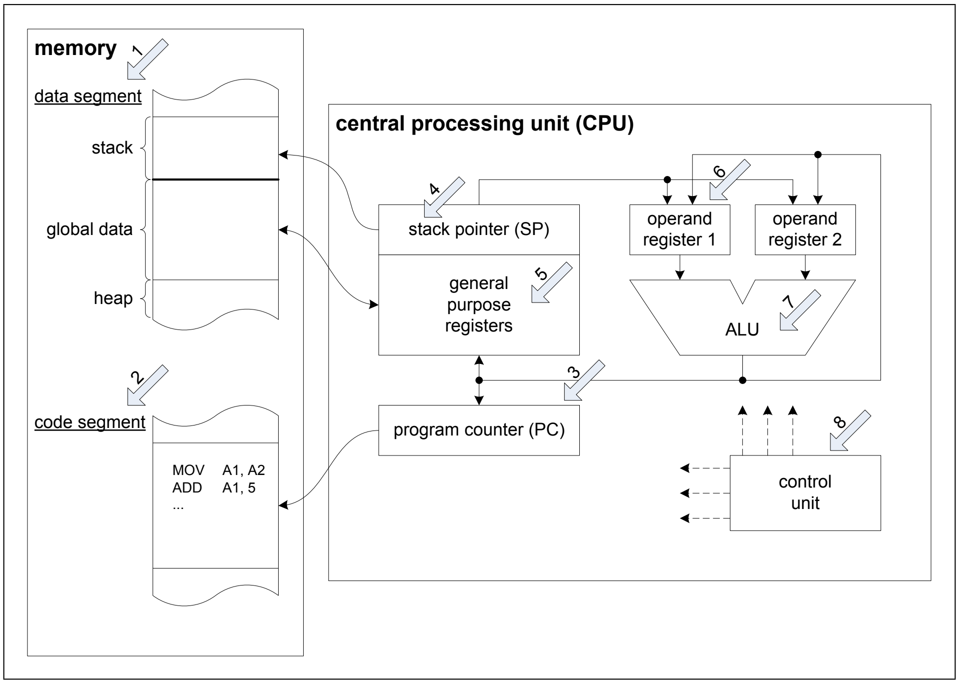


Fig. 3. Model of a generalized computer architecture (instruction layer). The potential occurrence of faults are marked with a label.

FMEDA, the appropriate fault reaction has to be added, regarding that SES is working on the application software layer.

The fault reaction on the application software layer depends on the functional and physical constraints of the considered automotive system. There are various options to select a fault reaction. For instance, fault recovery strategies, achieving degraded modes, shut off paths in the case of fail-safe systems, or the activation of cold redundancy in the case of fail-operational architectures are possible.

3.3 Coding of data

Safely Embedded Software is based on the (AN+B)-code of the Coded Monoprocessor (Forin, 1989) transformation of original integer data x_f into diverse coded data x_c. Coded data are data fulfilling the following relation:

$$x_c = A * x_f + B_x + D \quad where \quad x_c, x_f \in \mathbb{Z},\ A \in \mathbb{N}^+,\ B_x, D \in \mathbb{N}_0,\ and \quad B_x + D < A. \qquad (1)$$

The duplication of original instructions and data is the simplest approach to achieve a redundant channel. Obviously, common cause failures cannot be detected as they appear in both channels. Data are used in the same way and identical erroneous results could be produced. In this case, fault detection with a comparator is not sufficient.

label	area of action	fault	error	detection
1	stack, global data and heap	bitflip	incorrect data incorrect address	SES comparator SES logical program flow monitoring
2	code segment	bitflip	incorrect operator (but right PC)	SES comparator SES logical program flow monitoring
3	program counter	bitflip	jump to incorrect instruction in the code	SES logical program flow monitoring
4	stack pointer	bitflip	incorrect data incorrect address	SES comparator SES logical program flow monitoring
5	general purpose registers	bitflip	incorrect data incorrect address	SES comparator SES logical program flow monitoring
6	operand register	bitflip	incorrect data	SES comparator
7	ALU	bitflip	incorrect operator	SES comparator
8	control unit		incorrect data incorrect operator	SES comparator SES logical program flow monitoring

Table 1. Faults, errors, and their detection ordered by their area of action. (The labels correspond with the numbers presented in Fig. 3.)

The prime number A (Forin, 1989; Ozello, 1992) determines important safety characteristics like Hamming Distance and residual error probability $P = 1/A$ of the code. Number A has to be prime because in case of a sequence of i faulty operations with constant offset f, the final offset will be $i * f$. This offset is a multiple of a prime number A if and only if i or f is divisible by A. If A is not a prime number then several factors of i and f may cause multiples of A. The same holds for the multiplication of two faulty operands. Additionally, so called deterministic criteria like the above mentioned Hamming distance and the arithmetic distance verify the choice of a prime number.

Other functional characteristics like necessary bit field size etc. and the handling of overflow are also caused by the value of A. The simple transformation $x_c = A * x_f$ is illustrated in Fig. 4.

