Formalizing UML-based Development of Fault Tolerant Control Systems*

Dubravka Ilić[1], Elena Troubitsyna[1], Linas Laibinis[1], and Colin Snook[2]

[1] Åbo Akademi University, Department of Information Technologies,
20520 Turku, Finland
{Dubravka.Ilic, Elena.Troubitsyna, Linas.Laibinis}@abo.fi
[2] School of Electronics and Computer Science,
University of Southampton, SO17 1BJ, UK
cfs@ecs.soton.ac.uk

Abstract. In this paper we demonstrate how to formalize UML-based development of protective wrappers for tolerating transient faults. In particular, we focus on the fault tolerance mechanisms common in the avionics domain and show the development of a protective wrapper, called Failure Management System. We demonstrate how to integrate the formal refinement approach proposed earlier into the UML-based development.

Keywords: Event-B, fault tolerance, refinement, statemachines, transient faults, UML-B.

1 Introduction

To guarantee dependability [1] of safety-critical software-intensive systems, we should ensure that they are not only fault-free but also tolerant to faults [2] of system components. This paper focuses on designing controlling software for tolerating transient faults [3]. Transient faults are temporal defects within the system. The mechanisms for tolerating this type of faults should ensure that the controlling software does not overreact on isolated faults yet does not allow the errors caused by these faults to propagate further into the system. These mechanisms constitute a large part of software in complex systems and hence they could be perceived as a separate subsystem dedicated to fault tolerance. In avionics, such a subsystem is traditionally called *Failure Management System* (FMS).

Earlier we proposed a generic formal pattern for specifying and developing the FMS [4] in the B Method [5, 6]. However, industrial engineers often perceive constructing a formal specification from informal requirements to be too complex to be done without an intermediate modeling stage. They usually use graphical modeling, mostly in UML [7], to facilitate this process. In this paper we

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

demonstrate how to integrate the formal approach proposed previously into the UML-based development. We use a subset of UML called UML-B [8] to specify and develop the FMS. To automate the process of obtaining a formal specification from UML models, we use the U2B tool [9], which translates UML-B models into Event-B [10]. Event-B is an extension of the B Method for developing reactive and distributed systems. We use the automated tool support for Event-B to verify the correctness of our development. Therefore, the proposed approach has a high degree of automation.

The paper is structured as follows. In Section 2 we shortly describe the FMS. Section 3 gives a brief introduction into our modeling frameworks – Event-B and UML-B. Section 4 demonstrates the process of developing 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. Moreover, we show how to translate these models into Event-B and verify their correctness. Finally, Section 5 concludes the paper.

2 Failure Management System

The Failure Management System (FMS) [11, 12] is a part of the embedded safety-critical control system as shown in Fig. 1. It can be perceived as a protective "wrapper" with the task to detect erroneous inputs from the sensors and prevent their propagation into the controller.

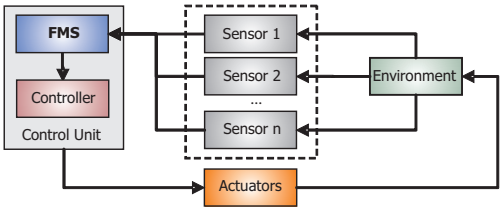

Fig. 1. Structure of an embedded control system

Based on sensor readings, the FMS calculates the output and forwards it to the controller. While calculating the output, the FMS has to ensure that only fault-free inputs received from the system environment are passed to the controller. This is achieved by considering the following pattern of the FMS behavior. We assume that initially the system is fault-free. Since control systems are usually cyclic, it is natural to describe the behavior of the FMS as cyclic as well. The FMS operating cycle starts by obtaining the readings from the monitored sensors as the inputs to the FMS. The FMS then tests the inputs by applying a certain error detection procedure. As a result, depending on whether the error was detected or not, the inputs are categorized as fault-free or faulty. Then the FMS takes the corresponding remedial actions that can be classified as healthy, temporary or confirmation [12]. An important part of these actions is input analysis, which distinguishes between recoverable and non-recoverable faulty inputs by assigning them different statuses.

