COMPUTING SCIENCE

A Method for Rigorous Development of Fault-Tolerant Systems

Ilya Lopatkin and Alexander Romanovsky

TECHNICAL REPORT SERIES

No. CS-TR-1374 February 2013

TECHNICAL REPORT SERIES

No. CS-TR-1374 February, 2013

A Method for Rigorous Development of Fault-Tolerant Systems

I. Lopatkin and A. Romanovsky

Abstract

With our increasing dependency on computer-based systems, ensuring their dependability becomes one the most important concerns during system development. This is especially true for mission- and safety-critical systems. Critical systems typically use fault tolerance mechanisms to mitigate run-time errors. However, fault tolerance modelling and, in particular, rigorous definitions of fault tolerance requirements, fault assumptions and system recovery have not been given enough attention during formal system development. This paper proposes a development method for stepwise modelling of high-level system fault tolerant behaviour. The method provides an environment for explicit modelling of fault tolerance and modal aspects of system behaviour and is supported by tools that are smoothly integrated into an industry-strength development environment. A case study from the aerospace domain is used to demonstrate the proposed method.

Printed and published by Newcastle University,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

LOPATKIN, I., ROMANOVSKY, A.

A Method for Rigorous Development of Fault-Tolerant Systems
[By] I. Lopatkin, A. Romanovsky

Newcastle upon Tyne: Newcastle University: Computing Science, 2013.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1374)

Added entries

NEWCASTLE UNIVERSITY
Computing Science. Technical Report Series. CS-TR-1374

Abstract

With our increasing dependency on computer-based systems, ensuring their dependability becomes one the most important concerns during system development. This is especially true for mission- and safety-critical systems. Critical systems typically use fault tolerance mechanisms to mitigate run-time errors. However, fault tolerance modelling and, in particular, rigorous definitions of fault tolerance requirements, fault assumptions and system recovery have not been given enough attention during formal system development. This paper proposes a development method for stepwise modelling of high-level system fault tolerant behaviour. The method provides an environment for explicit modelling of fault tolerance and modal aspects of system behaviour and is supported by tools that are smoothly integrated into an industry-strength development environment. A case study from the aerospace domain is used to demonstrate the proposed method.

About the authors

Ilya Lopatkin is a PhD candidate and a Research Assistant in the Centre for Software Reliability, Newcastle University. His main research interests are system dependability, fault tolerance, cyber-physical aspects of systems, formal modelling, and system verification. In 2008 - 2012 he was involved in the FP7 DEPLOY Integrated Project on Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity. He is now involved in EPSRC/RSSB research project SafeCap on Overcoming the Railway Capacity Challenges without Undermining Rail Network Safety, and the TrAmS-2 EPSRC/UK platform grant on Trustworthy Ambient Systems.

Alexander (Sascha) Romanovsky is a Professor in the Centre for Software Reliability. He is the leader of the School's Dependability Group. His main research interests are system dependability, fault tolerance, software architectures, exception handling, error recovery, system structuring and verification of fault tolerance.
He received a M.Sc. degree in Applied Mathematics from Moscow State University and a PhD degree in Computer Science from St. Petersburg State Technical University. He was with this University from 1984 until 1996, doing research and teaching. In 1991 he worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland. In 1993 he was a visiting fellow at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993-94 he was a post-doctoral fellow with the Department of Computing Science, University of Newcastle upon Tyne.
In 1993-94 he was a post-doctoral fellow with the Department of Computing Science, University of Newcastle upon Tyne. In 1992-1998 he was involved in the Predictably Dependable Computing Systems (PDCS) ESPRIT Basic Research Action and the Design for Validation (DeVa) ESPRIT Basic Project. In 1998-2000 he worked on the Diversity in Safety Critical Software (DISCS) EPSRC/UK Project. Prof Romanovsky was a co-author of the Diversity with Off-The-Shelf Components (DOTS) EPSRC/UK Project and was involved in this project in 2001-2004. In 2000-2003 he was in the executive board of Dependable Systems of Systems (DSoS) IST Project.
In 2004-2007 he was the Coordinator of the FP6 ICT Rigorous Open Development Environment for Complex Systems Project (RODIN). In 2008-12 Prof Romanovsky was the Coordinator of the major FP7 Integrated Project on Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity (DEPLOY). The DEPLOY IP, that followed RODIN, developed the Rodin tooling environment for formal stepwise design of complex dependable systems using Event-B. Rodin is now widely used by companies in Europe, China, Japan, Canada and Brazil.
He is now the Principle Investigator of the TrAmS-2 EPSRC/UK platform grant on Trustworthy Ambient Systems (2012-16) and of the EPSRC/RSSB research project SafeCap on Overcoming the Railway Capacity Challenges without Undermining Rail Network Safety (2011-14).

