Rigorous Development of Dependable Systems using Fault Tolerance Views

Ilya Lopatkin, Alexei Iliasov, Alexander Romanovsky
Centre for Software Reliability
Newcastle University
Newcastle upon Tyne, UK
Email: {ilya.lopatkin, alexei.iliasov, alexander.romanovsky}@ncl.ac.uk

Abstract—This paper introduces the Mode and Fault Tolerance Views approach to stepwise rigorous development of critical systems. It supports systematic, structured and recursive modelling of system fault tolerance, including error detection, error recovery and degraded modes. Built on our previous work extending the Event-B method with reasoning about fault tolerance, the paper focuses on a practical application and evaluation of the approach. The proposed modelling approach is backed by an integrated toolset. The paper is illustrated with a case study from the aerospace domain.

Index Terms—formal methods; Event-B; fault tolerance; modal systems; case study; AOCs

I. INTRODUCTION

This work is based on the analysis of the requirements documents and models produced by the industrial partners within the FP7 DEPLOY Integrated Project 1 on industrial deployment of system engineering methods providing high dependability and productivity. The partners represent the aerospace, automotive and transportation sectors. During this analysis we have investigated the ways in which fault tolerance, fault assumptions, and, in particular, error detection and error recovery are described by the system stakeholders. First of all, we have found that dealing with these aspects represent a substantial part of system requirements (up to 35-40%). Moreover, we have found that the major source of faults to be dealt with by 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. In spite of the success of this analysis we should mention here that, unfortunately, this information is rarely stated as the priority requirements (too often we had to deduce this information from other requirements). A possible source of confusion is the terminology - industrial partners may prefer to use euphemisms to replace terms such as system failures, faults and errors.

We have found that nearly all system requirements involve the concept of operational modes to refer to various operational conditions resulting in differing 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 at the modelling stage where one can hardly comprehend which part of the model represents the recovery activities, and which part corresponds to 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 [1]. 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 formal models of software and hardware system and describe a supporting tool that provides a substantial mechanisation of the handling and validation of FT views.

The rest of the paper is organised as follows. Section II briefly describes Event-B - a formal notation and method for the correct-by-construction system development. Section III gives a general overview of the FT Views approach. We give a detailed description of the case study in section IV and discuss its outcomes in section V.

II. MODELLING AND REFINEMENT IN EVENT-B

The Event-B framework [2] is an evolution of the B Method [3]. 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

---

1ICT DEPLOY project - http://www.deploy-project.eu/
(assignment) describing how the system state is affected by
the event. The occurrence of events represents the observable
behaviour of the system. When the condition WHERE is satis-
fi ed, an event is enabled and its action can be executed. The
action $S$ can be either a deterministic or a non-deterministic
assignment.

The INVARIANT clause of the machine contains the prop-
erties 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,

$$\text{Inv}(v) \land g_e(v) \Rightarrow \exists v'. \text{Post}_e(v, v')$$
$$\text{Inv}(v) \land g_e(v) \land \text{Post}_e(v, v') \Rightarrow \text{Inv}(v')$$

The main development methodology of Event-B is refine-
ment – the process of transforming an abstract specification
to gradually introduce implementation details while preserv-
ing its correctness. Refinement allows us to reduce non-
determinism present in an abstract 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 corre-
spondence between the abstract and concrete postconditions.
Formally,

$$\text{Inv}(v) \land \text{Inv}'(v, w) \land g'_e(w) \Rightarrow g_e(v)$$
$$\text{Inv}(v) \land \text{Inv}'(v, w) \land g'_e(w) \land \text{Post}'_e(w, v') \Rightarrow \exists v'. (\text{Post}_e(v, v') \land \text{Inv}'(v', v''))$$

where the primed expressions $g'$, $\text{Inv}'$, $\text{Post}'$ belong to
the refined model. The machines linked with each other by
refinement form a development chain, or in a more general
case development represents a tree.

The consistency of Event-B models as well as correctness of
refinement steps should be formally demonstrated by discharg-
ing proof obligations. The Rodin platform[4], a tool supporting
Event-B, automatically generates the required proof obliga-
tions 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.