To explain how the remedial actions work, for simplicity we consider a single sensor. *Healthy action* describes the "normal" FMS reaction when a received input is fault-free. In such a case, the input is assigned the status *ok* and it is forwarded unchanged to the controller. *Temporary action* describes the FMS reaction when a received input is faulty and recovering, meaning that the number of previously received faulty inputs has not yet reached some predefined limit. If this is the case, the input is assigned the status *suspected*. Then, the FMS calculates the output using the last good value of this input obtained in the previous FMS cycles. Finally, *confirmation action* describes the FMS reaction when a received input is faulty and it has failed to recover. Then, the input is assigned the status *confirmed failed* and the system proceeds with the control actions defined for freezing (stopping) the system or switching to a backup controller, if possible.

The pattern of the FMS behavior described above can be used in the product line development of the controlling software [13]. We use this pattern for developing the aircraft engine FMS in UML-B (and indirectly in Event-B). In the next section we introduce these modeling frameworks.

3 Modeling Frameworks – Event-B and UML-B

Event-B. The Event-B Method [10] is an approach for modeling dependable systems, which extends the B Method [5, 6]. In Event-B, a model of a system is described by *contexts* and *machines*. Contexts describe the static part of the system using *carrier sets*, *constants* and *axioms*. Machines describe system dynamics using *variables*, *invariants*, *theorems*, *events* and *variants*. Variables of the machine define the machine state. They are strongly typed by invariants and can be altered by events. Events are given in the form event=**WHEN** *guard* **THEN** *action* **END**, where *guard* is a state predicate on the variables, and *action* is a set of assignments, which simultaneously update the machine variables. If *guard* is satisfied, the event is enabled and the behavior of the event corresponds to the execution of its *action*. If *guard* is false, then the event is disabled, i.e., its execution is blocked.

The development methodology adopted by Event-B is based on stepwise refinement [14]. The result of a refinement step in Event-B is the machine that refines the state and events of an abstract machine. The invariant of this machine additionally contains the gluing invariant that describes the connection between the state spaces of the more abstract and refined machines.

To ensure correctness of a specification, we should verify that each event of the machine, including the initialisation, preserves the invariant. A high degree of automation in verifying correctness is provided by the available Event-B tool support [15].

UML-B. UML-B [8] is a specialisation of UML [7], which combines UML and Event-B to define a graphical formal modeling notation. UML is widely used graphical modeling language. However, it lacks precise semantics. Event-B, on

the other hand, is a formal modeling framework, but it requires significant mathematical training from the users. The UML-B is developed as an alliance of these two modeling approaches. It contains a limited subset of UML entities which semantics is provided by their translation into Event-B using the U2B [9] translator tool. U2B converts a UML-B model into its equivalent Event-B model. We can then verify the model correctness by using the Event-B tool support.

In UML-B, a model of a system is described by *package*, *context*, *class*, and *state-machine diagrams*. A package diagram describes the abstract view on the system architecture. In other words, it describes the packages encapsulating the system on different levels of abstraction and the dependencies between them. In addition, it allows separating specification of the static and the dynamic parts of the system. This is achieved by defining two types of packages: *Context* and *Machine* package, which coincide with the concepts defined in Event-B. Each context has the associated context diagram defining the constants and properties of these constants (axioms). Each machine has the associated class diagram capturing the functional requirements of the modeled system. The classes of the class diagram define system components whose properties are specified as class attributes. The behavior of each component is defined by a statemachine diagram. Hence, on the abstract level the system is described by a set of class diagrams and statemachines encapsulated within the abstract machine package.

UML-B adopts the same approach to system development as Event-B, i.e., stepwise refinement. In particular, it uses superposition refinement [14], which allows us to extend the state space while preserving the existing data structures unchanged. The first step of refining a UML-B model is 'cloning' the current model in order to preserve the old class diagrams and statemachines. Then, we introduce new UML-B elements gradually by incorporating more details about the system structure and behavior. Specifically, more detailed behavior of the system is modeled with hierarchical states by adding sub-states and new transitions to the existing statemachines. Refinement of UML-B statemachines is described in detail in [16].