Suggested keywords

DEPENDABILITY
FAULT TOLERANCE
FORMAL METHODS
REFINEMENT
PATTERNS
MULTI-VIEW DEVELOPMENT

A Method for Rigorous Development of Fault-Tolerant Systems

Ilya Lopatkin and Alexander Romanovsky

CSR, School of Computing Science, Newcastle University,
Newcastle upon Tyne, NE1 7RU, UK
{ilya.lopatkin,alexander.romanovsky}@ncl.ac.uk

Abstract. With our increasing dependency on computer-based systems, ensuring their dependability becomes one the most important concerns during system development. This is especially true for mission- and safety-critical systems. Critical systems typically use fault tolerance mechanisms to mitigate runtime errors. However, fault tolerance modelling and, in particular, rigorous definitions of fault tolerance requirements, fault assumptions and system recovery have not been given enough attention during formal system development. This paper proposes a development method for stepwise modelling of high-level system fault tolerant behaviour. The method provides an environment for explicit modelling of fault tolerance and modal aspects of system behaviour and is supported by tools that are smoothly integrated into an industry-strength development environment. A case study from the aerospace domain is used to demonstrate the proposed method.

Keywords: Dependability, fault tolerance, formal methods, refinement, patterns, multi-view development

1 Introduction

Our society is becoming increasingly dependent on computer-based systems due to the falling costs and improving capabilities of computers. There is a class of systems called *critical* that operate with resources of the highest value. Defects in such systems can have a significant impact on the environment, assets, and human life. Critical systems have to be *dependable* [4], so that they can be justifiably trusted to provide the required services.

One of the prominent solutions to ensuring systems dependability by fault prevention and/or fault removal is the inclusion of formal modelling in various stages of the software development process. Usage of formal methods in development of dependable systems is increasing and is proven to be cost-effective [12]. Among the main current obstacles to adopting formal methods by industry are the lack of tools and engineers' experience in formal development. We believe this situation can be significantly improved by teaching best practices of modelling and providing modelling guidelines and reusable solutions.

It is well-known that one cannot produce a faultless system functioning in a perfect fault-free environment [9]. While it is theoretically possible to formally

produce a system free of bugs, developers cannot assume system environment to be fault-free. A number of safety and reliability analysis techniques are being successfully used nowadays in industry such as Failure Modes and Effects Analysis (FMEA), Fault Tree Analysis, and HiP-HOPS. Deterioration of physical components makes it necessary for systems to employ *fault tolerance* mechanisms [9] in both hardware and software. Systems need to survive low-level sensor and actuator failures to provide acceptable levels of dependability properties.

There are a number of studies on formal modelling of fault tolerance. Some research is done on extending original semantics of formal methods with additional fault tolerance modelling constructs. Other techniques provide patterns and modelling styles for modelling fault tolerance within the formal semantics of a particular formalism. For example, [8] provides a guidance to modelling fault tolerant control system in the B formalism. The authors focus on modelling low-level component failures that may be masked at a system level. Another example of a style-based approach is introduced in [7]. The authors propose a style for modelling systems with a layered architecture using an exception propagation mechanism. The style implies operational "execution" of a model, and thus makes it difficult to specify and verify system-level safety properties.

We follow a pattern-based approach and propose a method for modelling high-level fault tolerant system behaviour in the Event-B formalism [3]. The method offers a refinement strategy for modelling with Event-B, and a number of modelling patterns to assist in development of fault tolerant systems. The method uses an additional viewpoint called Fault Tolerance (FT) Views for modelling modal and fault tolerant behaviour. In contrast to the above-mentioned studies, we focus on reactive models of system-level fault tolerant behaviour, and provide support for explicit reasoning about safety properties.

The paper is organised as follows. We give a brief introduction into the Event-B method in Section 2. The FT Views language is introduced in Section 3. We introduce the method, its main principles, a refinement strategy, and the most important refinement patterns in Section 4. We demonstrate method application in Section 5. Section 6 summarises the work and draws conclusions.

2 Event-B

The development method described in this thesis is exemplified on Event-B formal method [3]. Event-B is a state-based formalism closely related to Classical B and Action Systems. The Event-B development is stepwise: the basic idea is to design the system implementation gradually, by a number of correctness preserving steps called *refinements*.

