# Programming Language Constructs Supporting Fault Tolerance

**Christina Houben**[1] **and Sebastian Houben**[2]

1 **Rheinische Friedrich-Wilhelms-Universität**
**Chemical Institutes, Bonn, Germany**
c-k-houben@uni-bonn.de

2 **Ruhr-Universität Bochum**
**Institute for Neural Computation, Bochum, Germany**
http://orcid.org/0000-0002-2036-419X
sebastian.houben@ini.rub.de

## Abstract

In order to render software viable for highly safety-critical applications, we describe how to incorporate fault tolerance mechanisms into the real-time programming language PEARL. Therefore, we present, classify, evaluate and illustrate known fault tolerance methods for software. We link them together with the requirements of the international standard IEC 61508-3 for functional safety. We contribute PEARL-2020 programming language constructs for fault tolerance methods that need to be implemented by operating systems, and code-snippets as well as libraries for those independent from runtime systems.

**2012 ACM Subject Classification** Control Structures and Microprogramming Control Structure Reliability, Testing, and Fault-Tolerance, Programming Languages, Language Constructs and Features, Computers in Other Systems, Real-time, Advanced Driver Assistance Systems, Space Flight

**Keywords and phrases** fault tolerance, functional safety, PEARL, embedded systems, software engineering

**Digital Object Identifier** 10.4230/LITES-v003-i001-a001

**Received** 2015-05-02 **Accepted** 2016-04-05 **Published** 2016-06-10

## 1 Introduction

Highly safety-critical applications need automation systems that are failsafe or at least fault tolerant. An automation system is called failsafe if it falls back into a stable state with a sufficient degree of functional safety. Functional safety is a system's property guaranteeing that the risk to harm human beings, environment or other assets is below a risk limit [3]. Automation systems are increasingly composed of software, thereby allowing to control more complex applications than pure hardware-based solutions, providing greater flexibility in adapting systems to changing requirements [22], consuming less space than mechanical constructions [9], and permitting to change system functionality by remote maintenance [17, 25]. Unfortunately, software cannot fall back into a safe state, triggered by laws of nature, like hardware [22]. Therefore, software systems have to use fault tolerance methods in order to provide functional safety as demanded in the international standard IEC 61508-3 [3]. Fault tolerance (FT) refers to a system fulfilling a specified function, even if a limited number of subsystems are erroneous [38]. FT methods rather prevent the consequences of a software error than the occurrence of the error itself [41]. FT methods, hence, are composed of error recognition and error treatment [38].

Our long-term objective is to adapt the real-time programming language PEARL-90 to functional safety as defined in the normative part of IEC 61508-3 [3]. In this paper we focus on the topic of fault tolerance by propagating syntax and semantics to its derivative PEARL-2020

and elucidating the rationale behind our adaptations. Introducing FT into PEARL is beneficial, since PEARL was the first language providing user-friendly concurrency concepts. These simple but powerful concurrency concepts render PEARL appropriate for highly safety-related software [22] which leads to a wide deployment throughout Europe [31]. PEARL is particularly tailored to process control due to its system part that addresses peripherals. It is standardised in DIN 66253-2 as PEARL-90 [4] and DIN 66253 Part 3 as Multiprocessor PEARL [1]. The latter is composed of two parts: the first part is a parametrisation language for compilers, binders and loaders and the second part describes various communication protocols for synchronous and asynchronous task cooperation and message passing. For a detailed introduction we refer to [19]. Currently, Müller and Schaible [32] map out a PEARL-90 compiler which will serve as a base for a future PEARL-2020 compiler. In our paper we contribute to the topic of fault tolerance in software as follows:

- We provide a complete overview of fault tolerance methods including their definition, classification and one or more coding examples, and
- present an evaluation of each FT method according to the insights from the examples with respect to stumbling blocks for programmers and suitability as language elements, code snippets or library implementations. In detail, we arrive at (c.f. Table 4):
  - additional PEARL syntax elements and semantics for assertions, load shedding, monotone tasks and the Byzantine method,
  - code snippets in PEARL for forward recovery, functional and implementation diversity,
  - PEARL library implementations for majority voting and other plausibility checks,
  - further changes to the PEARL-90 programming language, i.e. judiciously removing backward recovery, temporal redundancy and dynamic reconfiguration.

The outline of this paper is as follows. We start with related work in Section 2. Section 3 presents a comprehensive set of known fault tolerance methods for software with their classification and examples. A few examples came from literature, most of them are devised by ourselves in the application domains space flight and advanced driver assistance systems. Only one example per FT method will be realised in the final language standard, but we state them for comparison. This is the base for our framework addressing fault tolerance for software, in particular for PEARL. The framework comprises examples, code-snippets, library procedures and language primitives. In Section 4, we further explain the rationale behind this subdivision and link the FT methods with requirements from IEC 61508-3. We publish our implementations under the following web addresses:

- concrete examples and general fault tolerance handlers for PEARL-90:
  `http://sourceforge.net/projects/openpearl/files/Example`
- a fault tolerance library for PEARL-2020:
  `http://www.real-time.de/service/downloads.html`

## 2 Related Work

Most of the literature focuses on a single FT method. There are few papers that aim at combining more than one method: Shelton [36] derived patterns from three different FT methods considering task allocation and graceful degradation. DanYong and YongDuan [15] and Chen et al. [13] inspected fault tolerance from the mathematical point of view setting up differential equations in order to assess discrepancies from nominal values. Anwar [6] monitors a mainly software driven system and switches to a redundant mechanical system only in case of an error. A different area of application is treated by [30] who invented a fault tolerant TTP/C communication protocol. In contrast to these, our paper targets a larger amount of FT methods as can be seen in Table 4.

PEARL encompasses multiple derivatives, several of which already provide approaches for integrating FT. Multiprocessor PEARL offers keywords for dynamic reconfiguration. PEARL-90 allows for redundancy with the help of its easy to use concurrency concepts. This language derivative is branched into four subsets by [24], each addressing one of the four safety integrity levels (SILs) of IEC 61508. These subsets are Table-PEARL for SIL 4, Verifiable PEARL for SIL 3, Safe PEARL for SIL 2 and High Integrity-PEARL for SIL 1. Out of these, High Integrity-PEARL entails language elements to state alternative procedure bodies. Object-oriented derivatives of PEARL are PEARL* and Object-PEARL. PEARL* differs from PEARL-90 by keywords for objects and interfaces. Object-PEARL implements alternative methods and monotone tasks as FT concepts [14].

Fault tolerance in other languages is covered in Ada and languages like GRAFCET [2] for programmable logic controllers (PLCs). SPARK and Ravenscar inherit from Ada but restrict its variability for highly safety-critical applications. SPARK avoids error-prone language constructs and provides a certified tool chain for compiling [7], while Ravenscar parametrises the scheduling policy used in runtime systems in order to detect deadlocks and to guarantee timely task execution. PLC languages are standardised with libraries encompassing building blocks that support fault tolerance.

## 3 Fault Tolerance Methods

Fault tolerance methods are composed of error recognition and error treatment [38]. Error recognition is covered in the next section. A categorisation of error treatment methods is given in Section 3.2 and their elementary techniques in Section 3.3.