In general, while developing the system in a number of refinement steps, we create a chain of machine packages, where each subsequent package is a refinement of the previous package, i.e., of its class diagrams and statemachines. The refinement relation is established by adding the association *Refines* between the corresponding packages.

A more detailed description of UML-B entities is given in the following section, where we demonstrate how to specify and refine the FMS in UML-B. We also show how to obtain the Event-B models of FMS from their UML-B counterparts and verify their correctness.

4 Developing the FMS with specification and refinement templates in UML-B

The development of the FMS in UML-B is done in several phases. Each development phase corresponds to a refinement step. It is characterized by a set of

UML-B models (class and statemachine diagrams) representing the main structural and behavioral aspects of the FMS at the corresponding level of abstraction.

FMS abstract specification. At the highest level of abstraction, we consider a very simple FMS as shown in Fig. 2. In the class diagram FMS0, the fixed class SENSORS describes the set of n analogue sensors that are monitored by the FMS. Signals from each sensor are modeled as the class attribute Value. The output of the FMS is modeled as the machine variable Output. At this development phase, the FMS nondeterministically calculates the output using the last good sensor readings. Hence, we introduce an additional attribute to the class SENSORS – Last_Good_Value. Moreover, the subclass FAILED_SENSORS is introduced to model the sensors that have failed.

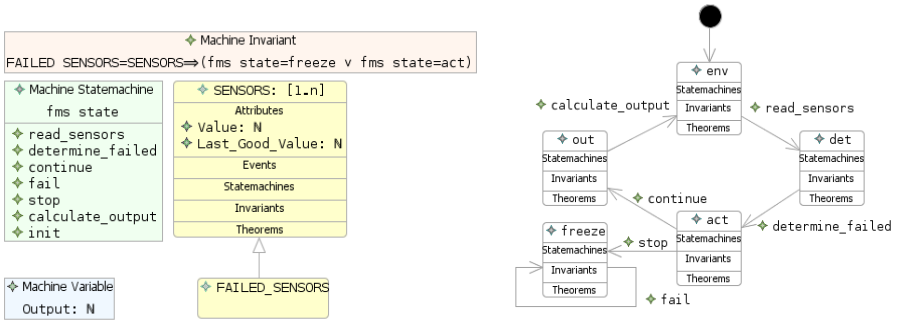

Fig. 2. (a) class diagram FMS0 and (b) statemachine fms_state for the 1st FMS development phase

The way in which the FMS behaves is described via the statemachine fms_state. The states env, det, act, out and freeze in this statemachine denote different stages of the FMS cycle. At this phase, we model the FMS cycle very abstractly: the FMS reads input values from the sensors, then it performs error detection, and either continues the cycle by calculating the output or fails. If the output is successfully calculated, the FMS cycle starts again. The FMS state changes are described by transitions between the states in the statemachine fms_state. For instance, the transition determine_failed simulates the error detection by nondeterministically choosing failed sensors, i.e., FAILED_SENSORS:$\in \{x \mid x \in \mathbb{P}(\mathsf{SENSORS})\}$. At the later development stages this transition will be refined to implement a more detailed error detection procedure.

To ensure that the FMS can proceed operating only with the sensors that have not failed, we define state invariants in the statemachine fms_state. Formally, the invariant $(\exists s \cdot s \in \mathsf{SENSORS} \wedge s \notin \mathsf{FAILED\_SENSORS})$ is associated with the states env, det, and out. It means that, when the FMS is in these states, it processes readings from at least one operational (non-failed) sensor. The machine invariant, which is a part of the class diagram, additionally states the properties of the FMS when all the sensors have failed.