The basic unit of Event-B specification is called a *machine*. A machine encapsulates a local state (program variables v) and provides operations on the state. The operations (called *events*) can be defined as

$$\textbf{ANY } p \textbf{ WHERE } H(v) \textbf{ THEN } S(v, v') \textbf{ END}$$

where p is a list of local variables (parameters), the guard $H(v)$ is a state predicate, and the action $S(v, v')$ is a statement (assignment) describing how the

system state is affected by the event. The occurrence of events represents the observable behaviour of the system. When the condition **WHERE** is satisfied, an event is *enabled* and its action can be executed. The action $S(v, v')$ can be either a deterministic or a non-deterministic assignment.

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

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

$$I(v) \wedge H(v) \Rightarrow \exists v' \cdot S(v, v')$$
$$I(v) \wedge H(v) \wedge S(v, v') \Rightarrow I(v')$$

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. For a refinement step to be valid, every possible execution of the refined machine must correspond to some execution of the abstract machine. To demonstrate that each event is a correct refinement of its abstract counterpart, we need to prove that the guard is strengthened in the refinement, and also demonstrate a correspondence between the abstract and concrete postconditions. Formally,

$$I(v) \wedge I'(v, w) \wedge H'(w) \Rightarrow H(v)$$
$$I(v) \wedge I'(v, w) \wedge H'(w) \wedge S'(w, w') \Rightarrow \exists v' \cdot S(v, v') \wedge I'(v', w')$$

where the primed expressions H', I', S' belong to the refined model. The machines linked with each other by refinement form a development chain, or in a more general case development represents a tree.

The consistency of Event-B models as well as correctness of refinement steps should be formally demonstrated by discharging *proof obligations*. The Rodin platform[1], 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 in proving (usually over 90%).

3 Fault Tolerance Views

The Fault Tolerance Views [10] (also called Modal Views) is a modelling environment for describing modal and fault tolerance aspects of a system in a concise manner while formally linking them to the main model. Much of the formal foundations of Fault Tolerance Views are based on the previous work on modelling modal systems [5].

A modal view is a graph diagram developed alongside an Event-B model. The two basic building blocks of a modal view are *mode* and *transition*. Mode is

a general characterisation of a system behaviour. It describes the functionality of a system and the operating conditions under which the system provides this functionality. A system switches from one mode to another through a mode transition. A transition can be given additional meaning in the context of fault tolerance: there are *error* and *recovery* transition types which are specialisations of the *normal* type. Relative to the transition and its type, we differentiate the fault tolerance types of modes: we say that an error originates in a *normal mode* and leads to switching to a *degraded mode* or a *recovery mode*. The recovery transition leads from the recovery mode back to normal.

A view is linked with a formal model and ensures that the model implements the features described in the view. For that, modes and transitions are mapped into groups of events. A mode is characterised by a pair of predicates A/G where:

- $A(v)$ is a mode *assumption*, a predicate over the current system state. By assumption we denote different operating conditions.
- $G(v, v')$ is a mode *guarantee*, a relation over the current and next states of the system. By guarantee we denote the functionality ensured by the system under the corresponding assumption.

With assumption and guarantee of a mode being predicates expressed on the variables of a model, we are able to impose restrictions on the way modes and transitions are mapped into model events and thus cross-check design decisions in either part. A view must satisfy internal consistency conditions, such as mode feasibility and invariant preservation, as well as consistency with the formal model. For example, the following condition applied to an event requires that the event of a mode must satisfy the mode guarantee $G(v)$ and reestablish the assumption $A(v)$:

$$I(v) \wedge A(v) \wedge H(v) \wedge S(v, v') \implies \\ [G(v, v') \wedge A(v')] \vee [\neg A(v') \wedge (A_1(v') \vee ...A_n(v'))]$$

If the same event is used for outgoing transitions it must satisfy the assumptions of one of the target modes $A_1(v')...A_n(v')$ and invalidate the source assumption.

Diagrams are built in a step-wise manner, starting from the most primitive and introducing details using our *modal refinement* process [10]. A Modal Views development comprises a chain of documents similar to the Event-B development. Modal views are built by incrementally refining modes, errors and recoveries and proving the refinement relation between each two consequent views. Views and Event-B models describe the same system and are refined in parallel.

The tool support for the Modal Views is a plug-in [2] to the Rodin Platform providing a diagram editor, a static checker, and a proof obligation generator. The full list of verification conditions (proof obligations) and details on the meaning and purpose may be found in [2, 5, 10].

