Patterns for Representing FMEA in Formal Specification of Control Systems

Ilya Lopatkin, Alexei Iliasov,
Alexander Romanovsky
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
{Ilya.Lopatkin, Alexei.Iliasov,
Alexander.Romanovsky}@ncl.ac.uk

Yuliya Prokhorova, Elena Troubitsyna
Turku Centre for Computer Science
Department of Information Technologies
Åbo Akademi University
Turku, Finland
{Yuliya.Prokhorova, Elena.Troubitsyna}@abo.fi

***Abstract* — Failure Modes and Effects analysis (FMEA) is a widely used technique for inductive safety analysis. FMEA provides engineers with valuable information about failure modes of system components as well as procedures for error detection and recovery. In this paper we propose an approach that facilitates representation of FMEA results in formal Event-B specifications of control systems. We define a number of patterns for representing requirements derived from FMEA in formal system model specified in Event-B. The patterns help the developers to trace the requirements from safety analysis to formal specification. Moreover, they allow them to increase automation of formal system development by refinement. Our approach is illustrated by an example - a sluice control system.**

Keywords - formal specification; Event-B; FMEA; patterns; safety; control systems

I. INTRODUCTION

A. Motivation and Overview of an Approach

Formal modelling and verification are valuable for ensuring system dependability. However, often formal development process is perceived as being too complex to be deployed in the industrial engineering process. Hence, there is a clear need for methods that facilitate adopting of formal modelling techniques and increase productivity of their use.

Reliance on patterns – the generic solutions for certain typical problems – facilitates system engineering. Indeed, it allows the developers to document the best practices and reuse previous knowledge.

In this paper we propose an approach to automating formal system development by refinement. We connect formal modelling and refinement with Failure Modes and Effects Analysis (FMEA) via a set of patterns.

FMEA is a widely-used inductive technique for safety analysis [5,13,16]. We define a set of patterns formalising the requirements derived from FMEA and automate their integration into the formal specification. Our formal modelling framework is Event-B – a state-based formalism for formal system development by refinement and proof-based verification [1]. Currently, the framework is actively used by several industrial partners of EU FP7 project Deploy [2] for developing dependable systems from various domains.

The approach proposed in this paper allows us to automate the formal development process via two main steps: *choice of suitable patterns that generically define FMEA result*, and *instantiation of chosen patterns with model-specific information.* We illustrate this process with excerpts from the automated development of a sluice gate system [7].

Our approach allows the developers to verify (by proofs) that safety invariants are preserved in spite of identified component failures. Hence we believe that it provides a useful support for formal development and improves traceability of safety requirements.

B. Related Work

Over the last few years integration of the safety analysis techniques into formal system modelling has attracted a significant research attention. 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. [15] 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. Our approach is different – we aim at automating the formal system development with the set of patterns instantiated by FMEA results. The application of instantiated patterns automatically transforms a model to represent the results of FMEA in a coherent and complete way. The available automatic tool support for the Event-B modelling as well as for plug-in instantiation and application ensures better scalability of our approach.

In our previous work, we have proposed an approach to integrating safety analysis into formal system development within Action Systems [18]. 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 [19] aims at demonstrating how to use statecharts as a middle ground between safety analysis and formal system specifications in the B Method. This work has inspired our idea of deriving Event-B patterns.

Patterns defined for formal system development by Hoang et al. [17] focus on describing model manipulations only and do not provide the insight on how to derive a formal model from a textual requirements description that has a negative impact on requirements traceability.

Another strand of research aims at defining general guidelines for ensuring dependability of software-intensive systems. For example, Hatebur and Heisel [6] 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.

II. MODELLING CONTROL SYSTEMS IN EVENT-B

A. *Event-B Overview*

Event-B [1] is a specialisation of the B Method aimed at facilitating modelling of parallel, distributed and reactive systems [9]. The Rodin Platform provides an automated support for modelling and verification in Event-B [4].

In Event-B system models are defined using the Abstract Machine Notation. An abstract machine encapsulates the state (the variables) of a model and defines operations on its state. The machine is uniquely identified by its name. 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. Usually the invariants also define the properties of the system that should be preserved during system execution. The data types and constants of the model 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:

$$E = \textbf{ANY}\ lv\ \textbf{WHERE}\ g\ \textbf{THEN}\ S\ \textbf{END}$$

where *lv* is a list of new local variables, the guard *g* is a conjunction of predicates defined over the state variables, and the action *S* is an assignment to the state variables.

