Dubravka Ilić | Elena Troubitsyna |
Linas Laibinis | Colin Snook

Formal Model-Driven Development of Fault Tolerant Control Systems

Turku Centre *for* Computer Science

TUCS Technical Report
No 828, August 2007

Formal Model-Driven Development of Fault Tolerant Control Systems

Dubravka Ilić

TUCS, Åbo Akademi University, Department of Information Technologies, Joukahaisenkatu 3-5 A, 5th floor
20520 Turku, FINLAND

Elena Troubitsyna

Åbo Akademi University, Department of Information Technologies
Technologies, Joukahaisenkatu 3-5 A, 5th floor
20520 Turku, FINLAND

Linas Laibinis

Åbo Akademi University, Department of Information Technologies
Technologies, Joukahaisenkatu 3-5 A, 5th floor
20520 Turku, FINLAND

Colin Snook

School of Electronics and Computer Science,
University of Southampton, SO17 1BJ, UK

TUCS Technical Report
No 828, August 2007

Abstract

Fault tolerance techniques aim at ensuring that a system continues to operate properly even in the presence of faults. Fault tolerance is especially important in safety-critical systems, where system failures might have catastrophic consequences. Transient faults – temporal defects within the system – are typical for control systems. However, they require complex mechanisms to tolerate them. In this paper, we propose an approach to formal model-driven development of Failure Management System (FMS) – a software component implementing a mechanism for tolerating transient faults in control systems in avionics. In our development we integrate formal modelling and verification in the B Method with graphical modelling in UML-B. UML-B is a specialized subset of UML which supports automatic translation of the subset of UML diagrams into the B specifications. By integrating formal and graphical modelling we combine benefits of visual modelling with rigorous verification. We develop generic patterns for modelling the FMS at different development stages starting at high level of abstraction till detailed pre-implementation specification. Correctness of model transformations is ensured by refinement relation. The developed generic patterns can be easily reused in the product line development in the avionics domain. Hence, the proposed approach facilitates reuse and leverages efficiency while ensuring dependability of developed systems.

Keywords: B Method, control systems, fault tolerance, refinement, transient faults, UML-B

TUCS Laboratory
Distributed Systems Design

1. Introduction

Nowadays complexity of software increases constantly and rapidly. Moreover, competitive software markets impose additional demands on software development in terms of its time efficiency and cost effectiveness. However, speeding up software development to meet this requirement involves significant risks, especially when developing *dependable* systems [1].

To guarantee dependability, we should ensure that software is not only fault-free but also is able to cope with the faults of the other system components. In this paper we propose an approach to developing a software component called *Failure Management System* (FMS), which implements a mechanism for tolerating transient faults in control systems. FMS is a typical subcomponent of an embedded control system in the avionic domain, dedicated to fault tolerance. The main purpose of the FMS is to protect the controlling software – controller – from failures caused by transient sensor faults.

Transient faults [2], are common type of faults in control systems. Transient faults are temporal defects within the system. They may appear for a short time while the system is operating, then disappear, but possibly reappear later. However, even their short presence may lead to a system failure. Obviously, the lack of tolerance to these faults in the system could have severe consequences. Hence, ensuring correct functioning of the mechanism for tolerating transient faults is essential for ensuring dependability of the overall control system.

To provide cost-effective and time-efficient development of such system, we consider it in the context of software product lines [3]. Recently, software product lines development paradigm has emerged as a technique leveraging cost-effectiveness and time-efficiency of development process. It is based on well-structured reuse of already developed software assets. If correctness of reusable software assets is guaranteed and reuse is done in a rigorous manner then such a development technique potentially increases dependability of developed systems. To achieve this we present an approach to development of generic reusable patterns modelling FMS at different abstraction levels. Our approach is an example of formalized model-driven development [4]. It integrates formal framework of the B Method [5] with visual modelling in UML-B [6]. The B Method is a formal framework for the development of dependable systems correct by construction. It adopts refinement approach – a paradigm for top-down development of systems correct by construction. To ensure a wide acceptance of our approach we combine formal modelling and verification with graphical modelling in UML-B. UML-B is a specialized subset of UML [7], which enables automatic translation of UML diagrams into the B specifications. In other words, it allows us to hide the formal B syntax behind the UML notation while still rely on the B semantics.

We show how to develop the FMS generic models in UML-B, through a number of development phases supported by refinement-based model transformations. Development starts from an abstract FMS model expressed in UML-B. In general, we model fault tolerance as an intrinsic part of the system by specifying its main steps: error detection and error recovery. The system structure and behaviour are specified using different types of UML-B diagrams. This results in a well-structured system

specification. Each new development phase incorporates more details of the fault tolerance mechanisms into previous development phases, in a structured manner, while preserving already specified system properties and behaviour. The development completes with a fully described model of the FMS structure and behaviour, represented by a set of development templates. They can be instantiated by concrete data to obtain specifications of similar systems in the application domain.

To automate the process of obtaining formal B specifications from UML-B models, we use the translator tool U2B [8]. Correctness of the development is verified using AtelierB [9] – an automated tool support for the B Method. Therefore, the proposed approach has a high degree of automation.

The paper is organized as follows. In Section 2 we introduce the FSM by describing its structure and typical patterns of its behaviour. Then, Section 3 briefly outlines our formal modelling frameworks – the B Method and UML-B. We describe the main modelling concepts of both and their development paradigms. The main contribution of the paper is presented in Section 4, where we describe in detail the development of the FMS in UML-B. We start from an abstract model of the FMS and obtain more detailed FMS models through a number of development phases. Each phase first shortly describes what is modelled, and then proceeds with presenting the models in detail. Each phase concludes with the corresponding B model, obtained automatically from the UML-B models using the tool support, U2B. Finally, in Section 5, we discuss some related work and conclude the presented approach by outlining our future work.

2. Failure Management System

Embedded control systems are typical examples of reactive systems. The controlling software of such systems, called *controller*, is designed to react to stimuli of the application by setting actuators to certain values. The behaviour of the application is monitored by sensors – the devices that transform physical parameters of the application into signals, which are then used as inputs by the controller. Typical structure of a control system is shown in Fig. 1.

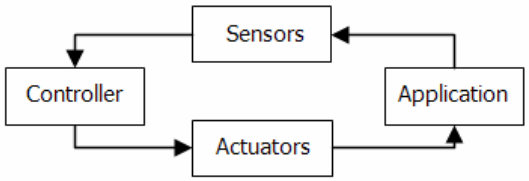


Fig. 1. Structure of an embedded control system

In general, sensors can be classified into analogue or switch-type sensors, and hence, the corresponding inputs can be represented as numerical or Boolean values. The behaviour of a control system is cyclic. The controller processes inputs obtained from the sensors, calculates new values to which the actuators will be set, and starts a new cycle.

In this paper we adopt the systems approach, where both the controlled application and the controller are modelled together.

The Failure Management System (FMS) [10, 11] is a part of the controller (shown in Fig. 2). It is designed to protect the system from erroneous sensor inputs and prevent their further propagation. Hence, it can be perceived as its protective "wrapper".

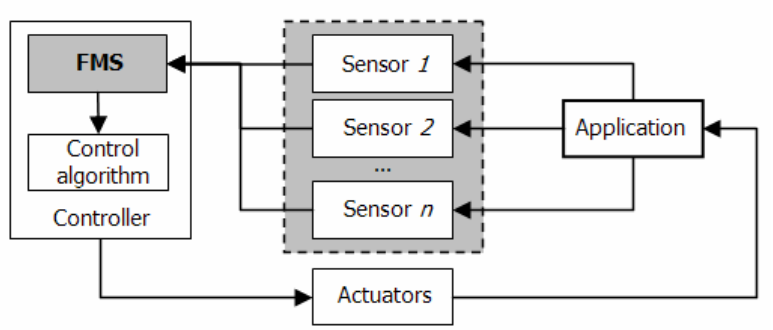


Fig. 2. Position of the FMS within an embedded control system

In this paper we consider inputs to be obtained from multiple homogenous analogue sensors, i.e., a set of redundant sensors of the same type, monitoring the same physical process. The FMS obtains sensor readings as its input, analyses them and produces the result of analysis as the output to the control algorithm. In its turn, the control algorithm performs the system controlling actions. While calculating the output, the FMS has to ensure that only fault-free inputs received from the system environment are passed to the controller. To achieve this, we assume that initially the system is error-free. The FMS operating cycle starts by obtaining the readings from the monitored sensors as the inputs to the FMS. Then, the FMS performs error detection on these inputs by applying the predefined *error detection mechanism*. As a result, the inputs are categorized as fault-free or faulty. Then, the FMS performs the *input analysis* and, depending on which action is chosen, it either simply outputs the sensor reading, calculates it based on the last obtained fault-free sensor reading, or proceeds with the system shut down. Next we describe the error detection mechanism and the input analysis in detail.

2.1. Error detection mechanism

The error detection mechanism is the most complex part of the FMS. It detects errors in sensor readings and classifies inputs as faulty or fault-free. The mechanism has a predefined architecture of so called *evaluation tests* as shown in Fig. 3. An evaluation test is a computation required to classify sensor reading as faulty or fault-free. Since we consider only homogeneous sensors, the same set of evaluation tests should be executed on input readings from each sensor. In case of heterogeneous sensors, the architecture of detection would be different – possibly different sets of evaluation tests would be needed for each sensor.

The architecture of evaluation tests determines the order of test execution based on the increasing test complexity. The first tests to be executed are called *simple tests*. An input reading may pass through several simple tests, which can be applied in any order.

When triggered, a simple test runs using solely an input reading from the sensor. After the test is executed, it is marked as passed for the current input, which in turn may trigger the execution of some other associated test defined by the architecture. After all simple test are executed, the control is passed to the *complex tests with the level of complexity 1*. The complex tests may use input readings from several sensors. However, all simple tests required for these sensors should be executed before any complex test is performed, as shown in Fig. 3.

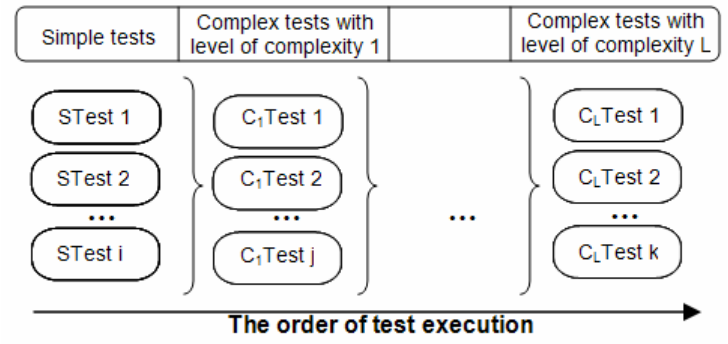


Fig. 3. Architecture of error detection mechanism

In general, there might be L+1 levels of test execution, where the last one applies the *complex tests with the level of complexity L*. The execution of a test of this level depends not only on the execution of the previous simple tests, but also on the execution of the complex tests with the level of complexity up to L-1. If the input requires several tests of the same complexity level, they can be executed in any order. However, all the applicable tests of the lower levels should be already executed. Hence, the detection procedure operates in stages, first executing all the simple tests associated with a certain input and then all complex tests of increasing complexity.

Both simple and complex evaluation tests may vary depending on the application domain. In avionics, the most commonly used simple tests for analogue sensors are the magnitude test, the rate test, and the predicted value test [11].

The *magnitude test* compares the value of an input reading with some predefined limit. If the limit is exceeded, the error is detected and the input is classified as faulty. Otherwise, it is considered to be fault-free.

Similarly, the *rate test* compares a rate of the value of an input reading over a fixed time interval with a predefined rate limit. The error is detected if this limit is exceeded.

The *predicted value test* compares the value of an input reading with a pre-computed, i.e., expected value. The error is detected if the discrepancy between the obtained reading and the calculated value is outside of predefined margins.

An example of a complex test applicable to homogeneous sensors is a dual sensor *difference test*, which compares readings from two sensors and detects an error if the difference between their values exceeds a predefined limit. According to the architecture of tests, this test is enabled only after the magnitude, rate, and predicted value tests are passed.

The results of tests performed at the error detection stage, are used in input analysis. This stage is shortly described in the next section.

2.2. Input analysis

The input analysis performs error recovery by masking faulty, yet recoverable, inputs. Hence, it prevents propagation of erroneous inputs further into the controller.

To explain how remedial actions work, during the error recovery, let us for simplicity consider a single sensor. Assume that a system is fault-free and receives a fault-free input. Then, *healthy action* is activated. It assigns the input the status *ok* and forwards it unchanged as an output. Let us now assume that the system receives a faulty input. Then, to not overreact, the FMS reserves a certain time limit for an input to recover. *Temporary action* assigns the input the status *suspected* and calculates the output using the last good value of this input obtained in the previous FMS cycle. Temporary actions are performed until the number of consequently received faulty inputs reaches some predefined limit. If the sensor before reaching this predefined limit starts to produce fault-free inputs, the system returns to the normal operating state and healthy actions are activated again. However, when the sensor fails to recover within the predefined number of cycles, then *confirmation action* is activated. It assigns the input the status *confirmed failed*. Then, the FMS proceeds with the control actions defined for freezing (stopping) the system.

2.3. The summary of the FMS behaviour

The behaviour of the FMS described above can be summarized as shown in Fig. 4.

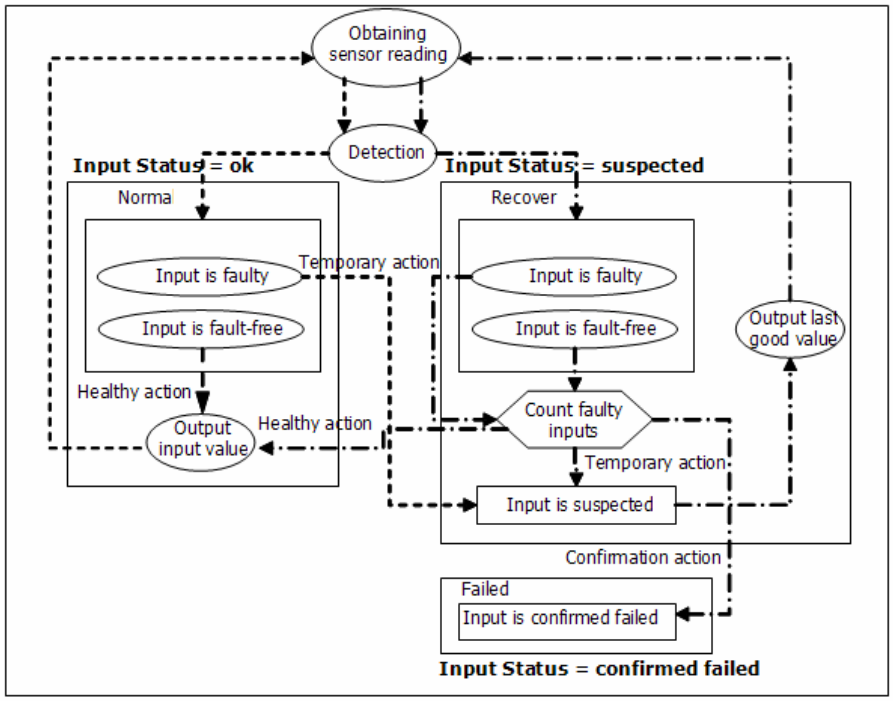


Fig. 4. The FMS behaviour pattern

For simplicity, it presents possible FMS operating cycles for a single sensor. Namely, it shows the flow of the detection decisions and the effect of the FMS actions after the

input is received from the system environment. In the absence of faults, the system operates in the state Normal. However, when the input is faulty, it transits to the state Recover. The system switches back to the state Normal, if the input has recovered, or stays in the state Recover, if the input is still suspected. The system enters the state Failed, if the input has failed to recover.