The static signature B_x ensures the correct memory addresses of variables by using the memory address of the variable or any other variable specific number. The dynamic signature D ensures that the variable is used in the correct task cycle. The determination of the dynamic signature depends on the used scheduling scheme (see Fig. 6). It can be calculated by a clocked counter or it is offered directly by the task scheduler.

The instructions are coded in that way that at the end of each cycle, i. e. before the output starts, either a comparator verifies the diverse channel results $z_c = A * z_f + B_z + D$?, or the coded channel is checked directly by the verification condition $(z_c - B_z - D) \bmod A = 0$? (cf. Equation 1).

In general, there are two alternatives for the representation of original and coded data. The first alternative is to use completely unconnected variables for original data and the coded ones. The second alternative uses a connected but separable code as shown in Fig. 5. In the

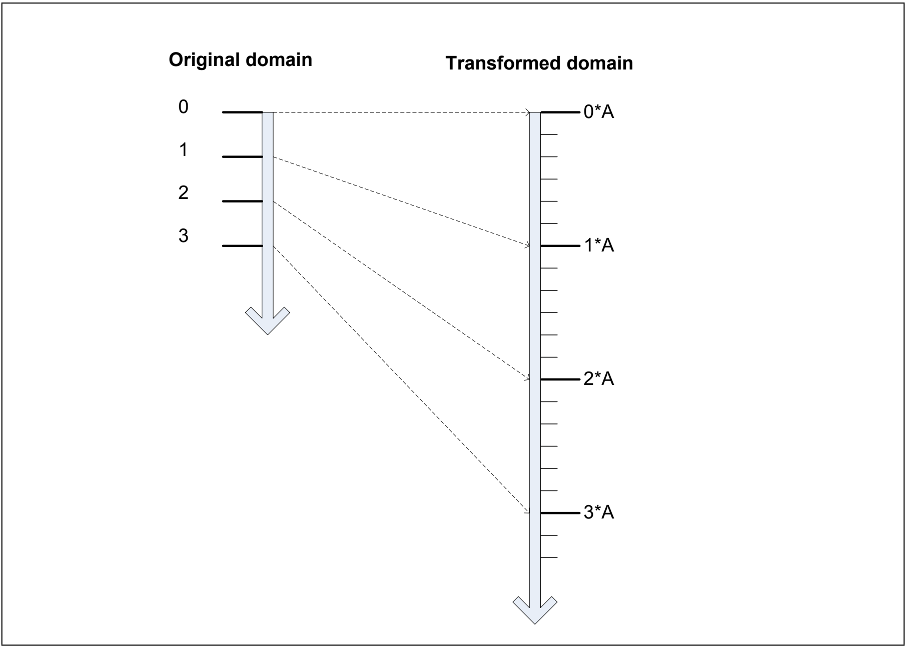


Fig. 4. Simple coding $x_c = A * x_f$ from the original into the transformation domain.

separable code, the transformed value x_c contains the original value x_f. Obviously, x_f can be read out easily from x_c.

The coding operation for separable code is introduced in (Forin, 1989):

Separable coded data are data fulfilling the following relation:

$$x_c = 2^k * x_f + (-2^k * x_f) \text{ modulo } A + B_x + D \tag{2}$$

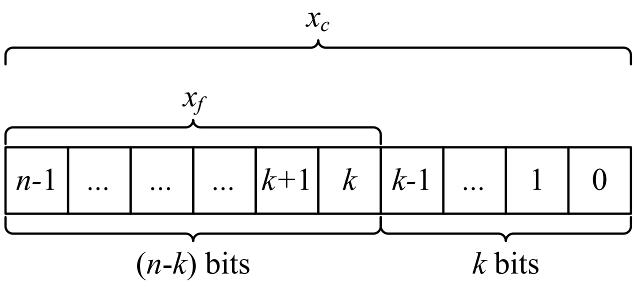
The factor 2^k causes a dedicated k-times right shift in the n-bit field. Therefore, one variable can be used for representing original data x_f and coded data x_c.

Without loss of generality, independent variables for original data x_f and coded data x_c are used in this study.

In automotive embedded systems, a hybrid scheduling architecture is commonly used, where interrupts, preemptive tasks, and cooperative tasks coexist, e. g. in engine control units on base of the OSEK operating system. Jitters in the task cycle have to be expected. An inclusion of the dynamic signature into the check will ensure that used data values are those of the current task cycle.

Measures for logical program flow and temporal control flow are added into the SES approach.

One goal is to avoid the relatively high probability that two instruction channels using the original data x_f and produce same output for the same hardware fault. When using the transformation, the corresponding residual error probability is basically given by the



$$x_c = 2^k * x_f + (-2^k * x_f) \bmod A + B_x + D$$

conditions:

$$2^k > (A - 1) + B_x + D,\ B_x + D < A$$

Fig. 5. Separable code and conditions for its application.

reciprocal of the prime multiplier, A^{-1}. The value of A determines the safe failure fraction (SFF) in this way and finally the safety integrity level of the overall safety-related system (IEC61508, 1998).

