Dubravka Ilić | Elena Troubitsyna

Formal Development of Software for Tolerating Transient Faults

TURKU CENTRE *for* COMPUTER SCIENCE

TUCS Technical Report
No 694, Jun 2005

Formal Development of Software for Tolerating Transient Faults

Dubravka Ilić

Elena Troubitsyna

Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14A, FIN - 20520 Turku, Finland

TUCS Technical Report
No 694, Jun 2005

Abstract

Transient faults constitute a wide-spread class of faults that appear for some time during system operation and might disappear and reappear later. They are very common in control systems. However, by appearing even for a short time, they might result in dangerous system error. Hence designing mechanisms for tolerating transient faults is an acute issue especially in the development of safety-critical control systems. In this paper we propose a formal approach to specifying software-based mechanisms for tolerating transient faults in the B Method. We focus on deriving a general specification and development pattern which can be applied in the development of various control systems. We illustrate an application of the proposed patterns by an example from avionics software product line.

Keywords: fault tolerance, transient faults, B Method, refinement

TUCS Laboratory
Distributed Systems Design

1. Introduction

Nowadays software-intensive control systems are in heart of many safety-critical applications. To guarantee *dependability* [4] of such systems we should ensure that software is not only fault free but also is able to cope with faults of other system components. In this paper we focus on designing controllers able to withstand transient physical faults of the system components. Transient faults are temporal defects within the system [11]. They frequently occur in hardware functioning. The design of mechanisms for tolerating temporal faults is inherently complex. On the one hand side, controlling software (further referred to as controller) should not over-react to an isolated transient fault. On the other side, it should ensure that even isolated transient faults are not propagated or if the fault persists, appropriate recovery actions are initiated.

In the design of complex control systems, fault tolerance mechanisms constitute a large part of the controller and are often perceived as separate subsystems dedicated to fault tolerance. In avionics, such subsystem is traditionally called Failure Management System (further referred to as FMS). The major role of FMS is to mask faulty readings obtained from sensors and hereby provide the controller with the correct information about system state using which it executes the required control functions. Design of the FMS is particularly difficult since often requirements changes are introduced at the late 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 and developing FMS. The proposed pattern can be used in the product line development.

Obviously correctness of FMS is essential for ensuring dependability of the overall system. Formal methods are traditionally used for reasoning about software correctness. In this paper we demonstrate how to develop the FMS by stepwise refinement in the B Method [5, 12]. The B Method is a formal framework for the development of dependable systems correct by construction. AtelierB [1] – the tool, supporting the method, provides a high degree of automation of the verification process, which facilitates a better acceptance of the method in the industrial practice. The approach proposed in this paper is validated by a case study from avionics industry conducted within EU project RODIN [9].

The paper is structured as follows: in Section 2 we describe FMS by presenting its structure, behaviour and the mechanism for error detection as well as giving the graphical representation of the FMS pattern. In the Section 3 we demonstrate the process of development of FMS from an abstract specification of FMS in our formal modelling framework – the B Method – till the refined specification close to implementation. Section 4 presents an example of the FMS pattern instantiation. In Section 5 we conclude our work.

2. Failure Management System

2.1 Structure and behaviour

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

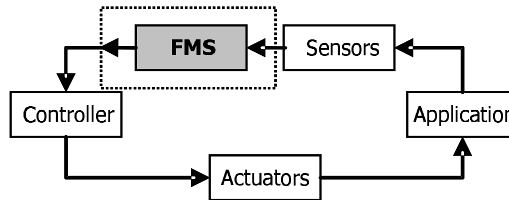


Figure 1. Place of the FMS in an embedded control system

As an input FMS takes sensor readings. The outputs from the FMS are fed into the controller. The role of FMS is to detect erroneous inputs and prevent their propagation into controller. Hence the main purpose of FMS is to supply the controller of the system with fault free inputs from the system environment. We assume that initially the system operates in the Normal mode, i.e., is error free. The operating cycle starts with obtaining sensor readings as the inputs of FMS. FMS applies certain detection mechanism on inputs. As a result of this detection inputs are categorized as: fault free or faulty. Depending on this result, FMS takes certain remedial action. The remedial actions can be classified as: healthy, temporary or confirmation actions. They determine the behaviour of the FMS. This classification is adopted according to [9].

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

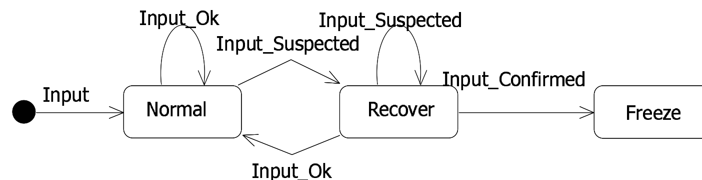


Figure 2. Specification of the FMS behaviour

Healthy action. When the FMS operates in Normal mode and detects that incoming input is fault free, the input is forwarded unchanged to the controller and FMS continues the operating cycle by accepting another input from the environment.

Temporary action. However, when the FMS operates in Normal mode and detects the first faulty input, it marks the status of that input as suspected and changes the operating mode from Normal to Recover (Figure 2). In the Recover mode FMS starts to count the number of faulty inputs. In this mode the input can get recovered during a certain number of operating cycles. Since the aim of the FMS is to give fault free output even when the input is faulty, the output of FMS while operating in Recover mode is the last

good value of the input obtained before entering the mode Recover. Once a temporary action is triggered, it will keep the system in the mode Recover as long as the status of the input coming from the environment is suspected. The counting mechanism determines whether the input gets recovered. If this is the case, the system changes its mode from Recover to Normal and healthy action is triggered again.

