Rigorous Development of Dependable Systems using Fault Tolerance Views

Ilya Lopatkin, Alexei Iliasov, and Alexander Romanovsky
Rigorous Development of Dependable Systems using Fault Tolerance Views

Ilya Lopatkin, Alexei Iliasov, and Alexander Romanovsky

Abstract

This paper introduces the Mode and Fault Tolerance Views approach to stepwise rigorous development of critical systems. This supports structured and recursive modelling of system fault tolerance, including error detection, error recovery and degraded modes. The paper offers a formal and detailed definition of the approach, proposing a way to extend the Event-B modelling with reasoning about fault tolerance. To support the approach, a tool which is integrated into the Rodin development environment is employed. The method is shown by developing a medium-scale case study from the aerospace domain; this models complex mode management enriched with graceful degradation caused by errors.
Bibliographical details

LOPATKIN, I., ILIASOV, A., ROMANOVSKY, A.
Rigorous Development of Dependable Systems using Fault Tolerance Views
[By] I. Lopatkin, A. Iliasov, and A. Romanovsky
(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-1234)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE

Abstract

This paper introduces the Mode and Fault Tolerance Views approach to stepwise rigorous development of critical systems. This supports structured and recursive modelling of system fault tolerance, including error detection, error recovery and degraded modes. The paper offers a formal and detailed definition of the approach, proposing a way to extend the Event-B modelling with reasoning about fault tolerance. To support the approach, a tool which is integrated into the Rodin development environment is employed. The method is shown by developing a medium-scale case study from the aerospace domain; this models complex mode management enriched with graceful degradation caused by errors.

About the authors

Ilya Lopatkin graduated from the Kyrgyz Russian Slavic University, Kyrgyzstan in 2008. He is currently interested in formal modelling of fault tolerance for improving the resilience of critical systems.

Alexei Iliasov is a Researcher Associate at the School of Computing Science of Newcastle University, Newcastle-upon-Tyne, UK. He got his PhD in Computer Science in 2008 in the area of modelling artefacts reuse in formal developments. His research interests include agent systems, formal methods for software engineering and tools and environments supporting modelling and proof.

Alexander (Sascha) Romanovsky is a Professor in the Centre for Software and Reliability, Newcastle University. His main research interests are system dependability, fault tolerance, software architectures, exception handling, error recovery, system structuring and verification of fault tolerance. He received a M.Sc. degree in Applied Mathematics from Moscow State University and a PhD degree in Computer Science from St. Petersburg State Technical University. He was with this University from 1984 until 1996, doing research and teaching. In 1991 he worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland. In 1993 he was a visiting fellow at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993-94 he was a post-doctoral fellow with the Department of Computing Science, the University of Newcastle upon Tyne. In 1992-1998 he was involved in the Predictably Dependable Computing Systems (PDCS) ESPRIT Basic Research Action and the Design for Validation (DeVa) ESPRIT Basic Project. In 1998-2000 he worked on the Diversity in Safety Critical Software (DISCS) EPSRC/UK Project. Prof Romanovsky was a co-author of the Diversity with Off-The-Shelf Components (DOTS) EPSRC/UK Project and was involved in this project in 2001-2004. In 2000-2003 he was in the executive board of Dependable Systems of Systems (DSoS) IST Project. He has been the Coordinator of the Rigorous Open Development Environment for Complex Systems (RODIN) IST Project (2004-2007). He is now the Coordinator of the major FP7 DEPLOY Integrated Project (2008-2012) on Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity.

Suggested keywords

AOCS
FAULT TOLERANCE
FORMAL METHODS
EVENT-B
VIEWS
Rigorous Development of Dependable Systems using Fault Tolerance Views

Ilya Lopatkin, Alexei Iliasov, and Alexander Romanovsky

\{ilya.lopatkin, alexei.iliasov, alexander.romanovsky\}@ncl.ac.uk
Newcastle University, CSR, Newcastle upon Tyne, UK, NE1 7RU

Abstract. This paper introduces the Mode and Fault Tolerance Views approach to stepwise rigorous development of critical systems. This supports structured and recursive modelling of system fault tolerance, including error detection, error recovery and degraded modes. The paper offers a formal and detailed definition of the approach, proposing a way to extend the Event-B modelling with reasoning about fault tolerance. To support the approach, a tool which is integrated into the Rodin development environment is employed. The method is shown by developing a medium-scale case study from the aerospace domain; this models complex mode management enriched with graceful degradation caused by errors.

