Failure Mode and Effect Analysis for Safety-Critical Systems with Software Components

Tadeusz Cichocki [1], Janusz Górski [2]

[1] Adtranz Zwus, Modelarska 12, 40-142 Katowice, Poland,
tadeusz.cichocki@pl.adtranz.com
[2] Technical University of Gdańsk, Narutowicza 11/12, 80-952 Gdańsk, Poland,
jango@pg.gda.pl

Abstract. One of possible ways to achieve a very high level of confidence in a system is to develop its adequate model and then to analyse the properties of this model. The paper presents how object oriented modelling extended with formal specifications is used to support FMEA of software intensive systems. The paper refers to the case study of a computerised railway signalling system.

1 Introduction

As a result of the progress in technology development an increasing number of applications include a software component which can directly affect risks associated with the system. In some applications, e.g. rail and air transportation, medical, nuclear, a very high level of confidence that the system will not do any harm has to be achieved before the system is commissioned to use. The criteria to be fulfilled and the guidance how to do this are provided by numerous national, international and sectoral standards and regulations, e.g. [5], [6], [4].

One of possible ways of achieving a very high level of confidence in the system is to develop its adequate model and then to analyse the properties of this model. If the model is a valid representation of the problem under consideration, the properties of the model can be understood as the properties of the real system. To increase confidence in the analytical results, the model and the associated analytical framework are often formal, i.e. are represented in terms of mathematical objects.

In this paper we demonstrate how object oriented modelling extended with formal notations, namely Z [20] and CSP [17] are used to model a problem related to computer based railway signalling in order to support FMEA (*Failure Mode and Effect Analysis*) [2], [3] of this system. Section 2 refers to the related works. Section 3 gives a rationale for the presented approach. Sections 4 and 5 demonstrate how object-oriented modelling was used to our case study system. Section 6 give more detail on performing FMEA on this system. The conclusions are given in section 7.

2 Related Works

Identification of the system architecture, modelling of functional dependencies between identified components, developing the inventory of their potential faults, and identification of error propagation scenarios initiated by those faults are the basic activities of FMEA [11]. The technique is commonly used for mechanical and/or electrical equipment. Its direct application to software encountered some difficulties, however, and required some extensions and modifications. Present standards, e.g. IEC 812 [11] and EN 50128 [5] recommend application of FMEA to software. When applied to software, FMEA is often known under the name SEEA - *Software Error Effect Analysis*. IEC 812 stresses limits in the scope of the analyses while including software. FMEA is also recommended for hardware parts of systems to extend requirements for the embedded software [14], [12]. A general approach of performing SEEA was presented in [15] where software was specified by data flows. The method aims at introducing safety constraints early in software development. Data paths are checked, potential errors are systematically searched for in the specifications, and when discovered, are eliminated or mitigated by applying proper fault avoidance or fault tolerance approaches. In [16] and [7] some further extensions of SEEA were proposed as well as suggestions on how SEEA can be integrated with other safety techniques.

Our approach differs from the previous works in two respects. Firstly, we choose object oriented modelling as the base for system description on the assumption that this provides for including both, software and hardware aspects in a common modelling framework. Secondly, we investigate a possibility to apply formal methods in order to support a systematic search for failure modes and to analyse the impact of failures.

3 Overview of the Approach

We have chosen the object-oriented framework to represent the system architecture. The advantages of the object orientation in the context of FMEA analysis include:

- provision of a common framework within which various system components can be represented (software, hardware, people),
- stress on univocal system decomposition and component independence (precisely written ‘contracts’ for object interfaces),
- covering static, dynamic and functional aspects of the system (e.g. [19], [18]),
- flexibility and stability: the models relatively easily can be modified, and modifications are usually localised in a limited number of objects,
- “smooth” transition from analysis to design and code: the objects identified during the analysis stage have their direct representation in implementation.

