Developing Mode-Rich Satellite Software by Refinement in Event B

Alexei Iliasov[1], Elena Troubitsyna[2], Linas Laibinis[2], Alexander Romanovsky[1], Kimmo Varpaaniemi[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}@ssf.fi

Abstract. To ensure dependability of on-board satellite systems, the designers should, in particular, guarantee correct implementation of the mode transition scheme, i.e., ensure that the states of the system components are consistent with the global system mode. However, there is still a lack of scalable approaches to formal verification of correctness of complex mode transitions. In this paper we present a formal development of an Attitude and Orbit Control System (AOCS) undertaken within the ICT DEPLOY project. AOCS is a complex mode-rich system, which has an intricate mode-transition scheme. We show that refinement in Event B provides the engineers with a scalable formal technique that enables both development of mode-rich systems and proof-based verification of their mode consistency.

1 Introduction

Currently the use of formal methods in the industrial practice is getting a new momentum. For instance, in the EU FP7 Integrated Project Deploy [13] the project partners work on advancing methods and tools for refinement based-development and verification. The goal of the project is to enable deployment of these techniques in the industrial practice. Recently, Space Systems Finland in cooperation with the academic partners has undertaken a formal development of the Attitude and Orbit Control System within the Event B framework. In this paper we present this development and discuss the lessons learnt.

The Attitude and Orbit Control System (AOCS) [6] is a generic component of satellite onboard software. The main purpose of AOCS is to achieve and maintain optimal attitude of a satellite. While achieving it, the system components and the overall system correspondingly go through several stages, called *operational modes*. These modes are mutually exclusive sets of the system behaviour [9, 14], and form a useful structuring concept that facilitates design of dependable systems in various domains. AOCS 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

one is long running (i.e., non-instantaneous) mode transitions that are caused by slow dynamics of the involved electro-mechanical components. The second characteristic is an integration of error recovery with 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 [18].

In this paper we demonstrate how to employ a correct-by-construction development approach to circumvent this problem. We use the Event B framework [2, 16] (extended with modularisation capabilities [11]) as our modelling language. The Rodin platform [20] and its modularisation plug-in [17] provide us with an automated modelling and verification environment. We define a generic module interface for mode-rich components and demonstrate how to create different mode-managing AOCS components by instantiating the generic module. We develop the system in a layered fashion, i.e., by gradually unfolding system architectural layers while proving consistency between mode transitions on adjacent layers. This approach allows us to cope 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 Event B extended with modularisation facilities shows good potential for the use in the industrial practice.

2 Event B

We start by briefly describing our development framework. The Event B formalism [2, 16] is an extension of the B Method [1], a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. Event B enables modelling of event-based (reactive) systems by incorporating the ideas of the Action Systems formalism [3] into the B Method. Event B is actively used within the FP7 ICT project DEPLOY to develop dependable systems from various domains.

2.1 Modelling and Refinement in Event B

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 = ...
E_1 = ...
...
E_N = ...
END

Such a specification encapsulates a local state (program 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 (parameters), the guard g is a state predicate, and the action S is a statement (assignment). In case when vl is 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 guard g defines the conditions for the statement to be executed, i.e., when the event is *enabled*.

The statement S can be either a deterministic assignment to the variables or a non-deterministic assignment from a given set or according to a given post-condition. One way to denote a non-deterministic assignment is $v :\in Set$, where Set is an non-empty set (or type) of possible values that can be assigned to v.

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

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

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

where BA_e is a before-after predicate relating the variable values before and after the event e. The semantic for each concrete B statement is given in the form of a predefined before-after predicate.

The main development methodology of Event B is refinement – the process of transforming an abstract specification by gradually introducing implementation details while preserving correctness. Refinement allows us to reduce non-determinism present in an abstract model. It can also introduce new 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 [20], 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 80%) in proving.