III. FT VIEWS

The FT Views [5] is a modelling environment for describing
the fault tolerance aspect of a system in a concise manner
while formally linking it to the main model. Much of the
formal foundations of FT Views are based on the previous
work on modelling modal systems [6].

This work briefly presents the overall approach and de-
scribes its recent improvements. 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 supporting tool, and to gain
experience in rigorous engineering of fault tolerance systems
using this approach.

A. Overview

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 two basic concepts of the
Mode View are mode and transition. Mode is a general charac-
terisation 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 Views 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. The distinction
between the degraded and recovery modes is that the 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 and introducing details using our detalisation
process [5]. An FT Views development comprises a chain of
documents similar to Event-B development. FT views are built
by incrementally adding new modes, errors and recoveries
using the provided templates, and proving the refinement
relationship between each two consequent views.

[5] introduces the concept of detalisation templates for FT
Views development with two general classes of fault tolerant
systems that a modeller should use as an initial step during
the FT modelling. The first class comprises the systems which
are able to mask all the detected errors. The systems of
the second class have recovery or degraded modes at the
abstract level as they are incapable of masking some errors.
By using detalisation templates a modeller can refine a mode
and differentiate recovery activities from normal operation, or
split an abstract recovery into a number of concrete recoveries
from specific errors thus covering the requirements.

B. Event-B Link

An FT view is linked with a formal model and ensures that
the model implements the features described in the view. For
that, modes are mapped into groups of events.

We use the terms assumption to denote the different op-
rating conditions and guarantee to denote the functionality
ensured by the system under the corresponding assumption.
Formally, a mode is characterised by a pair $A/G$ where:

- $A(v)$ 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.

With assumption and guarantee of a mode being predicates expressed on the variables of a model, 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.

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 each of which must implement an instantaneous transition action.

The link with event guards and actions is ensured by generating a number of proof obligations derived from the study on modal systems [6]. The full list is provided on a Mode/FT Views wiki page [7].

C. Building diagrams

The tool support for the Mode/FT Views is a plug-in [7] to the Rodin Platform providing a diagram editor, static checker, and a proof obligation (PO) generator.

The cornerstone of the technique is an assisted construction of Mode/FT views coordinated with a chain of Event-B refinements. One starts building a Mode/FT diagram by placing modes and linking them with transitions. 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 full list of verification conditions (proof obligations) and details on the meaning and purpose may be found in [7], [6], [8], [5].

IV. AOCS case study

The Attitude and Orbit Control System (AOCS) [9] is a generic component of a satellite onboard software, the main function of which is to control the attitude and the orbit of a satellite. Due to the 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 fulfill 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. AOCS consists of seven physical units: four sensors providing measurements to control algorithms, two actuators and the payload instrument.

A satellite can be in various operational modes, largely determining its behaviour [9]: 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 timeouts), 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.).

A. AOCS Modelling

In this work we are not attempting to model the complete system although such models can be found elsewhere [9], [10]. 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 Mode/FT is most fitting.

As was mentioned in section II, 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) and 6 views (Fig. 1). In the first two Event-B models M0 and M1 we define the process of system undergoing reconfiguration and trying to progress through modes. 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.

1) M0 model and its two modal views: In the abstract 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 $\text{MODES} \subset \mathbb{N}$. We know that the autonomous scenario of the AOCS is sequential and arrange modes into a sequence (formally a strict partial order). At this abstraction level, we only define the initial mode $\text{OFF} = 0$.

The AOCS system is always either in a stable mode or being reconfigured. The variable $\text{currentMode}$ defines the last stable mode, and $\text{targetMode}$ defines the target mode of reconfiguration. When $\text{currentMode} = \text{targetMode}$, the system is considered to be in a stable mode. There are two events $\text{stable}$ and $\text{reconf}$ as shown on Snippet 1.