To provide for precision and unambiguity we extended the object-oriented model with its formal specification. Following [8] and [10] we have represented the object structure, object states, object operations and system state invariants in terms of Z schemas [20] and the dynamic aspects of object co-operation by using the CSP [17] notation. Within this framework we model faults and then analyse the consequences

of those faults. By a *fault* we mean any deviation from desired properties of objects or their interaction, including disturbances of event timing and precedence. Faults may manifest themselves as *failures* of components, represented as a negation of the component function's desired result (its post-condition) or negation of any other assumption made while specifying the desired behaviour of the component (e.g. the pre-condition of the function or the state invariant of the component). In the CSP-based specification we represent them as additional *fault events*. Consequently the specification is extended by the cases handling the fault events occurrences.

Like in the standard FMEA, we develop *complete* lists of component failures. Two *domains of failure* are analysed and, respectively, two tables are constructed, consistent with the usual classification of defects in software [1], [13]:

- *Data Table,* involving communication failures and input deviations, used to analyse data dependencies and software interface errors, and
- *Event Table,* involving process failures and constraints on software states, used to analyze the effects of failures and deviations in outputs, possibly caused by software that fails to function correctly.

The general procedure of FMEA, based on the formal specifications, includes the activities to:

- develop an object-oriented model of the system,
- formalise the components of the model and its structure,
- develop an inventory of possible component failures,
- argue on the completeness of the list of failures,
- relate the effects of anticipated lower level faults to the higher level assumptions and failure lists.

In the subsequent sections we demonstrate some of those steps in reference to the Line Block System case study.

4 Line Block System – Object Model

The Line Block System (LBS) is supposed to control railway traffic on a rail track between two stations. It uses light signals shown on semaphores, each protecting a line segment of the rail track. Each segment is continuously tested in order to determine its occupation or its availability to move a train toward the next segment. The main goal is then to maintain separation of trains on the track while maintaining smooth passing of trains in the required direction.

An object-oriented model of LBS is shown in Fig.1. At the highest abstraction level, the LBS system is composed of the *operator*, rail track *sectors* and *trains*. Rail track sectors are further specialised from the point of view of an arbitrary sector (called *block*) and divided into sectors which are before (*units_before*) and after (*units_next*) the block. The block is further decomposed into train *detecting* device, train *signalling* device and *Line Block Controller* (LBC). The association *Line_Block_Control_Rules* describes the dependencies between the objects, reflecting the basic rules defined by the Polish State Railways (PKP) regulations. For instance, one of them requires that:

Safety Requirement	Each occupied line block must be protected by the 'STOP' signal displayed on its protecting SIGNALLING_DEVICE

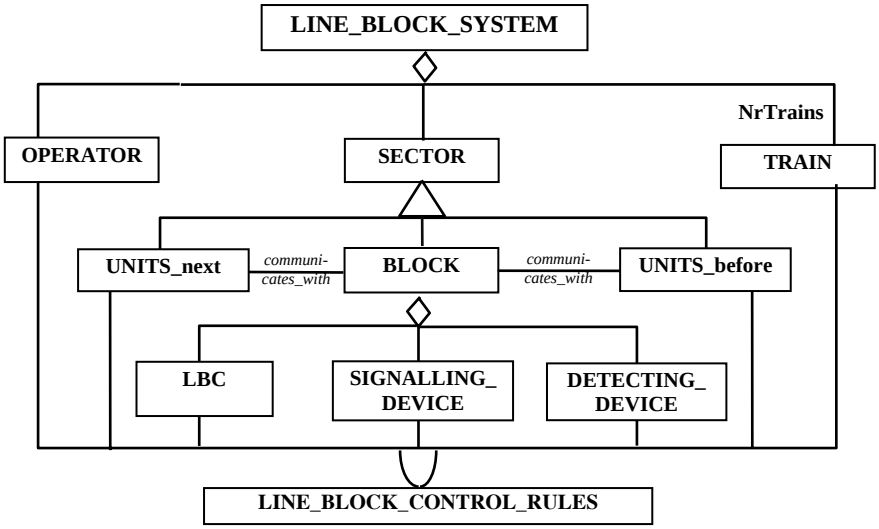

Fig.1. LINE_BLOCK_SYSTEM object structure.

