Yuliya Prokhorova | Elena Troubitsyna |
Linas Laibinis

Integrating FMEA into Event-B Development of Safety-Critical Control Systems

Turku Centre *for* Computer Science

TUCS Technical Report
No 986, October 2010

Integrating FMEA into Event-B Development of Safety-Critical Control Systems

Yuliya Prokhorova
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, FIN-20520 Turku, Finland
Yuliya.Prokhorova@abo.fi

Elena Troubitsyna
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, FIN-20520 Turku, Finland
Elena.Troubitsyna@abo.fi

Linas Laibinis
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, FIN-20520 Turku, Finland
Linas.Laibinis@abo.fi

TUCS Technical Report
No 986, October 2010

Abstract

Application of formal methods, in particular Event-B, helps us to verify correctness of controlling software. However, to guarantee dependability of software-intensive control systems, we also need to ensure that safety and fault tolerance requirements are adequately represented in a system specification. In this paper we demonstrate how to integrate the results of safety analysis, in particular, failure mode and effect analysis (FMEA), into formal system development in Event-B. FMEA allows us to systematically study the causes of components faults, their global and local effects, and the means to cope with these faults. The fault tolerance mechanisms are often implemented as a part of the developed software, therefore the information obtained as the result of FMEA constitutes the necessary requirements that the controlling software should fulfil. Our approach enables extraction and traceability of the safety requirements that thus potentially increase system dependability. The proposed methodology is exemplified by a case study.

Keywords: Event-B, FMEA, formal methods, safety, safety-critical control systems

TUCS Laboratory
Distributed Systems Laboratory

1. Introduction

A widespread use of software for controlling critical applications necessitates development of techniques for ensuring its correctness. In other words, these techniques should guarantee that software behaves according to its specification. However, to achieve a high degree of system dependability, we should address not only software correctness but also ensure that safety requirements are adequately represented in a software specification.

Safety [11] is property of the system requiring that it will not harm its environment or users. It is a system-level property that can be achieved via a combination of various techniques for safety analysis. The aim of safety analysis is to uncover possible ways in which system might breach safety and then devise the means to avoid these situations or mitigate their consequences. There is a wide spectrum of techniques that facilitate the analysis of possible hazards associated with the system, the means for introducing fault tolerance to prevent occurrence of dangerous faults, as well as the techniques for deriving functional requirements from the conducted safety analysis.

In this paper we focus on the use of Failure Modes and Effect Analysis (FMEA) – a widely-used inductive technique for safety analysis [4] and [11]. We propose a methodology that allows us to incorporate the results of FMEA into a formal system specification. FMEA aims at a systematic study of the causes of components faults, their global and local effects, and the means to cope with these faults. Since the fault tolerance mechanisms are often implemented as a part of the developed software, this information constitutes the necessary requirements that the controlling software should fulfil.

Since safety is a system-level property, it requires modelling techniques that are scalable to analyse the entire system. Scalability in the system analysis is achieved via abstraction, proof and decomposition. The Event-B formalism [1] provides a suitable framework that satisfies all these requirements. Event-B is a state-based formalism for development of highly-dependable systems. The main development technique of Event-B is refinement. In Event-B, we start system modelling at a highly-abstract level and, by a number of correctness-preserving transformations called refinement steps, arrive at a system specification that is close to the eventual implementation. Correctness of each refinement step is verified by proofs.

In this paper we show how to incorporate the results of FMEA into the formal Event-B development. Our approach enables elicitation and traceability of the safety requirements that thus potentially enhance system dependability. The proposed methodology is illustrated by a small case study.

The paper is structured as follows. Section 2 gives an overview of the related work. In Section 3 we briefly present the Event-B method and also describe modelling of control systems in Event-B. In Section 4 we propose a methodology for integrating the results of FMEA into the Event-B development. Section 5 illustrates the proposed approach by a case study – a heater controller. In Section 6 we give concluding remarks and discuss our future work.

2. Modelling Control Systems in Event-B

2.1. Event-B Overview

The B Method is an approach for the industrial development of highly dependable control systems. The method has been successfully used in the development of several complex real-time applications [6]. Event-B [1] is a recent variation of the B Method [2] to model parallel, distributed and reactive systems. The automated tool, which provides a support for modelling and verification in Event-B, is the Rodin platform [3].

To construct and verify system models, Event-B uses the Abstract Machine Notation. An abstract machine encapsulates the state (the variables) of a model and defines operations on its state. A machine has the following general form:

MACHINE *MachineName*
VARIABLES *list of variables*
INVARIANTS *constraining predicates of variables and invariant properties of the machine*
EVENTS
INITIALISATION
evt_1
...
evt_N
END

The machine is uniquely identified by its name *MachineName*. The state variables of the machine are declared in the **VARIABLES** clause and initialized in the *INITIALISATION* event. The variables are strongly typed by constraining predicates of invariants given in the **INVARIANTS** clause. The invariant is usually defined as a conjunction of the state defining the properties of the system that should be preserved during system execution. The model data types and constants are defined in a separate component called **CONTEXT**. The behaviour of the system is defined by a number of atomic events specified in the **EVENTS** clause. An event is defined as follows:

$$E = \textbf{WHEN } g \textbf{ THEN } S \textbf{ END}$$

where the guard g is a conjunction of predicates over the state variables, and the action S is an assignment to the state variables.

The guard defines the conditions when the action can be executed, i.e., when the event is enabled. If several events are enabled simultaneously then any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks.

In general, the action of an event is a composition of variable assignments executed simultaneously. Variable assignments can be either deterministic or non-deterministic. The deterministic assignment is denoted as $x := E(v)$, where x is a state variable and $E(v)$ expression over the state variables v. The non-deterministic assignment can be denoted as $x :\in S$ or $x :| Q(v, x')$, where S is a set of values and $Q(v, x')$ is a predicate. As

a result of the non-deterministic assignment, x gets any value from S or it obtains such a value x' that $Q(v, x')$ is satisfied.