3.4 Coding of operations

A complete set of arithmetic and logical operators in the transformed domain can be derived. The transformation in Equation (1) is used. The coding of addition follows (Forin, 1989) whereas the coding of the Greater or Equal Zero operator has been developed within the Safely Embedded Software approach.

A coded operator OP$_\mathrm{c}$ is an operator in the transformed domain that corresponds to an operator OP in the original domain. Its application to uncoded values provides coded values as results that are equal to those received by transforming the result from the original domain after the application OP for the original values. The formalism is defined, such that the following statement is correct for all x_f, y_f from the original domain and all x_c, y_c from the transformed domain, where $x_c = \sigma(x_f)$ and $y_c = \sigma(y_f)$ is valid:

$$\begin{aligned} x_f &\;\circ\!\!-\!\!\bullet\; x_c \\ y_f &\;\circ\!\!-\!\!\bullet\; y_c \\ z_f &\;\circ\!\!-\!\!\bullet\; z_c \\ z_f = x_f\,\mathrm{OP}\,y_f &\;\circ\!\!-\!\!\bullet\; x_c\,\mathrm{OP_c}\,y_c = z_c \end{aligned} \tag{3}$$

Accordingly, the unary operators are noted as:

$$z_f = \mathrm{OP}\, y_f \quad \circ\!\!-\!\!\bullet \quad \mathrm{OP_c}\, y_c = z_c \tag{4}$$

In the following, the derivation steps for the addition operation and some logical operations in the transformed domain are explained.

3.4.1 Coding of addition

The addition is the simplest operation of the four basic arithmetic operations. Defining a coded operator (see Equation (3)), the coded operation $\oplus$ is formalized as follows:

$$z_f = x_f + y_f \qquad \Rightarrow \qquad z_c = x_c \oplus y_c \tag{5}$$

Starting with the addition in the original domain and applying the formula for the inverse transformation, the following equation can be obtained for z_c:

$$\begin{aligned}
z_f &= x_f + y_f \\
\frac{z_c - B_z - D}{A} &= \frac{x_c - B_x - D}{A} + \frac{y_c - B_y - D}{A} \\
z_c - B_z - D &= x_c - B_x - D + y_c - B_y - D \\
z_c &= x_c - B_x - D + y_c - B_y + B_z \\
z_c &= x_c + y_c + \underbrace{(B_z - B_x - B_y)}_{const.} - D
\end{aligned} \tag{6}$$

The Equations (5) and (6) state two different representations of z_c. A comparison leads immediately to the definition of the coded addition $\oplus$:

$$z_c = x_c \oplus y_c \; = \; x_c + y_c + (B_z - B_x - B_y) - D \tag{7}$$

3.4.2 Coding of comparison: Greater or equal zero

The coded (unary) operator $\mathrm{geqz_c}$ (greater or equal zero) is applied to a coded value x_c. $\mathrm{geqz_c}$ returns $\mathrm{TRUE_c}$, if the corresponding original value x_f is greater than or equal to zero. It returns $\mathrm{FALSE_c}$, if the corresponding original value x_f is less than zero. (This corresponds to the definition of a coded operator (see Definition 3) and the definition of the ≥ 0 operator of the original domain.)

$$\mathrm{geqz_c}(x_c) = \begin{cases} \mathrm{TRUE_c}, & \text{if } x_f \geq 0, \\ \mathrm{FALSE_c}, & \text{if } x_f < 0. \end{cases} \tag{8}$$

Before deriving the transformation steps of the coded operator $\mathrm{geqz_c}$, the following theorem has to be introduced and proved.

The original value x_f is greater than or equal to zero, if and only if the coded value x_c is greater than or equal to zero.

$$\begin{aligned} x_f \geq 0 \Leftrightarrow x_c \geq 0 \text{ with } x_f \in \mathbb{Z} \text{ and } x_c = \sigma(x_f) = A * x_f + B_x + D \\ \text{where } A \in \mathbb{N}^+,\; B_x, D \in \mathbb{N}_0,\; B_x + D < A \end{aligned} \tag{9}$$

Proof.

$$
\begin{aligned}
& x_c && \geq 0 \\
\Leftrightarrow\quad & A * x_f + B_x + D && \geq 0 \\
\Leftrightarrow\quad & A * x_f && \geq -(B_x + D) \\
\Leftrightarrow\quad & x_f && \geq \underbrace{-\frac{\overbrace{B_x + D}^{<A}}{A}}_{\in\,]\text{-}1,\, 0]} \\
\Leftrightarrow\quad & x_f && \geq 0, \text{ since } x_f \in \mathbb{Z}
\end{aligned}
$$

□

The goal is to implement a function returning TRUE_c, if and only if the coded value x_c (and thus x_f) is greater or equal to zero. Correspondingly, the function has to return FALSE_c, if and only if x_c is less than zero. As an extension to Definition 8, ERROR_c should be returned in case of a fault, e. g. if x_c is not a valid code word.