The guard defines 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 $E(v)$ is an 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 main development methodology of Event-B is *refinement*. Refinement formalises model-driven development and allows us to develop systems correct-by-construction. Each refinement transforms the abstract specification to gradually introduce implementation details. For a refinement step to be valid, every possible execution of the refined machine must correspond to some execution of the abstract machine.

Next we describe specification and refinement of control systems in Event-B. It follows the specification pattern proposed earlier [11].

B. *Modelling Control Systems*

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 the corresponding state variables. We follow the systems approach, i.e., model the controller together with its environment – plant. This allows us to explicitly state the assumptions about environment behaviour. At each cycle the plant assigns the variables modelling the sensor readings. They depend on the physical processes of the plant and the current state of the actuators. In its turn, the controller reads the variables modelling sensors and assigns the variables modelling the actuators. We assume the controller reaction takes negligible amount of time and hence the controller can react properly on changes of the plant state.

In this paper, we focus on modelling failsafe control systems. A system is failsafe if it can be put into a safe but non-operational state to preclude an occurrence of a hazard.

The general specification pattern **Abs_M** for modelling a failsafe control system in Event-B is presented in [14]. It represents the overall behaviour of the system as an interleaving between the events modelling the plant and the controller. The behaviour of the controller has the following stages: *Detection; Control (Normal Operation* or *Error Handling); Prediction.* The variable *flag* of type **PHASE:**{ENV, DET, CONT, PRED} models the current stage.

In the model invariant we declare the types of the variables and define the operational conditions. The system is operational if it has not failed. However, it must be stopped at the end of the current cycle if a failure occurred.

The events **Environment, Normal_Operation** and **Prediction** abstractly model environment behaviour, controller reaction and computation of the next expected states of system components correspondingly. The event **Detection** non-deterministically models the outcome of the error detection. A result of error recovery is abstractly modelled by the event **Error_Handling**.

In the next section we demonstrate how to arrive at a detailed specification of a control system by refinement in Event-B. We use the sluice gate control system to exemplify the refinement process.

III. REFINEMENT OF CONTROL SYSTEMS IN EVENT-B

A. *The Sluice Gate Control System*

The general specification pattern **Abs_M** given in [14] defines the initial abstract specification for any typical control system. The sluice gate control system shown in Fig. 1 is among them. The system is a sluice connecting areas with dramatically different pressures [7]. The purpose of the system is to adjust the pressure in the sluice area. The sluice gate system consists of two doors - *door1* and *door2* that can be operated independently of each other and *a pressure chamber pump* that changes the pressure in the sluice area. To guarantee safety, a door may be opened only if the pressure in the locations it connects is equalized. Moreover, at most one door can be opened at any moment and the pressure chamber pump can only be switched on when both doors are closed.

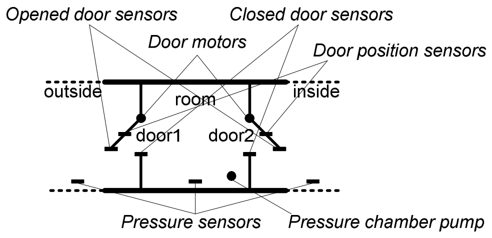

Figure 1. Sluice gate system.

The sluice gate system is equipped with the sensors and actuators shown in Fig.1. The system has physical redundancy - the door position sensors have spares; and information redundancy - when the doors are fully opened or closed, the door position sensor readings should match the readings of the switch sensors.

B. *Introducing Error Detection and Recovery by Refinement*

At the first refinement step we aim at introducing models of system components, error detection procedures as well as error masking and recovery actions.

To systematically define failure modes, detection and recovery procedures, for each component, we conduct Failure Modes and Effects Analysis. FMEA [5,13,16] is a well-known inductive safety analysis technique. For each system component it defines its possible failure modes, local and system effects of a failure as well as detection and recovery procedures. Fig. 2 shows an excerpt from FMEA of the *Door1* component of our sluice system.

