Newcastle University e-prints

Date deposited: 4th April 2011

Version of file: Author final

Peer Review Status: Peer reviewed

Citation for item:

Iliasov A, Troubitsyna E, Laibinis L, Romanovsky A, Varpaaniemi K, Vaisanen P, Ilic D, Latvala T. Verifying Mode Consistency for On-Board Satellite Software. *In:* E. Schoitsch (Ed.): SAFECOMP 2010, LNCS 6351, pp. 126–141, 2010

Further information on publisher website:

http://www.springerlink.com

Publisher's copyright statement:

The original publication is available at www.springerlink.com at the following link:

http://dx.doi.org/10.1007/978-3-642-15651-9_10

Always use the definitive version when citing.

Use Policy:

The full-text may be used and/or reproduced and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not for profit purposes provided that:

- A full bibliographic reference is made to the original source
- A link is made to the metadata record in Newcastle E-prints
- The full text is not changed in any way.

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Robinson Library, University of Newcastle upon Tyne, Newcastle upon Tyne.
NE1 7RU. Tel. 0191 222 6000

Verifying Mode Consistency for On-Board Satellite Software

Alexei Iliasov[1], Elena Troubitsyna[2], Linas Laibinis[2], Alexander Romanovsky[1], Kimmo Varpaaniemi[3], Pauli Väisänen[3], Dubravka Ilic[3], and Timo Latvala[3]

[1] Newcastle University, UK
[2] Åbo Akademi University, Finland
[3] Space Systems Finland
{alexei.iliasov, alexander.romanovsky}@ncl.ac.uk
{linas.laibinis, elena.troubitsyna}@abo.fi
{Dubravka.Ilic, Timo.Latvala, Kimmo.Varpaaniemi, Pauli.Vaisanen}@ssf.fi

Abstract. Space satellites are examples of complex embedded systems. Dynamic behaviour of such systems is typically described in terms of operational modes that correspond to the different stages of a mission and states of the components. Components are susceptible to various faults that complicate the mode transition scheme. Yet the success of a mission depends on the correct implementation of mode changes. In this paper we propose a formal approach that ensures consistency of mode changes while developing a system architecture by refinement. The approach relies on recursive application of modelling and refinement patterns that enforce correctness while implementing the mode transition scheme. The proposed approach is exemplified by the development of an Attitude and Orbit Control System undertaken within the ICT DEPLOY project.

1 Introduction

Operational modes – mutually exclusive sets of the system behaviour [13] – form a useful structuring concept that facilitates the design of complex systems in different industrial sectors, including avionic, transportation and space. There are several well-known problems associated with mode-rich systems, e.g., correctness of complex mode transitions, mode consistency in distributed systems, mode confusion etc. However, there is still a lack of generic architectural-level approaches that would facilitate solving these difficult problems.

In this paper we propose a formal approach to developing complex mode-rich systems that allows us to ensure mode consistency. The approach generalizes the results of a large pilot development carried out by Space Systems Finland within the FP7 ICT project DEPLOY [11]. In collaboration with the academic partners the company has undertaken formal development and verification of (a part of) a satellite Attitude and Orbit Control System (AOCS) [6].

AOCS is a typical representative of mode-rich component-based control systems. It consists of several instruments that control the attitude and the orbit of a satellite as well as perform different scientific measurements. The distinctive features of the system are long-running mode transitions and strong impact of component failures on the mode transition scheme.

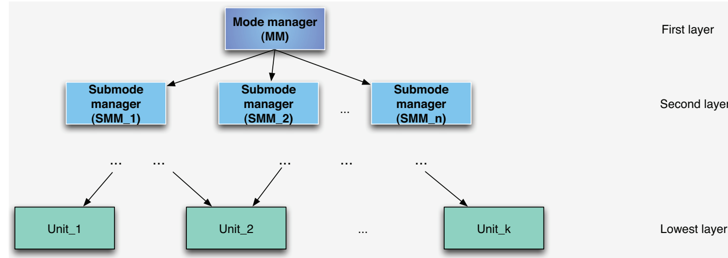

Fig. 1. Architecture of mode-rich layered systems

In this paper we formalize the reasoning about mode consistency in layered mode-rich systems. As a result, we propose a generic pattern for specifying components of such systems. This pattern defines a generic module interface that can be instantiated by component-specific data and behaviour. We demonstrate that such systems can be developed by recursive instantiation of the proposed pattern. Our approach can be also seen as stepwise unfolding of architectural layers. The approach is illustrated by briefly describing the AOCS development.

