Developing Mode-Rich Satellite Software by Refinement in Event-B

Alexei Iliasov[a], Elena Troubitsyna[b], Linas Laibinis[b,*], Alexander Romanovsky[a], Kimmo Varpaaniemi[c], Dubravka Ilic[c], Timo Latvala[c]

[a] *Newcastle University, UK*
[b] *Åbo Akademi University, Finland*
[c] *Space Systems Finland*

Abstract

One of the guarantees that the designers of on-board satellite systems need to provide, so as to ensure their dependability, is that the mode transition scheme is implemented correctly, i.e. that the states of system components are consistent with the global system mode.There is still, however, a lack of scalable approaches to developing and verifying systems with complex mode transitions. This paper presents an approach to formal development of mode-rich systems by refinement in Event-B. We formalise the concepts of modes and mode transitions as well as deriving specification and refinement patterns which support correct-by-construction system development. The proposed approach is validated by a formal development of the Attitude and Orbit Control System (AOCS) undertaken within the ICT DEPLOY project. The experience gained in the course of developing of as complex an industrial-size system as AOCS shows that refinement in Event-B provides the engineers with a scalable formal technique that enables both mode-rich systems development and proof-based verification of their mode consistency.

Keywords: Mode consistency, Event-B, refinement, components, formal verification, on-board software

1. Introduction

Operational modes, mutually exclusive sets of the system behaviour [1], are a useful structuring concept that facilitates the design of complex systems in different industrial sectors. There are several well-known problems associated

*Corresponding author
Email addresses: alexei.iliasov@ncl.ac.uk (Alexei Iliasov), elena.troubitsyna@abo.fi (Elena Troubitsyna), linas.laibinis@abo.fi (Linas Laibinis), alexander.romanovsky@ncl.ac.uk (Alexander Romanovsky), Kimmo.Varpaaniemi@ssf.fi (Kimmo Varpaaniemi), Dubravka.Ilic@ssf.fi (Dubravka Ilic), Timo.Latvala@ssf.fi (Timo Latvala)

with mode-rich systems, e.g., correctness of complex mode transitions, mode consistency in distributed systems, mode confusion, etc. Developers of mode-rich systems clearly need generic scalable approaches that would help them solve these notoriously difficult problems.

This paper formalises reasoning about mode consistency and proposes a rigorous approach to developing complex mode-rich systems by adopting the top-down correct-by-construction development paradigm. The Event-B framework [2, 3] (extended with modularisation capabilities [4]) is used as the modelling language, while the Rodin platform [5] and its modularisation plug-in [6] provide an automated modelling and verification environment.

The proposed formalisation of mode consistency properties makes it possible to derive a generic pattern for specifying components of mode-rich systems in Event-B. This pattern defines a generic module interface, which can be instantiated by component-specific data and behaviour during system refinement. The proposed development process allows the designers to develop a system in a layered fashion. Essentially, it formalises gradual unfolding of system architectural layers by refinement. We prove consistency between mode transitions on adjacent architectural layers as part of refinement verification.

The development approach described in this paper generalises the results of the development of satellite Attitude and Orbit Control Systems (AOCS) [7] that has been undertaken by Space Systems Finland within the EU project DEPLOY [8]. AOCS [9] is a generic component of satellite onboard software. It is a typical example of a mode-rich system with a complex mode transition scheme. There are two distinctive characteristics that make AOCS development and verification challenging. The first characteristic is long running (i.e., non-instantaneous) mode transitions that are caused by slow dynamics of the controlled electro-mechanical components. The second one is integration of error recovery with the mode transition scheme, i.e., error recovery is implemented as rollbacking to certain degraded modes. Together, these two features may lead to cascading mode transitions, i.e., the situations when a system transition to one mode is preempted by a transition to another (degraded) mode due to failure occurrence(s). It has been noted that testing and model checking of the systems with such cascading mode transitions is difficult and suffers from poor scalability [10].

Our approach supports incremental verification of global mode consistency properties by proof and demonstrates a good scalability. It allows us to cope efficiently with complexity of AOCS. We argue that the AOCS development presented in this paper is a successful experiment in formal refinement-based development of a complex industrial size system. Hence we believe that the proposed approach provides the designers with a scalable formal technique for developing and verifying complex mode-rich systems.

The structure of the paper is as follows. Section 2 introduces our formal development framework – Event-B. We continue by presenting our formalised reasoning about modes and mode transitions in Section 3. A formal pattern for developing layered mode-rich systems is proposed in Section 4. Section 5 presents the case study – development of an Attitude and Orbit Control System

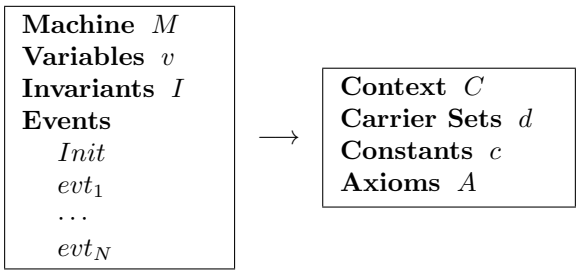

Figure 1: Event-B machine and context components

(AOCS). We summarise our lessons learnt in Section 6, while Section 7 overviews the related work. Finally, some concluding remarks are given in Section 8.

2. Event-B

We start by briefly describing our development framework. The B Method [11] is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. The Event-B formalism [2, 3] is a specialisation of the B Method. It enables modelling of event-based (reactive) systems by incorporating the ideas of the Action Systems formalism [12] into the B Method.

2.1. Modelling and Refinement in Event-B

In Event-B, a system specification (model) is defined using the notion of an *abstract state machine* [3]. An abstract state machine encapsulates the model state, represented as a collection of model variables, and defines operations on this state. Therefore, it describes the dynamic part (behaviour) of the modelled system. Usually a machine also has the accompanying component, called *context*, which contains the static part of the model. In particular, a context can include user-defined carrier sets, constants and their properties, which are given as a list of model axioms. A general form of Event-B models is given in Figure 1.

The machine is uniquely identified by its name M. The state variables, v, are declared in the **Variables** clause and initialised in the *Init* event. The variables are strongly typed by the constraining predicates I given in the **Invariants** clause. The invariant clause might also contain other predicates defining properties that should be preserved during system execution.

The dynamic behaviour of the system is defined by the set of atomic events specified in the **Events** clause. Generally, an event can be defined as follows:

ANY lv **WHERE** g **THEN** S **END**

where lv is a list of new local variables (parameters), the guard g is a state predicate, and the action S is a statement (assignment). In case when lv is

Action	$BA(x, y, x')$
skip	$x' = x \ \wedge \ y' = y$
$x := E(x, y)$	$x' = E(x, y) \ \wedge \ y' = y$
$x :\in Set$	$x' \in Set \ \wedge \ y' = y$
$x :\mid P(x, y, x')$	$P(x, y, x') \ \wedge \ y' = y$

Figure 2: Before-after predicates

empty, the event syntax becomes **WHEN** g **THEN** S **END**. If g is always true, the syntax can be further simplified to **BEGIN** S **END**.

The occurrence of events represents the observable behaviour of the system. The guard defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically. If none of the events is enabled then the system deadlocks.

In general, the action of an event is a parallel composition of assignments. The assignments can be either deterministic or non-deterministic. A deterministic assignment, $x := E(x, y)$, has the standard syntax and meaning. A nondeterministic assignment is denoted either as $x :\in Set$, where Set is a set of values, or $x :\mid P(x, y, x')$, where P is a predicate relating initial values of x and y to some final value of x'. As a result of such a non-deterministic assignment, x can get any value belonging to Set or according to P.

Semantics of an Abstract Model. The semantics of Event-B actions is defined using before-after (BA) predicates [2, 3]. A before-after predicate describes a relationship between the system states before and after an execution of an event action, as shown in Figure 2. Here x and y are disjoint lists (partitions) of state variables, and x', y' represent their values in the state after the action execution – the after-state. The notion of BA predicate can be easily generalised to formally define model events. For an event e of the form **ANY** lv **WHERE** g **THEN** S **END**, its BA predicate is as follows:

$$BA_e(x, y, x') \quad = \quad \exists lv.\ g(lv, x, y) \ \wedge \ BA_S(x, y, lv, x')$$

The semantics of a whole Event-B model is formulated as a number of *proof obligations*, expressed in the form of logical sequents. Below we present only the most important proof obligations that should be verified (proved) for the initial and refined models. The full list of proof obligations can be found in [2].

The initial Event-B model should satisfy the event feasibility and invariant preservation properties. For each event of the model – evt_i – its feasibility means that, whenever the event is enabled, its before-after predicate (BA) is well-defined, i.e., exists some reachable after-state:

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

where A are model axioms, I is the model invariant, g_i is the event guard, d are model sets, c are model constants, and v, v' are the variable values before and after event execution.

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

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

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

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

Semantics of a Refined Model. Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract system specification that models the most essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into the abstract specification. These new events correspond to stuttering steps that are not visible at the abstract level.

The variables of a more abstract model in the refinement chain are called the *abstract variables*, whereas the variables of the next refined model are called the *concrete variables*.[1] Event-B formal development supports data refinement, allowing us to replace some abstract variables with their concrete counterparts. In that case, the invariant of the refined machine formally defines the relationship between the abstract and the concrete variables.

To verify correctness of a refinement step, we need to prove a number of proof obligations for the refined model. Intuitively, those proof obligations allow us to demonstrate that the refined machine does not introduce new observable behaviour, more specifically, that concrete states are linked to the abstract ones via the given (gluing) invariant of the refined model. In general, these proofs guarantee that the concrete model adheres to the abstract one, thus all proved properties of the abstract model are automatically "inherited" by the refined one.

For brevity, below we will show only the most essential proof obligations. These are those proof obligations that go beyond checking well-definedness of basic model elements, which are usually automatically discharged by the tool support of Event-B.

Let us first introduce a shorthand $H(d,c,v,w)$ that collectively stands for the hypotheses $A(d,c),\ I(d,c,v),\ I'(d,c,v,w)$, where I and v are the invariant and variables of the abstract machine, while I' and w are correspondingly the invariant and variables of the refined machine. Then the feasibility refinement property for an event evt_i of a refined model can be expressed as follows:

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

[1] Here the Event-B terminology differs from the that of Classical B. In Classical B, the abstract variables are those that cannot be implemented, while the concrete variables are correspondingly those that can be implemented.

where g_i' is the refined guard and BA_i' is a before-after predicate of the refined event.

