Deriving a mode logic using failure modes and effects analysis

Yuliya Prokhorova*, Linas Laibinis and Elena Troubitsyna

TUCS – Turku Centre for Computer Science,
Department of Information Technologies,
Åbo Akademi University,
Joukahaisenkatu 3-5A, 20520 Turku, Finland
E-mail: Yuliya.Prokhorova@abo.fi
E-mail: Linas.Laibinis@abo.fi
E-mail: Elena.Troubitsyna@abo.fi
*Corresponding author

Kimmo Varpaaniemi and Timo Latvala

Space Systems Finland,
Kappelitie 6 B, 02200 Espoo, Finland
E-mail: Kimmo.Varpaaniemi@ssf.fi
E-mail: Timo.Latvala@ssf.fi

Abstract: Modes are widely used to structure the behaviour of control systems. However, derivation and verification of a mode logic for complex systems is challenging due to a large number of modes and intricate mode transitions. In this paper, we propose an approach to deriving, formalising and verifying consistency of a mode logic for fault-tolerant control systems. We propose to use failure modes and effects analysis (FMEA) to systematically derive the fault tolerance part of the mode logic. We formalise the mode logic and define mode consistency properties for layered systems with reconfigurable components. We use our formalisation to develop and verify a mode-rich system by refinement in Event-B.

Keywords: Event-B; formal specification; fault tolerance; failure modes and effects analysis; FMEA; layered control systems; mode logic.

Reference to this paper should be made as follows: Prokhorova, Y., Laibinis, L., Troubitsyna, E., Varpaaniemi, K. and Latvala, T. (xxxx) 'Deriving a mode logic using failure modes and effects analysis', *Int. J. Critical Computer-Based Systems*, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Yuliya Prokhorova is a PhD student at Åbo Akademi University and TUCS – Turku Centre for Computer Science. She received her MSc in Computer Systems and Networks from National Aerospace University 'KhAI', Kharkiv, Ukraine in 2008. She is currently interested in formal modelling and verification of safety-critical and fault-tolerant systems.

Linas Laibinis is an Adjunct Professor at the Department of Information Technologies of Åbo Akademi University. He received his PhD in Computer Science in 2000 on mechanised formal reasoning about computer programmes. His research interests include interactive environments for proof and programme construction, as well as application of formal methods to modelling and development of fault-tolerant and distributed software systems.

Elena Troubitsyna is an Associate Professor at the Department of Information Technologies of Åbo Akademi University. She received her PhD in Computer Science in 2000 on design methods for dependable systems. Her research interests include application of formal methods to development of dependable fault-tolerant systems. She also conducts research on combining formal methods with informal techniques of safety analysis and semi-formal design techniques such as UML. She has worked on applying formal methods to development of an industrial fault-tolerant system within EU IST projects MATISSE, RODIN, and DEPLOY.

Kimmo Varpaaniemi is a formal methods expert. He has worked in technical positions, especially related to software verification and validation, at Space Systems Finland since 2006. Before that, he worked as a researcher in computer science with emphasis on formal methods by model checking, for several years at Helsinki University of Technology. He is also a docent in formal verification methods for parallel and distributed systems at Aalto University School of Science.

Timo Latvala is a Technical Director in Software Reliability at Space Systems Finland. He has worked in technical and management positions, especially related to software verification and validation, at Space Systems Finland since 2007. Before that, he worked as a researcher in computer science with emphasis on formal methods by model checking, for several years at Helsinki University of Technology and for one year at University of Illinois at Urbana-Champaign.

This paper is a revised and expanded version of a paper entitled ‘Deriving mode logic for fault-tolerant control systems’ presented at 5th Nordic Workshop on Dependability and Security (NODES 2011), Copenhagen, Denmark, 27–28 June 2011.

1 Introduction

Complexity poses major threat to dependability (Avizienis et al., 2004). To cope with complexity, control systems are often developed in a layered fashion, which provides the designers with a convenient mechanism for structuring a system behaviour according to the identified architectural layers. However, the dynamic part of the system behaviour is frequently defined in terms of operational modes – mutually exclusive sets of the system behaviour (Leveson et al., 1997). Therefore, it is important to formally study the principles of designing layered mode-rich systems.

While designing layered mode-rich systems, we should ensure mode consistency and guarantee that a *mode logic*, i.e., the system modes together with transitions between

them (Leveson et al., 1997), also caters to fault tolerance. In this paper, we propose to conduct failure modes and effects analysis (FMEA) of each operational mode to identify mode transitions required to implement fault tolerance. In our approach fault tolerance is achieved by two main means – transitions to degraded modes and system reconfiguration using redundant components. We investigate a complex interplay between these two main mechanisms within Event-B framework (Abrial, 2010).

While developing a system by refinement in Event-B, we start from an abstract specification and gradually introduce implementation details. Our development process starts from an analysis of each operational mode and defining the fault tolerance part of the mode logic. Then we create an abstract specification of the upper architectural layer and gradually unfold lower layers by refinement. Since refinement allows us to develop a system in a correct-by-construction fashion, stepwise unfolding of the architectural layers also guarantees preserving mode consistency between lower and upper layers. Such an approach is inspired by the works of Iliasov et al. (2010a, 2010b).

The paper is structured as follows. In Section 2, we review related works. In Section 3, we present general guidelines for deriving a mode logic of complex layered systems. Section 4 describes our formalisation of fault-tolerant mode-rich systems and their properties. In Section 5, we briefly overview the basic modelling concepts of the Event-B framework. In Section 6, we illustrate the proposed approach by an example – attitude and orbit control system (AOCS). Finally, in Section 7, we give concluding remarks as well as discuss future work.

2 Related work

There are several well-known problems associated with mode-rich systems including mode confusion and automation surprises (Buth, 2004; Miller and Potts, 1999; Rushby, 2002). These studies conducted retrospective analysis of mode-rich systems to spot the discrepancies between the actual system mode logic and the user mental picture of the mode logic. Most of the approaches have relied on model-checking (Buth, 2004; Heimdahl and Leveson, 1996; Rushby, 2002), while Butler (1996) and Miller and Potts (1999) are based on theorem proving in PVS. For example, to identify potential sources of mode confusion, Miller and Potts (1999) propose to proceed trough two complementary strategies. The first strategy aims at creating a clear, executable model of the system and simulating it. It uses this combination to review the behaviour of the system and the man-machine interface with the designers, users, and experts in human factors. The second strategy is to conduct mathematical analyses of the model by translating it into a formal specification suitable for analysis with automated tools.

Our approach focuses on designing fully automatic systems and ensuring their mode consistency. Unlike Heimdahl and Leveson (1996), in our approach we also emphasise the complex relationships between system fault tolerance and the mode logic. Our approach allows the developers to systematically design the system and formally check mode consistency within the same framework.

Kelly and Bartlett (2006) propose a method that adopts directed graphs (digraphs) to model fault propagation through a system with operating modes. This method allows for dealing with multiple faults and enabling real-time diagnosis. The approach is exemplified by a case study with a control loop of the following elements: a sensor, controller and controlled device. The authors also define a number of the system

scenarios that reflect a nominal and faulty behaviour of the system. To diagnose faults using digraphs, system sensor readings should be compared to those which are expected for a current operating mode of the system. If a deviation is detected, the back-tracing mechanism is enabled through the system digraph from the location of the detected deviation. Our work is different from these approaches. We rely on a well-known safety analysis technique FMEA to investigate the impact of faults on system modes. This allows us to systematically derive the part of the mode logic that caters to fault tolerance.