1 Introduction

This work is based on the analysis of the requirements documents and models produced by the industrial partners of the FP7 DEPLOY project\(^1\). During this analysis we investigated the ways in which fault tolerance, fault assumptions, and in particular, error detection and error recovery are described and modelled. First of all, we have found that dealing with these aspects represent a substantial proportion of system requirements (up to 35-40%). Moreover, we have found that the major source of faults considered in these systems is the environment, including sensors, external networks and operators. These requirements typically include descriptions of degraded functionalities, the most typical example being system safestop. More generally we observe that the requirements include information about how the general system behaviour is affected by various abnormal situations. Unfortunately this information is rarely stated as the priority requirement (sometimes, we had to deduce this information from other requirements).

We have found that nearly all system requirements use the concept of operational modes to refer to different operational conditions resulting in different functionalities provided by the system. As a result of this, system modes and mode transitions are often intertwined with error recovery; sometimes this includes fault handling by system degradation. The same intertwining can be observed in the corresponding Event-B models where one can hardly comprehend

\(^1\) ICT DEPLOY project - http://www.deploy-project.eu/
which part of the model represents the recovery activities, and which part models the normal system operation. This and similar analysis clearly demonstrate that for the majority of critical system developments it is crucial to have an explicit view on the fault tolerance-related part of the system to reduce the chance of a design fault, to improve the dependability requirement traceability and to meet the certification needs.

It is widely accepted by the software engineering community that it is beneficial to support multiple views on the model, so that each of the views can focus on a particular concern of the model/system [2]. This facilitates system development by explicitly bounding the modeller into a specific context without cluttering the model (an example of this are multiple views provided by UML). In this paper we present an approach to expressing fault tolerance (FT) views on Event-B models and a supporting tool that mechanises the formal link between the views and the models and provides the user with a simple environment for editing the FT views.

2 Modelling and Refinement in Event B

The Event B framework [5] is an extension of the B Method [4]. The Event B development starts from creating a formal system specification. The basic idea underlying stepwise development in Event B is to design the system implementation gradually, by a number of correctness preserving steps called refinements.

A simple Event B specification (called machine) encapsulates a local state (program variables) and provides operations on the state. The operations (called events) can be defined as

\[
\text{ANY } vl \text{ WHERE } g \text{ THEN } S \text{ END}
\]

where \(vl\) is a list of new local variables (parameters), the guard \(g\) is a state predicate, and the action \(S\) is a statement (assignment) describing how the system state is affected by the event. The occurrence of events represents the observable behaviour of the system. When the conditions WHEN or WHERE are satisfied, an event is enabled and its action can be executed. The action \(S\) can be either a deterministic assignment to the variables or a non-deterministic assignment from a given set or according to a given postcondition.

The INVARIANT clause of the machine contains the properties of the system (expressed as state predicates) that should be preserved during system execution. The data types and constants needed for specification of the system are defined in a separate component called Context.

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

\[
\begin{align*}
\text{Inv}(v) \land g_e(v) & \Rightarrow \exists v'. \text{Post}_e(v, v') \\
\text{Inv}(v) \land g_e(v) \land \text{Post}_e(v, v') & \Rightarrow \text{Inv}(v')
\end{align*}
\]

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

To demonstrate that each event is a correct refinement of its abstract counterpart, we should prove that the guard is strengthened in the refinement, and also demonstrate a correspondence between the abstract and concrete postconditions. Formally,

$$Inv(v) \land Inv'(v, w) \land g'_e(w) \Rightarrow g_e(v)$$

$$Inv(v) \land Inv'(v, w) \land g'_e(w) \land Post'_e(w, w') \Rightarrow \exists v'. (Post_e(v, v') \land Inv'(v', w'))$$

where the primed expressions $g'_e$, $Inv'$, $Post'$ belong to the refined model.

The consistency of Event B models as well as correctness of refinement steps should be formally demonstrated by discharging proof obligations. The Rodin platform[14], a tool supporting Event B, automatically generates the required proof obligations and attempts to automatically prove them. Sometimes it requires user assistance by invoking its interactive prover. However, in general the tool achieves high level of automation (usually over 90%) in proving.

3 FT Views

The FT Views is a modelling environment for constructing fault tolerance features in a concise manner and formally linking them to Event-B models. It provides fault tolerance modelling facilities explicitly supporting the traceability of the FT/dependability requirements. The FT View is a special case of the Mode View, developed in our previous work on modelling modal systems [7]. It is essentially an application of the Mode View approach to fault tolerance. The FT View approach extends the Mode View with additional fault tolerance semantics, structural checks, and helps the modeller by offering reusable refinement templates. The FT View approach was initially introduced in [13].

The new paper presents the overall approach and its evaluation. It reports on our recent work on introducing a tool support and on approach evaluation during development of an industrial system from the aerospace domain. This evaluation allowed us to improve both the theoretical foundations and the tool.