FMS refinement. The abstract FMS model is actually encapsulated in the machine package FMS0[1], as shown in Fig. 3. We further continue the FMS development by creating the refinement package FMSR1, which introduces changes into the abstract FMS model. At this development phase we refine the error detection from the abstract model by introducing sensor testing.

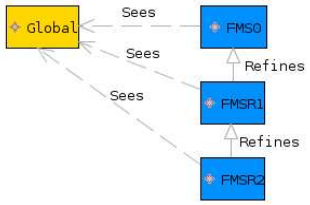

Fig. 3. FMS package diagram

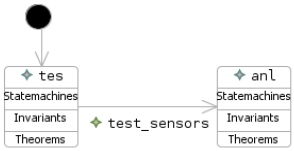

Fig. 4. sub-statemachine det_state

The diagrams from the previous phase remain the same. However, to introduce sensor testing, we refine the statemachine fms_state by creating the sub-statemachine det_state inside the state det, as shown in Fig. 4. The newly introduced sub-statemachine defines two new states tes and anl, designating the steps of the FMS error detection. Namely, after obtaining the sensor readings, the FMS performs testing the sensors (tes) and then analysis of inputs (anl) in order to detect errors, and then continues by determining which sensors have failed. The actual testing procedure is modeled as the transition test_sensors in the sub-statemachine det_state. It nondeterministically decides on the result of error detection. This result is modeled as a value assigned to a newly introduced attribute of the existing class SENSORS – Error_Detected. The FMS now uses this information to decide which sensors have failed. Hence, the transition determine_failed from the statemachine fms_state is refined as follows:

$$\begin{aligned} \mathsf{FAILED\_SENSORS} :\in \{ x \mid x \in \mathbb{P}(\mathsf{SENSORS}) \wedge \\ (\forall s \cdot s \in x \Rightarrow \mathsf{Error\_Detected}(s) = \mathsf{TRUE}) \} \end{aligned}$$

In addition, a new machine invariant is added to the existing class diagram. It describes in detail the properties of sensors: i.e., it requires that all failed sensors should be detected. The invariant is formally expressed as follows:

$$\mathsf{fms\_state} = \mathsf{act} \Rightarrow (\forall s \cdot s \in \mathsf{FAILED\_SENSORS} \Rightarrow \mathsf{Error\_Detected}(s) = \mathsf{TRUE})$$

The way in which sensors are analyzed after testing is further refined in the 3rd development phase by creating the refinement package FMSR2. It contains all the class and statemachine diagrams from the previous phase. In addition, it contains a new sub-statemachine inside the state anl from the sub-statemachine det_state. This sub-statemachine defines more precisely the FMS behavior after performing tests on the sensors. Namely, the FMS decides about the status of each particular input before taking the corresponding remedial actions.

[1] FMS0 gets the access to the context Global by the association type Sees.

The structure of the FMS is refined as well, by introducing a new attribute Sensor_Status for modeling the result of this decision. The machine invariants can now be further strengthened to describe situations in which faulty sensors can recover. If they can not recover, the invariant guarantees that they will be considered as failed.

The following FMS development phases continue to refine the structure and the behavior of the original system. Due to the lack of space, we only outline these further development phases and omit their detailed description: the 4^{th} development phase introduces detailed analysis of inputs based on the results of error detection. The input analysis is further elaborated in the 5^{th} development phase by specifying a customizable counting mechanism, which reevaluates the status of the analyzed inputs at each FMS cycle. In a similar way, the 6^{th} development phase describes in detail the error detecting procedure performed on each sensor and continues by introducing error detection tests in the 7^{th} development phase. The 8^{th} development phase further elaborates on different types of these tests.