The first view of the system is shown on Fig. 2. On the 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. Each event is mapped to the corresponding mode as well as to its outgoing transition. 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. Note how the Reconfiguration mode does not guarantee to preserve the initial current and target values. During the reconfiguration the

system might decide to change its target mode and downgrade to one of lower modes. The corresponding model events have non-deterministic assignments: the event stable may assign any value to the variable targetMode thus initiating a reconfiguration, and the event reconf non-deterministically assigns to both currentMode and targetMode. Such non-deterministic abstraction allows us to map a single event to multiple modes and transitions and thus be flexible with the abstract modelling. The more deterministic behaviour will be defined via refinement.

Another view on the same model (Fig. 3) 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

**Snippet 1** The abstract model M0

**variables**

\[ \text{currentMode}, \text{targetMode} \]

**invariant**

\[ \text{inv1: currentMode} \in \text{MODE} \]
\[ \text{inv2: targetMode} \in \text{MODE} \]

**events**

- **Initialisation**
  \[ \text{act1: currentMode} := \text{OFF} \]
  \[ \text{act2: targetMode} := \text{OFF} \]

- **event stable**
  \[ \text{any newMode} \]
  \[ \text{where} \]
  \[ \text{grd1: currentMode} = \text{targetMode} \]
  \[ \text{grd2: newMode} \in \text{MODE} \]
  \[ \text{then} \]
  \[ \text{act1: targetMode} := \text{newMode} \]

- **event reconf**
  \[ \text{any newTargetMode newCurrentMode} \]
  \[ \text{where} \]
  \[ \text{grd1: currentMode} \neq \text{targetMode} \]
  \[ \text{grd2: newTargetMode} \in \text{MODE} \]
  \[ \text{grd3: newCurrentMode} \in \text{MODE} \]
  \[ \text{then} \]
  \[ \text{act1: currentMode} := \text{newCurrentMode} \]
  \[ \text{act2: targetMode} := \text{newTargetMode} \]

end
fine positioning), the other depicts the stage when the AOCS performs the collection of scientific data using its payload instrument.

Although the model is abstract, the graphical views already convey 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 formal link between the views and the model is achieved by generating a number of proof obligations by the tool. These are machine checked and at this step are discharged automatically.

2) **M1 model with a refined view**: In the first refinement step we refine the two abstract events with the guards that add determinism to the system behaviour. We restrict the system safe states by imposing an invariant and adding appropriate guards to ensure that the system can only initiate the 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 as it would break the invariant (Snippet 2). During reconfiguration the system may decide to change the target mode to downgrade. In such case we consider the system to achieve the previously desired mode and start downgrade atomically. This is to ensure that the system does not arrive at a stable mode if an error forbidding that mode is detected.

**Snippet 2 Extended model M1**

<table>
<thead>
<tr>
<th>invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv1: currentMode = targetMode ∨ currentMode + 1 = targetMode ∨ targetMode &lt; currentMode</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>events</th>
</tr>
</thead>
<tbody>
<tr>
<td>event stable ≜ extends stable</td>
</tr>
<tr>
<td>where</td>
</tr>
<tr>
<td>grd3: newMode = currentMode ∨ newMode &lt; currentMode ∨ newMode = currentMode + 1</td>
</tr>
</tbody>
</table>

| event reconf ≜ extends reconf |
| where |
| grd4: (newTargetMode = targetMode ∧ newCurrentMode = currentMode) ∨ (newTargetMode < targetMode ∧ newCurrentMode = targetMode) ∨ (newTargetMode = targetMode ∧ newCurrentMode = targetMode) |

The reconfiguration view on this model (Fig. 4) 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 shown by a bold arrow. The Downgrade recovery always leads to the Stable mode (a bold transition starting with a diamond) and there is no transition to the Advance mode. Both Advance and Downgrade modes, as well as outgoing transitions, refer to the reconf event. The diagram embodies the additional modal properties of the system that are not easy to manually encode as Event-B safety properties.