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

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

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

Finally, the *simulation* proof obligation requires to show that the "execution" of a refined event is not contradictory with its abstract version:

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

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

The Event-B refinement process allows us to gradually introduce implementation details, while preserving functional correctness during stepwise model transformation. The model verification effort, in particular, automatic generation and proving of the required proof obligations, is significantly facilitated by the provided tool support – the Rodin platform [5].

2.2. Modelling Modular Systems in Event-B

Recently the Event-B language and tool support have been extended with a possibility to define modules [4, 6] – components containing groups of callable operations. Modules can have their own (external and internal) state and the invariant properties. The important characteristic of modules is that they can be developed separately and then composed with the main system.

Module Structure. A module description consists of two parts – *module interface* and *module body*. Let M be a module. The module interface is a separate Event-B component. It allows the user of the module M to invoke its operations and observe the external variables of M without having to inspect the module implementation details. The module interface consists of the module interface description MI and its context MI_Context. The context defines the required constants c and sets s. The interface description consists, respectively, of the external module variables w, the external module invariant M_Inv(c, s, w), and a collection of module operations, characterised by their pre- and postconditions, as shown in Figure 3. The primed variables in the operation postcondition stand for the variable values after operation execution.

A module development always starts with the design of an interface. After an interface is formulated, it cannot be altered in any manner. This ensures correct relationships between a module interface and its body, i.e., that the specification of an operation call is recomposable with an operation implementation. A module body is an Event-B machine. It implements each operation described in the module interface by a separate group of events. Additional proof obligations are generated to verify correctness of a module. They guarantee that each event group faithfully satisfies the given pre- and postconditions of the corresponding interface operation.

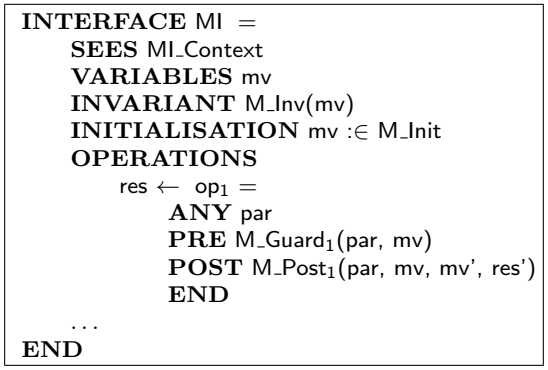

Figure 3: Structure of interface component

Importing of a Module. When the module M is imported into another Event-B machine, this is specified by a special clause **USES** in the importing machine, N. As a result, the machine N can invoke the operations of M as well as read the external variables of M listed in the interface MI.

To make a module interface generic, in MI_Context we can define some abstract constants and sets (types). Such data structures become module parameters that can be instantiated when a module is imported. The concrete values or constraints needed for module instantiation are supplied within the **USES** clause of the importing machine. Alternatively, the module interface can be *extended* with new sets, constants, and the properties that define new data structures and/or constrain the old ones. Such an extension produces a new, more concrete module interface. Via different instantiation of generic parameters the designers can easily accommodate the required variations when developing components with similar functionality. Hence module instantiation provides us with a powerful mechanism for reuse.

We can create several instances of the given module and import them into the same machine. Different instances of a module operate on disjoint state spaces. Identifier prefixes can be supplied in the **USES** clause to distinguish the variables and the operations of different module instances or those of the importing machine and the imported module. The syntax of USES then becomes as follows:

$$\textbf{USES} \ < \textit{module interface} > \ \textbf{with prefix} \ < \textit{prefix\_} >$$

Semantics of Module Interface. Similarly to a machine component, the semantics of an interface component is defined by the following proof obligations. The module initialisation must establish the module invariant M_Inv:

$$M\_Init(mv) \ \vdash \ M\_Inv(mv) \qquad \text{(MOD\_INIT)}$$

Let us assume $Oper_i$, $i \in 1..N$, is one of module operations. The module

invariant M_Inv should be preserved by each operation execution:

$$M\_Inv(mv),\ Pre_i(pars, mv),\ Post_i(pars, mv, mv', res') \ \vdash\ M\_Inv(mv') \quad \text{(MOD\_INV)}$$

where Pre_i and $Post_i$ are respectfully the precondition and the postcondition of $Oper_i$.

Finally, there is a couple of feasibility proof obligations for each $Oper_i,\ i \in 1..N$. Firstly, the operation precondition should be true for at least some of parameter values:

$$M\_Inv(mv) \ \vdash\ \exists pars.\ Pre_i(pars, mv) \quad \text{(MOD\_PARS)}$$

Secondly, at least some operation post-state containing the required result must be reachable:

$$M\_Inv(mv),\ Pre_i(pars, mv) \ \vdash\ \exists(mv', res').\ Post_i(pars, mv, mv', res') \quad \text{(MOD\_RES)}$$

Semantics of an Operation Call. A machine importing a module instance operates on the extended state consisting of its own variables v and module variables mv. The module state can be updated in event actions only via operations calls. The semantics of an event containing an operation call is as follows.

Let us consider the model event E that contains a call to the module operation Op with the given arguments $args$, i.e., it is of the form

$$\textbf{ANY}\ lvars\ \textbf{WHERE}\ g\ \textbf{THEN}\ S[Op(args)]\ \textbf{END}.$$

The BA predicate of such an event can be defined as follows:

$$\begin{aligned} BA_E(v, mv, v', mv') \quad = \quad & \exists(lvars, res, new\_mv).\ g(lvars, v, mv) \ \wedge \\ & Post(args, mv, new\_mv, res) \ \wedge \\ & BA_{S*}(lvars, v, mv, res, v') \ \wedge \ (mv' = new\_mv) \end{aligned}$$

where $S*$ is S with all the occurrences of $Op(args)$ replaced by res. Once this is done, we can rely on the existing proof semantics to verify the invariant preservation, the event simulation and other required properties.

Moreover, we need an additional proof obligations to ensure call correctness by checking that the operation precondition holds at the place of an operation call:

$$g(lv, v, mv),\ Inv(v, mv),\ M\_Inv(mv) \ \vdash\ Pre(args, mv) \quad \text{(CALL\_CORR)}$$

The modularisation extension of Event-B facilitates formal development of complex industrial-size systems by allowing the designers to decompose large specifications into separate components and verify system-level properties at the architectural level.

In the next section we rely on the modularisation extension while proposing our approach to reasoning about mode consistency. Then, in Section 4, we use this approach to formalise development of mode-rich systems in Event-B.

3. Formal Reasoning about Modes and Mode Transitions

3.1. Mode Logic in Layered Architectures

Leveson et al. [1] define *mode* as a mutually exclusive set of system behaviours. Essentially, a mode can be understood as an abstraction of the system state, i.e., it signifies the class of states associated with a certain system functionality. Functionality of mode-rich systems can be represented as a certain *scenario* defined in terms of system modes. A set of all the modes and mode transition rules defined by this scenario constitute the mode logic of the system [1].

We formalise mode logic as a special kind of a state transition system, i.e., as a triple $(Modes, Next, InitMode)$, where $Modes$ is a set of all possible modes of the system, $Next$ is a relation on $Modes$, containing all the allowed mode transitions, and $InitMode$ is the initial system mode.

In this formalisation $Next$ is a relation and hence it can contain several predefined scenarios. Sometimes $Next$ can be defined more precisely as an *ordering relation*. For instance, the predefined scenario of the AOCS system presented in Section 5 is a typical example of this. The scenario describes the sequence of modes from powering-on the instruments to bringing them into the mode that enables collection of valuable scientific data. For systems that implement that kind of scenarios, we can define the mode logic as $(Modes, \leq, InitMode)$, where $\leq$: $Modes \leftrightarrow Modes$ is a partial *ordering* on $Modes$. Here the fact that $Mode_1 \leq Mode_2$ means that in $Mode_2$ the system provides a richer set of functions than in $Mode_1$.[2]

The coarse-grained global modes allow us to represent the system-level mode logic as a process of instantaneous change from one mode to another. In reality, a mode transition may involve certain physical (e.g., electro-mechanical) processes in the controlled components and hence have a duration. Indeed, the system should perform several iterations of its control loop to bring all the involved components into the *consistent* states for entering the next target mode. To facilitate reasoning about consistency of component modes, in this paper we adopt the layered approach to architecting complex mode-rich systems. It is recognised that a layered architecture is advantageous in designing complex component-based systems [13] since it facilitates structuring the system behaviour according to the identified abstraction levels. A generalised tree representation of the architecture of a mode-rich system is given in Figure 4.

To correctly define the mode logic and guarantee that a layered system faithfully implements it, we need to ensure mode consistency of the components residing at different layers. For each architectural layer, we define the corresponding mode (or sub-mode) logic. Moreover, we introduce a typical component, called a *Mode Managing Component* (MMC), which is responsible for implementing a specific mode logic. MMCs can be present on different architectural layers. For

[2]Defining $Next$ as an ordering relation makes it a special case that does not affect the general approach presented next.

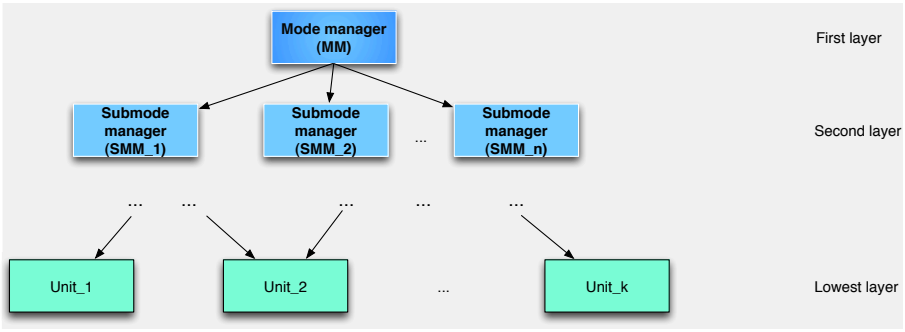

Figure 4: Architecture of mode-rich layered systems

example, in Figure 4 all the components that are above of the lowest layer may be mode managing components.

