Modelling Fault Tolerance of Transient Faults

Dubravka Ilic and Elena Troubitsyna

Åbo Akademi, TUCS, Department of Computer Science,
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland
{Dubravka.Ilic, Elena.Troubitsyna}@abo.fi

Abstract. In this paper we focus on analysis of transient physical faults and designing mechanisms to tolerate them. Transient faults are temporal faults that appear for some time and might disappear and reappear later. They are common in control systems. However transient fault appearing even for a short time might result in a system error. Hence fault tolerance mechanisms for detecting and recovering from temporal faults are of great importance in the design of control systems. Often the system module which detects errors and performs error recovery is called a Failure Management System. Its purpose is to prevent the propagation of errors in the system. In this paper we propose a formal approach to specifying the Failure Management System in the B Method. We focus on deriving a general specification and development pattern for Failure Management Systems for tolerating transient faults.

1 Introduction

Nowadays software-intensive control systems are in heart of many safety-critical applications. Hence dependability of such systems is a great concern. While designing controlling software for such systems we should ensure that it is able not only to detect errors in system functioning but also to confine the damage and perform error recovery. In this paper we focus on designing controllers able to withstand transient physical faults of the system components [9]. Transient faults are temporal defects within the system. We focus on analysis and design of a special subsystem of control systems – a Failure Management System (further referred to as FMS) – which performs error detection, damage confinement and error recovery. The FMS is a subsystem of the embedded control system responsible for providing the controller with the error free inputs obtained from the environment. Since controller is relying only on the input from FMS, it is important to ensure its correctness.

Design of the FMS is particularly difficult since often requirements changes are introduced at the late stages of the development cycle. These changes are unavoidable since many requirements result from empirical performance studies executed under failure conditions. To overcome this difficulty we propose a formal pattern for specifying fault tolerance mechanism in the FMS. The contribution of our work is in verifying the suggested pattern rather then a particular specification. The proposed pattern can be reused in the product line development and hence its correctness is crucial.

We demonstrate how to develop the FMS by stepwise refinement in the B Method [3]. Our approach is validated by a realistic case study conducted within EU project RODIN [7].

2 Fault tolerance mechanism in FMS

Failure Management System (FMS) [2] is a part of the embedded control system responsible for managing failures of the system inputs as shown on Figure 1.

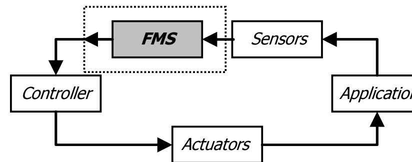

Figure 1. Place of the FMS in an embedded control system

The main role of FMS is to supply the controller of the system with the error free inputs from the system environment.

All inputs supplied to the FMS are analysed. The analysis of each input results in invocation of the corresponding remedial action. There are three categories of remedial actions: healthy, temporary or confirmation actions. If an input is considered to be error free, it is forwarded unchanged to the controller. This is a *healthy* system action. If an error is detected, the input gets suspected and the FMS decides on error recovery. The aim of FMS is to give error free output even when input is in error, i.e., during recovery phase. Hence, when the input is suspected, the system sends the last good value of the input as the error free output toward the controller. This is a *temporary* system action. In the recovery phase the input can get recovered during certain number of operating cycles. If the input fails to recover, the *confirmation* action is triggered and the system becomes frozen.

In Figure 2 we illustrate the behaviour of FMS over one analogue input.

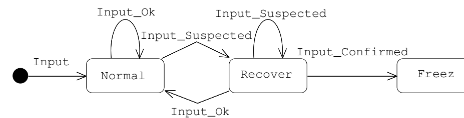

Figure 2. Specification of the FMS behaviour

A general description of FMS behaviour is as follows: after getting the input from the environment through the system sensors, the FMS determines whether the input is in error or error free. If the input is error free, the FMS applies healthy remedial action. If it is in error, it is classified as suspected and the system initiates recovery phase. When the recovery starts, a counting mechanism responsible for ensuring the recovery termination is triggered. If after recovery the input is still suspected, the con-

firmation action is applied, i.e., the input is confirmed failed and the system freezes. Otherwise, the system considers the input again as error free, applies the healthy action and continues the operation without any interruption.