The above rules and the properties of objects of Fig.1 are expressed using the Z notation. Because of the space limitations we just recall a sample of it.

Safety_Requirement

LB_Traffic_direction : {-1, 0, 1} -- '1' corresponds to the traffic
DETECTING_DEVICE -- direction '*from right to left*',
SIGNALLING_DEVICE -- '-1' to the opposite direction,
-- and '0' denotes '*no traffic direction defined*'

(DETECTING_DEVICE.Detection = YES & LB_Traffic_direction = 1) ⇒
SIGNALLING_DEVICE.Signalisation_1 ∈ {S0, S1}

Safety_Requirement schema defines a state invariant of the object BLOCK.

Comment. The attribute *DETECTING_DEVICE.Detection* indicates if a block is occupied by a train. *LB_Traffic_direction = 1* means that the traffic direction is set to '*from right to left*' and the attribute *SIGNALLING_DEVICE.Signalisation_1* restricts possible signals that can be displayed to trains in such situation.

5 Line Block System – Dynamic Model

Each object is specified as a CSP process. As a train is passing through subsequent rail track sectors, the system state changes in response to the signals from train detectors. Fig.2 shows the event channels between rail track sectors and the channels from and to train detectors and semaphores.

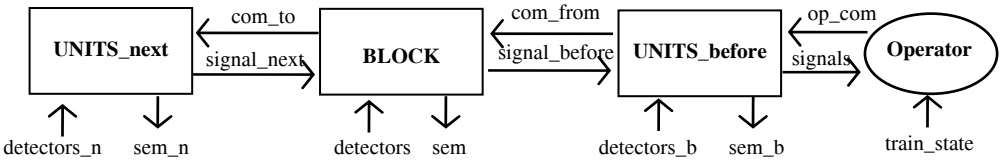

Fig. 2. Communication channels between objects.

A formal specification of the behaviour of the BLOCK object of Fig.2 is given below. The text prefixed by '--' is a comment.

```
-- BLOCK process Specification
channel signal_before : {S0, S1, S2, S3, S4, S5}
channel signal_next : {S0, S1, S2, S3, S4, S5}
   -- Signals {S0, S1}, called 'STOP' signals, refuse the
   -- permission to move a train into the considered block
   -- ('S0' means 'dark' and 'S1' means 'red' ),
   -- and signals of {S2, S3, S4, S5} grant this permission.
channel sem : {S0, S1, S2, S3, S4, S5}
   -- shows signals on the semaphore protecting the block.
channel com_from, com_to
   -- carry the operator's commands (not developed here).
channel detectors : {IN, OUT}
   -- signals from train detectors

   -- The BLOCK process' alphabet is:
B = { |signal_before, signal_next, sem, detectors| }

BLOCK =
   detectors.IN →
       ( sem.S1 → signal_before.S1 → BLOCK
         [] sem.S0 → signal_before.S1 → BLOCK )
   [] detectors.OUT → BLOCK_not_occupied

BLOCK_not_occupied =
   signal_next.S0 → sem.S1 → signal_before.S1 → BLOCK
   []  signal_next.S1 → sem.S5 → signal_before.S5 → BLOCK
   []  signal_next.S2 → sem.S2 → signal_before.S2 → BLOCK
   []  signal_next.S3 → sem.S2 → signal_before.S2 → BLOCK
   []  signal_next.S4 → sem.S3 → signal_before.S3 → BLOCK
   []  signal_next.S5 → sem.S3 → signal_before.S3 → BLOCK
```

If the detector indicate the presence of a train in a block, the block's semaphore should signal S1 ('red light'). Otherwise the block considers the situation in the following unit (detected through the *signal_next* channel). The sequence: *detectors.IN* → *sem.S1* represents the BLOCK's desired (perfect) behaviour. We model *deviations* from this behaviour as visible events, offered by the object and activated by the environment. For instance, the specification includes the case for *sem.S0* (the 'dark' signal instead of the 'red' one). This specific case is classified by the railway regulations as a non-critical failure.