By applying the $\geq$ operator according to Equation (9), it can be checked whether x_c is negative or non-negative, but it cannot be checked whether x_c is a valid code word. Additionally, this procedure is very similar to the procedure in the original domain. The use of the unsigned modulo function umod is a possible solution to that problem. This function is applied to the coded value x_c. The idea of this approach is based on (Forin, 1989):

$$x_c \text{ umod } A = \text{unsigned}(x_c) \bmod A = \text{unsigned}(A * x_f + B_x + D) \bmod A$$

In order to resolve the unsigned function, two different cases have to be distinguished:

case 1: $x_f \geq 0$

$$
\begin{aligned}
x_c \text{ umod } A &= \text{unsigned}(\underbrace{A * x_f + B_x + D}_{x_f \geq 0 \;\Rightarrow\; x_c \geq 0 \text{ (cf. Eqn. (9))}}) \bmod A \\
&= (\underbrace{(A * x_f) \bmod A}_{=0} + \underbrace{B_x + D}_{<A}) \bmod A \\
&= B_x + D
\end{aligned}
$$

case 2: $x_f < 0$

$$
\begin{aligned}
x_c \text{ umod } A &= \text{unsigned}(\underbrace{A * x_f + B_x + D}_{x_f < 0 \;\Rightarrow\; x_c < 0 \text{ (cf. Eqn. (9))}}) \bmod A \\
&= \underbrace{(A * x_f + B_x + D + 2^n)}_{\text{resolved unsigned function}} \bmod A \\
&= (\underbrace{(A * x_f) \bmod A}_{=0} + B_x + D + 2^n) \bmod A \\
&= (B_x + D + 2^n) \bmod A \\
&= (B_x + D + \underbrace{(2^n \bmod A)}_{\text{known constant}}) \bmod A
\end{aligned}
$$

Conclusion of these two cases:

Result of case 1:

$$x_f \geq 0 \quad \Rightarrow \quad x_c \,\text{umod}\, A = B_x + D \tag{10}$$

Result of case 2:

$$x_f < 0 \quad \Rightarrow \quad x_c \,\text{umod}\, A = (B_x + D + (2^n \bmod A)) \bmod A \tag{11}$$

Remark: The index n represents the minimum number of bits necessary for storing x_c. If x_c is stored in an int32 variable, n is equal to 32.

It has to be checked, if in addition to the two implications (10) and (11) the following implications

$$\begin{aligned}
x_c \,\text{umod}\, A &= B_x + D && \Rightarrow \quad x_f \geq 0 \\
x_c \,\text{umod}\, A &= (B_x + D + (2^n \bmod A)) \bmod A && \Rightarrow \quad x_f < 0
\end{aligned}$$

hold. These implications are only valid and applicable, if the two terms $B_x + D$ and $(B_x + D + (2^n \bmod A)) \bmod A$ are never equal. In the following, equality is assumed and conditions on A are identified that have to hold for a disproof:

$$B_x + D \quad = \quad (\underbrace{\underbrace{B_x + D}_{\in\,[0,\,\text{A-1}]} + \underbrace{(2^n \bmod A)}_{\in\,[0,\,\text{A-1}]}}_{\in\,[0,\,\text{2A-2}]}) \bmod A$$

case 1: $\quad 0 \quad \leq \quad (B_x + D + (2^n \bmod A)) \quad < \quad A$

$$\begin{aligned}
& B_x + D && = \underbrace{(B_x + D + (2^n \bmod A))}_{\in\,[0,\,\text{A-1}]} \bmod A \\
\Leftrightarrow \quad & B_x + D && = B_x + D + (2^n \bmod A) \\
\Leftrightarrow \quad & 2^n \bmod A && = 0 \\
\Leftrightarrow \quad & 2^n && = k * A \qquad \forall\, k \in \mathbb{N}^+ \\
\Leftrightarrow \quad & A && = \frac{2^n}{k}
\end{aligned}$$

Since $A \in \mathbb{N}^+$ and 2^n is only divisible by powers of 2, k has to be a power of 2, and, therefore, the same holds for A. That means, if A is not a number to the power of 2, inequality holds in case 1.

case 2: $\quad A \quad \leq \quad (B_x + D + (2^n \bmod A)) \quad \leq \quad 2A - 2$

$$\begin{aligned}
& B_x + D && = \underbrace{(B_x + D + (2^n \bmod A))}_{\in\,[\text{A},\,\text{2A-2}]} \bmod A \\
\Leftrightarrow \quad & B_x + D && = B_x + D + (2^n \bmod A) - A \\
\Leftrightarrow \quad & A && = \underbrace{2^n \bmod A}_{\in\,[0,\,\text{A-1}]}
\end{aligned}$$

This cannot hold since the result of the modulo-operation is always smaller than A.

The two implications (10) and (11) can be extended to equivalences, if A is chosen not as a number to the power of 2. Thus for implementing the geqz$_c$ operator, the following conclusions can be used:

1. IF x_c umod $A = B_x + D$ THEN $x_f \geq 0$.
2. ELSE IF x_c umod $A = (B_x + D + (2^n \bmod A)) \bmod A$ THEN $x_f < 0$.
3. ELSE x_c is not a valid code word.