A mode/FT view is a graph diagram developed alongside an Event-B model which contains modes and transitions along with additional information necessary for establishing a formal connection with the model. The tool statically checks the views and generates a number of proof obligations for a target model. A user is required to demonstrate the consistency between the views and the Event-B model by discharging the obligations. We provide a brief introduction into the Mode/FT Views which are in more detail described in [13].
3.1 Overview

The two basic concepts of the Mode View are *mode* and *transition*. Mode is a general characterisation of a system behaviour. It describes the functionality of a system and the operating conditions under which the system provides this functionality. A system switches from one mode to another through a mode *transition*.

The FT View adds two types of transition specialisation: an *error* and a *recovery* transitions. Relative to the transition and its type, we differentiate the FT types of modes: we say that an error originates in a *normal mode* and leads to switching to a *degraded mode* or a *recovery mode*. The recovery transition leads from the recovery mode back to normal. Note that mode attribution to the specific type is relative and depends on the scope of discussion: what is a degraded mode in respect to one mode may be a normal mode in respect to another. In Fig. 1, mode *A* is a normal one; there is an error *e1* leading to alternative mode *B* from which the system could arrive at mode *C* upon an occurrence of another error *e2*. *B* is a degraded mode relative to *e1*, however it is a normal mode relative to *e2*.

![Fig. 1. FT View basic concepts](image1)

![Fig. 2. Degraded and recovery modes](image2)

The building blocks of a diagram are primitives describing the initiation of a degraded mode and a transition into a recovery mode (Fig. 2). The principle distinction between the two is that recovery mode is obliged to terminate and pass control back to the mode from which the initiating error originated. Safe-stop is regarded as a special case of a degraded mode.

Diagrams are built in a step-wise manner, starting from the most primitive case and introducing details using our *detalisation* process. The FT Views development process is a chain of documents similar to Event-B models. FT diagrams are built by incrementally adding new modes, errors and recoveries using the provided templates, and proving the refinement relationship between each two consequent views.

In our previous work [13] we introduced the detalisation templates for FT Views development and two general classes of fault tolerant systems that a modeller should use as an initial step during the FT modelling.

3.2 Event-B Link

Mode is a characterisation of the system behaviour. To match this notion in terms of Event-B models, modes are mapped into groups of events. A modal view is a set of modes providing different functionality under differing operating
conditions. We use the terms *assumption* to denote the different operating conditions and *guarantee* to denote the functionality ensured by the system under the corresponding assumption. With assumption and guarantee of a mode being predicates expressed on the same variables as an Event-B machine, we are able to impose restrictions on the way modes and transitions are mapped into model events and thus cross-check design decisions in either part.

Formally, a mode is characterised by a pair $A/G$ where:

- $A$ is an assumption - a predicate over the current system state;
- $G(v, v')$ is a guarantee, a relation over the current and next states of the system; and
- vector $v$ is the set of model variables.

A system switches from one mode into another through a mode transition that non-deterministically updates the state of $v$ in such a way that the assumption of the source mode becomes false while the assumption of the target mode becomes true. Transitions are also mapped into groups of events. While mode events represent internal mode transitions which must preserve the mode assumption, the transition events represent possible switches between modes that must enable the target mode assumption. If guards of two or more events are true at the time of transition, the usual Event-B demonic choice semantic applies to choose which event fires.

The link with event guards and actions is ensured by generating a number of proof obligations derived from the study on modal systems [7]. The full list is provided on a Mode/FT Views wiki page [1].

3.3 Building diagrams

The tool support for the Mode/FT Views is a plugin [1] to the Rodin platform providing a diagram editor, static checker, and a proof obligation (PO) generator. The openness of the Rodin platform allowed us to seamlessly integrate our tool into the Platform UI and the Event-B development method.

The cornerstone of the technique is an assisted construction of mode/FT diagrams coordinated with a chain of Event-B refinements. One starts building a mode/FT diagram by placing modes (nodes) and linking them with transitions (directed edges). Then a user fills the properties of modes, transitions and the diagram as he/she progresses with modelling modes, fault tolerance, and the associated Event-B model. The main feedback from the tool is in the form of the consistency proof obligations. The proof obligation generator and the automated provers run in background and a user may almost immediately observe the change in the number of discharged theorems. Analysing undischarged conditions is an efficient technique in debugging a model. After some time, a user of the Platform becomes quite adept at spotting missing hypothesis and contradictory statements and mentally translating them into the concepts of the modelled system.