The mode logic on the system level is implemented by a MMC called *Mode Manager*. At the lower layers MMCs (denoted as *Submode Managers*) implement specific sub-mode logics defined for the corresponding layers. At the layer above the bottom one, there reside MMCs that directly communicate with the lowest layer components (often called *units*). They implement the corresponding control algorithms required to bring the units into the desirable states.

When a MMC chooses a new target mode, it initiates (sequentially or in parallel) the corresponding mode transitions in lower layer components. As a result, the affected Submode Managers at lower layers start to execute their own predefined scenarios. At each architectural layer, the MMC follow the same behavioural pattern: it monitors the states of lower layer components to detect when the conditions for entering the desired mode are satisfied, i.e., the lower layer components have successfully completed the required transitions to the corresponding submodes. At the same time, the MMC monitors the health of lower layer components. The errors detected by these components may prevent a successful transition to the target mode and, as a consequence, require to initiate the predefined recovery procedures.

To formalise those two aspects of the MMC behaviour, we introduce two functions, *Mode_ent_cond* and *Mode_error_handling*. The function *Mode_ent_cond* expresses the relationship between the target mode, the component state, and the corresponding modes of the monitored components on the lower layer. The second function, *Mode_error_handling*, describes the effect of all the errors detected by the monitored components on the MMC mode. Please note here the difference between the notions of the component state and a component mode. A component mode is an abstraction of the component state aggregated with the externally visible states of the monitored components, thus representing the class of such states associated with this mode.

Let *MState* be the component state. We define *Mode_ent_cond* as a function of the following type:

$$Mode\_ent\_cond \ : \ Modes \rightarrow \mathcal{P}_1(MState \times LocalModes_1 \times \cdots \times LocalModes_k) \quad (1)$$

where $LocalModes_1, .., LocalModes_k$ are modes of the monitored components.

For each (global) mode, the function returns a non-empty set of the allowed combinations of the component state and the monitored local modes. Here we assume that the local modes belong to the externally visible state of those components.[3]

The mode entry conditions can be recursively constructed throughout the entire architecture for each pair of a MMC and a mode. We also use *Mode_ent_cond* to determine which components are affected when a MMC initiates a new mode transition, i.e., to which components it should send the corresponding (local mode) transition requests.

As mentioned above, an execution of a mode transition usually takes several control cycles. At each cycle MMCs monitor the progress of the requested mode transitions. An occurrence of faults in the controlled low-level components might result in a failure to complete some requested mode transition. As a result, the corresponding MMC assesses the error and initiates error recovery either by itself or by propagating the error to a MMC on the higher level. In mode-rich systems, error recovery is often implemented as a rollback to some preceding (and usually more degraded) mode in the predefined scenario.

We define the function *Mode_error_handling* to model the mode transitions executed as error recovery

$$Mode\_error\_handling \ : \ MState \times LocalErrors_1 \times ... \times Local\ Errors_k \rightarrow \ Modes$$

where *MState* is the component state and $LocalErrors_1 ... Local\ Errors_k$ are all the errors detected by lower layer components. The function defines the mode to which the system should rollback to execute error recovery. The pair of the new target and current modes should belong to the transitive closure of *Next*.

While the system is recovering from one error, another error requiring a different mode transition might occur. Due to a large number of components and their failure modes, ensuring mode consistency becomes especially difficult. Hence we need to precisely define the mode consistency criteria for the layered control systems. Next we describe our approach to achieving this.

3.2. Properties of Mode-Rich Systems

To guarantee that the mode logic is unambiguous, we have to ensure that a component can be only in one mode at a time, i.e., the mode entry conditions for different modes cannot overlap:

$$\begin{aligned} \forall i, j \ \cdot \ m_i \in Modes \ \wedge \ m_j \in Modes \ \wedge \ i \neq j \ \ \Rightarrow \\ Mode\_ent\_cond(m_i) \cap Mode\_ent\_cond(m_j) = \varnothing \end{aligned} \tag{2}$$

Overall, the definition (1) and the property (2) postulate *mode consistency* conditions that should be guaranteed for each MMC of a system.

[3] In general, the definition of *Mode_ent_cond* could have been further generalised by replacing the local modes with the entire external states of the monitored components.

Let us now address another important issue in designing mode-rich systems – ensuring *mode invariants*. These are the properties that should be preserved while the system stays in some particular mode. However, in the systems where mode transitions take time and can be interrupted by errors, this is not a straightforward task. To tackle it, let us define the following attributes of a MMC:

- *last_mode* – signifies the last successfully reached MMC mode;
- *next_target* – signifies the target mode that a MMC is currently in transition to;
- *previous_target* – signifies the previous mode that a MMC was in transition to (though it has not necessarily reached it).

where *last_mode*, *next_target* and *previous_target* are component modes.

Collectively, these three attributes unambiguously describe the actual mode of a MMC. Based on them, we define the notion of component *status* that might be either *Stable, Decreasing* or *Increasing* as follows:

- $Stable \triangleq last\_mode = previous\_target \wedge next\_target = previous\_target$
  a MMC is maintaining the last successfully reached mode (i.e., there are no mode transition requests);
- $Increasing \triangleq last\_mode = previous\_target \wedge previous\_target < next\_target$
  a MMC is in transition to a next, more advanced mode;
- $Decreasing \triangleq next\_target < previous\_target$
  MMC stability or a mode transition to *previous_target* was interrupted (e.g., by error handling) by a new mode request to proceed to a more degraded mode.

A graphical diagram showing mode status changes is given in Figure 5.

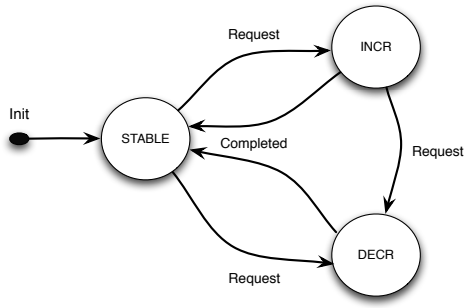

Figure 5: Component mode status

Here Request represents a mode change change request issued by the upper layers, while Completed represents the system state change when the target

mode is successfully reached, i.e., the corresponding mode entry condition is satisfied.

We assume that, when a mode transition is completed, the component status is changed to *Stable*. The Mode Manager (MMC at the top layer) will maintain this status only if the final mode or modes of the scenario (defined by *Next*) are reached. On the lower layers, Submode Managers will maintain their stability until receiving a request for a new mode transition. In its stable state, Mode Manager would change its status to *Increasing* to execute the next step of the mode scenario, which in turn would trigger the corresponding mode transitions in the lower layer MMCs. Irrespectively of the component status, an occurrence of an error would result in changing it to *Decreasing* that designates a rollback in the predefined scenario.

Now we can formally connect the mode status and a mode invariant. When a mode manager is stable, the mode entry condition is a mode invariant, i.e.,

$$Stable \;\Rightarrow\; (s, l_1, ..., l_k) \in Mode\_ent\_cond(last\_mode)$$

where $s \in MState$ is the current state, and $l_1, ..., l_k$ are the local modes.
The other mode invariants are also preserved when a component is stable:

$$Stable \;\Rightarrow\; Mode\_Inv(last\_mode)$$

Hence, in general, mode invariant properties are not preserved while MMC is engaged in a mode transition.

In a layered system, the stability property can actually be recursively unfolded downwards:

$$Stable(m) \quad \Rightarrow \quad Stable(lm_1) \;\wedge\; Stable(lm_2) \;\wedge \cdots \wedge\; Stable(lm_k)$$

where $m \in Modes$ and $(lm_1, ..., lm_k) \in LocalModes_1 \times ... \times LocalModes_k$ are the monitored local modes that the mode m depends on. This property can be proved as an invariant property preserved by the each mode managing components at certain points of its execution, specifically, after it had evaluated situation (i.e., the mode changes and the detected errors of the monitored components) and decided on its next course of action.

The discussion above sets the general guidelines for defining mode managers in layered mode-rich systems. While specifying a particular mode manager, we instantiate the abstract data structures *Modes*, *Next*, *Mode_ent_cond*, and *Mode_error_handling* and ensure that

R1 In a stable state, a MMC makes its decision to initiate a new mode transition to some more advanced mode according to the relation *Next*;

R2 In a transitional state, a MMC monitors the state of lower layer components. When *Mode_ent_cond*(*next_target*) becomes satisfied for the local state and the submodes of the monitored components, the mode manager completes the mode transition and becomes stable;

R3 In both stable and transitional states, a MMC monitors the lower layer components for the detected errors. When such errors occur, a MMC makes its decisions based on *Mode_error_handling*, which is applied to the mode manager state and all the detected errors.

Next we will show how these guidelines can be implemented in the proposed formal specification and development patterns.

4. Development Pattern

In this section we propose a formal pattern for developing layered mode-rich systems in the Event-B framework. It consists of two main parts: a generic interface of a MMC and a generic refinement step for unfolding architectural layers.

4.1. Generic Interface

As discussed earlier, the structure and behaviour of mode managing components at different layers are very similar. This suggests the idea of modelling a MMC as a generic module that can be adapted to different contexts by instantiating its generic parameters.

In Event-B, we can formalise this by first creating a generic module interface that can be later implemented in different ways, thus creating implementations of specific mode managers. The proposed interface (see Figure 6) contains four operations that can be called by a MMC from a higher layer. It also defines the external module variables that are visible from a calling component.

The external state of a component is formed by four variables – *last*, *next*, *prev* and *error*. The first three variables define the component mode status, i.e., they represent the respective MMC attributes *last_mode*, *next_target* and *previous_target*, formally introduced in Section 3. These variables describe the actual mode of a component and also the mode transition dynamics. Based on their values, it is possible to find out whether the component has settled in a stable mode ($last = prev \wedge next = prev$), is working towards a more advanced mode ($last = prev \wedge prev \mapsto next \in Next$), or is degrading its mode due to error recovery ($prev \mapsto next \in Next^{-1}$).

The variable *error* abstractly models the errors currently detected by a MMC. The variable types MODE and ERROR are defined in the interface context component MMC_Context, described in detail below.