The geqz$_c$ operator is implemented based on this argumentation. Its application is presented in Listing 2, whereas its uncoded form is presented in Listing 1.

4. Safety code weaving for C control structures

In the former sections, a subset of SES transformation was discussed. The complete set of transformations for data, arithmetic operators, and Boolean operators are collected in a C library. In the following, the principle procedure of safety code weaving is motivated for C control structures. An example code is given in Listing 1 that will be safeguarded in a further step.

Listing 1. Original version of the code. It will be safeguarded in further steps.

```
int af = 1;
int xf = 5;

if ( xf >= 0 )
{
    af = 4;
}
else
{
    af = 9;
}
```

In general, there are a few preconditions for the original, non-coded, single channel C source code: e. g. operations should be transformable and instructions with short expressions are preferred in order to simplify the coding of operations.

Safety code weaving is realized in compliance with nine rules:

1. *Diverse data.* The declaration of coded variables and coded constants have to follow the underlying code definition.
2. *Diverse operations.* Each original operation follows directly the transformed operation.
3. *Update of dynamic signature.* In each task cycle, the dynamic signature of each variable has to be incremented.
4. *Local (logical) program flow monitoring.* The C control structures are safeguarded against local program flow errors. The branch condition of the control structure is transformed and checked inside the branch.

5. *Global (logical) program flow monitoring.* This technique includes a specific initial key value and a key process within the program function to assure that the program function has completed in the given parts and in the correct order (Leaphart, 2005). An alternative operating system based approach is given in Raab (2011).
6. *Temporal program flow monitoring.* Dedicated checkpoints have to be added for monitoring periodicity and deadlines. The specified execution time is safeguarded.
7. *Comparator function.* Comparator functions have to be added in the specified granularity in the program flow for each task cycle. Either a comparator verifies the diverse channel results $z_c = A * z_f + B_z + D$?, or the coded channel is checked directly by checking the condition $(z_c - B_z - D) \bmod A = 0$?.
8. *Safety protocol.* Safety critical and safety related software modules (in the application software layer) communicate intra or inter ECU via a safety protocol (Mottok, 2006). Therefore a safety interface is added to the functional interface.
9. *Safe communication with a safety supervisor.* Fault status information is communicated to a global safety supervisor. The safety supervisor can initiate the appropriate (global) fault reaction (Mottok, 2006).

The example code of Listing 1 is transformed according to the rules 1, 2, 4, and 5 in Listing 2. The C control structures while-Loop, do-while-Loop, for-Loop, if-statement, and switch-statement are transformed in accordance with the complete set of rules. It can be realized that the geqz$_c$ operator is frequently applied for safeguarding C control structures.

5. The case study: Simplified sensor actuator state machine

In the case study, a simplified sensor actuator state machine is used. The behavior of a sensor actuator chain is managed by control techniques and Mealy state machines.

Acquisition and diagnosis of sensor signals are managed outside of the state machine in the input management whereas the output management is responsible for control techniques and for distributing the actuator signals. For both tasks, a specific basic software above the application software is necessary for communication with D/A- or A/D-converters. As discussed in Fig. 1, a diagnosis of D/A-converter is established, too.

The electronic accelerator concept (Schaueffele, 2004) is used as an example. Here diverse sensor signals of the pedal are compared in the input management. The output management provides diverse shut-off paths, e. g. power stages in the electronic subsystem.

Listing 2. Example code after applying the rule 1, 2, 4 and 5.

```
int af;                   int ac;
int xf;                   int xc;
int tmpf;                 int tmpc;

cf = 152; /* begin basic  block 152 */
af   = 1;                 ac   = 1*A + Ba + D; //coded 1
xf   = 5;                 xc   = 5*A + Bx + D; //coded 5
tmpf = ( xf >= 0 );       tmpc = geqz_c( xc );
                          // greater/equal zero operator

if ( cf != 152 ) { ERROR } /* end basic block 152 */
if ( tmpf )
{
    cf = 153; /* begin basic block 153 */
    if ( tmpc - TRUE_C ) { ERROR }
    af = 4;               ac   = 4*A + Ba + D; //coded 4
    if ( cf != 153 ) { ERROR } /* end basic block 153 */
}
else
{
    cf = 154; /* begin basic block 154 */
    if ( tmpc - FALSE_C ) { ERROR }
    af = 9;               ac   = 9*A + Ba + D; //coded 9
    if ( cf != 154 ) { ERROR } /* end basic block 154 */
}
```

The input management processes the sensor values (s1 and s2 in Fig. 6), generates an event, and saves them on a blackboard as a managed global variable. This is a widely used implementation architecture for software in embedded systems for optimization performance, memory consumption, and stack usage. A blackboard (Noble, 2001) is realized as a kind of data pool. The state machine reads the current state and the event from the blackboard, if necessary executes a transition and saves the next state and the action on the blackboard. If a fault is detected, the blackboard is saved in a fault storage for diagnosis purposes.

Finally, the output management executes the action (actuator values a1, a2, a3, and a4 in Fig. 6). This is repeated in each cycle of the task.