Confirmation. When the system operates in the mode Recover, the counting mechanism triggers confirmation action if the input fails to recover. Input is then confirmed failed and the system changes the operating mode to Freeze.

Next we describe error detection mechanism in details.

2.2 Error detection

The detection mechanism is the most important part of the FMS. Its role is to determine whether the input is faulty or fault free. We propose the architecture of the detection mechanism as shown in Figure 3. It is built out of series of tests Test*i*
- where $i, M, N \in \mathrm{N}$ and $i=1, 2, ..., M+n$ and $N=M+n$.

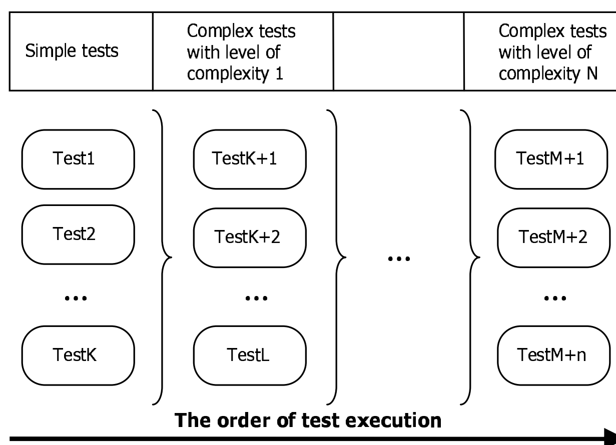


Figure 3. Detection mechanism architecture

Detection in FMS starts with applying the tests in the order defined by its architecture. For each input we define tests required to detect whether that particular input is faulty. We differentiate between different kinds of tests. The basic category is a *simple test*. An input signal may pass through several simple tests. They can be applied in any order. When triggered, simple test runs solely based on the input reading from the sensor. After the test is executed, it should be marked as passed for the current input and some other test may be triggered.

The second test category is *complex test with level of complexity 1*. The execution of this kind of test depends on the execution of several simple tests. Input may go through several complex tests of this level. Tests are executed in random order. However, in order to perform complex tests over some input, the system should first execute all required simple tests for that particular complex test as given in Figure 3.

In general, there might be N test categories, where the last test category is the *complex test with level of complexity N*. The execution of this kind of test depends not

only on the previous execution of simple tests, but also on the execution of the *complex tests with level of complexity N-1*. If input requires several tests of this kind they are executed in random order, but all the tests required for their execution should be passed in previous detection steps.

This means that detection itself operates in stages, first executing all simple tests associated with the certain input. Afterwards, all complex tests associated with the current input are executed in ascending complexity levels as shown in Figure 3.

The described detection template holds for different kinds of inputs. Sensors in the system are measuring the values of certain physical processes and they can provide continuous value readings to the FMS or can be used to display some binary conditions. Hence, inputs to the FMS can be described as analogue or boolean. For both, detection template holds in general, but different tests are applied on each of them.

2.3 FMS pattern

Described FMS actions (Figure 2) and detection template (Figure 3) constitute the generic FMS structure and behaviour pattern shown on Figure 4. The figure shows the flow of the detection decisions and effect of remedial actions after inputs are received from the system environment. Notice that three main FMS states (Normal, Recover and Freeze) now include details about input status. The additional component – counting mechanism – enables tolerance of the transient faults. It takes the system into the Normal mode after the input has recovered, leaves it in the Recover mode if the input is still suspected or freezes the system if the input failed to recover.

The given pattern can be applied in the controlling software product line [3] for creating a collection of similar control systems with the mechanism for fault tolerance of transient faults.

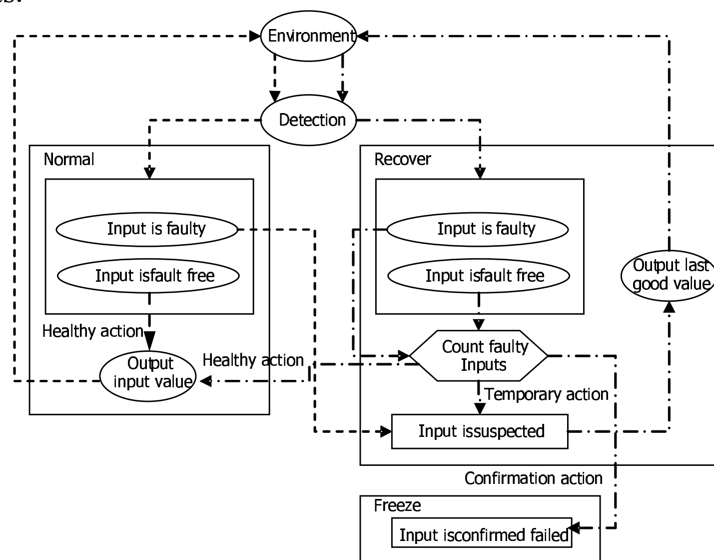


Figure 4. FMS pattern

Before presenting the formal pattern for handling fault tolerance in FMS, according to the Figure 4, we give the short introduction to the B Method.