The given behavioural pattern can be easily generalized for N multiple sensors. In this case, the system failure state might be reached when several or all sensors have failed. The transition to the failure state corresponds to freezing the system or switching to a backup controller (if possible).

The behaviour pattern described above can be used in the product line development [3] of the controlling software for fault tolerant systems.

In the next section we introduce our modelling frameworks – the B Method and UML-B.

3. Frameworks for formal modelling and refinement – the B Method and UML-B

3.1. The B Method

The B Method [5, 12] (further referred to as B) is an approach for specifying and designing dependable software systems. It is based on set theory and first-order logic.

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

MACHINE *Name*
SETS *Types*
VARIABLES *v*
INVARIANT *I*
INITIALISATION *Init*

EVENTS
$E_1 = \ldots$
…
$E_n = \ldots$
END

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 the **INVARIANT** clause. The constraining predicates are conjoint by conjunction (denoted as $\wedge$). All types in B are represented by non-empty sets and hence set membership (denoted as $\in$) expresses typing constraint for a variable, e.g., $x \in TYPE$. Local types can be introduced

by enumerating the elements of the type, e.g., *TYPE = {element1, element2,…}* in the **SETS** clause. Sometimes, it is useful to introduce user's own definitions as the abbreviations for certain complex expressions. Such definitions can be formulated in the **DEFINITIONS** clause.

In this paper we adopt the event-based approach to system modelling [13, 14]. The *events* are defined in the **EVENTS** clause as the guarded operations of the form:

Event = **SELECT** cond **THEN** body **END**

Here *cond* is a state predicate on the variables, and *body* is a B statement describing how the state variables are affected by the event. If *cond* is satisfied, the behaviour of the event corresponds to the execution of its *body*. If *cond* is false at the current state then the event is disabled, i.e., its execution is blocked.

The events of the machine are atomic, meaning that, once an execution of an event has started, it cannot be interrupted until completion. The list of B statements that we are using to describe the computation in events is shown in Table 1.

Table 1. List of some commonly used B statements

Statement	***Description***
x := e	Assignment
x, y := e1, e2	Multiple assignment
S1 ; S2	Sequential composition
S1 || S2	Parallel execution of *S1* and *S2*
$x :\in T$	Nondeterministic assignment – assigns variable *x* arbitrary value from given set *T*
ANY *x* **WHERE** *Q* **THEN** *S* **END**	Nondeterministic block – introduces a new local variable *x* according to the predicate *Q*, which is then used in *S*

The first three constructs – assignments and sequential composition – have the standard meaning. The remaining constructs allow us to model parallel and nondeterministic behaviour in a specification. The detailed description of the B statements can be found elsewhere [12].

3.1.1. Verifying correctness

The B Method has the weakest precondition semantics [15]. Let S be a statement and P a postcondition predicate, i.e., a set of states which can be reached after executing S. Then [S]P represents the weakest precondition that guarantees establishing P after executing S.

The weakest precondition rules for a subset of B statements used in this paper are defined as follows:

$[skip]\ P \Leftrightarrow P$
$[x:=E]\ P \Leftrightarrow P[E/x]$
$[S1 \parallel S2]\ P \Leftrightarrow [S1]\ P \text{ and } [S2]\ P$
$[\text{ANY } x \text{ WHERE } Q \text{ THEN } S \text{ END}]\ P \Leftrightarrow \forall x\ (Q \Rightarrow [S]\ P)$

These rules serve as a basis for verifying correctness of specifications in B. Namely, to ensure correctness of a B machine, we should verify that the initialisation preserves the

invariant and that the invariant is valid, i.e., that there are some possible machine states which satisfy it. In other words, initialisation statement *INIT* must always guarantee the machine invariant *I*:

$$[INIT]\ I \Leftrightarrow true \qquad (1)$$

and

$$\exists x \,.\, I \Leftrightarrow true \qquad (2)$$

Moreover, we should ensure that every event E_i also preserves the invariant *I*:

$$I \wedge g_i \Rightarrow [S_i]\ I \qquad (3)$$

Here g_i is the guard and S_i is the body of the event E_i. Finally, we should also verify that the system is deadlock-free, i.e., whenever the invariant *I* holds, at least one event E_i is enabled:

$$I \Rightarrow \vee_1^n g_i \qquad (4)$$

Here g_i *(i=1,..., n)* is the guard of the event E_i. When all (1), (2), (3), and (4) are established by proofs, the correctness of the B machine is verified.

3.1.2. Refinement of B models

The formal development in B is based on stepwise refinement [16]. The development starts from an abstract model, which is then gradually augmented by implementation details. We build a sequence of more concrete models via a number of correctness preserving steps, called refinements. The result of a refinement step in B is a machine called **REFINEMENT**. Its structure coincides with the structure of the abstract machine. In addition, it explicitly states which machine it refines.

We refine a machine by refining its state and events. In this paper we extensively use data refinement – a general form of refinement, which allows us to change the state space of a machine. To replace abstract data structures with the refined ones, we define the refinement relation, so called *gluing invariant*, that explicitly states the connection between the newly introduced variables and the variables that they replace. The refinement relation constitutes a part of the invariant of the refining machine.

To ensure correctness of a refinement, we should verify that initialization and each event of the refining machine refine the initialization and the corresponding events of the more abstract machine:

$$[INIT']\neg[INIT]\neg I' \Leftrightarrow true \qquad (5)$$

and

$$I \wedge I' \wedge g_i' \ \Rightarrow\ g_i \wedge [S_i']\neg[S_i]\neg I' \qquad (6)$$

Here *INIT'* and *INIT* are respectively the initializations of the refining and the abstract machine, and *I'* and *I* are their invariants. g_i' and g_i are the guards of the refining and its corresponding abstract event, and S_i' and S_i are their bodies. Informally, (5) and (6) require that any execution of the concrete model should correspond to one of the allowed executions of the abstract model.

Moreover, we should verify that refinement does not introduce additional deadlocks:

$$I \wedge I' \wedge g_i \Rightarrow \bigvee_1^n g_i' \qquad (7)$$

where g_i' $(i=1,\ldots, n)$ is the guard of the event E_i'.

While developing a system by refinement, we often need to introduce new variables while leaving the existing data structure unaffected. This is a specific form of data refinement called superposition refinement [16]. It allows us to introduce new events describing computations on the new variables. A new event should refine the statement skip of the abstract machine. In other words, it cannot affect the old variables. In addition, we should guarantee that it does not take control indefinitely, i.e., that it terminates. To ensure this, we define the *variant* – a natural number expression which should be decreased by each execution of a new event. Therefore, for each new event we should verify that whenever the invariant I' of the refining machine holds and the guard g_i' of the new event is enabled, the execution of its body S_i' decreases the variant V:

$$I' \wedge g_i \Rightarrow [n:=V;\, S_i'](V < n) \quad \text{and} \quad V \in NAT \qquad (8)$$

A high degree of automation in verifying correctness is provided by the available tool support, e.g., AtelierB [9]. The verification can be completely automatic or user-assisted. In the former case, the tool generates the required proof obligations (1) – (8) and discharges them without user's help. In the latter case, the user proves certain proof obligations using the interactive prover provided by the tool.

3.2. UML-B

Since it was introduced by Rumbaugh, Jacobson and Booch, a graphical modelling language UML [7] has achieved significant dissemination among the industrial engineers worldwide. However, despite its popularity, it has been criticized for the lack of formal semantics and ambiguity. Although a freedom to interpret the UML graphical notation in different ways largely contributed to its growing popularity, today the UML community pays increasing attention to providing a precise semantics to UML. This motivated development of the UML-B [6].

UML-B is a specialisation of UML which defines a formal graphical modelling notation. It is based on the UML built-in light-weight extension mechanism called *profiles*, which allows customization of UML to different domains. *Profiles* are only allowed to contain tagged values, stereotypes, constraints and data types [17]. *Stereotypes* represent variations of existing UML modelling elements with the same form (having the same attributes and relationships) but with a modified intent [17]. A stereotype can have additional constraints on the base element it extends. It also has tagged values which add additional information to the stereotyped element. *Tagged values* are defined as stereotype properties expressed as name-value pairs, where the name is used as a tag.

UML-B customizes UML by introducing specific stereotypes that allow us to use the concepts of the formal modelling language B while creating UML models. Moreover, it adds a corresponding semantics to the predefined subset of UML entities. To support

modelling in UML-B, we use the Rational Rose tool [18] with an additional plugin – the translator tool U2B [8]. It allows us to automatically convert the UML-B models into their equivalent B models. We can then verify the model correctness by using the B tool support, e.g., AtelierB.

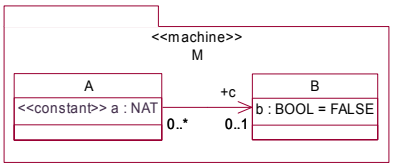
A UML-B model of a system is represented by *package diagram*, *class diagram* and *statechart*. Next we briefly describe each of these concepts.

3.2.1. UML-B Package

A UML-B package corresponds to a B machine or a refinement. Namely, a package with the stereotype <<machine>> represents a B machine, whereas a package with the stereotype <<refinement>> represents a B refinement. A package is used to group together a class diagram and the associated statecharts. It describes a system structure and behaviour at a certain level of abstraction. The overall UML-B development results in a set of packages and dependencies between them. The top level package represents an abstract view on the system, whereas the subsequent packages describe the system on the lower levels of abstraction.

3.2.2. UML-B Class diagram

A UML-B class diagram captures the functional requirements of the modelled system. The classes of a class diagram define sets, constants, variables and events of a B machine or a refinement, as shown in Fig. 5. Since a UML-B class represents a set of instances, its attributes are replicated for each instance. Hence, they correspond to B variables which type is a function from the instance set to the attribute type. For instance, the attribute b of the class B in Fig. 5 is typed in the corresponding B machine M as a total function from the set of instances of the class B to the type of b, i.e., b∈B_SET → BOOL.




```
MACHINE    M
DEFINITIONS
    type_invariant ==
            (A ∈ ℙ(A_SET) ∧ B ∈ ℙ(B_SET) ∧
             b ∈ B_SET ⟶ BOOL ∧ c ∈ A_SET ⇸ B_SET) ;
    invariant == (type_invariant)
SETS       A_SET; B_SET
CONSTANTS
            a
PROPERTIES
            a ∈ A_SET ⟶ NAT
VARIABLES
            A, B, b, c
INVARIANT
            invariant
INITIALISATION
            A := ∅ || B := ∅ || b:= B_SET × {FALSE} || c := ∅
END
```

Fig. 5. An example of a UML-B class diagram and its corresponding B machine

UML-B allows adding extra modelling information in the form of UML-B clauses for modelling B specific constructs that cannot be expressed diagrammatically. For instance, SETS is a clause that is usually attached to a class. It allows us to explicitly define attribute types as enumerated or deferred sets in addition to the already predefined types NAT, BOOL etc. Another frequently used clause is INVARIANT, which can be attached to a class to specify system properties.

UML-B class attributes can be stereotyped as <<constant>> or <<static>>. The attribute with the stereotype <<constant>> defines a B constant (e.g., see the constant a in Fig. 5). An attribute with the stereotype <<static>> belongs to a particular class but not its instance. Hence, its type is just the attribute type rather then a function from the set of instances. A class with the only one instance is called *utility class*. The attributes of a utility class are all static.

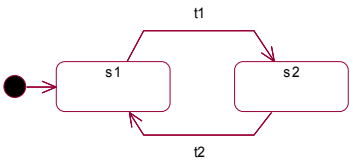
UML-B classes can be specialized by introducing the inheritance relationship between a (super)class and its subclasses. A subclass is then typed as a subset of current instances of its superclass.

Associations between classes are handled similarly to class attributes. They designate B variables that are typed depending on the association multiplicity. The detailed list of association representations can be found elsewhere [6].

The methods of a class explicitly describe its behaviour. However, every method specification is combined with the behaviour expressed in a statechart, associated with that class. UML-B introduces a stereotype for class methods as well. In addition to the already described stereotype <<static>>, a method can also have the stereotype <<definition>>. Such method then represents a (parameterized) B definition.

3.2.3. UML-B Statechart

A UML-B statechart describes the behaviour of the class to which it is attached. The name of a statechart represents a state variable in B (e.g., see the statechart a_state shown in Fig. 6).




```
MACHINE   M
DEFINITIONS
    type_invariant == ( ... ∧ a_state ∈ A ⟶ A_STATES) ;
    invariant == (type_invariant)
SETS        A_STATES={s1,s2}
VARIABLES
            ... , a_state
INVARIANT
            invariant
INITIALISATION
            ... ||  a_state := A ×  {s1}
EVENTS
t1 = ANY thisA WHERE thisA∈A
    THEN
            SELECT a_state(thisA)=s1
            THEN   a_state(thisA):=s2
            END
    END;
t2 = ANY thisA WHERE thisA∈A
    THEN
            SELECT a_state(thisA)=s2
            THEN   a_state(thisA):=s1
            END
    END
END
```

Fig. 6. The statechart a_state describing the behaviour of the class A from the class diagram in Fig. 5 and its corresponding B machine

The set of state names in a statechart defines the type of this variable (e.g., see A_STATES={s1,s2} in the same figure). The transitions between states are defined as B events that update the state variable. An event is enabled for execution only if the value of the state variable is equal to the state name from which there is a corresponding transition. For instance, the event t1 corresponding to the transition t1 in Fig. 6 occurs only if, for a particular instance thisA of the class A, the state variable a_state has the

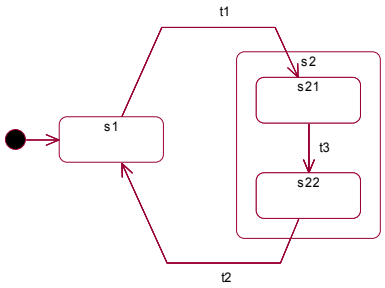
value s1, i.e., a_state(thisA)=s1. Furthermore, transitions may have additional guard conditions and actions. They correspond, respectively, to the guards and bodies of the appropriate B events.

