Formal Support for Fault Modelling and Analysis

Tadeusz Cichocki [1] and Janusz Górski [2]

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

Abstract. The paper presents how CSP and the associated tool FDR are used to support FMEA of a software intensive system. The paper explains the basic steps of our approach (formal specification, systematic fault identification, fault injection experiments and follow-up) and gives some results related to the application of this method to the industrial case study, a railway signalling system that is presently under development.

1 Introduction

Success of a safety-critical system development project depends to large extent on the designer's ability to include in the design adequate defences against a justifiably complete class of faults and to prove the correctness of the design against the normative rules and requirements. This objective is supported by the FMEA (*Failure Mode and Effect Analysis*) method. The method recognises the system components structure and admits that components' failures can affect higher level (system) properties. FMEA assumes that component faults are identified in a systematic way, investigates the effects of those faults on the system properties and then, if necessary, the findings are taken into account in the following design decisions. It has been argued [1, 3, 4] that, for software intensive systems, FMEA can benefit if it is supported by a formal method – this way we can increase precision and remove ambiguities from the analyses.

In [1] a general context of our research is presented. There and in this paper we investigate the application of the CSP [5] notation as a formal method supporting the FMEA process. Our presentation refers to the Line Block System industrial case study that is presently under development. The present paper shows how we use CSP and the associated tool FDR [2] to identify component faults and to analyse the consequences those faults can have in the higher-level system components. Our method comprises the following main steps:

- formal specification of the system and its components,
- systematic fault identification referring to the formal specification,
- fault consequences analysis through fault injection experiments,
- follow up: fault acceptance or specification redesign.

The steps are explained in the subsequent sections of the paper.

2 The Line Block System Case Study

Line Block System (LBS) is a railway signaling system presently under development at Adtranz Zwus in Katowice, Poland. In [1] it has been shown how we used object-oriented approach to develop a model of this system. The model is hierarchical and represents the system architecture on subsequent levels of increasing detail. Fig.1 presents the collaboration diagram [6] of LBS. It shows the objects of the system and the channels of their co-operation. The BLOCK object supervises a single sector of the rail track. It communicates with train detectors (det) and semaphores (sem). It also communicates with similar blocks on its left and right (represented by the UNIT_next and UNIT_before objects).

In Fig.1 we also show the internal structure of BLOCK (included in the dotted rectangle). This is a more refined level of the model where the higher level object is represented in terms of its components. BLOCK is shown as being composed of LBC, DETECTING_DEV and SIGNALLING_DEV. Input and output channels of BLOCK are inputs and outputs of its component objects. In addition, the components can exchange signals that are not visible outside BLOCK (are internal to BLOCK). *Test*, *confirm*, *dd* and *sd* are examples of those.

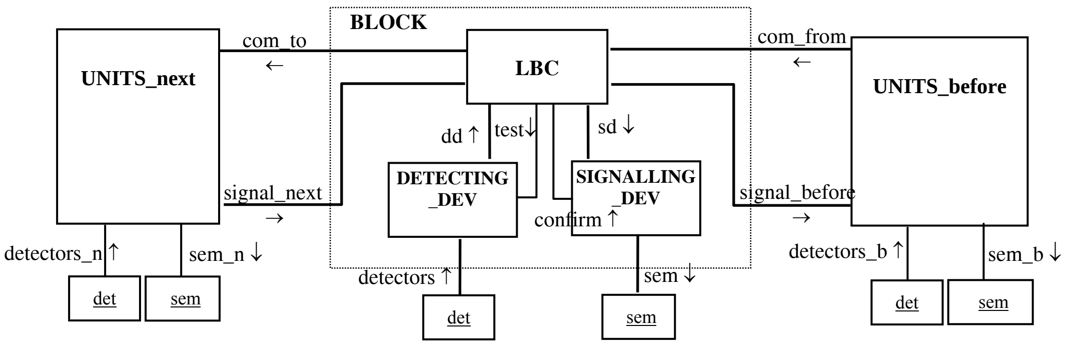

Fig. 1. Line Block System collaboration diagram.

The advantage object-orientation is that the model closely follows the actual structure of the system and therefore is easily understood by system designers and implementors. Hierarchical modelling supports the distinction between the design and implementation oriented views of the system. This provides a framework within which we can analyse possible influences the lower level components can have on their higher level "containers", which is the essence of FMEA.