In our previous work (Laibinis and Troubitsyna, 2004), we have introduced a general formal specification approach to the development of dependable systems with layered architectures. The approach is based on using the exception handling mechanism on each layer of such systems. The exceptions that cannot be handled at a certain layer are propagated to the upper layers. As a result, error recovery has a hierarchical structure. A similar idea is explored in this paper as well. In addition, we considered a mode-based mechanism for structuring the system behaviour. This paper extends the work presented in Laibinis and Troubitsyna (2004) by incorporating FMEA into the process of designing layered mode-rich systems. Moreover, we consider an interplay between unit reconfiguration and mode consistency.

The work presented by Bozzano et al. (2010) introduces a methodology for the design of complex safety-critical systems. The approach is based on a formal specification language that is derived from AADL. It supports verification and validation activities such as consistency analysis, dynamic fault tree generation, FMEA table generation, and diagnosability analysis. Since AADL has a support for explicit definition of component modes, the work presented in this paper can also be undertaken within such a framework. However, verification by model checking might impose limitations on complexity of a mode logic to be verified.

Application of a symbolic model checking approach to verifying mode-rich satellite software was considered in Gan et al. (2011). The proposed approach involves the following steps: representation of a system behaviour in a form of extended state machine diagram with prioritised transitions, its translation to a set of linear temporal logic properties, and execution of the symbolic model checker NuSMV2. The authors emphasise that they do not use the refinement methodology to derive correct implementations. However, to avoid state explosion, they introduce an abstraction to a model. Moreover, the authors point out that the properties been checked are not strictly safety properties.

The work conducted by Javed and Troubitsyna (2012) aims at demonstrating how to ensure consistency of mode transitions logic implemented in SystemC code. In this work, the authors consider forward and backward mode transitions as well. The proposed system architecture in SystemC is verified using the SPIN model checker. For this purpose, the SystemC code is translated to Promela – an input language for SPIN. The authors highlight the fact that the verification by model checking was successful. However, they do not provide a reader with quantitative evaluation of the results.

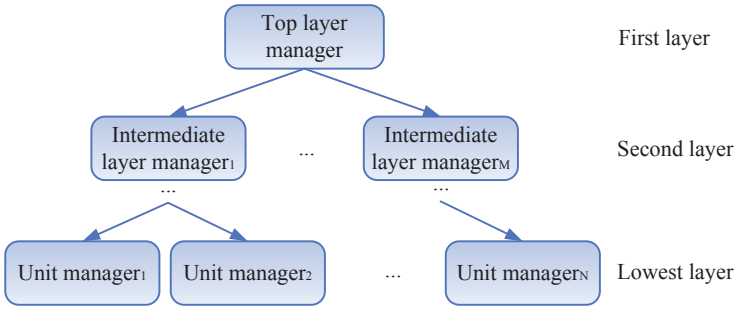
This paper generalises the results presented in Prokhorova et al. (2011a, 2011b), Laibinis and Troubitsyna (2004) and Javed and Troubitsyna (2012) by defining rigorous theory of designing layered mode-rich systems by refinement.

3 Deriving a mode logic

3.1 Layered mode-rich systems

Often a layered architecture is used to facilitate development of complex control systems (Rubel, 1995). It allows the designers to structure the system behaviour according to the identified abstraction levels. The lowest layer usually consists of the components (often called *units*) that work directly with hardware devices. The layer above contains the components encapsulating the lowest layer units by providing abstract interfaces to them. Depending on the system complexity and design decisions, there might be several intermediate layers. Finally, the top component provides an interface to the overall system (Figure 1).

Figure 1 Architecture of a layered system (see online version for colours)



Operational modes structure the dynamic behaviour of components at different layers of abstraction. Leveson et al. (1997) define an *(operational) mode* as a mutually exclusive set of system behaviours. A *mode logic* includes all the available modes and the rules for transitioning between them (Leveson et al., 1997). System complexity makes derivation of the mode logic and verification of its consistency challenging. Therefore, there is a clear need for the techniques that allow us to design mode-rich systems in a systematic and disciplined way.

Each software component in layered mode-rich systems can be viewed as a *mode manager (MM)*. For simplicity, let us consider a two-layer system. It consists of the lower layer MMs called *unit managers* (*UMs*), which are monitored and controlled by the top layer MM.

We identify system modes according to the system operational constraints. We assume that the system executes a certain *scenario* defined in terms of its global modes. The scenario usually describes a sequence of modes leading to the most advanced state of the system (e.g., the state where it delivers the richest set of services). However, occurrence of failures may prevent the system from straightforward implementation of the scenario. This requires an introduction of the fault tolerance mechanisms into the system design. They are implemented as backward mode transitions (rollbacks) within the predefined scenario.

The dynamic behaviour of the overall system is cyclic. At each cycle MM monitors the current state of the lower layer units. If units are fault free then MM either completes the ongoing mode transition or maintains the current mode. It might also initiate the forward transition according to the scenario. If some failure that cannot be locally handled by UMs occurs, MM starts a backward mode transition.

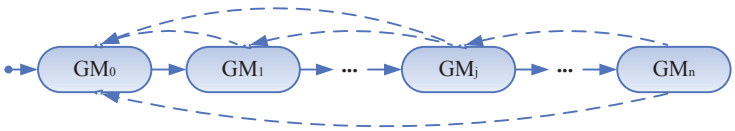
An execution of the mode scenario by MM corresponds to certain mode transitions initiated by UMs. Mode consistency conditions are defined as interdependencies between global and local modes. Specifically, each global mode of MM may correspond to one or several combinations of the local modes of UMs. While the nominal part of a mode logic is usually well understood and given, the designers need to derive the fault tolerance part of the mode logic, i.e., the set of backward transitions to execute error recovery.

In the next section, we demonstrate how to define the fault tolerance part of the mode logic in a well-structured way using FMEA as well as to introduce fault tolerance mechanisms into the mode logic.

3.2 FMEA in a mode logic derivation

Let us assume that the scenario (shown in Figure 2 by solid arrows) defines how to bring a system from the non-operational mode GM_0 to the most advanced mode GM_n. Such a forward mode transition scenario is usually given in the system requirements document. Moreover, let us assume that the mode transition scenario can be interrupted

Figure 2 Mode transition scenario (see online version for colours)



either by transitional errors (i.e., errors that appear during a mode transition step) or unit usability errors (i.e., errors that occur when a unit performs below its required level).

In this paper, we consider units, i.e., the lowest layer components, to be dynamically reconfigurable. They consist of a main device (the nominal branch) and a spare device (the redundant branch). A unit encapsulates the errors and the reconfiguration state of its branches. In particular, if an error occurs in the nominal branch of the unit, it handles this error by switching to the redundant branch. The units have a special attribute called *status*. The unit status is *locked*, when it is in an operational state and there is no ongoing reconfiguration, and *unlocked* otherwise.

When MM chooses a new target mode, it initiates the corresponding mode transitions in the lower layer UMs. If an error is detected, the corresponding UM assesses the error and either initiates error recovery by itself or propagates the error to MM. MM, in its turn, makes a decision how to handle such an error. This decision usually involves rolling back to some less advanced (i.e., degraded) mode, as shown by dashed arrows in Figure 2.