The semantics of Event-B events is defined using before-after predicates [8]. A before-after predicate describes a relationship between the system states before and after execution of an event. The formal semantics provides us with a foundation for establishing correctness of Event-B specifications. To verify correctness of a specification, we need to prove that its initialization and all events preserve the invariant.

To check consistency of an Event B machine, we should verify two types of properties: event feasibility and invariant preservation. Formally, for any event e,

$$Inv(v) \wedge g_e(v) \Rightarrow \exists v' .\ BA_e(v, v')$$

$$Inv(v) \wedge g_e(v) \wedge BA_e(v, v') \Rightarrow Inv(v')$$

where Inv is the model invariant, g_e is the guard of the event e and BA_e is the before-after predicate of the event e.

The main development methodology of Event B is *refinement* – the process of transforming an abstract specification to gradually introduce implementation details while preserving its correctness. Refinement allows us to reduce non-determinism present in an abstract model as well as introduce new concrete variables and events. The connection between the newly introduced variables and the abstract variables that they replace is formally defined in the invariant of the refined model. For a refinement step to be valid, every possible execution of the refined machine must correspond to some execution of the abstract machine.

The consistency of Event B models as well as correctness of refinement steps should be formally demonstrated by discharging *proof obligations*. The Rodin platform [3], a tool supporting Event B, automatically generates the required proof obligations and attempts to automatically prove them. Sometimes it requires user assistance by invoking its interactive prover. However, in general the tool achieves high level of automation (usually over 90%) in proving.

2.2. Modelling Control Systems

In our previous work, we have described how to model control systems in the B method [7]. Therefore, here we just briefly summarize the part that will be necessary for our current research.

In general, a control system is a reactive system with two main entities: a plant and a controller. The plant behaviour evolves according to the involved physical processes and the control signals provided by the controller. The controller monitors the behaviour of the plant and adjusts it to provide intended functionality and maintain safety. In this paper we advocate a system approach to designing controllers for failsafe systems, i.e., we will specify a control system as an event-based system which includes both a plant and a controller.

The control systems are usually cyclic, i.e., at periodic intervals they get input from sensors, process it and output the new values to the actuators. In our specification the sensors and actuators are represented by state variables shared by the plant and the controller. At each cycle the plant reads the variables modelling actuators and assigns

the variables modelling the sensors. In contrast, the controller reads the variables modelling sensors and assigns the variables modelling the actuators (Fig. 1). We assume that the reaction of the controller takes negligible amount of time so the controller can react properly on changes of the plant state.

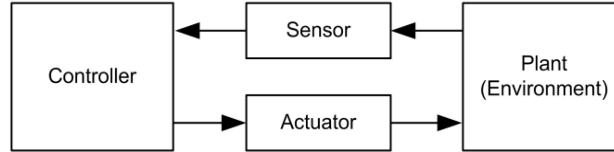
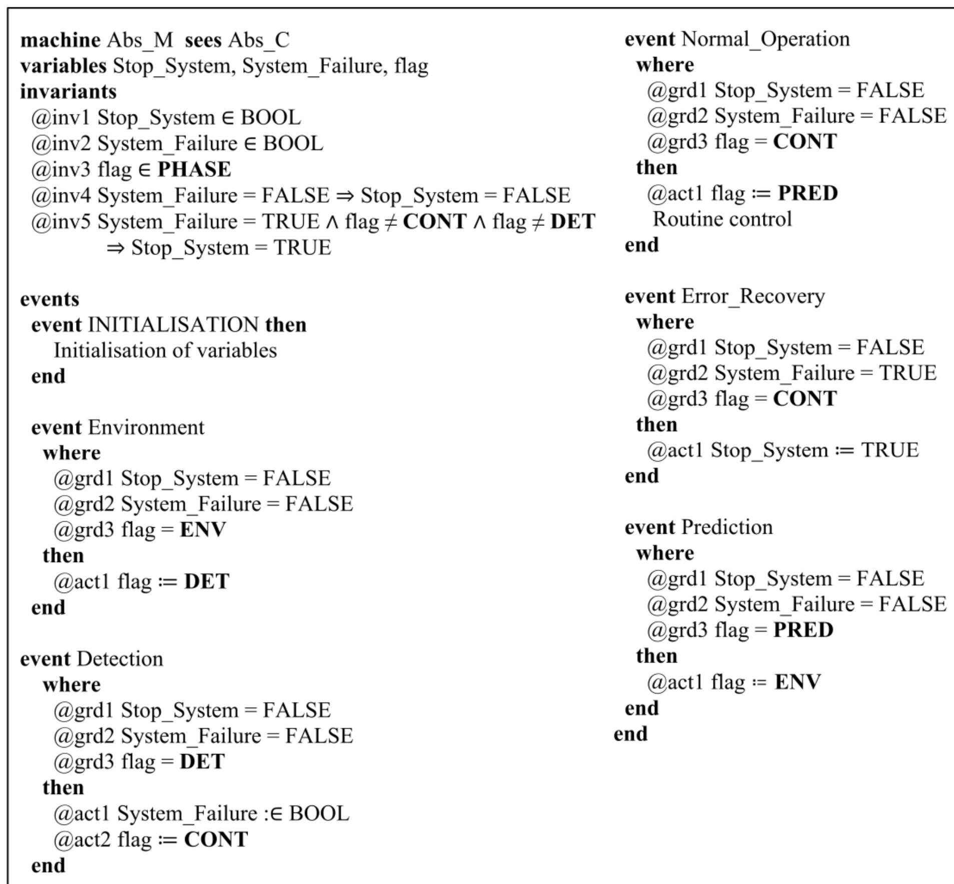


Figure 1: A control system

The development of a failsafe control system in Event-B starts from modelling the abstract machine **Abs_M,** which is shown in Fig. 2. The overall behaviour of the system is an alternation between the events modelling plant evolution and controller reaction. As a result of the initialisation, the plant's operation becomes enabled. Once completed, the plant enables the controller. The behaviour of the controller follows the general pattern