As it is shown in Fig.1, BLOCK is decomposed into its composite objects: LBC, SIGNALLING_DEVICE and DETECTING_DEVICE. In Fig.3 we show the communication structure that results from this decomposition.

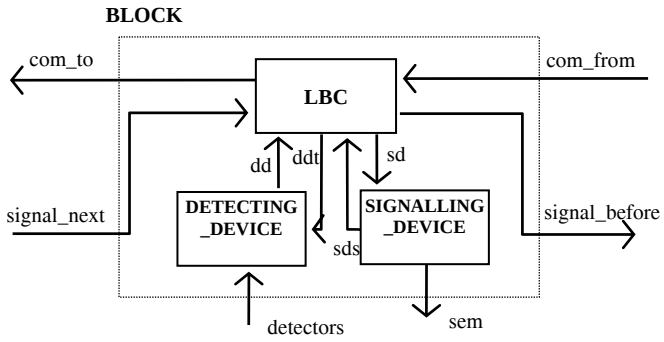

Fig. 3. Communication structure of the BLOCK component.

The formal specification of the components of BLOCK follows.

```
-- LBC process Specification
channel signal_before : {S0, S1, S2, S3, S4, S5}
channel signal_next : {S0, S1, S2, S3, S4, S5}
channel com_from, com_to
   -- carry the operator's commands (not developed here).
channel dd : {IN, OUT}
   -- detecting device's output
channel ddt : {IN}
   -- DETECTING_DEVICE state test
channel sd : {S0, S1, S2, S3, S4, S5}
   -- setting the signalling device view
channel sds : {S0, S1, S2, S3, S4, S5}
   -- the signalling device state confirmation

   -- The LBC process' alphabet is:
L = {|signal_before, signal_next, dd, ddt, sd, sds|}

LBC = ddt.IN →
   ( dd.IN → sd.S1 → sds.S1 → signal_before.S1 → LBC
   [] dd.OUT → LBC_not_occupied )
```

```
LBC_not_occupied =
signal_next.S0 → sd.S0 → sds.S0 → signal_before.S1 → LBC
[] signal_next.S1 → sd.S5 → sds.S5 → signal_before.S5 → LBC
[] signal_next.S2 → sd.S2 → sds.S2 → signal_before.S2 → LBC
[] signal_next.S3 → sd.S2 → sds.S2 → signal_before.S2 → LBC
[] signal_next.S4 → sd.S3 → sds.S3 → signal_before.S3 → LBC
[] signal_next.S5 → sd.S3 → sds.S3 → signal_before.S3 → LBC

-- DETECTING_DEVICE process Specification
channel detectors : {IN, OUT}
channel dd : {IN, OUT}
channel ddt : {IN}

   -- The DETECTING_DEVICE process' alphabet is:
D = { |detectors, dd, ddt| }

DETECTING_DEVICE =
   detectors.IN → ddt.IN → dd.IN → DETECTING_DEVICE
   [] detectors.OUT  → ddt.IN → dd.OUT → DETECTING_DEVICE

-- SIGNALLING_DEVICE process Specification
channel sem : {S0, S1, S2, S3, S4, S5}
   -- the signals shown by the signalling device
   -- (on the semaphore protecting the block)
channel sd : {S0, S1, S2, S3, S4, S5}
channel sds : {S0, S1, S2, S3, S4, S5}

   -- The SIGNALLING_DEVICE process' alphabet is:
S = { |sem, sd, sds| }

SIGNALLING_DEVICE =
   sd.S0 → sem.S0 → sds.S0 → SIGNALLING_DEVICE
   [] sd.S1 →
       ( sem.S1 → sds.S1 → SIGNALLING_DEVICE
       []
       sem.S0 → sds.S1 → SIGNALLING_DEVICE )
       -- failure of the sinchronisation on sem.S1
       -- by environmental choice
   [] sd.S2 → sem.S2 → sds.S2 → SIGNALLING_DEVICE
   [] sd.S3 → sem.S3 → sds.S3 → SIGNALLING_DEVICE
   [] sd.S4 → sem.S4 → sds.S4 → SIGNALLING_DEVICE
   [] sd.S5 → sem.S5 → sds.S5 → SIGNALLING_DEVICE
```