To systematically define the rollback procedures for each mode, we propose to conduct FMEA (FMEA IC, 2011; Leveson, 1995; Storey, 1996). FMEA is a well-known inductive safety analysis technique. For each system function or component, it defines

possible failure modes, local and system effects, as well as detection and recovery procedures. The information is collected in a table form.

There are several types of FMEA, among them:

- System FMEA (SFMEA) – used to analyse complete systems and/or subsystems. It focuses on global functions of the system but not just individual failure modes and their direct impact.
- Design FMEA (DFMEA) – used to examine the functions of a component, subsystem or main system separately. It focuses on potential failure modes of products caused by design faults.
- Process FMEA (PFMEA) – used to analyse manufacturing and assembly processes at the system, subsystem or component levels.
- Software FMEA (SWFMEA) – used to analyse a system with respect to software faults which may cause failure events.

The traditional DFMEA allows us to discover and structure failure modes of components. One of possible interpretations of a FMEA worksheet with explanation of its fields is given in Figure 3. In this paper, we propose to conduct FMEA of each

Figure 3 FMEA worksheet

Global mode	**Name of a component**
Failure mode	Potential failure modes
Possible cause	The most probable causes associated with the assumed failure mode
Local effects	Caused changes in the component behaviour
System effects	Caused changes in the system behaviour
Detection	A description of the methods by which occurrence of the failure mode is detected
Remedial action	Actions to tolerate the failure

operational mode. We tailor DFMEA to fit our purposes. In particular, we introduce an additional structure into the description of remedial actions. Namely, the remedial action field now contains three new subfields ‘target mode’, ‘precondition’, and ‘action’, describing respectively a new target mode, the conditions when the rollback to this mode should be initiated, and the system remedial actions in terms of new local mode transitions and, if needed, reconfiguration actions. If there are several preconditions given for the same target mode, the respective conditions should describe mutually exclusive situations.

For instance, Figure 4 shows two excerpts from a FMEA worksheet of the operational mode GM_j. The excerpts cover the situations when a hardware failure occurs in the unit U_i with and without an available spare, respectively. Here GM_t is some degraded global mode. U_k stands for a unit which is supposed to be in an operational mode in the global mode GM_t.

FMEA of each system mode allows us to obtain a systematic textual description of all the procedures required to detect unit errors and perform their recovery. The main procedure of rolling back is as follows.

If a single unit fails then MM sets the new target mode to a degraded, however as advanced as possible, mode where the failed unit is not used, i.e., it is in a

non-operational (*Off*) local mode. If several units fail then MM puts the system into a global mode where all failed units are in the *Off* modes.

Figure 4 Modified FMEA worksheets

Mode	**GM_j**		
Failure mode	Unit U_i failure with an available spare		
Possible cause	Hardware failure		
Local effects	Reconfiguration between unit branches. Change of unit status		
System effects	Remain the current global mode		
Detection	Comparison of received data with the predicted one		
Remedial action	**Target mode**	**Precondition**	**Action**
	GM_j	A state transition error in the nominal branch of U_i.	For a nominal branch of unit U_i, the status is set to Unlocked, and reconfiguration between branches is initiated.
		Insufficient usability of a selected nominal branch of U_i.	

Mode	**GM_j**		
Failure mode	Unit U_i failure without an available spare		
Possible cause	Hardware failure		
Local effects	Loss of preciseness in unit output data. Change of unit status		
System effects	Switch to a degraded mode		
Detection	Comparison of received data with the predicted one		
Remedial action	**Target mode**	**Precondition**	**Action**
	GM_t, $t < j$	A state transition error in the redundant branch of U_i. No state transition error in the redundant branch of U_k.	For unit U_i, any ongoing unit reconfiguration is aborted. For each branch in unit U_i, the status is set to Unlocked, and a state transition to non-operational state is initiated.
		Insufficient usability of a selected redundant branch of U_i. No branch state transition error. No problem on the redundant branch of U_k.	

In this section, we have demonstrated how to derive the fault tolerance part of the mode logic. Essentially, the process of deriving the mode logic consists of the following generic steps:

1 Conduct FMEA for each operational (global) mode of a system.

2 Analyse failures of units that are operational in the current global mode.

3 Consider cases when a spare (redundant branch) of the failed unit is available and when it is not.

4 Define remedial actions (actions to tolerate a failure):

   a if the redundant brunch is available, define the failed unit reconfiguration procedure

   b if the redundant brunch is not available, define a backward mode transition (rollback) to a degraded mode. In the chosen degraded mode the failed unit should be non-operational.

5 While performing the rollback, take into account failures of other units that are supposed to be in operational modes in the target global mode.

It is easy to note that, even for a simple system, backward mode transitions significantly complicate the mode logic. Therefore, there is a clear need for techniques that facilitate not only derivation of the mode logic but also verification of its correctness. In the next section, we demonstrate how to formalise the essential notions and properties of the mode logic.

4 Formalising a mode logic

Our formalisation aims at defining interdependencies between global modes, unit modes and their status as well as verifying mode consistency between different system layers.

We introduce a set of all possible modes, *MODES*. A relation on *MODES*, called *Scenario*, formally defines forward mode transitions:

$$Scenario \in MODES \leftrightarrow MODES \tag{1}$$

where $\leftrightarrow$ stands for a relation between elements of two types. All the possible forward transitions are represented by the relation *Mode_Order*:

$$Mode\_Order = Scenario^{*}$$

where $*$ is the transitive closure operation. Essentially, *Mode_Order* is a partial order relation on modes. While *Scenario* formally represents the nominal part of a mode logic, the inverse relation *Mode_Order*$^{\sim}$ defines all possible rollbacks.

The mode to which the system should rollback to execute error recovery can be defined as a function *Error_Handling*:

$$Error\_Handling : MODES \times UNIT\_ERRORS \rightarrow MODES \tag{2}$$

where the first parameter is the current global mode of the system and the second parameter stands for all the detected errors of units. For a mode transition to be completed, certain mode entry conditions should be satisfied. To formally define this, we introduce a function *Mode_entry_cond*:

$$Mode\_entry\_cond :$$
$$MODES \rightarrow \mathcal{P}(UNIT\_MODES_{11} \times \cdots \times UNIT\_MODES_{jm})$$

where $\mathcal{P}(T)$ is a power set on elements of T and $UNIT\_MODES_{11} \ldots UNIT\_MODES_{jm}$ are the local modes $1 \ldots j$ of the monitored units $1 \ldots m$. For each global mode, the function returns a set of all valid combinations of unit modes.

The global mode transition cannot be started or completed if there is an ongoing reconfiguration of any unit in the system. To model this, we define an attribute *In_Reconfig*:

$$In\_Reconfig = Reconfig(U_1) \vee \cdots \vee Reconfig(U_m)$$

where $Reconfig(U_1)\ \ldots\ Reconfig(U_m)$ are the corresponding reconfiguration flags of the units $1 \ldots m$. If there is no ongoing reconfiguration in the system, the flag *In_Reconfig* is *FALSE*, otherwise it is *TRUE*.