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 [11, 17] – 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
        res ← op1 =
            ANY par
            PRE M_Guard1(c, s, par, w)
            POST M_Post1(c, s, par, w, w', res')
            END
... END
```

Fig. 1. Interface Component

The primed variables in the operation postcondition stand for the final variable values after operation execution. If some primed variables are not mentioned, this means that the corresponding variables are unchanged by an operation.

A module development always starts with the design of an interface. After 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 machine can invoke the operations of M and read the external variables of M. To make a specification of a module generic, in MI_Context we can define some constants and sets (types) as parameters. The properties over these sets and constants define the constraints to be verified when the module is instantiated.

Module instantiation allows us to create several instances of the same module. Different instances of a module operate on disjoint state spaces. 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.

In the next section we demonstrate the use of Event B extended with modularisation capabilities in the development of AOCS.

3 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 disturbances of the environment, the attitude needs to be continuously monitored and adjusted. An optimal attitude is required to support the needs of payload instruments and to fulfill the mission of the satellite.

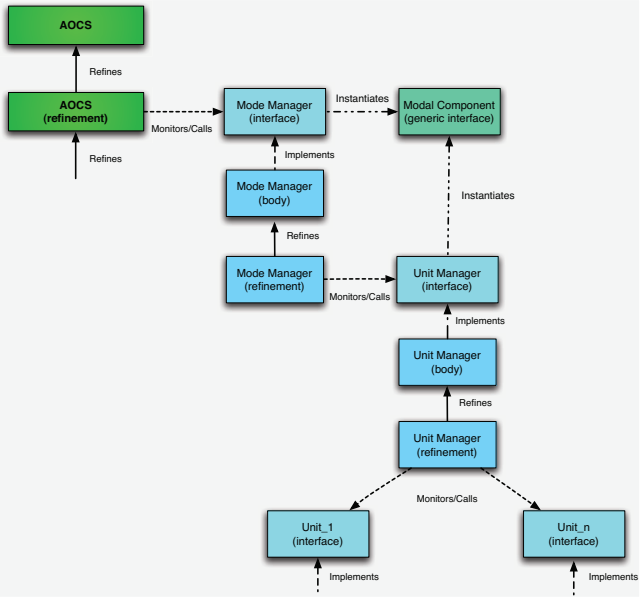

Fig. 2. AOCS Development Hierachy

In general, the behaviour of AOCS is cyclic. At each iteration the sensors provide the control algorithms with various measurements. They are used to generate the commands to the actuators that adjust the positioning of the spacecraft to ensure correct pointing of the payload instrument. AOCS consists of seven physical units: four sensors, two actuators and the payload instrument.

We formally develop the AOCS system as follows. Our initial specification models the overall system in an abstract way. The following refinements introduce implementation details in a structured manner, by unfolding system components and gradually delegating part of system functionality to them. Moreover, we identify a generic template for such components in the form of a generic module interface. Actual components will be introduced by instantiating this template, thus formally decomposing the overall system in a structured and well-defined way. The general development structure is presented in Figure 2.

On the architectural level, such a refinement strategy corresponds to gradual unfolding of system layers. The control logic of the system components residing on different layers is expressed in the terms of operational modes and their transitions. One of the main objectives of the AOCS formal development is ensure mode consistency of different layer components. The case study presented below is based on our previous work on formalisation of mode-rich systems [12].

3.1 Abstract Model

The purpose of the system is to position a satellite so that scientific instruments are oriented towards a particular region of Earth. At the most abstract level, we capture this as a succession of two atomic steps: the preparation step, orienting the satellite, and the activation step, initiating the instrument operation. Each step is associated with a boolean flag. The system is in the preparation stage

when $pr = FALSE$, is in the activation stage when $pr = TRUE \wedge act = FALSE$ and, finally, it has activated the instrument when $act = TRUE$.

Whenever a non-recoverable error occurs ($err = TRUE$), the system enters a permanently disabled state (until the underlying hardware platform is reset). It is possible for the preparation step to be interrupted by a recoverable error. In such a situation, the preparation is restarted. In this abstract model this is depicted by a non-deterministic assignment $pr :\in BOOL$.