3. Formal system development

3.1 Formal modelling in the B Method

In this paper we have chosen the B Method [5, 12] 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 [8]. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [1], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. The high degree of automation in verifying correctness improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

In B a specification is represented by a module or a set of modules, called Abstract Machines. The common pseudo-programming notation, called Abstract Machine Notation (AMN), is used to construct and formally verify them. An abstract machine encapsulates a state and operations of the specification and has the following general form:

```
MACHINE          name
SETS             Set
VARIABLES        v
INITIALISATION   Init
INVARIANT        I
OPERATIONS       Op
```

Each machine is uniquely identified by its name. The state variables of the machine are declared in the VARIABLES clause and initialized in the INITIALISATION clause. The variables in B are strongly typed by constraining predicates of INVARIANT clause. The constraining predicates are conjoint by conjunction (denoted as &). All types in B are represented by non-empty sets and hence set membership (denoted as :) expresses typing constraint for a variable, e.g., x:TYPE. Local types can be introduced by enumerating the elements of the type, e.g., TYPE = {element1, element2,...} in the SETS clause. The operations of the machine are defined in OPERATIONS clause. The operations are atomic meaning that, once an operation is chosen, its execution will run until completion without interference.

In this paper we adopt event-based approach to system modelling [6]. 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.

We use the following B statements to describe the computation in operations:

Statement	Informal meaning
X := e	Assignment
X, y := e1, e2	Multiple assignment
IF P THEN S1 ELSE S2 END	If P is true then execute S1, otherwise S2
S1 ; S2	Sequential composition
S1 || S2	Parallel execution of S1 and S2
X :: T	Nondeterministic assignment – assigns variable x arbitrary value from given set T

B also provides structuring mechanisms for modularization which allows us to express machines as compositions of other machines. For instance, we use SEES clause. When in the specification of machine M1 we have a clause M1 SEES M2, where M2 is another machine, then the sets, constants and the state of M2 are available to M1 for use in its own initializations and within preconditions and bodies of operations. This allows us to define sets and constants separately from the main machine.

The development methodology adopted by B is based on stepwise refinement [10]. Refinement can be seen as a process of gradual incorporation of implementation details into the system 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. In the former case, the tool generates the required proof obligations and discards them without user's help. In the later case, the user proves certain proof obligations in the interactive mode.

Next we demonstrate how to formally specify failure management system described in the previous section.

3.2 FMS specification pattern

Control systems are usually cyclic, i.e., their behaviour is essentially an interleaving between the environment stimuli and controller reaction on these stimuli. The controller 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. 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 still produce the fault free output or to stop the system without producing any output at all.

The abstract specification pattern given in Figure 6 is obtained following the informal FMS description represented graphically in Figure 4. The FMS machine reads sets and constants defined in the machine Global and hence this machine is mentioned in the SEES clause of the FMS machine. The variables declared in the FMS machine define its state. 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 faulty 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 producing fault free output. Then the operating cycle starts again. Hence, the behaviour of the FMS can be described as in Figure 5.

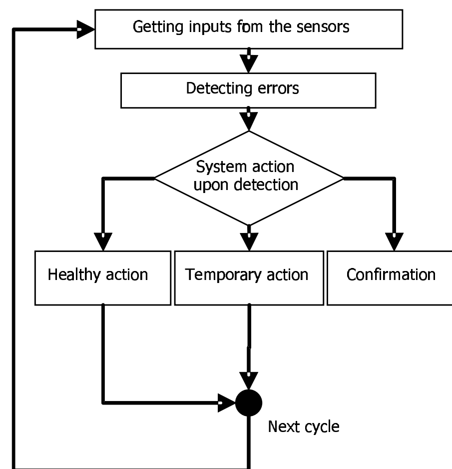


Figure 5. Behaviour of the FMS

Since the controller relies only on the input from the FMS, we should guarantee that it obtains the fault free output from the FMS. 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)).

In our abstract specification the input values produced by the environment are modelled nondeterministically. After getting the inputs, FMS detects whether they are faulty. 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 fault free. If the input is faulty, FMS initiates