2 Layered Mode-Rich Systems

It is recognized that a layered architecture is advantageous in designing complex component-based systems [16]. It provides the designers with a convenient mechanism for structuring system behaviour according to the identified abstraction levels. The lowest layer usually consists of the components that work directly with hardware devices. The layer above contains the components encapsulating the lowest components by providing abstract interfaces to them. Depending on system complexity and design decisions, there might be several intermediate layers. Finally, the top component provides an interface to the overall system.

In this paper we study the issues in designing and verifying layered mode-rich control systems. Leveson et al. [13] define *mode* as a mutually exclusive set of system behaviours. There is a large variety of mode-rich systems, including control systems that cyclically monitor the controlled environment. Here we focus on one particular aspect of the control system behaviour – mode management.

2.1 Mode Logic in Layered Architectures

In the core of mode management is the *mode logic* that consists of all the available modes and rules for transitioning between them [13]. The typical problem associated with mode-rich systems is to ensure mode consistency of the components residing at different layers, i.e., to correctly define the mode logic and guarantee that the system faithfully implements it. The (somewhat simplified) architecture of mode-rich layered systems is shown in Figure 1.

On the top layer is *Mode Manager (MM)* – the component implementing the mode logic on the system level. We assume that during its mission the system should execute a certain *scenario* defined in terms of its (global) modes. On the one hand, the *MM* mode logic is defined by this scenario. On the other hand, component failures can prevent the system from implementing the mode scenario and force it to redo certain steps. Hence, to fully define the mode logic, we should take into account both the component states and their possible errors.

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 processes and hence have a duration. Indeed, to make a transition from the current to a new target mode, the system should bring all the involved components into the *consistent* states for entering the target mode. Therefore, while nominally being in one global mode, the system can be in two different states – stable and transitional.

When *MM* chooses a new target mode, it initiates (sequentially or in parallel) the corresponding mode transitions in lower layer components. As a result, the Submode Managers *(SMMs)* start to execute their own predefined scenarios to enable the global mode transition. Essentially, the behaviour of *SMMs* is similar to the behaviour of *MM*, i.e., while executing these scenarios they monitor the state of lower layer components to detect when the submode change is completed or an error has occurred. This allows us to define mode managers at different layers by instantiating the same generic specification pattern, as we show later.

If an error is detected, the responsible mode manager assesses the error and either initiates error recovery by itself or propagates the error to a higher layer. In mode-rich systems, error recovery is often implemented as a rollback to some preceding (and usually more degraded) mode in the predefined scenario.

The dynamic behaviour of the overall system is cyclic. At each cycle, MM assesses the SMM states and, based on these observations, it either

- initiates a forward transition according to the predefined scenario;
- initiates a backward transition (if error(s) has occurred). The actual target mode depends on severity of the occurred error(s);
- completes a transition to the target mode and becomes stable (if the conditions for entering the target mode are satisfied);
- maintains the current mode (if neither the conditions for entering the next global mode are satisfied nor an error has occurred).

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. Next we define the mode consistency criteria for layered control systems more formally.

2.2 Formal Reasoning about Modes and Mode Transitions

Essentially, a mode can be understood as an abstraction of the system state, i.e., the class of states associated with a certain system functionality. The mode logic is usually defined as a set of all the modes and mode transition rules [13]. Thus we can consider it as a special kind of a state transition system or, formally, 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 allowed mode transitions, and $InitMode$ is the initial mode. Since $Next$ is a relation, it can contain several predefined scenarios that can be executed by a mode manager.