```
MACHINE aocs
  VARIABLES pr, act, err
  INVARIANT
      pr ∈ BOOL ∧ act ∈ BOOL ∧ err ∈ BOOL
      pr = FALSE ⇒ act = FALSE
      err = TRUE ⇒ pr = FALSE ∧ act = FALSE
  INITIALISATION
      pr, act, err := FALSE, FALSE, FALSE
  EVENTS
      preparation  =  WHEN err = FALSE ∧ pr = FALSE THEN pr :∈ BOOL END
      activation   =  WHEN
                          err = FALSE ∧ pr = TRUE ∧ act = FALSE
                      THEN
                          act := TRUE
                      END
      recovery     =  WHEN err = FALSE THEN pr, act := FALSE, FALSE END
      error        =  BEGIN err, pr, act := TRUE, FALSE, FALSE END
END
```

The model at this stage is just a simple state transition system. This is done to portray the high-level properties of the system in clear and concise terms.

At some point, the *AOCS* development is decomposed into two independent strands. One focuses on unfolding of the functionality abstracted by the *preparation* event. The other deals with activation of the scientific instruments by expanding the *activity* event of the abstract model. To obtain two independent developments, we show how to refine a machine into the composition of a refined machine and a module. The composition with a module, while being a part of the refinement process, is also a formal proof of a model decomposition. As a result, we decompose the overall AOCS specification into a top level component (a refinement of the *aocs* machine) and a subsystem in charge of the initialisation and control of the positioning hardware. The subsystem is responsible for the positioning of the satellite and the execution of necessary corrective actions.

3.2 Modal Component

To single out the preparation subsystem into a separate development, we start by defining a module interface specifying the contract between the subsystem and the environment. Let us note that derivations of this generic interface will be used several times to structure the development into subsystems.

Our structuring strategy is to identify subsystems that are components of a cyclic control system. As any control system, it observes environment changes and controls the actuators. The control logic, though, is fragmented. Each such fragment deals with a specific class of environment and subsystem conditions. In our previous research, we have proposed to apply the notion of operational

INTERFACE ModalComponent
VARIABLES $last, prev, next, error$
SEES ModalContext
INVARIANT
inv1 : $last \in \text{MODE} \wedge next \in \text{MODE} \wedge prev \in \text{MODE} \wedge error \in \text{ERROR}$
inv2 : $next = prev \implies next = last$
inv3 : $next \neq prev \Rightarrow next \mapsto prev \in \text{ORDER} \vee prev \mapsto next \in \text{ORDER}$
inv4 : $\{last \mapsto prev, last \mapsto next\} \subseteq \text{ORDER} \cup \text{ORDER}^{-1}$
INITIALISATION
$last, prev, next := \text{InitMode}, \text{InitMode}, \text{InitMode}$
$error := \text{NoError}$
OPERATIONS
$r \leftarrow$ ToMode = ANY m PRE $error = \text{NoError} \wedge m \in \text{MODE}$ $m \neq next \wedge m \mapsto next \in \text{ORDER} \cup \text{ORDER}^{-1}$ POST $r' = last \ \wedge \ prev' = next \ \wedge \ next' = m$ END
$r \leftarrow$ ResetError = PRE $error \neq \text{NoError}$ POST $r' = last \ \wedge \ error' = \text{NoError}$ END
$r \leftarrow$ Mode_Advance = PRE $next = prev \wedge error = \text{NoError}$ POST $r' = last \ \wedge \ error' \in \text{ERROR} \ \wedge \ prev \mapsto next' \in \text{ORDER}$ END
$r \leftarrow$ Continuation = PRE $next \neq prev \wedge error = \text{NoError}$ POST $r' = last' \ \wedge \ error' \in \text{ERROR} \wedge$ $last' \mapsto next \in \text{ORDER} \cup \text{ORDER}^{-1} \wedge$ $((last' \neq next \wedge prev' = prev) \vee (last' = next \wedge prev' = last'))$ END
END

Fig. 3. Generic Modal Component Interface.

modes in the formal development of such systems [12]. The essential idea is that a mode-rich control system evolves in two dimensions: as a conventional control system and as a mode transition system.

A mode can be seen as an encapsulation of a piece of the control logic. Hence, a mode transition is a change in the set of control laws. In such class of systems, it is typical to have a mode comparing relation such that a 'better' mode satisfies stronger constraints. While attending to its sensor/control/actuator duties, a mode-rich control system also tries to progress towards a more advanced mode. In the process of this it may encounter adverse environment conditions and switch to a more basic (i.e., degraded) mode.

In this section we give the definition of a generic module interface (see Figure 3) for mode-rich control systems. It is essentially a template that we will use several times in our development. The interface declares four variables. The detected component errors are modelled by the variable *error*. The remaining three variables characterize the mode transitioning part of the component:

- *last* signifies the last successfully reached mode;
- *next* signifies the target mode a component is currently in transition to;
- *prev* signifies the previous mode that a component was in transition to (though it has not necessarily reached it).

These variables describe the actual mode of a component and also the mode transition dynamics. Based on their values, an environment is able to tell whether the component has settled in a stable mode ($last = prev \wedge next = prev$), is working

towards a more advanced mode ($last = prev \ \wedge \ prev \mapsto next \in \text{ORDER}$), or is degrading its mode due to error recovery ($prev \mapsto next \in \text{ORDER}^{-1}$).

The operation ToMode can be called by an upper layer component to set a new target mode. The operation ResetError is to clear the raised error flag when the detected error is being handled. Finally, the operations Preparation and Continuation model the component behaviour when it receives the control while being correspondingly in a stable or a mode transitional state.

The interface constants MODE, InitMode, ORDER, ERROR, NoError, which are defined in a separate context component, contribute to abstract characterization of the mode logic. MODE is a set of possible modes of a component, ORDER is a relation containing all the allowed mode transitions, InitMode is a predefined initial mode, ERROR is a set of component errors, and NoError is a special value denoting the absence of errors.

CONTEXT ModalContext
  CONSTANTS MODE, InitMode, ORDER, ERROR, NoError
  AXIOMS
    axm1 : InitMode ∈ MODE
    axm2 : ORDER ∈ MODE ↔ MODE
    axm3 : id ⊆ ORDER
    axm4 : ORDER ∩ ORDER^{-1} ⊆ id
    axm5 : ORDER; ORDER ⊆ ORDER
    axm6 : NoError ∈ ERROR
    axm7 : ERROR \ NoError ≠ ∅
END

where id is an identity relation and ";" stands for relational composition.

The relation ORDER also defines a partial order on modes ($axm3$, $axm4$, and $axm5$ express, correspondingly, the reflexivity, antisymmetry and transitivity properties). For any two modes, it states whether the modes are comparable and, if they are, which one of them is closer to the top mode.

3.3 Mode Manager Interface

The new subsystem introduced in the development is called Mode Manager. It is a control system with its own set of modes and an internal mode transition scenario. The Mode Manager interface is the product of extending (instantiating) the generic module interface.

INTERFACE ModeManager EXTENDS ModalComponent
  SEES ModeManagerContext

More specifically, the set of modes and the mode ordering relation are given concrete definitions at the interface level. The following is the definition of the Mode Manager context.

CONTEXT ModeManagerContext
  ...
  AXIOMS
    iaxm1 : MODE = {OFF, STANDBY, SAFE, NOMINAL, PREPARATION, SCIENCE}
    iaxm2 : Scenario = {OFF ↦ STANDBY, STANDBY ↦ SAFE, SAFE ↦ NOMINAL,
            NOMINAL ↦ PREPARATION, PREPARATION ↦ SCIENCE}
    iaxm3 : ORDER = closure(Scenario)
    iaxm4 : OFF = InitMode
    iaxm5 : partition(ERROR, RecovErrors, UnrecovErrors, {NoError}}
    iaxm6 : RecovErrors ≠ ∅ ∧ UnrecovErrors ≠ ∅

In the above, *Scenario* defines the sequence of steps needed to bring the system to the mode where the scientific payload instrument is ready to perform its tasks. This sequence consists of the following modes: *OFF* - the satellite is in this mode right after system (re)booting; *STANDBY* - this mode is maintained until the separation from the launcher; *SAFE* - a stable attitude is acquired, which allows the coarse pointing control; *NOMINAL* - the satellite is trying to reach the fine pointing control, which is needed to use the 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 it is needed.

Let us note that *Scenario* is merely a helper construct used to constrain the ORDER relation. Specifically, ORDER is defined as relational closure of *Scenario*. Moreover, the abstract set $ERROR$ is now partitioned into the disjoint parts $RecovErrors$, $UnrecovErrors$, and the predefined constant $NoError$.

First Refinement To integrate Mode Manager with the main development, the (instantiated) Mode Manager interface is included into a refinement of the abstract *aocs* machine. The refined machine *aocs1* imports the module *ModeManager* and thus has the read access to the module interface variables. The first step in decomposition refinement is to link the *aocs1* state with that of the imported module. In our case, the link is quite strong. In fact, we are able to replace the abstract variable *pr* with an expression on the module variables.