### 3.1 Error Recognition

In order to recognise an error, an automation system needs additional information on the range of values of an expected result. This additional information can either entail functional content, checked by voters, or is based on flow of control, checked by watchdogs. Examples for watchdogs are given in Table 1. Voters can be classified into absolute or relative voters. Absolute voters use hard-coded predicates as additional information, while relative voters compare the results of various implementations [10]. As a conclusion, absolute voters return whether a result is feasible or not. Since hard-coded predicates can be checked faster and more easily, they are predominantly used for dynamic redundancy. In contrast, a relative voter additionally returns a certain value of the output range or a combination of the output values. Examples of both types of voters are shown in Table 1.

Our first PEARL example is a majority voter. It can be used for `N` floating point numbers and returns the majority value if more than `N // 2` units possess a similar value or `ERROR_FLOAT` if not. Similar values are found by initial clustering. Afterwards, the cluster with most elements is processed.

```
MajorityVoting: PROCEDURE(Values() FLOAT IDENT) RETURNS(FLOAT);
  DCL Precision  FLOAT INIT(0.01),
      CntClasses FIXED INIT(0),
      (RoundValues(N), ClassValues(N)) FLOAT,
      CntClassMembers(N) FIXED, Found BIT,
      (CntMaxMembers, IdxMaxMembers) FIXED;
  ! Clustering of input values.
  FOR i FROM 1 TO N REPEAT
    RoundValues(i) := ROUND(Values(i) / Precision) * Precision;
    Found := '0'B;
    FOR k FROM 1 TO CntClasses REPEAT
      IF RoundValues(i) EQ ClassValues(k) THEN
        ! Existing class gets new member.
```

**Table 1** Types of watchdogs and voters. This table shows error recognition methods that can be applied before any fallback state of a fault tolerance method is triggered.

| |
|---|
| watchdogs:<br>- heartbeat to monitor the system's availability<br>- compatibility of actual with formal arguments [8]<br>- checking data integrity with respect to contents and structure [8]<br>- supervision of infinite loops or other unintended branching of the control flow, realisable by diverse conditions in branches [8]<br>- monitoring of runtime expenditure of tasks and noticing anomalies like frequent interruptions [8] |
| absolute voters:<br>- pre- and post-conditions<br>- assertions<br>- invariants of loops or objects |
| relative voters:<br>- majority voting, i.e. at least $\lceil \frac{n}{2} \rceil$ units out of $n$ must provide the same result<br>- consensus voting, i.e. largest number of members with equal results [42]<br>- combinations like arithmetic mean, weighted sum, median or fuzzy logic [42]<br>- statistical prediction, e.g. Kalman filtering [18]<br>- test against the inverse of a function [8], e.g. matrix inversion by $A \cdot A^{-1} = I$ [33]<br>- checksums and parity bits, e.g. checksums for matrix multiplication [33] |

```
        CntClassMembers(k) := CntClassMembers(k) + 1;
        Found := '1'B;
        EXIT;
      FIN;
    END;
    IF Found EQ '0'B THEN
      ! A new class has to be created.
      CntClasses := CntClasses + 1;
      ClassValues(CntClasses) := RoundValues(i);
      CntClassMembers(CntClasses) := 1;
    FIN;
  END;
  ! Find class with max member count.
  CntMaxMembers := -1;
  IdxMaxMembers := -1;
  FOR i FROM 1 TO CntClasses REPEAT
    IF CntClassMembers(i) > CntMaxMembers THEN
      CntMaxMembers := CntClassMembers(i);
      IdxMaxMembers := i;
    FIN;
  END;
  IF CntMaxMembers > N // 2 THEN
    RETURN(Result);
  ELSE
    RETURN(ERROR_FLOAT);
  FIN;
END;
```

## 3.2 Error Treatment

### 3.2.1 Redundancy

Redundancy means deploying more resources than necessary [38]. Redundant units increase functional safety, because automation systems with $n$ redundant and independent resources are robust against failing of $n-1$ resources. Furthermore, failsafe hardware does not need an error detection unit, since it fails due to environmental effects. Software, however, is in need of additional error detection. If a software implementation produces an erroneous result, this is revealed by the results from the other redundant implementations. For this reason, software is only robust against

**Table 2** Matrix explaining hierarchy of error treatment with software fault tolerance.

| | redundancy | | graceful degradation | |
|---|---|---|---|---|
| | homogeneous | diversity | homogeneous | diversity |
| functional diversity | | ✓ | | ✓ |
| load shedding | | | | ✓ |
| milestone method | | | ✓ | |
| implementation diversity | | ✓ | | ✓ |
| backward recovery | | ✓ | | ✓ |
| temporal redundancy | ✓ | | | |
| forward recovery | | ✓ | | ✓ |
| timed forward recovery | | ✓ | | ✓ |
| timed data diversity | | | ✓ | |
| dynamic reconfiguration | ✓ | | | |
| Byzantine method | ✓ | | | |

less than $n-1$ errors. Hierarchical redundancy refers to nesting different fault tolerance methods. Redundancy is further divided into temporal, analytic, static and dynamic methods [36, 23]. We explain these types later on.

### 3.2.2 Diversity

Plain redundancy refers to a homogeneous composition of multiple units, while diversity is a form of redundancy with each unit being different from the others. Diversity is differentiated into the types implementation, functional, physical, manufacturing diversity and diversity of operating conditions [22]. Physical and manufacturing diversity refer to hardware-based automation systems. Since we consider software safety, we will not target hardware topics in the following. The other diversity types are integrated into the presented fault tolerance methods later on. For software, only diverse implementations increase functional safety. Homogeneous implementations would contain the same errors by definition [23]. Exceptions to this rule are race conditions and transient hardware errors influencing registers used by a program.

### 3.2.3 Graceful Degradation

An automation system degrades gracefully if it provides reduced albeit specified automation behaviour in case of too many errors [40]. Here, a system can degrade either in functionality or availability [40]. Graceful degradation comes along with all fault tolerance methods for software aside from static redundancy.

## 3.3 Elementary Techniques for Error Treatment

After the error recognition step one or more of the following treatment methods are applied for fault tolerance. These elementary treatment methods can be categorised into homogeneous redundancy, diversity and graceful degradation as described above, see Table 2.