3 Formal specification and verification

To provide for precision and unambiguity we apply formal specifications to our models. Our choice was CSP [5] as this notation provides for specification of objects interacting through communication channels. The CSP objects (processes) interact with each other and the environment by means of communication events (instantaneous atomic actions). The CSP expressions describe patterns of event causality and the way the cooperating processes synchronize. The synchronization is achieved on specific events and may involve data exchange between processes.

A CSP process is observed by the traces (the sequences of events) the process can engage in. For a process P, we define *alpha*(P) as the set of all events P is able to synchronize on and *traces*(P) as the set of all possible finite traces of P. A *failure* of a process is a (finite) trace together with a *refusal set* which is a set of events that the process might refuse to engage in after performing the trace (note that if this set equals *alpha*(P) then P deadlocks – refuses to engage in any event). For a process P, *failures*(P) denotes the set of all failures of P. The *divergences* of a process P, denoted *divergences*(P), is a set of traces after which P may diverge, i.e. perform an infinite sequence of internal (invisible) events.

Three models of process behaviour are considered: *traces* (T), *failures* (F) and *failures-divergences* (FD). In the *traces* model, P is characterised by *traces*(P) and, by definition, the *traces refinement relation* between two processes P and Q (denoted P [T= Q) holds iff *alpha*(P) = *alpha*(Q) and *traces*(Q) $\subseteq$ *traces*(P).

In the *failures* model, P is characterised by the *failures*(P) set and the *failures refinement relation* between two processes, P and Q (denoted P [F= Q) holds iff *alpha*(P) = *alpha*(Q) and *failures*(Q) $\subseteq$ *failures*(P).

In the *failures-divergences* model, P is characterised by the *failures*(P) and *divergences*(P) pair of sets and the *failures-divergences refinement relation* between two processes, P and Q (denoted P [FD= Q) holds iff *alpha*(P) = *alpha*(Q), *failures*(Q) $\subseteq$ *failures*(P) and *divergences*(Q) $\subseteq$ *divergences*(P).

Note that for divergence free processes, the relations [FD= and [F= are equivalent.

To specify our processes we use the CSP dialect supported by the FDR (*Failures Divergence Refinement*) [2] tool. This gives rise for subsequent application of FDR as an analytical tool. Using FDR we can verify various properties of the specifications, including:

- deadlock freedom – a process never enters a state where there is no possibility of continuation (execution of the next event),
- divergence freedom – a process never enters a state where an infinite sequence of internal events is possible without any external event occurrence,
- semantic relations of processes – verification of the *traces refinement, failures refinement* and *failures-divergences refinement* relations CSP between processes.

Below we enclose formal specifications of BLOCK and its components. To save space we omit the specification of communication channels and concentrate on the behaviours of the objects. The specification of BLOCK slightly differs from this presented in [1]. Here we additionally distinguish a safe state of BLOCK (BLOCK_safe_state) and in the mission oriented part of BLOCK specification (BLOCK_interlocking) we explicitly handle a case of possible detector malfunction (the *detectors.INO* clause). In the specification we refer to five standard signals defined for a line-block system: S1 – S5 that are displayed on semaphores.

```
  -- BLOCK
BLOCK = BLOCK_interlocking |~| BLOCK_safe_state
-- The symbol |~| denotes the internal choice operator.
-- Behaviour of BLOCK in its two basic states is schown.
BLOCK_interlocking = detectors.IN -> BLOCK_occupied
       [] detectors.OUT -> BLOCK_not_occupied
       [] detectors.INO -> BLOCK_occupied
-- The symbol [] denotes the external choice operator

BLOCK_occupied = sem.S1 -> signal_before.S1 -> BLOCK
       |~| sem.S0 -> signal_before.S1 -> BLOCK
-- Possible failure of the synchronization on sem.S1;
-- S1 is 'red' (stop) signal and S0 is a dark one.

BLOCK_not_occupied =
signal_next.S0 -> sem.S5-> signal_before.S5 -> 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
-- Implementation of the signalling interlocking rules

BLOCK_safe_state = sem.S1 -> signal_before.S6 -> STOP
       |~| sem.S0 -> signal_before.S6 -> STOP
-- The BLOCK's fail safe state.
```

The above specification gives all possible traces of events that can be observed at the BLOCK interface. The specification can be submitted to the FDR tool in order to verify some of its properties. Examples of assertions to be validated are given below:

```
assert BLOCK :[deadlock free]
assert BLOCK :[divergence free]
```

Positive validation of the above assertions means that the BLOCK specification is deadlock and divergence free.