temporary action, i.e., error recovery. 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 while the system is in the mode Recover. The first counter cc is used for accumulating inputs in error. While the system is in the mode Recover, each time when the obtained input is found faulty, 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 fault free, the cc is decremented by the given value yy.

```
MACHINE
    FMS
SEES
    Global
VARIABLES
    Input, Input_Error, Input_Status, FMS_State, Flag, cc, num
INVARIANT
    Input : T_INPUT &          /*actual input to the
                                 FMS*/
    Input_Error : BOOL &       /*variable modelling
                                 input state*/
    Input_Status : I_STATUS & /*variable modelling
                                 input status
                                 during recovering*/
    FMS_State : STATES &    /*variable modelling
                              system state*/
    Flag:FLAGS &            /*variable modelling
                              system substates*/
    cc : NAT &              /*cc and num are
                              counters*/
    num : NAT & <safety invariant>
INITIALISATION
    Input::T_INPUT || Input_Error:=FALSE ||
    Input_Status:=ok ||
    Flag:=Normal || FMS_State:=env || cc:=0 || num:=0
OPERATIONS
Environment=
SELECT FMS_State=env
THEN
     Input::T_INPUT || FMS_State:=det
END;

Detection=
SELECT FMS_State=det
THEN
     Input_Error :: BOOL || FMS_State:=act
END;

Action=
SELECT
     FMS_State=act & Input_Error=FALSE &
     Flag=Normal
THEN
     Input_Status:=ok || FMS_State:=out
WHEN
     FMS_State=act & Input_Error=TRUE & Flag=Normal
THEN
     Flag:=Recover || cc:=cc+xx || num:=num+1 ||
     FMS_State:=rcv
WHEN
      FMS_State=act & Input_Error=FALSE &
      Flag=Recover
THEN
     cc:=cc-yy || num:=num+1 || FMS_State:=rcv
WHEN
     FMS_State=act & Input_Error=TRUE &
     Flag=Recover
THEN
     cc:=cc+xx || num:=num+1 || FMS_State:=rcv
END;

Recovering=
SELECT FMS_State=rcv & Flag=Recover &
       (num>=Limit or cc>=zz)
THEN
     Input_Status:=confirmed || FMS_State:=stop
WHEN
     FMS_State=rcv & Flag=Recover &
     num<Limit & cc=0
THEN
     Input_Status:=ok || FMS_State:=out
WHEN
     FMS_State=rcv & Flag=Recover &
     num<Limit & cc/=0 & cc<zz
THEN
     Input_Status:=suspected || FMS_State:=out
END;

Return=
SELECT FMS_State=out & Flag=Normal &
       Input_Status=ok
THEN
     <Execute healthy action> || FMS_State:=env
WHEN
     FMS_State=out & Flag=Recover &
     Input_Status=suspected
THEN
     < Execute temporary action> || FMS_State:=env
WHEN
     FMS_State=out & Flag=Recover & Input_Status=ok
THEN
     Flag:=Normal || < Execute healthy action> ||
     num:=0 ||
     FMS_State:=env
END;

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

Figure 6. Excerpt from the abstract FMS specification pattern

Each operating cycle sets 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 system operation and freezing the system. Since each faulty input increments the value of cc

and each fault free input decrements it, eventually the counter cc is set to zero. This is possible if the FMS starts receiving fault free inputs. If cc reaches zero the input is considered recovered and the system returns to the Normal mode 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.

Although our initial specification of FMS shown in Figure 6 is very abstract it anyway completely describes the intended behaviour of the FMS. However it leaves the mechanism of detecting errors in input underspecified. Next, we demonstrate how to obtain the detailed specification of error detection in the refinement process.

3.3 FMS refinement pattern – refining error detection in FMS

We aim at considering multiple sensors detecting the signal from the environment. Our first refinement step is data refinement which replaces the variable Input modelling one input reading with the variable InputN which models the sequence of input readings from N sensors as shown on Figure 9.

After refining the data structure as described, we continue the development by refining the nondeterministic assignment of values to the variable Input_Error in the *Detection* operation of the abstract machine. In the abstract machine we have specified that the input can be either faulty or fault free. As a result of data refinement, we introduce the variable Input_ErrorN which models the detection result for each input reading from N sensors, instead of previously defined variable Input_Error modelling only the detection result of one sensor input.

Detection results in Input_ErrorN are determined by running the detection tests. The expression Run: seq(T_INPUTS)<->TESTS in the machine Global defines which tests are required for detecting particular erroneous inputs. Since we observe homogeneous multiple sensors measuring the same physical process from the environment, for each of N sensor readings the same series of tests has to be applied.

Hence, the tests are defined for InputN sequence in general as shown in Figure 7a) and not for particular sensor readings. However, the pattern for heterogeneous multiple sensors can easily be adapted from the one presented here. This can be achieved by defining tests for each one of the N sensor readings separately, as shown in Figure 7b).

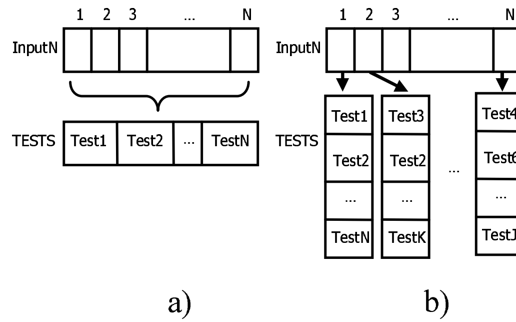


Figure 7. Defining tests for homogeneous and heterogeneous multiple sensors

The application of test follows the rules given in Section 2.2 and in the order defined by Figure 3. For each input reading in the sequence InputN, we apply test as follows from the definition of Run given in the machine Global.

The *Detection* operation starts by running first all simple tests and recording their results in the variable Result. The predicate Test*i* : ran({InputN}<|Cond), which is a conjunct of the condition of the *Detection* operation, restricts the test execution only on the test defined for the particular sensor reading.

The purpose of the simple test is to detect anomalies in the input reading solely based on the signal coming from the environment. However, sometimes it is useful to combine results of several simple tests to search for anomalies in the input reading. To achieve this we use complex test. When all simple tests are executed over one input reading, one of the complex tests is triggered. Regardless on how many simple or complex tests are executed, one of them has to set the final status value for the particular input reading as shown on Figure 8.

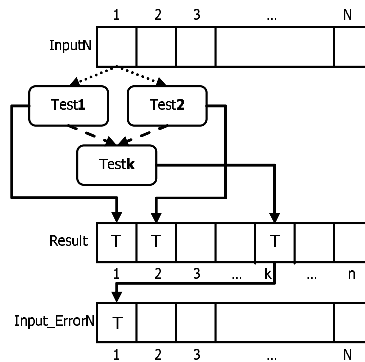


Figure 8. Process of setting the status values for N sensor inputs

When all the input readings from N sensors are detected and classified as fault free or failed, the single output should be given to the system controller. Hence, we apply some complex test to determine if the incoming input to the FMS is faulty or not. The decision is made based on the value of the variable Input_ErrorN.

After the status of the input is detected, FMS makes a decision how to proceed with handling it, i.e., which action to apply as specified in the abstract machine. The refinement relation for this step is as follows:

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

The above relation expresses the fact that if the state of the abstract variable Input_Error is TRUE (the input is faulty), then in the refining variable Input_ErrorN the number of fault free inputs is smaller then the number of faulty inputs.

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