The following characterisations must be provided for a mode. An *assumption* is a condition that holds as long as a system stays in the mode. Any state
transition occurring in a mode must respect the mode *Guarantee*. The pair of an assumption and guarantee gives rise to the consistency and Event-B model link proof obligations. One also needs to define a list of Event-B machine *Events* that are mapped into the mode. The mapping allows us to determine the mode of an Event-B model without requiring that each model state is uniquely associated with a mode. Finally, there is the *Refines* clause pointing to an abstract mode diagram refined by the current diagram. For a transition there must be one or more Event-B events. These events are understood to implement corresponding mode transitions.

A mode/FT refinement relation is subject to the following conditions:

- a view refines but one abstract view;
- a mode in a refined diagram must be a refinement of an abstract mode, i.e. no new modes may appear unlike Event-B events;
- an abstract mode must be refined by at least one concrete mode;
- new transitions may connect the modes refining the same abstract mode; in other words, new mode transitions are previously hidden internal transitions of abstract modes;
- transition in a refined diagram may be a refinement of an abstract transition, i.e. the concrete source and target modes must be refinements of the abstract source and target correspondingly
- a normal transition may be transformed into an error or recovery transition; this cannot be undone in subsequent refinement: error and recovery transitions may only be refined by compatible error and recovery transitions.

We consider a view of a fault tolerant system to be valid if it does not contain cycles formed entirely from error transitions, and if an error eventually ends at either a degraded or recovery mode. A mode is considered to be a recovery if there is a path back to the normal mode that contains at least one recovery transition. I.e., if an error leads to a mode switch, and the system can eventually go back to the initial mode without any recovery action involved, then such model is invalid. There should be at least one recovery action on each such transition path.

The static checker is run upon a view file when a change in the relevant documents is detected. It checks for mode reachability, names uniqueness, parses and type-checks the predicates against the Event-B model, resolves references.

Below we give a short summary of the verifications conditions (proof obligations). These include both consistency condition of a mode diagram alone and also the conditions establishing the agreement between a mode diagram and an Event-B model. Note that these conditions constitute the (proof) semantics of mode diagrams. More details on the meaning and purpose may be found in [7, 10, 13]. Here I are invariants, H and S are event guards and actions, A and G are mode assumptions and guarantees, v and u are variables on abstract and concrete levels where applicable, and primed variables depict the after-values.

- $I(v) \Rightarrow A_1 \lor A_2 \lor \cdots \lor A_n$

  **COVER**: The mode assumptions exhaust the invariant and thus cover all the system states
4 AOCS case study

4.1 System overview

The Attitude and Orbit Control System (AOCS) [3] is a generic component of satellite onboard software, the main function of which is to control the attitude and the orbit of a satellite. Due to a tendency of a satellite to change its orientation because of disturbances of the environment, the attitude needs to be continuously monitored and adjusted. An optimal attitude is required to support the needs of payload instruments and to fulfil the mission of the satellite. For example, attitude control may ensure that an optical system of the spacecraft will continuously cover the required area on the ground. In general, the behaviour of AOCS is cyclic. At each iteration the sensors provide the control algorithms with various measurements. They are used to generate the commands to the actuators that adjust the positioning of the spacecraft to ensure correct pointing of the payload instrument. AOCS consists of seven physical units: four sensors, two actuators and the payload instrument.
A satellite can be in various operational modes, largely determining its behaviour [3]: Off, Standby, Safe, Nominal, Preparation and Science. The satellite is in the Safe mode from the moment separation from the launcher is achieved. In this mode it tries to acquire and preserve a stable attitude. From Safe, satellite progresses to modes where more sensors and actuators are involved. The overall aim is to enter and stay in the Science mode where fine positioning is achieved and scientific instruments are reporting readings.

The AOCS is expected to handle the mode transition errors (such as time-outs), the control algorithm related errors (such as attitude computation errors) and the unit errors (including all errors related to failures of redundant units, loss of accuracy, invalid data, etc.).

4.2 AOCS Modelling

In this work we are not attempting to model the complete system although such models can be found elsewhere [3, 11]. The goal is to investigate the applicability of the method and the tool in the context of a realistic system. We focus on the modal and fault tolerance aspects of the system and investigate the Mode/FT Views modelling technique in the context of the AOCS case study. In particular, we want to understand the benefits and possible drawbacks of the method, define the level of abstraction at which modelling modes/FT is most fitting.

As was mentioned in section 2 the process of modelling in Event-B is based on the stepwise refinement of the models. We start with an abstract specification and create more detailed models proving each time their correctness and the refinement relation. During the modelling of the AOCS we have produced 6 machines (Event-B models of behaviour) together with 5 contexts (static parts of the models), and 6 views (Fig. 3). In the first two Event-B models M0 and M1 we define the process of system undergoing reconfiguration and trying to progress to the Scientific mode. In M2 we add units and mode properties, we verify that the current unit states correspond to the required mode configuration. In M3 we model errors, unit redundancy, and verify that the units required for the mode configuration are always available. The PLI model is an instantiation of the M3 showing the modal behaviour of a specific unit (payload instrument) in presence of errors. M4 finalises the modelling by showing that the required scenario of the autonomous mode switching agrees with the unit and mode management.