The operation ToMode can be called by an upper layer component to set a new target mode. The operation ResetError is called to clear the raised error flag after the detected errors are handled by an upper layer component (e.g., by initiating the appropriate error recovery). Since the behaviour of the overall system is cyclic, we assume that within each single cycle the control is passed

```
INTERFACE MMC_I
  VARIABLES last, prev, next, error
  SEES MMC_Context
  INVARIANT
      inv1 :  last ∈ MODE ∧ next ∈ MODE ∧ prev ∈ MODE ∧ error ∈ ERROR
      inv2 :  next = prev  ⇒  next = last
      inv3 :  next ≠ prev  ⇒  next ↦ prev ∈ Next  ∨  prev ↦ next ∈ Next
      inv4 :  {last ↦ prev, last ↦ next} ⊆ Next ∪ Next^−1
  INITIALISATION
      last, prev, next := InitMode, InitMode, InitMode
      error := NoError
  OPERATIONS
      r ← ToMode          =   ANY m PRE
                                  error = NoError ∧ m ∈ MODE
                                  m ≠ next ∧ m ↦ next ∈ Next ∪ Next^−1
                              POST
                                  r' = last  ∧  prev' = next  ∧  next' = m
                              END
      r ← ResetError      =   PRE
                                  error ≠ NoError
                              POST
                                  r' = last  ∧  error' = NoError
                              END
      r ← Mode_Advance    =   PRE
                                  next = prev ∧ error = NoError
                              POST
                                  r' = last  ∧  error' ∈ ERROR  ∧  prev ↦ next' ∈ Next
                              END
      r ← Continuation    =   PRE
                                  next ≠ prev ∧ error = NoError
                              POST
                                  r' = last'  ∧  error' ∈ ERROR ∧
                                  last' ↦ next ∈ Next ∪ Next^−1 ∧
                                  ((last' ≠ next ∧ prev' = prev) ∨ (last' = next ∧ prev' = last'))
                              END
END
```

Figure 6: Interface of a generic mode manager

through the entire hierarchy of components[4]. The operations Mode_Advance and Continuation model the component behaviour when it receives the control while being correspondingly in a stable or a transitional state.

In the operation Mode_Advance, the component might start a new forward mode transition by setting a new target mode. In practice, such situations are often governed by the autonomous mode scenario provided for the component. In the operation Continuation, the component is still in mode transition or just finished its latest forward or backward mode transition. These two distinct cases are covered by two different disjuncts in the operation postcondition.

The interface context MMC_Context (see Figure 7) defines generic sets and constants of the pattern, which contribute to abstract characterisation of the mode logic. In addition to the deferred sets MODE and ERROR mentioned

[4]Our case study AOCS is a cyclic system. For general modelling and verification of mode-rich systems this assumption is not essential.

CONTEXT MMC_Context
CONSTANTS MODE, InitMode, $Next$, ERROR, NoError
AXIOMS
axm1 : InitMode $\in$ MODE
axm2 : $Next \in$ MODE $\leftrightarrow$ MODE
axm3 : dom($Next$) $\cup$ ran($Next$) = MODE
axm4 : id $\subseteq Next$
axm5 : $Next \cap Next^{-1} \subseteq$ id
axm6 : $Next; Next \subseteq Next$
axm7 : NoError $\in$ ERROR
axm8 : ERROR $\setminus$ NoError $\neq \varnothing$
END

Figure 7: Context of generic interface

above, MMC_Context introduces the constant InitMode representing the predefined initial mode, the relation *Next* containing all the allowed mode transitions, and the constant NoError, a special value denoting the absence of errors. All these structures should be instantiated with concrete data when a module instance is created. If *Next* is a partial order, its required reflexivity, antisymmetry and transitivity properties (i.e., the axioms axm4–axm6) are also checked during instantiation. Otherwise, they can be omitted.

Let us note that the functions *Mode_ent_cond* and *Mode_error_handling* from Section 3 are not introduced in the MMC generic interface. These important functions are defined in terms of the component state and the modes of lower layer components, i.e., in terms of two adjacent layers. Therefore, they can be defined only during a refinement step that integrates the lower layer MMCs into the refined specification of a MMC at a higher layer.

In a module body that implements the presented generic interface, each of interface operations is represented (essentially refined) by a corresponding group of Event-B events. The body might also contain additional module-specific data structures, local variables and operations. However, any refinement of a module body is proved to be a refinement of the initial specification defined in the interface.

4.2. Refinement Strategy

In general, refinement process aims at introducing implementation details into an abstract system specification. However, in this paper we demonstrate that refinement can also be used to incrementally build a system architecture. This is especially well-suited for layered control systems, where refinement can be used to gradually unfold system layers by using the predefined specification and refinement patterns [14]. Indeed, the generic interface *MMC_I* that we described above can be seen as an abstract representation of the top level interface of a mode-rich system. Yet it can also be seen as an interface of any submode manager at a lower layer. Therefore, by instantiating *MMC_I* with the mode logic specific for a particular MMC, we can obtain a MMC of any layer. Hence

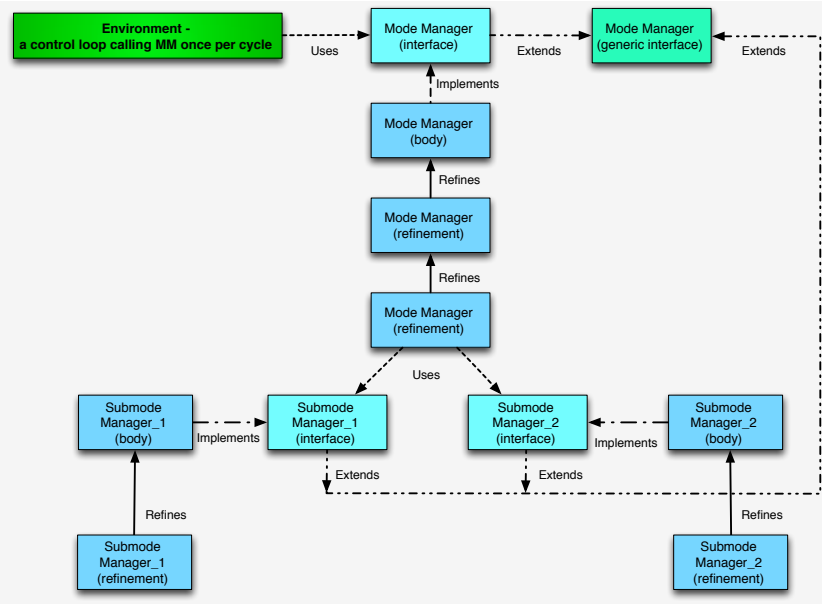

Figure 8: Development hierarchy

our development strategy can be seen as a process of introducing specific module types into an Event-B development, as shown in Figure 8.

We assume that the system executes cyclically, with the environment periodically invoking the top MMC. In its turn, it calls lower layer MMCs. This behaviour is recursively repeated throughout the hierarchy.

The refinement process starts by instantiating the top level mode manager interface with the global mode logic. The body of the obtained mode manager can be further developed by refinement. This is similar to building a normal refinement chain although the starting point is an interface rather than an abstract machine. At some point of our development, a number of the lower layer MMCs that the mode manager controls are introduced. This refinement step essentially introduces calls to the corresponding interface operations of these MMCs.

At the same time, the submodes and errors of the lower layer become visible for the mode manager. To properly integrate new components into the development hierarchy, we need to extend the context of the mode manager with the data structures (such as *Mode_ent_cond* and *Mode_error_handling*) describing the required consistency of mode transitions and error handling on two adjacent levels. This allows us to refine the mode manager operations as well as define the mode consistency conditions as additional invariants of the form

$$Stable \Rightarrow Mode\_ent\_cond(last\_mode)$$

that are verified in this refinement step. In a similar way we handle errors of

new components.

On the architectural level, such a refinement step corresponds to unfolding one more layer of the system hierarchy. From this point, we can focus on refining bodies of the introduced MMCs. These bodies would implement their own mode logics and also, if needed, call operations of the MMCs residing on the layer below. Hence we follow the same refinement pattern as before, unfolding the system architectural layers until the entire hierarchy is built.

The main strength of our development is that we ensure global mode consistency by simply conjuncting the mode linking conditions introduced at each level. Hence, despite a strict hierarchical structure, there is a simple procedure for enforcing conformance of mode changes for any two or more components of a system. We avoid reasoning about the entire global mode consistency and instead enforce by refinement the mode consistency between any two adjacent layers.

Our approach allows us to design a layered mode-rich system in a disciplined structured way. It makes a smooth transition from architectural modelling to component implementation, yet ensuring the overall mode consistency. This approach generalises our experience in developing AOCS [9], presented next.

5. Attitude and Orbit Control System

The Attitude and Orbit Control System (AOCS) is a generic component of satellite onboard software. The main function of AOCS 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. AOCS consists of seven physical units: four sensors, two actuators and the payload instrument. An optimal attitude is required to support the needs of payload instruments and to fulfil the mission of the satellite.

5.1. Abstract Model

While positioning the satellite, AOCS goes through a succession of stages – modes. When the satellite achieves the required attitude, i.e., enters so called *Science mode*, the system activates payload instruments. Hence, we can identify two main phases in the system behaviour:

- *preparation* phase – achieving the required satellite attitude,
- *activation* phase – activation and operation of scientific instruments.

In the abstract specification we represent each phase by the corresponding events. The preparation phase is abstractly specified in the event preparation. The system is in the preparation stage when the boolean flag *prep* is equal to *FALSE*. It can take more than one atomic step to complete this phase, which is expressed by a non-deterministic assignment $prep :\in BOOL$. When the preparation is finished, the system can start the activation phase that is designated by

MACHINE aocs
VARIABLES $prep, act, aocs\_error$
INVARIANT
$prep \in BOOL \wedge act \in BOOL \wedge aocs\_err \in BOOL$
$prep = FALSE \Rightarrow act = FALSE$
$aocs\_error = TRUE \Rightarrow prep = FALSE \wedge act = FALSE$
INITIALISATION
$prep, act, aocs\_error := FALSE, FALSE, FALSE$
EVENTS

preparation	=	WHEN
		$aocs\_error = FALSE \wedge prep = FALSE$
		THEN
		$prep :\in BOOL$
		END
activation	=	WHEN
		$aocs\_error = FALSE \wedge prep = TRUE \wedge act = FALSE$
		THEN
		$act := TRUE$
		END
activity	=	WHEN
		$act = TRUE$
		THEN
		$skip$
		END
recovery	=	WHEN
		$aocs\_error = FALSE$
		THEN
		$prep, act := FALSE, FALSE$
		END
error	=	BEGIN
		$aocs\_error, prep, act := TRUE, FALSE, FALSE$
		END

END

the guard $prep = TRUE \wedge act = FALSE$ of the event activation. When the scientific payload instruments become fully functional, the event activity becomes enabled, i.e., the flag act becomes $TRUE$.