```
REFINEMENT
    FMSR1
REFINES
    FMS
SEES
    Global
VARIABLES
    InputN, Input_Error,
    Result,
    Input_ErrorN,
    FMS_State,
    cc, num,
    Passed
    ...
INVARIANT
    InputN : seq(T_INPUT) &     /*N sensor input
                                  reading*/
    Input_Error : BOOL &
    Result : TESTS → BOOL &     /*test results*/
    Input_ErrorN : seq(BOOL) & /*faulty status for
                                  N sensor inputs*/
    FMS_State : STATES &
    cc : NAT & num : NAT &
    Passed : TESTS → BOOL &  /*variables for modelling
                                test application*/
    <safety and gluing invariants>
INITIALISATION
    InputN := [] || Input_Error := FALSE ||
    Result := TESTS*{TRUE} ||
    Input_ErrorN := [] ||
    FMS_State := env ||
    cc := 0 || num:=0 ||
    Passed := TESTS*{FALSE}
OPERATIONS

    <Obtaining the input from the environment>

Detection=
SELECT
    FMS_State=det & InputN/=[] &
    Test1:ran({InputN}<|Cond) & Passed(Test1)=FALSE
THEN
    Result(Test1)::BOOL ||
    Passed(Test1):=TRUE ||
    FMS_State:=det
WHEN
    FMS_State=det & InputN/=[] &
    Test2:ran({InputN}<|Cond) & Passed(Test2)=FALSE
THEN
    Result(Test2) :: BOOL ||
    Passed(Test2):=TRUE ||
    FMS_State:=det
WHEN
    ...
THEN
    ...
WHEN
    FMS_State=det & InputN/=[] &
    TestK:ran({InputN}<|Cond) & Passed(TestK)=FALSE
THEN
    Result(TestK) :: BOOL ||
    Passed(TestK):=TRUE ||
    FMS_State:=det
WHEN
    ...
THEN
    ...
WHEN
    FMS_State=det & InputN/=[] &
    TestN:ran({InputN}<|Cond) & TestN:dom(ComlexTest) &
    Passed(ComplexTest(TestN)(1))=TRUE &
    Passed(ComplexTest(TestN)(2))=TRUE &
    ...
    Passed(ComplexTest(TestN)(size(ComplexTest(TestN
    )))=TRUE
THEN
     <Execute the complex test – TestN to determine the
       values of N sensor readings in Input_ErrorN> ||
     Passed:=TESTS*{FALSE} ||
     FMS_State:=det
WHEN
     FMS_State=det & InputN = []
THEN
     <Vote the input status based on the values in
       Input_ErrorN and calculate the output to
       controller> ||
     FMS_State:=act
END;
     <System action upon detection>
END
```

Figure 9. Excerpt from FMS refinement pattern

In Figure 9 we give the excerpt from the FMS refinement pattern with introduced error detection mechanism. Observe that the SELECT statement in the *Detection* operation of the FMSR1 machine allows us to model random test execution when more than one branch of the SELECT statement is enabled.

Next we will validate our approach on a case study.

4. Case study – an example

The proposed FMS refinement pattern gives the template for the instantiation of a domain-specific reliable FMS. We validate our approach by an example from the avionics industry – mechanism for tolerating transient faults of multiple temperature sensors. These analogue sensors measure the temperature of the aircraft engine and their values range between -200 and 2000°F. There are five temperature sensors obtaining the input for the Engine FMS. Hence we set the value of the constant SeqSize in the machine Global (given in the Appendix) to 5 for this particular example.