4 Development Method

In this section, we describe a method for top-down development of fault tolerant systems with a focus on abstract levels of modelling. The method addresses a

number of issues of the state-of-the-art approaches and is designed to fill the existing gap in modelling and verification of abstract fault tolerant behaviour.

The development method includes the following three constituents:

- the *modelling principles* stating the key points for modelling fault tolerant systems in refinement-based methods,
- the *refinement strategy* defining a sequence of refinement steps that need to be performed to arrive at a meaningful model of a fault tolerant system, and
- a set of *modelling patterns* and *FT view templates* that provide a reuse mechanism during modelling.

The three constituents together represent modelling guidelines for building fault tolerant systems in refinement-based formal methods in a systematic way. Although the method does not require the usage of the Modal Views, it greatly benefits from additional formal constraints provided by the views. Thus, we describe the application of the Modal Views as a part of the method.

The method is designed for modelling reactive and control systems as labelled transition systems. It specifies particular steps that need to be performed over the state transition system to adequately represent the system environment and correctly model the system fault tolerant behaviour. The guidelines proposed in this chapter can be applied during modelling in any state-based formal method with interleaving semantics such as Action Systems, B, Event-B, Z, and VDM. The method is exemplified on the Event-B formalism.

4.1 Assumptions and Principles

The development method is based on a number of assumptions and principles. Our first assumption is that the system requirements were elicited prior to modelling. We also assume that the purpose of formal modelling is to specify implementable system behaviour satisfying the desired properties described in requirements. To satisfy the purpose, properties must be expressed in the model. These can be safety properties expressed as invariants and proved by a theorem prover or liveness properties verified by a model checker. The additional viewpoint as described in Section 3 serves as a source of diagrammatically represented properties that otherwise could be difficult to express in the model, or can be missed. The modal views complement the correctness criterion for the models.

The method facilitates the expression of safety properties by providing patterns that follow a *reactive style* of modelling. By the reactive modelling style we mean such a way of behaviour definition that uses atomic reactions and allows developers to express high-level properties in a form $cause \Rightarrow reaction$.

One of the most important principles used in the method is the principle of *behaviour restriction*. We treat the system model as a transition system that is "composed" of two parts: an *unconstrained behaviour* and a set of functional and fault tolerance *constraints*. An unconstrained behaviour contains all system states and all transitions, it is merely a declaration of the system structure using variables. A model without constraints has a non-deterministic behaviour as it can go from any state to any other state. The development departs from an

unconstrained declaration of the state space and step-wise arrives at a model which "behaves" in a safe and sensible manner within the given constraints, i.e. requirements.

Another important principle is the need for an adequate model of the *system environment*. This principle is induced by semantic gaps between the state-based formalisms for modelling the system logic and conventionally used languages for expressing the system environment (e.g. physical processes). This leads to a requirement for the developer: he/she must understand the nature of the variables in the model, and their future implementation in the real system.

In our method, we assume that a system observes some part of its environment and reacts to its changes. All state transitions that occur during system execution need to satisfy the *implementable causality rule*: a cause (environmental change) must not depend on a reaction (system change). In other words, a system being in a certain state may not "forbid" environment to change. Otherwise, the model would contain unrealistic assumptions about system environment that cannot be implemented.

Closely related to unimplementable assumptions are faults and the resulting errors which are inevitable phenomena of the final systems [9]. To adequately model them, we state the following as a separate *error modelling principle*: errors must be abstractly modelled from the early modelling steps where functionality depends on the environmental conditions.

In order to abstractly model system-level errors, developers need to know possible low-level errors and account for them at higher levels of behaviour specification. Therefore, one needs to plan ahead and construct an abstract model knowing possible and acceptable solutions of low-level modelling. This is the essence of *refinement planning*. The method described in this work facilitates such planning by providing a refinement strategy and a number of modelling patterns that support decision making.

4.2 Refinement Strategy

The development method prescribes a number of steps that need to be performed to arrive at a correct and meaningful model of a fault tolerant system. At each step, certain development actions are taken such as editing or refining a system state model or refining a modal view. The schematic procedure of the development method is shown on Figure 1. The development method is divided into two parts: the first part contains steps for a generic development of reactive fault tolerant systems and is applicable in any problem domain, the second part, that follows the first one, focuses on control systems and facilitates modelling of low-level components with an intention to support the implementation step.

Abstract modelling of a reactive fault tolerant system starts with defining a *failure-free* functionality of the system (Step 1). By failure-free functionality we mean the abstract behaviour that is only restricted by functional requirements.

At the first abstract level where fault tolerance requirements impact the system model, a designer has to choose an *abstract fault tolerance class* of the system (Step 2). We give more details on system FT classes in Section 4.3.