The overall behaviour of AOCS is cyclic. At each cycle it reads the sensor data and controls the actuators to achieve the desired system behaviour. However, hardware faults as well as disturbances in the environment may alter the attitude of the satellite. Hence, upon the occurrence of such undesirable events, the activity phase should be interrupted and the preparation phase restarted to adjust the attitude. This is modelled by the event recovery. It is implicitly assumed that most errors are recoverable and the targeted attitude can be reacquired. However, some errors might be unrecoverable and require the reset of the underlying hardware platform. The occurrence of a non-recoverable error is modelled in the event error that becomes enabled when such an error is detected, i.e., when $aocs\_error$ becomes $TRUE$. This event puts the system in a permanently deadlocked state.

The abstract system model is deliberately simple. It is merely a state transition system that depicts the high-level behaviour of AOCS in a clear and concise

way. Two distinct phases of the system behaviour – preparation and activation – suggest the idea of carrying the further system development in two independent strands. The first strand focuses on specifying mode transitions in the preparation phase. This strand refines the functionality abstractly represented by the events preparation, recovery and error. The second strand focuses on specifying the behaviour of scientific payload instruments in the activation phase, i.e., refines the events activation and activity. In this paper we describe in detail the first strand, i.e., the formal development of the preparation subsystem of AOCS.

To obtain two independent developments, we decompose by refinement the overall AOCS specification into the top-level component (a refinement of the aocs machine) and the subsystem (module) Mode Manager that is in charge of the preparation and recovery activities of the abstract AOCS model. Before presenting details of the decomposition refinement step, let us now briefly explain how the model of Mode Manager is created.

5.2. Mode Manager

Mode Manager is a control system with its own set of modes and the mode transition scenario. This scenario defines the steps needed to acquire the optimal satellite attitude, i.e., to achieve the *Science* mode. The interface of Mode Manager is a result of instantiating the generic module interface MMC_I (defined in Figures 6 and 7) with satellite-specific data. More specifically, the context of Mode Manager constrains the abstract data structures of MMC_Context with the values specific to the top-level of AOCS. This includes concrete definitions of the set of modes MODE, the constant InitMode, and the relation *Next*. The resulting interface ModeManager relying on the given context component ModeManagerContext are presented below.

in ModeManagerContext, *Scenario* defines the sequence of modes required to achieve the desired satellite attitude and bring the system into the *Science* mode, i.e., the state where the scientific payload instrument can be activated. This sequence consists of the following modes:

- *OFF* – the mode established right after system (re)booting;
- *STANDBY* – the mode is maintained until the separation from the launcher;
- *SAFE* – the mode designates that a stable attitude is acquired, which allows the coarse pointing control;
- *NOMINAL* – the satellite is trying to reach the fine pointing control;
- *PREPARATION* – the payload instrument is getting ready;
- *SCIENCE* – the payload instrument is ready to perform its tasks.

Let us note that the role of *Scenario* is twofold. Firstly, here it is used as an auxiliary construct to constrain the *Next* relation. Specifically, *Next* is defined as reflexive transitive closure of *Scenario* (formally defined in the axioms iaxm3-6).

INTERFACE ModeManager EXTENDS MMC_I
SEES ModeManagerContext

CONTEXT ModeManagerContext
...
AXIOMS
iaxm1 : MODE = {OFF, STANDBY, SAFE, NOMINAL, PREPARATION, SCIENCE}
iaxm2 : Scenario = {OFF ↦ STANDBY, STANDBY ↦ SAFE, SAFE ↦ NOMINAL,
NOMINAL ↦ PREPARATION, PREPARATION ↦ SCIENCE}
iaxm3 : Scenario ⊆ Next
iaxm4 : id ⊆ Next
iaxm5 : Next; Scenario ⊆ Next
iaxm6 : ∀z · Scenario ⊆ z ∧ z; Scenario ⊆ z ⇒ Next ⊆ z
iaxm7 : OFF = InitMode
iaxm8 : partition(ERROR, RecovErrors, UnrecovErrors, {NoError})
iaxm9 : RecovErrors ≠ ∅ ∧ UnrecovErrors ≠ ∅

Secondly, *Scenario* specifies the very next mode the system should transition to, when it is in a stable state but the final mode (i.e., *SCIENCE*) is not yet reached. In such a situation, the system should follow the predefined scenario without skipping any required intermediate modes. On the other hand, if the system tries to recover by rollbacking to one of the preceding modes, the next system mode is chosen using the inverted *Next* relation, i.e., the mode belongs to $Next^{-1}$.

Finally, the abstract set ERROR is now partitioned into the disjoint subsets *RecovErrors* and *UnrecovErrors* standing for the respective sets of recoverable and unrecoverable errors, as well as the predefined constant NoError. The specific sets of errors can be defined (as additional axioms) during module implementation.

5.3. AOCS Decomposition

For brevity, we omit the presentation of possible intermediate refinements of the machine aocs and demonstrate the decomposition as if it would be the first refinement of aocs. The REFINES and USES clauses show that aocs1 refines aocs and imports the module ModeManager. The prefix mm_ is used to clearly separate the module variables and the module operations from those of aocs1.

Since the machine aocs1 has the read access to the external (i.e., interface) variables of the module instance ModeManager, we strengthen its invariant to define the link between the variables of aocs1 and ModeManager. In the invariant inv1, we expresses the connection between the system level errors and the errors detected by Mode Manager. Essentially, inv1 postulates that all the recoverable errors are handled locally by Mode Manager. In the invariant inv2, we require that the preparation phase is completed only after Mode Manager stabilises in the *Science* mode.

It is easy to observe that the link between the top-level component aocs1 and Mode Manager is quite strong – we replace the abstract variable *prep* by the external variables of Mode Manager and link the errors detected by Mode Manager with the system-level errors.

```
REFINEMENT aocs1
  REFINES aocs
  USES ModeManager with prefix mm_
  INVARIANT
      inv1 :  mm_error ∉ UnrecovErrors ⇒ aocs_error = FALSE
      inv2 :  prep = TRUE  ⇔  (mm_next = mm_last ∧ mm_last = SCIENCE)
  ...
END
```

To write-access the state of Mode Manager, we include calls of the Mode Manager interface operations into the actions of the aocs1 events. In particular, the abstract event preparation is refined into a pair of events mode_advance and intermediate (see below). When Mode Manager successfully completes the transition to a certain mode, it chooses the next target mode from the predefined scenario. The guard of the event mode_advance states that the event is enabled only if the *Science* mode has not been reached, the system is stable, and no errors have been detected. In the action of this event, the interface operation Mode_Advance is called. This results in initiating a new mode transition in Mode Manager.

On the other hand, the event intermediate models the situation when Mode Manager has not yet reached the target mode, yet still continues the transition to it. In such a situation the module operation Continuation is called. In both events we use the short hand notation Mode_Advance and Continuation to denote the operation calls that ignore the return values.

```
mode_advance refines preparation   =  WHEN
                                        mm_error = NoError ∧ mm_last ≠ SCIENCE
                                        mm_last = mm_prev
                                      THEN
                                        mm_Mode_Advance
                                      END
intermediate refines preparation   =  WHEN
                                        mm_error = NoError ∧ mm_last ≠ SCIENCE
                                        mm_last ≠ mm_prev
                                      THEN
                                        mm_Continuation
                                      END
```

Now let us explain the role of Mode Manager in handling errors. As we mentioned before, AOCS integrates error recovery into its mode logic. This allows the AOCS system to achieve error confinement. Indeed, each mode requires active control over a certain subset of hardware units. When a unit fails, it is deactivated. To achieve an error-free state, AOCS performs a rollback to a mode in which the failed unit is inactive (i.e., not needed). Once the system stabilises in such a mode, Mode Manager initiates the transition to a more advanced mode and attempts to reactivate the failed unit (or, if possible, to activate its spare).

While refining the event recovery, we abstractly model this behaviour as follows. If an error is recoverable, the top-component calls the ResetError and ToMode operations. These operations correspondingly reset the error flag *error*

of Mode Manager and initiate the transition to the appropriate degraded mode. Moreover, if an error occurs during the activation phase, the preparation phase is resumed. If Mode Manager classifies the detected error as an unrecoverable one then the system is deadlocked or shut down, as modelled in the refined event error.

```
recovery = ANY m WHERE
               m ↦ mm_next ∈ Next^−1
               mm_error ∈ RecovErrors
           THEN
               mm_ResetError
               mm_ToMode(m)
               act := FALSE
           END
error    = WHEN
               mm_error ∈ UnrecovErrors
           THEN
               aocs_error, act := TRUE, FALSE
           END
```

The model aocs1 can be further refined by integrating new concrete implementation details of the modelled system. However, the interface of the imported module remains fixed during this refinement process.

5.4. Implementation of Mode Manager

The achieved decomposition of the model aocs1 and, by transitivity, of the abstract model aocs allows us to conduct further refinement of Mode Manager independently from the overall system development. This development follows the pattern for developing mode-rich systems described in Section 4. We start by creating an Event-B machine MMBody that implements the ModeManager interface. An excerpt from MMBody is shown below.

In MMBody, each interface operation is modelled by the corresponding group of events. For the sake of brevity, we show here only the group of events implementing the operation Continuation. Each group contains at least one event denoted FINAL that returns the control to the calling component. Any non-final event must pass control to another event in the same event group.

The interface operation Continuation operation is realised by a group containing three events. Please recall that Continuation models the behaviour of a MMC while it is trying to reach a certain target mode. The events adv_skip, adv_partial and adv_comp model correspondingly three possible outcomes of the call of Continuation: (1) no new mode is reached, (2) an intermediate mode between the current and target modes is reached, and (3) Mode Manager stabilises in the reached target mode. Within each event we also reserve the possibility of error occurrence.