The Safety Supervisor supervises the correct work of the state machine in the application software. Incorrect data or instruction faults are locally detected by the comparator function inside the state machine implementation whereas the analysis of the fault pattern and the initiation of a dedicated fault reaction are managed globally by a safety supervisor (Mottok, 2006). A similar approach with a software watchdog can be found in (Lauer, 2007).

The simplified state machine was implemented in the Safely Embedded Software approach. The two classical implementation variants given by nested switch statement and table driven design are implemented. The runtime and the file size of the state machine are measured and compared with the non-coded original one for the nested switch statement design.

The measurements of runtime and file size for the original single channel implementation and the transformed one contain a ground load corresponding to a simple task cycle infrastructure of 10,000,000 cycles. Both the NEC Fx3 V850ES 32 bit microcontroller, and the Freescale S12X 16 bit microcontroller were used as references for the Safely Embedded Software approach.

5.1 NEC Fx3 V850ES microcontroller

The NEC Fx3 V850ES is a 32 bit microcontroller, being compared with the Freescale S12X more powerful with respect to calculations. It runs with an 8 MHz quartz and internally with 32 MHz per PLL. The metrics of the Simplified Sensor Actuator State Machine (nested switch implemented) by using the embedded compiler for the NEC are shown in Table 2. The compiler "Green Hills Software, MULTI v4.2.3C v800" and the linker "Green Hills Software, MULTI v4.2.3A V800 SPR5843" were used.

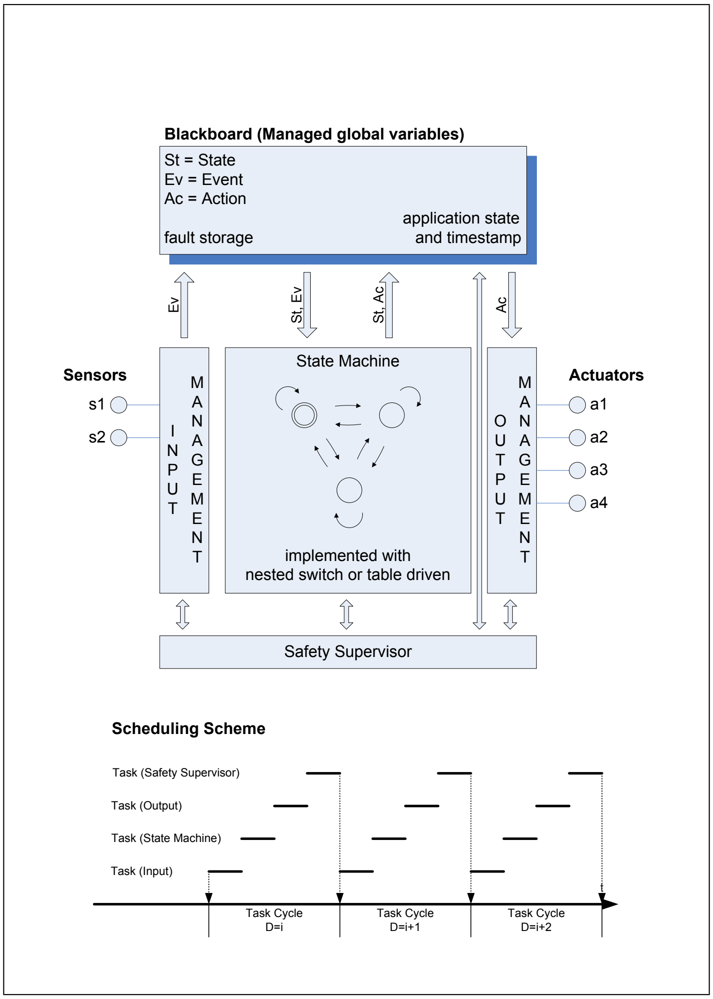


Fig. 6. Simplified sensor actuator state machine and a scheduling schema covering tasks for the input management, the state machine, the output management and the safety supervisor. The task cycle is given by dynamic signature *D*, which can be realized by a clocked counter.

5.2 Freescale S12X microcontroller

The Freescale S12X is a 16 bit microcontroller and obviously a more efficient control unit compared to the NEC Fx3 V850ES. It runs with an 8 MHz quartz and internally with 32 MHz per PLL. The processor is exactly denominated as “PC9S12X DP512MFV”. The metrics of the Simplified Sensor Actuator State Machine (nested switch implemented) by using the compiler for the Freescale S12X are shown in Table 3. The compiler “Metrowerks 5.0.28.5073” and the linker “Metrowerks SmartLinker 5.0.26.5051” were used.

	minimal code	original code	trans-formed code	factor	annotation
CS (init)	2	48	184	3.96	init code, run once
CS (cycle)	2	256	2,402	9.45	state machine, run cyclic
CS (lib)	0	0	252	-	8 functions for the transformed domain used: add_c, div_c, geqz_c, lz_c, ov2cv, sub_c, umod, updD
DS	0	40	84	2.10	global variables
SUM (CS, DS)	4	344	2,922	8.58	sum of CS(init), CS(cycle), CS(lib) and DS
RUN-TIME	0.20	4.80	28.80	6.22	average runtime of the cyclic function in μs
FILE-SIZE	4,264, 264	4,267, 288	4,284, 592	6.72	size (in bytes) of the binary, executable file