The general description of FMS behaviour lacks, however details about the error detection.

When an input is received by FMS, FMS performs certain tests on the inputs to determine its status: in error or error free. We differentiate between the individual and collective tests. Individual tests (e.g., `Test1` and `Test2` in Figure 3) are obligatory for each input and they determine the preliminary abnormality in the input. When triggered, individual tests run solely based on the input reading from the sensor. We use two kinds of individual tests: the magnitude test and the rate test. In the magnitude test the input is compared against some predefined limit (bound) and if exceeds, it is considered in error. The rate test is detecting erroneous input while comparing the change of the input readings in consecutive cycles. Namely, the current value of the input is compared against the previous input value and if some predefined limit is exceeded, the input is considered in error. It is obvious that both tests have some preconfigurations expressed through the predefined limits which allow dynamic test changes as appropriate.

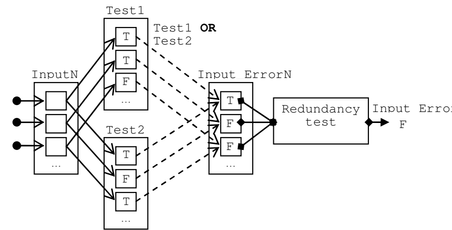

Figure 3. Introducing error detection

The error detection for multiple sensors (`InputN` in Figure 3) implies first the application of individual tests and then, when these tests are passed, the collective test is applied. The collective test is commonly a redundancy test. It is applied on the group of multiple sensor inputs. As presented on the Figure 3, redundancy test takes the detected multiple inputs (`Input_ErrorN`) and based on their values (`TRUE` or `FALSE`) votes for the input status (`Input_Error`). This status becomes `TRUE` (i.e., the input is considered in error) if there are more erroneous inputs for the multiple sensor readings then error free ones. When the input status is finally detected, FMS proceeds with the corresponding remedial actions.

Before presenting our formal pattern for handling fault tolerance in FMS, we give the short introduction to the B Method.

3 Formal system modelling in the B Method

In this paper we have chosen the B Method [3] as our formal modelling framework. The B Method is an approach for the industrial development of correct software. The

method has been successfully used in the development of several complex real-life applications [6]. The tool support available for B, for instance - Atelier B [1], provides us with the assistance for the entire development process.

In this paper we adopt event-based approach to system modelling [4]. The events are specified as the guarded operations `SELECT cond THEN body END`. Here `cond` is a state predicate, and `body` is a B statement describing how state variables are affected by the operation. If `cond` is satisfied, the behaviour of the guarded operation corresponds to the execution of its `body`. If `cond` is false at the current state then the operation is disabled, i.e., cannot be executed. Event-based modelling is especially suitable for describing reactive systems. Then `SELECT` operation describes the reaction of the system when particular event occurs.

For describing the computation in operations we used following B statements:

Statement	Informal meaning
`X := e`	Assignment
`IF P THEN S1 ELSE S2 END`	If `P` is true then execute `S1`, otherwise `S2`
`S1 || S2`	Parallel execution of `S1` and `S2`
`X :: T`	Nondeterministic assignment – assigns variable `x` arbitrary value from given set `T`

The last statement allows for abstract modelling and hence, postponing implementation decisions till later development stages.

The development methodology adopted by B is based on stepwise refinement [8]. While developing a system by refinement, we start from an abstract formal specification and transform it gradually into an implementable program by a number of correctness preserving steps, called refinements. In the refinement process we reduce non-determinism of the original specification and eventually arrive at deterministic implementable specification.