```
REFINEMENT
    Instance
REFINES
    FMS

OPERATIONS

<obtaining the input from the environment>

Detection=
SELECT
      FMS_State=det & InputN /= [] &
      Test1:ran({InputN}<|Cond) &
      Passed(Test1)=FALSE
THEN IF
            (first(InputN)>=Low_Bound-Delta) &
            (first(InputN)<=Upp_Bound+Delta)
      THEN
            Result(Test1):=FALSE
      ELSE
            Result(Test1):=TRUE
      END ||
      Passed(Test1):=TRUE ||
      FMS_State:=det
WHEN
      FMS_State=det & InputN /= [] &
      Test2:ran({InputN}<|Cond) &
      Passed(Test2)=FALSE
THEN IF
            first(Previous)>=first(InputN)
      THEN
            IF
               (first(Previous)-first(InputN))
                >Allowed_difference
            THEN
               Result(Test2):=TRUE
            ELSE
               Result(Test2):=FALSE
            END
      ELSIF
            first(Previous)<first(InputN)
      THEN IF
               (first(InputN)-first(Previous))
               >Allowed_difference
            THEN
               Result(Test2):=TRUE
            ELSE
               Result(Test2):=FALSE
            END
      END ||
      Passed(Test2):=TRUE ||
      FMS_State:=det
WHEN
      FMS_State=det & & InputN /= [] &
      Test3:ran({InputN}<|Cond) & Test3:dom(ComplexTest) &
      Passed(ComplexTest(Test3)(1))=TRUE &
      Passed(ComplexTest(Test3)(2))=TRUE
THEN
      IF
            Result(Test1)= Result(Test2) &
            Result(Test1)=TRUE
      THEN
            Input_ErrorN:=Input_ErrorN <- TRUE ||
      ELSE
            Input_ErrorN:=Input_ErrorN <- FALSE
      END ||
      InputN:=tail(InputN) ||
      Previous:=tail(Previous) ||
      Passed:=TESTS*{FALSE} ||
      FMS_State:=det
WHEN
      FMS_State=det & InputN = []
THEN
      IF
            card(Input_ErrorN|>{FALSE})>
            card(Input_ErrorN|>{TRUE})
      THEN
            Input_Error:=FALSE ||
            <Calculate median of fault free inputs
             as the output value>
      ELSE
            Input_Error:=TRUE
      END ||
      FMS_State:=act
END;
      <system action upon detection>
END
```

Figure 10. Excerpt from an instance of FMS pattern

The detection of this kind of input implies the application of two simple tests: the magnitude test and the rate test. In the magnitude test the input is compared against some predefined limit (bound). If the limit is exceeded then it is considered to be faulty. The rate test is detecting incorrect 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 faulty. The excerpt from the resulting machine implementing described tests is shown in Figure 10. The full specification is given in the Appendix.

It is obvious that both tests have a certain preconfigurations expressed through the predefined limits which allow dynamic test changes as appropriate. These limits are defined as constants in the Global machine.

If the input passes the magnitude test, the value of the variable Result(Test1) is set to FALSE, otherwise to TRUE (i.e., the test on this input failed). Similarly, if the input passes the rate test, the value of the temporary variable Result(Test2) is set to FALSE, otherwise to TRUE.

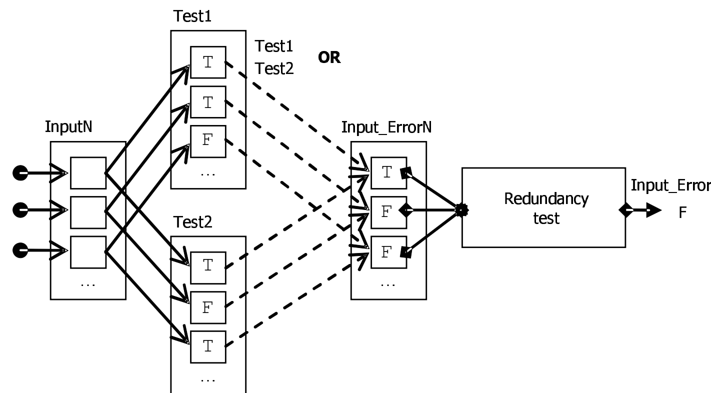


Figure 11. Introducing error detection

The input is fault free if none of these tests fail. Hence we define the status of the input as the disjunction of Result(Test1) and Result(Test2) and set the variable Input_ErrorN accordingly. The error detection for multiple sensors (InputN) implies first the application of magnitude and rate tests. When they are passed, the complex test is applied. Usually complex test is a redundancy test. We consider N sensor readings, where N is an odd number (in our example 5). The status of each one of the N sensor inputs is recorded in the variable Input_ErrorN. The redundancy test performs majority voting. If the majority of values of Input_ErrorN sequence is TRUE, then the whole input is considered failed, otherwise it is fault free. As presented on the Figure 11, redundancy test takes the detected multiple inputs (Input_ErrorN) and based on their values (TRUE or FALSE) votes for the input status (Input_Error). After the input status is finally detected, FMS proceeds with the corresponding remedial actions, as specified in the refinement pattern described previously.

5. Conclusion

This paper proposes a formal pattern for specifying and refining a part of the safety-critical control system – the Failure Management System. We demonstrated how to ensure that safety requirement – confinement of faulty inputs – stays preserved in the entire development process.

We derived a general specification of the error detection mechanism which defines the appropriate tests run on the obtained inputs. Our approach considers inputs obtained from the homogeneous multiple sensors. However we showed that it can be adapted for handling inputs from heterogeneous multiple sensors as well.

The proposed error detection mechanism is based on the execution of series of predefined tests suitable for managing both analogue and boolean inputs. The tests execution is defined by the layered detection mechanism architecture in such a way that enables tackling the input anomalies more efficiently.

Laibinis and Troubitsyna have proposed a formal approach to model-driven development of fault tolerant control systems in B [7]. However, they did not consider transient faults. Since we consider this type of faults our approach is an extension of the pattern they proposed.

Development of FMS in UML has been undertaken by Johnson et. al [2]. They focused on reusability and portability of FMS modelled using UML in combination with formal methods. The error detection mechanism we proposed is based on a specific test architecture which combined with the counting mechanism builds the core of our approach to tolerating the transient faults.

We showed how to instantiate the proposed pattern in a realistic case study. We verified the instantiation with the automatic tool support used for the pattern development – Atelier B. The tool support has significantly simplified the development process and increased our confidence in the correctness of the obtained results. 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.

As a future work it would be interesting to implement more sophisticated conditions for test execution, since now all the required tests are explicitly predefined for each input.

References

[1] ClearSy, 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. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach, Addison-Wesley, 2000

[4] J.-C. Laprie, Dependability: Basic Concepts and Terminology, Springer-Verlag, Vienna, 1991

[5] J.-R. Abrial, *The B Book: Assigning Programs to Meanings*, Cambridge University Press, 1996

[6] J. R. Abrial. Event Driven Sequential Program Construction, 2001. http://www.atelierb.societe.com/ressources/articles/seq.pdf

[7] L. Laibinis and E. Troubitsyna. “Refinement of fault tolerant control systems in B”, In *ComputerSafety, Reliability, and Security - Proceedings of SAFECOMP 2004* Lecture Notes in Computer Science, Num: 3219, Page(s): 254-268, Springer-Verlag, Sep, 2004

[8] *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

[9] RODIN - Rigorous Open Development Environment for Complex Systems, Project Number: IST 2004-511599
http://rodin-b-sharp.sourceforge.net

[10] R. J. Back and J. von Wright, *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998

[11] Storey N. Safety-critical computer systems. Addison-Wesley, 1996

[12] S. Schneider, The B Method. An introduction, Palgrave, 2001

Appendix

B development of the FMS system