```
REFINEMENT aocs1
  REFINES aocs
  USES ModeManager
  INVARIANT
      inv1 :  error ∉ UnrecovErrors ⇒ err = FALSE
      inv2 :  pr = TRUE ⇔ (next = last ∧ last = SCIENCE)
  ...
```

In the model fragment above, $inv1$ expresses the connection between global and local errors. Intuitively, it means that the Mode Manager component is currently the only source of errors (though some errors may be tolerated). $inv2$ expresses a connection between the mode logic of Mode Manager and the state of preparedness of the abstract model. Here we simply state that the preparation is complete once Mode Manager has reached the *SCIENCE* mode.

The second step of decomposition is the integration of the Mode Manager operations into the functionality of the top-level component. The abstract event *preparation* is refined into a pair of events.

```
mode_advance REF preparation  =  WHEN
                                     error = NoError ∧ last ≠ SCIENCE
                                     last = prev
                                 THEN
                                     Mode_Advance
                                 END
intermediate REF preparation  =  WHEN
                                     error = NoError ∧ last ≠ SCIENCE
                                     last ≠ prev
                                 THEN
                                     Continuation
                                 END
```

Here Mode_Advance and Continuation use a shortcut notation for an operation call where the return value is ignored. Both events refine *preparation* and use subsystem operations to advance the model state. The events try to accomplish the same goal – reach the mode SCIENCE. The first one is enabled when Mode Manager is in a stable mode, while the second addresses the case when a mode transition is on its way. These events do not assign to the *a*ocs variables and thus this part of the system functionality is completely delegated to Mode Manager.

The other group of events deals with error conditions. Mode Manager distinguishes unrecoverable and recoverable errors. Sometimes, the system would simply remove an error, treating it as recoverable one. This is an abstraction of the error handling activity at this level. In other cases, to recover from an error, it may be necessary to reconfigure Mode Manager. This happens when there is a malfunction in some hardware unit and, as a result, the unit must be switched off to put the system into a healthy state. Since the failed unit is no longer available, the Mode Manager mode is downgraded to the one where the system does not need the failed unit. Since the system is cyclic, once the error is cleared, the preparation would restart and attempt to switch on the failed unit.

```
recovery = ANY m WHERE
               m ↦ next ∈ ORDER^-1
               error ∈ RecovErrors
           THEN
               ResetError
               ToMode(m)
               act := FALSE
           END
error    = WHEN error ∈ UnrecovErrors THEN err, act := TRUE, FALSE END
```

3.4 Mode Manager

Let us now consider the Mode Manager development. It starts with an Event B machine implementing the Mode Manager interface. For each interface operation, there is one event group realising the operation. Some groups events are *final* designating the group exit point – the terminal events returning the control to the calling environment. An event that is not final must pass control to another event in the same event group. The following is an excerpt from the abstract machine of the Mode Manager development.

```
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 ∈ ORDER ∪ ORDER^-1
                          THEN
                              last := m ‖ error :∈ ERROR
                          END
      FINAL adv_comp    = WHEN
                              next ≠ prev
                          THEN
                              error :∈ ERROR ‖ last := next ‖ prev := next
                          END
... END
```

The Continuation operation is realised by a group containing three events. The event *adv_skip* models the behaviour when no mode change happens during the call. This is needed to model mode transitions that take substantial time and thus are spread over several control cycles. A transition to some intermediate mode is modelled by *adv_partial*. Intermediate modes are observed when a component is progressing to some mode that is not reachable directly from the current mode. Finally, *adv_comp* specifies when the system successfully reached the target mode (and thus arrived to a stable state).

Mode Manager does not directly control the satellite hardware. Instead it relies on a special subsystem, called Unit Manager. The purpose of Unit Manager is to abstract the specifics of a hardware configuration and provide a simple common control interface to the hardware. We approach Unit Manager design as another instance of a mode-rich control system.

Unit Manager Interface The Unit Manager interface is a specialisation of the generic interface defined in Figure 3. Like Mode Manager, it defines its own set of modes and a mode transition scenario.