The result of a refinement step in B is a machine called `REFINEMENT`. Its structure coincides with the structure of the abstract machine. However, refined machine should contain an additional clause `REFINES` which defines the machine refined by the current specification. Besides definitions of variable types, the invariant of the refinement machine should contain the refinement relation. This is a predicate which describes the connection between state spaces of more abstract and refined machines.

To ensure correctness we should verify that initialization and each operation preserve the invariant. Verification can be completely automatic or user-assisted.

Next we demonstrate how to formally specify failure management system described in Section 2.

4 Formal development of FMS

4.1 Specifying the failure management system

Control systems are usually cyclic, i.e., their behaviour is essentially an interleaving between the environment stimuli and controller reaction on these stimuli. The control-

ler reaction depends on whether the FMS has detected error in the obtained input. Hence, it is natural to consider the behaviour of FMS in the context of the overall system.

The FMS gets certain inputs from the environment, applies specific detection mechanisms and depending on the detection results produces output to the controller or freezes the whole system. Inputs that FMS receives from the environment are inputs from various sensors. In this paper we consider only analogue sensors. In absence of errors the output from the FMS is the actual input to the controller. However, if error is detected the FMS should try to tolerate it and produce the error free output or to stop the system without producing any output at all.

In our abstract specification given in Figure 5, for modelling fault tolerance on given input we used different variables. The variable `FMS_State` defines the phases of control cycle execution. Its values are as follows: `env` – obtaining inputs from the environment, `det` – detecting erroneous inputs, `act` – changing the system operating mode, `rcv` – recovering of the erroneous input, `out` – supplying the output of the FMS to the controller, `stop` – freezing the system. The variable `FMS_State` models the evolution of system behaviour in the operating cycle. At the end of the operating cycle the system finally reaches either the terminating (freezing) state or produces the error free output. After the error free output was produced, the operating cycle starts again. Hence, the behaviour of the FMS can be described as in Figure 4.

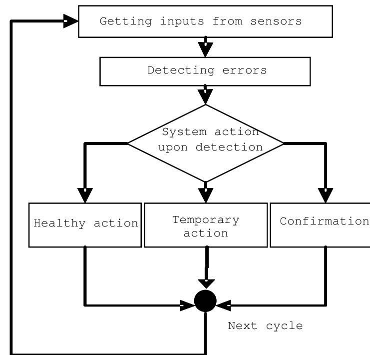

Figure 4. Behaviour of the FMS

Since the controller relies only on the input from the FMS, we should guarantee that it obtains the error free output from the FMS. Hence, our safety invariant expresses this: whenever the input is confirmed failed, the FMS output is not produced (i.e., `Input_Status=confirmed => FMS_State=stop`) and, whenever the input is confirmed `ok`, the output should have the same value as input or be different if the input is `suspected` (i.e., `(Input_Status=ok => Output=Input) & (Input_Status=suspected => Output/=Input))`.

Although the abstract specification of FMS is highly abstract it anyway specifies the fault tolerance mechanism allowing us to ensure the desired behaviour of the system. In this abstract specification the input values produced by the environment are modelled nondeterministically. After getting the inputs, FMS performs detection on