At the more refined level the model explains the internal structure of BLOCK (see the dotted rectangle in Fig.1). Again, we use CSP to specify the BLOCK components. The specifications follow (again, the declarations of channels are omitted).

```
  -- LBC
-- Line block controller
LBC = test.interlocking -> LBC_interlocking
       |~| test.safe_state -> LBC_safe_state

LBC_interlocking =
        dd.IN -> LBC_occupied
       [] dd.OUT -> LBC_not_occupied
       [] dd.INO -> LBC_occupied

LBC_safe_state = sd.S6 ->
             (confirm.S1 -> signal_before.S6 -> STOP
             [] confirm.S0 -> signal_before.S6 -> STOP)
```

```
LBC_occupied = sd.S1 ->
       (confirm.S1 -> signal_before.S1 -> LBC
       [] confirm.S0 -> signal_before.S1 -> LBC)

LBC_not_occupied =
signal_next.S0 -> sd.S5 -> confirm.S5 ->
    signal_before.S5 -> LBC
[] signal_next.S1 -> sd.S5 -> confirm.S5 -> signal_before.S5 ->
    LBC
[] signal_next.S2 -> sd.S2 -> confirm.S2 -> signal_before.S2 ->
    LBC
[] signal_next.S3 -> sd.S2 -> confirm.S2 -> signal_before.S2 ->
    LBC
[] signal_next.S4 -> sd.S3 -> confirm.S3 -> signal_before.S3 ->
    LBC
[] signal_next.S5 -> sd.S3 -> confirm.S3 -> signal_before.S3 ->
    LBC

  -- DETECTING_DEV
-- Device detecting presence of a train in the BLOCK
DETECTING_DEV =
       test.interlocking -> DETECTING_DEV_interlocking
       [] test.safe_state -> STOP

DETECTING_DEV_interlocking =
       detectors.IN -> dd.IN -> DETECTING_DEV
       [] detectors.OUT -> dd.OUT -> DETECTING_DEV
       [] detectors.INO -> dd.INO -> DETECTING_DEV

  -- SIGNALLING_DEV
-- Device signalling states of the BLOCK to trains.
SIGNALLING_DEV =
       sd.S1 -> SIGNALLING_occupied
       [] sd.S2 -> SIGNALLING_not_occupied(S2)
       [] sd.S3 -> SIGNALLING_not_occupied(S3)
       [] sd.S4 -> SIGNALLING_not_occupied(S4)
       [] sd.S5 -> SIGNALLING_not_occupied(S5)
       [] sd.S6 -> SIGNALLING_safe_state
-- S6 is a special purpose signal used in LBS.
SIGNALLING_safe_state =
       sem.S1 -> confirm.S1 -> STOP
       |~| sem.S0 -> confirm.S0 -> STOP
       -- Failure of the synchronization on sem.S1
SIGNALLING_occupied =
       sem.S1 -> confirm.S1 -> SIGNALLING_DEV
       |~| sem.S0 -> confirm.S0 -> SIGNALLING_DEV
       -- failure of the synchronization on sem.S1
SIGNALLING_not_occupied(spar) =
       sem.spar -> confirm.spar -> SIGNALLING_DEV
```

Having specified the components we formally declare that together they form BLOCK_IMP – the implementation of BLOCK.

```
  -- BLOCK_IMP
BLOCK_IMP = DETECTING_DEV
[|{|dd,test|}|] (LBC [|{|sd,confirm|}|] SIGNALLING_DEV)
```

A standard step to be performed now is to compare the specifications of BLOCK and BLOCK_IMP in order to verify if they are consistent. This can easily be done with the help of FDR. The assertions to be validated are given in Table 1 below.

Table 1. Verification conditions

Name	Refinement condition
R1	`BLOCK [T= BLOCK_IMP \ {|dd, test, sd, confirm|}`
R2	`BLOCK [FD= BLOCK_IMP \ {|dd, test, sd, confirm|}`
R3	`BLOCK_IMP \ {|dd, test, sd, confirm|} [T= BLOCK`
R4	`BLOCK_IMP \ {|dd, test, sd, confirm|} [FD= BLOCK`

The tool reports the positive result by showing (according to the FDR convention) *green tick* ✓ before each of the assertions.

4 Systematic Fault Identification

Let us consider a CSP specification of an object A. Consider all possible deviations of the interface events from their specifications preventing the object's synchronisation with its environment. The deviations include modification of the external events set and modifications of the channel types. Each deviation is then assessed concerning the likelihood of its occurrence in a real system. Those deviations that are positively validated are then included in the Fault Table of the object A. We call them *syntactic faults*.