### 3.3.1 Analytical or Functional Diversity

The diverse components in a system using analytical redundancy or functional diversity, respectively, possess different specifications. They provide distinct although related functions. The relationship between the components allows either to restore a corrupted value with the help of the other

functions, or permits relative plausibility checks. One example is distance, velocity and acceleration [36] linked by the formula $a = \dot{v} = \ddot{s}$. Another one describes the dynamics of gas [36] inside a combustion chamber given by the formula $pV = nRT$, where $T$ is the temperature, $p$ the pressure, $V$ the gas volume, $n$ the amount of substance and $R$ the gas constant. Advantages and drawbacks of analytical redundancy can be studied by the following implementation:

```
TYPE TFunction REF PROC RETURNS(FLOAT);
TYPE TRelation REF PROC (Values() INV FLOAT IDENT) RETURNS(FLOAT);

TYPE TRelatedFunctions STRUCT (/
  Count            FIXED,
  Functions(1:Max) TFunction,
  Results  (1:Max) FLOAT,
    /* Results = measured values retrieved by Functions */
  Relations(1:Max) TRelation,
  Restored (1:Max) FLOAT
    /* Restored = calculated values retrieved by Relations */
/);

Initialisation: PROC (RelFuns TRelatedFunctions IDENT) GLOBAL;
  FOR i FROM 1 TO RelFuns.Count REPEAT
    RelFuns.Results(i) := RelFuns.Functions(i);
  END;
  FOR i FROM 1 TO RelFuns.Count REPEAT
    RelFuns.Restored(i) := RelFuns.Relations(i)(RelFuns.Results);
  END;
END;

Plausibility: PROC (RelFuns TRelatedFunctions INV IDENT) RETURNS(BIT) GLOBAL;
  FOR i FROM 1 TO RelFuns.Count REPEAT
    IF ABS(RelFuns.Results(i) - RelFuns.Restored(i)) GT Epsilon THEN
      RETURN(False);
    FIN;
  END;
  RETURN(True);
END;

Restoration: PROC(RelFuns TRelatedFunctions IDENT, Index FIXED) GLOBAL;
  RelFuns.Results(Index) := RelFuns.Relations(Index)(RelFuns.Results);
  FOR i FROM 1 TO RelFuns.Count REPEAT
    RelFuns.Restored(i) := RelFuns.Relations(i)(RelFuns.Results);
  END;
END;
```

- The precision of plausibility checks depends on the choice of the acceptance gap between the measured input values and the values calculated from the given formulas [33], see `Epsilon` in `Plausibility`.
- Plausibility checks between all participating diverse functions' results do not reveal which result is corrupt.
- This fault tolerance type is not suited as library implementation due to the following drawbacks:
- The complexity of this method is not situated in the library procedures, but in the implementation of the diverse functions.
- Additionally, the library procedures increase the program's complexity, since the linkage of the measured input values to arguments of the relationship functions has to be realised by a generic array `TRelatedFunctions.Results` in `Restoration` with no relation to the original variables' nomenclature.
- The restore-functions' signatures depend on the type of relationship, e.g. direct calculation in the case of gas dynamics vs. multiple inputs for derivatives in the case of movements.

In the next example we show how to use the library implementation from above in order to realise the gas dynamics formula. Line 3 shows the problem of mapping the application-related variables `(p, V, n, T)` to the formal parameters of the library `Values(1:Max)`, which is only viable by error-prone pointers (`REF` in PEARL):

```
DCL Max FIXED INIT(4);
DCL Values(1:Max) FLOAT;
DCL (p, V, n, T) REF FLOAT INIT(Value(1), Value(2), Value(3), Value(4));
DCL Funs() TFunction INIT(Piezo.Read, Flow.Read, GasScale.Read, Thermo.Read);
SPC (Pressure, Volume, Substance, Temperature) TRelation;
DCL Relations() TRelation INIT(Pressure, Volume, Substance, Temperature);
DCL GasChamberFuns TRelatedFunctions INIT(Max, Functions, Values, Relations);

Pressure: PROC(Values() INV FLOAT IDENT) RETURNS(FLOAT);
  RETURN(n * R * T / V);
END;

Volume: PROC(Values() INV FLOAT IDENT) RETURNS(FLOAT);
  RETURN(n * R * T / p);
END;

Substance: PROC(Values() INV FLOAT IDENT) RETURNS(FLOAT);
  RETURN((R * T) / (p * V));
END;

Temperature: PROC(Values() INV FLOAT IDENT) RETURNS(FLOAT);
  RETURN((n * R) / (p * V));
END;

GasChamberControl: TASK MAIN;
  REPEAT
    AFTER 5 MSEC RESUME;
    Initialisation;
    IF NOT Plausibility(GasChamberFuns) THEN
      ...
      Restoration(GasChamberFunctions, ErrIdx);
      ...
    FIN;
  END;
END;
```

### 3.3.2 N-Version Programming

With N-version programming, a software part addressing a certain problem provides at least two solution methods with the same in- and output interface. These solutions can be diverse implementations or further enrichment of a result. A solution may be fortified with information from a task that may also be skipped, or with information from a task that provides more precise results the longer it is executed. N-version programming is divided into three sub-methods, namely sieve method, milestone method and implementation diversity [29].

### 3.3.3 Load Shedding or Sieve Method

If a hazardous or unanticipated situation occurs, an embedded computing system might react by scheduling more tasks than in normal mode. The higher the number of unanticipated requests is, the higher the number of tasks to be executed will be. In such situations, not all of these tasks can meet their deadlines. A fault tolerance method handling such cases is load shedding or the sieve method. There are two variants, i.e. skipping all tasks of minor importance and extending the periods [34]. Properties are:

- In contrast to hardware redundancy, load shedding does not underutilise processors when no overloads occur, at the expense of losing minor functionality [34].
- The difference to monotone tasks is, that sieve methods shall be either completed or skipped entirely, because there is no benefit to partly execute them [29].
- A programmer shall be able to group sieve methods, since for some applications, it is useless or even flawed to execute certain tasks if others were skipped.
- Load shedding is interwoven with the runtime system, since the applied scheduling strategy must allow to detect transient overloads before important tasks miss their deadlines. Earliest

deadline first (EDF) is applicable as scheduling strategy. Moreover, worst case execution times (WCETs) must be determined by a compiler, as explained in the following paragraph.
- Load shedding needs a mechanism for stating which tasks to skip primarily. On one hand, this can be done by priorities as in the next example, on the other hand, by modes as in the example thereafter.

If load shedding is to be applied, worst case execution times for each affected task should be provided by the compiler in the declaration part. A compiler can determine WCETs statically by summing up known maximum runtimes for each assembler instruction, multiplying them for loops, and using the worst case branch on conditional program jumps [22]. This procedure nearly always yields too pessimistic WCETs [37]. Therefore, compilers can build upon one of the following three alternatives:
- restricting language constructs (e.g. prohibit unbounded loops) [22],
- simplifying processor architecture (e.g. omit pipelining and caches, only fixed point arithmetic) [8],
- assessing WCETs empirically with the help of very pessimistic cache trashers (guaranteeing maximum number of cache misses) [20].

The following PEARL specifications illustrate language constructs for load shedding in the context of space vehicles. `TIMING` and `LOADSHEDDING` are module parts like `SYSTEM` and `PROBLEM`. The timing part is a synopsis of the timing analysis of all tasks, where `WCET` states the worst case execution time determined by a compiler and `RESPONSE` states the available response duration determined by an engineer. The first example shows how to state priorities. In case of transient overloads, a runtime system has to remove all tasks beginning with those carrying the smallest priority value. It gradually removes all tasks with the next priority value until all remaining tasks can meet their deadlines.

```
TIMING;
  TelescopeAdjustment:
    WCET(5.1 MSEC) RESPONSE(10 MSEC);
  ...
LOADSHEDDING;
  PRIO(1): ! skip first when in emergency mode
    TelescopeAdjustment, AntennaAdjustment;
  PRIO(2): ! skip second when in emergency mode
    EngineFineControl, SolarCellsFineControl;
```