```
INTERFACE UnitManager EXTENDS ModalComponent
  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. NAV_ADV and NAV_FINE use the GPS unit to compute the satellite position in respect to the Earth surface. The mode NAV_INSTR is the final target mode meaning that the scientific instrument hardware is enabled.

```
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}
END
```

Unit Manager Integration After a number of refinement steps, the Mode Manager development is decomposed to separate the Unit Manager development. The link between the two developments is quite tight. Mode Manager relies on Unit Manager in most of its operations as Mode Manager does not have a direct access to the controlled hardware. The required mode consistency between these components is defined as a a relation linking the modes of Mode Manager and Unit Manager. Moreover, the added invariant properties (in the Mode Manager model) guarantee that the modes of two components are always in agreement with each other. A model excerpt specifying this is given in Figure 4.

The mode mapping relation is defined as the constant *um_mode* under the *USES* clause. To avoid name clashes, the Unit Manager module is instantiated with the prefix *um*. Consequently, all the names imported from the module appear with the prefix.

MACHINE MMBody3
...
USES um : UnitManager
CONSTANTS um_mode
AXIOMS
$um\_mode = \{\text{OFF} \mapsto \text{um\_InitMode}, \text{STANDBY} \mapsto \text{um\_InitMode},$
$\text{SAFE} \mapsto \text{um\_NAV\_EARTH}, \text{SAFE} \mapsto \text{um\_NAV\_SUN},$
$\text{NOMINAL} \mapsto \text{um\_NAV\_ADV}, \text{PREPARATION} \mapsto \text{um\_NAV\_FINE},$
$\text{SCIENCE} \mapsto \text{um\_NAV\_INSTR}\}$
...
INVARIANT
...
gi1 : $next = prev \Rightarrow last \mapsto um\_last \in um\_mode$
gi2 : $next = prev \Rightarrow next \mapsto um\_next \in um\_mode$
gi3 : $next = prev \Rightarrow prev \mapsto um\_prev \in um\_mode$
gi4 : $async = FALSE \wedge um\_error \neq \text{um\_NoError} \implies error \neq \text{NoError}$
...
END

Fig. 4. Unit Manager Integration

The gluing invariants, $gi1, ..., g4$, define the correspondence between the Mode Manager and Unit Manager modes and errors. All the events of Mode Manager must maintain this correspondence. As a result, an update of the Unit Manager mode often necessitates an update of the Mode Manager mode.

The Unit Manager development, in its turn, is split 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 a version of the generic Modal Component interface. However, unlike Mode Manager and Unit Manager, the hardware units are not a part of the control logic we are developing. Collectively, the units define the environment of the system and thus are only characterised by their interfaces.

3.5 Unit Interface

The hardware unit subsystems differ by their set of modes and mode transition rules. Each one also define its own set of error conditions. Instead of defining an extended interface for each individual unit we use a single parameterised interface. Consequently, unit modes and mode transitions are specified at the point of module integration.

INTERFACE UnitComponent EXTENDS ModalComponent
PARAMETERS MODE, InitMode, ORDER, ERROR, NoError

In the specific hardware configuration that we are modelling there are six hardware units. To construct a faithful model close to the executable program, we explicitly introduce each unit subsystem by importing the (correspondingly instantiated) generic module interface.

4 Lessons Learnt

The AOCS system described here is a modified (due to confidentiality reasons) version of a realistic 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 with a formal AOCS development to ensure correctness of mode transitions.

The initial attempt [21] to formally develop a system was rather unsuccessful. This modelling was significantly influenced by the code that was developed for the real AOCS. It started from modelling the overall control cycle that consisted of a sequence of events abstractly modelling the entire system structure and functionality – the mode manager, the unit manager and fault tolerance mechanisms. Then, in the further refinement steps, we had to introduce a large number of variables and events (modelling program counters and procedure calls) to continue representing interdependencies between the system components and functions. Moreover, at the time of this development, Event B was still lacking modularisation support. As a result, fairly soon the developed monolithic model became unreadable for the developers and unmanageable for the Rodin platform. We 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 different level components;
- It is important to maintain readability of models.