Let us consider one of the proof obligations showing the link between mode guarantees and the corresponding event actions:

\[ I(v) \land A(v) \land H(v) \land S(v, v') \Rightarrow G(v, v') \lor A_1(v') \lor \ldots A_n(v') \] (EVT\_G)

Each event within a mode must satisfy the mode guarantee upon action. If the same event is also associated with the outgoing transitions then the goal would include the assumptions of the target modes. Thus, the event must either preserve the system functionality in the current mode or correctly switch to another mode. For example, mode Downgrade and its outgoing transition into Stable are associated with the event reconf, and hence the proof obligation:

\[ inv1 \land A_{Downgrade} \land grd1 \land grd4 \land act1 \land act2 \Rightarrow G_{Downgrade} \lor A_{Stable} \]
Rodin provers looks as the following:

\[
\begin{align*}
\text{currentMode} & \neq \text{targetMode} \\
(\text{newTargetMode} = \text{targetMode} \\
\land \text{newCurrentMode} = \text{currentMode}) \\
\lor (\text{newTargetMode} < \text{targetMode} \\
\land \text{newCurrentMode} = \text{targetMode}) \\
\lor (\text{newTargetMode} = \text{targetMode} \\
\land \text{newCurrentMode} = \text{targetMode}) \\
\text{targetMode} < \text{currentMode} \\
\implies \text{newCurrentMode} = \text{newTargetMode} \\
\lor \text{newTargetMode} < \text{newCurrentMode}
\end{align*}
\]

There are also other proof obligations that formally show the consistency with the model (see the full list at [7]). All of them are discharged automatically.

3) **M2 model**: This refinement step is not associated with a view. In this model we extend the abstract mode management with the notion of unit management. We declare a set \(\text{UNIT}\) that represents hardware units of the AOCS. Constant function \(\text{FUnitConf} \in \text{MODE} \times \text{UNIT} \rightarrow \mathbb{N}\) defines the mapping from modes into unit states. We track the current states of the units in the variable \(\text{unitStates}\). For the sake of simplicity, a zero value corresponds to a switched-off state of a unit.

The stabilisation part of the \(\text{reconf}\) event (when \(\text{newCurrentMode} = \text{targetMode}\)) is atomically refined into the unit reconfiguration process implemented by two events: \(\text{unitReconf}\) and \(\text{reconfFinish}\). The rest of \(\text{reconf}\) is left non-deterministic for further refinements. 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 (inv2 at Snippet 3).

4) **M3 model**: So far the model represented an idealised system free from adverse interference from the environment. At this level we introduce errors and unit redundancy to mask these 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 abstractly represent the errors by a set of constants \(\text{ERROR} = \{\text{NoError}, \text{UnitError}, \text{AttitudeError}\}\) and a variable \(\text{error}\) that we non-deterministically assign in one of new events, this abstractly represents a source of errors in the environment. Variable \(\text{units}\) returns a number of available units of a certain kind. Our system initially has two units of each kind. The new variables and invariants are shown on Snippet 4.

inv4 states that the units necessary for the target mode are always available. inv5 and inv6 ensure that non-operating units cannot produce errors. inv7 requires the target mode to be less or equal to the maximum mode possible under current units availability. This is equivalent to inv4, and is used to simplify proofs and FT degradation view (see next section).

Formal definition of \(\text{FMaximumMode}\) is given on Snippet 5. According to axiom2, for all modes lesser or equal to the