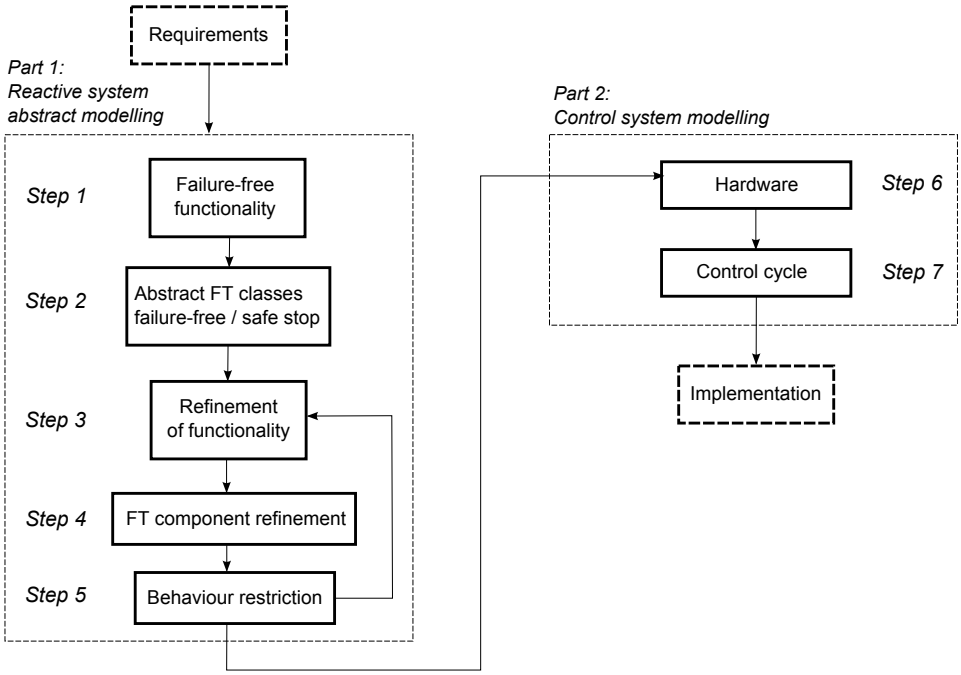

Fig. 1: The steps of the method

Steps 3, 4 and 5 are repeated iteratively until all the required properties of the reactive system behaviour are expressed and verified. Step 3 is refinement of functionality which is project-specific. Step 4 is called the *fault tolerant component refinement*: it refines the abstract component errors and FT behaviour into sub-component errors. This is described in Section 4.4. Step 5 is the *behaviour restriction* step that is used to strengthen the functional behaviour with operational conditions which is described in Section 4.5.

The second part of the method refines the reactive model into a model of a control system. The two steps performed are inclusion of low-level *hardware* units (sensors and actuators) and realisation of a *control cycle*. We briefly describe the modelling of control systems in Section 4.6.

4.3 Abstract System Fault Tolerance Classes

According to the proposed method, the first decision a developer has to make is to choose an *abstract fault tolerance class* of a system. We identify two abstract classes of systems from the fault tolerance modelling perspective: a class of *failure-free* systems, and a class of *safe stop* systems. Any system belongs to one of these classes depending on whether stop conditions are defined in requirements. The two classes of system fault tolerance are associated with two possible initial modal views accordingly (Figure 2).

Systems of the first class can mask all internal errors and operate indefinitely long. Abstract models of such systems shall contain functional behaviour under ideal environmental conditions. This is represented by a single mode *Normal* on Figure 2. The mode only represents the "normal" behaviour at the abstract level and can be refined into recoveries and/or graceful degradation at later steps.

Systems of the second class cannot tolerate certain errors and can eventually stop. The errors that can cause a system stop are called *unrecoverable*. In this work, we focus on safe stop systems and do not model consequences of system failure. If the safe stop abstraction is chosen for the system, the modeller has to apply the *safe stop pattern* to the system abstract model:

> Define a single variable representing the operational state of the system (*stopped*). One of its values shall represent the stopped state (e.g., $stopped = TRUE$). Separate the functional behaviour from the stopped state by using the declared variable. Define a transition that switches the system to the stopped state.

The pattern explicitly separates the operational system behaviour from the stopped state. On the modal view, the two are represented by modes *Normal* and *Stop* correspondingly. The stop transition depicts an abstraction of unrecoverable errors. The actual errors will refine this transition during subsequent refinement steps.