This second development attempt [10] was preceded by a preparatory work that aimed at alleviating discovered problems. We have developed a modularisation plug-in [17] implementing the modularisation extension for Event B that we have proposed previously [11]. Moreover, while formalizing reasoning about mode-rich systems [12], 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 indeed demonstrated its viability.

The second development attempt – the one which is described in this paper – achieved the desired goal. We succeeded in building a detailed AOCS model and verified (by proofs) that it correctly implements the desired mode transition scheme. 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 established the consistency of mode transitions between adjacent layers as a part of refinement verification. The specifications of components were produced as a result of instantiating the generic module interface that is common for mode managing components on different layers of abstractions.

Refinement by instantiating the generic components significantly simplified the development and proof activity. As a result, we have alleviated the problem of manipulating large monolithic models. The produced models of modules (components) are much smaller. They are also easier to understand and verify. The overall system model is also rather compact and can be easily maintained because it includes only references to the components visible state and interface.

In our development we have made a smooth transition from the architectural modelling to modelling 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 the 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.

5 Related Work

Formal validation of the mode logic and, in particular, fault tolerance mechanisms of satellite software has been undertaken by Rugina et al [18]. 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. 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, 9, 19], while [5] relied on theorem proving in PVS. Our approach focuses on designing fully automatic systems and ensuring their mode consistency. Unlike [9], 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 [15]. 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.

6 Conclusions

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, refinement-based development techniques coped well with modelling the complex mode transition scheme and verification of its correctness.

Verification of all possible mode transitions (including complex cascading effects) was done by proofs and did not require any simplifications. Currently that level of assurance cannot be delivered neither by model-checking, simulation or testing alone nor by combination of these techniques. The proposed modularisation and stepwise development style allowed us to keep manual proof efforts at a reasonable level (about 17 percent of proofs had to be carried out interactively). Hence formal verification by theorem proving has become more accessible for industry practitioners.

In the presented work we aimed at not merely experimenting with modelling a particular industrial-size system in Event B, but rather at creating a generic solution facilitating development of AOCS-like systems. Indeed, our approach to modelling mode-rich components using generic instantiation supports both reuse and composition. Such reuse is safe, since while developing a component by refinement we formally ensure its conformance to the instantiated specification of its interface. Moreover, it becomes manageable to verify composition of components whose state and behaviour are succinctly and formally modelled.

Our work can be seen as a step towards creating a formal approaches for model-driven development and establishing the reference architecture for the space sector – the two recent initiatives of European Space Agency [8]. As a future work it would be interesting to connect our approach to the languages specifically dedicated to architectural modelling. Moreover, it would be useful to continue experimenting with formal modelling of various types of architectures of mode-rich systems as well as address 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 platform grant. We also wish to thank the anonymous reviewers for their very valuable comments and suggestions.

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, Lecture Notes in Computer Science, 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, Lecture Notes in Computer Science, Vol. 5885, Springer, December 2009.
8. European Cooperation for Space Standardization. Software general requirements ECSS-E-ST-40C. 2009.
9. 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.
10. A. Iliasov, L. Laibinis, and E. Troubitsyna. An Event-B model of the Attitude and Orbit Control System. 2010. DEPLOY publication repository: http://deploy-eprints.ecs.soton.ac.uk/.
11. 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.
12. A. Iliasov, E. Troubitsyna, L. Laibinis, A. Romanovsky, K. Varpaaniemi, P. Väisänen, D. Ilic, and 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.
13. Industrial deployment of system engineering methods providing high dependability and productivity (DEPLOY). IST FP7 project, online at http://www.deploy-project.eu/.
14. N. Leveson, L. D. Pinnel, S. D. Sandys, S. Koga, and 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.
15. 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.
16. Rigorous Open Development Environment for Complex Systems (RODIN). Deliverable D7, Event B Language, online at http://rodin.cs.ncl.ac.uk/.
17. RODIN modularisation plug-in. Documentation at http://wiki.event-b.org/index.php/Modularisation_Plug-in.
18. 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.
19. 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.
20. The RODIN platform. Online at http://rodin-b-sharp.sourceforge.net/.
21. K. Varpaaniemi. Event-B Project DepSatSpec015Model000. January 2010. DEPLOY publication repository: http://deploy-eprints.ecs.soton.ac.uk/168.