**M0 model and its two modal views** In the first model we introduce the main aspect of the AOCS system that is the system-level mode management. To represent the modes we define a set of constants $MODES$ according to the six modes described previously. We know that the autonomous scenario of the AOCS is sequential and arrange modes into a sequence (formally a strict partial order: antisymmetric, irreflexive and transitive). The AOCS system is always either in a stable mode or being reconfigured. The variable $currentMode \in MODES$ defines the last stable mode, and $targetMode \in MODES$ defines the target mode of reconfiguration. If the system is in a stable mode, then $currentMode =$
targetMode. There are two events `stable` and `reconf` in the model. The first view of the system is shown on Fig. 4.

![Development diagram of the AOCS modelling](image)

**Fig. 3.** Development diagram of the AOCS modelling

On the first view, each mode is mapped to the event with the corresponding name in the model. This is the simplest one-to-one mapping at this level, although, generally, it is allowed to associate several events with a mode and even have the same event linked with several modes. On the view, a mode transition leading from a given mode is mapped to the same event as the mode. When the system is in the mode `Stable`, its assumption must hold, that is the AOCS target mode must be equal to the current mode. When the target is set to a different mode, the system switches to the `Reconfiguration` mode. It stays in this mode until the reconfiguration completes. The corresponding model events have
non-deterministic assignments - this allows us to map a single event to multiple modes and transitions and thus be flexible with the abstract modelling.

Another view on the same model (Fig. 5) shows the partitioning of the AOCS modes into two subsets: Safe and Science. Although the view is different it still characterises the same model though from a new angle. The perspective of the view is defined by the assumption and guarantee predicates. We partitioned the set $MODES$ into two parts - one represents the preliminary stage of the AOCS operation (initiation of fine positioning), the other depicts the stage when the AOCS performs the collection of scientific data using its payload instrument.

![Diagram](attachment:diagram.png)

**Fig. 5.** The first modal view of the AOCS system

Although model is abstract, the graphical views already conveys some important properties of the system. One of the views shows two distinct phases of the AOCS operation - with and without the payload involved. The second view makes a distinction between the stable and reconfiguration modes. In formal terms, there is a proof that the abstract model contains phenomena described by the views. These phenomena would be preserved and developed during the refinement process. In fact, even at the level of the most detailed Event-B model we are able to observe the mode switches described by the views of the abstract model.

The proof obligations produced at this step were discharged automatically by the Rodin provers.

**M1 model with a refined view** In the first refinement step we refine the abstract stable and reconfiguration events with the guards that add determinism to the system behaviour. The abstract events could initiate the reconfiguration (assign to $targetMode$) non-deterministically. In the concrete stable event we add the guard

$$(newMode = currentMode) \lor (newMode = currentMode + 1) \lor \lor (newMode < currentMode)$$
where newMode is an event parameter denoting the new target mode. This guard restricts the initiation of reconfiguration to the next advanced mode or one of the lower modes. Hence, the system cannot directly switch from Off to Science jumping over the Safe mode. The guard of the reconf event is extended in a similar way. The system safe states on this level are restricted in comparison to the abstract model where there are no invariants except the type definitions. Such restriction is captured in the invariant

\[(\text{currentMode} = \text{targetMode}) \lor (\text{currentMode} + 1 = \text{targetMode}) \lor (\text{targetMode} < \text{currentMode})\]

which eliminates the reconfigurations to more than one mode ahead.

![Diagram](image)

Fig. 6. The second reconfiguration view

The reconfiguration view on this model (Fig. 6) splits the Reconfiguration mode into two: Advance and Downgrade. On the diagram we emphasize that the Downgrade mode is a recovery activity engaged as a consequence of some erroneous action. This is shown by the type of arrows: a bold arrow with a filled pointer is an error transition; a bold arrow starting with a diamond is a recovery transition. The Downgrade recovery always leads to the Stable mode and there is no transition to the Advance mode. The diagram embodies the additional modal properties of the system that are not easy to manually encode as Event-B safety properties.

Both modes refer to the reconf event. Their assume/guarantee pair splits the guard and action of the event into two parts. To formally show the consistency with the model, there is a number of proof obligations generated. These were discharged automatically.
One of the interesting proof obligations is demonstrating the link between mode guarantees and event actions (EVT\_G) for the mode Stable. Event stable plays dual role: it is an event of the mode and it is also an event realising the outgoing transitions. Therefore, the event action must non-deterministically represent all these possible state changes. One can observe that the disjunction of the mode guarantee and both assumptions of the other two modes follow from the conjunction of the invariant, mode assumption, event guard, and event action that we provided above.