Reconfiguration is enabled only if an error of the nominal branch of a unit is detected. To reflect this, for each unit i, we introduce a function *Unit$_i$_reconf*:

$$Unit_i\_reconf : UNIT_i\_ERRORS \times BRANCH \rightarrow BRANCH$$

defined by its properties

$$\forall x \cdot Unit_i\_reconf(x \mapsto Branch\_A) = Branch\_B$$
$$\forall x \cdot Unit_i\_reconf(x \mapsto Branch\_B) = Branch\_B$$

where *UNIT$_i$_ERRORS* represents the errors of a particular unit *i*. Here *Branch_A* is the nominal branch and *Branch_B* is the redundant branch. The decision to start dynamic reconfiguration between branches of a unit *i* is defined by a function *Unit$_i$_reconf_need:*

$$Unit_i\_reconf\_need : UNIT_i\_ERRORS \times BRANCH \rightarrow BOOL \tag{3}$$

In the real system, mode transitions may take time and can be interrupted by errors. To unambiguously describe the actual state of a mode managing component *C* at any layer (i.e., the top layered MM and all the lower layered UMs), we define the following attributes for each of them:

1 *prev_target$_C$* is the previous mode that a component was in transition to

2 *last_mode$_C$* is the last successfully reached mode

3 *next_target$_C$* is the target mode that a component is currently in transition to.

Based on these attributes, we define the notion of the mode managing component state that might be either stable, decreasing or increasing as follows:

- *Stable$_C$:* a component *C* is maintaining the last successfully reached mode, i.e., *last_mode$_C$* = *prev_target$_C$* $\wedge$ *next_target$_C$* = *prev_target$_C$*
- *Increasing$_C$:* a component *C* is in transition to a next, more advanced mode, i.e., *last_mode$_C$* = *prev_target$_C$* $\wedge$ *prev_target$_C$* $<$ *next_target$_C$*
- *Decreasing$_C$:* component *C* stability or a mode transition to the previous target was interrupted (e.g., by error handling) by a new mode request to a more degraded mode, i.e., *next_target$_C$* $<$ *prev_target$_C$*.

When a mode transition is completed, the component state becomes stable. Furthermore, when the top layered component *MM* is in a stable state, all the lower layer components, i.e., UMs, should be stable as well. This can be formulated as an invariant property of the system:

$$Stable_{MM} \Rightarrow Stable_{UM_1} \wedge \cdots \wedge Stable_{UM_m}$$

Another main invariant property of the system states that during reconfiguration a component cannot be stable:

$$\forall C \cdot In\_Reconf(C) = TRUE \;\Rightarrow\; \neg Stable_C$$

Moreover, each unit in the system has not only its local mode but also the status (*locked* or *unlocked*). The correspondence between these two notions can be formalised as the following predicate:

$$(Reconfig(UM_i) = TRUE \vee last\_mode_{UM_i} = Off \vee next\_target_{UM_i} = Off) \Rightarrow Status_{UM_i} = Unlocked \quad (4)$$

or, equivalently,

$$Status_{UM_i} = Locked \;\Rightarrow (Reconfig(UM_i) = FALSE \wedge last\_mode_{UM_i} \neq Off \wedge next\_target_{UM_i} \neq Off) \quad (5)$$

Such a property follows from the definition of the unit status: each unit can have the status *Locked* only if there is no ongoing reconfiguration in this unit and the unit is neither in the non-operational mode *Off*, nor in the transition from or to the mode *Off*.

The discussion above sets general guidelines for defining different layer MMs and reconfiguration procedures. In Sections 5 and 6, we will show how these guidelines can be implemented in a formal specification in Event-B.

5 Modelling in Event-B

5.1 Event-B overview

Event-B (2011) is a state-based formal method for system level modelling and analysis (Abrial, 2010). It is an extension of the B method (Abrial, 1996), which has been successfully used in the development of several complex real-life applications (ClearSy, 2011). Event-B aims at facilitating modelling of parallel, distributed and reactive systems. Automated support for modelling and verification in Event-B (2011) is provided by the Rodin platform.

In Event-B system models are defined using the notion of an *abstract state machine*. An abstract machine encapsulates the state (the variables) of a model and defines operations (events) on its state. The machine is uniquely identified by its name *MachineName*. The state variables of the machine are declared in the *Variables* clause and initialised in the *INITIALISATION* event. The variables are strongly typed by the 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 data types and constants of the model are defined in a separate component called *CONTEXT*, where their properties are postulated as axioms. The behaviour of the system is determined by a number of atomic *Events*. An event can be defined as follows:

$evt \;\hat{=}$ **any** lv **where** g **then** R **end**

where *lv* is a list of local variables, the guard *g* is a conjunction of predicates defined over model variables, and the action *R* is a composition of assignments on the variables executed simultaneously.

The guard defines when an 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 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 semantics of Event-B events is defined using before-after predicates (Metayer et al., 2005). 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 initialisation 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 each event evt_i of the model, its feasibility means that, whenever the event is enabled, its before-after (BA) predicate is well-defined, i.e., there exists some reachable after-state:

$$A(d, c),\ I(d, c, v),\ g_i(d, c, v)\ \vdash\ \exists v' \cdot BA_i(d, c, v, v') \qquad \text{(FIS)}$$

where A stands for the conjunction of the axioms, I is the conjunction of the invariants, g_i is the guard of the event evt_i, BA_i is the before-after predicate of this event, d stands for the sets, c are the constants, and v, v' are the variable values before and after event execution.

Each event evt_i of the Event-B model should also preserve the given model invariant:

$$A(d, c),\ I(d, c, v),\ g_i(d, c, v),\ BA_i(d, c, v, v')\ \vdash\ I(d, c, v') \qquad \text{(INV)}$$

Since the initialisation event has no initial state and guard, its proof obligation is simpler:

$$A(d, c),\ BA_{Init}(d, c, v')\ \vdash\ I(d, c, v') \qquad \text{(INIT)}$$

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. Refinement leads to reducing non-determinism in an abstract model as well as introducing new variables and events. The connection between the newly introduced variables and the abstract variables 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.

5.2 Refinement in Event-B

Event-B employs a top-down refinement-based approach to formal system development. Development starts from an abstract system specification that models some of essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into an abstract specification. These new events correspond to stuttering steps that are not visible in the abstract specification. We call such model refinement as *superposition refinement*. Moreover, Event-B formal development supports data refinement, allowing us to replace some abstract variables with their concrete counterparts. In that case, the invariant of a refined model formally defines the relationship between the abstract and concrete variables; this type of invariants is called a *gluing invariant*.

To verify correctness of a refinement step, we need to prove a number of *proof obligations* for a refined model. For brevity, here we show only a few essential ones. The full list of proof obligations can be found in Abrial (2010).

Let us introduce a shorthand $H(d, c, v, w)$ that stands for the hypotheses $A(d, c)$, $I(d, c, v)$ and $I'(d, c, v, w)$, where I and I' are respectively the abstract and refined invariants and v, w are respectively the abstract and concrete variables. Then the feasibility refinement property for an event evt_i of a refined model can be presented as follows:

$$H(d, c, v, w),\ g_i'(d, c, w) \ \vdash\ \exists w'.\ BA_i'(d, c, w, w') \qquad \text{(REF\_FIS)}$$

where g_i' is the refined guard, BA_i' is a before-after predicate of the refined event, w and w' are the concrete variable values before and after the refined event execution.