Sometimes *Next* can be defined more precisely as an *ordering relation*. Indeed, some predefined scenarios define how to take a system from a non-operational mode (e.g., *Off*) to a fully operational one. The predefined scenario of the AOCS system presented in Section 5 is a typical example of this. This scenario describes

the sequence of modes from powering-on the instruments to bringing them into the mode that enables collection of valuable scientific data.

While *Next* is based on the predefined scenario(s), the mode transitions executed as error recovery are governed by the function $Mode_error_handling$:

$$Mode_error_handling \; : \; MState \times (LocalErrors_1 \times ... \times LocalErrors_k) \rightarrow \; Modes$$

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

However, *Next* represents only a possibility of mode transitions. For a mode transition to be completed, certain mode entry conditions should be satisfied. We can formally define this by introducing a function $Mode_ent_cond$ of the type:

$$Mode_ent_cond \; : \; Modes \rightarrow \mathcal{P}(MState \times LocalModes_1 \times \cdots \times LocalModes_k) \qquad (1)$$

where $LocalModes_1, .., LocalModes_k$ are modes of the monitored components. For each (global) mode, the function returns a 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.

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

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:

$$\forall i, j \bullet M_i \in Modes \; \wedge \; M_j \in Modes \; \wedge \; i \neq j \;\; \Rightarrow$$
$$Mode_ent_cond(M_i) \cap Mode_ent_cond(M_j) = \varnothing \qquad (2)$$

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

Let us now address another important issue in designing mode-rich systems – ensuring *mode invariants*. These are system properties that are required to be preserved in each 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 mode manager:

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

Collectively, these three attributes unambiguously describe the actual mode of a mode manager. 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 component is maintaining the last successfully reached mode

– $Increasing \triangleq last_mode = previous_target \wedge previous_target < next_target$
 a component is in transition to a next, more advanced mode;
– $Decreasing \triangleq next_target < previous_target$
 component stability or a mode transition to $previous_target$ was interrupted (e.g., by error handling) by a new mode request to a more degraded mode.

A graphical diagram showing mode status changes is given in Figure 2.

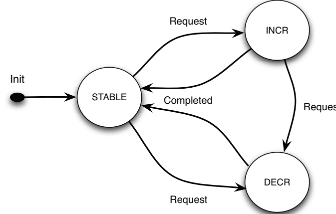

Fig. 2. Component mode status

We assume that, when a mode transition is completed, the component status is changed to *Stable*. The mode manager *MM* will maintain this status only if the final mode(s) of the scenario (defined by *Next*) is reached. On the lower layers, mode managers *(SMMs)* will maintain their stability until receiving a request for a new mode transition. In its stable state, *MM* would change its status to *Increasing* to execute the next step of the mode scenario, which in turn would trigger the corresponding mode transitions of the lower layer components. 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.,

$$\forall i \bullet m_i \in Modes \wedge last_mode = M_i \wedge Stable \Rightarrow (s, l_1, ..., l_k) \in Mode_ent_cond(M_i)$$

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

$$\forall i \bullet m_i \in Modes \wedge last_mode = M_i \wedge Stable \Rightarrow Mode_Inv(M_i)$$

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

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, the mode manager makes its decision to initiate a new mode transition to some more advanced mode according to the relation $Next$;
R2 In a transitional state, the mode manager monitors the state of lower layer components. When $Mode_ent_cond(next_target)$ becomes satisfied for the local state and the submodes of monitored components, the mode manager completes the mode transition and becomes stable;
R3 In both stable and transitional states, the mode manager monitors the lower layer components for the detected errors. If such errors occurred in the last cycle, the mode manager makes its decisions based on $Mode_error_handling$, which is applied to the mode manager state and all the detected errors.

In Section 4 we will show how these guidelines can be implemented in the proposed formal specification and development patterns.

3 Event B

Our chosen formal specification framework – Event B – allows us to model and verify state transition systems. Since it relies on theorem proving rather than model checking, it scales well to reasoning about systems with large state space.

3.1 Modelling and Refinement in Event B

The Event B framework [2] is an extension of the B Method [1]. The framework enables modelling of event-based (reactive) systems by incorporating the ideas of the Action Systems formalism [3]. Event B is actively used within FP7 ICT project DEPLOY to develop dependable systems from various domains.