Snippet 3 M2 with unit reconfiguration

<table>
<thead>
<tr>
<th>variables</th>
<th>unitStates currentMode targetMode</th>
</tr>
</thead>
<tbody>
<tr>
<td>invariant</td>
<td></td>
</tr>
<tr>
<td>inv1:</td>
<td>unitStates (\in) UNIT (\rightarrow) (\mathbb{N})</td>
</tr>
<tr>
<td>inv2:</td>
<td>currentMode = targetMode (\Rightarrow) ((\forall a \cdot a \in UNIT \Rightarrow unitStates(a) = \text{FUnitConf}(currentMode \mapsto a)))</td>
</tr>
<tr>
<td>inv3:</td>
<td>(\forall) unit \cdot unit (\in) UNIT \land unitStates(unit) (\in) ran((MODE \times {unit}) \land \text{FUnitConf})</td>
</tr>
<tr>
<td>events</td>
<td></td>
</tr>
<tr>
<td>Initialisation</td>
<td></td>
</tr>
<tr>
<td>act3:</td>
<td>unitStates := UNIT \times {0}</td>
</tr>
<tr>
<td>event stable (\equiv) extends stable</td>
<td></td>
</tr>
<tr>
<td>event reconf (\equiv) refines reconf</td>
<td></td>
</tr>
<tr>
<td>any newCurrentMode newTargetMode</td>
<td></td>
</tr>
<tr>
<td>where</td>
<td></td>
</tr>
<tr>
<td>grd1:</td>
<td>currentMode \neq targetMode</td>
</tr>
<tr>
<td>grd2:</td>
<td>newTargetMode \in MODE</td>
</tr>
<tr>
<td>grd3:</td>
<td>newCurrentMode \in MODE</td>
</tr>
<tr>
<td>grd4:</td>
<td>(\text{newTargetMode} = \text{targetMode} \land)</td>
</tr>
<tr>
<td></td>
<td>(\text{newCurrentMode} = \text{currentMode}) \lor)</td>
</tr>
<tr>
<td></td>
<td>(\text{newTargetMode} &lt; \text{targetMode} \land)</td>
</tr>
<tr>
<td></td>
<td>(\text{newCurrentMode} = \text{targetMode}))</td>
</tr>
<tr>
<td>then</td>
<td></td>
</tr>
<tr>
<td>act1:</td>
<td>unitStates(unit) := \text{FUnitConf}(targetMode \mapsto unit)</td>
</tr>
<tr>
<td>event reconfFinish (\equiv) refines reconf</td>
<td></td>
</tr>
<tr>
<td>any unit</td>
<td></td>
</tr>
<tr>
<td>where</td>
<td></td>
</tr>
<tr>
<td>grd1:</td>
<td>currentMode \neq targetMode</td>
</tr>
<tr>
<td>grd2:</td>
<td>unit (\in) UNIT</td>
</tr>
<tr>
<td>grd3:</td>
<td>unitStates(unit) \neq \text{FUnitConf}(targetMode \mapsto unit)</td>
</tr>
<tr>
<td>then</td>
<td></td>
</tr>
<tr>
<td>act1:</td>
<td>unitStates(unit) := \text{FUnitConf}(targetMode \mapsto unit)</td>
</tr>
<tr>
<td>event targetMode newCurrentMode := newTargetMode</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>

Snippet 4 Variables and properties of M3

<table>
<thead>
<tr>
<th>variables</th>
<th>error units erroneousUnit</th>
</tr>
</thead>
<tbody>
<tr>
<td>invariant</td>
<td></td>
</tr>
<tr>
<td>inv1:</td>
<td>error (\in) ERROR</td>
</tr>
<tr>
<td>inv2:</td>
<td>units (\in) UNIT (\rightarrow) {0, 1, 2}</td>
</tr>
<tr>
<td>inv3:</td>
<td>erroneousUnit (\in) UNIT</td>
</tr>
<tr>
<td>inv4:</td>
<td>(\forall a \cdot a \in UNIT \land)</td>
</tr>
<tr>
<td></td>
<td>(\text{FUnitConf}(targetMode \mapsto a) &gt; 0 \Rightarrow unitStates(a) &gt; 0)</td>
</tr>
<tr>
<td>inv5:</td>
<td>error = UnitError (\Rightarrow) unitStates(erroneousUnit) (&gt; 0 \land)</td>
</tr>
<tr>
<td></td>
<td>unitStates(erroneousUnit) (&gt; 0)</td>
</tr>
<tr>
<td>inv6:</td>
<td>(\forall a \cdot a \in UNIT \land unitStates(a) = 0 \Rightarrow)</td>
</tr>
<tr>
<td></td>
<td>(error \neq UnitError) \lor (erroneousUnit \neq a)</td>
</tr>
<tr>
<td>inv7:</td>
<td>targetMode (\leq) (\text{FMaximumMode}(\text{units}))</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>
maximal the required units must be available, and there exists a unit which is required for the next mode but is not available. axm3 is a helper axiom used for proofs regarding inv4 and inv7.