Environment; Detection; Control (Normal Operation or Error Recovery); Prediction
modelled by the corresponding assignments to the variable *flag* of the type **PHASE,** whereas **PHASE** is an enumerated set {ENV, DET, CONT, PRED}.



```
machine Abs_M  sees Abs_C
variables Stop_System, System_Failure, flag
invariants
  @inv1 Stop_System ∈ BOOL
  @inv2 System_Failure ∈ BOOL
  @inv3 flag ∈ PHASE
  @inv4 System_Failure = FALSE ⇒ Stop_System = FALSE
  @inv5 System_Failure = TRUE ∧ flag ≠ CONT ∧ flag ≠ DET
          ⇒ Stop_System = TRUE

events
  event INITIALISATION then
     Initialisation of variables
  end

  event Environment
    where
      @grd1 Stop_System = FALSE
      @grd2 System_Failure = FALSE
      @grd3 flag = ENV
    then
      @act1 flag ≔ DET
  end

event Detection
    where
      @grd1 Stop_System = FALSE
      @grd2 System_Failure = FALSE
      @grd3 flag = DET
    then
      @act1 System_Failure :∈ BOOL
      @act2 flag ≔ CONT
  end

event Normal_Operation
  where
    @grd1 Stop_System = FALSE
    @grd2 System_Failure = FALSE
    @grd3 flag = CONT
  then
    @act1 flag ≔ PRED
     Routine control
end

event Error_Recovery
  where
    @grd1 Stop_System = FALSE
    @grd2 System_Failure = TRUE
    @grd3 flag = CONT
  then
    @act1 Stop_System ≔ TRUE
end

event Prediction
  where
    @grd1 Stop_System = FALSE
    @grd2 System_Failure = FALSE
    @grd3 flag = PRED
  then
    @act1 flag ≔ ENV
end
end
```

Figure 2: An abstract specification of a control system

The operation (event) **Environment** is used for modelling the plant. The operation **Detection** models error occurrence by non-deterministic assignment to the variable *System_Failure*. The operation **Error_Recovery** aborts the system if system failure is detected, i.e., the variable *System_Failure* equals TRUE. The operation **Prediction** is used for modelling expected values of variables. Such a behaviour essentially represents a *failsafe* system. The failsafe error recovery is performed by forcing the system permanently to a safe though non-operational state (obviously, this strategy is only appropriate where shutdown of system is possible). The routine control is specified by the operation **Normal_Operation**.

In this paper we consider safety-critical control systems, therefore safety properties (formalized as safety invariants) should be verified formally, starting from the abstract specification. The safety invariants added to the abstract specification are shown below

$$System_Failure = FALSE \Rightarrow Stop_System = FALSE \text{ and}$$
$$System_Failure = TRUE \wedge flag \neq \mathbf{CONT} \wedge flag \neq \mathbf{DET} \Rightarrow Stop_System = TRUE.$$

The first one states that, while no failure occurred, the system is not stopped. The second requires that, when system failure is detected, the system has to be stopped by the controller.

3. Incorporation of Fault Analysis results in Event-B

In this section we describe a methodology which helps us to incorporate the information obtained as a result of FMEA into our formal specification. The top-down development paradigm adopted by Event-B allows us to implement the fault analysis requirements in a stepwise manner, starting with the abstract specification.

3.1. A Methodology

The development of safety-critical systems starts by identifying possible hazards and proceeds with accumulating the detailed description of them, containing also the necessary means to cope with the identified hazards.

Our methodology based on incorporation of the FMEA results in an Event-B specification of a control system, as it is shown in Fig. 3.

Each refinement step may introduce one or a few system components into our formal specification. According to our methodology, this introduction consists of three steps. We start by making FMEA, which results in a worksheet for each component. It allows us to identify failure modes, possible causes, local and system effects. Then, as an intermediate form, we build an Event-B counterpart worksheet for each component in order to represent each FMEA table field in Event-B terms.

Finally, the obtained results are incorporated into the refined specification. Please, note that system components can be introduced on different abstraction levels, which

means that an abstract component, once introduced, may be late refined, e.g., replaced by several concrete ones.

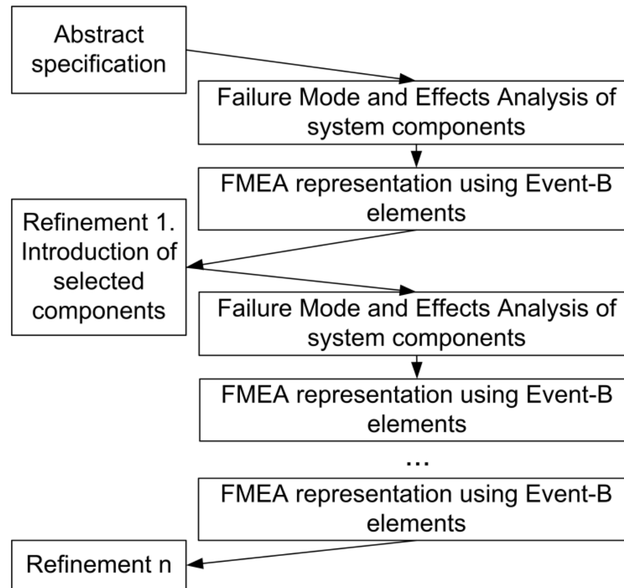


Figure 3: An illustration of the proposed methodology

3.2. FMEA Representation in Event-B

According to our methodology, we built FMEA tables for separate control system components such as sensors (Fig. 4) and actuators (Fig. 5). A fault of any of these components may cause system failure, thus they have similar level of criticality.

The aim of the controller is to keep the behaviour of the environment according to certain design goal. Controller affects the behaviour of the environment by setting certain values to actuators so that desired behaviour can be achieved. Controller observes the effect of changing actuators state by reading the corresponding sensors. Therefore, it is logical to introduce the represent of the actuator and the sensor into system specification at the same time, i.e., within the same refinement step.