The Event B development starts from creating a formal system specification. A simple Event B specification has the following general form:

```
MACHINE AM
SEES Context
VARIABLES v
INVARIANT Inv
EVENTS
  INITIALISATION = ...
  E1 = ...
  ...
  EN = ...
END
```

Such a specification encapsulates a local state (model variables) and provides operations on the state. The operations (called *events*) can be defined as

ANY vl **WHERE** g **THEN** S **END**

where vl is a list of new local variables, the guard g is a state predicate, and the action S is an assignment on model variables. In case when vl is empty, the event syntax becomes **WHEN** g **THEN** S **END**. Both ordinary and non-deterministic assignments can be used to specify state change. The non-deterministic assignments are of the form $v \; :\mid Post(v, v')$, where $Post$ is the postcondition relating the variable values before and after the assignment.

The events describe system reactions when the given **WHEN** or **WHERE** conditions are satisfied. The **INVARIANT** clause contains the properties of the system (state predicates) that should be preserved during system execution. The model data types and constants are defined in a separate component called Context.

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

$$Inv(v) \wedge g_e(v) \;\Rightarrow\; \exists v'.\; Post_e(v, v')$$

$$Inv(v) \wedge g_e(v) \wedge Post_e(v, v') \;\Rightarrow\; Inv(v')$$

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

The consistency of Event B models as well as correctness of refinement steps should be formally demonstrated by discharging *proof obligations*. The Rodin

platform[19], a tool supporting Event B, automatically generates the required proof obligations and attempts to automatically prove them. Sometimes it requires user assistance by invoking its interactive prover. However, in general the tool achieves high level of automation (usually over 90%) in proving.

3.2 Modelling modular systems in Event B

Recently the Event B language and tool support have been extended with a possibility to define modules [10, 15] – 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, when needed, composed with the main system.

A module description consists of two parts – *module interface* and *module body*. Let M be a module. A module interface MI is a separate Event B component. It allows the user of module M to invoke its operations and observe the external variables of M without having to inspect the module implementation details. MI consists of external module variables w, constants c, and sets s, the external module invariant M_Inv(c, s, w), and a collection of module operations, characterised by their pre- and postconditions, as shown below.

INTERFACE MI =
  SEES MI_Context
  VARIABLES w
  INVARIANT M_Inv(c, s, w)
  OPERATIONS
    op_1(par,res) =
      PRE M_Guard$_1$(c, s, par, w)
      POST M_Post$_1$(c, s, par, w, w', res')
. . . **END**

A module development always starts with the design of an interface. Once an interface is defined, it cannot be altered in any manner. This ensures correct relationships between a module interface and its body. A module body is an Event B machine, which implements each interface operation by a separate group of Event B events. Additional proof obligations guarantee that each event group faithfully implement the corresponding pre- and postconditions.

When the module M is "included" into another Event B machine, the including Event B machine can invoke the operations of M as well as read all the external variables of M. Several instances of the same module operating on disjoint state spaces can be created. Moreover, module abstract types and constants, defined in the interface context, can be instantiated with concrete data.

The modularisation extension of Event B was motivated by the pilot deployment of a satellite system within the DEPLOY project [11]. The extension was needed not only to enable architectural level reasoning but also to significantly improve scalability of Event B. As we show next, the modularisation extension also facilitates modelling and verification of layered mode-rich systems.

4 Development pattern

In this section we propose a formal development pattern for layered mode-rich systems in the Event B framework. This pattern is based on formal reasoning about modes and mode transitions presented in Section 2.

```
INTERFACE Mode_Manager
SEES Mode_Manager_Context            (* introduces abstract Modes, Errors, and Next *)
VARIABLES last_mode, next_target,    (* list of external variables of a module *)
          previous_target, error
INVARIANTS
    types of external variables
    other invariant properties
OPERATIONS

SetTargetMode =
  ANY m
  PRE
    Component has not failed
    m is a new target mode
  POST
    new target mode is set

ResetError =
  PRE
    the error flag is raised
  POST
    the error flag is cleared

RunStable =
  PRE
    Component is stable and not failed
  POST
    Component either remains stable
    or changes its mode according to the scenario
    or raises the error flag

RunNotStable =
  PRE
    Component is in a mode transition
  POST
    A mode transition is completed
    or a mode transition continues
    or the error flag is raised
```

Fig. 3. Interface of a generic mode manager

4.1 Generic interface

As discussed earlier, the structure and behaviour of mode managers at different layers are very similar. This suggests the idea of modelling such a component 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 contains four operations that can be called from a higher layer. It also defines the external module variables that are visible from a calling component. An informal description of the interface pattern is given in Figure 3.