The aforementioned solution is based on two modes, namely normal and emergency mode [26], whereas the second example uses multiple modes and tasks assigned to them. The `TIMING` part equals the example from above, while the `LOADSHEDDING` part consists of several modes with several tasks, each of which is executed when the runtime system finds itself in the respective mode. `SCIENCEMODE` is the default and desired operating state. However, if no timely execution can be guaranteed, the runtime system switches into one of the other modes. In the worst case, `SAFEMODE` has to be executed. The mode is chosen as follows: Each time the schedule changes, i.e. a task is activated, suspended or terminated, the runtime slack of all active tasks is computed. If the slack is negative, the system has to switch into a lower mode. This is iterated until the slack is non-negative or `SAFEMODE` is reached. On positive slack the runtime system can change into a higher mode. In order to avoid toggling between two modes in both our examples, the following options are viable:
- the slack on the active mode must exceed a given threshold,
- after a given period of time, the system automatically switches into the next higher mode,
- the emergency mode is revoked by a human operator.

```
TIMING;
  MainEngineRoughControl:
    WCET(2.7 MSEC) RESPONSE(8 MSEC);
  ...
LOADSHEDDING;
  SCIENCEMODE:
    EngineFineControl, SolarCellsFineControl,
    TelescopeAdjustment, AntennaAdjustment;
  INTERMEDIATE:
    ...
  SAFEMODE:
    RollControl, EngineRoughControl, ...;
```

### 3.3.4 Monotone Tasks or Milestone Method

Monotone tasks produce results with higher quality the longer they are running. If such tasks are terminated before final completion, they return the most recent valid result. Each intermediate result represents a milestone. A monotone task is decomposable into a mandatory part and a number of optional parts [29]. In the following, we evaluate three implementation types for the milestone method:

```
DCL Flag BIT;
ZeroByNewton:
  PROCEDURE((F, D) TFunction) RETURNS(FLOAT);
  DCL Xi       FLOAT      INIT(0.0);
  DCL MaxLoops INV FIXED  INIT(20);
  DCL Loops    FIXED      INIT(0);
  WHILE (NOT Flag) AND (Loops <= MaxLoops)
  REPEAT
    Xi := Xi - F(Xi)/D(Xi);
    Loops := Loops + 1;
  END;
  RETURN(Xi);
END;
```

```
DCL Result FLOAT;
ZeroByNewton: PROCEDURE((F, D) TFunction);
  DCL Xi FLOAT INIT(0.0);
  FOR i FROM 1 TO 20 REPEAT
    Xi := Xi - F(Xi)/D(Xi);
    Result := Xi;
    UPDATE;
  END;
END;
```

```
ZeroByNewton:
  PROCEDURE((F, D) TFunction) RETURNS(FLOAT);
  DCL Xi FLOAT INIT(0.0);
  MILESTONE(Xi);
  ON EARLYEND: BEGIN RETURN(Xi); END;
  FOR i FROM 1 TO 20 REPEAT
    Xi := Xi - F(Xi)/D(Xi);
  END;
  RETURN(Xi);
END;
```

- The first implementation uses a flag for each task that can be terminated beforehand. To terminate a task, its flag has to be set. This implementation suffers from interspersing the pure task functionality with milestone checks and the unbounded duration between two milestones.
- Another implementation type by [14] is to introduce a keyword `UPDATE`. At each such update point the scheduler can decide whether to terminate or to continue the task. The drawbacks are the same as in the implementation before. Moreover, the result variable must be non-local.
- Our third proposition is to forbid termination heteronomy. Instead, a `TERMINATE` instruction for a monotone task shall produce a signal `EARLYEND`, which must be caught by monotone tasks. A signal handler inside such tasks, then, releases all resources and returns the most recent result. With this method, monotone tasks can be terminated at arbitrary points in time. The WCET of the signal handler is clear to state and allows to bound the termination's duration in an intelligible way. This implementation type needs atomic sections, which are not interruptible by signals in order to avoid race conditions for assignments to the result variable. Therefore, all commands executed on variables stated in a `MILESTONE` list are compiled as atomic.

### 3.3.5 Implementation Diversity

Implementation diversity uses at least two alternatives that fulfil the same function, but with different designs or implementations [22]. With respect to software, one can vary architecture, algorithms, data representations on the one hand, and operating systems, runtime systems, compilers, programming languages, integrated development environments and test methods on the other [23]. Implementation diversity can be used dynamically or statically. Dynamic execution means that a scheduler decides at runtime which alternative to execute next. In static execution, it is known beforehand that all alternatives are executed and, afterwards, a voter receives all results to determine the correct one. With the static variant, the alternatives can be executed sequentially or in parallel [23]. The following three implementations show these FT types by conventional PEARL constructs, while the fourth example demonstrates new language constructs proposed by [22].

With respect to functional safety, one can vary the alternatives in simplicity or in runtime complexity. These categories are not exclusive. Simplicity increases the quality of an alternative, since less complex solutions contain less programming errors [36]. Using the runtime consideration, one alternative shall produce an exact result with long processing time, the other an imprecise result with short processing time. Before executing an alternative, the scheduler calculates the amount of time available and chooses the alternative with the highest result quality under the constraint of timeliness. Table 3 shows which execution types are reasonable to combine. Dynamic implementation diversity is related to recovery blocks, as we will explain further on.

Without using specialised PEARL syntax, the following three examples demonstrate how to realise the aforementioned three implementation diversity types. By contrast, the fourth example uses specialised syntax from [22]. Therein, `DIVERSE` introduces a block of alternatives, where each block starts with `ALTERNATIVE`. The keyword `ASSURE` signals the beginning of the plausibility check.

```
SequentialStaticRedundancy: PROC(Input STRUCT, Output STRUCT IDENT) RETURNS(BIT);
  DCL Results(1:N) STRUCT;
  /* execute all alternatives */
  Results(1) := Alternative1(Input);
  ...
  Results(N) := AlternativeN(Input);
  /* relative plausibility check */
  RETURN(VoterDecision(Results, Output));
END;
```

**Table 3** Implementation diversity types: In dynamic implementation diversity, a scheduler decides at runtime which alternative to execute next, while static means that all alternatives are executed and the results are passed on to a voter. These alternatives can be executed sequentially or in parallel. During design of the alternative functions one can aim at optimising simplicity, runtime or other goals.

| | dynamic sequential | static sequential | static parallel |
|---|---|---|---|
| simplicity | ✗ | ✓ | ✓ |
| runtime | ✓ | ✗ | ✗ |
| other | ✓ | ✓ | ✓ |

```
DCL Input STRUCT;
DCL Results(1:N) STRUCT;
DCL Barrier(1:N) SEMAPHORE PRESET(1);

ParallelStaticRedundancy: PROCEDURE(X STRUCT, Y STRUCT IDENT) RETURNS(BIT);
  Input := X;
  /* execute all alternatives */
  ACTIVATE Alternative1;
  ...
  ACTIVATE AlternativeN;
  /* join with all alternatives */
  REQUEST Barrier(1);
  ...
  REQUEST Barrier(3);
  /* relative plausibility check */
  RETURN(VoterDecision(Results, Y));
END;
```

```
DynamicRedundancy: PROCEDURE(X STRUCT, Y STRUCT IDENT) RETURNS(BIT);
  /* execute first alternative */
  Y := Alternative1(X);
  /* absolute plausibility check */
  IF PlausiCheck(X,Y) THEN RETURN('1'B); FIN;
  /* evtl. execute and check others */
  ...
  /* evtl. execute last alternative */
  Y := AlternativeN(X);
  /* absolute plausibility check */
  IF PlausiCheck(X,Y) THEN RETURN('1'B); FIN;
  /* return error if all tasks failed */
  RETURN('0'B);
END;
```

```
DIVERSE
  ALTERNATIVE
    Segments1 := RegionGrowing(Bitmap);
  ALTERNATIVE
    Segments2 := EnergyMinimisation(Bitmap);
  ASSURE
    IF Diff(Segments1, Segments2) < 0.2 THEN
      RETURN(Intersect(Segments1,Segments2));
    ELSE
      INDUCE Error;
FIN;
```

### 3.3.6 Recovery Blocks

Recovery blocks are a fault tolerance method, where the "blocks" represent diverse implementations executed dynamically in sequential order and "recovery" refers to variables that need to be kept in memory for roll-back in order to restore a stable program status if a block failed. The following subsections describe both variants, namely backward and forward recovery.