3.2.4. Action and constraint language – μB

Obtaining a complete behavioural model in UML-B requires defining an additional notation – the specific action and constraint language μB (micro B) – allowing us to explicitly specify class methods, transition guards and actions, invariants etc. It is largely reusing the B abstract machine notation, shown in Table 1. The detailed description of the language will be introduced later.

3.2.5. Refinement in UML-B

UML-B adopts the stepwise refinement approach to system development. In particular, it supports superposition refinement [16], a special kind of data refinement, which allows us to extend the state space while preserving the existing data structures unaffected. The first step of refining a UML-B model is duplicating the current model in order to preserve the old class diagrams and statecharts. Then, we introduce new UML-B elements gradually by incorporating more details representing the system structure and behaviour.




```
REFINEMENT   M1
REFINES      M
DEFINITIONS
    type_invariant == ( ... ∧ s2_state ∈ A ⟶ S2_STATES) ;
    invariant == (type_invariant)
SETS         S2_STATES={s21,s22}
VARIABLES ... , s2_state
INVARIANT invariant
INITIALISATION
            ... ǁ  s2_state := A ×  {s21}
EVENTS
t1 = ANY thisA WHERE thisA∈A
    THEN
            SELECT a_state(thisA)=s1
            THEN  a_state(thisA):=s2 ǁ s2_state(thisA):=s21
            END
    END;
t3 = ANY thisA WHERE thisA∈A
    THEN
            SELECT a_state(thisA)=s2 ∧ s2_state(thisA)=s21
            THEN  s2_state(thisA):=s22
            END
    END;
t2 = ANY thisA WHERE thisA∈A
    THEN
            SELECT a_state(thisA)=s2 ∧ s2_state(thisA)=s22
            THEN  a_state(thisA):=s1
            END
    END
END
```

Fig. 7. Refinement of the statechart a_state from Fig. 6 and its corresponding B machine

On the abstract level the system is described by a set of class diagram and statecharts defined within the abstract <<machine>> package. While developing the system in a number of refinement steps, we create a chain of <<refinement>> packages, where each subsequent package is a refinement of the previous package. Each refinement step incorporates more details into the system structure (hence changing the class diagram), and the behaviour (affecting the statecharts). Specifically, a more detailed behaviour of

the system is modelled by hierarchically adding substates and new transitions to the existing statecharts.

Let us again consider the simple statechart a_state shown in Fig. 6. Assume that this is an abstract statechart modelling the behaviour of some hypothetical system. Assume also that we are refining the system by focusing on more detailed system behaviour within the state s2. Namely, after executing t1, the system goes first into the substate s21 of the state s2, and then, by executing t3, it reaches the substate s22 within the state s2. Finally, as previously specified, from the substate s22 it goes back to s1. To depict this refinement step graphically, we use hierarchical states as shown in Fig. 7.

To reduce their complexity, we represent statecharts together with hierarchical states only when introducing them for the first time. In later refinement steps, we adopt the flat statechart representation. Namely, we omit representing hierarchical states, but rather directly show the introduced substates and transitions between them. Refinement of UML-B statecharts is described in detail in [19].

A more detailed description of development in UML-B is given in the following section in the context of FMS development, where we also show how to obtain B models of the FMS from their UML-B counterparts and verify their correctness.

4. Developing the FMS by specification and refinement in UML-B

Outline of the FMS development in UML-B. Our UML-B development of the FMS is performed in phases. Each development phase is described by a set of UML-B models depicting the main structural and behavioural aspects of the FMS at a corresponding level of abstraction. The subsequent phases correspond to refinement steps. We describe each phase according to the following template. We start with a short elaboration on what is modelled at the current phase. Then, we describe the FMS structure by a class diagram. For each class in the class diagram, we explain its newly introduced attributes and what they are modelling. After describing the FMS structure, we continue by describing its behaviour via a statechart. The states of the statechart diagram correspond to the steps of the FMS operational cycle. The transitions between states describe the way the FMS cycle evolves.

While describing the subsequent development phases, we focus only on a refined FMS structure and behaviour. Description of each development phase concludes with the corresponding B machine obtained by translating the modelled FMS class diagram and statechart for that particular phase.

Next, we present the FMS development phases in detail.

4.1. Abstract specification of the FMS

In the 1st development phase we model the FMS cycle very abstractly: the FMS reads input values from the sensors, and it either calculates the output or fails. If the output is successfully calculated, the FMS cycle starts again. In case of a failure, the system enters the ‘freezing’ state.

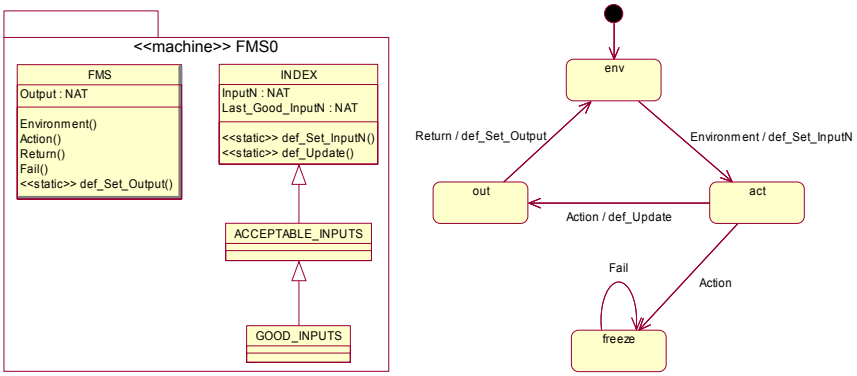


Fig. 8. The class diagram and the statechart fms_state for the 1st FMS development phase

Structure. The abstract FMS is modelled as the stereotyped package <<machine>> FMS0, as shown in Fig. 8. The corresponding class diagram of this package outlines the general system structure. The utility class FMS models the generic part of the system, i.e., properties of the whole system. For instance, the calculated system output is modelled as the attribute Output of the class FMS. Since it is the attribute of the utility class, it is stereotyped as <<static>>, meaning that the FMS calculates only one output. The concrete sensor readings, i.e., input values to the FMS are modelled as the attribute InputN of the class INDEX that models the monitored sensors.

To model the procedure of calculating the FMS output, we introduce an additional attribute to the class INDEX – Last_Good_InputN. Moreover, INDEX becomes a superclass of the subclass ACCEPTABLE_INPUTS, which models the inputs from sensors that did not fail. In other words, it considers only those inputs that are either fault-free or faulty but recoverable. Similarly, the subclass GOOD_INPUTS further partitions the space of ACCEPTABLE_INPUTS by modelling the fault-free inputs only. In B, a subclass is represented as an element of the power set of superclass instances.

Behaviour. Utility class FMS encapsulates the overall system behaviour within the attached fms_state statechart, as shown in Fig. 8. The names of the states within this statechart correspond to the phases of the FMS operating cycle. They have the following meaning:

- env – the state in which the FMS obtains inputs from the monitored sensors,
- act – the state in which the FMS analyses the inputs and performs recovery actions, if needed,
- out – the state in which the FMS calculates and sends the output to the controller,
- freeze – the state in which the FMS freezes (i.e., shuts down the system).

The transitions between states are directly related with the FMS main methods defined in the class FMS. In general, *main methods* are the methods of a utility class, which may trigger other specific methods stereotyped as definitions and often referred to as *actions*.

After the FMS is initialized, the FMS operating cycle starts by executing the main method Environment. It triggers the action def_Set_InputN, specified on the corresponding

transition in the statechart fms_state. Since it changes the values of the attribute InputN, it is defined as the method of the class INDEX. It simulates the inputs readings by arbitrarily setting the values of the attribute InputN for all instances of the INDEX class. The method is specified in μB as follows:

$$def\_Set\_InputN == (InputN :\in INDEX \longrightarrow NAT)$$

The prefix def designates that the method is a B definition. It is a shorthand notation for the stereotype <<definition>>. We use it in case when the method is already stereotyped as <<static>> to avoid duplication of stereotypes. Therefore, whenever def is used, the stereotype <<definition>> is applied.

After reading the input values, the FMS executes the method Action defined by the corresponding transition on the statechart. As a result of executing Action, the FMS either fails or continues by calculating the output. In case the FMS has successfully calculated the output, the execution of Action is extended with the def_Update action part. def_Update is defined as a static method of the class INDEX updating the values of the attributes of this class. Namely, it alters the set of the monitored sensors, considering in further FMS cycles only those which did not fail. The acceptable inputs are arbitrarily chosen from the set of inputs of all operational (non-failed) sensors, as shown in the following μB method definition:

$$\begin{aligned} &def\_Update == (ACCEPTABLE\_INPUTS :\in \{pp \mid pp \subseteq INDEX\}; \\ &\quad INDEX := ACCEPTABLE\_INPUTS \Vert \\ &\quad InputN := ACCEPTABLE\_INPUTS \lhd InputN \Vert \\ &\quad Last\_Good\_InputN := ACCEPTABLE\_INPUTS \lhd Last\_Good\_InputN) \end{aligned}$$

In further refinement steps the method Action will be refined to include the concrete mechanism for error detection and input analysis.

If the FMS has not failed at the current cycle, it produces the system output by executing the method Return, which corresponds to the statechart transition with the same name, as shown in Fig. 8. Its action part is the static method def_Set_Output, defined within the class FMS. The output is calculated based on the last good input values, which are systematically updated by the values of the fault-free inputs at each FMS cycle, as shown in the corresponding μB definition:

$$\begin{aligned} &def\_Set\_Output == (GOOD\_INPUTS :\in \{pp \mid pp \subseteq ACCEPTABLE\_INPUTS\}; \\ &\quad Last\_Good\_InputN := Last\_Good\_InputN \Leftarrow (GOOD\_INPUTS \lhd InputN); \\ &\quad Output :\in ran(Last\_Good\_InputN)) \end{aligned}$$

If the FMS fails, it does not resume its cyclic behaviour and stays in the failed (i.e., frozen) state.

To ensure system safety, we define an additional invariant within the specific invariant clause attached to the FMS class. This *safety invariant* specifies that the FMS can operate relying only on the sensors that have not failed. Formally, the invariant is expressed as follows:

safety invariant1==

(fms_state ∈ {out,env} ⟹ ∃ss . (ss∈INDEX ∧ ss∈ ACCEPTABLE_INPUTS)) ∧

(Indx=ø ⟹ fms_state=freeze)

In other words, when the FMS is in the states out or env, it processes the readings from at least one operational (non-failed) sensor. When there are no operational sensors, the FSM should be in the state freeze.

B machine for the phase 1. Before continuing with the FMS development in UML-B, we use the U2B [8] tool to automatically generate the B model from the obtained UML-B model. The resulting specification (shown in Fig. 9) is then verified using the prover tool supported by AtelierB [9].

```
MACHINE FMS0

DEFINITIONS
        type_invariant == ( INDEX∈ ℙ(NAT) ∧
                            ACCEPTABLE_INPUTS∈ ℙ(INDEX) ∧
                            GOOD_INPUTS∈ ℙ(ACCEPTABLE_INPUTS) ∧
                            fms_state ∈ FMS_STATE ∧
                            Output ∈ ℕ ∧
                            InputN ∈ INDEX ⟶ NAT ∧ Last_Good_InputN ∈ INDEX ⟶ NAT  );
        invariant == ( type_invariant );
        safety_invariant1==...;
        def_Set_InputN== ...;
        def_Update==...;
        def_Set_Output==...
SETS
        FMS_STATE={env,act,out,freeze}
VARIABLES
        INDEX, ACCEPTABLE_INPUTS, GOOD_INPUTS,
        fms_state, Output, InputN, Last_Good_InputN
INVARIANT
        invariant ∧ safety_invariant1
INITIALISATION
        ANY xx WHERE xx∈ℙ(NAT) THEN INDEX:=xx ||  ACCEPTABLE_INPUTS:=xx ||
        GOOD_INPUTS:=xx END || fms_state:=env ||  ...
EVENTS
Environment =
        SELECT   fms_state=env
        THEN     fms_state:=act || def_Set_InputN
        END;
Action =
        SELECT   fms_state=act
        THEN     fms_state:=out || def_Update
        WHEN     fms_state=act
        THEN     fms_state:=freeze
        END;
Return =
        SELECT   fms_state=out
        THEN     fms_state:=env ||  def_Set_Output
        END;
Fail =
        SELECT   fms_state=freeze
        THEN     skip
        END
END
```

Fig. 9. Excerpt from the abstract specification of the FMS in B

As explained in Sections 3.2.2 and 3.2.3, the UML-B package represented in Fig. 8 corresponds to the B machine FMS0. The classes and subclasses of the class diagram are translated into B variables, with the exception of a utility class representing the system in general. Let us observe that the names of the states from the statechart fms_state create an enumerated set, which in turn is used as the type for the state variable fms_state. Moreover, the transitions of this diagram create the corresponding B events. The invariant defines the types of the introduced variables and the desired safety property, i.e., the fact that the output is produced only if the inputs are fault-free or recovering. As a result of the first stage we have obtained an initial abstract specification defining the purpose of the FMS – to produce the output – and formulated the safety invariant that describes the conditions for doing this.

4.2. Phase 2: Introducing error detection by refinement

The 2nd FMS development phase introduces an abstract representation of error detection, performed by the FMS after obtaining the sensor readings. Hence, we enhance the specification of the FMS cycle to include the error detection mechanism. Namely, after reading the input values from the monitored sensors, the FMS performs the predefined error detection procedure on them. As a result, it classifies the inputs as faulty or fault-free. Then, it continues its operation as specified in the previous development phase.

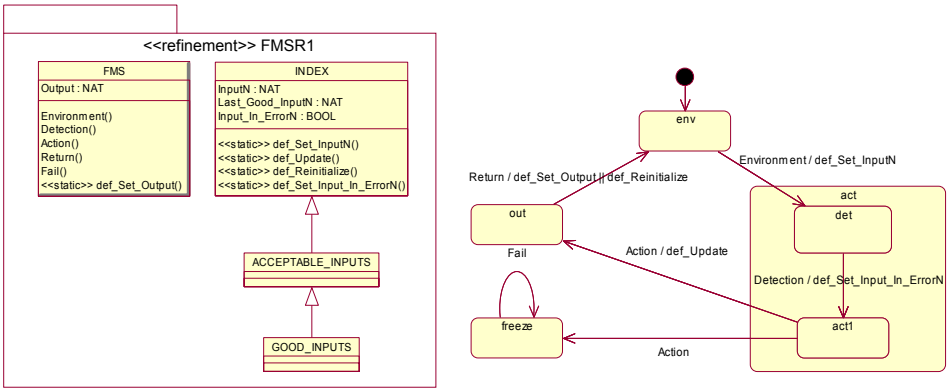


Fig. 10. The class diagram and the statechart fms_state for the 2nd FMS development phase

Structure. The FMS at the 2nd development phase is represented as the stereotyped package <<refinement>> FMSR1, as shown in Fig. 10. This allows us to connect the models created at the previous and the current development phase.