Component	**Sensor**
Failure mode	Open circuit or short circuit
Possible cause	Primary hardware failure
Local effects	Sensor constantly sends signal which is less than a min legitimate value or more than a max legitimate value
System effect	System failure
Detection	Sensor value is out of range
Remedial action	Stop system

Figure 4: FMEA worksheet for a sensor

Component	**Actuator**
Failure mode	Stuck at one of two possible modes
Possible cause	Primary hardware failure
Local effects	Non adequate reaction on signals
System effect	System failure
Detection	The received sensor value is not adequate to the predicted one according to the actuator mode
Remedial action	Stop system

Figure 5: FMEA worksheet for an actuator

To illustrate our methodology, let us consider an abstract sensor and an abstract actuator. The failure of the sensor can be detected by the comparison of its received value with the possible one. When the sensor sends a signal that is outside of the legitimate range, we consider such a situation as a fault. The actuator fault can be detected by assumption based on the actuator current mode and the predicted sensor value according to this mode.

Each field of the FMEA table can be represented in an Event-B model by its corresponding elements: variables, constants, events and event guards.

To make the development process in Event-B more clearly to developers, we present guidelines how to represent the results of FMEA of system components in Event-B terms:

- Any system component corresponds to a particular subset of Event-B model variables and events.
- Every component of a failsafe system has to have at least two variables, one to model its current value and the other one to model possible fault occurrence.
- A failure mode is represented by the pre-defined condition on the component variables and a set of the dedicated events enabled when the condition is true. Additional constants (system parameters) may be defined in the accompanying model context.
- System effect is modelled in a formal specification by defining the safety invariants and introducing special variables to model system failure or other degraded state of the system.
- Detection events are tied up with the corresponding failure modes by adding the failure mode condition as an additional guard.
- Remedial action translates into a special operation modelling error recovery.

For example, to represent the sensor in our example, we declare the following variables (Fig. 6): *Sensor_Value* and *Sensor_Fault*. These variables are used in the following events: **Environment**, **Detected_Sensor_Fault**, **Detected_No_Fault**.

The identified failure mode can be formally defined using the constants *Sensor_max_threshold* and *Sensor_min_threshold* (added into the model context). They are detected in the dedicated event: **Detected_Sensor_Fault**. The condition corresponding to the failure mode is *Sensor_Fault = TRUE*.

In this paper we do not consider a situation when components faults can be recovered without shutdown of the whole system. Therefore, any sensor or actuator fault lead to system failure. In Event-B this is represented via the safety invariant

$System_Failure = TRUE \Leftrightarrow Sensor_Fault = TRUE \vee Actuator_Fault = TRUE.$

In other words, when a sensor fault occurs, system has to be stopped. The special event **Error_Recovery** models this situation.

Component	**Variables** Sensor_Value ∈ ℤ Sensor_Fault ∈ BOOL	**Events** Environment Detected_Sensor_Fault Detected_No_Fault
Failure mode	**Constants** Sensor_max_threshold Sensor_min_threshold where Sensor_min_threshold < Sensor_max_threshold	**Events** Detected_Sensor_Fault
	Condition Sensor_Fault = TRUE	
Possible cause	Occur non-deterministically in **event** Environment	
Local effects	Sensor_Value ≤ Sensor_min_threshold ∨ Sensor_Value ≥ Sensor_max_threshold	
System effect	System_Failure = TRUE ⇔ Sensor_Fault = TRUE ∨ Actuator_Fault = TRUE	
Detection	**Event** Detected_Sensor_Fault **Guard** Sensor_Value ≥ **Sensor_max_threshold** ∨ Sensor_Value ≤ **Sensor_min_threshold**	
Remedial action	**Event** Error_Recovery **Guard** Sensor_Fault = TRUE ∨ Actuator_Fault = TRUE **Action** Stop_System ≔ TRUE	

Figure 6: Event-B representation of the FMEA results for a sensor

Similarly, we declare the variables *Actuator_Value* and *Actuator_Fault* to represent an actuator in Event-B (Fig. 7).

Component	**Variables** Actuator_Value ∈ SWITCH {ON, OFF} Actuator_Fault ∈ BOOL	**Events** Environment Detected_Actuator_Fault Detected_No_Fault
Failure mode		**Events** Detected_Actuator_Fault
	Condition Actuator_Fault = TRUE	
Possible cause	Occur non-deterministically in **event** Environment	
Local effects	Sensor_Value > next_s_value_max ∨ Sensor_Value < next_s_value_min	
System effect	System_Failure = TRUE ⇔ Sensor_Fault = TRUE ∨ Actuator_Fault = TRUE	
Detection	**Event** Detected_Actuator_Fault **Guard** Sensor_Value > next_s_value_max ∨ Sensor_Value < next_s_value_min	
Remedial action	**Event** Error_Recovery **Guard** Sensor_Fault = TRUE ∨ Actuator_Fault = TRUE **Action** Stop_System ≔ TRUE Actuator_Value ≔ **OFF**	

Figure 7: Event-B representation of the FMEA results for an actuator

As we described above, to detect the actuator fault, we have to compare the received sensor value with predicted one. The corresponding detection events model the system reaction when the guard *Sensor_Value* > *next_s_value_max* ∨ *Sensor_Value* < *next_s_value_min* is true. The remedial action for the actuator is the same as for the sensor (i.e., system shutdown).

In the following we summarize the proposed methodology:

- the development of a failsafe safety-critical control system in Event-B starts from modelling an abstract machine where system failure and error recovery mechanisms are introduced;
- failure mode and effects analysis for each system component that may causes the system failure is done by building a FMEA worksheet;
- an intermediate representation table where the FMEA results are represented in Event-B terms is created according to the given guidelines;
- the abstract specification is modified according to the FMEA results represented in the intermediate table and proved to be a refinement;
- the described process is iterative. For example, if the control system consists not only from system components that causes the system failure but also from other components, which introduce some redundancy of existing components, the FMEA table is built for each such a component, the intermediate table is created and then the FMEA results are incorporated into the next refined specification;
- all steps can be repeated until we receive the final (most refined) specification, which includes all the system components and formalized requirements.

In the next section we show an application of the proposed methodology.