**M2 model** This model is not associated with a view. In this model we extend the abstract mode management with the notion of unit management. We declare set UNIT of constants that represent seven hardware units of the AOCS. Constant function FUnitConf ∈ MODE × UNIT → N defines the mapping from modes into unit states. Also there defined are a number of axioms for the specific configurations given by the requirements. For instance, in the Preparation mode the payload instrument must be in the Standby unit mode: FUnitConf(PREPARATION ↦ PLI) = PliStandby. We track the current states of the units in the variable unitStates ∈ UNIT → N. The reconf event is split into three concrete events. The property we verify for this model state that our units have to be in a particular configuration when the overall system is in a stable mode:

\[(\text{currentMode} = \text{targetMode}) \Rightarrow (\forall a \cdot a \in \text{UNIT} \Rightarrow \text{unitStates}(a) = FUnitConf(\text{currentMode} \mapsto a))\]

**M3 model** So far the model represented an idealised system free from adverse environmental interference. At this level we introduce errors and unit redundancy to mask the errors. As specified for the AOCS system, each unit has a redundant spare which is enabled when an error in the current unit is detected. We add the following to the model:

- set of constants ERROR = NoError, UnitError, AttitudeError to represent two possible kinds of errors, and a variable error ∈ ERROR that we non-deterministically assign in one of events; this abstractly represents a source of errors in the environment;
- variable units ∈ UNIT → N that gives a number of available units of a certain kind. Our system initially has two units of each kind.

An attitude error is easily traceable in the model, the only reaction of the system is the degradation according to the rules provided in requirements. However, the reaction of the system to the unit errors is more complex. Firstly, the system does not change the mode if there is a spare unit available. It disables the erroneous unit and enables the spare one. In case a failure occurs in the only remaining unit, the system marks that the units of this kind are not available and degrades to a previous (less advanced) mode in which
this kind of unit is not required. To model this, we define a constant function 
\( F_{\text{MaximumMode}} \in (UNIT \rightarrow \mathbb{N}) \rightarrow \text{MODE} \) which returns the maximum possible mode for the current set of available units. The events representing the reactions of the system to the unit errors have the following in their guards:

\[
\text{newMode} \leq F_{\text{MaximumMode}}(\text{units} \leftarrow \{\text{erroneousUnit} \mapsto 0\})
\]

where \( \text{erroneousUnit} \) contains the failed unit. The guard ensures that the units required by the target mode are available. The full definition of the function \( F_{\text{MaximumMode}} \) results in 27 proof obligations, 20 of which were discharged automatically. These helper axioms helped to create more comprehensible models and views. For example, this is an event of the reaction of the system to the unit error during a stable mode:

\[
\text{recStable} = \text{any newMode where} \\
\begin{align*}
\text{currentMode} &= \text{targetMode} & \text{//the mode is stable} \\
\text{newMode} &= \in \text{MODE} \\
\text{newMode} &< \text{currentMode} & \text{//the system will downgrade} \\
\text{error} &= \text{UnitError} & \text{//a unit error is detected} \\
\text{units}(&\text{erroneousUnit}) &< 2 & \text{//one or no such units left} \\
F_{\text{UnitConf}}(\text{currentMode} \mapsto \text{erroneousUnit}) &> 0 \\
\text{newMode} &\leq F_{\text{MaximumMode}}(\text{units} \leftarrow \{\text{erroneousUnit} \mapsto 0\})
\end{align*}
\]

then

\[
\text{targetMode} := \text{newMode} & \text{//initiate the reconfiguration} \\
\text{error} := \text{NoError} & \text{//mask the error} \\
\text{units}(&\text{erroneousUnit}) := 0 & \text{//disable the unit}
\]

end

The event models the choice of a new target mode from a stable mode. The system exhibits similar behaviour during the reconfiguration process. The property to prove for this model is that the units necessary for the target mode are available during both stable and reconfiguration activities:

\[
\forall a \cdot a \in UNIT \land F_{\text{UnitConf}}(\text{targetMode} \mapsto a) > 0 \Rightarrow \text{units}(a) > 0
\]

The property can be expressed in a more compact way thanks to the function we have defined on mode configurations:

\[
\text{targetMode} \leq F_{\text{MaximumMode}}(\text{units})
\]

An error in a unit may only be detected when the unit is operational:

\[
\text{error} = \text{UnitError} \Rightarrow \\
\text{units}(\text{erroneousUnit}) > 0 \land \text{unitStates}(\text{erroneousUnit}) > 0
\]

A specific view on a model is one way to communicate design decision taken in a formal model to domain experts. We build such a view to explain how the system switches into a degraded mode. If a unit fails, the system can no longer
be in certain modes but it still does not signify a failure at the global level. We call this a graceful degradation of the behaviour (Fig. 7).