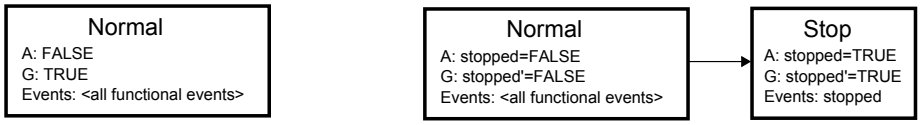

Fig. 2: Two abstract classes of fault tolerant systems

The purpose of this step is to "reserve" an abstract representation of the overall system fault tolerant behaviour for further refinements.

4.4 Fault Tolerant Component Refinement

Fault tolerant component is a structural system unit that is described by its *functional* and *error states*. For example, a door can be in a functional state of being open or closed, and in an error state of being operational or broken. In this work, we represent the fault tolerant system under development as a layered hierarchy of fault tolerant (FT) components. Each layer represents a level of abstraction with the system being the root of the hierarchy. The method traverses the hierarchy starting from its root by modelling component functional and error states at each layer. At each step, the set of FT components is decomposed into its FT subcomponents and the system behaviour is refined in terms of finer-grained FT components.

The fault tolerant component refinement step consists in applying three patterns. The first pattern to be applied is the *error state variable pattern*:

> For each component, specify a variable depicting the error state of the component at the current abstraction level.

The *safe stop pattern* described in Section 4.3 is a special case of the *error state variable pattern*. At the abstract level, the whole system is considered as a single FT component and its error state is represented by a single boolean variable.

Each two subsequent layers of FT components should be formally connected by applying the *error state invariant pattern*:

> Define a relation between the concrete and abstract error states. The relation can include functional states.

The error states are used in definition of fault tolerant behaviour using the *fault tolerant behaviour pattern*:

> All error detection transitions must refine the corresponding reaction transitions of the system. All relevant functional states have to be covered by outgoing error detection transitions to satisfy the causality principle.

Thus, error detection transitions are also system reactions: they change FT components' error states as well as functional states. Such behaviour conforms to the reactive style of modelling and allows us to express safety properties which include both functional and error states.

The refinement of system fault tolerant behaviour in the state model is accompanied by a refinement of the modal view. The refinement of modes involves both the functional and fault tolerant system behaviour and the exact way of modal refinement is project-specific. To assist in construction of modal views, we offer two templates for modal refinement.

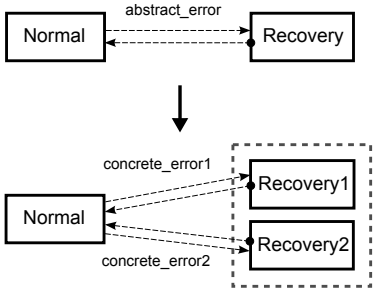

Fig. 3: Error split template

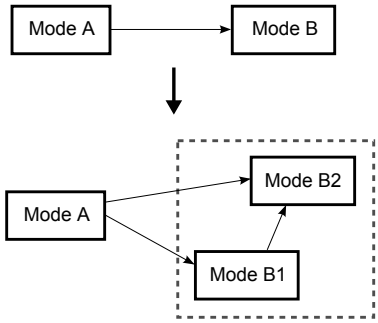

Fig. 4: Behavioural split template

The first template refines an abstract error with two or more concrete errors. The process may be repeated and the result is a family of errors derived from a single abstract error. The template is shown on Figure 3 with errors leading to recovery modes, however, they can be applied to degraded modes as well.

The second template is a behavioural template (Figure 4), it splits a system mode into a chain of two (or possibly more) consecutive modes. This template can be used to model intermediate operations that the system needs to perform to arrive at a supposedly stable mode. A particular application of the behavioural template can be used for specifying the system graceful degradation.

4.5 Behaviour Restriction

During the behaviour restriction step, refined FT component error states and modal views are used to restrict the system functional behaviour. When applied to the model, the step implements those fault tolerance requirements that define the changes in system behaviour under different operating conditions. To restrict the functional behaviour, we apply the *behaviour restriction pattern* to the system model:

> Restrict the functional transitions that are not allowed with respect to the error state variables defined earlier: use the error state variables to strengthen the domain restriction predicates of functional transitions.

Under different operational conditions, system functionality may provide different functions. For example, safety requirements might only permit a partial operation of certain components after errors have been detected. Moreover, fault tolerance requirements may contain specifications of different system behaviours under different operational conditions, e.g., graceful degradation. The behaviour restriction pattern ensures that the behavioural model satisfies such types of requirements by disallowing functional transitions that are not valid.