inputs to determine if they are in error or error free. This is modelled in the *Detection* operation of the FMS machine as a nondeterministic assignment of some boolean value (`TRUE` or `FALSE`) to the variable modelling input state (i.e., `Input_Error :: BOOL`). After the input state is detected, FMS triggers the healthy action if the input is error free. If the input is in error, FMS initiates temporary action, i.e., error recovery.

```
MACHINE
    FMS
SEES
    Global
VARIABLES
    Input, Input_Error, FMS_State, cc, num
INVARIANT
    Input : T_INPUT &       /*actual input to the FMS*/
    Input_Error : BOOL &    /*variable modelling input
                              status*/
    FMS_State : STATES &    /*variable modelling system
                              state*/
    cc : NAT &              /*cc and num are counters*/
    num : NAT &
    <safety invariant>
INITIALISATION
    FMS_State :=env || cc:=0 || num:=0
OPERATIONS

    Environment=
    SELECT <the system is functioning normally>
    THEN
           <nondeterministically choose some input> ||
           FMS_State:=det
    END;

    Detection=
    SELECT <the system is in the detection state>
    THEN
           Input_Error :: BOOL || FMS_State:=act
    END;

    Action=
    SELECT <the input is not in error>
    THEN
           <healthy action > || FMS_State:=out
    WHEN
           <the input is in error and the
            error is just discovered>
    THEN
           <input is marked as suspected> ||
           cc:=cc+xx || num:=num+1 || FMS_State:=rcv
    WHEN
           <the input is not in error but it is already
            marked suspected>
    THEN
           <input stays suspected> ||
           cc:=cc-yy || num:=num+1 || FMS_State:=rcv
    WHEN
           <the input is in error and it is already
            marked suspected>
    THEN
           <input stays suspected> ||
           cc:=cc+xx || num:=num+1 || FMS_State:=rcv
    END;

    Return=
    SELECT <healthy action>
    THEN
           <input is passed to the output> ||
           FMS_State:=env
    WHEN
           <temporary action>
    THEN
           <output is assigned the last good
            value of the input> || FMS_State:=env
    END;

    Recovering=
    SELECT <input is suspected> & (num>=Limit or cc>=zz)
    THEN
           <input confirmed failed> || FMS_State:=stop
    WHEN
           <input is suspected> & num<Limit & cc=0
    THEN
           <input has recovered> || FMS_State:=out
    WHEN
           <input is suspected> & num<Limit & cc/=0 &
           cc<zz
    THEN
           FMS_State:=env
    END;

    Stopping=
    SELECT FMS_State=stop
    THEN
           skip
    END
END
```

Figure 5. Excerpt from the abstract FMS specification

Error recovery is modelled by introducing the two counters: `cc` and `num`. At the beginning of the operating cycle, both counters are set to zero and their values are changed only in the recovery phase. The first counter `cc` counts inputs which are in error. While the system is in the recovery phase, every time when the obtained input is found in error, the system sets as the output the last good value of the input and the counter `cc` is incremented by some given value `xx`. However if the input is error free, the `cc` is decremented by the given value `yy`. In each operating cycle system is setting some values for the counter `cc` either by decrementing or incrementing it. If at one point the value of the `cc` exceeds some predefined limit `zz` the counting stops and the system confirms the input failure by terminating the operation and freezing the system. Since each erroneous input increments the value of `cc` and each error free input decrements it, eventually the counter `cc` is set to zero. This is possible if eventually the FMS starts to receive error free inputs. If `cc` reaches zero the input is considered to be recovered and the system returns to normal functioning initializing `cc` to zero and making it thus ready for the next recovering cycle. The way `cc` reaches zero or exceeds the limit `zz` is determined via setting the parameters `xx`, `yy` and `zz`. These parameters are set by observing the real performance of the failure. By setting the

value of xx higher then the value of yy, the counter cc is going to yield the limit zz faster. However, such a specification is insufficient for guaranteeing termination of recovery. Observe that the input may vary in such a way that the counter cc is practically oscillating between some values but never reaching the limit zz or zero. Hence, we introduce the second counter num which is counting each recovering cycle. When some allowed limit for num is exceeded, the recovery terminates and if cc is different than zero the input is confirmed failed.

Our initial specification completely describes the intended behaviour of the FMS but leaves the mechanism of detecting errors in input unspecified. Next, we demonstrate how to obtain the detailed specification of error detection in the refinement process.