Fig. 7. The degraded modes of the AOCS

Each mode on the diagram represents the maximum (according to the ordering sequence defined above) mode of the AOCS that is reachable with the currently available set of units. Initially, satisfying its purpose, the maximum mode is the Science mode. This is the most desirable reachable mode after the start. After some time, both payload instruments (PLI) can fail, and the maximum mode for the system becomes the Nominal. If the steering devices fail, the system cannot advance beyond the Safe mode. Finally, when any kind of the tracking devices fail, the system reboots and stays in the Standby mode.

**PLI unit instantiation** At the third refinement step we implement an important requirement regarding the availability of units and their redundant counterparts. We explicitly tell how a single unit and its spare interact. In other words, the behaviour of a system when it detects a unit error and has to switch to its spare. Event-B model elements relevant to such interaction are scattered all over the model. To obtain a clean picture of this aspect of the model, we instantiate a unit refining error handling events, and create a view on a single unit (PLI) and its spare (Fig. 8).
There are three modes for a nominal PLI unit and another three for the spare unit. The assumptions and guarantees for the six modes are of the form

\[ \text{unitStates}(PLI) = state \land \text{units}(PLI) = a \]

where \( state \in \{\text{UnitOff}, \text{PliScience}, \text{PliStandby}\} \) corresponding to the PLI modes, \( a = 2 \) for the main unit, and \( a = 1 \) for the redundant unit modes. A transition to the spare unit happens upon an occurrence of a PLI error within the main unit, and it keeps the corresponding mode for the redundant unit. On occurrence of this error in the redundant unit, the PLI unit becomes unavailable (\( \text{units}(PLI) = 0 \)). There are no transitions between \( \text{Off} \) modes, and no error originates from them - this is due to the fact that no error can arise in a non-working unit. The assumption/guarantees concern the current status and availability of the payload. Note that the assumptions of the modes must be consistent with the safe states defined by the model invariants. The proof obligation named COVER establishes such consistency showing that the invariant implies the disjunction of all mode assumptions. Since the assumptions cover all possible variations of the unit status and availability, it is sufficient to use the constant definitions and typing invariants to prove that the following holds:

\[
\begin{align*}
(\text{unitStates}(PLI) = \text{UnitOff} \land \text{units}(PLI) = 2) \lor \\
(\text{unitStates}(PLI) = \text{PliStandby} \land \text{units}(PLI) = 2) \lor \\
(\text{unitStates}(PLI) = \text{PliScience} \land \text{units}(PLI) = 2) \lor \\
(\text{unitStates}(PLI) = \text{UnitOff} \land \text{units}(PLI) = 1) \lor \\
(\text{unitStates}(PLI) = \text{PliStandby} \land \text{units}(PLI) = 1) \lor \\
(\text{unitStates}(PLI) = \text{PliScience} \land \text{units}(PLI) = 1) \lor \text{units}(PLI) = 0
\end{align*}
\]

**M4 and an autonomous reconfiguration scenario** The next step is to define specific mode transitions according to the specified AOCS sequential scenario which respects the unit management rules. For this, we create a number
of constant functions:

\[ F_{\text{TransitionNewTarget}}, F_{\text{FdirNewTarget}}, F_{\text{StableAttitudeNewTarget}} \in \text{MODE} \rightarrow \text{MODE} \]

These return the new target mode for a given current target of reconfiguration. We distinguish three cases: an error during the advance, an error during the downgrade, and attitude error during the stable mode accordingly. The other cases including unit errors are already covered on the previous level by the unit management logic. The refined events now initiate the downgrade according to these functions derived from the requirements. For such refinement to be correct, the functions need to respect the mode and unit management rules defined previously. In particular, there must be a correspondence between the transition functions and the \( F_{\text{MaximumMode}} \) function on the unit states. Such correspondence is ensured by the standard Event-B GRD and SIM proof obligations which link guards and actions of the events correspondingly on the two consequent levels of abstraction.

The modal view on the system (Fig. 9) contains six modes. The assumptions and guarantees are no more complex than they were on the first modal view:

\[
\begin{align*}
\text{currentMode} = \text{mode} \\
\text{mode} \in \{\text{OFF, STANDBY, SAFE, NOMINAL, PREPARATION, SCIENCE}\}
\end{align*}
\]

However, the mapping to events is significantly larger because of the higher number of events representing different functional and fault tolerance behaviour. All transitions are mapped to the single event which changes the value of the \( \text{targetMode} \) and switches the mode according to this view. The reconfiguration is therefore a part of a previous mode.