The class diagram of the package FMSR1 (see Fig. 10) preserves the structure defined at the previous development phase. However, to model the results of the error detection, we introduce the new attribute Input_In_ErrorN into the class INDEX. It is a boolean attribute, which is set to TRUE if the error is detected on the monitored input, and to FALSE otherwise. Initially, we consider that the inputs are fault-free.

Behaviour. To incorporate an abstract model of error detection, we modify the statechart by adding the new substates det and act1 within the existing state act, and the corresponding new transition Detection between them. We preserve the flat statechart representation as explained in Section 3.2.5. The resulting statechart is represented in Fig. 10.

Since the transition Detection describes the behaviour of the FMS, it is introduced as one of the main methods of the utility class FMS. The action part of this transition is defined as the <<static>> method def_Set_Input_In_ErrorN of the class INDEX. It nondeterministically assigns values to the variable Input_In_ErrorN:

$$def\_Set\_Input\_In\_ErrorN == (Input\_In\_ErrorN :\in INDEX \rightarrow BOOL)$$

The newly introduced attribute Input_In_ErrorN together with the existing attributes also needs to be updated to ensure that only acceptable (fault-free or faulty but recoverable) inputs are considered in the next FMS cycles. Hence, the method def_Update, specifying the action part of the main method Action, is refined as follows:

$$def\_Update == (<unchanged> \parallel Input\_In\_ErrorN := ACCEPTABLE\_INPUTS \lhd Input\_In\_ErrorN)$$

Here <unchanged> part refers to the expressions defined in the previous development phase for this particular method.

After successfully calculating the output, the FMS starts a new cycle. However, since at each FMS cycle all the inputs are initially considered fault-free, the attribute Input_In_ErrorN has to be reinitialized. Therefore, we introduce the method def_Reinitialize into the class INDEX. This method specifies the FMS actions taken while executing the main method Return. It sets the values of Input_In_ErrorN to FALSE, meaning that none of the monitored inputs is considered faulty before actual detection is performed, i.e.:

$$def\_Reinitialize == (Input\_In\_ErrorN := INDEX \times \{FALSE\})$$

B machine for the phase 2. After translating the created UML-B models for the 2nd FMS development phase into B, we obtain the refinement machine shown in Fig. 11.

```
REFINEMENT      FMSR1

REFINES         FMS0

DEFINITIONS
        type_invariant == ( Input_In_ErrorN ∈ INDEX ⟶ BOOL ∧ act_state ∈ ACT_STATE );
        invariant == ( type_invariant );
        def_Set_Input_In_ErrorN== ...;
        def_Update==...;
        def_Reinitialize==...
SETS
        ACT_STATE={det,act1}
VARIABLES
        ... Input_In_ErrorN, act_state
                                                                                ...
```

```
INVARIANT
        invariant ∧ ...
INITIALISATION
        ... ||  Input_In_ErrorN := INDEX × {FALSE} || act_state:=det
EVENTS
Environment =
        SELECT   fms_state=env
        THEN     fms_state:=act || act_state:=det || def_Set_InputN
        END;
Detection =
        SELECT   fms_state=act ∧ act_state=det
        THEN     act_state:=act1 || def_Set_Input_In_ErrorN
        END;
Action =
        SELECT   fms_state=act ∧ act_state=act1
        THEN     fms_state:=out || def_Update
        WHEN     fms_state=act ∧ act_state=act1
        THEN     fms_state:=freeze
        END;
Return =
        SELECT   fms_state=out
        THEN     fms_state:=env || def_Set_Output || def_Reinitialize
        END;
Fail = ...
END
```

Fig. 11. Excerpt from the refinement FMSR1 in B

The refinement FMSR1, created according to Fig. 10, introduces the new (sub)state variable act_state modelling the substates in the state act. Moreover, the existing event Action is refined correspondingly. Let us observe that the newly introduced transition Detection between the substates det and act1 becomes the event Detection, which refines the old event Action. Its body is specified by the action part def_Set_Input_In_ErrorN of the corresponding transition. As a result of the second phase we have abstractly modelled error detection procedure. The abstract specification has been refined by strengthening the invariant, introducing the new variable and computation on it.

4.3. Phase 3: Introducing input analysis by refinement

In the 3rd FMS development phase we introduce an abstract representation of input analysis performed by the FMS after the error detection. Once the FMS detects a faulty input, it uses the input analysis to decide whether it can be recovered or not. Then it saves the results of the analysis as the current input status and continues its operation either by calculating the output or failing when a certain predefined stopping condition is satisfied.

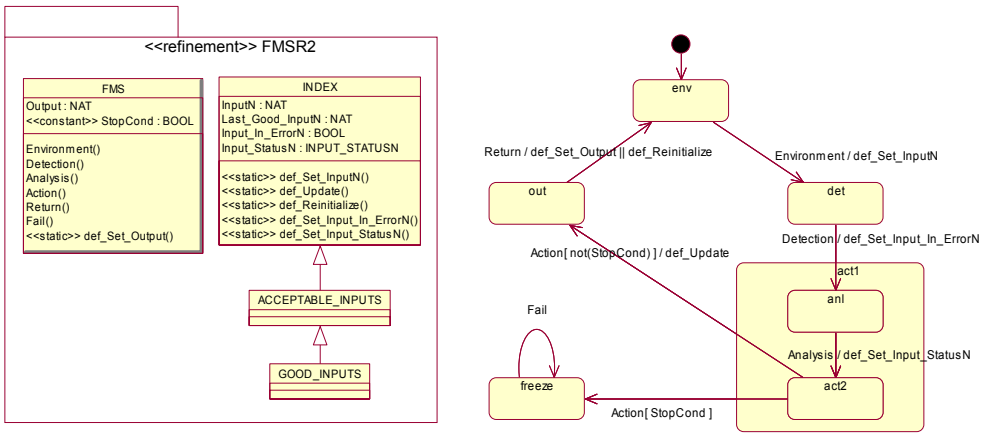


Fig. 12. The class diagram and the statechart fms_state for the 3rd FMS development phase

Structure. At this development phase, the FMS is represented as the stereotyped package <<refinement>> FMSR2 (see Fig. 12), which refines the package FMSR1 from the 2nd development phase. To introduce the details of the input analysis, we first modify the structure of our model by altering the class diagram of the package FMSR2. To model the obtained result of the input analysis, we add the attribute Input_StatusN to the class INDEX. The type of this attribute is introduced in the additional clause SETS attached to the class. It is defined as follows:

INPUT_STATUSN = {ok, suspected, confirmed_failed}

where ok stands for a fault-free input, suspected for a faulty yet recoverable input, and confirmed_failed represents a faulty but non-recoverable input.

At this phase, we also introduce an abstract representation of the stopping condition as a stereotyped boolean attribute <<constant>> StopCond in the class FMS. If StopCond is evaluated to TRUE, the system should be stopped (i.e., shut down).

Behaviour. To specify the input analysis in the FMS operating cycle, we refine the state act1 in the statechart fms_state. We add the new substates anl and act2 to the state act1 and the transition Analysis between them, as shown in Fig. 12. This transition describes the specific behaviour of the FMS and hence it is introduced as an additional main method in the utility class FMS. Its action part, explicitly describing the input analysis calculations, is defined as the <<static>> method def_Set_Input_StatusN of the class INDEX. The method produces a result of the input analysis on the basis of the error detection results from the previous step. Namely, the inputs detected as faulty become either suspected or confirmed_failed. On the other hand, the inputs detected as fault-free are given the status of either ok or suspected. More precisely:

def_Set_Input_StatusN ==
(Input_StatusN :∈ {ff | ff ∈ INDEX → INPUT_STATUSN ∧
∀ee.(ee∈INDEX ∧ Input_In_ErrorN(ee)=FALSE⇒ff(ee)∈{ok,suspected}) ∧
∀ee.(ee∈INDEX ∧ Input_In_ErrorN(ee)=FALSE⇒ff(ee)∈{suspected,confirmed_failed})})

At the previous development phases, we defined the abstract subclasses ACCEPTABLE_INPUTS and GOOD_INPUTS. Now we can refine them using the information about the input status. Namely, we define the acceptable inputs as the inputs whose status is ok or suspected. Similarly, the good inputs are the inputs whose status is ok. Then, the methods determining those particular instances of the class INDEX that belong to these two subclasses are refined as follows:

def_Update == (<unchanged> ||
ACCEPTABLE_INPUTS := Input_StatusN~[{ok,suspected}] ;
Input_StatusN := Input_StatusN⩥{ok,suspected})

def_Set_Output == (<unchanged> || GOOD_INPUTS := Input_StatusN~[{ok}])

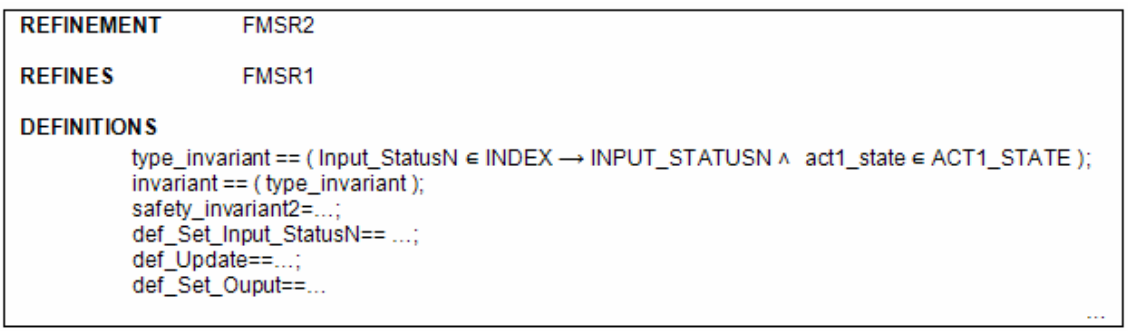
In addition to refining the values assigned to the subclass ACCEPTABLE_INPUTS, the method def_Update updates the attribute Input_StatusN so that only inputs which did not fail are considered in the subsequent FMS cycle.

Since the controller of the system relies only on the inputs it obtains from the FMS, to guarantee system safety, we define an additional *safety invariant*. It specifies an error confinement condition for the FMS:

safety invariant2==
(fms_state=act ∧ act_state=act1 ∧ act1_state=act2 ⟹
∀ee.(ee∈INDEX ⟹
(Input_In_ErrorN(ee)=FALSE ⩞ Input_StatusN(ee)∈{ok,suspected}) ∧
(Input_In_ErrorN(ee)=TRUE ∧ Input_StatusN(ee)∈{suspected,confirmed_failed})))

This predicate states that, whenever the FMS is in the substate act2 and some input ee is detected fault-free, the value assigned to the variable Input_StatusN is either ok or suspected. Similarly, if an error is detected for some input ee, the value assigned to the variable Input_StatusN is either suspected or confirmed_failed.

B machine for the phase 3. The obtained B machine for the 3rd FMS development phase is shown in Fig. 13.



```
SETS
        ACT1_STATE={anl,act2}
CONSTANTS
        StopCond
PROPERTIES
        StopCond ∈ BOOL
VARIABLES
        ... Input_StatusN , act1_state
INVARIANT
        invariant ∧ safety_invariant2 ∧ ...
INITIALISATION
        ... ||  Input_StatusN := INDEX × {ok} || act1_state:=anl
EVENTS
Environment =
        SELECT   fms_state=env
        THEN     fms_state:=act || act_state:=det || def_Set_InputN
        END;
Detection =
        SELECT   fms_state=act ∧ act_state=det
        THEN     act_state:=act1 || act1_state:=anl || def_Set_Input_In_ErrorN
        END;
Analysis =
        SELECT   fms_state=act ∧ act_state=act1 ∧ act1_state=anl
        THEN     act1_state:=act2 || def_Set_Input_StatusN
        END;
Action =
        SELECT   fms_state=act ∧ act_state=act1 ∧ act1_state=act2 ∧ ¬(StopCond)
        THEN     fms_state:=out || def_Update
        WHEN     fms_state=act ∧ act_state=act1 ∧ act1_state=act2 ∧ StopCond
        THEN     fms_state:=freeze
        END;
Return = ... ;
Fail = ...
END
```

Fig. 13. Excerpt from the refinement FMSR2 in B

The result of this phase is the formal specification which now contains an abstract representation of the input analysis. Similarly to the previous refinement step, the refinement FMSR2 further unfolds the system state. In the specification it is reflected by the new variable act1_state and the corresponding computation on it defined in the new event Analysis.

4.4. Phase 4: Refining the input analysis

The 4th FMS development phase further refines the input analysis. In the previous phase, we defined the input analysis as an atomic action, which assigns the statuses of all monitored sensors at once, where in reality the sensor inputs are analyzed independently, i.e., one by one, until all the inputs are analyzed (processed). In this phase, we also specify in detail the procedure of determining the input status. It is based on using a specific counting mechanism, which re-evaluates the status of the analyzed inputs at each FMS cycle, allowing us to introduce input recovery. As a result, some of the suspected inputs can be recovered and used in the next FMS cycle.