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. To trace that, we create a number of constant functions:

\[
FTransitionNewTarget, FFdirNewTarget, FStableAttitudeNewTarget \in MODE \rightarrow MODE
\]

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 an attitude error during the stable mode accordingly. However, the reaction of the system to the unit errors is more complex. Firstly, 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.

We have a set of events covering all the cases of error handling (we omit the details for brevity):

- redundantRecovery
- attitudeRecoveryDuringStable refines stable
- attitudeRecoveryDuringAdvance refines reconf
- attitudeRecoveryDuringDowngrade refines reconf
- unitRecoveryDuringStable refines stable
- unitRecoveryDuringAdvance refines reconf
- unitRecoveryDuringReconf refines reconf (this one is for downgrade)

The function \( FMaximumMode \) is used in the unit recovery events to obtain a correct mode. The attitude recovery events set the new target mode according to the constant functions defined above. These do not contradict the invariants inv4 and inv7 as the values of the functions are always less than their arguments, and the units in a lesser mode are always available in absence of unit errors. The events representing the reactions of the system to the unit errors have the following in their guards:

\[
newMode \leq FMaximumMode(\text{units} \iff \{\text{erroneousUnit} \rightarrow 0\})
\]

where erroneousUnit contains the failed unit. The guard ensures that the units required by the target mode are available.

Note that on this level we abstractly define attitude and unit errors and recoveries, but do not specify exact units and transitions.

5) M4 and an autonomous reconfiguration scenario: The next model instantiates M3 with specific units, configurations, and mode transitions according to the specified AOCS sequential scenario which respects the unit management rules. The purpose of this step is to check the specific transition scenario against the required abstract behaviour, mode diagram, and a view of degraded modes.

On this level, we define the other five modes of the AOCS (OFF mode is defined in M0) and seven units with their possible states. Also there are a number of axioms defined for the specific configurations given by the requirements. For instance, in the Preparation mode the payload instrument must be in the Standby unit state:

\[
FUnitConf(\text{PREPARATION} \rightarrow \text{PLI}) = \text{PliStandby}
\]

We instantiate the abstract attitude recovery transition functions declared on the previous step with specific values, e.g.:

\[
FTransitionNewTarget(\text{PREPARATION}) = \text{SAFE}
\]

The modal view on the system contains six modes (Fig. 5). The three modes Off, Standby and Safe refine the Safe mode of the first modal diagram (Fig. 3), the other three modes refine Science. Such refinement relation is shown by proving REF_A and REF_G obligations [7]. The assumptions and guarantees are of a similar form to those on the first modal view (e.g. \( \text{currentMode} = \text{STANDBY} \)) and are easily proved by provers:

\[
\text{SAFE} = \{\text{OFF, STANDBY, SAFE}\}
\]

\[
\text{currentMode} \in \text{SAFE} \quad \text{(REF_A)}
\]

The mapping to events is significantly large because of the higher number of events representing different functional and fault tolerance behaviour. The transitions are mapped to those events which change the value of currentMode as the result of reconfiguration:

- Advance:
  - reconfFinish
- attitudeRecoveryDuringAdvance
- unitRecoveryDuringAdvance
- Downgrade:
  - reconfFinish
The downgrade transitions leading to Off mode are mapped to a single event reconfFinish. This is due to inability of the system to downgrade further during reboot.

A specific view on a model is one way to communicate design decision taken in a formal model to domain experts. We build another view to explain when 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. 6).

Each mode on the diagram represents the maximum (according to the ordering sequence defined above) mode of 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.