The pattern should be used together with modal refinement. The modal views provide consistency conditions that can be used to identify invalid functional transitions. Restriction predicates of relevant transitions are being strengthened using this pattern to satisfy the associated modal views.

We regard the assumption predicate (introduced in Section 3) as an abstraction of the system environment that should refer to error state variables. The assumptions therefore state the combinations of available components that allow the system to provide its subsets of functionality. The subsets of functionality should refer to the logical state variables and are contained in the guarantee predicates. A modal view specified in such way can be treated as a specification of the system fault tolerant behaviour.

4.6 Modelling Control Systems

The component refinement and the behaviour restriction steps are performed in a top-down manner until the reactive model of the system contains the required

safety properties. At this step, we refine the reactive model to include hardware units such as sensors and actuators, and extend it with a sequential control cycle.

To model hardware units, we apply the *fault tolerant component refinement* step to the lowest level of abstraction and define error states and error transitions for hardware units such as sensors and actuators.

At the final step, we extend the previously developed reactive system with a sequential control cycle using the *control cycle pattern*:

Define a variable depicting the current *phase* of a control cycle iteration. The control cycle consists of four phases:

1. *Sensing* phase: the system non-deterministically reads the sensor values.
2. *Error detection* phase: the system uses the new sensor readings and expected values from previous iteration to detect deviations and low-level errors.
3. *Control* phase contains the control algorithm, i.e. the reactive model of the system that was developed previously. The control part of the model is fed with low-level errors detected during the previous phase, and produces new functional component states and a set of signals for the actuators.
4. *Prediction* phase: the new states and the actuator signals are used at the final phase to predict the sensor readings at the next iteration of the cycle.

The reactive part of the system takes place of a control algorithm step of the cycle. Sequential decomposition of the reactive model is a part of implementation: the step does not help to specify *what* the system does but it specifies *how* the system operates. The control cycle step is partly derived from our previous work on incorporating FMEA into Event-B specifications [11]. The control cycle pattern described in this section can accommodate the outcomes of the FMEA analysis, and thus the results of [11] can be reused for this work.

This step finalises the modelling and provides placeholders in the model for further implementation.

5 Case Study

In this section, we demonstrate an application of the reactive-style steps of the proposed method to modelling the Attitude and Orbit Control System (AOCS). This case study was extensively used in the FP7 DEPLOY project [6]. The AOCS is a generic component of a satellite onboard software the main function of which is ensuring the desired attitude and the orbit of a satellite. In our case study, the AOCS is equipped with three hardware units: the Earth sensor (ES), the GPS sensor, and the payload instrument (PLI). The system functions by switching through a sequence of modes ($Off \rightarrow Nominal \rightarrow Science$) where each mode requires a certain configuration of units. The ultimate goal is to stay in the *Science* mode and collect data with the PLI unit. To tolerate hardware

failures, every unit of the system has a spare unit, and upon detecting an error the system must reconfigure itself to either use a spare unit if it is available or switch to a mode that does not use the failed units, i.e. degrade gracefully.

We follow the refinement strategy described in Section 4.2 and start with an abstract model (M0) of system failure-free functionality. We define two events for each of the system units: event *unit_work* corresponds to a particular unit operation, and *unit_switch* represents the switch of the unit between its functional states (e.g., on/off for the ES unit).

The AOCS is a safe stop system: it can safely stop its operation when it cannot provide the required service. Such situation can arise when the units required for the first functional mode ($Nominal$) are not available. At M1 we apply the *safe stop* step described in Section 4.3. We extend the Event-B model according to the *safe stop pattern*, and the modal view follows the *safe stop template* and contains two modes: *Normal* and *Stop*.

The next step (M2) contains functional refinement: we introduce mode switching functionality and restrict unit operation to certain modes. We represent the current system mode by a variable *mode* and define three new events that change its value: events *goAdvance* and *downgrade* initiate the reconfiguration process towards a "higher" and a "lower" mode correspondingly, and event $reconfFinish$ finalises reconfiguration. The modal view of this step is shown on Figure 5. We refine the $Normal$ mode from M1 into the three functional modes. We define the mode assumptions and guarantees using the variable *mode*, and formally prove that the model satisfies the view.