### 3.3.7 Backward Recovery

Backward recovery is employed as follows [35, 36]: At first, backward recovery uses a checkpoint, if one of the following alternative blocks changes input- or program-state-variables during execution. A checkpoint is a snapshot of at least all the variables that could be changed by one of the diverse alternatives in the recovery block. Second, a primary implementation representing the conventional algorithm is applied. Third, an absolute plausibility test is evaluated after each alternative has been processed. This test is called acceptance test. It can contain an alternative's own post-condition or the post-condition of all alternatives. If the acceptance test fails, the system is rolled back to the last checkpoint, i.e. all variables are restored, and the next alternative is executed. When passing the acceptance test, all other alternatives of the block are ignored. If all alternatives fail, a computation error has to be reported followed by a fall-back to a surrounding fault tolerance level. Further reasons to roll back and retry are internal computation errors like division by zero and time-outs. Backward recovery embodies the following properties:

– Building a checkpoint consumes runtime and memory even if no error occurs [39]. For a supercomputer, creating a checkpoint file on a disc is reported to consume up to 25 min [11].

– Rolling back to a checkpoint takes runtime, but only in case of errors [39].

– Restoring a checkpoint takes time as well [11].

± If preventive checkpoints are not possible due to runtime overhead, periodic checkpointing can be applied [11].

± The post-conditions must be simple and ideally proven correct in order to prevent introducing further design errors [22].

± For checkpointing, it would be beneficial to have a primitive at hand, that closely packs all checkpoint variables without padding in order to transfer a single memory block with one instruction. The packing becomes an optimisation problem if different checkpoints are necessary.

```
Segmentation: PROCEDURE RETURNS(TSegments);
  DCL (PreSegments, Segments) TSegments;
  PreSegments := PreSegmentation;
  ! checkpoint
  Segments := PreSegments;
  ! primary implementation
  RegionGrowing(Bitmap, Segments);
  ! acceptance test
  IF Quality(Segments) < 0.5 THEN
    ! roll-back
    Segments := PreSegments;
    ! alternative
    EnergyMinimisation(Bitmap, Segments);
    ! acceptance test
    IF Quality(Segments) < 0.5 THEN
      ! fall-back
      INDUCE Error;
    END;
  FIN;
  RETURN(Segments);
END;
```

If only one alternative is executed multiple times, backward recovery degrades to temporal redundancy. It is not reasonable to execute the same code twice due to the systematic nature of software errors. Hence, an error would be produced twice [22]. This is only useful in case of corrupt data, e.g. flipped bits, or race conditions.

### 3.3.8 Forward Recovery

Forward recovery is a fault tolerance method working in the same way as backward recovery with the sole difference that each alternative has its own pre-condition. The first alternative whose pre-condition is fulfilled is executed [21].

- The pre-conditions of forward recovery allow to skip alternatives if it is known beforehand that certain data would cause an alternative to fail.
- This fact saves runtime and decreases the hazard of executing program errors that could render the program unstable.
- Moreover, pre-conditions facilitate reading and understanding of the program text and contribute information for program verification.
- If an alternative triggers a peripheral process that irrevocably changes the program state, checkpointing is not possible, but forward recovery is [21].
- The diverse alternatives must employ different input interfaces in such a way that each subsequent pre-condition is not stronger than its predecessors' pre-conditions, i.e. weaker or not related. Otherwise, the program would contain unreachable code.
- The order of alternatives and their pre-conditions shall be checked during compilation.

```
EdgeDetection: PROCEDURE RETURNS(BIT);
  DECLARE Success BIT;
  CALL CopyImages;
  IF Weather.Bright AND Weather.Dry THEN
    Success := SobelOperator(VisualImage);
    IF Success THEN RETURN(True);
    ELSE CALL RestoreImages; FIN;
  FIN;
  IF Weather.Dark AND Weather.Dry THEN
    Success := SobelOperator(InfraredImage);
    IF Success THEN RETURN(True);
    ELSE CALL RestoreImages; FIN;
  FIN;
  IF Weather.DARK AND (Weather.Fog OR Weather.Rain) THEN
    Success := SobelOperator(RadarImage);
    IF Success THEN RETURN(True);
    ELSE CALL RestoreImages; FIN;
  FIN;
  RETURN(False);
END;
```

```
TYPE TVoid REF STRUCT (/ /);
TYPE TProcedure REF PROC;
TYPE TFunction REF PROC(TVoid) RETURNS(BIT);
TYPE TBlock REF PROC(TVoid) RETURNS(TVoid);

TYPE TAlternative STRUCT (/
  Precondition   TFunction,
  Implementation TBlock,
  Postcondition  TFunction
/);

ApplyForwardRecovery: PROCEDURE(
  Alternatives() INV TAlternative IDENT,
  Input TVoid) RETURNS(TVoid) GLOBAL;
  DCL (Checkpoint, Output) TVoid;
  DCL Next BIT INIT(True);
  Checkpoint := Input;
  FOR i FROM 1 TO UPB(Alternatives) REPEAT
    IF Alternatives(i).Precond(Input) THEN
      Output := Alternatives(i).Impl(Input);
      IF Alternatives(i).Postcond(Output)
        THEN RETURN(Output);
        ELSE Input := Checkpoint;
      FIN;
    FIN;
  END;
  RETURN(NIL);
END;
```

The preconditions can be restricted on response times, in order to meet deadlines in case of transient overloads. Here, a scheduler decides which alternative to execute depending on the response time the system must achieve and the WCET of the alternatives. The first example shows the syntax of [22], while the second one shows the syntax of [14].

```
EvasiveManeuvre: TASK RUNTIME SELECTABLE;
  BODY
    ALTERNATIVE WITH RUNTIME 2500 MSEC;
      CALL ParticleFilter;
      ! the best parameter estimation
      ! out of many simulations
    ALTERNATIVE WITH RUNTIME 25 MSEC;
      CALL SimulationOnlyWithAvr;
      ! improves based on average value
      ! with only one simulation
  FIN;
END;
```

```
BrakeControl CLASS [
  EvasiveManeuvre: PROCEDURE RETURNS(FLOAT) VIRTUAL;
    CALL ParticleFilter;
  END;
  EvasiveManeuvre:: ALTPROCEDURE;
    CALL SimulationOnlyWithAvr;
  END;
];
```