The *Door1* component is composed of several hardware units. Their failures correspond to the failure modes of the *Door1* component. Next we discuss how to specify error detection and recovery for the failure mode described in the FMEA table in Fig. 2.

Component	Door1
Failure mode	Door position sensor value is different from the door closed sensor value
Possible cause	Failure of position sensor or closed sensor
Local effects	Sensor readings are not equal in corresponding states
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of the values received from position and closed sensors
Remedial action	Retry. If failure persists then switch to redundant sensor, diagnose motor failure. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm.

Figure 2. FMEA table

In the refined specification we introduce the variables representing the units of *Door1*: door position sensor - *door1_position_sensor*, motor - *door1_motor* and door opened and closed sensors - *door1_opened_sensor*, *door1_closed_sensor*. In the event **Environment** we introduce the actions that change the values of *door1_position_sensor*, *door1_closed_sensor* and *door1_opened_sensor*. The event **Normal_Operation** defines the action that non-deterministically changes the value of *door1_motor*.

We refine the event **Detection** by splitting it into a group of events responsible for the detection of each failure mode of all system components. We introduce the variable *door1_fail* to designate a failure of the door component. This failure is assigned TRUE when any failure mode of *Door1* component is detected. The event **Detection_door1_checks** included in this group contains the actual checks for the value ranges and consistency. The variables *d1_exp_min* and *d1_exp_max* are the new variables introduced to model the next expected sensor readings. These variables are updated in the **Prediction** event. The event **Detection_Door1** combines the results of the checks of the status of the *door1* component.

```
event Detection_Door1_checks
  where
   flag = DET ∧ Stop = FALSE
  then
   door1_position_sensor_pred ≔ bool((door1_position_sensor <
        d1_exp_min ∨ door1_position_sensor > d1_exp_max) ∧
        door1_sensor_disregard=FALSE)
   door1_closed_sensor_inconsistent ≔
        bool(¬(door1_closed_sensor=TRUE ⇔
           (door1_position=0 ∨ door1_sensor_disregard=TRUE)))
   <other checks>
end
```

The failure of the component *Door1* is detected if any check of the error detection events for any of its failure modes finds a discrepancy between a fault free and the observed states. In a similar manner, the system failure is detected if a failure of any of the system components – *Door1*, *Door2* or *PressurePump* is detected, as specified in the event **Detection_Fault**.

```
event Detection_Door1
 where
  flag = DET ∧ Stop = FALSE
 then door1_fail ≔  bool( door1_position_sensor_pred=TRUE ∨
               door1_closed_sensor_inconsistent=TRUE ∨
               <other check statuses>)
end
event Detection_Fault refines Detection
 where
  flag = DET ∧ Stop = FALSE
  door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail = TRUE
 with Failure' Failure'=TRUE
 then flag ≔ CONT
end
```

Observe that by performing FMEA of each system component we obtain a systematic textual description of all procedures required to detect component errors and perform

Component	Door1
Failure mode	Door position sensor value is different from the expected range of values
Possible cause	Failure of the position sensor
Local effects	Sensor reading is out of expected range
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of the received value with the predicted range of values
Remedial action	The same as for Fig. 2

Figure 3. FMEA table for "out of predicted range" failure mode of a positioning sensor

their recovery. We gradually (by refinement) introduce the specification of these requirements into the system model.

While analysing the refined specification, it is easy to note that there are several typical specification solutions called patterns that represent certain groups of requirements. This observation prompts the idea of creating an automated tool support that would automatically transform a specification by applying the patterns chosen and instantiated by the developer. In the next section we describe the essence of such a tool.

IV. Patterns and Tool for Representing Results of FMEA in Event-B

A. *Patterns for Representing FMEA Results*

Our approach aims at structuring and formalising FMEA results via a set of generic patterns. These patterns serve as a middle hand between informal requirements description and their formal Event-B model.

While deriving the patterns we assume that the abstract system specification adheres to the generic pattern given in [14]. Moreover, we also assume that components can be represented by the corresponding state variables. Our patterns establish a correspondence between the results of FMEA and the Event-B terms.