4.2 Refining error detection in FMS

Since we observe multiple sensors the refinement of the FMS starts with replacing the Input variable with the InputN variable modelling the sequence of input values received by the FMS as N sensor readings, instead of only one sensor reading. The nondeterministic assignment of value to the variable Input_Error in the *Detection* operation of the abstract machine is further refined. By introducing new variable Input_ErrorN we can set the value for each particular sensor reading. Input_ErrorN is a sequence with Boolean values TRUE or FALSE. These values are determined for each multiple sensor input by running two detection tests: the magnitude test and the rate test. If the input passes the magnitude test, the value of the temporary variable Input_Error1 is set to FALSE, otherwise is TRUE (i.e., the test on this input failed). Similarly, if the input passes the rate test, the value of the temporary variable Input_Error2 is set to FALSE, otherwise TRUE.

The input is error free if none of these tests fail. Hence we define the status of the input as the disjunction of Input_Error1 and Input_Error2 and set the variable Input_ErrorN accordingly.

After setting the values of the variable Input_ErrorN in described way, we apply the redundancy test (as shown in Figure 3). We consider N sensor readings which values are stored in introduced variable InputN. Moreover, our assumption is that this number is odd to prevent the situation in which the number of the erroneous and error free inputs is the same. The status of each one of the N sensor inputs is recorded in the variable Input_ErrorN. The redundancy test performs majority voting. It means that if there are more values TRUE in the Input_ErrorN sequence, the whole input is considered failed, otherwise it is error free. After the status of the input is detected, FMS makes a decision how to proceed with handling it, i.e., which action it is going to apply as specified in the abstract specification.

The essence of our refinement step is to introduce modelling of the N sensor inputs instead of only one and replace the nondeterministic assignment to the variable Input_Error with deterministic error detection. The refinement relation for this step is as follows:

```
(Input_Error=TRUE =>
(card(Input_ErrorN|>{TRUE}) > card(Input_ErrorN|>{FALSE})))
```

The above refinement relation establishes connection between the abstract variable `Input_Error` and the concrete variable `Input_ErrorN`. Namely, if the value of `Input_ErrorN` is such that the number of error free inputs is smaller then the number of erroneous inputs then it should correspond to the value `TRUE` of `Input_ErrorN`.

To produce the final output, FMS calculates the median value of all error free inputs and passes it as the output from the FMS.

In the Figure 6 we give the excerpt from this refinement step of the FMS with introduced error detection.

```
REFINEMENT
    FMSR1
REFINES
    FMS
SEES
    Global
VARIABLES
    InputN, Input_Error, Input_Error1, Input_Error2,
    Input_ErrorN,
    FMS_State,
    cc,num,
    Passed1, Passed2
INVARIANT
    InputN : seq(T_INPUT) & /*N sensor input reading*/
    Input_Error : BOOL &
    Input_Error1 : BOOL &   /*test results for 1 input*/
    Input_Error2 : BOOL &
    Input_ErrorN : seq(BOOL) & /*test results for
                                 N sensor inputs*/
    FMS_State : STATES &
    cc : NAT & num : NAT &
    Passed1 : BOOL &           /*variables for modeling
                                 test application*/
    Passed2 : BOOL &
    <safety and gluing invariants>
INITIALISATION
    InputN := [] || Input_Error := FALSE ||
    Input_Error1 := FALSE || Input_Error2 := FALSE ||
    Input_ErrorN := [] ||
    FMS_State := env ||
    cc := 0 || num:=0 ||
    Passed1 := FALSE || Passed2 := FALSE

OPERATIONS

    <obtaining the input from the environment>

    Detection=
    SELECT <magnitude test not passed yet>
    THEN
        IF
           <the input is in defined low and high limits>
        THEN
           Input_Error1:=FALSE
        ELSE
           Input_Error1:=TRUE
        END ||
        Passed1:=TRUE ||
        FMS_State:=det
    WHEN
        <rate test not passed yet>
    THEN
        IF
           <the input change exceeds the limit>
        THEN
           Input_Error2:=TRUE
        ELSE
           Input_Error2:=FALSE
        END ||
        Passed2:=TRUE ||
        FMS_State:=det
    WHEN
        <both test are passed>
    THEN
        IF
           /*simulate disjunction*/
           Input_Error1=Input_Error2 &
           Input_Error1=TRUE
        THEN
           /*record the input status*/
           Input_ErrorN:=Input_ErrorN <- TRUE
        ELSE
           Input_ErrorN:=Input_ErrorN <- FALSE
        END ||
        /*remove the detected input from further
          observation*/
        InputN:=tail(InputN) ||
        FMS_State:=det
    WHEN
        <input sequence InputN is empty>
    THEN
        /*apply the redundancy test*/
        IF
           <the number of TRUE values in Input_ErrorN
            greater then the number of FALSE values>
        THEN
            Input_Error:=TRUE
        ELSE
            Input_Error:=FALSE
        END ||
        FMS_State=act
    END;

    <system action upon detection>

END
```