### 3.3.9 Time-constrained Data Diversity

Data diversity means that the same algorithm is executed with input data that is represented in different ways. Data diversity is split into three further types [5]. One type uses dynamic redundancy, where data is reformulated if the used implementation raises an error. Another type uses static redundancy, where data is reformulated $n$ times and each version is processed by the same algorithm. Afterwards, a voter decides which output to return. Both methods circumvent errors due to unstable algorithms, but we recommend to use forward recovery instead and to use algorithms that are stable for a certain input space. In contrast, the third data diversity type can be employed to meet deadlines by resizing data to a smaller extent. For the implementation of a procedure using time-constrained data diversity, the procedure needs the remaining processing time and a worst case execution time with respect to a certain data size. Apart from that, an implementation is very clear to read.

```
ImageProcessing: PROCEDURE(Image TBitmap, RemainingTime DURATION);
  DCL WCETperPixel INV DURATION INIT(7 MSEC);
  Compress(Bitmap, Image.Width * Image.Height * WCETperPixel / RemainingTime);
  ...
END;
```

### 3.3.10 Dynamic Reconfiguration

After a watchdog has detected an error, it sets an error flag and, thereby, provokes the runtime system to shift certain task sets from one processor to another processor or node. Multiprocessor-PEARL realises dynamic reconfiguration by a `CONFIGURATION` part [1]. Its organisation is similar to High Integrity-PEARL, that uses `STATE`s with Boolean expressions and `LOAD...TO` as well as `REMOVE...FROM` keywords to shift task sets [22]. Dynamic reconfiguration creates a host of drawbacks:

– Dynamic reconfiguration suffers from hard-wiring of peripheral components to certain processors such that a shifted task cannot access certain resources [22].
– Some processors have to be reserved as spare units [11].
– Moreover, reconfiguration is a dynamic language construct. This implies that shifting a task set to another processor can cause unanticipated timing behaviour complicating the program schedule, provoking transient overload, or causing failures due to consuming an unexpected high amount of memory.

In order to avoid a system shut-down during reconfiguration, preventive migration together with error prediction can be applied [11]. Here, software execution continues, but processing speed slightly degrades during preventive migration. Preventive migration has to be combined with other fault tolerance methods if an error was not predicted [11].

### 3.3.11 Rejuvenation and Byzantine Method

Rejuvenation is restarting a system from a checkpoint [42]. The restart can be of the form rebooting, garbage collection, swapping space etc. [12]. It can be used for the whole system, a node or a task [12]. The Byzantine method uses rejuvenation for $n$ systems that are restarted periodically, each at a different point in time [27]. After a spare unit restarts, it uses the state stored by the other units.

\+ The Byzantine method can bridge the time of switching to another processor [25].
\+ Rejuvenation can be used to discharge a processor from radiation [25].
\+ It avoids numerical error accumulation [12].
± The downtime for rejuvenation is planned and, therefore, less hazardous than an unplanned downtime due to an error [12].
± It is applicable when system resources are exhausted or data is corrupt [12]. In this case, rejuvenation is a crude method, because the error source is not detected.
– Program execution is interrupted during reboot [11].

## 4 Conclusions for PEARL-2020

After insights in multiple fault tolerance methods for software and their implementations, we select appropriate methods for highly safety-critical applications. Table 4 gives a synopsis of all methods presented, their objective, their linkage to IEC 61508-3 regulations and recommendations with respect to the four safety integrity levels (SILs). The higher the SIL is, the higher the integrity of an automation system has to be. The SIL is determined by questions considering extent and limitation of damage for human beings, environment and assets, probability of failure and duration of stay in a danger area. We marked which methods are applicable and recommend which implementation type to use in PEARL-2020, the new standard that shall substitute PEARL-90 and Multiprocessor-PEARL. Therefore, we can choose from four implementation types, namely hand-coded, code-snippets, library procedures and language primitives. For election, we consider the following constraints formulated with respect to language primitives. This leads us to the aim of devising language primitives only for FT methods that must be monitored by a runtime system.

\+ If a language provides more primitives than necessary, a programmer can choose the best-suited one, which improves readability of the source text.
– With too many primitives, programs become unintelligible if the primitives are mixed [28].

**Table 4** Synopsis of the presented fault tolerance methods.

| fault tolerance method | objective | IEC 61508 | SIL 1 | SIL 2 | SIL 3 | SIL 4 | PEARL 2020 | implementation |
|---|---|---|---|---|---|---|---|---|
| abs. plausibility checks | correctness | A.2.3a | + | + | + | ++ | ✓ | primitives |
| rel. plausibility checks | correctness | A.2.3b | ± | + | + | ± | ✓ | library |
| functional diversity | correctness | A.2.3e | ± | ± | + | ++ | ✓ | code-snippets |
| load shedding | timeliness | – | | | | | ✓ | primitives (states) |
| milestone method | timeliness | – | | | | | ✓ | primitives (update) |
| implementation diversity | correctness | A.2.3d | ± | ± | ± | + | ✓ | code-snippets |
| backward recovery | correctness | A.2.3f | + | + | ± | – | ✗ | – |
| temporal redundancy | correctness | A.2.4a | + | + | ± | ± | ✗ | – |
| forward recovery | correctness | – | | | | | ✓ | code-snippets |
| timed forward recovery | timeliness | – | | | | | ✓ | code-snippets |
| timed data diversity | timeliness | – | | | | | ✓ | hand-coded |
| dynamic reconfiguration | correctness | A.2.6 | ± | – – | – – | – – | ✗ | – |
| Byzantine method | correctness | – | | | | | ✓ | primitives |

– Libraries are more appropriate than programming language constructs tailored to a certain area of application since the latter can become deprecated and cannot be exchanged easily if application areas develop further [16].
– With a minimum of language primitives, a language is easier to learn, better to understand and verify [22].
\+ With FT primitives, a compiler can choose whether to translate into sequential or parallel execution [22].
± Primitives as well as libraries allow to change program behaviour with only little adaptations to the source text, either by parametrisation of the compiler or by exchanging class and module names.

## 4.1 Individual Language Design Decisions

We suggest to introduce absolute plausibility checks by the primitives `PRECOND` for pre-conditions, `POSTCOND` for post-conditions and `ASSERT` for invariants within a procedure body. The explicit distinction of those three keywords addresses semi-automatic program verification for the pre- and post-conditions. Whether or not to check those assertions is to be parametrised in the compiler. Relative plausibility checks will be provided by a library implementation since the necessary checks, e.g. for majority voting, tend to be more complex and regulation A.2.8 from IEC 61508-3 demands to use verified software components.

We advocate for providing functional diversity and implementation diversity by use of code-snippets, because mapping the nomenclature of a program's area of application to arguments of an FT handler is misleading and IEC 61508-3 regulation B.1.5 advises against the use of pointers, refer to the example from Section 3.3.1. Benefits of code-snippets are that they guide programmers to best-practice and allow to easily switch from sequential to parallel execution or vice versa.