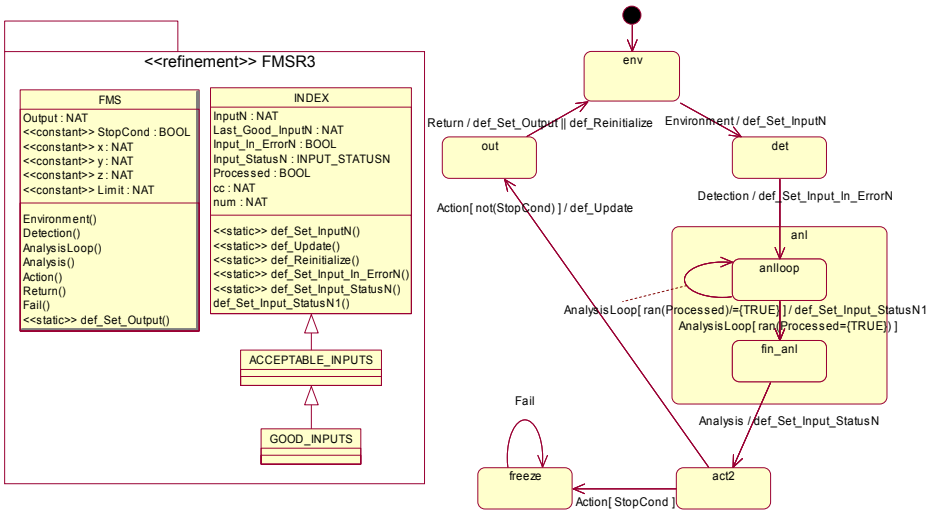


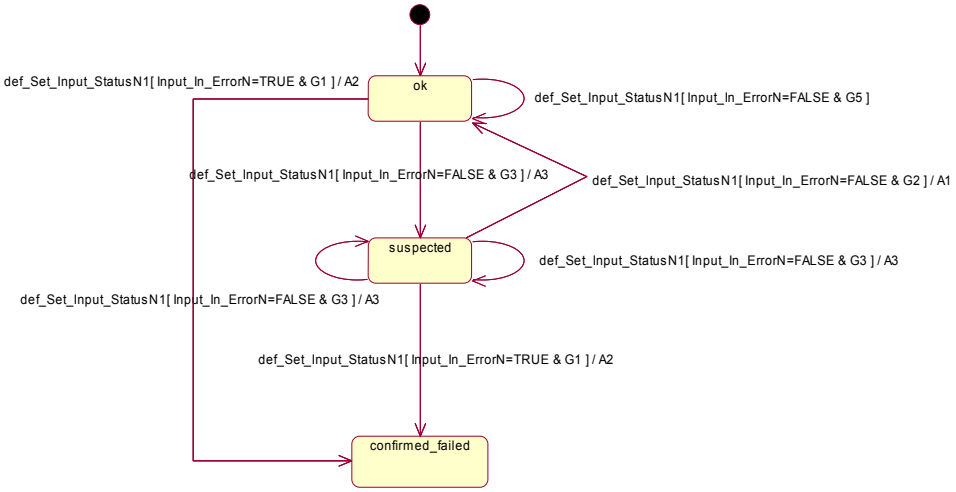
Fig. 14. The class diagram and the statechart fms_state for the 4th FMS development phase

Structure. The refined FMS at this development phase is represented by the stereotyped package <<refinement>> FMSR3, as shown in Fig. 14. The structure of the FMS defined previously by the class diagram in Fig. 12 is preserved. To model realistic input analysis and a counting mechanism (i.e., recovery) required for the input analysis, we extend the class diagram with additional attributes. Let us describe the introduced attributes in detail.

First, we focus on the data structures needed to model the step-by-step input analysis. Since the analysis is performed on each input (i.e., each instance of the class INDEX), we need to keep the record of those inputs that are already analysed within the current operating cycle. Hence, in the class INDEX we introduce the boolean attribute Processed. It is set to TRUE, if the input has been processed, and to FALSE otherwise.

The attributes introduced to support the counting mechanism should enable controlled input recovery. To ensure error recovery termination, we need a counter that keeps track of input behaviour. To achieve this, we introduce the attribute cc into the class INDEX. It accumulates the values determining how trustworthy a particular input is. These values depend on the result of the error detection. Namely, if the input is determined as faulty, its trustworthiness is "measured" by a certain predefined value x, generic for the system and hence introduced as a constant attribute in the class FMS. On the other hand, if the input is determined as fault-free, its trustworthiness is evaluated by another predefined value y, introduced similarly as the attribute x. To ensure finite error recovery, we should keep cc below the predefined upper limit z, which is introduced as an additional configuration parameter. Moreover, we introduce an additional counter num, which counts the number of the consequent recovery cycles for each recovering input. In addition, we specify the maximum number of the allowed recovery cycles for all inputs as the constant attribute Limit of the class FMS. Both num and Limit are specific for the whole system and hence are defined as attributes of the class FMS.

Behaviour. To model the input analysis, we need to extend the FMS state space by adding a new hierarchical state to the existing statechart fms_state. Specifically, we refine the state anl by unfolding its substates anlloop and fin_anl, as shown in Fig. 14. A new transition between these substates specifies an additional FMS main method – AnalysisLoop. In general, after performing the error detection, the FMS starts analyzing the inputs one by one, until all the inputs are processed. Hence, the guard ran(Processed)={TRUE} of the transition AnalysisLoop defines the terminating condition for the analysis. The FMS implements the gradual input analysis as specified by a newly introduced statechart attached to the class INDEX – Input_StatusN1, as represented in Fig. 15. Let us observe that it will be translated into a variable in the B specification, thus allowing us to save the intermediate results of the analysis.



where:

G1=(num+1 ≥ Limit OR cc+x ≥ z) G2=(num+1<Limit ∧ cc-y=0) G3=(num+1<Limit ∧ cc-y>0) G4=(num+1<Limit ∧ cc+x<z) G5=(num=0 ∧ cc=0)	A1=(num:=0 ‖ cc:=cc-y) A2=(num:=num+1 ‖ cc:=cc+x) A3=(num:=num+1 ‖ cc:=cc-y)

Fig. 15. The statechart Input_StatusN1 specifying the behaviour of the class INDEX

The statechart Input_StatusN1, shown in Fig. 15, describes a deterministic procedure of determining the status of a single input. Hence, the states of this statechart are the input statuses: ok, suspected, and confirmed_failed, as introduced in the previous phases. The input status changes depending on the values of the configuration parameters x, y, z, cc, Limit, and num. For clarity, in the statechart we use the abbreviations to express the guards and the corresponding actions specifying the transition def_Set_Input_StatusN1. Let us observe that def_Set_Input_StatusN1 is defined as the method of the class INDEX, and it is not stereotyped as <<static>>, as the other definitions in the model. This is due to the fact that it operates on the instances of the class INDEX (single inputs), rather then on the whole class.

Let us describe the behaviour represented by the statechart Input_StatusN1 in detail. Initially, all inputs are considered to be ok. After the error detection determines for each input whether it is faulty or not, the analysis uses these results to calculate their statuses.

If the input is determined as faulty, the FMS increments the counter cc of the input trustworthiness by x. On the other hand, if the input is fault-free, the FMS decrements cc by y. Then it examines the value of cc. If it is equal to zero for the given input, then the status of that input is ok. If 0<cc<z, then the input status is suspected (i.e., the input is recovering). Otherwise, the input status is considered to be confirmed_failed. The configuration parameters x, y, and z are set for each system individually, after observing its real performance. By setting the value of x higher then the value of y, the counter cc is biased towards failure. To prohibit the counter cc oscillating between some values and never reaching the limit z or zero for recovering inputs, the counter num should be kept below Limit. As soon as it exceeds this limit, the recovery terminates and, if cc is different from zero, the input status becomes confirmed_failed.

In our previous development phases, we defined the attribute Input_StatusN modelling the results of the input analysis performed within the method def_Set_Input_StatusN. Now, the statechart Input_StatusN1 describes the change of the input status for a single input. To establish the refinement relationship between the old attribute Input_StatusN and the newly introduced statechart Input_StatusN1, we refine the main method Analysis from the previous development phase. Namely, after all inputs are analyzed (i.e., AnalysisLoop is completed), the intermediate results of the analysis are assigned to the attribute Input_StatusN. This is specified within the action def_Set_Input_StatusN of the method Analysis:

def_Set_Input_StatusN == (Input_StatusN := Input_StatusN1)

Then, the FMS performs previously defined Action. It includes updating the newly introduced attributes Processed, cc, and num and the obtained results of the input analysis – Input_StatusN1. It allows us to discard the information about those inputs whose status has been confirmed_failed in the following FMS cycles. Hence, we refine the action def_Update of the main method Action:

def_Update == (<unchanged> ‖
Input_StatusN1 := Input_StatusN ▷ {ok,suspected} ‖
Processed := ACCEPTABLE_INPUTS ◁ Processed ‖
cc := ACCEPTABLE_INPUTS ◁ cc ‖
num := ACCEPTABLE_INPUTS ◁ num)

After that, the FMS cycle continues as specified earlier. However, since each new FMS operating cycle should start with unprocessed inputs, the attribute Processed should be reinitialized. The action def_Reinitialize of the otherwise unchanged main method Return, is refined to implement this requirement, as follows:

def_Reinitialize == (<unchanged> ‖ Processed := INDEX × {FALSE})

The safety invariants specified at the previous development phases are preserved. However, we define an additional safety invariant specifying the extended error confinement condition for the individual input analysis as follows:

safety invariant3==

(∀ee.(ee∈INDEX ∧ Processed(ee)=TRUE ∧ Input_In_ErrorN(ee)=TRUE ⟹
Input_StatusN1(ee)∈{suspected,confirmed_failed}) ∧
∀ee.(ee∈INDEX ∧ Processed(ee)=TRUE ∧ Input_In_ErrorN(ee)=FALSE ⟹
. Input_StatusN1(ee)∈{ok,suspected}))

B machine for the phase 4. Translating the created UML-B models for this development phase into B results in the refinement machine FMSR3, as represented in Fig. 16.

```
REFINEMENT      FMSR3

REFINES         FMSR2

DEFINITIONS
        type_invariant == ( Input_StatusN1 ∈ INDEX → INPUT_STATUSN ∧ anl_state ∈ ANL_STATE ∧
                            Processed ∈ INDEX → BOOL ∧ num ∈ INDEX → NAT ∧ cc ∈ INDEX → NAT );
        invariant == ( type_invariant );
        safety_invariant3=...;
        def_Set_Input_StatusN== ...;
        def_Update==...;
        def_Reinitialize==...;
        def_Set_Input_StatusN1==
        ( ANY ii WHERE ii∈INDEX
          THEN    SELECT Processed(ii)=FALSE
                  THEN
                     SELECT Input_StatusN1(ii)=ok ∧ Input_In_ErrorN(ii)=FALSE ∧ G5
                     THEN  skip
                     WHEN  Input_StatusN1(ii)=ok ∧ Input_In_ErrorN(ii)=TRUE ∧ G4
                     THEN  Input_StatusN1(ii):=suspected || A2
                     WHEN  Input_StatusN1(ii)=ok ∧ Input_In_ErrorN(ii)=TRUE ∧ G1
                     THEN  Input_StatusN1(ii):=confirmed_failed || A2
                     WHEN  Input_StatusN1(ii)=suspected ∧ Input_In_ErrorN(ii)=FALSE ∧ G2
                     THEN  Input_StatusN1(ii):=ok || A1
                     WHEN  Input_StatusN1(ii)=suspected ∧ Input_In_ErrorN(ii)=FALSE ∧ G3
                     THEN  A3
                     WHEN  Input_StatusN1(ii)=suspected ∧ Input_In_ErrorN(ii)=TRUE ∧ G4
                     THEN  A2
                     WHEN  Input_StatusN1(ii)=suspected ∧ Input_In_ErrorN(ii)=TRUE ∧ G1
                     THEN  Input_StatusN1(ii):=confirmed_failed || A2
                     END || Processed(ii):=TRUE
                  END
          END )
SETS
        ANL_STATE={anlloop,fin_anl}
CONSTANTS
        x, y, z, Limit
PROPERTIES
        x∈NAT ∧ y∈NAT ∧ z∈NAT ∧ Limit∈NAT
VARIABLES
        ... Input_StatusN1, anl_state, Processed, num, cc
INVARIANT
        invariant ∧ safety_invariant3 ...
                                                                                    ...
```

```
INITIALISATION
        ... || Input_StatusN1 := INDEX × {ok} || anl_state:=anlloop || Processed := INDEX × {FALSE} ||
        num:= INDEX × {0} || cc:=INDEX × {0}
EVENTS
Environment =...;
Detection =...
        SELECT  fms_state=act ∧ act_state=det
        THEN    act_state:=act1 || act1_state:=anl || anl_state:=anlloop || def_Set_Input_In_ErrorN
        END;
AnalysisLoop =
        SELECT  fms_state=act ∧ act_state=act1 ∧ act1_state=anl ∧ anl_state=anlloop ∧
                ran(Processed)≠{TRUE}
        THEN    def_Set_Input_StatusN1
        WHEN    fms_state=act ∧ act_state=act1 ∧ act1_state=anl ∧ anl_state=anlloop ∧
                ran(Processed)={TRUE}
        THEN    anl_state:=fin_anl
        END;
Analysis =
        SELECT  fms_state=act ∧ act_state=act1 ∧ act1_state=anl ∧ anl_state=fin_anl
        THEN    act1_state:=act2 || def_Set_Input_StatusN
        END;
Action =...;
Return =...;
Fail =...
END
```

Fig. 16. Excerpt from the refinement FMSR3 in B

The result of the fourth phase is a refined specification containing the detailed representation of the input analysis and input classification. We further refined the event Analysis and introduced the new event AnalysisLoop. This allowed us to model the input analysis done individually, i.e., one-by-one, on each sensor and the procedure for determining the input status.

4.5. Phase 5: Refining the error detection – introducing the evaluation tests

In the 2nd development phase, we already abstractly specified the error detection part of the FMS. In this development phase, we further refine it by introducing the evaluation tests that are consecutively applied on the obtained inputs. They determine the result of the detection for each input separately, rather than for all of them at once, as modelled in the previous phases. After executing all predefined tests on the obtained inputs, the FMS proceeds with the input analysis based on the results of the applied tests, as described earlier.

Structure. The FMS at the 5th development phase is represented as the stereotyped package <<refinement>> FMSR4, refining FMSR3 from the 4th development phase. To model evaluation tests, we introduce an additional class into our previous class diagram – the class TEST. The tests applied to the inputs obtained by the FMS form a specific architecture expressing the dependencies between them, as explained in Section 2.1. These dependencies are modelled as the association ComplexTest, stereotyped as <<constant>>. This allows us to distinguish between tests that are independent and those

that depend on the results of other tests. The additional constraint attached to the association ComplexTest requires that a test can not depend on itself.