Figure 6. Excerpt from refining the error detection in FMS

5 Conclusion

In this paper we proposed a formal pattern for specifying and refining fault tolerant control systems susceptible to transient faults. We demonstrated how to ensure that safety requirement – confinement of erroneous inputs – is preserved in the entire development process. We focused on the design of subsystem of the control system – the failure management system, which enables error detection, confinement and recovery. Our approach has currently focused on considering multiple analogue sensors. We derived a general specification of the corresponding error detection mechanism which defines the appropriate tests run on the obtained inputs. We verified our pattern on a case study.

Laibinis and Troubitsyna [5] proposed a formal approach to model-driven development of fault tolerant control systems in B. However, they did not consider transient faults. Since we consider this type of faults our approach can be seen as an extension of the pattern they proposed.

More work on specifying FMS has been done by Johnson et. al [2]. However, they focused on reusability and portability of FMS modelled using UML in combination with formal methods. The error detection mechanism proposed here is based on the application of specific tests combined with the counting mechanism. Hence we focused on specifying the essence of mechanism for tolerating transient faults.

We verified our approach with the automatic tool support – Atelier B. Around 95% of all proof obligations have been proved automatically by the tool. The rest has been proved using the interactive prover. We believe that the availability of the tool supporting formal specification and verification can facilitate acceptance of our approach in industry.

In this paper we addressed a specific subset of transient faults. As a future work we are planning to enlarge this subset and derive generic patterns for specification and development of control systems tolerating them. Moreover, it would be interesting to investigate the possibility of automatic instantiation of specific requirements from which the general pattern is obtained.

Acknowledgments

This work is supported by EU funded research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

6 References

1. CClearSy, Aix-en-Provence, France. *Atelier B - User Manual*, Version 3.6, 2003.
2. I. Johnson, C. Snook, A. Edmunds and M. Butler. "Rigorous development of reusable, domain-specific components, for complex applications", In *Proceedings of 3rd International Workshop on Critical Systems Development with UML*, pages pp. 115-129, Lisbon, 2004.
3. J.-R. Abrial. *The B Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.
4. J. R. Abrial. Event Driven Sequential Program Construction, 2001. http://www.atelierb.societe.com/ressources/articles/seq.pdf
5. L. Laibinis and E. Troubitsyna. "Refinement of fault tolerant control systems in B", In *Computer Safety, Reliability, and Security - Proceedings of SAFECOMP 2004* Lecture Notes in Computer Science, Num: 3219, Page(s): 254-268, Springer-Verlag, Sep, 2004.
6. *MATISSE Handbook for Correct Systems Construction*. EU-project MATISSE: Methodologie and Technologies for Industrial Strength Systems Engineering, IST-199-11345, 2003. http://www.esil.univ-mrs.fr/~spc/matisse/Handbook
7. RODIN - Rigorous Open Development Environment for Complex Systems, Project Number: IST 2004-511599, http://rodin-b-sharp.sourceforge.net
8. R. J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.
9. Storey N. *Safety-critical computer systems.* Addison-Wesley, 1996.