The BLOCK' composed of LBC, DETECTING_DEVICE and SIGNALLING_DEVICE processes is modelled as follows:

```
  BLOCK' = DETECTING_DEVICE
                   | [D∩L] | LBC
                          | [L∩S] | SIGNALLING_DEVICE
```

The desired relation between the BLOCK specification (more abstract) and BLOCK' specification (more detailed), expressed in terms of the usual trace semantics of CSP processes [17] is as follows:

$$\texttt{BLOCK} \sqsubseteq \texttt{BLOCK'} \setminus \{dd, ddt, sd, sds\}$$

i.e. the more detailed specification should be consistent with the more abstract one. This verification step is to be performed with the help of tools.

6 Performing FMEA

6.1 Analysing data failures

Systematic review of component failures is supported by a checklist of generic component faults. From the data modelling point of view we distinguish:

- a change of an attribute value outside its type range,
- a change of an object invariant.

To illustrate the first case, let us consider the following Z specification of BLOCK.

BLOCK

Traffic_direction : {-1, 0, 1}
Occupation : {YES, NO}
Signalisation_1: {S0, S1, S2, S3, S4, S5}
Signalisation_-1: {S0, S1, S2, S3, S4, S5}

Traffic_direction = 1 ⇒
((Occupation = YES ⇒ Signalisation_1 ∈ {S0, S1})
& (Occupation = NO ⇒ Signalisation_1 ∈ {S2, S3, S4, S5}))
& Signalisation_-1 ∈ {S0, S1}

While analysing possible deviations of attribute values we refer to the interpretation of the attribute in the real world. For instance, for the attribute *Occupation,* its expected range of values is {YES, NO}. However the experience shows that we should also consider a value 'Y/N' corresponding to the train detection problems. Similarly, knowing that the attribute *Signalisation_1* represents possible 'legal' combinations of semaphore lights, from the physical implementation we can realise that also other, not 'legal' combinations, e.g. 'red+green', are possible and therefore should be considered. Those extensions of attribute range that are assessed as being likely (although undesired) should be then included in further analyses.

The second case is concerned with state invariants. Below we show the failure condition resulting from contradicting the *Safety_Requirement* (a state invariant) given in Section 5. By definition, a violation of such property (being a part of the top level *Safety specification*) constitutes a hazardous state of the system. For instance,

the contradiction of the *Safety_Requirement* of Section 5 means that a train occupies a given rail track segment (*DETECTING_DEVICE.Detection = YES*) and at the same time the protecting semaphore's state is different than 'STOP' (compare Table 1). Protection against such situation has to be implemented in the LBC, DETECTING_DEVICE and SIGNALLING_DEVICE objects. Violation of properties of components can impact safety requirements in indirect ways. Systematic search of the specification aims at discovering the impact the violating a property of a component has on fulfilment of the specified safety requirements of the system (note that contradicting a component invariant does not necessarily implies that a safety requirement is contradicted; some components are mission oriented and can have no impact on safety). The next step is to consider possible modifications of the specification that can decrease/remove this impact.

Table 1. State invariant failure condition

Condition	Consequence
DETECTING_DEVICE.Detection = YES & LB_Traffic_direction = 1 & SIGNALLING_DEVICE.Signalisation_1 $\notin$ {S0, S1}	**Hazard:** signals other then S0 or S1 could be interpreted as *permission* signals

6.2 Analysing communication failures

An occurrence of a communication between two objects (e.g. one object invokes an operation of another object) is the result of simultaneous execution by the CSP processes representing those objects of the matching communication commands. From the point of view of BLOCK, the train detectors read the state of 'reality' (the rail track state, in this case) and invoke a BLOCK operation associated with the *detectors* channel. This operation, let us call it *Monitor* (as it monitors the state of the rail track) assumes that the signals received from the environment consistently represent the situation on the rail track (the operations' *precondition*). It also assumes that the events signalled through the channel belong to the predefined domain (the *guard* condition of the operation). Finally, as the result of the execution of the operation, some visible effects are observed on the interface of BLOCK (this is specified by the *post-condition* associated with the operation).