```
MACHINE         Global

SETS            STATES = {env, det, act, out, rcv, stop, det_red};
                I_STATUS = {ok, suspected, confirmed};
                FLAGS = {Normal, Recover};
                TESTS={Test1, Test2, Test3}

ABSTRACT_CONSTANTS

                T_INPUT, Limit, xx, yy, zz, Low_Bound, Upp_Bound,
                Delta, Allowed_difference,  SeqSize, Cond, ComplexTest

PROPERTIES
                T_INPUT <: NAT & 0:T_INPUT &
                !nn.(nn:NAT => nn<2147483645) &
                SeqSize:NAT &
                !ee.(ee:seq(T_INPUT) => size(ee)=SeqSize) &
                Limit : NAT &
                xx = 2 &
                yy = 1 &
                zz = 8 &
                Low_Bound : NAT &
                Upp_Bound : NAT &
                Delta : NAT &
                Low_Bound < Upp_Bound &
                Allowed_difference : NAT &
                Cond : seq(T_INPUT) <-> TESTS &
                ComplexTest : TESTS --> iseq(TESTS)

END
```

```
MACHINE     FMS

SEES        Global

VARIABLES
            Input, Input_Status, Input_Error,
            LastGood,
            Output,
            FMS_State,
            cc,num,
            Flag

INVARIANT

            Input : T_INPUT & Input_Status : I_STATUS &
            Input_Error : BOOL &
            LastGood : T_INPUT &
            Output : T_INPUT &
            FMS_State : STATES &
            cc : NAT & num : NAT &
            Flag : FLAGS &

            /* Safety invariants. */
            (Input_Status=confirmed => FMS_State=stop) &
            (Input_Status=ok=>(cc=0 & num=0 & Output=Input)) &
            (Input_Status=suspected => Output/=Input)

INITIALISATION

            Input :: T_INPUT || Input_Status := ok ||
            Input_Error := FALSE ||
            LastGood := Input ||
            Output := Input  ||
            FMS_State := env ||
            cc := 0 || num:=0 ||
            Flag := Normal

OPERATIONS

        Environment=
        SELECT
            FMS_State=env
        THEN
            Input::T_INPUT || FMS_State:=det
        END;

        Detection=
        SELECT
            FMS_State=det
        THEN
            Input_Error::BOOL || FMS_State:=act
        END;
```

```
Action=
SELECT
        FMS_State=act & Input_Error=FALSE & Flag=Normal
THEN
        Input_Status:=ok ||
        FMS_State:=out
WHEN
        FMS_State=act & Input_Error=TRUE & Flag=Normal
THEN
        Flag:=Recover ||
        cc:=cc+xx ||
        num:=num+1 ||
        FMS_State:=rcv
WHEN
        FMS_State=act & Input_Error=FALSE & Flag=Recover
THEN
        cc:=cc-yy ||
        num:=num+1 ||
        FMS_State:=rcv
WHEN
        FMS_State=act & Input_Error=TRUE & Flag=Recover
THEN
        cc:=cc+xx ||
        num:=num+1 ||
        FMS_State:=rcv
END;

Recovering=
SELECT
        FMS_State=rcv & Flag=Recover &
        (num>=Limit or cc>=zz)
THEN
        Input_Status:=confirmed ||
        FMS_State:=stop
WHEN
        FMS_State=rcv & Flag=Recover & num<Limit & cc=0
THEN
        Input_Status:=ok ||
        FMS_State:=out
WHEN
        FMS_State=rcv & Flag=Recover &
        num<Limit & cc/=0 & cc<zz
THEN
        Input_Status:=suspected ||
        FMS_State:=out
END;
```

```
        Return=
        SELECT
                FMS_State=out & Flag=Normal & Input_Status=ok
        THEN
                LastGood:=Input ||
                Output:=Input ||
                FMS_State:=env
        WHEN
                FMS_State=out & Flag=Recover &
                Input_Status=suspected
        THEN
                Output:=LastGood ||
                FMS_State:=env
        WHEN
                FMS_State=out & Flag=Recover & Input_Status=ok
        THEN
                LastGood:=Input ||
                Output:=Input ||
                Flag:=Normal ||
                num:=0 ||
                FMS_State:=env
        END;

        Stopping=
        SELECT
                FMS_State=stop
        THEN
                skip
        END

END
```

```
REFINEMENT  Instance

REFINES     FMS

SEES        Global

ABSTRACT_VARIABLES

            Input, InputN,
            Input_Status, Input_Error,
            Result, Input_ErrorN,
            LastGood, Previous, Current,
            Output,
            FMS_State,
            cc, num,
            Flag,
            Passed,
            sum, counter

INVARIANT

            Input : T_INPUT & InputN : seq(T_INPUT) &
            Input_Status : I_STATUS & Input_Error : BOOL &
            Result : TESTS → BOOL & Input_ErrorN : seq(BOOL) &
            LastGood : T_INPUT & Previous : seq(T_INPUT) &
            Current : seq(T_INPUT) &
            Output : T_INPUT &
            FMS_State : STATES &
            cc : NAT & num : NAT &
            Flag : FLAGS &
            Passed : TESTS → BOOL &
            sum:NAT & counter:NAT &

            /* Safety invariants. */
            (Input_Status=confirmed => FMS_State=stop) &
            (Input_Status=ok=>(cc=0 & num=0 & Output=Input)) &
            (Input_Status=suspected => Output/=Input) &
            (Input_Error=TRUE =>(card(Input_ErrorN|>{TRUE})
            >card(Input_ErrorN|>{FALSE}))) &
            (Output=LastGood=>
            Input_Error=TRUE & Input_Status/=suspected)