![Fig. 9. The second modal view](image)

Overall, the modelling effort resulted in 650 proof obligations of which 550 were discharged automatically. Approximately 70% of proof obligations are concerned with the modal/ft views consistency and Event-B link. Importantly, the percentage of interactive proofs for mode-related theorems is within the bounds of what is expected in an Event-B development. Clearly, the proportion of modal
proof obligations is high in this case study due to a deliberate attempt to emphasize the working of modes/ft approach.

5 Discussion and lessons learnt

The two types of models, Event-B models and FT views, are developed in parallel and proved to be consistent with each other and with the abstract models. A user is free to choose which of the two documents to develop first. We developed our case study using the following steps:

- the requirements were written;
- we made sketches of the mode/FT diagrams that correspond to the requirements, but without formal part;
- we started a stepwise development, where we performed the following for each model:
  - each model was developed involving some inevitable but small changes in the abstract model
  - a mode/FT view was linked to the model, and the model was proved to implement that view
  - if we could not discharge some proof obligation, then changes were made either to the model, or to the view until they became consistent

Shortly, we first create the model and then show that it implements the view. If they are not consistent, then usually the model is the document to change since the view is typically simpler and more comprehensible.

As can be seen from the case study, the Mode/FT Views approach costs proofs and therefore platform performance and manual efforts. On the other side, the explicitness, separation of concern, and potential improvement in traceability can dramatically increase the development quality and improve the communication between modellers and requirements engineers. A single case study performed by developers is certainly not enough to come to a single conclusion about long term benefits. However, even in a short term we found it worthwhile to use such approach for modelling modal systems which come with fault tolerance requirements. In particular, tracing such requirements into views gives a better understanding of the system behaviour and leaves the models uncluttered, additional POs often point at the inconsistencies in the models which otherwise would be difficult to cover using safety invariants.

The views are orthogonal to Event-B and each other. This gives a flexibility in choosing the "angle" of the view on the model. Changing the assumption/guarantees leads to a different view and provides with a powerful tool for covering the behavioural aspects of the system. As we showed in the case study, a single model can have a number of views, and each view can describe a different aspect of a modal behaviour or FT features. It is not only beneficial but essential to separate the views from each other. From our experiments, placing slightly different behavioural viewpoints on a single diagram clutters the view significantly. In particular, mixing modal descriptions with various error transitions
leads to almost a “multiplication” of the diagram elements. On the opposite, separation simplifies the views, and greatly reduces the PO complexity. One of such simplified viewpoints that we produced as a result of our experiments was dedicated to degraded modes of the system.

6 Related work

In [8] Hall shares his experience in using formal techniques in industrial projects. In particular, he discusses the importance of using specific methods and notations for specifying certain aspects of the systems under development. Also related to our work are the characteristics of the specification notations Hall defines as being the most important for users: clarity and expressiveness, these are the properties we provide for the users of the FT Views.

Separation of concerns has been always of a high importance to the computer science research. The recent standard on architectural descriptions [2] puts such separation in a framework. The concern is a framework term used for the set of properties and aspects that one of the stakeholders is interested to see in the system. The examples of concerns are the performance, safety, fault-tolerance, real-time – related, etc. A more specific description of the concern comprises a viewpoint on the system and is typically supported by domain-specific tools and notations. A view is an instance of a viewpoint within a project on a specific (sub)system. The existence of multiple views on the problem/system gives rise to the consistency and parallel refinement issues. A particular example of verifying model transformations that involve multiple views is presented in [6]. The paper presents a technique for proving the behaviour preservation of the overall model transformation in presence of the sub-transformations on the individual views that are not behaviour preserving. The views chosen as an example are Object-Z as a static part with its data refinement and a CSP process view for dynamic behaviour.

The work presented in this paper is based on our previous work on formal specification of modal systems [7, 10] where we provided the theory of modes and a sound link to the state-based formalisms, exemplified by Event-B. Other related works include [12] where the concept of mode is introduced into the component behaviour specification using the formalism of extended behaviour protocols. The component behavioural modes are then used on the system level to model the behaviour of the product lines. The approach supports formal specification of modes and transitions, and verification by model-checking. Another paper on introducing modes on the architectural level [9] talks about modes as architectural constraints over subsystem configurations. A system mode is linked with a system subtask and is a composition of component modes. The authors introduce the notion of modes to the Darwin architectural language and give an example from the automotive domain.
7 Acknowledgements

This work is supported by the FP7 DEPLOY Project on industrial deployment of system engineering methods providing high dependability and productivity. Ilya Lopatkin is partially supported by Newcastle University (UK). We are grateful to our colleagues in DEPLOY for all their suggestions for improving this work.

References