The specification of the *Monitor* operation is given below. The operation is invoked through the *detectors* channel and its effects are visible through the *sem* channel.

```
pre Monitor =
    ('occupied rail track generates detectors.IN event and
    non-occupied, detectors.OUT event')
-- This condition is not formalised here.
guard Monitor = ( detectors : {IN, OUT} )
post Monitor = ( ( detectors.IN ⇒ sem.[S0, S1] )
          & ( detectors.OUT  ⇒ sem.[S2, S3, S4, S5] ) )
```

Successful communication requires the conjunction of three conditions:

post *op_sending* & **guard** *op_receiving* & **pre** *op_receiving.*

The list of component failures (together with the impact of the failures) is represented in the following FMEA table:

Table 2. The structure of FMEA table

	PROCESS: *name*		
Case	**Channel**	**Fault assumption**	**Corresponding deviations of …**
(1)	(2)	(3)	(4)
	INPUT Channels	¬ **guard** *op* ∨ ¬ **pre** *op* (*Data Table support*)	*availability and correctness of operation's input data*
	OUTPUT Channels	¬ **post** *op* (*Event Table support*)	*correctness of operation's performance and data transformation*

In the case of *Monitor* operation the input conditions to be considered include:

```
¬ pre Monitor =     ('non-occupied rail track        (case 1a)
                    generates detectors.IN event or
                    occupied, detectors.OUT event')  (case 1b)
¬ guard Monitor =   ¬ ( detectors : {IN, OUT} )
                    = detectors.I/O                  (case 1c)
```

The results of the analysis are shown in Table 3.

Table 3. FMEA table for the BLOCK object (an extract)

	PROCESS: **BLOCK**		
Case	**Channel**	**Fault assumption**	**Corresponding deviations of …**
(1)	(2)	(3)	(4)
1a	detectors	detectors.IN - *failed interpretation of the rail segment non-occupation state*	sem.[S0, S1] - 'STOP' signals, according to the *perfect* specification - **system availability lost.**
1b		detectors.OUT - *failed interpretation of the rail segment occupation state*	sem.[S2, S3, S4, S5] - one of the *permission* signals - **hazard state.**
1c		detectors.I/O - *nonidentified rail track state* (*new environment offer*)	**design decision**: to extend specification by *detectors.I/O* occurrence and consequences.
2a	sem	*successful* detectors.IN *and failed output on* ***sem***	sem.[S2, S3, S4, S5] - one of the *permission* signals - **hazard state.**

Using the information included in the above table we can modify the specification of BLOCK in order to prevent or reduce the risk associated with identified hazardous situations. An example modification follows (the extensions are shown in bold):

```
channel detectors : { IN, OUT, I/O }
BLOCK = detectors.IN →
             ( sem.S1 → signal_before.S1 → BLOCK
             []
             sem.S0 → signal_before.S1 → BLOCK )
        []
        detectors.OUT → BLOCK_not_occupied
        []
        detectors.I/O → sem.S1 → signal_before.S1 → BLOCK
```

On a more detailed level (shown in Fig.3) this extension also requires an extension of the specification of BLOCK's components, to resolve the *detectors.I/O* event acceptance in DETECTING_DEVICE and to show how LBC handles the event *dd.I/O*.

7 Conclusions

The paper presents application of FMEA to a software intensive safety related system. One of our objectives was to extend FMEA in such a way that it can be applied to software without significant changes in the overall method. The core of the proposed extension is to base the FMEA process on formalised object oriented specifications. This way we can better handle the problems that are software specific. Another objective is to fit the proposed method into present engineering practice in the company, the railway signalling systems developer. The project is still in progress and presently is performed in parallel to the traditional forms of analysis and documentation practised in the company. What can be already concluded about the proposed approach is:

- at the first stages of the project we achieved a significant concentration on relevant details and a strong developer guidance to the problems and solutions,
- the formal notation forces designers to concentrate on system structure and properties at the same time focusing attention on a proper abstraction level,
- the formal specifications support systematic search for possible deviations and inconsistencies, and support the analysis of their consequences,
- the proposed notations significantly improve precision, completeness and compactness of the specifications. Although it needed some training in formal notations, the approach could be used in an everyday engineering practice.