The invariant preservation proof obligation for a refined model is the following:

$$H(d, c, v, w),\ g_i'(d, c, w),\ BA_i'(d, c, w, w') \ \vdash\ I'(d, c, v', w') \qquad \text{(REF\_INV)}$$

The event guards in a refined model can be only strengthened in a refinement step:

$$H(d, c, v, w),\ g_i'(d, c, w) \ \vdash\ g_i(d, c, v) \qquad \text{(REF\_GRD)}$$

where g_i, g_i' are respectively the abstract and concrete guards of the event evt_i.

The *simulation* proof obligation (REF_SIM) requires to show that the action (i.e., assignment on the state variables) of a refined event is not contradictory to its abstract version:

$$H(d, c, v, w),\ g_i'(d, c, w),\ BA_i'(d, c, w, w') \ \vdash\ \exists v'.BA_i(d, c, v, v') \ \wedge\ I'(d, c, v', w') \qquad \text{(REF\_SIM)}$$

where BA_i, BA_i' are respectively the abstract and concrete before-after predicates of the same event evt_i.

The consistency (invariant preservation) and well-definedness of Event-B models as well as correctness of refinement steps are demonstrated by discharging the described proof obligations. The Rodin platform (Event-B, 2011), a tool supporting Event-B, automatically generates these 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 80%) in proving.

6 Case study – AOCS

The AOCS is a typical layered control system. The main function of the system is to control the attitude and the orbit of a satellite. Since the orientation of a satellite may change due to disturbances of the environment, the attitude needs to be continuously monitored and adjusted. The optimal attitude is required to support the needs of payload instruments (PLIs) and to fulfil the mission of the satellite (DEPLOY, 2010).

At the top layer of AOCS is a *MM*. It controls several *UMs*, which are responsible for a number of hardware units. AOCS has seven units – four sensors [star tracker (STR), sun sensor (SS), earth sensor (ES) and global positioning system (GPS)], two actuators [reaction wheel (RW) and thruster (THR)], and one PLI producing mission measurements. The predefined mode scenario determines the sequence of steps needed to reach the state where PLI is ready to perform its tasks. This sequence includes the following modes:

- *OFF* – the satellite is typically in this mode right after system (re)booting.
- *Standby* – this mode is maintained until separation from the launcher.
- *Safe* – a stable attitude is acquired, which allows the coarse pointing control.
- *Nominal* – the satellite is trying to reach the fine pointing control, which is needed to use the PLI.
- *Preparation* – the PLI is getting ready.
- *Science* – the PLI is ready to perform its tasks. The mission goal is to reach this mode and stay in it as long as needed.

In this paper, we consider a formal derivation of the AOCS mode logic focusing only on ES and GPS. ES is a typical representative of AOCS units with two simple local modes: non-operational, *Off*, and operational, *On*. GPS represents the units with one non-operational mode, *Off,* and two operational modes, *Coarse* and *Fine*. When the global mode is *Safe*, ES is in the operational mode *On*, otherwise it is in the mode *Off.* Similarly, GPS is required to be in the mode *Coarse*, when the global mode is *Nominal*, and in the operational mode *Fine* when the global mode is either *Preparation* or *Science*. It should be in the mode *Off* otherwise (see Table 1).

Table 1 Correspondence between unit modes and global modes of AOCS

Mode *Unit*	*OFF*	*Standby*	*Safe*	*Nominal*	*Preparation*	*Science*
ES	Off	Off	On	Off	Off	Off
SS	Off	Off	On	Off	Off	Off
GPS	Off	Off	Off	Coarse	Fine	Fine
STR	Off	Off	Off	On	On	On
RW	Off	Off	On	On	On	On
THR	Off	Off	Off	On	On	On
PLI	Off	Off	Off	Off	Standby	Science

To identify rollbacks according to the method proposed in Section 3, we have conducted FMEA for all the global modes where either ES or GPS is in an operational mode. We take into account other AOCS units that could influence the rollback. For brevity, in Figure 5 we show an excerpt from FMEA worksheet. It corresponds to the global mode *Nominal* with the failure mode *GPS failure without an available spare*.

Figure 5 FMEA of the AOCS global mode nominal

Mode	**Nominal**		
Failure mode	GPS failure without an available spare		
Possible cause	Hardware failure		
Local effects	Sensor reading is out of expected range. Change of GPS status		
System effects	Switch to a degraded mode		
Detection	Comparison of received data with the predicted one		
Remedial action	**Target mode**	**Precondition**	**Action**
	Safe	A state transition error in some of the redundant branches in GPS, STR and THR. No state transition error in the redundant branch of RW.	For each unit other than RW, any ongoing unit reconfiguration is aborted. For each branch in each unit other than RW, the status is set to Unlocked, and a state transition to Off is initiated.
		Insufficient usability of a selected redundant branch of GPS, STR or THR. No branch state transition error. No problem on the redundant branch of RW.	
	OFF	A state transition error in the redundant branch of RW.	For each unit, any ongoing unit reconfiguration is aborted. For each branch in each unit the status is set to Unlocked, and a state transition to Off is initiated.
		Insufficient usability of a selected redundant branch of RW. No branch state transition error.	

By relying on the FMEA results, we define rollback rules for the ES and GPS units. Specifically, if the system is in the mode *Nominal* and both branches of GPS failed, the system should perform a rollback to the mode *Safe*. On the other hand, if the system is in the mode *Safe* and both branches of ES failed, the system should perform a rollback to the mode *OFF*.

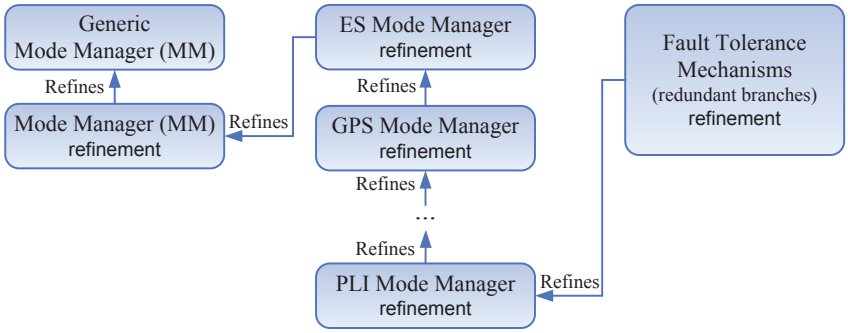
To illustrate how FMEA is performed for other AOCS hardware units, let us consider the most critical unit of the system – its PLI. The corresponding FMEA worksheet is shown in Figure 6. Here the system is in the mode *Science* and the nominal branch of the PLI unit failed (i.e., the failure mode *PLI failure with an available spare*). To tolerate the failure, the system performs a rollback to the mode *Preparation*. If during the transition phase an error in the redundant branch of PLI occurred the next target mode must be set to *Nominal*. If any error occurred in other units that should be in operational states in the target mode, the transition either to *Safe* or *OFF* must be initiated.