4. Case Study

To illustrate the proposed methodology, we describe a failsafe control system, which has a controller, a sensor and an actuator. In our case it is a heater case study. The sensor is a temperature sensor and the actuator is a heater. The controller receives a temperature value from the sensor and switches the heater to one of two possible states (ON or OFF) depending on the given temperature range.

Following our methodology we analyse system components and their faults, build a FMEA table and represent the FMEA table fields in Event-B terms, then proceed by refining an abstract specification using the obtained results.

4.1. Temperature Sensor and Heater Implementation

The abstract specification of our case study is very similar to the abstract specification presented in Section 3.2. Therefore, we are going to reuse it for our case study.

As the temperature sensor can be described in a FMEA table in the same way as an abstract sensor, we also reuse its table in this section. The variable *Sensor_Fault* is

Temp_ Sensor_Fault and the variable *Actuator_Fault* is *Heater_Fault* in the renewed case study. The variables and invariants of the refined specification are shown in Fig. 8.

In the refinement we also replace the variable *System_Failure* modelling error occurrence by the variables representing faults of system components, i.e., *Temp_ Sensor_Fault* and *Heater _Fault*. It is an example of data refinement. This data refinement expresses our modelling assumption that the system error occurs only when one or several system components fail. The refinement relation defines the connection between the newly introduced variables and the variables that they replace. While refining the specifications, we add this refinement relation as an additional invariant of the refined machine:

$$System_Failure = TRUE \Leftrightarrow Temp_\ Sensor_Fault = TRUE \vee Heater_Fault = TRUE.$$

The safety invariant then changes from

$$System_Failure = TRUE \wedge flag \neq \mathbf{CONT} \wedge flag \neq \mathbf{DET} \Rightarrow Stop_System = TRUE$$

to

$$(Temp_Sensor_Fault = TRUE \vee Heater_Fault = TRUE) \wedge flag \neq \mathbf{CONT} \wedge flag \neq \mathbf{DET} \Rightarrow Stop_System = TRUE.$$

Also, we formulate an extra safety invariant

$$Temp_Sensor_Fault = FALSE \wedge Heater_Fault = FALSE \wedge flag \neq \mathbf{CONT} \wedge flag \neq \mathbf{DET} \Rightarrow Temp_Sensor_Value < \mathbf{Sensor_max_threshold} \wedge Temp_Sensor_Value > \mathbf{Sensor_min_threshold}.$$

It states that, if there are no temperature sensor and heater faults, the current sensor value is within the expected range.

```
machine Temp_Sensor_Heater_M refines Abs_M  sees Sensor_Actuator_C

variables
Stop_System, System_Failure, flag, Temp_Sensor_Value, next_s_value_max, next_s_value_min
Temp_Sensor_Fault, Heater_Value, Heater_Fault

invariants
  @inv1 Temp_Sensor_Value ∈ ℤ
  @inv2 Temp_Sensor_Fault ∈ BOOL
  @inv3 Heater_Value ∈ SWITCH
  @inv4 Heater _Fault ∈ BOOL
  @inv5 System_Failure = TRUE ⇔ Temp_Sensor_Fault = TRUE ∨ Heater _Fault = TRUE
  @inv6 (Temp_Sensor_Fault = TRUE ∨ Heater _Fault = TRUE) ∧ flag ≠ CONT ∧ flag ≠ DET ⇒
          Heater _Value = OFF
  @inv7 (Temp_Sensor_Fault = TRUE ∨ Heater _Fault = TRUE) ∧ flag ≠ CONT ∧ flag ≠ DET ⇒
          Stop_System = TRUE
  @inv8 Temp_Sensor_Fault = FALSE ∧ Heater _Fault = FALSE ∧ flag ≠ CONT ∧ flag ≠ DET ⇒
          Temp_Sensor_Value < Sensor_max_threshold ∧ Temp_Sensor_Value > Sensor_min_threshold

events
...
end
```

Figure 8: The invariants of the refined specification Temp_Sensor_Heater_M

The operation **Environment**, which is shown in Fig. 9, is used for modelling the plant (i.e., the environment) of the heater. The variable *Temp_Sensor_Value* is updated non-deterministically to model possible value change of the temperature sensor.

```
event Environment refines Environment
  where
    @grd1 Stop_System = FALSE
    @grd2 flag = ENV
    @grd3 Heater_Fault = FALSE
    @grd4 Temp_Sensor_Fault = FALSE
  then
    @act1 flag ≔ DET
    @act2 Temp_Sensor_Value :∈ ℤ
end
```

Figure 9: The operation Environment of the Temp_Sensor_Heater_M specification

The operation **Detected_Sensor_Fault** refines the operation **Detection** at the abstract specification (Fig. 10). We extended the guards clause by adding the results of FMEA according to the Fig. 6. The non-deterministic assignment to the variable *System_Failure* is replaced by the deterministic assignment of the variable *Temp_Sensor_Faul*. It becomes equal to TRUE, thus indicating a detected sensor fault.

```
event Detected_Sensor_Fault refines Detection
  where
    @grd1 Stop_System = FALSE
    @grd2 flag = DET
    @grd3 Temp_Sensor_Value ≥
          Sensor_max_threshold ∨
          Temp_Sensor_Value ≤
          Sensor_min_threshold
    @grd4 Temp_Sensor_Fault = FALSE
    @grd5 Heater_Fault = FALSE
  then
    @act1 Temp_Sensor_Fault ≔ TRUE
    @act2 flag ≔ CONT
end

event Detected_Actuator_Fault refines Detection
  where
    @grd1 Stop_System = FALSE
    @grd2 flag = DET
    @grd3 Temp_Sensor_Fault = FALSE
    @grd4 Heater_Fault = FALSE
    @grd5 Temp_Sensor_Value > next_s_value_max ∨
       Sensor_Value < next_s_value_min
    @grd6 Temp_Sensor_Value >
       Sensor_min_threshold
    @grd7 Temp_Sensor_Value <
       Sensor_max_threshold
  then
    @act1 Heater_Fault ≔ TRUE
    @act2 flag ≔ CONT
end

event Detected_No_Fault refines Detection
  where
    @grd1 Stop_System = FALSE
    @grd2 flag = DET
    @grd3 Temp_Sensor_Value <
          Sensor_max_threshold
    @grd4 Temp_Sensor_Value >
          Sensor_min_threshold
    @grd5 Temp_Sensor_Fault = FALSE
    @grd6 Heater_Fault = FALSE
    @grd7 Sensor_Value ≤ next_s_value_max
    @grd8 Sensor_Value ≥ next_s_value_min
  then
    @act1 flag ≔ CONT
end
```