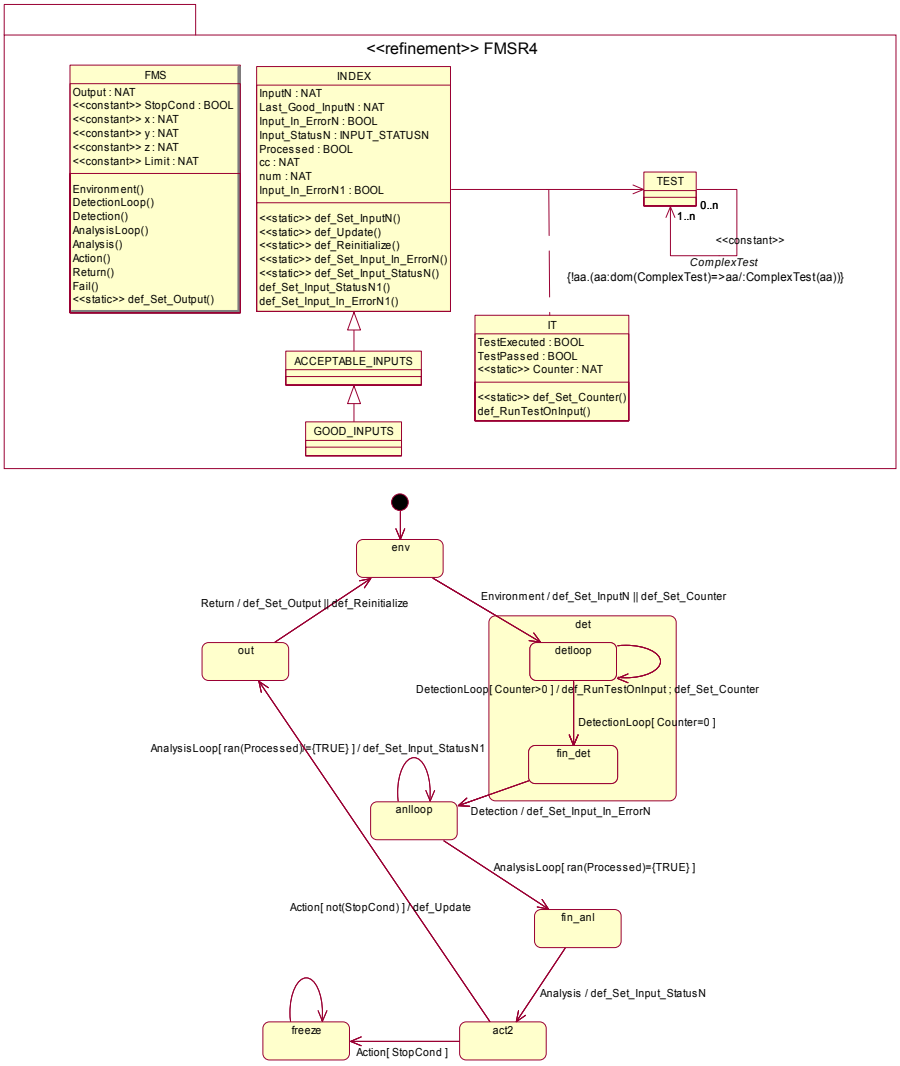


Fig. 17. The class diagram and the statechart fms_state for the 5th FMS development phase

We need to keep track of all tested inputs and their test results. Hence, we introduce the association class IT, modelling the set of all (test, index) pairs in the following way. If it is an instance of IT, then it.test refers to its first element, and it.index to the second element. For each such instance, we first define whether the particular input has been tested by its corresponding test. We model this by introducing the boolean attribute TestExecuted into the class IT. For each instance the attribute either has the value TRUE, if it.index has been tested by it.test, or FALSE otherwise. Similarly, the boolean attribute TestPassed models the results of test execution for instances of IT. The attribute has the value TRUE, if the test has been successfully passed by the corresponding input, and FALSE otherwise.

Since the decision whether some particular input is faulty may be based on more than single test execution, in the class INDEX we introduce the attribute Input_In_ErrorN1, which represents the final result of the error detection based on all tests executed on that input. In addition, to model the terminating condition for the error detection, we introduce the static attribute Counter in the class IT. This attribute defines the number of the remaining tests still to be executed on the inputs from the monitored sensors.

Behaviour. Refinement of the error detection introduces the substates detloop and fin_det within the state det, as shown in Fig. 17. The transition DetectionLoop between these substates is specified as an additional FMS main method. In general, after obtaining the inputs from the monitored sensors, the FMS proceeds with error detection on single inputs, until all the inputs are determined faulty or fault-free, i.e., until all the tests required to be executed on each input are applied. Hence, the guard Counter=0 of the transition DetectionLoop defines when the detection process is completed. The value of the Counter is set prior to the error detection by the action def_ Set_Counter within the Environment main method. In addition, Counter is re-evaluated after each detection loop by the same action. This action sets Counter to the number of IT instances that have not been tested yet. Precisely, it is defined as follows:

def_Set_Counter ==
(Counter := **card** ({ it | it ∈ IT ∧ TestExecuted(it)=FALSE ∧
Input_In_ErrorN1(it.index)=FALSE ∧
(it.test∈dom(ComplexTest)⟹
∀tt.(tt∈ComplexTest(it.test)⟹TestExecuted(it.index,tt)=TRUE)) }))

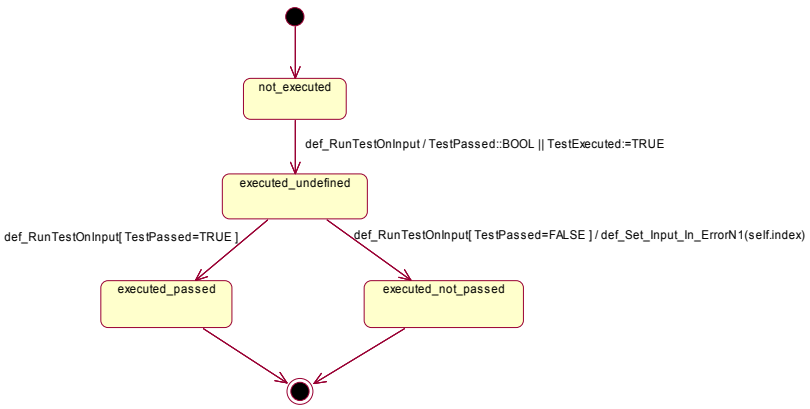


Fig. 18. The statechart diagram TestOnInput

After determining the initial number of DetectionLoop iterations, the FMS implements step-by-step error detection, as specified by a newly introduced statechart attached to the class IT – TestOnInput (see Fig. 18). In the corresponding B specification, TestOnInput becomes a state variable, whose values denote the states of the instances of the class IT.

Initially, none of the tests is executed. The method def_RunTestOnInput of the class IT specifies the detection in detail. It is associated with the transitions of the statechart TestOnInput. Since it operates over given instances (i.e., pairs (test, index)), it is not

stereotyped as <<static>>. The implemented mechanism of the error detection is shown in Fig. 19.

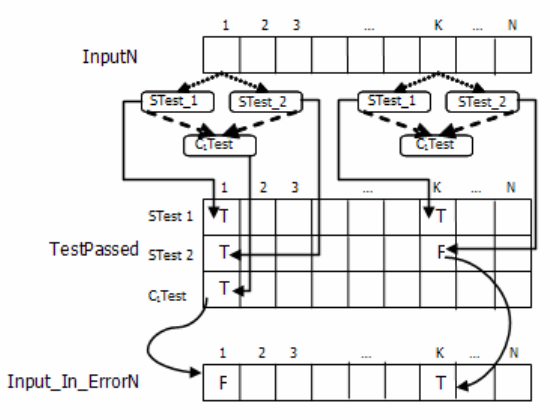


Fig. 19. Process of the error detection

Let us explain the error detection procedure over sensor inputs. Assume that, according to the architecture of tests, there are simple tests STest_1 and STest_2, and a complex test C1Test. Both simple tests have successfully passed on the input 1, hence, the values in the matrix TestPassed for these tests on input 1 are T (i.e., TRUE). After the simple tests have successfully passed on input 1, the complex test can be executed. Its result determines whether the input is in error or not. Since C1Test has successfully passed on input 1, the input 1 is fault-free, i.e., Input_In_ErrorN for input 1 is F (i.e, FALSE).

Let us now observe the error detection procedure for the input K. Since the test STest_1 has successfully passed, the value in the matrix TestPassed for this input is T. However, since STest_2 has failed, the complex test C1Test is not executed. Overall, the input K is considered to be faulty, i.e., Input_In_ErrorN is T for input K.

According to the description above, the method def_RunTestOnInput results in distinguishing between faulty and fault-free inputs. Namely, when an input has successfully passed a certain test (TestPassed=TRUE), the value of Input_In_ErrorN1 stays unchanged, i.e., it remains FALSE as specified initially. However, if the test has failed (TestPassed=FALSE), the value of Input_In_ErrorN1 for the tested input is set to TRUE by the corresponding action def_Set_Input_In_ErrorN1. Since it alters the attribute of the class INDEX, it is defined as the method of this class. The μB definition of the method is as follows:

def_Set_Input_In_ErrorN1(ii) == (Input_In_ErrorN1(ii):=TRUE)

Let us observe that this method is parameterized by a particular input. In the statechart TestOnInput in Fig. 18, def_Set_Input_In_ErrorN1 is parameterized by self.index. Here, self is a μB extension referring to the current class instance whose behaviour is described by the statechart, i.e., it is the current instance of IT. However, since IT is an association class, its instances are pairs (test,index). Hence, self.index represents the current input.

This refinement step focuses on refining the main method Detection. In the previous models, its action def_Set_Input_In_ErrorN nondeterministically set the error detection results at once, on all obtained inputs. Now, however, these results are determined

consecutively, for each single input and then after accumulated in Input_In_ErrorN1 assigned to Input_In_ErrorN:

def_Set_Input_In_ErrorN == (Input_In_ErrorN := Input_In_ErrorN1)

After detection is completed, the FMS continues with the input analysis as specified earlier. However, the newly introduced attributes need to be updated together with the existing attributes so that the failed inputs are no longer used in the FMS cycle. Hence, we refine the method def_Update as shown below:

def_Update == (<unchanged> ||
Input_In_ErrorN1 := ACCEPTABLE_INPUTS ◁ Input_In_ErrorN1 ||
IT := (TEST×ACCEPTABLE_INPUTS) ;
TestPassed := IT ⩤ TestPassed ||
TestExecuted := IT ⩤ TestExecuted ||
TestOnInput := IT ⩤ TestOnInput)

Then, the FMS calculates the output and starts the new operating cycle. However, before a new cycle starts, the newly introduced attributes TestExecuted, TestPassed and Input_In_ErrorN1 need to be reinitialized. Hence, the existing method def_Reinitialize is refined to implement this as follows:

def_Reinitialize == (<unchanged> ||
Input_In_ErrorN1 := INDEX × {FALSE} ||
TestPassed := IT×{FALSE} || TestExecuted := IT×{FALSE})

The safety invariant of this development phase (safety_invariant4) guarantees that, if any of the tests applied on a certain input has failed, the input is considered in error:

∀it.(it∈IT ∧ TestOnInput(it)=executed_not_passed ⟹ Input_In_ErrorN1(it.index)=TRUE)

However, this would be insufficient without additionally requiring that, for some input to be fault-free, it should successfully pass all the executed tests:

∀it.(it∈IT ∧ TestOnInput(it)=executed_passed ⟹ Input_In_ErrorN1(it.index)=FALSE)

B machine for the phase 5. The B machine, obtained by translating the created UML-B models for this development phase, is shown in Fig. 20.

REFINEMENT FMSR4

REFINES FMSR3

DEFINITIONS

type_invariant == (Input_In_ErrorN1 ∈ INDEX → BOOL ∧ det_state ∈ DET_STATE ∧
IT ⊆ TEST×INDEX ∧ TestExecuted ∈ IT → BOOL ∧ Counter ∈ NAT ∧
TestPassed ∈ IT → BOOL ∧ TestOnInput ∈ IT → TESTONINPUT_STATE);
invariant == (type_invariant);

...

```
safety_invariant4=...;
        def_Update==...;
        def_Reinitialize==...;
        def_Set_Input_In_ErrorN==...;
        def_Set_Input_In_ErrorN1==...;
        def_Set_Counter==...;
        def_RunTestOnInput ==
          ( ANY it,ii,tt WHERE it∈IT ∧ ii∈INDEX ∧ tt∈TEST ∧ it=(tt↦ii)
            THEN
                SELECT
                        TestExecuted(it)=FALSE ∧ Input_In_ErrorN1(ii)=FALSE ∧
                        (tt∈dom(ComplexTest)⟹
                                ∀pp.(pp∈ComplexTest(tt)⟹TestExecuted(pp,ii)=TRUE))
                THEN
                        SELECT TestOnInput(it)=not_executed
                        THEN   TestOnInput(it):=executed_undefined ||
                            ANY xx WHERE xx∈BOOL THEN TestPassed(it):=xx END ||
                            TestExecuted(it):=TRUE
                        WHEN   TestOnInput(it)=executed_undefined ∧ TestPassed(it)=FALSE
                        THEN   TestOnInput(it):=executed_not_passed ||
                            def_Set_Input_In_ErrorN1(ii)
                        WHEN   TestOnInput(it)=executed_undefined ∧ TestPassed(it)=TRUE
                        THEN   TestOnInput(it):=executed_passed
                        END
                END
            END )
SETS
        DET_STATE={detloop,fin_det}; TEST;
        TESTONINPUT_STATE={not_executed,executed_undefined,executed_passed,executed_not_passed}
CONSTANTS
        ..., ComplexTest
PROPERTIES
        ..., ComplexTest ∈ TEST ⇸ ℙ(TEST) ∧ ∀(aa).(aa∈dom(ComplexTest)⟹aa∉ComplexTest(aa))
VARIABLES
        ..., Input_In_ErrorN1, det_state, IT, TestExecuted, TestPassed, Counter, TestOnInput
INVARIANT
        invariant ∧ safety_invariant4 ...
INITIALISATION
        ... ||  Input_In_ErrorN1 := INDEX × {FALSE} || det_state:=detloop ||
            IT := TEST×INDEX || TestExecuted := (TEST×INDEX)×{FALSE} ||
            TestPassed := (TEST×INDEX)×{FALSE} || Counter := 0 || TestOnInput := IT×{not_executed}
EVENTS
Environment =
        SELECT  fms_state=env
        THEN    fms_state:=act || act_state:=det || det_state:=detloop ||
                def_Set_InputN || def_Set_Counter
        END;
DetectionLoop =
        SELECT  fms_state=act ∧ act_state=det ∧ det_state=detloop ∧ Counter>0
        THEN    def_RunTestOnInput ; def_Set_Counter
        WHEN    fms_state=act ∧ act_state=det ∧ det_state=detloop ∧ Counter=0
        THEN    det_state:=fin_det
        END;
                                                                                    ...
```

```
Detection =
            SELECT   fms_state=act ∧ act_state=det ∧ det_state=fin_det
            THEN     act_state:=act1 || act1_state:=anl || anl_state:=anlloop || def_Set_Input_In_ErrorN
            END;
AnalysisLoop =...;
Analysis =...;
Action =...;
Return =...;
Fail =...
END
```

Fig. 20. Excerpt from the refinement FMSR4 in B

As shown in the diagrams in Fig. 17 and Fig. 18, the focus of this refinement step is on the error detection procedure of the FMS. Hence, the B refinement FMSR4 introduces new data structures derived from the introduced class TEST and the association ComplexTest. They allow us to represent the detection architecture formally. Apart from that, the B machine obtains an additional event DetectionLoop, refining the existing detection procedure. It corresponds to the transition between the newly introduced substates detloop and fin_det within the state det from the diagram fms_state. The body of the event DetectionLoop is specified via action def_RunTestOnInput of the corresponding transition in the statechart fms_state. The action is defined within the newly introduced statechart TestOnInput. Its formal counterpart is given in the definitions clause of the B refinement machine.