Figure 6 FMEA of the AOCS global mode science

Mode	**Science**		
Failure mode	PLI failure with an available spare		
Possible cause	Hardware failure		
Local effects	Sensor reading is out of expected range. Change of PLI status		
System effects	Switch to a degraded mode		
Detection	Comparison of received data with the predicted one		
Remedial action	**Target mode**	**Precondition**	**Action**
	Preparation	No branch state transition error. No ongoing unit reconfiguration.	For the nominal branch of PLI, the state is set to Off, for the redundant branch a state transition to Standby is initiated.
	Nominal	A state transition error in the redundant branch of PLI. No state transition error in the redundant branches of GPS, RW, STR and THR.	Any ongoing unit reconfiguration in PLI is aborted. For each branch in PLI the status is set to Unlocked, a state transition to Off is initiated.
		Insufficient usability of a selected redundant branch of PLI. No branch state transition error. No problem on the redundant branches of GPS, RW, STR and THR.	
	Safe	A state transition error in some of the redundant branches in GPS, STR and THR. No state transition error in the redundant branch of RW.	For each unit other than RW, any ongoing unit reconfiguration is aborted. For each branch in each unit other than RW, the status is set to Unlocked, and a state transition to Off is initiated.
		Insufficient usability of a selected redundant branch of GPS, STR or THR. No branch state transition error. No problem on the redundant branch of RW.	
	OFF	A state transition error in the redundant branch of RW.	For each unit, any ongoing unit reconfiguration is aborted. For each branch in each unit the status is set to Unlocked, and a state transition to Off is initiated.
		Insufficient usability of a selected redundant branch of RW. No branch state transition error.	

Refinement strategy. We use refinement process to incrementally build the system architecture, i.e., gradually unfold system layers. The refinement process starts from developing the top level MM that implements the global mode logic. The MM itself can be developed in a stepwise manner by several refinements. Once the MM is specified in sufficient details, we start modelling the lower layer managers together with their mode logics. Each UM and its logic can be introduced at separate refinement steps. At the last refinement step, we introduce modelling of the redundant branches of each unit (Figure 7).

Figure 7 Refinement strategy (see online version for colours)



The abstract specification models a generic MM. The variables representing the state of an abstract MM (*prev_targ*, *last_mode* and *next_targ*) and the detected errors (the variable *error*) are introduced at this step. The event *Run_Mode_Scenario* models autonomous scenario execution of the abstract MM and updates the variable *next_targ*. The events *Error_Handling* and *Error_Reset* describe different error handling procedures (with and without rolling back) of the MM, while the event *Error_Occurrence* abstractly models the occurrence of failures. The variable *error* is updated by all these events, while the variables representing the state of the MM are only changed by the event *Error_Handling*. The event *Advance_Partial* models partial reaching of the target mode, i.e., after the execution of this event the MM state is still *Increasing*. On the other hand, the event *Mode_Reached* implements complete achievement of the target mode, i.e., after the execution of this event the MM state is *Stable*. The AOCS abstract specification is shown in Figure 8.

The first refinement is an implementation of the MM and its mode logic as a specialisation of the abstract model. According to the formalisation guidelines proposed in Section 4, we define the allowed AOCS mode transitions (Figure 9). We represent them formally by instantiating the relation *Scenario* (1):

$$Scenario = \{OFF \mapsto Standby,\ Standby \mapsto Safe,\ Safe \mapsto Nominal,\\ Nominal \mapsto Preparation,\ Preparation \mapsto Science\}$$

The execution of the scenario is modelled as a sequence of events. A more detailed description is given in Iliasov et al. (2010b).

To model rollbacks shown in Figure 9, we introduce a function *MM_Error_Handling* that is an instantiation of the function *Error_Handling* (2) given in Section 4.

The second refinement step introduces one of the considered units, ES. The next target mode for ES can be set only by a higher level manager, in our case, by the MM. The scenario relation for ES is very simple:

$$ES\_scenario = \{ES\_Off \mapsto ES\_On\}$$

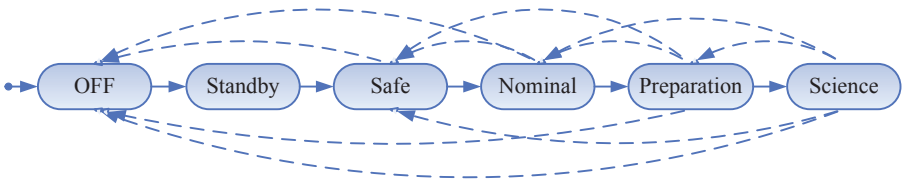
The correspondence relation between the MM and ES modes (shown in Table I) is formalised as follows:

$$ES\_mode = \{OFF \mapsto ES\_Off,\ Standby \mapsto ES\_Off,\ Safe \mapsto ES\_On,\\ Nominal \mapsto ES\_Off,\ Preparation \mapsto ES\_Off,\ Science \mapsto ES\_Off\}$$

Figure 8 Abstract specification

```
machine MM_Abs_M sees MM_Abs_C
variables last_mode prev_targ next_targ error
invariants
  last_mode ∈ MODES
  prev_targ ∈ MODES
  next_targ ∈ MODES
  error ∈ ERRORS
events
  event INITIALISATION
    then
//initialisation of variables
  end
  event Run_Mode_Scenario
// autonomous scenario of MM
    any m
    where
      m ∈ MODES
      next_targ = prev_targ
      error = No_Error
      prev_targ ↦ m ∈ Mode_Order
    then
      next_targ ≔ m
  end
  event Error_Handling
// error handling by MM
    any m
    where
      m ∈ MODES
      error ≠ No_Error
      prev_targ ↦ m ∈ Mode_Order~
    then
      next_targ ≔ m
      error ≔ No_Error
      prev_targ ≔ next_targ
      last_mode ≔ next_targ
  end
  event Normal_Operation
// successful execution cycle
    where
      next_targ = prev_targ
      error = No_Error
  end
  event Failure_Operation
// failed execution cycle
    where
      next_targ = prev_targ
      error = No_Error
    then
      error :| error' ≠ No_Error
  end
  event Error_Occurrence
// non-deterministic error occurrence
    then
      error :| error' ≠ No_Error
  end
  event Error_Reset
// error reset
    then
      error ≔ No_Error
  end
  event Advance_Partial
// mode is advanced but target mode is not
reached yet
    any m
    where
      prev_targ ≠ next_targ
      m ∈ MODES
      m ↦ next_targ ∈ Mode_Order
                        ∪ Mode_Order~
      m ≠ next_targ
      error = No_Error
    then
      last_mode ≔ m
  end
  event Advance_Complete
// target mode is reached
    where
      next_targ ≠ prev_targ
      error = No_Error
    then
      last_mode ≔ next_targ
      prev_targ ≔ next_targ
  end
end
```

Figure 9 AOCS mode transitions (see online version for colours)



Essentially, the *ES_mode* relation expresses the required consistency conditions between the modes of MM and ES. It is also a concrete projection of the generic function *Mode_entry_cond* introduced in Section 4.