Figure 10: The operations Detection of the Temp_Sensor_Heater_M specification

The operation **Detected_Actuator_Fault** also refines the operation **Detection**. We strengthened the operation guard by adding new guards according to the results of FMEA, shown in Fig. 7. The non-deterministic assignment to the variable *System_Failure* is replaced by the deterministic assignment of the variable *Heater_Fault*. It becomes equal to TRUE. **Detected_No_Fault** is another refinement of the operation **Detection**. However, the non-deterministic assignment to the variable *System_Failure* is not replaced by any of two variables, because they are already equal to FALSE.

After the execution of one of the detection events discussed above the system has three ways to continue its execution. The first case is when the temperature sensor or the heater faults occur and as a result the system has to be stopped. Thus, the operation **Error_recovery,** which is identical to its abstract counterpart**,** becomes enabled. The other two cases are when there is no fault and system is functioning in the normal mode (Fig. 11). These two events differ from each other by their guards and respective actions. In one case, if the temperature sensor value is less than the maximum value but more or equal than the middle value, the variable *Heater_Value* is assigned OFF. In the other case, if the temperature sensor value is more than the minimum value but less than the middle value, the variable *Heater_Value* is assigned ON.

```
event Normal_Operation_1 refines Normal_Operation
 where
  @grd1 Stop_System = FALSE
  @grd2 flag = CONT
  @grd3 Heater_Fault = FALSE
  @grd4 Temp_Sensor_Fault = FALSE
  @grd5 Temp_Sensor_Value <
          Sensor_max_threshold
  @grd6 Temp_Sensor_Value ≥
          Sensor_middle_val
 then
  @act1 Heater_Value ≔ OFF
  @act2 flag ≔ PRED1
end

event Normal_Operation_2 refines Normal_Operation
 where
  @grd1 Stop_System = FALSE
  @grd2 flag = CONT
  @grd3 Heater_Fault = FALSE
  @grd4 Temp_Sensor_Fault = FALSE
  @grd5 Temp_Sensor_Value >
          Sensor_min_threshold
  @grd6 Temp_Sensor_Value <
          Sensor_middle_val
 then
  @act1 Heater_Value ≔ ON
  @act2 flag ≔ PRED2
end

event Prediction1 refines Prediction
 where
  @grd1 flag = PRED1
  @grd2 Stop_System = FALSE
  @grd3 Heater_Fault = FALSE
  @grd4 Temp_Sensor_Fault = FALSE
  @grd5 Heater_Value = OFF
 then
  @act1 next_s_value_max ≔
min_dec(Sensor_Value)
  @act2 next_s_value_min ≔
max_dec(Sensor_Value)
  @act3 flag ≔ ENV
 end

event Prediction2 refines Prediction
 where
  @grd1 flag = PRED2
  @grd2 Stop_System = FALSE
  @grd3 Heater_Fault = FALSE
  @grd4 Temp_Sensor_Fault = FALSE
  @grd5 Heater_Value = ON
 then
  @act1 next_s_value_max ≔
max_inc(Sensor_Value)
  @act2 next_s_value_min ≔
min_inc(Sensor_Value)
  @act3 flag ≔ ENV
 end
```

Figure11: The operations Normal_Operation and Prediction of the Temp_Sensor_Heater_M specification

In the next section we will make our model more tolerant by introducing the triple module redundancy (TMR) arrangement for our sensor.

4.2. TMR Implementation of the Temperature Sensor

In the specification obtained at the previous refinement step all errors are considered to be equally critical, i.e., leading to the shutdown. While introducing redundancy at our next refinement step, we obtain a possibility to distinguish between criticality of errors and mask a single error of a system component. Application of Triple Modular Redundancy (TMR) [11] in that case allows us to mask faults of a single sensor. TMR is a well-known mechanism based on static redundancy. The general principle is to triplicate a system module and introduce the majority voting to obtain a single result of the module, as shown in Fig. 12.

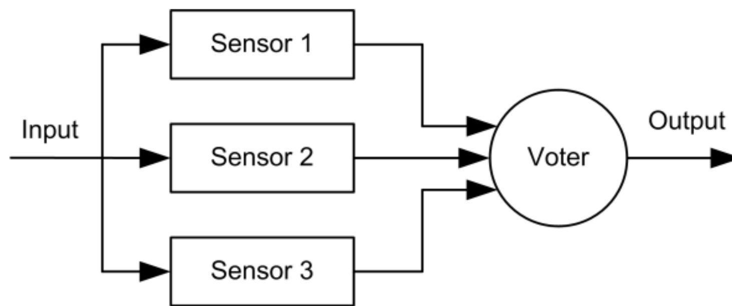


Figure 12: Sensor TMR

Fig. 13 shows the control system described in Section 3.2 with three temperature sensors. In our case study we model the temperature sensors and a voter as parts of a plant. The controller only receives the result of voting and does not see particular sensors.