4.6. Phase 6: Refining the error detection – introducing the time scheduling

The 6th FMS development phase further specifies the mechanism of the error detection. The applicability of the evaluation tests, introduced in the previous development phase, depends on the test frequencies and the internal state of the system. At this development phase, we introduce this information into the error detection procedure. Namely, to enable tests executions according to the given frequencies, we introduce *time scheduling*. We model a global clock, which is used to guarantee that the tests with the same frequencies are executed at the same time instances.

Structure. The FMS at the 6th development phase results in the stereotyped package <<refinement>> FMSR5, as shown in Fig. 21. The main structural change is introduction of two additional classes into the system. The first one – the class STATES – allows us to model the set of internal states of the system. The second one – the class CONDITION – is an association class. It has only one boolean attribute <<constant>> Cond, modelling the enableness of a certain test with respect to the internal system state. If Cond is TRUE then the corresponding test is enabled for execution at the given internal system state. Otherwise, it is disabled.

By introducing the attributes Time and State into the class FMS, we actually implement the concepts of the current time and the current internal system state. Since the enableness of an evaluation test depends not only on the internal system state but also on the given test frequency, we add the attribute <<constant>> Freq to the class TEST. It

models the predefined execution frequency for each test. Furthermore, we explicitly define how and when the time progresses in our system. Hence, we introduce the attribute Clock_Flag into the class FMS, modelling the state of the time scheduler. It can be either enabled or disabled. Initially, we assume it to be disabled.

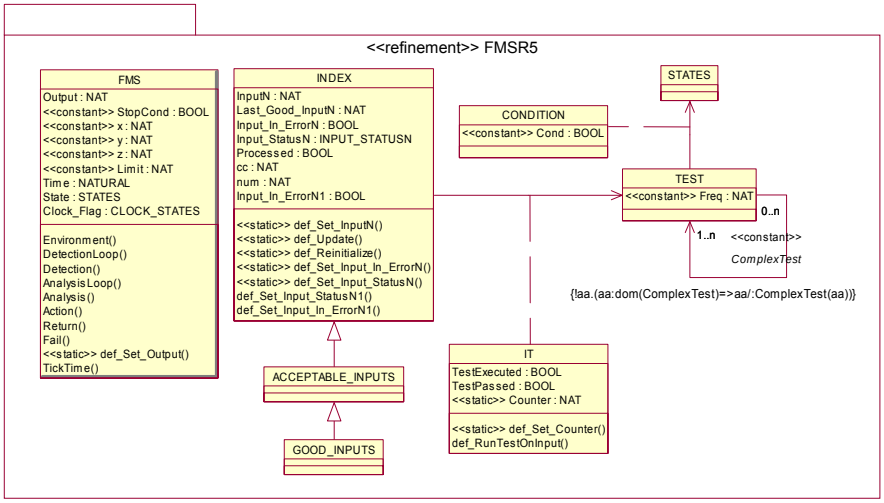


Fig. 21. The class diagram for the 6th FMS development phase

Behaviour. To model time scheduling of tests depending on their frequencies and the internal system state, we need to specify how the time in the system changes and how it affects the evaluation tests. This is defined by the method TickTime in the class FMS. The method is completely specified in μB. It increments the value of the current time, whenever Clock_Flag is enabled and there exist the tests enabled for execution at the current time instance. In addition, it models a possible change of the internal system state by nondeterministically updating the attribute State. When there are no more tests enabled for execution, Clock_Flag is disabled and the FMS cycle proceeds as specified earlier. A new FMS cycle can start only after the previous one finishes, i.e., the time should not progress before the cycle is finished. Hence, we add the guard Clock_Flag=disabled on the transition Environment in the diagram fms_state.

In the previous FMS development phase, the method DetectionLoop was modelling the error detection performed on inputs by applying certain tests and observing their results. The action part of this method – def_RunTestOnInput – was actually specifying the testing mechanism on an instance of the association class IT. At the current phase we refine this action by introducing additional guards to the corresponding transitions in the statechart TestOnInput. These guards specify that:

- the tests are executed with certain given frequencies;
- for some complex test to be executed, its frequency has to be divisible by the frequencies of all the simple tests required for its execution;
- execution of each test depends on the current internal state of the system.

Except the error detection the other FMS actions remain as specified in previous development phases. However, to allow time to progress before another FMS cycle starts, we refine the method def_Reinitialize as follows:

def_Reinitialize == (<unchanged> ǁ Clock_Flag:=enabled)

B machine for the phase 6. Translating the class diagram and statecharts developed in the current development phase results in the B machine shown in Fig. 22.

```
REFINEMENT          FMSR5

REFINES             FMSR4

DEFINITIONS
        type_invariant == ( Time ∈ NATURAL ∧ State ∈ STATES ∧ Clock_Flag ∈ CLOCK_STATES );
        invariant == ( type_invariant );
        def_Reinitialize==...;
        Exist_Test_For_Execution==
        (∃(ii,tt).((ii,tt)∈IT ∧ TestExecuted(ii,tt)=FALSE ∧ (ii∈dom(ComplexTest)⟹
            ∀mm.(mm∈ComplexTest(tt)⟹TestExecuted(ii,mm)=TRUE ∧ (Freq(tt) mod Freq(mm)=0)) ∧
            (Time mod Freq(tt)=0) );
        def_RunTestOnInput ==
        ( ANY it,ii,tt WHERE it∈IT ∧ ii∈INDEX ∧ tt∈TEST ∧ it=(tt↦ii)
          THEN
                SELECT   TestExecuted(it)=FALSE ∧ Input_In_ErrorN1(ii)=FALSE ∧
                         Cond(tt,State)=TRUE ∧ (Time mod Freq(tt)=0) ∧
                         (tt∈dom(ComplexTest)⟹
                                 ∀mm.(mm∈ComplexTest(tt)⟹TestExecuted(ii,mm)=TRUE ∧
                                 (Freq(tt) mod Freq(mm)=0)))
                THEN
                         SELECT TestOnInput(it)=not_executed
                         THEN   TestOnInput(it):=executed_undefined ||
                             ANY xx WHERE xx∈BOOL THEN TestPassed(it):=xx END ||
                             TestExecuted(it):=TRUE
                         WHEN   TestOnInput(it)=executed_undefined ∧ TestPassed(it)=FALSE
                         THEN   TestOnInput(it):=executed_not_passed ||
                                def_Set_Input_In_ErrorN1(ii)
                         WHEN   TestOnInput(it)=executed_undefined ∧ TestPassed(it)=TRUE
                         THEN   TestOnInput(it):=executed_passed
                         END
                END
          END )
SETS
        STATES; CLOCK_STATES={enabled,disabled}
CONSTANTS
        ..., Freq, Cond
PROPERTIES
        ..., Freq ∈ TEST ⟶ NAT ∧ Cond ∈ (TEST×STATES ) ⟶ BOOL
VARIABLES
        ... Time, State, Clock_Flag
INVARIANT
        invariant ∧ ...
INITIALISATION
        ... || Time := 0 || State :∈ STATES || Clock_Flag := disabled
EVENTS
<unchanged>;

TickTime =
        SELECT   Clock_Flag=enabled
        THEN     Time:=Time+1 || State:∈STATES;
                 IF Exist_Test_For_Execution
                 THEN Clock_Flag:=disabled
                 END
        END
END
```

Fig. 22. Excerpt from the refinement FMSR5 in B

At this phase we have introduced an abstract representation of time required to model frequency of test execution and system state needed for modelling test enabledness. This has resulted in formal definition of the architecture of test execution.

4.7. Phase 7: Refining the error detection – introducing types of evaluation tests

The 7th development phase continues to elaborate on the error detection mechanism by modelling different types of evaluation tests. We replace the nondeterministic detection procedure by a deterministic one, which introduces concrete steps of test application.

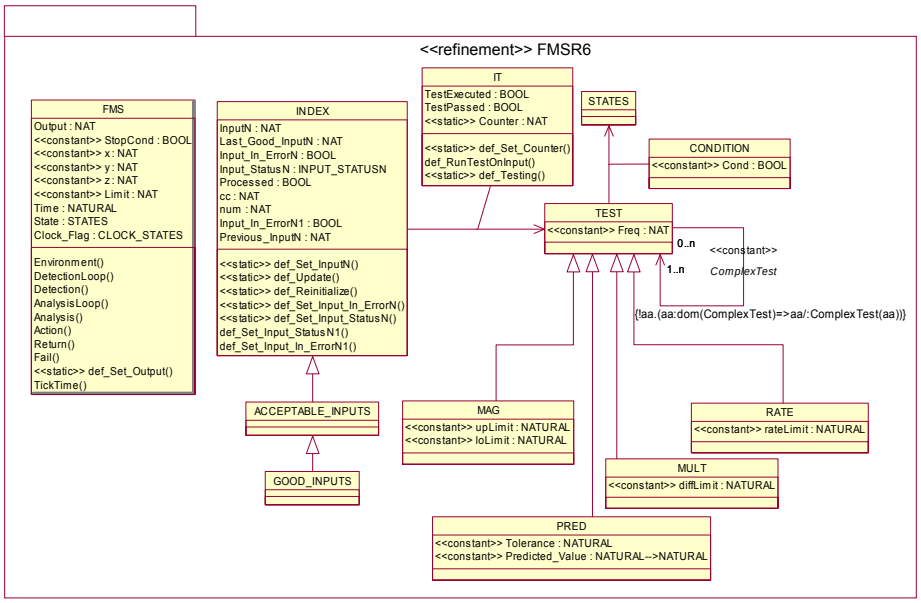


Fig. 23. The class diagram for the 7th FMS development phase

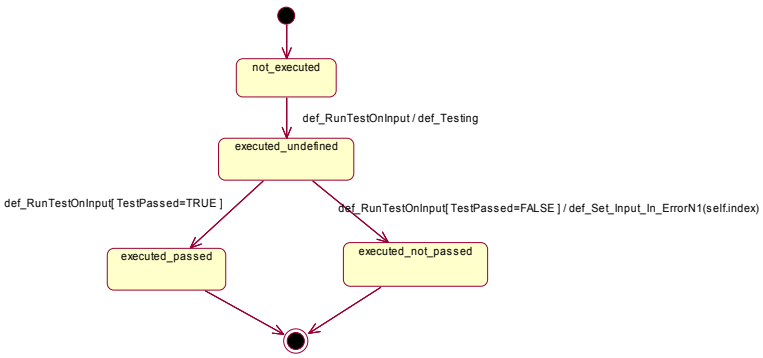


Fig. 24. The refined statechart TestOnInput

Structure. The stereotyped package <<refinement>> FMSR6 represents the FMS at the 7th development phase. To introduce specific types of the evaluation tests, we define the subclasses of the class TEST, as shown in Fig. 23. The subclasses are:

- MAG – the magnitude tests,
- PRED – the predicted value tests,
- RATE – the rate tests,
- MULT – the dual sensor difference tests.

Each of the subclasses also introduces test specific properties, by defining them as subclass attributes. For instance, the subclass MAG has two constant attributes upLimit **(**the upper limit of an input) and loLimit **(**the lower limit of an input) needed for the execution of the magnitude test. Similarly, the subclass PRED contains two constant attributes Predicted_Value (the predicted value of an input at a given time) and Tolerance (the allowed tolerance between the predicted and the current input value) required for execution of predicted value test. The subclass RATE contains only one attribute rateLimit (the allowed difference between the previous and the current input value). However, to apply a rate test, we need to model the previous value of a particular sensor reading. Hence, we introduce it as the attribute Previous_InputN in the class INDEX. Finally, to execute a difference test on dual sensors, we introduce the attribute diffLimit (the allowed difference between the values of two sensors) into the subclass MULT.

Behaviour. We refine the main method DetectionLoop to model the tests executed with given frequencies and their dependency on the current internal state of the system. The guard of the action def_RunTestOnInput within the main method DetectionLoop controls the enableness of tests for execution. However, def_RunTestOnInput does not specify in detail how the actual testing of the input value is performed. Instead, this is modelled as a nondeterministic assignment to the variable TestPassed. At this development phase, we refine this nondeterminism by introducing the method def_Testing in the association class IT. It is defined as an action triggered by the transition def_RunTestOnInput in the statechart TestOnInput, as shown in Fig. 24. Here def_Testing is specified as follows:

```
def_Testing ==
( SELECT tt∈MAG
  THEN   IF InputN(ii) < upLimit(tt) ∧ InputN(ii) > loLimit(tt)
         THEN     TestPassed(ii,tt):=TRUE
         ELSE
                  skip
         END
  WHEN   tt∈RATE
  THEN   IF (InputN(ii)-Previous_InputN(ii)) > 0
         THEN     IF       (InputN(ii)-Previous_InputN(ii)) < rateLimit(tt)
                  THEN     TestPassed(ii,tt):=TRUE ||
                           Previous_InputN(ii):=InputN(ii)
                  ELSE
                           skip
                  END
         ELSE     IF       (Previous_InputN(ii)-InputN(ii)) < rateLimit(tt)
                  THEN     TestPassed(ii,tt):=TRUE ||
                           Previous_InputN(ii):=InputN(ii)
                  ELSE
                           skip
                  END
         END
                                                                          ...
```

```
WHEN  tt∈MULT
THEN  ANY ii2 WHERE ii2∈Indx ∧ ii2≠ii
      THEN
            IF (InputN(ii) - InputN(ii2)) > 0
            THEN    IF  (InputN(ii) -InputN(ii2)) < diffLimit(tt)
                    THEN    TestPassed(ii,tt):=TRUE
                    ELSE
                            skip
                    END
            ELSE    IF  (InputN(ii2) - InputN(ii)) < diffLimit(tt)
                    THEN    TestPassed(ii,tt):=TRUE
                    ELSE
                            skip
                    END
            END
      END
WHEN  tt∈PRED
THEN  IF      ii < Predicted_Value(tt)[Time]+Tolerance(tt) ∧
              ii > Predicted_Value(tt)[Time]-Tolerance(tt)
      THEN    TestPassed(ii,tt):=TRUE
      ELSE
              skip
      END
END )
```

The newly introduced attribute Previous_InputN is updated to ensure that only inputs which are ok or suspected are considered in the following FMS cycles. The refined method def_Update now corresponds to the following μB:

def_Update == (<unchanged> ||
Previous_InputN := ACCEPTABLE_INPUTS ◁ Previous_InputN)

B machine for the phase 7. Elaborating on the error detection procedure in the UML-B diagrams in Fig. 23 and Fig. 24 and their translation into B results in the refined B machine, as shown in Fig. 25.

```
REFINEMENT      FMSR6

REFINES         FMSR5

DEFINITIONS
        type_invariant == ( ... );
        invariant == ( type_invariant );
        def_Update==...;
        def_Testing==...;
        def_RunTestOnInput ==
        ( ANY it,ii,tt WHERE it∈IT ∧ ii∈INDEX ∧ tt∈TEST ∧ it=(tt↦ii)
          THEN
                SELECT  < unchanged >
                THEN
                        SELECT TestOnInput(it)=not_executed
                        THEN  TestOnInput(it):=executed_undefined || def_Testing
                        WHEN  < unchanged >
                        THEN ...
                END
        END )
                                                                          ...
```