Regarding load shedding, two design decisions have to be made: First, whether to provide two states (normal and emergency) and assign a priority to every task group or to provide multiple user-defined states with associated tasks, second, how to indicate state transitions to the runtime system. We also point out that we neglect regulation A.2.3g from IEC 61508-3 which recommends a stateless program design. We prefer to define multiple states in the program system part since they

allow to discard but also re-enable tasks if a transient overload becomes more critical. Considering the second design choice there are two possible solutions as well: define the current state only by using the current processor load or let the transitions be initiated by the programmer. Both options bear the risk of toggling between two or more states. We prefer the second option since it allows to query diverse conditions. The conditions have to be stated within the `LOADSHEDDING` part of a PEARL module due to readability.

We base the milestone method on the language primitive `UPDATE` as proposed by [14], but with a single change: instead of using a global return variable we use PEARL's conventional function header syntax with the new attribute `MONOTONE` that entails the name of the return variable. With the example from Section 3.3.4 in mind, this means `ZeroNewton: PROC((F, D) TFunction) RETURNS(FLOAT) MONOTONE(Xi);` The hand-coded variant with flags is inappropriate, since abort conditions and setting the flag can be misimplemented by a programmer. The termination heteronomy variant is insufficient to handle semaphore operations in case of early function exits.

We decided to drop language constructs for backward recovery and temporal redundancy, as they are not recommended by IEC 61508-3 for higher SILs. Furthermore, backward recovery should be substituted by forward recovery and temporal redundancy only aims at race conditions that should be eliminated beforehand. Dynamic reconfiguration is left out as well, because IEC 61508-3 strongly advises against it due to its dynamic nature as stated in regulation B.1.2. In order to prevent the disadvantages that come with dynamic reconfiguration we support the use of the Byzantine method.

Forward recovery should be supported with code-snippets that entail sequential `IF`s for pre-conditions and nested `IF`s for post-conditions. Language primitives are ruled out, because they unnecessarily enlarge the set of keywords. Library procedures are excluded, since they imply the need for parametric polymorphism or inheritance. The code-snippet for time-constrained forward recovery extends the forward recovery code-snippet by one further input argument for the response duration. Time-constrained data diversity, on the other hand, should be hand-coded, because data can be simply and readably resized before the actual function implementation.

The rejuvenation of the Byzantine method has two aspects: what to rejuvenate and how. Since subsystems are addressed by other FT methods, we restrict it to the whole embedded system. Therefore, the time period and offset for rejuvenation of certain processors can be stated within the program's `ARCHITECTURE` part. We allow only restarting, since it enables radiation discharging without complicating the language with other options. PEARL is a real-time programming language, hence, memory operations, e.g. garbage collection, are forbidden. In principle, compiler hints for swapping would be allowed, but we prefer not to intervene into these processes. Finally, there is a need for a hand-coded initialisation procedure based on program states of the other Byzantine processors.

## 4.2 Criteria of Effectiveness

We conclude our paper with several measures on how to obtain quantitative results to demonstrate the effectiveness of the proposed approach, see Table 5. The coverage of IEC regulations can be derived from column *IEC 61508* of Table 4 for fault tolerance methods. For general language constructs, PEARL-2020 is as well-suited as other safety-related languages like MISRA-C, but no other language provides such powerful fault-tolerance language constructs. Thus, PEARL-2020 has a higher coverage than state-of-the-art languages. The IEC regulations enforce functional safety. The programmers, hence, are relieved from checking them explicitly and may concentrate on program validation and verification. Likewise, official certification authorities require less effort. Another criterion of effectiveness would be to compare accident statistics from programs developed with PEARL-2020 and other safety-related languages. We want to address this issue in future work.

**Table 5** Criteria of effectiveness.

| | |
|---|---|
| | main goal: functional safety |
| ↗ | coverage of IEC regulations (see column *IEC 61508* of Table 4) |
| ↗ | seamless implementation of various functional safety concepts, programmers do not need to address them explicitly |
| ↘ | programming effort and more effortless verification |
| ↘ | examination effort for certification authorities [22] |
| ? | accident statistics for machines and plants that are programmed with PEARL-2020 compared to other languages like MISRA-C (future work) |
| ↗ | maintainability and debuggability |
| | side constraints: resources |
| ↗ | length of source code |
| → | memory for application data |
| → | runtime |

We would like to juxtapose general program metrics of PEARL-2020 to other languages in a qualitative fashion: The length of PEARL-2020 source code is expected to become longer as many diverse checks are demanded. Memory requirements are equivalent to that of other safety-related languages because, even today, all devices implement diverse memory in agreement with IEC regulations. Runtime should remain at the same level.

**Acknowledgement.** We thank Daniela Horn for thoroughly proofreading the manuscript and the anonymous reviewers for their invaluable comments.

## References


1 DIN 66253 Part 3. *PEARL for Distributed Systems*. Beuth, 1989.

2 IEC 60848. *GRAFCET Specification Language for Sequential Function Charts*. IEC, 2013.

3 IEC 61508-3. *Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 3: Software requirements*. IEC, 2010.

4 DIN 66253-2. *PEARL-90*. Beuth, 1998.

5 Paul Ammann and John C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Trans. Computers*, 37(4):418–425, 1988. doi:10.1109/12.2185.

6 Sohel Anwar, editor. *Fault Tolerant Drive By Wire Systems: Impact on Vehicle Safety and Reliability*. Bentham, 2011. doi:10.2174/9781608053070112010l.

7 John Barnes. *High Integrity Ada – The SPARK Approach*. Addison-Wesley, 1997.

8 Juliane Benra and Wolfgang A. Halang, editors. *Software-Entwicklung für Echtzeitsysteme*. Springer, 2009. URL: http://www.springer.com/de/book/9783642015953.

9 William Bolton. *Mechatronics: Electronic Control Systems in Mechanical and Electrical Engineering*, volume 3. Prentice Hall, 2004.

10 Josef Börcsök. *Funktionale Sicherheit*. VDE, 4th edition, 2014. URL: https://www.vde-verlag.de/buecher/483590/funktionale-sicherheit.html.

11 Franck Cappello, Henri Casanova, and Yves Robert. Checkpointing vs. migration for post-petascale supercomputers. In *39th International Conference on Parallel Processing, ICPP 2010, San Diego, California, USA, 13-16 September 2010*, pages 168–177. IEEE Computer Society, 2010. doi:10.1109/ICPP.2010.26.

12 Vittorio Castelli, Richard E. Harper, Philip Heidelberger, Steven W. Hunter, Kishor S. Trivedi, Kalyanaraman Vaidyanathan, and William P. Zeggert. Proactive management of software aging. *IBM Journal of Research and Development*, 45(2):311–332, 2001. doi:10.1147/rd.452.0311.

13 Henan Chen, Yongduan Song, and Danyong Li. Fault-tolerant tracking control of fw-steering autonomous vehicles. In *2011 Chinese Control and Decision Conference (CCDC)*, pages 92–97, May 2011. doi:10.1109/CCDC.2011.5968152.

14 Matjaž Colnarič and Domen Verber. Dealing with tasking overload in object oriented real-time applications design. In *6th Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2001), 8-10 January 2001, Rome, Italy*, pages 214–222. IEEE Computer Society, 2001. doi:10.1109/WORDS.2001.945133.