We distinguish four groups of patterns: detection, recovery, prediction and invariants. The detection patterns reflect such generic mechanisms for error detection as discrepancy between the actual and expected component state, sensor reading outside of the feasible range etc. The recovery patterns include retry of actions or computations, switch to redundant components and safe shutdown. The prediction patterns represent the typical solutions for computing estimated states of components, e.g., using the underlying physical system dynamics or timing constraints. Finally, the invariant patterns are usually used in combination with other types of patterns to postulate how a model transformation affects the model invariant. This type contains safety and gluing invariant patterns. The safety invariant patterns define how safety conditions can be introduced into the model. The gluing invariant patterns depict the correspondence between the states of refined and abstract model.

A *pattern* is a model transformation that upon instantiation adds or modifies certain elements of Event-B model. By *elements* we mean the terms of Event-B mathematical language such as variables, constants, invariants, events, guards etc. A pattern can add or modify several elements at once. Moreover, it can be composed of several other patterns.

To illustrate how to match FMEA results with the proposed patterns, let us consider FMEA of a door1 position sensor shown in Fig. 3.

To simplify illustration, the patterns are shown in a declarative form. The identifiers shown in brackets should be substituted by those given by a user during the pattern instantiation (see next sections).

Our sensor is a value type sensor. Hence we can apply the *Value sensor pattern* to introduce the model of the sensor into our specification:

```
variables [sensor]_value
invariants
   [sensor]_value : NAT
events
   event INITIALISATION
   then
      [sensor]_value := 0
   end
end
```

An application of the value sensor pattern leads to creating a new variable, its typing invariant, and an initialisation action. To model identified detection of the failure mode, we use the *Expected range pattern*:

```
variables
   [component]_[sensor]_[error], [component]_fail, [sensor]_exp_min,
   [sensor]_exp_max
invariants
   [component]_[sensor]_[error] : BOOL
   [component]_fail : BOOL
   [sensor]_exp_min : NAT
   [sensor]_exp_max : NAT
events
  event Detection_[component]_checks
  where flag = DET ∧ Stop = FALSE then
  [component]_[sensor]_[error] ≔ bool(
  [sensor]_value<[sensor]_exp_min∨[sensor]_value>[sensor]_exp_max)
  <other checks>
   end
  event Detection_[component]
  where flag = DET ∧ Stop = FALSE then
  [component]_fail ≔  bool([component]_[sensor]_[error] ∨
  <other check statuses>)
  end
end
```

This pattern adds the detection events and the variables required to model error detection: expected minimal and maximal values. The pattern ensures that the detection checks added previously by other patterns are preserved (this is informally shown in the angle brackets). The expected range of values used by this pattern must be assigned by some event in the previous control cycle. To ensure that such assignment exists in the model, the *Expected range pattern* instantiates the *Range prediction pattern*. An application of this pattern results in a non-deterministic specification of prediction. It can be further refined to take into account the specific functionality of the system under development.

Let us observe that the *Expected range pattern* and *Range prediction pattern* affect the same variables. To avoid conflicts and inconsistencies, only the first pattern to be instantiated actually creates the required variables. The same rule applies to events, actions, guards etc.

To establish refinement between the model created using patterns and the abstract model, we use the *Gluing invariant pattern*, which links the sensor failure with the component failure:

```
variables
   [component]_fail
invariants
   flag≠DET ⇒ (Failure=TRUE ⇔ [component]_fail=TRUE ∨
                    <other component failures>)
   flag≠CONT ⇒  ([component]_fail=TRUE ⇔
                    [component]_[sensor]_[error]=TRUE ∨
                    <other sensor errors>)
```

In our example, the remedial action can be divided into three actions. The first action retries reading the sensor for a specified number of times (*Retry recovery pattern*). The second action deactivates the faulty component and activates its spare (*Component redundancy recovery pattern*). The third action is enabled when the spare component has also failed. It switches the system from the operational state to the non-operational one (*Safe stop recovery pattern*). The system effect can be represented as a safety property (*Safety invariant pattern*). We omit showing all the patterns due to the lack of space.

As shown in the example, each FMEA field is mapped to one or more patterns. The patterns have interdependencies. Moreover, they are composable. For instance, the *recovery patterns* reference the variables set by the sensor, and thus depend on the results of the *Value sensor pattern*. The *Expected value detection pattern* needs to instantiate the *Range prediction pattern* to rely on the values predicted at the previous control cycle. Each pattern creates Event-B elements specific to the pattern, and requires elements created by other patterns. Such interdependency and mapping to FMEA is schematically shown in Fig. 4.