The assumption of each mode in this view specifies the availability of units, i.e. operating conditions. The guarantee states the maximum mode in which the system can be during operation, i.e. specifies the functionality under given assumption. For example, A/G of the Nominal mode is the following:

- Assumption: all units except PLI are available:
  $\text{units}(PLI) = 0 \land (\forall u \in UNIT \setminus \{PLI\} \land \text{units}(u) > 0)$
- Guarantee: the system can progress up to the Nominal mode:
  $F_{\text{MaximumMode}}(\text{units'}) = \text{NOMINAL}$

The transitions at Fig. 6 are mapped to the events which change the availability status of units (variable \text{units}); unitRecoveryDuringStable, unitRecoveryDuringReconf, unitRecoveryDuringAdvance.

6) PLI unit instantiation: At the last 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. 7).

There are three modes for a nominal PLI unit and three more for the spare unit. 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. When an error is detected in the redundant unit, the PLI unit becomes unavailable. There are neither transitions between the Off modes, nor error originates from them - this is due...
to the fact that no error can arise in a non-working unit. The assumption/guarantees concern only the current status and the 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 statuses and availability, it is sufficient to use the constant definitions and typing invariants to prove such obligation.

Overall, the modelling effort resulted in 650 proof obligations of which 500 were discharged automatically. Approximately 70% of proof obligations are concerned with the Mode/FT Views consistency and Event-B link. It is important that 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 demonstrate in detail the Mode/FT approach. A version of the model can be obtained from [7].

V. DISCUSSION AND LESSONS LEARNT

As the case study demonstrates, the Mode/FT Views approach requires extra proofs and therefore may affect the platform performance and result in extra manual efforts. On the other side, the explicitness, separation of concerns, and potential improvement in traceability will increase the development quality and improve the communication between modellers and requirements engineers. Even with this medium-scale study we have found it is worthwhile to use such approach for modelling modal systems which need to meet various fault tolerance requirements. In particular, tracing such requirements to views gives a better understanding of the system behaviour and leaves the models uncluttered. Additional POs often point to the inconsistencies in the models which otherwise would be difficult to cover using safety invariants.

While most of the proof obligations can be proved automatically or require little effort, some may need significant time to prove. The latter concern the whole set of modes or events, or many-to-many mappings between modes and events. For instance, the PO COVER has a disjunction of all mode assumptions in its goal. This can be comprehensible by a user, but automatic provers practically never can prove it without manual efforts. Other examples are EVT_A and ENBL proof obligations which also have disjunctions of mode assumptions and event guards in their goals correspondingly. FT views give more benefits when targeting features orthogonal to the behavioural part of the formal model, such as two modal views and a view of degraded modes in the case study. On the views, most of the events are mapped to each mode and therefore contribute to the level of complexity of the mentioned proof obligations.

However, the views orthogonality to the main, functional Event-B models and to each other gives extra flexibility in choosing the perspective of the view on the model. Changing the assumption/guarantee pairs leads to a different view and provides with a powerful tool for covering different 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 modal behaviour or different FT features.

One of the specific views that we produced as a result of our experimental development is dedicated to degraded modes of the system. It is not only beneficial but also essential to separate different viewpoints to avoid "multiplication" of the diagram elements leading to exponential growth of a single model.

Note that the ability to formally develop several views has great potentials for verifying complex system properties capturing consistency of these views in the situations where these views overlap. We did not investigate this topic in the AOCS case study in detail - these issues will be further addressed in our future work.

Another interesting benefit of using the FT/Mode Views is the ability to use proof obligations as a means to check compliance of the static definitions to the required abstract behaviour. In the case study, we firstly modelled the required reconfiguration behaviour of the system as a series of abstract unit switches restricted by rules. The actual rules and scenario of reconfiguration (modes and units mappings) have been defined later and proved to satisfy the abstract behaviour by representing the former in views. To achieve that using the traditional Event-B refinement, one would need to create extra events for each mode and unit transition, thus complicating further refinement steps.

The views developed in the case study are equivalent to informal diagrams used as the requirements documents for the AOCS.
VI. RELATED WORK

In [11] 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 [1] 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 [12]. 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 [6], [8] where we provided the theory of modes and a sound link to the state-based formalisms, exemplified by Event-B. Other related works include [13] 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 [14] 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.

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 and reviewers for their suggestions for improving this work.

REFERENCES