The external state of a component is formed by four variables – $last_mode$, $next_target$, $previous_target$ and $error$. The first three variables define the component mode status, while the last one models the currently detected errors. Moreover, the interface context introduces the abstract sets $Modes$ and $Errors$, and the abstract functions $Next$ and $Mode_error_handling$. These structures should be instantiated with concrete data when a module instance is created. If $Next$ is a partial order, its required properties are also checked during instantiation.

The operation *SetTargetMode* is called to set a new target mode, while *ResetError* is called 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 the cycle the control is passed from layer to layer to each component. The operations *RunStable* and *RunNotStable* model component behaviour when it receives the control while being correspondingly in a stable or a transitional state. The actual state is unambiguously determined by the external mode status variables.

Let us now show that our interface pattern follows the guidelines of Section 2. The requirement (**R1**) stipulates the use of the predefined scenario $Next$ in a stable state. In the presented interface, this requirement is incorporated into the postcondition of *RunStable*. The requirement (**R2**) prescribes the use of $Mode_ent_cond$ to monitor whether the current mode transition has succesfully completed. The requirement is a part of the postcondition of *RunNotStable*.

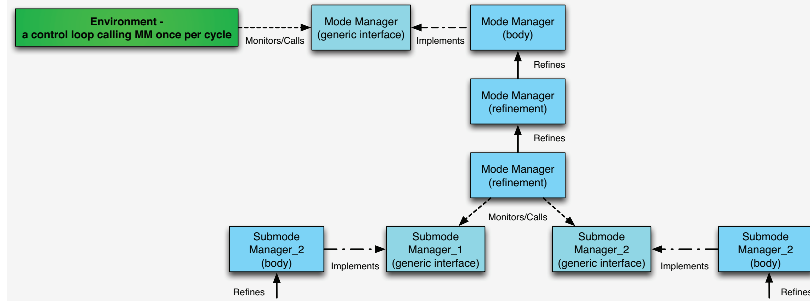

Fig. 4. Development hierarchy

Finally, the requirement (**R3**) calls for *Mode_error_handling* to be used when, upon detection of error(s), a new mode request has to be sent to lower layers. This requirement is defined in terms of the component state and the modes of lower layer components, i.e., in terms of two adjacent layers. This requirement can only be demonstrated during refinement, i.e., when lower layer components are introduced. Nevertheless, even in the generic pattern we require that implementations of *RunStable* and *RunNotStable* have to use this function to assess the errors flagged by the lower layer components.

All the operations update the variables *last_mode*, *next_target*, *previous_target* to reflect the changing mode status. Due to a lack of space, we postpone presenting all formal details of the interface specification pattern until Section 5, where we discuss modelling of AOCS.

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 the 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 [12]. 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 mode manager at a lower layer. Therefore, by instantiating *MMC_I* with the mode logic specific for a particular mode manager, we can obtain a mode manager of any layer. Hence our development strategy can be seen as a process of introducing specific module types into an Event-B development, as shown in Figure 4.

We assume that the system executes cyclically, with the environment periodically invoking the top mode manager. In its turn, it calls lower layer mode managers. 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 lower layer mode managing components that the mode manager controls are introduced. This refinement step essentially introduces calls to the corresponding interface

operations of these submode managers. At the same time, the submodes and errors of the lower layer become visible for the mode manager. Hence we can define the mode consistency conditions as additional invariants 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 submode managers. These bodies would implement their own mode logics and also, if needed, call operations of the mode managers residing on the layer below. Hence we follow the same refinement pattern as before, unfolding the 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 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 generalizes our experience in developing AOCS [6], presented next.