The next refinement steps further elaborate on the implementation of Mode Manager. After several refinement steps we unfold the next architectural layer of AOCS. Specifically, at the layer below Mode Manager, there resides a component called *Unit Manager*. In the next section we are going to decompose the

```
MACHINE MMBody
  IMPLEMENTS ModeManager
  ...
  GROUP Continuation BEGIN
        FINAL adv_skip    =  WHEN next ≠ prev THEN error :∈ ERROR END
      FINAL adv_partial   =  ANY m WHERE
                                 next ≠ prev
                                 m ∈ MODE ∧ m ≠ next
                                 m ↦ next ∈ Next ∪ Next^-1
                             THEN
                                 last := m || error :∈ ERROR
                             END
       FINAL adv_comp     =  WHEN
                                 next ≠ prev
                             THEN
                                 error :∈ ERROR || last := next || prev := next
                             END
  END
  ...
END
```

implementation of Mode Manager to separate Unit Manager according to the same development pattern.

5.5. Unit Manager

Unit Manager is yet another example of a mode-managing component. The purpose of Unit Manager is to abstract the specifics of a hardware configuration and provide a simple common control interface to the hardware. The Unit Manager interface is an instance of the generic MMC_I interface shown in Figures 6 and 7. The interface specification of Unit Manager is obtained by constraining by concrete values the set of modes, the mode transition scenario, and the initial mode of the generic interface MMC_I. The resulting interface UnitManager relying on the given context component UnitManagerContext are presented below.

```
INTERFACE UnitManager EXTENDS MMC_I
  SEES UnitManagerContext
```

The Unit Manager modes define the positioning algorithms and are closely related to the set of hardware units involved in computing the positioning commands. The modes NAV_EARTH and NAV_SUN use crude algorithms based on the input from the Earth and Sun sensors, while NAV_ADV and NAV_FINE use the GPS unit to compute the satellite position with respect to the Earth surface. The mode NAV_INSTR is the final target mode of Unit Manager. It is reached when the scientific instrument hardware becomes enabled.

5.6. Integration of Mode Manager and Unit Manager

We separate the development of Unit Manager from Mode Manager in a similar way as we did separating the Mode Manager development from aocs.

```
CONTEXT UnitManagerContext
  ...
  AXIOMS
    uaxm1 : MODE = {OFF, NAV_EARTH, NAV_SUN, NAV_ADV,
                    NAV_FINE, NAV_INSTR}
    uaxm2 : Scenario = {OFF ↦ NAV_EARTH, OFF ↦ NAV_SUN,
                    NAV_EARTH ↦ NAV_ADV, NAV_SUN ↦ NAV_ADV,
                    NAV_ADV ↦ NAV_FINE, NAV_FINE ↦ NAV_INSTR},
    uaxm3 : Next = closure(Scenario)
    uaxm4 : OFF = InitMode
    ...
END
```

Namely, we decompose the refined model of Mode Manager (MMBody3) to introduce the UnitManager module as shown in Figure 9.

```
MACHINE MMBody3
  ...
  USES UnitManager with prefix um_
    CONSTANTS mode_map, error_map
    AXIOMS
      axm_1u : mode_map ∈ MODE ↔ um_MODE
      axm_2u : mode_map = {OFF ↦ um_InitMode, STANDBY ↦ um_InitMode,
          SAFE ↦ um_NAV_EARTH, SAFE ↦ um_NAV_SUN,
          NOMINAL ↦ um_NAV_ADV, PREPARATION ↦ um_NAV_FINE,
          SCIENCE ↦ um_NAV_INSTR}
      axm_3u : error_map ∈ um_ERROR → ERROR
      ...
  INVARIANT
    ...
    gi1 : next = prev ⇒ last ↦ um_last ∈ mode_map
    gi2 : next = prev ⇒ next ↦ um_next ∈ mode_map
    gi3 : next = prev ⇒ prev ↦ um_prev ∈ mode_map
    gi4 : in_sync = TRUE ∧ um_error ≠ um_NoError ⇒
            error ≠ NoError ∧ error = error_map(um_error)

  ...
END
```

Figure 9: Unit manager integration

Mode Manager does not have a direct access to the controlled hardware units and relies on the operations of Unit Manager to control hardware. The required mode and error handling consistency properties between these components are defined via the additional data structures *mode_map* and *error_map*, which are concrete implementations of the respective abstract functions *Mode_ent_cond* and *Mode_error_handling* introduced in Section 3.

Specifically, the relation *mode_map* constrains the allowed mode combinations of Mode Manager and Unit Manager, while the function *error_map* translates the detected errors of Unit Manager into the corresponding errors of Mode Manager. The definitions of *mode_map* and *error_map* are given in special fields

of the USES clause. Essentially, these definitions become a part of the Mode Manager context. To avoid name clashes, the Unit Manager module is instantiated with the prefix *um*. Consequently, all the names of external variables and interface operations of the Unit Manager module appear with the prefix *um*.

The new invariants of Mode Manager, $gi1, ..., g4$, require that the given mode and error mapping between Unit Manager and Mode Manager should be preserved. This implies that, in particular, an update of the Unit Manager mode often necessitates an update of the Mode Manager mode.

In particular, the invariants $gi1, gi2$ and $gi3$ express the mode consistency properties between the current and the lower layer that are preserved by the system. Likewise, the invariant $gi4$ states the expected relationship between the detected errors on two adjacent layers. In $gi1, gi2$ and $gi3$, the given premise $next = prev$ stipulates that Mode Manager should be in a stable state to guarantee the required mode consistency. For the invariant $gi4$, the formulated condition is that error consistency can be enforced only after Mode Manager synchronises with Unit Manager. The latter stipulation is related to the fact that in this paper we consider a sequential implementation of AOCS. During one cycle the components get control in a specific fixed order. Since Mode Manager gets control before Unit Manager, it cannot react on errors detected by Unit Manager until the next cycle starts.

All the events of Mode Manager must maintain the correspondence between the Mode Manager and Unit Manager modes and errors given in $gi1, ..., gi4$. As a result, an update of the Unit Manager mode often necessitates an update of the Mode Manager mode and vice versa. For example, the event set_mode presented below synchronously updates the Mode Manager and Unit Manager modes. This event belongs to the event group implementing the operation ToMode of Mode Manager.

set_mode	=	ANY m, p WHERE
		$m \in \text{MODE} \wedge error = \text{NoError}$
		$m \neq next \wedge m \mapsto next \in Next \cup Next^{-1}$
		$p \in \text{um\_MODE} \wedge m \mapsto p \in mode\_map$
		$um\_error = \text{um\_NoError} \wedge p \neq um\_next$
		THEN
		$prev := next \parallel next := m$
		um_ToMode(p)
		END

For the sake of brevity, we omit detailed presentation of the remaining development. It follows the refinement strategy described above. After several refinements of body of the Unit Manager, we split the development into the main control part and a number of subsystems modelling individual hardware units. Each such subsystem follows the same modelling pattern and starts with instantiating the generic MMC_I interface. Collectively, the units define the environment of the system and thus are only characterised by their interfaces.

In the specific hardware configuration that we are modelling, there are six hardware units. To construct a faithful model close to the executable program,

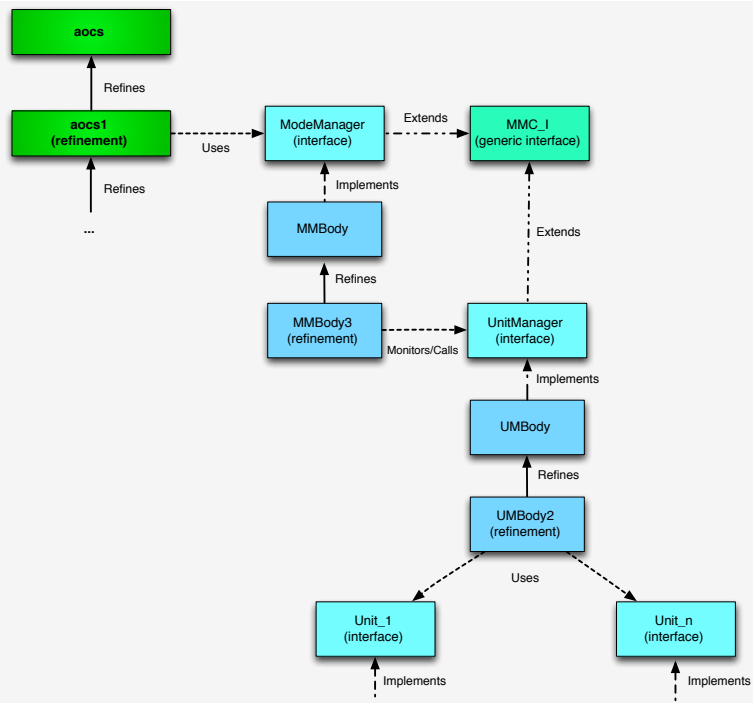

Figure 10: AOCS development hierarchy

we explicitly introduce each unit subsystem by importing the (correspondingly instantiated) generic module interface. The further development allows us to arrive at a well structured specification of AOCS. The overall hierarchical structure of the presented development process is given in Figure 10.

5.7. Proof Obligations

As mentioned already, design correctness is ensured via a number of verifications conditions, called proof obligations, automatically derived from the given machines and interfaces. There are three major kinds of verification conditions. The first one shows that a model is defined consistently, that is, the behaviour described in events is in agreement with the safety invariants. These conditions address the verification of mode invariants and mode stability properties. The second kind are the refinement obligations ensuring that a more concrete design is observably equivalent to the abstract design. In our approach, for the Mode Manager and Unit Manager components, these verification conditions also ensure that a refined machine meets all the obligations of an interface it is implementing. The last kind of verification conditions is specific to the modularisation extension of Event-B. Whenever a model is decomposed into a parent machine and a subordinated module, one has to show that a combination of two is observably equivalent to some abstract design. These define a special case of

refinement relation between models but the verification conditions are different.