INITIALISATION

            Input := 0 || InputN := [] ||
            Input_Status := ok || Input_Error := FALSE ||
            Result := TESTS*{TRUE} || Input_ErrorN := [] ||
            LastGood := Input || Previous:=[] || Current:=InputN ||
            Output := Input  ||
            FMS_State := env ||
            cc := 0 || num:=0 ||
            Flag := Normal ||
```

```
        Passed := TESTS*{FALSE} ||
        sum:=0 || counter:=0

OPERATIONS

    Environment=
    SELECT
        FMS_State=env & Flag=Normal
    THEN
        Previous:=Current;
        InputN::seq(T_INPUT);
        Current:=InputN;
        FMS_State:=det
    END;

    Detection=
    SELECT
        FMS_State=det & InputN /= [] &
        Test1:ran({InputN}<|Cond) &
        Passed(Test1)=FALSE
    THEN
      IF
          (first(InputN)>=Low_Bound-Delta) &
          (first(InputN)<=Upp_Bound+Delta)
      THEN
          Result(Test1):=FALSE
      ELSE
          Result(Test1):=TRUE
      END ||
        Passed(Test1):=TRUE ||
        FMS_State:=det
    WHEN
        FMS_State=det & InputN /= [] &
        Test2:ran({InputN}<|Cond) &
        Passed(Test2)=FALSE
    THEN
      IF
          first(Previous)>=first(InputN)
      THEN IF
              (first(Previous)-first(InputN))>Allowed_difference
          THEN
              Result(Test2):=TRUE
          ELSE
              Result(Test2):=FALSE
          END
      ELSIF
          first(Previous)<first(InputN)
      THEN IF
              (first(InputN)-first(Previous))>Allowed_difference
          THEN
              Result(Test2):=TRUE
          ELSE
```

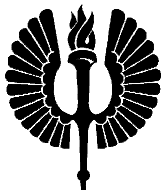
```
                Result(Test2):=FALSE
            END
        END ||
        Passed(Test2):=TRUE ||
        FMS_State:=det
WHEN
        FMS_State=det & Test3:ran({InputN}<|Cond) &
        Test3:dom(ComplexTest) &
        Passed(ComplexTest(Test3)(1))=TRUE &
        Passed(ComplexTest(Test3)(2))=TRUE
THEN
        IF
                Result(Test1)=Result(Test2) & Result(Test1)=TRUE
        THEN
                Input_ErrorN:=Input_ErrorN <- TRUE ||
                sum:=sum+first(InputN) ||
                counter:=counter+1
        ELSE
                Input_ErrorN:=Input_ErrorN <- FALSE
        END ||
        InputN:=tail(InputN) ||
        Previous:=tail(Previous) ||
        Passed:=TESTS*{FALSE} ||
        FMS_State:=det
WHEN
        FMS_State=det & InputN = []
THEN
        IF
                card(Input_ErrorN|>{FALSE})>
                card(Input_ErrorN|>{TRUE})
        THEN
                Input_Error:=FALSE ||
                Input:=sum/counter
        ELSE
                Input_Error:=TRUE
        END ||
        sum:=0 ||
        counter:=0 ||
        FMS_State:=act
END;

Action=
SELECT
        FMS_State=act & Input_Error=FALSE & Flag=Normal
THEN
        Input_Status:=ok ||
        FMS_State:=out
WHEN
        FMS_State=act & Input_Error=TRUE & Flag=Normal
THEN
        Flag:=Recover ||
        cc:=cc+xx ||
```

```
        num:=num+1 ||
        FMS_State:=rcv
WHEN
        FMS_State=act & Input_Error=FALSE & Flag=Recover
THEN
        cc:=cc-yy ||
        num:=num+1 ||
        FMS_State:=rcv
WHEN
        FMS_State=act & Input_Error=TRUE & Flag=Recover
THEN
        cc:=cc+xx ||
        num:=num+1 ||
        FMS_State:=rcv
END;

Recovering=
SELECT
        FMS_State=rcv & Flag=Recover &
        (num>=Limit or cc>=zz)
THEN
        Input_Status:=confirmed ||
        FMS_State:=stop
WHEN
        FMS_State=rcv & Flag=Recover & num<Limit & cc=0
THEN
        Input_Status:=ok ||
        FMS_State:=out
WHEN
        FMS_State=rcv & Flag=Recover &
        num<Limit & cc/=0 & cc<zz
THEN
        Input_Status:=suspected ||
        FMS_State:=out
END;

Return=
SELECT
        FMS_State=out & Flag=Normal & Input_Status=ok
THEN
        LastGood:=Input ||
        Output:=Input ||
        FMS_State:=env
WHEN
        FMS_State=out & Flag=Recover & Input_Status=suspected
THEN
        Output:=LastGood ||
        FMS_State:=env
WHEN
        FMS_State=out & Flag=Recover & Input_Status=ok
THEN
        LastGood:=Input ||
```

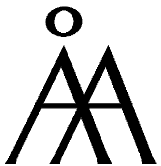
```
        Output:=Input ||
        Flag:=Normal ||
        num:=0 ||
        FMS_State:=env
    END;

    Stopping=
    SELECT
        FMS_State=stop
    THEN
        skip
    END

END
```

TURKU CENTRE *for* COMPUTER SCIENCE

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi



University of Turku
- Department of Information Technology
- Department of Mathematics



Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 952-12-1573-9
ISSN 1239-1891