5 Case study – Attitude and Orbit Control System

The Attitude and Orbit Control System (AOCS) is a generic component of satellite onboard software, the main function of which is to control the attitude and the orbit of a satellite. Due to a tendency of a satellite to change its orientation because of disturbance from the environment, the attitude needs to be continuously monitored and adjusted. An array of sensors provide the information required to compute corrective commands and issue them to the actuators. An optimal attitude is needed to support the needs of payload instruments.

The AOCS architecture is an instance of a layered architecture shown in Figure 1. On the highest layer is *Mode Manager (MM)*. It controls *Unit Manager (UM)*, which, in its turn, is responsible for a number of hardware units. The AOCS system has seven units – four sensors (Star tracker, Sun Sensor, Earth Sensor and Global Positioning system), two actuators (Reaction Wheel and Thruster), and one payload instrument producing mission measurements. *UM* provides a generic interface to units. It hides from *MM* the number and types of units, while monitoring their states, modes and error flags. *MM* is responsible for implementing the AOCS mode logic. The predefined mode scenario defines the sequence of steps needed to reach the state where the payload instrument is ready to perform its tasks. This sequence includes the following modes:

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

Mode Manager While modelling AOCS, we assume that there is a cyclic scheduler that invokes *MM* at each execution cycle. Our generic specification template for defining the interface of a mode manager is shown in Figure 5.

INTERFACE $Mode_Manager$
...
INVARIANT
...
$next_target = previous_target \implies next_target = last_mode$
$next_target \neq previous_target \implies$
$next_target \mapsto previous_target \in Next \wedge previous_target \mapsto next_target \in Next$
$last_mode \mapsto previous_target \in Next \cup Next^{-1}$
$last_mode \mapsto next_target \in Next \cup Next^{-1}$
OPERATIONS

$SetTargetMode(r) =$ **ANY** m
PRE
$error = NoError \wedge m \in MODES \wedge m \neq next_target \wedge m \mapsto next_target \in Next \cup Next^{-1}$
POST
$r' = last_mode \wedge previous_target' = next_target \wedge next_target' = m$

$ResetError(r) =$ **PRE** $error \neq NoError$ **POST** $r' = last_mode \wedge error' = NoError$

$RunStable(r) =$ **PRE** $next_target \neq previous_target \wedge error = NoError$
POST
$r' = last_mode \wedge error' \in ERROR \wedge previous_target \mapsto next_target' \in Next \cup Next^{-1}$

$RunNotStable(r) =$ **PRE** $next_target = previous_target \wedge error = NoError$
POST
$r' = last_mode \wedge error' \in ERROR \wedge$
$(last_mode' \mapsto next_target \in Next \cup Next^{-1} \wedge next_target' = next_target \wedge$
$previous_target' = previous_target) \vee (next_target' = next_target \wedge$
$previous_target' = next_target \wedge last_mode' = previous_target')$

Fig. 5. Specification of the Mode Manager Interface (an excerpt)

The first refinement step of *MM* is an abstract implementation of the interface operations. At this stage, the operations *ResetError* and *SetTargetMode* are each refined by single events. *RunStable* and *RunNotStable* have more complex postconditions and thus have to be represented by several events. The operation *RunStable* is realised by three events: an event for successful cycle (*run_success*), an event for mode advance (*run_mode_scenario*), and an event modelling error handling (*run_failure*).

$run_success =$ **WHERE**
$next_target = previous_target$
$error = NoError$
THEN
$WorkCycle_r := last_mode$

$run_failure =$ **WHERE**
$next_target = previous_target$
$error = NoError$
THEN
$WorkCycle_r := last_mode$
$error :| error' \neq NoError$

$run_mode_scenario =$ **ANY** m
WHERE
$next_target = previous_target$
$error = NoError \wedge m \in MODES$
$previous_target \mapsto m \in Next \cup Next^{-1}$
THEN
$WorkCycle_r := last_mode$
$next_target := m$

Likewise, *RunNotStable* is implemented by the events modelling situations when no mode advance happens, when the mode advances but the target mode has not been reached and, finally, when the target has been successfully reached.