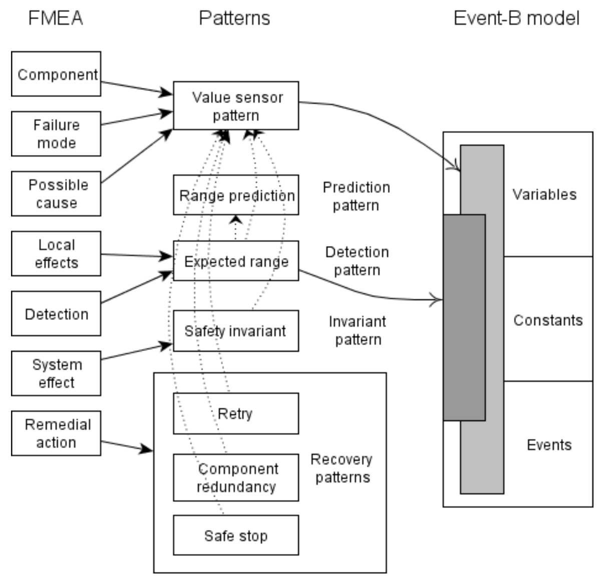

Figure 4. FMEA representation patterns

Let us note that the *Expected range pattern* creates new constants and variables (dark grey rectangle, variable [sensor]_exp_min from the example) and instantiates the *Value sensor pattern* to create the elements it depends on (light grey rectangle, variable [sensor]_value from the example).

B. Automation of Patterns Implementation

The automation of the pattern instantiation is implemented as a tool plug-in for the Rodin Platform [4]. Technically, each pattern is a program written in a simplified Eclipse Object Language (EOL). It is a general purpose programming language in the family of languages of the Epsilon framework [10] which operates on EMF [3] objects. It is a natural choice for automating model transformations since Event-B is interoperable with EMF.

The tool extends the application of EOL to Event-B models: it adds simple user interface features for instantiation, extends the Epsilon user input facility with discovery of the Event-B elements, and provides a library of Event-B and FMEA-specific transformations.

To apply a pattern, a user chooses a target model and a pattern to instantiate. A pattern application may require user input: variable names or types, references to existing elements of the model etc. The input is performed through a series of simple dialogs.

The requested input comprises the applicability conditions of the pattern. In many cases it is known that instantiation of a pattern depends primarily on the results of a more basic pattern. In those cases the former directly instantiates the latter and reuses the user input. Also more generally, if several patterns require the same unit of user input then the composition of such patterns will ask for such input only once. Typically, a single pattern instantiation requires up to 3-4 inputs.

If a pattern only requires user input and creates new elements then its imperative form is close to declarative as shown in the example below:

```
var flag: Variable=
chooseOrCreateVariable("Phase variable");
createTypingInvariant(flag, "PHASE");
var failure: Variable =
chooseOrCreateVariable("Failure variable");
createTypingInvariant(failure, "BOOL");
newEvent("Detection")
.addGuard("phase_grd", flag.name+" = DET")
.addGuard("failure_grd", failure.name+"=FALSE")
.addAction("phase_act", flag.name+":=CONT")
.addAction("failure_act",failure.name+"::BOOL");
```

Here the tool will ask the user to select two variables (or create new ones). It will create typing invariants and a new model event with several guards and actions. We have used the tool to automate the first refinement of the sluice gate control system. The complete specification can be found in [14].

C. Ensuring Safety by Refinement

In the second refinement step we introduce the detailed specification of the normal control logic. This refinement step leads to refining the event **Normal_Operation** into a group of events that model the actual control algorithm. These events model opening and closing the doors as well as activation of the pressure chamber pump.

Refinement of the normal control operation results in restricting non-determinism. This allows us to formulate safety invariants that our system guarantees:

failure = FALSE ∧ door1_position = door1_position ⇒ door1_position = 0

failure = FALSE ∧ (door1_position > 0 ∨ door1_motor=**MOTOR_OPEN**) ⇒ pressure_value = **PRESSURE_OUTSIDE**

failure = FALSE ∧ (door2_position > 0 ∨ door2_motor=**MOTOR_OPEN**) ⇒ pressure_value = **PRESSURE_INSIDE**

failure = FALSE ∧ pressure_value ≠ **PRESSURE_INSIDE** ∧ pressure_value ≠ **PRESSURE_OUTSIDE** ⇒ door1_position=0 ∧ door2_position=0

failure = FALSE ∧ pump≠**PUMP_OFF** ⇒ (door1_position=0 ∧ door2_position=0)

These invariants formally define the safety requirements informally described in subsection III.A. While verifying the correctness of this refinement step, we formally ensure (by proofs) that safety is preserved while the system is operational.