15 Li DanYong and Song YongDuan. Adaptive fault-tolerant tracking control of 4ws4wd road vehicles: A fully model-independent solution. In *Chinese Control Conference (CCC)*, volume 31, pages 485–

492. IEEE, July 2012. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6389978.

16 Leberecht Frevert. Lösung von Echtzeitproblemen mit PEARL90-Objekten, 1998. URL: http://www.real-time.de/service/misc/GrundlagenOOP.pdf.

17 Kevin Fu. Trustworthy medical device software. *Public Health Effectiveness of the FDA 510(k) Clearance Process – Measuring Postmarket Performance and Other Selected Topics*, 2011. URL: http://www.nap.edu/read/13020/chapter/10.

18 Arthur Gelb, editor. *Applied Optimal Estimation*. MIT Press, 1974. URL: https://mitpress.mit.edu/books/applied-optimal-estimation.

19 GI-Working Group 4.4.2 "Real-Time Programming, PEARL". *PEARL 90 Language Report*, September 1998. Version 2.2. URL: http://www.real-time.de/service/misc/PEARL90-LanguageReport-V2.2-GI-1998-eng.pdf.

20 Julian Godesa and Robert Hilbrich. Framework für die empirische Bestimmung der Ausführungszeit auf Mehrkernprozessoren. In Wolfgang A. Halang, editor, *Funktionale Sicherheit, Echtzeit 2013, Fachtagung des gemeinsamen Fachausschusses Echtzeitsysteme von Gesellschaft für Informatik e.V.(GI), VDI/VDE-Gesellschaft für Mess- und Automatisierungstechnik (GMA) und Informationstechnischer Gesellschaft im VDE (ITG), Boppard, 21. und 22. November 2013*, pages 77–86. Springer, 2013. doi:10.1007/978-3-642-41309-4_9.

21 Wolfgang A. Halang and Matjaž Colnarič. Dealing with exceptions in safety-related embedded systems. In *15th IFAC World Congress*, pages 983–988. Elsevier, 2002. doi:10.3182/20020721-6-ES-1901.00985.

22 Wolfgang A. Halang and Rudolf M. Konakovsky. *Sicherheitsgerichtete Echtzeitsysteme*. Springer, 2013. URL: http://www.springer.com/de/book/9783642372971.

23 Wolfgang A. Halang and Rudolf J. Lauber. *Echtzeitsysteme I*. FernUniversität Hagen, 2009.

24 Wolfgang A. Halang and Janusz Zalewski. Programming languages for use in safety-related applications. *Annual Reviews in Control*, 27(1):39–45, 2003. doi:10.1016/S1367-5788(03)00005-1.

25 F. Hubert. *Handbuch der Raumfahrttechnik*, volume 4, chapter Datenmanagement. Hanser, 2011.

26 Farnam Jahanian and Aloysius K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Trans. Software Eng.*, 12(9):890–904, 1986. doi:10.1109/TSE.1986.6313045.

27 Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982. doi:10.1145/357172.357176.

28 Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006. doi:10.1109/MC.2006.180.

29 Jane W. S. Liu, Kwei-Jay Lin, Riccardo Bettati, David Hull, and Albert Yu. Use of imprecise computation to enhance dependability of real-time systems. In Gary M. Koob and Clifford G. Lau, editors, *Foundations of Dependable Computing: Paradigms for Dependable Applications*, pages 157–182. Springer US, Boston, MA, 1994. doi:10.1007/978-0-585-27316-7_6.

30 Reinhard Maier, Günther Bauer, Georg Stöger, and Stefan Poledna. Time-triggered architecture: A consistent computing platform. *IEEE Micro*, 22(4):36–45, 2002. doi:10.1109/MM.2002.1028474.

31 Peter Marwedel. *Embedded Systems Design*. Springer, 2006. URL: http://www.springer.com/us/book/9789400702561.

32 Rainer Müller and Marcel Schaible. Die Programmierumgebung OpenPEARL90. In Wolfgang A. Halang and Herwig Unger, editors, *Industrie 4.0 und Echtzeit – Echtzeit 2014, Fachtagung des gemeinsamen Fachausschusses Echtzeitsysteme von Gesellschaft für Informatik e.V.(GI), VDI/VDE-Gesellschaft für Mess- und Automatisierungstechnik (GMA) und Informationstechnischer Gesellschaft im VDE (ITG), Boppard, 20. und 21. November 2014*, Informatik Aktuell, pages 31–40. Springer, 2014. doi:10.1007/978-3-662-45109-0_4.

33 Paula Prata and João Gabriel Silva. Algorithm based fault tolerance versus result-checking for matrix computations. In *Digest of Papers: FTCS-29, 29th Annual International Symposium on Fault-Tolerant Computing, Madison, Wisconsin, USA, June 15-18, 1999*, pages 4–11. IEEE Computer Society, 1999. doi:10.1109/FTCS.1999.781028.

34 Parameswaran Ramanathan. Fault-tolerance in real-time control applications using *(m, k)*-firm guarantee. In *Digest of Papers: FTCS-27, 27th Annual International Symposium on Fault-Tolerant Computing, Seattle, Washington, USA, June 24-27, 1997*, pages 132–141. IEEE Computer Society, 1997. doi:10.1109/FTCS.1997.614086.

35 B. Randell. System structure for software fault tolerance. *ACM SIGPLAN Notices – International Conference on Reliable Software*, 10(6):437–449, April 1975. doi:10.1145/390016.808467.

36 Charles Preston Shelton. *Scalable Graceful Degradation for Distributed Embedded Systems*. PhD thesis, Carnegie Mellon University, jun 2003. URL: https://users.ece.cmu.edu/~koopman/thesis/shelton.pdf.

37 Paulo Baltarejo Sousa, Konstantinos Bletsas, Eduardo Tovar, Pedro Souto, and Benny Akesson. Unified overhead-aware schedulability analysis for slot-based task-splitting. *Real-Time Systems*, 50(5-6):680–735, 2014. doi:10.1007/s11241-014-9204-x.

38 Jürgen J. Stoll. *Fehlertoleranz in verteilten Realzeitsystemen: Anwendungsorientierte Techniken*, volume 236 of *Informatik-Fachberichte*. Springer, 1990.

39 Dwight Sunada, David Glasco, and Michael J. Flynn. Multiprocessor architecture using an audit trail for fault tolerance. In *Digest of Papers: FTCS-29, 29th Annual International Symposium on Fault-Tolerant Computing, Madison, Wisconsin, USA, June 15-18, 1999*, pages 40–47. IEEE Computer Society, 1999. doi:10.1109/FTCS.1999.781032.

40 Matthias Tichy and Holger Giese. Extending Fault Tolerance Patterns by Visual Degradation Rules. In *2005 Workshop on Visual Modeling*

*for Software Intensive Systems (VMSIS) at the the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, Texas, USA*, pages 67–74, September 2005. URL: http://www.upb.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/2005/TG05.pdf.

41 Tjerk W. van der Schaaf and L. Kanse. *Human Error and System Design and Management*, chapter Errors and Error Recovery, pages 27–38. Number 253 in Lecture Notes in Control and Information Sciences. Springer, 2000. URL: http://www.springer.com/us/book/9781852332341.

42 Hongyu Sun Zaipeng Xie and Kewal Saluja. A survey of software fault tolerance techniques, 2006. URL: http://www.pld.ttu.ee/IAF0030/Paper_4.pdf.