In addition to some preparatory steps towards integration with *UM*, the next refinement step also specifies the predefined mode scenario enforced by *MM*. The scenario is defined as a constant relation on modes such that the *Next* relation a transitive closure of it. The constant function *Scenario* defines a linear progression of modes from *Off* to *Science*. The axioms connecting *Scenario* with the relation *Next* are necessary to demonstrate correctness of the refinement step. The event *run_mode_scenario* uses *Scenario* to select the next mode.

$$Scenario = \{OFF \mapsto STANDBY, STANDBY \mapsto SAFE, SAFE \mapsto NOMINAL, NOMINAL \mapsto PREPARATION, PREPARATION \mapsto SCIENCE\}$$
$$Scenario \subseteq Next \ \wedge \ Next; Scenario \subseteq Next \ \wedge \ (\forall z \cdot Scenario \subseteq z \wedge z; Scenario \subseteq z \Rightarrow Next \subseteq z)$$

Integration with the Unit Manager is the most complex refinement step in our development. Since we want to build a model adaptable to various hardware configurations, the *UM* interface comes in a parameterised form: some of its sets and constants may be instantiated at the point of composition. Instantiation examples for the correspondence relation between *MM* and *UM* modes (um_mode) and the *UM* mode scenario ($um_Scenario$) are given below:

$$um_mode = \{OFF \mapsto um_InitMode, STANDBY \mapsto um_InitMode, SAFE \mapsto um_NAV_EARTH, SAFE \mapsto um_NAV_SUN, NOMINAL \mapsto um_NAV_ADV, PREPARATION \mapsto um_NAV_FINE, SCIENCE \mapsto um_NAV_INSTR\}$$
$$um_Scenario = \{um_InitMode \mapsto um_NAV_EARTH, um_InitMode \mapsto um_NAV_SUN, um_NAV_EARTH \mapsto um_NAV_ADV, um_NAV_SUN \mapsto um_NAV_ADV, um_NAV_FINE \mapsto um_NAV_INSTR\}$$

Here and further, $um_$ is a module instantiation prefix, i.e., all the constants, sets and variable starting with $um_$ are a part of this particular module.

The *UM* linking invariant presented below specifies that the modes of *UM* and *MM* are in the correspondence relation defined during the *UM* instantiation. The last condition also states that there may be periods when the *UM* error flag is set but *MM* has not yet decided about its recovery actions. This allows the Mode Manager to mask errors by recovering locally rather than propagating an error. The module instantiation data and the correspondence relation together define the mode consistency conditions.

$$next_target = previous_target \ \Rightarrow \ last_mode \mapsto um_last_mode \in um_mode$$
$$next_target = previous_target \ \Rightarrow \ next_target \mapsto um_next_target \in um_mode$$
$$next_target = previous_target \ \Rightarrow \ previous_target \mapsto um_previous_target \in um_mode$$
$$async = FALSE \wedge um_error \neq um_NoError \ \Rightarrow \ error \neq NoError$$

Further refinement steps of the *MM* module introduce the control loop that queries the sensors and sends commands to the actuators.

Unit Manager *UM* is a generic module that can be configured during instantiation to any required hardware set-up. Having a parameterised interface allows a modeller to prove stronger properties by providing additional information during composition. In the case of *UM*, the parameters are $Modes$ and $Next$.

Since we are applying the same development template once again, the general development strategy is similar to the one of *MM*, e.g., the initial refinement steps are done to prepare for integration with unit modules. The instantiation and linking invariants of the units modules have a similar structure as well.

In our AOCS development generic specification and refinement patterns (proposed in Section 4) were applied several times to construct each major part of the system. The modelling was carried out in the Rodin Platform [19], extended by the modularisation plug-in [15]. The respective proof obligations were discharged using a collection of the provided automated theorem provers with a small number of interactive proofs. Full Event B models can be found in [9].

6 Related Work

Formal validation of the mode logic and, in particular, fault tolerance mechanisms of satellite software has been undertaken by Rugina et al [17]. 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.