As an example, let us consider the proof of correctness of the operation `ToMode` of the generic Mode Manager interface. One interesting verification is related to the invariant `inv4`. A verification condition due to the obligation of $inv4$ satisfaction by the operation gives rise to the following theorem.

axioms	$Next \cap Next^{-1} \subseteq \mathrm{id} \wedge Next; Next \subseteq Next$
invariant	$\{last \mapsto prev, last \mapsto next\} \subseteq Next \cup Next^{-1}$
operation guard	$m \in \mathrm{MODE} \wedge m \neq next \wedge m \mapsto next \in Next \cup Next^{-1}$
	$\vdash$
goal	$\{last \mapsto next, last \mapsto m\} \subseteq Next \cup Next^{-1}$

It trivially holds that $last \mapsto next \in Next \cup Next^{-1}$ and hence the goal may be simplified to $last \mapsto m \subseteq Next \cup Next^{-1}$. The proof strategy is to show that $last \mapsto m$ may be represented as a composition of two relations $last \mapsto next$ and $next \mapsto m$. From $m \mapsto next \in Next \cup Next^{-1}$, we are able to prove that $next \mapsto m \in Next \cup Next^{-1}$ holds. The proof is done by considering two different cases $m \mapsto next \in Next$ and $m \mapsto next \in Next^{-1}$. We now have to prove the following theorem.

axioms	$Next \cap Next^{-1} \subseteq \mathrm{id} \wedge Next; Next \subseteq Next$
hyp1	$last \mapsto next \in Next \cup Next^{-1}$
hyp2	$next \mapsto m \in Next \cup Next^{-1}$
	$\vdash$
goal	$last \mapsto m \subseteq Next \cup Next^{-1}$

To satisfy the goal, we introduce a supporting lemma stating that $Next \cup Next^{-1}$ is closed under relational composition.

axioms	$Next \cap Next^{-1} \subseteq \mathrm{id} \wedge Next; Next \subseteq Next$
	$\vdash$
goal	$(Next \cup Next^{-1}); (Next \cup Next^{-1}) \subseteq Next \cup Next^{-1}$

Distributing relational composition over set union we obtain the following, equivalent goal.

$$(Next; (Next \cup Next^{-1})) \cup (Next^{-1}; Next) \cup (Next^{-1}; Next^{-1}) \subseteq Next \cup Next^{-1}$$

We proceed by considering independently three simpler sub-goals:

$$(Next; (Next \cup Next^{-1})) \subseteq Next \cup Next^{-1},$$

$$Next^{-1}; Next \subseteq Next \cup Next^{-1},$$

and

$$Next^{-1}; Next^{-1} \subseteq Next \cup Next^{-1}.$$

The first two sub-goals are discharged by showing that $Next; Next^{-1} \subseteq \mathrm{id}$ and $Next^{-1}; Next \subseteq \mathrm{id}$. With the proof of the supporting lemma complete, the overall condition is now also proven.

The table illustrating the overall proof effort in terms of generated and automatically or manually proved proof obligations is given in the following table.

Step	Total	Auto	Manual	Manual %
aocs	11	11	0	0%
aocs1	39	27	12	31%
ModeManager	23	19	4	17%
MBody	9	8	1	11%
MBody3	37	34	3	8%
UnitManager	19	16	3	16%
UMBody	9	8	1	11%
UMBody2	25	19	6	24%
Unit_I	19	16	3	16%
Overall	191	156	33	17%

5.8. AOCS Development Summary

AOCS is a complex industrial-scale system and hence it would be impossible to describe its formal development in full detail. Nevertheless, in this section we presented the most interesting points of the development. Firstly, we have shown how different architectural layers can be unfolded by refinement. Secondly, we have demonstrated the mechanism of instantiating the generic pattern for specifying particular mode managing components (Mode Manager, Unit Manager and individual units) at different architectural layers. Finally, we have shown how to formally define the correspondence between modes at different architectural layers by strengthening invariants and instantiating the context parts of module components. As a result, we have verified the required mode consistency as an intrinsic part of the refinement process.

6. Lessons Learnt

The AOCS system described here is a generalised version of one of the implemented instances of AOCS. The real system was developed by Space Systems Finland some time ago using traditional development approaches. The company has observed that verification of the AOCS mode transitions via testing was quite difficult and time consuming. This has prompted the idea of experimenting whether a formal AOCS development would assist in ensuring correctness of mode transitions.

The initial attempt [15] to formally develop the system was undertaken by a verification engineer with a significant background in formal verification. The development was preceded by discussions with domain experts. However, the initial modelling was rather unsuccessful due to two major reasons. Firstly, modelling was significantly influenced by the code created for the real AOCS. As a result, to mimic the program counter, the major modelling efforts had to be spent on maintaining the heavy infrastructure enforcing the execution order of events. Secondly, at the time of this development, Event-B was still lacking

the modularisation support. As a result, fairly soon the developed monolithic model became unreadable for the developers and unmanageable for the Rodin platform. Hence it was concluded that further development would be quite problematic.

Apart from some technical issues that had to be resolved in the Rodin platform, we have learnt the following main lessons:

- Extensive support for modularisation is absolutely necessary to enable scalable formal development of complex industrial systems in Event-B;
- The development should support architectural-level modelling and allow us to express logical interdependencies between components at different levels;
- It is important to maintain readability of models.

The second development attempt [16] was preceded by a preparatory work that aimed at alleviating discovered problems. We have developed a modularisation plug-in [6] implementing the modularisation extension for Event-B that we have proposed previously [4]. Moreover, while formalising reasoning about mode-rich systems [17], we developed a pattern for specifying mode-managing components. However, probably most importantly, before starting the development as such, we drafted a refinement strategy. Our strategy was to build the system model in a hierarchical layered fashion via instantiation of generic modules. This approach has indeed demonstrated its viability.

The second development attempt – the one which is described in this paper – has achieved the desired goal. It was performed in a tight collaboration between the company and academia. We succeeded in building a detailed AOCS model and verified (by proofs) that it correctly implements the desired mode transition scheme. Experiments with generating code will be conducted after code generation capabilities of Rodin platform will be implemented. The development was performed in a structured way, where the levels of abstraction corresponded to the architectural layers. While performing a refinement step, we unfolded the architectural layers and ensured the consistency of mode transitions between adjacent layers as a part of refinement verification. The specifications of components were obtained by instantiating the generic module interface that is common for mode managing components on different layers of abstraction.

Refinement by instantiating the generic components has significantly simplified the development and proof activity. The overall system model is rather compact and can be easily maintained because it includes only references to the components visible state and interface. By developing each component as a separate module we obtained compact and easy to comprehend models. As a result, we have alleviated the problem of manipulating large monolithic models.

In our development we have made a smooth transition from the architectural modelling to modelling of the detailed behaviour of each particular component. The properties of generic module parameters determine the constraints on concrete data structures that should be proved during module instantiation.

Our mechanism of module instantiation and then subsequent development (refinement) of a module ensures that these constraints are satisfied by module implementation.

The layered development has also facilitated modelling and verification of the system fault tolerance mechanisms. The hierarchical architecture allowed us to distribute the responsibilities of error handling across different layers, which resulted in a well-structured implementation of the fault tolerance mechanisms.

The main lessons that we have learnt from this development are the following

- It is important to have a strategy of the development - a certain refinement plan that is drafted before the real development commences;
- It is beneficial to refrain from modelling major design decisions in the initial specification since it can significantly complicate the later development;
- Modularisation support is paramount in modelling large scale systems;
- Without a mature tool support a formal development of industrial systems is infeasible.

At various stages of the project, engineers with different background in formal methods have experimented with modelling in Event-B. In general, they liked the idea of top-down formal development and found the modularisation extension useful and intuitive to use. The detailed report on the use of Event-B in industrial practice can be found elsewhere [18].

7. Related Work

This paper builds on our previous research [17, 19] on modelling mode-rich systems. In this paper we have presented a solid theoretical justification of the modularisation extension of Event-B together with the accompanying proof obligations and defined the notion of a generic parameterised component. Such an extension greatly facilitates reuse, since it allows the designers to quickly create formally verified components by instantiation. In particular, it has allowed us to represent the formal development of a layered mode-rich system as a iterative process of instantiation of a generic mode-managing component. Another important theoretical contribution of this paper is a formalisation of the mode stability property. This property is especially important for systems with non-instantaneous mode transitions. Essentially, it allows the designers to unambiguously describe the status of components while modelling mode-consistency conditions. As such, it circumvents the problem of defining numerous auxiliary modes even while describing fine-grained mode consistency conditions.

Besides enriching the theoretical basis, in this paper we also have given a deep insight on practical aspects of engineering of mode-rich systems. We have presented a detailed description of a realistic development, discussed pitfalls and achievements as well as given an estimate of the verification efforts required to formally develop an industrial-scale mode-rich system.

The research on various aspects of mode-rich systems lasts over several decades. Among the most prominent works on modelling mode-rich systems is the modechart framework [20]. It uses the concept of modes from the pioneering works of Parnas, who proposed to consider modes as partitions of the system state space and facilitators of system modularisation. The goal of the modechart approach is similar to ours – to formally verify safety properties of complex mode-rich systems. Modecharts assign real-time logic formulae to various types of mode transitions and facilitate building a hierarchy of real-time logic assertions. The main difference between the modecharts approach and the approach presented in this paper is in treating mode transitions. In modecharts, transitions between modes are instantaneous. Such a treatment of mode transitions has allowed several researchers to build various automata-based formalisations of mode logic, e.g., [21]. In our work, mode transitions are non-instantaneous. They might be also interrupted, e.g., to perform error recovery. Since we precisely define which properties can be guaranteed depending on whether the system is stable or in transition, we avoid introducing a fine slicing of the mode logic to reflect each possible combination of the mode-submode relation. We believe that our approach better caters to abstract modelling and improves readability of formal models. Moreover, our approach offers not only a verification support but also a development method.

Modelling modes has been extensively studied within the software architectures field (e.g., [22, 23]). This research strand is mainly focuses on studying architectural language constructs to represent modes. In contrast, our approach puts strong emphasis on the development and verification methodology. However, as a future research direction it would be interesting to express the proposed method in an architecture description language, e.g., such as AADL.

Formal validation of the mode logic and, in particular, the fault tolerance mechanisms of satellite software has been undertaken by Rugina et al [10]. They have investigated different combinations of simulation and model checking. In general, simulation does not allow the designers to check all execution paths, while model checking often runs into the state explosion problem. To cope with these problems, the authors had to experiment with combination of these techniques as well as heavily rely on abstractions. Our approach is free from these problems. First, it allows the developers to systematically design the system and formally check mode consistency within the same framework. Second, it enables exhaustive check of the system behaviour, yet avoiding the state explosion problem. Hence our approach can potentially give the developers better confidence in the correctness of the obtained design.