Table 2. Metrics of the Simplified Sensor Actuator State Machine (nested switch implemented) using the NEC Fx3 V850ES compiler.

	minimal code	original code	trans-formed code	factor	annotation
CS (init)	1	41	203	5.05	init code, run once
CS (cycle)	1	212	1,758	8.33	state machine, run cyclic
CS (lib)	0	0	234	-	8 functions for the transformed domain used: add_c, div_c, geqz_c, lz_c, ov2cv, sub_c, umod, updD
DS	0	20	42	2.10	global variables
SUM (CS, DS)	2	273	2,237	8.25	sum of CS(init), CS(cycle), CS(lib) and DS
RUN-TIME	0.85	6.80	63.30	10.50	average runtime of the cyclic function in μs
FILE-SIZE	2,079, 061	2,080, 225	2,088, 557	8.16	size (in bytes) of the binary, executable file

Table 3. Metrics of the Simplified Sensor Actuator State Machine (nested switch implemented) using the Freescale S12X compiler.

5.3 Results

The results in this section are based on the nested switch implemented variant of the Simplified Sensor Actuator State Machine of Section 5. The two microcontrollers NEC Fx3 V850ES and Freescale S12X need roundabout nine times memory for the transformed code and data as it is necessary for the original code and data. As expected, there is a duplication of data segement size for both investigated controllers because of the coded data.

There is a clear difference with respect to the raise of runtime compared to the need of memory. The results show that the NEC handles the higher computational efforts as a result of additional transformed code much better than the Freescale does. The runtime of the NEC only increases by factor 6 whereas the runtime of the Freescale increases by factor 10.

5.4 Optimization strategies

There is still a potential for optimizing memory consumption and performance in the SES approach:

- Run time reduction can be achieved by using only the transformed channel.
- Reduction of memory consumption is possible by packed bit fields, but more effort with bit shift operations and masking techniques.
- Using of macros like inline functions.
- Using initializations at compile time.
- Caching of frequently used values.
- Using efficient assembler code for the coded operations from the first beginning.
- First ordering frequently used cases in nested switch(Analogously: entries in the state table).
- Coded constants without dynamic signature.

In the future, the table driven implementation variant will be verified for file size and runtime with cross compilers for embedded platforms and performance measurements on embedded systems.

6. Comprehensive safety architecture and outlook

Safely Embedded Software gives a guideline to diversify application software. A significant but acceptable increase in runtime and code size was measured. The fault detection is realized locally by SES, whereas the fault reaction is globally managed by a Safety Supervisor.

An overall safety architecture comprises diversity of application software realized with the nine rules of Safely Embedded Software in addition to hardware diagnosis and hardware redundancy like e. g. a clock time watchdog. Moreover environmental monitoring (supply voltage, temperature) has to be provided by hardware means.

Temporal control flow monitoring needs control hooks maintained by the operation system or by specialized basic software.

State of the art implementation techniques (IEC61508, 1998; ISO26262, 2011) like actuator activation by complex command sequences or distribution of command sequences (instructions) in different memory areas have been applied. Furthermore, it is recommended to allocate original and coded variables in different memory branches.

Classical RAM test techniques can be replaced by SES since fault propagation techniques ensures the propagation of the detectability up to the check just before the output to the plant.

A system partitioning is possible, the comparator function might be located on another ECU. In this case, a safety protocol is necessary for inter ECU communication. Also a partitioning of different SIL functions on the same ECU is proposed by coding the functions with different prime multipliers A_1, A_2 and A_3 depending on the SIL level. The choice of the prime multiplier is determined by maximizing their pairwise lowest common multiple. In this context, a fault tolerant architecture can be realized by a duplex hardware using in each channel the SES approach with different prime multipliers A_i. In contrast to classical faul-tolerant architectures, here a two channel hardware is sufficient since the correctness of data of each channel are checked individually by determination of their divisibility by A_i.

An application of SES can be motivated by the model driven approach in the automotive industry. State machines are modeled with tools like Matlab or Rhapsody. A dedicated safety code weaving compiler for the given tools has been proposed. The intention is to develop a single channel state chart model in the functional design phase. A preprocessor will add the duplex channel and comparator to the model. Afterwards, the tool based code generation can be performed to produce the required C code.

Either a safety certification (IEC61508, 1998; ISO26262, 2011; Bärwald, 2010) of the used tools will be necessary, or the assembler code will be reviewed. The latter is easier to be executed in the example and seems to be easier in general. Further research in theory as well as in practice will be continued.

7. References

AUTOSAR consortium. (2011). *AUTOSAR*, Official AUTOSAR web site:www.AUTOSAR.org.

Braband, J. (2005). *Risikoanalysen in der Eisenbahn-Automatisierung*,Eurailpress, Hamburg.