Creating FMS Event-B models from UML-B models. Using the U2B tool, the Event-B models are automatically generated from the above UML-B models. For instance, the machine package FMS0 containing the diagrams given in Fig. 2 corresponds to the FMS0 Event-B machine shown in Fig. 5.

```
MACHINE   FMS0
SEES Global, FMS0_implicitContext
VARIABLES
     fms_state, FAILED_SENSORS, Value, Output, Last_Good_Value
INVARIANTS
     fms_state∈fms_state_STATES ∧ FAILED_SENSORS∈ℙ(SENSORS) ∧
     Value∈SENSORS⟶ℕ ∧ …
EVENTS
INITIALISATION
     BEGIN fms_state≔env ∥ FAILED_SENSORS≔∅ ∥ Value≔SENSORS×{InitInput} ∥ … END
read_sensors ==
     WHEN fms_state=env THEN fms_state≔det ∥ Value:∈SENSORS⟶ℕ END
determine_failed ==
     WHEN fms_state=det THEN fms_state≔act ∥
     FAILED_SENSORS:∈{xx|xx∈ℙ(SENSORS)} END
continue ==
     ANY yy WHERE yy∈ℙ(Value) ∧ fms_state=act ∧ FAILED_SENSORS≠SENSORS
     THEN fms_state≔out ∥ Last_Good_Value≔Last_Good_Value⩤yy END
calculate_output ==
     ANY xx WHERE xx∈ℙ(Last_Good_Value) ∧ fms_state=out ∧ …
     THEN fms_state≔env ∥ Output:∈ran(xx) END
stop ==
     WHEN fms_state=act ∧ FAILED_SENSORS=SENSORS THEN fms_state≔freeze END
fail ==
     WHEN fms_state=freeze THEN skip END
END
```

Fig. 5. Excerpt from the Event-B abstract specification FMS0

Informally, the rules for mapping some frequently used UML-B concepts into Event-B can be summarized as follows:

– a fixed class is defined as a constant, e.g., the class SENSORS corresponds to a constant defined in the automatically generated context FMS0_implicitContext;
– a subclass is represented as a variable, which is typed as a subset of its superclass, e.g., the subclass FAILED_SENSORS is defined as a subset of SENSORS;
– the name of a statemachine corresponds to a variable, which type is defined by enumerating its states, e.g., the statemachine fms_state is defined as the variable fms_state of the type fms_state_STATES, where the type is the enumerated set {env,det,act,out,freeze} defined in FMS0_implicitContext;
– an attribute of a fixed class becomes a machine variable typed as a function from the constant designating that class to the given attribute type, e.g., Value ∈ SENSORS → ℕ is an array of input readings for n sensors;
– a machine variable and an invariant are equivalent to the same concepts in Event-B;
– the transitions from a statemachine correspond to the events defined using the transition properties stated in UML-B. The complete list of translation rules can be found elsewhere (e.g., [8, 9]).

Similarly to the translation of the package FMS0 into the Event-B machine FMS0, the package FMSR1 refining the abstract package FMS0 is translated into the corresponding refined Event-B machine as shown in Fig. 6.

```
MACHINE   FMSR1
REFINES   FMS0
SEES Global, FMSR1_implicitContext
VARIABLES
     …, det_state, Error_Detected
INVARIANTS
     … ∧ det_state∈det_state_STATES ∧ Error_Detected∈SENSORS⟶BOOL ∧
     fms_state=act⇒(∀s·s∈FAILED_SENSORS⇒Error_Detected(s)=TRUE)
EVENTS
INITIALISATION
     BEGIN … det_state≔tes ‖ Error_Detected≔SENSORS×{FALSE} END
read_sensors == …
test_sensors ==
     WHEN fms_state=det ∧ det_state=tes
     THEN det_state≔anl ‖ Error_Detected:∈SENSORS⟶BOOL END
determine_failed (refines determine_failed) ==
     WHEN fms_state=det ∧ det_state=anl
     THEN fms_state≔act ‖ det_state≔tes ‖
     FAILED_SENSORS:∈{x|x∈ℙ(SENSORS)∧(∀s·s∈x⇒Error_Detected(s)=TRUE)}
     END
continue == …
calculate_output == …
stop == …
fail == …
END
```