The mode-rich systems have been studied to investigate the problem of mode confusion and automation surprises. These studies conducted retrospective analysis of mode-rich systems to spot the discrepancies between the actual system mode logic and the user mental picture of the mode logic. Most of the approaches relied on model-checking [24, 25, 26], while [27] relied on theorem proving in PVS. Our approach focuses on designing fully automatic systems and ensuring their mode consistency. Unlike [25], in our approach we also emphasize the complex relationships between system fault tolerance and the mode logic.

In our previous work [28], we have studied a problem of specifying mode-rich systems from the contract-based rely-guarantee perspective. These ideas have been further applied for fault tolerance modes [29]. The resulting approach provides fault tolerance modelling facilities explicitly supporting the traceability of the fault tolerance and dependability requirements. Moreover, it extends [28] with additional fault tolerance semantics, structural checks, and helps the modeller by offering reusable refinement templates.

However, a mode-centric specification of the system, proposed in [28, 29], neither defines how the system operates in some specific mode nor how mode transitions occur. It rather imposes restrictions on concrete implementations. Such an approach complements traditional modelling but does not replace it. In this paper we have demonstrated how to combine the reasoning about the system mode logic and its functioning.

The AOCS framework has been developed by European Space Agency to facilitate reuse in the space domain. The Giotto framework [30] has aimed at providing a methodology for implementing embedded control with predictable timing properties. Among others, the framework addresses the issue of mode switching. However, in Giotto, the mode concept is centered around time aspect, while mode switching corresponds to changing the task schedule. In our approach we take a more state-centric approach and analyse mode consistency as a relation over component states.

8. Conclusions

In this paper we have proposed a formal approach to development of mode-rich layered systems. The paper extends our previous work presented in [17, 19]. The proposed approach is based on instantiation and refinement of a generic specification pattern for a mode manager. The pattern defined as a generic module interface captures the essential structure and behaviour of a component and can be instantiated by component specific data to model a mode manager at any layer of the system hierarchy. The overall process can be seen as a stepwise unfolding of architectural layers. Each such unfolding is accompanied by proving its correctness, while also verifying mode consistency between two adjacent layers. Such an incremental verification allows us to guarantee the global mode consistency, yet avoid checking the property for the whole architecture at once.

The generic specification pattern relies on our formalisation of reasoning about systems with non-instantaneous mode transitions, the mode logic of which is also integrated with error recovery. The formalisation of what constitutes mode consistency and mode invariance properties together with establishing precise relationships between error recovery and the mode logic allowed us to derive design guidelines and logical constraints for components of mode-rich systems.

In this paper we described formal development of the AOCS system by refinement in Event-B. The attempted case study has shown that the Event-B framework and the supporting RODIN platform have promising scalability. Our approach facilitated creating a clean system architecture and also allowed us to

make a smooth transition from the architectural-level system modelling to specification and refinement of each particular component. Moreover, the proposed refinement-based development techniques have coped well with modelling the complex mode transition scheme and verification of its correctness.

Verification of all possible mode transitions (including complex cascading effects) is done by proofs and does not require any simplifications. Currently that level of assurance cannot be delivered either by model-checking, simulation or testing alone, or by combination of these techniques. The proposed modularisation and stepwise development style allow us to keep manual proof efforts at a reasonable level (about 17

The aim of this research is not merely experimenting with modelling a particular industrial-size system in Event-B, but rather creating a generic solution that would help develop AOCS-like systems. It is important that our approach to modelling mode-rich components using generic instantiation supports both reuse and composition. Such reuse is safe, since in the course of developing a component by refinement it is formally ensured that it conforms to the instantiated specification of its interface. Moreover, it becomes manageable to verify the composition of components whose state and behaviour are succinctly and formally modelled.

Our work can be seen as a step towards creating a formal approach to model-driven development and a detailed definition of the reference architecture for the space sector – the two recent initiatives of European Space Agency [31]. In the future it would be interesting to link our approach to languages specifically dedicated to architectural modelling. Moreover, it would be useful to continue experimenting with formal modelling of various types of mode-rich systems architectures as well as addressing the problem of ensuring mode consistency in the presence of dynamic reconfiguration.

Acknowledgments

This work is supported by the FP7 ICT DEPLOY Project and the EPSRC/UK TrAmS-2 platform grant.

References

[1] N. Leveson, L. D. Pinnel, S. D. Sandys, S. Koga, J. D. Reese, Analyzing Software Specifications for Mode Confusion Potential, In Proceedings of Workshop on Human Error and System Development, C.W. Johnson, Editor, pg. 132-146, Glasgow, Scotland, March 1997.

[2] J.-R. Abrial, Modelling in Event-B, Cambridge University Press, 2010.

[3] Rigorous Open Development Environment for Complex Systems (RODIN), deliverable D7, Event B Language, online at http://rodin.cs.ncl.ac.uk/.

[4] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, T. Latvala, Supporting Reuse in Event B Development: Modularisation Approach, In *Proceedings of Abstract State Machines, Alloy, B, and Z (ABZ 2010), Lecture Notes in Computer Science*, Vol.5977, pp. 174-188, Springer, 2010.

[5] The RODIN platform, online at http://rodin-b-sharp.sourceforge.net/.

[6] RODIN modularisation plug-in, documentation at http://wiki.event-b.org/index.php/Modularisation_Plug-in.

[7] OBSW formal development in Event B, online at http://deploy-eprints.ecs.soton.ac.uk/view/type/rodin=5Farchive.html.

[8] Industrial deployment of system engineering methods providing high dependability and productivity (DEPLOY), iST FP7 project, online at http://www.deploy-project.eu/.

[9] DEPLOY Deliverable D20 – Report on Pilot Deployment in the Space Sector. FP7 ICT DEPLOY Project. January 2010, online at http://www.deploy-project.eu/.

[10] A. E. Rugina, J. P. Blanquart, R. Soumagne, Validating failure detection isolation and recovery strategies using timed automata, in: Proc. of 12th European Workshop on Dependable Computing, EWDC 2009, Toulouse, 2009.

[11] J.-R. Abrial, The B-Book, Cambridge University Press, 1996.

[12] R. Back, K. Sere, Superposition refinement of reactive systems, *Formal Aspects of Computing*, 8(3), pp.1-23, 1996.

[13] B. Rubel, Patterns for Generating a Layered Architecture, In J.O. Coplien, D.C. Schmidt (Eds.), Pattern Languages of Program Design, Addison-Wesley, 1995.

[14] L. Laibinis, E. Troubitsyna, Fault tolerance in a layered architecture: a general specification pattern in B, In *Proceedings of the Second International Conference on Software Engineering and Formal Methods (SEFM 2004), Beijing, China*, pp. 346-355, IEEE Press, 2004.

[15] K. Varpaaniemi, Event-B Project DepSatSpec015Model000, January 2010, DEPLOY publication repository: http://deploy-eprints.ecs.soton.ac.uk/168.

[16] A. Iliasov, L. Laibinis, E. Troubitsyna, An Event-B model of the Attitude and Orbit Control System, 2010, dEPLOY publication repository: http://deploy-eprints.ecs.soton.ac.uk/.

[17] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, P. Väisänen, D. Ilic, T. Latvala, Verifying Mode Consistency for On-Board Satellite Software, In *SAFECOMP 2010, The 29th International Conference on Computer Safety, Reliability and Security, September 2010, Vienna, Austria*, Lecture Notes for Computer Science, Springer, 2010.

[18] DEPLOY Deliverable D29 – Initial Assessment Results. FP7 ICT DEPLOY Project. September 2010, online at http://www.deploy-project.eu/.

[19] A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, P. Väisänen, D. Ilic, T. Latvala, Developing Mode-Rich Satellite Software by Refinement in Event B, In *FMICS 2010, The 15th International Workshop on Formal Methods for Industrial Critical Systems*, Lecture Notes for Computer Science, Springer, 2010.

[20] F.Jahanian, A.Mok, Modechart: A Specification Language for Real-Time Systems, *IEEE Transactions on Sofware Engineering*, 20, pp. 933-947, 1994.

[21] F. Maraninchi, Y. Rémond, Mode-automata: About modes and states for reactive systems, in: European Symposium On Programming, Springer verlag, 1998.

[22] J. M. D. Hirsch, J. Kramer, S. Uchitel, Modes for software architectures, in: LNCS 4344, Springer, 2006, pp. 113–126.

[23] F. P. J.Kofron, O. Sery, Modes in component behavior specification via ebp and their application in product lines, in: *Information and Software Technology 51/1*, Elsevier, 2009, pp. 31–41.

[24] B. Buth, Analysing mode confusion: An approach using fdr2, in: Proceedings of SAFECOMP, Springer, Lecture Notes in Computer Science, Vol. 3219, 2004, pp. 101–114.

[25] M. Heimdahl, N. Leveson, Completeness and Consistency in Hierarchical State-Based Requirements, IEEE Transactions on Software Engineering, Vol.22, No.6, pp. 363-377, June 1996.

[26] J. Rushby, Using model checking to help discover mode confusion and other automation suprises, in: Reliability Engineering and System Safety, Vol.75, 2002, pp. 167–177.

[27] R. W. Butler, An introduction to requirements capture using PVS: Specification of a simple autopilot, Technical report, NASA TM-110255, May 1996.

[28] F. Dotti, A. Iliasov, L. Ribeiro, A. Romanovsky, Modal Systems: Specification, Refinement and Realisation, Conference on Formal Engineering Methods - ICFEM 09, Rio de Janeiro, Brazil, Lecture Notes in Computer Science, Vol. 5885, Springer, December 2009.

[29] I. Lopatkin, A. Iliasov, A. Romanovsky, On fault tolerance reuse during refinement, in: Proc. of 2nd International Workshop on Software Engineering for Resilient Systems, 2010.

[30] T. Brown, A. Pasetti, W. Pree, T. Henzinger, C. Kirsch, A Reusable and Platform-Independent Framework for Distributed Control Systems, in: Proceedings of the 20th Digital Avionics Systems Conference, Vol.2, pp. 6A1/1 - 6A1/11, IEEE, 2001.

[31] European Cooperation for Space Standardization, Software general requirements ECSS-E-ST-40C, 2009.