Figure 10 ES mode transition events

event ES_Advance_Partial // mode is advanced but target mode is not reached yet
  any p
  where
    ES_prev_targ ≠ ES_next_targ
    p ∈ **ES_MODES**
    p ↦ ES_next_targ ∈ **ES_graph**
    p ≠ ES_next_targ
    last_mode ↦ p ∈ **ES_mode**
    flag_ES_rec = FALSE
  then
    ES_last_mode ≔ p
    ES_Status ≔ **fun_ES_status**
      (ES_Reconfig ↦ p ↦ ES_next_targ)
end

event ES_Mode_Reached // mode is reached
  where
    ES_next_targ ≠ ES_prev_targ
    last_mode ↦ ES_last_mode ∈ **ES_mode**
    last_mode ↦ ES_next_targ ∈ **ES_mode**
    last_mode ↦ ES_prev_targ ∈ **ES_mode**
    flag_ES_rec = FALSE
  then
    ES_last_mode ≔ ES_next_targ
    ES_prev_targ ≔ ES_next_targ
    ES_Reconfig ≔ FALSE
    ES_Status ≔ **fun_ES_status**
      (FALSE ↦ ES_next_targ ↦
       ES_next_targ)
end

In our Event-B model, we describe partial and complete reaching of a new target mode for ES by the respective events *ES_Advance_Partial* and *ES_Mode_Reached* (Figure 10). Here the function *fun_ES_status* is used to determine the ES status (*Locked* or *Unlocked*). The function result depends on the given values of the variables *ES_Reconfig*, *ES_last_mode* and *ES_next_targ*. Essentially, the function *fun_ES_status* is defined in such a way that its result (the ES status) always satisfies the following invariant properties of our model:

$$ES\_Reconfig = TRUE \lor ES\_last\_mode = ES\_Off \lor ES\_next\_targ = ES\_Off \Rightarrow ES\_Status = Unlocked \quad (6)$$

$$ES\_Status = Locked \Rightarrow ES\_Reconfig = FALSE \land ES\_last\_mode \neq ES\_Off \land ES\_next\_targ \neq ES\_Off \quad (7)$$

The first invariant (6) shows what conditions must be true for the system to have status *Unlocked*. The second property (7) follows from the first one and specifies which conditions are true when the ES status is *Locked*. Please note that these invariants are concrete instantiations of the general properties (4) and (5) given in Section 4.

The most challenging proofs were carried to demonstrate invariant preservation and feasibility – (REF_INV) and (REF_FIS), correspondingly. We had to interactively prove that the event *ES_Advance_Partial* preserves the invariant property (6). Moreover, we interactively proved the feasibility refinement property for the event *Failure_Operation1*. The event models failure operation of the system and refines the abstract event *Failure_Operation* by replacing the non-deterministic assignment of the variable *error* with possible errors of the ES unit (*action* such that *error* :| *error'* ∈ *ES_Any_Error*).

The third refinement step implements the GPS unit and its mode logic in a similar way. At the last refinement step, we introduce the reconfiguration procedures of ES and GPS units according to the remedial actions described in the FMEA worksheets.

To decide whether reconfiguration is needed, we introduce a function *fun_ES_rec_need_b* for the ES unit and a similar function *fun_GPS_rec_need_b* for the GPS unit. These functions are concrete instances of the abstract function $Unit_i\_reconf\_need$ (3). The functions return *TRUE*, if reconfiguration is needed, and *FALSE* otherwise.

We classify unit errors as *reconfigurable*, i.e., when a unit can recover from an error by its reconfiguration, and *non-reconfigurable*, i.e., when an error cannot be handled by reconfiguration even if the redundant branch is available. The required properties of the above functions are given as axioms in the accompanying CONTEXT model. The *Axiom 1* given below states that, if any reconfigurable error of ES occurs and the current branch is the nominal branch *Branch_A*, then reconfiguration has to be started. *Axiom 2* defines the situation when a reconfigurable error occurs but the current branch is already the redundant branch *Branch_B*. Therefore, unit reconfiguration is not possible and thus is not needed. Finally, *Axiom 3* specifies the case when reconfiguration is not possible due to a non-reconfigurable error.

$$\textit{Axiom 1: } \forall x \cdot x \in \textit{ES\_Reconf\_errors} \Rightarrow \textit{fun\_ES\_rec\_need\_b}(x \mapsto \textit{Branch\_A}) = \textit{TRUE}$$

$$\textit{Axiom 2: } \forall x \cdot x \in \textit{ES\_Reconf\_errors} \Rightarrow \textit{fun\_ES\_rec\_need\_b}(x \mapsto \textit{Branch\_B}) = \textit{FALSE}$$

$$\textit{Axiom 3: } \forall x \cdot x \in \textit{Any\_Error} \setminus \textit{ES\_Reconf\_errors} \Rightarrow \textit{fun\_ES\_rec\_need\_b}(x \mapsto \textit{Branch\_A}) = \textit{FALSE}$$

The introduced reconfiguration mechanism affects a number of model events. In particular, the events modelling unit failures that may trigger reconfiguration are refined to update the reconfiguration status (modelled as a separate flag variable) of a unit. The event *ES_Timeout_Err* presented in Figure 11 is an example of that. Moreover, separate events starting the reconfiguration process are introduced for each unit as the special refined versions of the abstract event *Error_Reset*. The event *ES_Reconfig_Start* shown in Figure 11 exemplifies it for the unit ES. Note that it is enabled only when the reconfiguration has been already triggered by some previous unit failure, i.e., when *flag_ES_rec = TRUE*.

Figure 11 Reconfiguration between ES branches

```
event ES_Timeout_Err refines Error_Occurrence
  where
    ES_next_targ ≠ ES_prev_targ
    ES_next_targ ≠ ES_Off
    error = No_Error
  then
    error ≔ ES_Timeout_Error
    flag_ES_rec ≔ fun_ES_rec_need_b
      (ES_Timeout_Error ↦ ES_branch)
end
```

```
event ES_Reconfig_Start refines Error_Reset
  where
    ES_Reconfig = FALSE ∧ flag_ES_rec = TRUE
  then
    ES_Reconfig ≔ TRUE
    ES_Status ≔ Unlocked
    flag_ES_rec ≔ FALSE
    error ≔ No_Error
    ES_last_mode ≔ ES_Off
    ES_prev_targ ≔ ES_Off
    ES_branch ≔ Branch_B
end
```

If the reconfiguration is not possible, i.e., both the nominal and redundant branches failed, the MM performs a rollback to such a global mode where the failed component is not used, i.e., it is in the mode *Off*. The corresponding event is shown in Figure 12.

Figure 12 System rollback

```
event MM_Error_Handling  refines MM_Error_Handling
  any m u gp
  where
    m ∈ MODES
    error ≠ No_Error
    prev_targ ↦ m ∈ Mode_Order∼
    m ≠ next_targ /\ next_targ ≠ OFF
    m ↦ u ∈ ES_mode /\ u ≠ ES_next_targ
    m ↦ gp ∈ GPS_mode /\ gp ≠ GPS_next_targ
  then
    error ≔ No_Error
    prev_targ ≔ next_targ
    last_mode ≔ next_targ
    next_targ ≔ m
    ES_Status ≔ fun_ES_status(ES_Reconfig ↦ ES_next_targ ↦ u)
    ES_prev_targ ≔ ES_next_targ
    ES_last_mode ≔ ES_next_targ
    ES_next_targ ≔ u
    ES_Reconfig ≔ FALSE || GPS_Reconfig ≔ FALSE
    GPS_Status ≔ fun_GPS_status(GPS_Reconfig ↦ GPS_next_targ ↦ gp)
    GPS_prev_targ ≔ GPS_next_targ
    GPS_last_mode ≔ GPS_next_targ
    GPS_next_targ ≔ gp
  end
```

To perform a rollback transition, all AOCS units have to be moved into appropriate modes according to the correspondence relations between MM and each unit modes (e.g., *ES_mode*, *GPS_mode*, etc.). All the variables reflecting the current state of AOCS and all the units (such as *prev_targ*, *last_mode*, *next_targ*, etc.) have to be assigned new values. The statuses of units are also changed according to the resulting values of the respective functions (e.g., *fun_ES_status*, etc.).