The mode-rich systems have been studied to investigate the problem of mode confusion and automation surprises [4, 18]. 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 [4, 8, 18], while [5] relied on theorem proving in PVS. Our approach focuses on designing fully automatic systems and ensuring their mode consistency. Unlike [8], in our approach we also emphasize the complex relationships between system fault tolerance and the mode logic.

In our previous work [7], 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 [14]. According to this approach, a mode-centric specification of the system neither defines how the system operates in some specific mode nor how mode transitions occur. It rather imposes restrictions on concrete implementations. In this paper we have demonstrated how to combine reasoning about the system mode logic and its functioning.

7 Conclusions

In this paper we have proposed a formal approach to development of mode-rich layered systems. It 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.

Our approach has been used in the development of AOCS. The approach has demonstrated good potential for facilitating design of complex mode-rich systems. Moreover, its support for formulating and verifying mode invariants has been especially appreciated in the industrial settings, since these invariants can be directly translated into assertions on the code level.

We are planning to further develop the proposed approach to enable reasoning about mode consistency in the presence of dynamic reconfiguration. Moreover, it would be also interesting to investigate how the mode ordering properties (when applicable) are inter-related with mode consistency.

Acknowledgments

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

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. J.-R. Abrial. *Modelling in Event-B*. Cambridge University Press, 2010.
3. R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Aspects of Computing*, 8(3), pp.1-23, 1996.
4. B. Buth. Analysing mode confusion: An approach using fdr2. In *Proceedings of SAFECOMP*, pages 101–114. Springer, LNCS, Vol. 3219, 2004.
5. R. W. Butler. An introduction to requirements capture using PVS: Specification of a simple autopilot. Technical report, NASA TM-110255, May 1996.
6. DEPLOY Deliverable D20 – Report on Pilot Deployment in the Space Sector. FP7 ICT DEPLOY Project. January 2010. Online at http://www.deploy-project.eu/.
7. F. Dotti, A. Iliasov, L. Ribeiro, and A. Romanovsky. Modal Systems: Specification, Refinement and Realisation. Conference on Formal Engineering Methods - ICFEM 09, Rio de Janeiro, Brazil. Springer, LNCS, Vol. 5885, December 2009.
8. M. Heimdahl and N. Leveson. Completeness and Consistency in Hierarchical State-Based Requirements. IEEE Transactions on Software Engineering, Vol.22, No.6, pp. 363-377, June 1996.
9. A. Iliasov, L. Laibinis, and E. Troubitsyna. An Event-B model of the Attitude and Orbit Control System. http://deploy-eprints.ecs.soton.ac.uk/.
10. A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, D. Ilic, and 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.
11. Industrial deployment of system engineering methods providing high dependability and productivity (DEPLOY). IST FP7 project, http://www.deploy-project.eu/.
12. L. Laibinis and E. Troubitsyna. Fault tolerance in a layered architecture: a general specification pattern in B. In *Proc. of the 2nd Int. Conference on Software Engineering and Formal Methods (SEFM), Beijing*, pp. 346-355, IEEE Press, 2004.
13. N. Leveson, L. D. Pinnel, S. D. Sandys, S. Koga, and J. D. Reese. Analyzing Software Specifications for Mode Confusion Potential. In Proc. of Workshop on Human Error and System Development, pg. 132-146, Glasgow, Scotland, 1997.
14. I. Lopatkin, A. Iliasov, and A. Romanovsky. On fault tolerance reuse during refinement. In *Proc. of 2nd International Workshop on Software Engineering for Resilient Systems*, April 2010.
15. RODIN modularisation plug-in. Documentation at http://wiki.event-b.org/index.php/Modularisation_Plug-in.
16. B. Rubel. Patterns for Generating a Layered Architecture. In J.O. Coplien, D.C. Schmidt (Eds.), Pattern Languages of Program Design, Addison-Wesley, 1995.
17. A. E. Rugina, J. P. Blanquart, and 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.
18. J. Rushby. Using model checking to help discover mode confusion and other automation suprises. In *Reliability Engineering and System Safety, Vol.75*, pages 167–177, 2002.
19. The RODIN platform. Online at http://rodin-b-sharp.sourceforge.net/.