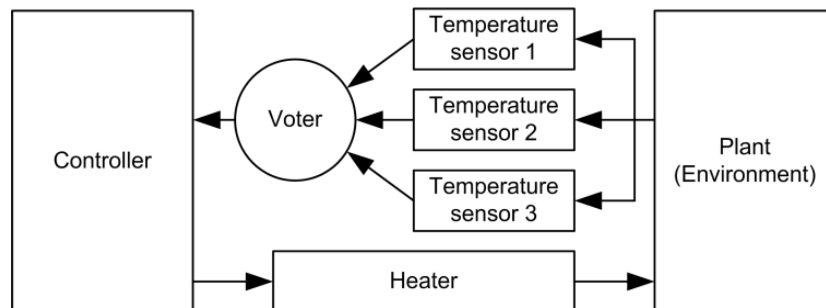


Figure13: The case study system with the temperature sensor TMR

Following our proposed methodology, we refine the specification obtained in the previous section in order to add the sensor TMR arrangement. However, before describing the refinement step formally, we have to analyse all possible failure modes and effects associated with new introduced sensors, and consequently build the FMEA table (Fig. 14) as well as the intermediate representation table (Fig. 15) for each newly introduced component.

Component	**Temperature sensor TMR: Temperature sensor1,** **Temperature sensor2,** **Temperature sensor3**
Failure mode	If more than one temperature sensor is failed then temperature sensor fault occurs
Possible cause	Primary hardware failure
Local effects	More than two temperature sensors failed
System effect	System failure
Detection	Values of all tree temperature sensors are not equal to each other
Remedial action	Stop system

Figure 14: The FMEA table for the temperature sensor TMR

The representation of FMEA results in Event-B is shown in Fig. 15. The temperature sensor TMR is modelled by using the following variables: *Temp_Sensor1_Value*, *Temp_Sensor2_Value*, *Temp_Sensor3_Value* and the variable *Temp_Sensor_Fault*, which is equal to the variable *Temp_Sensor_Fault* in the previous refinement step, and the events **Environment1** and **Environment2_1** … **Environment2_5**, which are shown in Fig. 16. The last five events are used for modelling the TMR voter.

Component	**Variables** Temp_Sensor1_Value ∈ ℤ Temp_Sensor2_Value ∈ ℤ Temp_Sensor3_Value ∈ ℤ Temp_Sensor_Fault ∈ BOOL	**Events** Environment1 Environment2_1 Environment2_2 Environment2_3 Environment2_4 Environment2_5
Failure mode		**Events** Detected_Sensor_Fault
	Condition Temp_Sensor_Fault = TRUE	
Possible cause	Occur non-deterministically in **event** Environment1	
Local effects	Temp_Sensor1_Value ≠ Temp_Sensor2_Value Temp_Sensor1_Value ≠ Temp_Sensor3_Value Temp_Sensor2_Value ≠ Temp_Sensor3_Value	
System effect	Temp_Sensor1_Value ≠ Temp_Sensor2_Value ∧ Temp_Sensor1_Value ≠ Temp_Sensor3_Value ∧ Temp_Sensor2_Value ≠ Temp_Sensor3_Value ⇒ Temp_Sensor_Fault = TRUE System_Failure = TRUE ⇔ Temp_Sensor_Fault = TRUE ∨ Heater_Fault = TRUE	
Detection	**Events** Environment2_5 **Guards** Temp_Sensor1_Value ≠ Temp_Sensor2_Value Temp_Sensor1_Value ≠ Temp_Sensor3_Value Temp_Sensor2_Value ≠ Temp_Sensor3_Value	
Remedial action	**Event** Error_Recovery **Guard** Temp_Sensor_Fault = TRUE ∨ Heater_Fault = TRUE **Action** Stop_System ≔ TRUE	

Figure 15: Event-B representation of FMEA for the temperature sensor TMR

The occurrence of three temperature sensors faults are introduced in the operation **Environment1** by non-deterministic assignment of the appropriate variables. When

new sensors values are assigned, the voter can make a decision by identifying the failed sensor and taking the majority view. The operations **Environment2_1**, **Environment2_2** and **Environment2_3** are similar. They have the guards checking whether two temperature sensors values are equal. The actions in these events assign one of the equal values to the variable *Temp_Sensor_Value*. The operation **Environment2_4** checks that all tree sensors have equal values, while its action assigns one of values to the variable *Temp_Sensor_Value*. The operation **Environment2_5** compares sensors values on non-equality and assigns the variable *Temp_Sensor_Value* with the constant *Sensor_Err_val* the value of which is less than *Sensor_min_threshold*. It means that, if there are more than one temperature sensor faults in the system, the system has to be stopped.

```
event Environment1
 where
  @grd1 Stop_System = FALSE
  @grd2 flag1 = ENV1
  @grd3 Heater_Fault = FALSE
  @grd4 Temp_Sensor_Fault = FALSE
 then
  @act1 Temp_Sensor1_Value :∈ ℤ
  @act2 Temp_Sensor2_Value :∈ ℤ
  @act3 Temp_Sensor3_Value :∈ ℤ
  @act4 flag1 ≔ ENV2
end

event Environment2_1 refines Environment
 where
  @grd1 Stop_System = FALSE
  @grd2 flag1 = ENV2
  @grd3 Heater _Fault = FALSE
  @grd4 Temp_Sensor1_Value =
        Temp_Sensor2_Value
  @grd5 Temp_Sensor1_Value ≠
        Temp_Sensor3_Value
  @grd6 Temp_Sensor2_Value ≠
        Temp_Sensor3_Value
  @grd7 Temp_Sensor_Fault = FALSE
 then
  @act1 Temp_Sensor_Value :∈
        {Temp_Sensor1_Value,
         Temp_Sensor2_Value}
  @act2 flag1 ≔ DET1
end

event Environment2_4 refines Environment
 where
  @grd1 Stop_System = FALSE
  @grd2 flag1 = ENV2
  @grd3 Heater _Fault = FALSE
  @grd4 Temp_Sensor1_Value =
Temp_Sensor2_Value
  @grd5 Temp_Sensor1_Value =
Temp_Sensor3_Value
  @grd6 Temp_Sensor2_Value =
Temp_Sensor3_Value
  @grd7 Temp_Sensor_Fault = FALSE
 then
  @act1 Temp_Sensor_Value :∈ {
Temp_Sensor1_Value,
        Temp_Sensor2_Value,
Temp_Sensor3_Value}
  @act2 flag1 ≔ DET1
end

event Environment2_5 refines Environment
 where
  @grd1 Stop_System = FALSE
  @grd2 flag1 = ENV2
  @grd3 Heater _Fault = FALSE
  @grd4 Temp_Sensor1_Value ≠
Temp_Sensor2_Value
  @grd5 Temp_Sensor1_Value ≠
Temp_Sensor3_Value
  @grd6 Temp_Sensor2_Value ≠
Temp_Sensor3_Value
  @grd7 Temp_Sensor_Fault = FALSE
 then
  @act1 Temp_Sensor_Value ≔
Sensor_Err_val
  @act2 flag1 ≔ DET1
end
```

Figure16: The operations Environment in the temperature sensor TMR specification

In this paper, we applied the proposed methodology for the heater case study. The resulting specification were proven to show that the final specification of the system meets all safety requirements, in particularly, that system failure always leads to the necessary error recovery actions.