As the specifications grow in size a tool support is unavoidable. This is presently one of our concerns and we are examining various possibilities of choosing from the existing tools (e.g. the FDR tool of Formal Systems (Europe) Ltd. [17], [9]).

References

1. Barbacci, B. R., Klein, M. H., Weinstock, C. B.: Principles for Evaluating the Quality Attributes of a Software Architecture, Software Engineering Institute, Carnegie Mellon University. Technical Report, CMU/SEI-96-TR-036, March 1997
2. Cichocki, T., Górski, J.: Safety assessment of computerized railway signalling equipment. Proc. of CENELEC SC9XA/WGA10 Workshop, Münich (Germany), May 11, 1999
3. Cichocki, T., Górski, J.: Safety assessment of computerized railway signalling equipment supported by formal techniques. Proc. of FMERail Workshop #5, Toulouse (France), September, 22-24, 1999
4. Defence Standard 00-55: Requirements for Safety Related Software in Defence Equipment (Part 1&2), Issue 1, UK Ministry of Defence, 1997
5. EN 50128: Railway applications. Software for Railway Control and Protection Systems, CENELEC, Final Draft version, July 1998
6. ENV 50129: Railway applications. Safety Related Electronic Systems for Signalling, CENELEC, May 1998
7. Fenelon P., McDermid J. A., Nicholson M., Pumfrey D. J., Towards Integrated Safety Analysis and Design. ACM Applied Computing Review, 2(1), pp. 21-32, 1994
8. Fischer, C.: Combining CSP and Z, Univ. of Oldenburg. Technical Report, TRCF-97-1
9. Formal Systems (Europe) Ltd.: *Failures-Divergence Refinement, FDR2 User Manual*, 24 October 1997
10. Heisel, M.: Methodology and Machine Support for the Application of Formal Techniques in Software Engineering, Habilitation Thesis, Technische Universität Berlin, Berlin, 1997
11. IEC 812 (1985): Procedure for failure mode and effects analysis (FMEA), TC56
12. J-C. Laprie, B. Littlewood, Quantitative Assessment of Safety-Critical Software: Why and How? Predictably Dependable Computing Systems (PDCS) Technical Report no. 45, ESPRIT BRA Project 3092, February 1991
13. Lutz, R. R., Woodhouse, R. M.: Requirements Analysis Using Forward and Backward Search. (Annals of Software Engineering, 1997) JPL California Institute of Technology Technical Report, May 2, 1997
14. NASA-GB-A201, NASA Software Assurance Guidebook, September 1989
15. E. Noe-Gonzales, The Software Error Effect Analysis and the Synchronous Data Flow Approach to Safety Software: Method, Results, Operational Lessons. Proc. of SAFECOMP'94, pp. 163-171
16. Papadopoulos Y., McDermid J., Sasse R., Heiner G., Analysis and Synthesis of the Behaviour of Complex Programmable Electronic Systems in conditions of Failure. Reliability Engineering and System Safety Journal (forthcoming, 2000), Elsevier Science Limited (*an extension of SAFECOMP'99 paper*)
17. Roscoe, A. W.: The Theory and Practice of Concurrency, Prentice Hall, 1998, ISBN 0-13-674409-5, pp. xv + 565
18. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W.: *Object-Oriented Modelling and Design*, Prentice-Hall Int., 1991
19. D'Souza, D., Wills, A. C.: Objects, Components, and Frameworks with UML. The Catalysis Approach, Addison Wesley Longman, Inc. 1998
20. Spivey, J. M.: The Z Notation: A Reference Manual, First published by Prentice Hall International (UK) Ltd., 1992 (Second edition), ISBN 0-13-629312-3