Another deviations that we consider are those that affect the causality pattern of the object behaviour. We consider possible event scenarios that are inconsistent with the object's internal state (are not implied by the state), but result in synchronisation between A and its environment. Such deviations include events which inconsistency can not be detected by the co-operating components. We consider all possibilities of such events and then assess the likelihood of their occurrence in the real system. Those that are positively validated are included in the Fault Table as well. We call them *semantic faults*.

The analysis of the BLOCK object proceeded as follows.

Possible syntactic and semantic faults of the component objects were generated from their specifications. The analysis covered all possible violations for each channel. The faults were subjected to the validation argumentation (to assess the likelihood of their occurrence) and then documented in the corresponding Fault Table. The following tables, Table 2, Table 3 and Table 4, contain examples of the faults of the components of BLOCK.

Table 2. Fault Table of DETECTING_DEV (extract)

Name	Fault description	
	Normal synchronization	**Faulty synchronization**
DV1	`detectors.IN -> dd.IN`	`detectors.IN -> dd.OUT`
DV2	`detectors.INO -> dd.INO`	`detectors.INO -> dd.OUT`

DV3	`detectors.INO -> dd.INO`	`detectors.INO -> dd.IN`
DV4	`detectors.OUT -> dd.OUT`	`detectors.OUT -> dd.IN`

Table 3. Fault Table of SIGNALLING_DEV (extract)

Name	Fault description	
	Normal causality	**Faulty causality**
SV1	`Sd.SB4 -> SIGNALLING_not_occupied(S4)`	`sd.SB4 -> SIGNALLING_not_occupied(S3)`
SV2	`Sd.SB5 -> SIGNALLING_not_occupied(S5)`	`sd.SB5 -> SIGNALLING_not_occupied(S4)`

Table 4. Fault Table of LBC (extract)

Name	Fault description	
	Normal synchronization or causality	**Faulty synchronization or causality**
LV1	`dd.IN`	Declaration of *dd* channel type extension by INOX and simulation of `dd.INOX`
LV2	`Signal_next.S5 -> sd.SB3`	`signal_next.S5 -> sd.SB4`

5 Fault Injection Experiments

Each fault included in the Fault Tables is then subjected to what we call a *fault injection experiment*. Such experiment includes two steps: (1) fault injection and (2) fault consequences analysis.

The fault injection step involves introducing changes to the specification. For syntactic faults it may require changes in the declarations of channel types and some redesign of the component interfaces.

The fault consequence analysis step is performed with the support of the FDR tool. The aim of the analysis is to verify if (and how) a given fault can violate the specification at the higher level (the specification of BLOCK, in our case study). The list of verification conditions for BLOCK is given in Table 1.

The results of the fault injection experiments for the faults of the Tables 2, 3 and 4 are documented in Tables 5, 6 and 7, respectively. The *red cross and dot* mark ×• (the convention of FDR) denotes that the check has been completed and (at least) one counter-example for the condition in question has been found. The dot • shows that the counter-example is available through the *debug* option of FDR.

Table 5. The results for the DETECTING_DEV faults.

Reference specification: **BLOCK** Component: **DETECTING_DEV**		
Fault name	**The result of FDR check**	
DV1	R1: ×• and R2: ×• BLOCK_IMP performs: _tau test.interlocking detectors.IN dd.OUT signal_next.S2	R3: ×• and R4: ×• BLOCK performs: _tau detectors.IN _tau sem.S1

DV2	R1: ×• and R2: ×• BLOCK_IMP performs: _tau test.interlocking detectors.INO dd.OUT signal_next.S2	R3: ×• and R4: ×• BLOCK performs: _tau detectors.INO _tau sem.S1
DV3	R1: ✓ and R2: ✓	R3: ✓ and R4: ✓
DV4	R1: ×• and R2: ×• BLOCK_IMP performs: _tau test.interlocking detectors.OUT dd.IN sd.SB1 _tau sem.S1	R3: ×• and R4: ×• BLOCK performs: _tau detectors.OUT signal_next.S2

Table 6. The results for the SIGNALLING_DEV faults.

Reference specification: **BLOCK** Component: **SIGNALLING_DEV**		
Fault name	**The result of FDR check**	
SV1	R1: ✓ and R2: ✓	R3: ✓ and R4: ✓
SV2	R1: ×• and R2: ×• BLOCK_IMP performs: _tau test.interlocking detectors.OUT dd.OUT signal_next.S0 sd.SB5 sem.S4	R3: ×• and R4: ×• BLOCK performs: _tau detectors.OUT signal_next.S0 sem.S5

Table 7. The results for the LBC faults.