5. Related Work

Integration of the safety analysis techniques with formal system modelling has attracted a significant research attention over the last few years. There are a number of approaches that aim at direct integration of the safety analysis techniques into formal system development. For instance, the work of Ortmeier et al. [9] focuses on using statecharts to formally represent the system behaviour. It aims at combining the results of FMEA and FTA to model the system behaviour and reason about component failures as well as overall system safety. Moreover, the approach specifically addresses formal modelling of the system failure modes. In our approach we define general guidelines for integrating results of FMEA into a formal Event-B specification and the Event-B refinement process. The available automatic tool support for the top-down Event-B modelling ensures better scalability of our approach.

In our previous work, we have proposed an approach to integrating safety analysis into formal system development within the Action System formalism [10, 13]. Since Event-B incorporates the ideas of Action Systems into the B Method, the current work is a natural extension of our previous results.

The research conducted by Troubitsyna [12] aims at demonstrating how to use statecharts as a middle ground between safety analysis and formal system specifications in the B Method. In our future work we will rely on this research to define patterns for formal representation of system components as formal specifications in Event-B.

Another strand of research aims at defining general guidelines for ensuring dependability of software-intensive systems. For example, Hatebur and Heisel [5] have derived patterns for representing dependability requirements and ensuring their traceability in the system development. In our approach we rely on specific safety analysis techniques rather than on the requirements analysis in general to derive guidelines for modelling dependable systems.

6. Conclusions

In this paper we presented an approach to integrating the safety analysis techniques into the formal system development in Event-B. We demonstrated how to derive safety requirements from FMEA in such a way that they could be easily captured in a formal system specification. Our methodology facilitates requirements elicitation as well as supports traceability of safety requirements within the formal development process. The proposed guidelines for modelling components in Event-B demonstrate how to relate specific fields in FMEA work-sheets with the corresponding elements of an Event-B specification. As a result, the proposed approach integrates the means for fault avoidance and fault tolerance and hence can potentially enhance dependability of safety-critical control systems.

In our future work we are planning to create a library of formal models representing typical components (sensors and actuators), error detecting mechanisms and recovery actions. Such a library would allow us to define the typical refinement transformations supporting correct incorporation of the safety analysis results into a formal system

specification. Moreover, it also would enable automatization of the refinement process to support such pre-defined model transformations. We aim at exploring this approach within a certain dedicated domain of critical systems.

In this paper we focused on analysing the requirements originating from the inductive safety techniques. However, safety analysis usually combines several different techniques that allow the designers to explore different aspects of system safety. While FMEA provides us with a systematic way to analyse the failure modes of components, it is unable to address the analysis of multiple system failures. In our future work we aim at investigating how to combine the FMEA approach with such techniques as fault tree analysis to guarantee safety in the presence of several component failures.

References

[1] J.-R. Abrial, "Modeling in Event-B: System and Software Engineering", Cambridge University Press, 2010.

[2] J.-R. Abrial, "The B-Book: Assigning Programms to Meanings", Cambridge University Press, 1996.

[3] Event-B and the Rodin Platform. Retrieved from http://www.event-b.org/, 2010.

[4] FMEA Info Centre. Retrieved from http://www.fmeainfocentre.com/, 2009.

[5] D. Hatebur and M. Heisel, "A Foundation for Requirements Analysis of Dependable Software", Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP), Springer, 2009, pp. 311-325.

[6] Industrial use of the B method. Retrieved from ClearSy: http://www.clearsy.com/pdf/ClearSy-Industrial_Use_of_%20B.pdf, 2008.

[7] L. Laibinis, and E. Troubitsyna, "Refinement of fault tolerant control systems in B", TUCS Technical Report, No. 603, 2004.

[8] C. Métayer, J.-R. Abrial, and L. Voisin, "Rigorous Open Development Environment for Complex Systems (RODIN). Event-B". Retrieved from http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, 2005.

[9] F. Ortmeier, M. Guedemann and W. Reif, "Formal Failure Models", Proceedings of the IFAC Workshop on Dependable Control of Discrete Systems (DCDS 07), Elsevier, 2007.

[10] K. Sere, and E. Troubitsyna, "Safety analysis in formal specification". In J. Wing, J. Woodcock, & J. Davies (Ed.), FM'99 – Formal Methods. Proceedings of World Congress on Formal Methods in the Development of Computing Systems, Lecture Notes in Computer Science 1709, II, 1999, pp. 1564-1583.

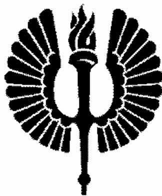
[11] N. Storey, "Safety-critical computer systems", Addison-Wesley, 1996.

[12] E. Troubitsyna, "Elicitation and Specification of Safety Requirements", Proceedings of the Third International Conference on Systems (ICONS 2008), 2008, pp. 202-207.

[13] E. Troubitsyna, "Integrating Safety Analysis into Formal Specification of Dependable Systems", Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), 2003, p. 215b.

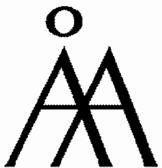
TURKU CENTRE *for* COMPUTER SCIENCE

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 978-952-12-2476-8
ISSN 1239-1891