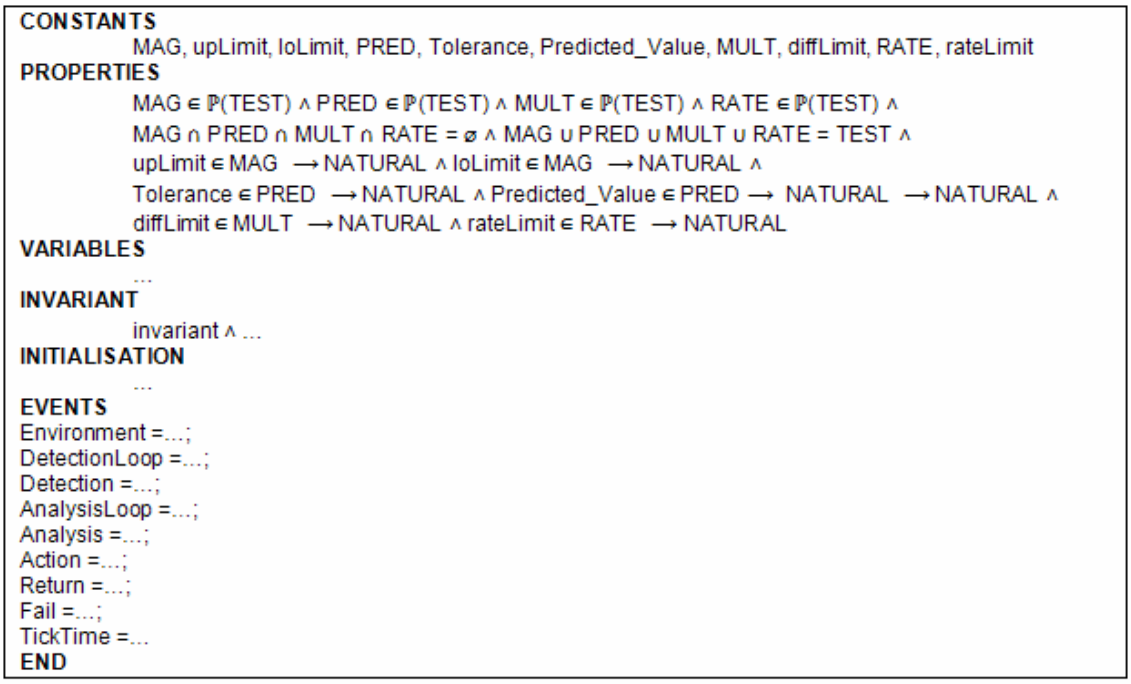


Fig. 25. Excerpt from the refinement FMSR6 in B

At the seventh phase we model how to differentiate between types of tests and hence further refine the error detection procedure. This allowed us to define the test architecture more accurately, depending on the type of tests.

Summary. Development of a fault tolerant control system tolerating transient faults – the FMS – is achieved through the number of development phases. At each development phase we represent the system structure (via UML-B class diagrams), and its behaviour (via UML-B statecharts).

The development starts from an abstract FMS description, modelling the basic system functionality. This 1st development phase outlines the stages of the FMS operating cycle, starting with obtaining the sensor readings, processing them, and either failing or calculating the output of the FMS. In the latter case the FMS operating cycle starts again. In addition, the system safety properties are specified as safety invariants. They are preserved in the 2nd development phase, which introduces processing of inputs performed after obtaining the sensor readings. Namely, it introduces the error detection procedure within the FMS. The error detection classifies the inputs as faulty or fault-free, continuing the operating cycle as previously specified. In the 3rd FMS development phase, we abstractly introduce the input analysis performed by the FMS after the error detection. The result of the input analysis is the input statuses. They determine possible FMS recovery actions. At this phase, we also introduce a certain predefined stopping (freezing) condition, and express additional system safety properties. The 4th FMS development phase refines the input analysis. We define the input analysis as performed independently on each monitored sensor. In addition, we specify in detail the procedure of determining the input status based on using a specific counting mechanism. The 5th development phase refines already specified error detection mechanism by introducing

the evaluation tests. They are applied on the obtained inputs, as defined by the given test architecture. The 6th FMS development phase further specifies the mechanism of the error detection. Namely, it introduces time scheduling to enable test execution according to the given frequencies and the internal state of the system. Finally, the 7th development phase focuses on the details of the error detection mechanism by modeling different types of evaluation tests, while still preserving specified safety properties.

The overall development results in UML-B diagrams representing general models, i.e., development templates, for developing similar systems. These templates can be instantiated for particular systems by populating the abstract data in templates by concrete data. For instance, we can consider different number of sensors, define concrete stopping conditions and internal system states, replace the abstract system configuration parameters (e.g., x, y, z etc.) with concrete values and so on.

5. Conclusions

In this paper we proposed an approach that integrates the formal refinement-based development of the *Failure Management System* (FMS) [4] with the UML-based development. The FMS is the typical subcomponent of controllers in avionics applications, which purpose is to prevent system failures due to erroneous inputs from its environment. Within the FMS, we specifically focus on designing the mechanisms for tolerating transient faults, as the most typical faults in control systems. In general, developing the FMS is costly and time consuming. Apart from that, dependability of such systems is crucial and should be appropriately ensured. Our UML-based development of the FMS aims at creating a set of development templates, which could be reused by means of instantiation for developing a family of similar components. Thus, they potentially increase development cost-efficiency and time-effectiveness. Moreover, the approach allows us to automatically obtain formal models from the corresponding UML models and ensure their correctness, which essentially contributes to the overall system dependability.

The development of the FMS is undertaken using the formal modelling language UML-B. It combines the B Method and UML, allowing us to use the familiar UML notation, yet more precisely. The development adopts the refinement-based, top-down approach. Hence, we initially specify the system abstractly, by modelling the FMS structure and behaviour describing only basic stages of its operating cycle. In general, system structure is described using the UML-B class diagram, whereas its behaviour is represented in the corresponding UML-B statechart. The further development phases introduce more details into existing UML-B models in a structured manner.

Using the graphical UML-like modelling language allowed us to obtain better structured models of the FMS. The separate representation of the system structure and behaviour within different types of diagrams enabled considering only part of the system requirements at a time. Moreover, representing the system behaviour in the form of statecharts improved our reasoning about the system and allowed us to specify some interesting system properties at particular system states.

After completing each development phase, we used the automatic translator tool, U2B, to obtain corresponding B models. This, in turn, allowed us to use the available

tool support, AtelierB, to automate verification and prove correctness of our development templates. We proved app. 65% of the generated proof obligations automatically, while the remaining proof obligations (most of which have been of the same form) required slight user interaction. They have been proved using the interactive prover within AtelierB.

Our approach to developing the FMS, is similar to the work of Laibinis and Troubitsyna [20]. They propose an approach to formal model-driven development of a fault tolerant controller in B. However, it mainly implements the fault tolerance based on triple modular hardware redundancy. Our approach is based on computational redundancy for dealing specifically with transient faults, which were not considered in their development.

The lack of methodology for the specification of system fault tolerance motivated the work of Dondossola and Botti [21]. They proposed an approach to systematization of fault tolerance concepts with the aim to support guided analysis of fault tolerance requirements, leading to specification of fault tolerant solutions in various safety related applications. The approach to fault tolerance specification is based on using UML and TRIO (Tempo Reale ImplicitO) temporal logic. However, although our approach is less general, since we focus on specific control systems and tolerating specific types of faults, we specify fault tolerance more explicitly, and allow considering requirements related to fault tolerance in a stepwise manner, which results in better structured system specification.

Liu and Joseph [22] consider the specification and verification of safety-critical systems by use of formal techniques. They show how to specify and verify fault tolerance within a single formal framework. For that purpose, they use transition systems as the computational model, and specify fault tolerant properties using the Temporal Logic of Actions (TLA) notation. They model physical faults in a system in general. Fault tolerance is achieved by adding the appropriate recovery actions, which make the program tolerant to these faults. In our approach, we do not use explicit representation of faults. On the contrary, we propose a specific mechanism to detect these faults, in particular transient faults, and then implicitly decide on the recovery actions. However, the approach of Liu and Joseph uses more advanced timing policy, which is in our approach simplified to cope only with the required frequencies of detection test execution.

There are many other approaches, e.g., [23, 24, 25] that tackle a similar problem – design of software-implemented fault tolerance. This work is complementary to ours: while it focuses on studying how to modify software at the code level to achieve fault tolerance, we aim at studying how to specify and develop software with the fault tolerance mechanisms integrated into it.

The research on building fault tolerant components is especially addressed in the work of Anderson et al. [26, 27] and Popov et al. [28]. They propose an approach to developing protective wrappers as means to improve the dependability of the systems, which are built upon reusable components. Similarly to our approach, they advocate that the development of a protective wrapper should be a systematic, stepwise process. However, they consider reusable off-the-shelf components usually perceived as black boxes, and specify mechanisms able to withstand their erroneous behaviour. This is the main difference between our approaches – we explicitly specify the fault tolerance

mechanisms as an essential part of the system. In this respect, we undertake a formal development of our fault tolerant control system and give a proof of its correctness by means of formal verification.

Formal UML-based development of the FMS has been also undertaken by Snook et al. [10, 29]. Similarly to our approach, they focus on reusability of the FMS modelled using UML-B. However, their approach focuses on the structure of the FMS, and modelling and instantiating complex FMS requirements. Our approach focuses on behavioural specification in the first place, while correspondingly specifying the FMS structure as well. In addition, our error detection procedure differs from the one they propose. They do not explicitly address the dependencies between evaluation tests used in this procedure, whereas we define a hierarchical test architecture allowing us to tackle the input deviations more efficiently. However, we believe that more solid foundations for developing fault tolerant controllers of the similar type could be achieved by merging these two approaches. It is our plan to investigate this integration of results in our future studies.

Acknowledgments

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

References

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

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

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

[4] Selic, B., "The Pragmatics of Model-Driven Development", *IEEE Software*, 20(5), 2003, pp. 19-25

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

[6] Snook, C. and Butler, M., "UML-B: Formal modelling and design aided by UML", *ACM Transactions on Software Engineering and Methodology*, 15(1), 2006, pp. 92-122

[7] Rumbaugh, J., Jacobson, I., and Booch, G., *Unified Modeling Language Reference Manual*, Addison Wesley, 1999

[8] Snook, C. and Butler, M., "U2B - A tool for translating UML-B models into B", In Mermet, J., Eds. *UML-B Specification for Proven Embedded Systems Design*, Chapter 6, Springer, 2004

[9] Clearsy System Engineering, *AtelierB: User Manual*, Version 3.7. Available at: http://www.atelierb.societe.com/ressources/DOC/english/user_manual.pdf

[10] Johnson, I., Snook, C., Edmunds, A., and Butler, M., "Rigorous development of reusable, domain-specific components, for complex applications", In *Proc. of 3rd International Workshop on Critical Systems Development with UML*, Lisbon, 2004, pp. 115-129
[11] Johnson, I., Snook, C., Rodin Project Case Study 2: Requirements Specification Document, *RODIN Deliverable D4 - Traceable Requirements Document for Case Studies*, Section 3, 2005, pp. 24-52
[12] Schneider, S., *The B Method. An introduction*, Palgrave, 2001
[13] Abrial, J.-R., "Event Driven Sequential Program Construction", 2001. Available at: http://www.atelierb.societe.com/ressources/articles/seq.pdf
[14] Abrial, J.-R. and Hallerstede, S., "Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B", *Fundamentae Informatica*, 77(1-2), 2007, pp. 1-28
[15] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall International, 1976
[16] Back, R.J., and von Wright, J., *Refinement Calculus: A Systematic Introduction*, Springer-Verlag, 1998
[17] OMG, *UML 2.0 Superstructure Specification*, 2005. Available at: http://www.omg.org/docs/formal/05-07-04.pdf
[18] IBM, Rational Rose, http://www-306.ibm.com/software/rational/
[19] Snook, C., and Walden, M., "Refinement of Statemachines Using Event B Semantics", In *B 2007: Formal Specification and Development in B*, *LNCS 4355*, Springer-Verlag, 2006, pp. 171-185
[20] Laibinis, L., and Troubitsyna, E., "Refinement of fault tolerant control systems in B", In *ComputerSafety, Reliability, and Security - Proceedings of SAFECOMP 2004* Lecture Notes in Computer Science, Vol. 219, Springer-Verlag, September 2004, pp. 254-268
[21] Dondossola, G., and Botti, O., "System Fault Tolerance Specification: Proposal of a Method Combining Semi-formal and Formal Approaches", In *Proceedings of the 3rd International Conference on Fundamental Approaches to Software Engineering (FASE'00)*, *LNCS 1783*, Springer-Verlag, 2000, pp. 82-96
[22] Liu, Z., and Joseph, M., "Specification and Verification of Fault-Tolerance, Timing, and Scheduling", *ACM Transactions on Programming Languages and Systems*, 21(1), 1999, pp. 46-89
[23] Rebaudengo, M., Reorda, M.S., Torchiano, M., and Violante, M., "A Source-to-Source Compiler for Generating Dependable Software", *IEEE International Workshop on Source Code Analysis and Manipulation*, 2001, pp. 33-42
[24] Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., and August, D.I., "SWIFT: Software Implemented Fault Tolerance", *Proceedings of the Third International Symposium on Code Generation and Optimization*, March 2005, pp. 243-254
[25] Oh, N., Mitra, S., and McCluskey, E.J., "ED4I: Error Detection by Diverse Data and Duplicated Instructions", *IEEE Transactions on Computers*, 51(2), 2002, pp. 180-199
[26] Anderson, T., Feng, M., Riddle, S., and Romanovsky, A., "Protective Wrapper Development: A Case Study", In *Proceedings of the 2nd International Conference on COTS-Based Software Systems*, *LNCS 2580*, Springer-Verlag, 2003, pp. 1-14

[27] Anderson, T., Feng, M., Riddle, S., and Romanovsky, A., "Error Recovery for a Boiler System with OTS PID Controller", In *Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, IEEE Computer Society, 2005, pp. 113-120

[28] Popov, P., Riddle, S., Romanovsky, A., and Strigini, L., "On systematic design of protectors for employing OTS items", In *Proceedings of the 27th EUROMICRO Conference*, IEEE Computer Society, 2001, pp. 22-29

[29] Snook, C., Poppleton, M., and Johnson, I., "The engineering of generic requirements for failure management", In *Proceedings of 11th International Workshop on Requirements Engineering: Foundation for Software Quality*, Oporto, 2005, pp. 145-160

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

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

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-1918-4
ISSN 1239-1891