In our Event-B development of AOCS, we have applied the general guidelines proposed in Sections 3 and 4. We have gradually introduced the concrete modes for different system layers, their interdependencies as well as the dynamic reconfiguration mechanism. The essential mode consistency conditions have been verified in the refinement process. The modelling and verification have been carried out in the Rodin platform (Event-B, 2011). The respective proof obligations (POs) have been discharged using a collection of the provided automated theorem provers with a small number of interactive proofs (Table 2).

The statistics presented in Table 2 shows that the number of POs has significantly increased at the second and third refinement steps. Moreover, the number of interactively discharged proofs also became high at these stages. This is a natural consequence of introducing complex properties, establishing correspondence between the modes of units and the global modes of the MM.

Table 2 Proof statistics

Model	*Number of POs*	*Automatically Discharged*	*Interactively Discharged*
Context	29	29	0
Abstract model	2	0	2
1st Refinement	1	1	0
2nd Refinement	63	59	4
3rd Refinement	56	49	7
4th Refinement	21	21	0
Total	172	159	13

7 Conclusions

In this paper, we have made two main technical contributions. Firstly, we have proposed a systematic FMEA-based approach to complementing the mode logic with fault tolerance. Secondly, we have formalised the mode consistency properties and demonstrated how to ensure them while developing a system by refinement in Event-B. Our development process can be seen as a mode-consistency preserving unfolding of architectural layers. Verification by theorem proving and stepwise refinement have allowed us to undertake a formal development of a complex control system without encountering a state explosion problem. Hence, the proposed approach demonstrates good scalability and relevance to industrial practice.

In Lopatkin et al. (2011), we have derived a set of generic patterns for eliciting and structuring safety and fault tolerance requirements from FMEA. We have also developed an automatic tool support that enables interactive pattern instantiation and automatic model transformation to capture these requirements in a formal system development. In our future work, we are planning to contribute to extending the existing library of domain-specific automated patterns with FMEA mode-transition patterns for layered control systems.

Acknowledgements

This work is supported by FP7 ICT DEPLOY. Additionally, the authors would like to acknowledge the many helpful suggestions of the anonymous reviewers.

References

Abrial, J-R. (1996) *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, New York, NY, USA.

Abrial, J-R. (2010) *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, New York, NY, USA.

Avizienis, A., Laprie, J-C., Randell, B. and Landwehr, C. (2004) 'Basic concepts and taxonomy of dependable and secure computing', *IEEE Transactions on Dependable and Secure Computing*, Vol. 1, No. 1, pp.11–33.

Bozzano, M., Cavada, R., Cimatti, A., Katoen, J-P., Yen Nguyen, V., Noll, T. and Olive, X. (2010) 'Formal verification and validation of AADL models', *Proc. of the Embedded Real Time Software and Systems Conference (ERTS2 '10)*.

Buth, B. (2004) 'Analysing mode confusion: an approach using fdr2', *Proc. of the 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP '04)*, pp.101–114.

Butler, R.W. (1996) 'An introduction to requirements capture using PVS: specification of a simple autopilot', Technical report, NASA TM-110255.

ClearSy (2011) 'ClearSy, safety critical systems engineering' [online] http://www.clearsy.com/ (accessed 20 October 2011).

DEPLOY (2010) 'DEPLOY Deliverable D20 – report on pilot deployment in the space sector. FP7 ICT DEPLOY project' [online] http://www.deploy-project.eu/ (accessed January 2010).

Event-B (2011) 'Event-B and the Rodin platform' [online] http://www.event-b.org/ (accessed 20 October 2011).

FMEA IC (2011) 'FMEA Info Centre' [online] http://www.fmeainfocentre.com/ (accessed 25 October 2011).

Gan, X., Dubrovin, J. and Heljanko, K. (2011) 'A symbolic model checking approach to verifying satellite onboard software', *Proc. of the 11th International Workshop on Automated Verification of Critical Systems (AVoCS '11)*, Vol. 46.

Heimdahl, M. and Leveson, N. (1996) 'Completeness and consistency in hierarchical state-based requirements', *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, pp.363–377.

Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D. and Latvala, T. (2010a) 'Developing mode-rich satellite software by refinement in Event-B', *Proc. of the 15th International Workshop on Formal Methods for Industrial Critical Systems (FMICS '10)*, pp.50–66.

Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Väisänen, P., Ilic, D. and Latvala, T. (2010b) 'Verifying mode consistency for on-board satellite software', *Proc. of the 29th International Conference on Computer Safety, Reliability and Security (SAFECOMP '10)*, pp.126–141.

Javed, K. and Troubitsyna, E. (2012) 'Designing a fault-tolerant satellite system in SystemC', *Proc. of the 7th International Conference on Systems (ICONS '12)*, pp.49–54.

Kelly, E.M. and Bartlett, L.M. (2006) 'Application of the digraph method in system fault diagnostics', *Proc. of the First International Conference on Availability, Reliability and Security (ARES '06)*, pp.693–700.

Laibinis, L. and Troubitsyna, E. (2004) 'Fault tolerance in a layered architecture: a general specification pattern in B', *Proc. of the 2nd International Conference on Software Engineering and Formal Methods (SEFM'04)*, pp.346–355.

Leveson, N.G. (1995) *Safeware: System Safety and Computers*, Addison-Wesley, Reading, Massachusetts.

Leveson, N.G., Pinnel, L.D., Sandys, S.D., Koga, S. and Reese, J.D. (1997) 'Analyzing software specifications for mode confusion potential', *Proc. of the Workshop on Human Error and system Development*, pp.132–146.

Lopatkin, I., Iliasov, A., Romanovsky, A., Prokhorova, Y. and Troubitsyna, E. (2011) 'Patterns for representing FMEA in formal specification of control system', *Proc. of the 13th IEEE International High Assurance Systems Engineering Symposium (HASE '11)*, pp.146–151.

Metayer, C., Abrial, J-R. and Voisin, L. (2005) 'Rigorous open development environment for complex systems (RODIN). Event-B' [online] http://rodin.cs.ncl.ac.uk/deliverables /D7.pdf (accessed 2 November 2011).

Miller, S.P. and Potts, J.N. (1999) 'Detecting mode confusion through formal modeling and analysis', Technical report, NASA/CR-1999-208971.

Prokhorova, Y., Troubitsyna, E., Laibinis, L., Varpaaniemi, K. and Latvala, T. (2011a) 'Deriving mode logic for fault-tolerant control systems', *Workshop Proc. of the 5th IFIP WG 11.11 International Conference on Trust Management (IFIPTM '11)*, pp.309–323.

Prokhorova, Y., Laibinis, L., Troubitsyna, E., Varpaaniemi, K. and Latvala, T. (2011b) 'Derivation and formal verification of a mode logic for layered control systems', *Proc. of the 18th Asia-Pacific Software Engineering Conference (APSEC '11)*, pp.49–56.

Rubel, B. (1995) 'Patterns for generating a layered architecture', in Coplien, J. and Schmidt, D. (Eds): *Pattern Languages of Program Design*, Chapter 7, pp.119–128, Addison-Wesley, Reading, Massachusetts.

Rushby, J. (2002) 'Using model checking to help discover mode confusion and other automation surprises', *Reliability Engineering and System Safety*, Vol. 75, No. 2, pp.167–177.

Storey, N. (1996) *Safety-critical Computer Systems*, Addison-Wesley, Harlow, UK.