At the consequent refinement steps we introduce the error recovery procedures. This allows us to distinguish between criticality of failures and ensure that if a non-critical failure occurs then the system can still remain operational.

V. CONCLUSIONS

In this paper we have made two main technical contributions. Firstly, we derived a set of generic patterns for elicitation and structuring of safety and fault tolerance requirements from FMEA. Secondly, we created an automatic tool support that enables interactive pattern instantiation and automatic model transformation to capture these requirements in formal system development. Our methodology facilitates requirements elicitation as well as supports traceability of safety and fault tolerance requirements within the formal development process.

Our approach enables *guided* formal development process. It supports the reuse of knowledge obtained during formal system development and verification. For instance, while deriving the patterns we have analysed and generalised our previous work on specifying various control systems [8,11,12].

We believe that the proposed approach and tool support provide a valuable support for formal modelling that is traditionally perceived as too cumbersome for engineers. Firstly, we define a generic specification structure. Secondly, we automate specification of a large part of modelling decisions. We believe that our work can potentially enhance productivity of system development and improve completeness of formal models.

As a future work we are planning to create a library of domain-specific patterns and automate their application. This would result in achieving even greater degree of development automation and knowledge reuse.

ACKNOWLEDGMENT

The work reported in this paper is supported by FP7 ICT DEPLOY.

REFERENCES

[1] J.-R. Abrial, “Modeling in Event-B: system and software engineering”, Cambridge University Press, 2010.

[2] Deploy project, www.deploy-project.eu.

[3] Eclipse GMT – Generative Modeling Technology, http://www.eclipse.org/gmt.

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

[5] FMEA Info Centre, http://www.fmeainfocentre.com/.

[6] 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.

[7] I. Lopatkin, A. Iliasov, A. Romanovsky, “On Fault Tolerance Reuse during Refinement”. In Proc. Of the 2nd International Workshop on Software Engineering for Resilient Systems (SERENE), April 13-16, 2010.

[8] D. Ilic and E. Troubitsyna, “Formal development of software for tolerating transient faults”. In Proc. of the 11th IEEE Pacific Rim International Symposium on Dependable Computing, IEEE Computer Society, Changsha, China, December 2005.

[9] ClearSy, Safety critical systems engineering, http://www.clearsy.com/.

[10] D. S. Kolovos, “Extensible platform for specification of integrated languages for model management (Epsilon)”, Official web-site: http://www.cs.york.ac.uk/~dkolovos/epsilon.

[11] L. Laibinis and E. Troubitsyna, “Refinement of fault tolerant control systems in B”, SAFECOMP 2004, Springer, Potsdam, Germany, 2004.

[12] L. Laibinis and E. Troubitsyna, “Fault tolerance in a layered architecture: a general specification pattern in B”, In Proc. of International Conference on Software Engineering and Formal Methods SEFM’2004, IEEE Computer Society Press, pp.346-355, Beijing, China, September 2004.

[13] N.G. Leveson, “Safeware: system safety and computers”, Addison-Wesley, 1995.

[14] I. Lopatkin, Y. Prokhorova, E. Troubitsyna, A. Iliasov and A. Romanovsky, “Patterns for Representing FMEA in Formal Specification of Control Systems”, Technical Report 1003, TUCS, March 2011.

[15] 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.

[16] N. Storey, “Safety-critical computer systems”, Addison-Wesley, 1996.

[17] Thai Son Hoang, A, Furst and J.-R. Abrial, “Event-B patterns and their tool support”, SEFM 2009, IEEE Computer Press, 2009, pp. 210-219.

[18] E. Troubitsyna, “Elicitation and specification of safety requirements”, Proceedings of the Third International Conference on Systems (ICONS 2008), 2008, pp. 202-207.

[19] 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.