Reference specification: **BLOCK** Component: **LBC**		
Fault name	**The result of FDR check**	
LV1	R1: ✓ and R2: ✓	R3: ✓ and R4: ✓
LV2	R1: ×• and R2: ×• BLOCK_IMP performs: _tau test.interlocking detectors.OUT dd.OUT signal_next.S5 sd.SB4 sem.S1	R3: ×• and R4: ×• BLOCK performs: _tau detectors.OUT signal_next.S5 sem.S3

Each fault injection experiment that is not positively validated by the FDR run is then analysed to find out the nature of the detected inconsistency. Of great help here are the counterexamples provided by the tool as they help to identify event scenarios that led to failures. For example, for the LV2 fault of Table 7, the faulty implementation of BLOCK_IMP performs the following sequence of external events (as shown by a trace for R1 and R2 in Table 7): `detectors.OUT -> signal_next.S5 -> sem.S1,` whereas the allowed sequence for BLOCK is as follows (shown by the trace for R3 and R4 in Table 7): `detectors.OUT -> signal_next.S5 -> sem.S3`. The analysis of such case leads, in general, to the following decisions:

- Acceptance: we accept the (negative) consequences of the fault. However a message is passed to the designers to increase efforts towards lowering the likelihood of the fault occurrence (e.g. by choosing more reliable technologies),
- Redesign: the present design of the system is changed in order to eliminate the negative consequences of possible occurrence of the fault (for instance, fault detection mechanism and safe state enforcement).

The analyses were performed using FDR ver. 2.66 running under the Red Hat LINUX operating system on a PC with INTEL Pentium III 600 MHz processor. The full specification of BLOCK comprised 35 lines of CSP/FDR code. The total size of DETECTING_DEV, SIGNALLING_DEV and LBC comprised 60 lines of CSP/FDR code. The total processing time used for the analyses was 12 hours (13 minutes for one fault). The total numbers of faults considered for the component objects are given in Table 8.

Table 8. The numbers of faults considered during analyses

Component name	**Number of faults**
LBC	21
DETECTING_DEV	10
SIGNALLING_DEV	25

6 Conclusions

The potential of formal methods to support safety analysis of software intensive systems has been well recognised [3, 4]. However, their industrial application still faces several difficulties. One of those is the complexity of formal analysis which, if not supported by powerful tools, quickly goes out of control. The advent of matured tools that can be used by engineers opens a new prospect for application of formal methods in the safety industry. This however requires investigation of possible usage patterns and finding ways the formal methods can support the techniques and methods widely applied by safety engineers.

In the paper we presented a case study of using CSP and the associated FDR tool to support FMEA of a safety related system. The tool was used to support fault injections into specifications and analysis of their consequences. FMEA seems to be well suited to be supported by a formal method because it assumes hierarchical decomposition of the system. Thanks to that the scope of formal analysis can be restricted and focuses on investigation of the relationship between the adjacent layers in the hierarchy. Consequently, the complexity of the formal analysis is limited even if the system under consideration is relatively large.

The models of our approach are developed top-down, starting from the system requirements and going down to the components that are implemented in software or hardware. The analysis goes bottom-up. We identify possible faults and then analyse their consequences in the higher levels of the system structure. In order to be able to pre-select faults (in order to focus only on those that really matter) we have to assess the candidate faults concerning the likelihood of their occurrence. This is achieved by referring to the knowledge that comes from outside of the formal framework (e.g. the component failure profiles, assessment of the technologies used to implement a component etc.).

The method is based on specifications and does not help projects with missing specifications. However, the process of building formal specifications can help in early identification of omissions and inconsistencies in specifications.

The documentation of the process of the formal FMEA can be (and it is) used as a crucial part of the safety case argumentation. It supports explicit enumeration of faults and the visibility of the following decisions (being it fault acceptance or system redesign). The help of the tool in finding an example fault consequence is also appreciated.

References

[1] T. Cichocki and J. Górski, *Failure Mode and Effect Analysis for Safety-Critical Systems with Software Components*, in: Springer Lecture Notes in Computer Science, vol. 1943, 2000, pp. 382-394

[2] Formal Systems (Europe) Ltd., *Failures-Divergence Refinement, FDR2 User Manual*, 24 October 1997.

[3] N. G. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley Publishing Company (680 pp), 1995, ISBN 0-201-11972-2.

[4] J. D. Reese, *Software Deviation Analysis*, University of California, Irvine, PhD Thesis, 1996.

[5] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice-Hall, 1998 (580 pp), ISBN 0-13-674409-5.

[6] OMG *Unified Modeling Language Specification*, Version 1.3, June 1999.