Douglass, B. P. (2011). *Safety-Critical Systems Design*, i-Logix, Whitepaper.

Ehrenberger W. (2011). *Software-Verifikation*, Hanser, Munich.

Forin, P. (1989). *Vital Coded Microprocessor Principles and Application for Various Transit Systems*, IFAC Control, Computers, Communications, pp. 79-84, Paris.

Hummel, M., Egen R., Mottok, J., Schiller, F., Mattes, T., Blum, M., Duckstein, F. (2006). *Generische Safety-Architektur für KFZ-Software*, Hanser Automotive, 11, pp. 52-54, Munich.

Mottok, J., Schiller, F., Völkl, T., Zeitler, T. (2007). *Concept for a Safe Realization of a State Machine in Embedded Automotive Applications*, International Conference on Computer Safety, Reliability and Security, SAFECOMP 2007, Springer, LNCS 4680, pp.283-288, Munich.

Wappler, U., Fetzer, C. (2007). *Software Encoded Processing: Building Dependable Systems with Commodity Hardware*, International Conference on Computer Safety, Reliability and Security, SAFECOMP 2007, Springer, LNCS 4680, pp. 356-369, Munich.

IEC (1998). *International Electrotechnical Commission (IEC):Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems*.

ISO (2011). *ISO26262 International Organization for Standardization Road Vehicles Functional Safety, Final Draft International Standard*.

Leaphart, E.G., Czerny, B.J., D'Ambrosio, J.G., Denlinger, C.L., Littlejohn, D. (2005). *Survey of Software Failsafe Techniques for Safety-Critical Automotive Applications*, SAE World Congress, pp. 1-16, Detroit.

Motor Industry Research Association (2004). *MISRA-C: 2004, Guidelines for the use of the C language in critical systems*, MISRA, Nuneaton.

Börcsök, J. (2007). *Functional Safety, Basic Principles of Safety-related Systems*, Hüthig, Heidelberg.

Meyna, A., Pauli, B. (2003). *Taschenbuch der Zuverlässigkeits- und Sicherheitstechnik*, Hanser, Munich.

Noble, J., Weir, C.(2001). *Small Memory Software, Patterns for Systems with Limited Memory*, Addison Wesley, Edinbourgh.

Oh, N., Mitra, S., McCluskey, E.J. (2002). *4I:Error Detection by Diverse Data and Duplicated Instructions*, IEEE Transactions on Computers, 51, pp. 180-199.

Rebaudengo, M., Reorda, M.S., Torchiano, M., Violante, M. (2003). *Soft-error Detection Using Control Flow Assertions*, 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 581-588, Soston.

Ozello, P. (2002). *The Coded Microprocessor Certification*, International Conference on Computer Safety, Reliability and Security, SAFECOMP 1992, Springer, pp. 185-190, Munich.

Schäuffele, J., Zurawka, T. (2004). *Automotive Software Engineering*, Vieweg, Wiesbaden.

Tarabbia, J.-F.(2004), *An Open Platform Strategy in the Context of AUTOSAR*, VDI Berichte Nr. 1907, pp. 439-454.

Torres-Pomales, W.(2000). *Software Fault Tolerance: A Tutorial*, NASA, Langley Research Center, Hampton, Virginia.

Chen, X., Feng, J., Hiller, M., Lauer, V. (2007). *Application of Software Watchdog as Dependability Software Service for Automotive Safety Relevant Systems*, The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, Edinburgh.

Steindl, M., Mottok, J., Meier,H., Schiller, F., and Fruechtl, M. (2009). *Diskussion des Einsatzes von Safely Embedded Software in FPGA-Architekturen*, In Proceedings of the 2nd Embedded Software Engineering Congress, ISBN 978-3-8343-2402-3, pp. 655-661, Sindelfingen.

Steindl, M. (200). *Safely Embedded Software (SES) im Umfeld der Normen für funktionale Sicherheit*, Jahresrückblick 2009 des Bayerischen IT-Sicherheitsclusters, pp. 22-23, Regensburg.

Mottok, J. (2009) *Safely Embedded Software*,In Proceedings of the 2nd Embedded Software Engineering Congress, pp. 10-12, Sindelfingen.

Steindl, M., Mottok, J. and Meier, H. (2010) *SES-based Framework for Fault-tolerant Systems*, in Proceedings of the 8th IEEE Workshop on Intelligent Solutions in Embedded Systems, Heraklion.

Raab, P., Kraemer, S., Mottok, J., Meier, H., Racek, S. (2011). *Safe Software Processing by Concurrent Execution in a Real-Time Operating System*, in Proceedings, International Conference on Applied Electronics, Pilsen.

Laumer, M., Felis, S., Mottok, J., Kinalzyk, D., Scharfenberg, G. (2011). *Safely Embedded Software and the ISO 26262*, Electromobility Conference, Prague.

Bärwald, A., Hauff, H., Mottok, J. (2010). *Certification of safety relevant systems - Benefits of using pre-certified components*, In Automotive Safety and Security, Stuttgart.