Fig. 6. Excerpt from the Event-B refinement FMSR1

Observe that the machine FMSR1 explicitly states which machine it refines. It obtains two additional variables: one for the newly introduced class attribute Error_Detected and another one modeling the current state in the sub-statemachine

det_state. The invariant of the machine FSMR1 is strengthened by typing the newly introduced variables and adding the gluing invariant that connects the new variable Error_Detected with the existing variable FAILED_SENSORS. Furthermore, FMSR1 introduces the new event test_sensors corresponding to the new transition in the sub-statemachine det_state from Fig. 4. It describes how the new variable Error_Detected is changed during the FMS error detection procedure. Moreover, the event determine_failed from the machine FMS0 is refined in the machine FMSR1. The guard of this event is strengthened by adding the new predicate that specifies the event enabling state of the sub-statemachine det_state. Correspondingly, the actions of the event are refined as well in order to incorporate the knowledge of the newly introduced variables.

Formal verification of the obtained Event-B machines is done using the automatic tool support for Event-B [15].

5 Conclusion

In this paper we demonstrated how to integrate the classical refinement development of the FMS [4] with UML-based development. Moreover, we showed how to use the available tool support to automate modeling and verification. Our approach has been validated by a case study – modeling and verification of the engine FMS for tolerating transient faults. The approach has several phases. Each phase is characterized by the set of UML-B models (class diagrams and statemachines). The complete specification of the FMS is obtained through a series of gradually refined UML-B models. Using the U2B translator tool, we generated Event-B models from the overall set of previously developed UML-B models. By translating UML-B models into Event-B, we were able to use the Event-B proof tool support to verify the correctness of our development. The results showed that we were able to prove the correctness of models significantly faster, with a higher percentage of automatic proofs than in our previous B development [4].

In the future, we are planning to investigate instantiation of the developed templates by using the obtained FMS UML-B contexts.

References

1. J. C. Laprie. *Dependability: Basic Concepts and Terminology.* Springer-Verlag, 1991.
2. A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1), 2004.
3. N. Storey. *Safety-critical computer systems.* Addison-Wesley, 1996.
4. D. Ilic, E. Troubitsyna, L. Laibinis, and C. Snook. Formal Development of Mechanisms for Tolerating Transient Faults. In *REFT 2005, LNCS 4157*, pages 189–209. Springer-Verlag, November 2006.
5. J.-R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

6. S. Schneider. *The B Method. An Introduction*. Palgrave, 2001.
7. J. Rumbaugh, I. Jakobson, and G. Booch. *The Unified Modelling Language Reference Manual*. Addison-Wesley, 1999.
8. C. Snook and M. Butler. UML-B: Formal modelling and design aided by UML. pages 92–122. ACM Transactions on Software Engineering and Methodology, ACM Transactions on Software Engineering and Methodology, 15(1), 2006.
9. C. Snook and M. Butler. *U2B - A tool for translating UML-B models into B*, chapter 6. UML-B Specification for Proven Embedded Systems Design. Springer, 2004.
10. J.-R. Abrial and S. Hallerstede. Refinement, Decomposition and Instantiation of Discrete Models: Application to Event-B. In *Fundamentae Informatic*, 2006.
11. I. Johnson, C. Snook, A. Edmunds, and M. Butler. Rigorous development of reusable, domain-specific components, for complex applications. In *Proceedings of 3rd International Workshop on Critical Systems Development with UML*, pages 115–129. Lisbon, 2004.
12. I. Johnson and C. Snook. Rodin Project Case Study 2: Requirements Specification Document. In *Rigorous Open Development Environment for Complex Systems(RODIN), Deliverable D4 - Traceable Requirements Document for Case Studies*, pages 24–52, 2005.
13. J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Addison-Wesley, 2000.
14. R.J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
15. J.-R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In *Proceedings of ICFEM'06, LNCS 4260*, pages 588–605. Springer-Verlag, 2006.
16. C. Snook and M. Walden. Refinement of Statemachines Using Event B Semantics. In *B 2007, LNCS 4355*, pages 171–185. Springer-Verlag, 2006.