At M2 the units operation is unrestricted: units can work and switch regardless of the current system mode. At the step M3 we refine the reconfiguration process and restrict the units operation to relevant modes. We refine event $reconfFinish$ to only finalise the reconfiguration process when all units are in the states required by the target mode. Reconfiguration consists in switching the units to those states, and we restrict events *unit_switch* to only act during reconfiguration to appropriate modes (e.g., the PLI unit can only be switched on for the *Science* mode). We also restrict the *work* events to fire in relevant modes. We apply the *behavioural split template* to the modal view of M2 and obtain a modal view M3 shown on Figure 6. Each of the functional modes is split into two modes: the first mode represents functionality during reconfiguration (e.g. mode $toNominal$), the second mode is the stable mode when all required units are in operation(e.g. mode $Nominal$). We redefine the mode guarantees in terms of unit states. For example, the guarantee of mode $Nominal$ is now:

$$unitES' = UNIT\_ON \wedge unitGPS' = GPS\_COARSE \wedge unitPLI' = UNIT\_OFF$$

At the next step M4 we refine the fault tolerant behaviour of the system and restrict its functional behaviour (Snippet 1). This corresponds to the method steps described in Sections 4.4 and 4.5. The abstract fault tolerant behaviour is represented by the variable *stopped* from M1 which depicts the system error state (operational/stopped). We apply the *error state variable pattern* and define error states of units. For each unit, we define an integer variable representing the

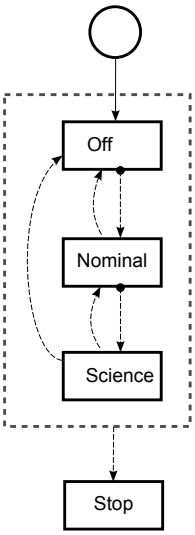

Fig. 5: Modal view M2

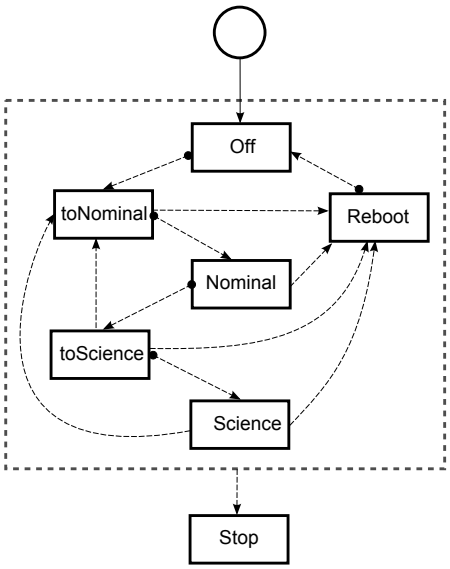

Fig. 6: Modal view M3

invariants

inv_ES_cond: $unitES\_cond \in \{0, 1, 2\}$

inv_glue: $stopped = TRUE \Leftrightarrow$
$unitES\_cond = 0 \vee$
$unitGPS\_cond = 0$

events

event ES_break $\widehat{=}$ **refines** *stop*

when

grd1: $unitES\_cond = 1$

grd2: $unitES = UNIT\_ON$

grd3: $unitGPS\_cond > 0$

then

act1: $unitES\_cond := 0$

act2: $stopped := TRUE$

end

event ES_downgrade $\widehat{=}$ **refines** *downgrade*

when

grd1: $mode > OFF$

grd2: $unitES = UNIT\_ON$

grd3: $unitES\_cond = 2$

grd4: $unitGPS\_cond > 0$

with

newMode: $newMode = OFF$

then

act1: $mode := OFF$

act2: $stable := FALSE$

act3: $unitES\_cond := 1$

end

event ES_work $\widehat{=}$ **extends** ES_work

when

grd3_0: $unitES\_cond > 0$

end

Snippet 1: Fault tolerant component refinement at M4

number of available units of that type (e.g. *unitES_cond* on the snippet). We apply the *error state invariant pattern* and define a relation *inv_glue* between abstract *stopped* state and unit availability variables.

At the abstract level, system reaction is represented by event *stop* that stops the system, and event *downgrade* that starts a reconfiguration to a "lower" mode. We define the error detection events as refinements of one of the appropriate reactions according to *fault tolerant behaviour pattern*. For example, event *ES_downgrade* may fire when an ES unit error is detected and there is a spare unit available. The system then downgrades and enables the spare unit. Event *ES_brake* represents a situation when the last ES unit fails and the system stops.

We apply the *behaviour restriction pattern* and strengthen the guards of functional events to satisfy the current error state. As shown on the snippet, the ES unit can only operate (this is represented by event *ES_work*) when there is at least one such unit available. The modal view of the system is changed accordingly: although there are no new modes or transitions on the view, the assumptions of the existing modes are redefined in terms of unit error states. For example, the assumption of mode *Nominal* is now as follows:

$$mode = NOMINAL \wedge stable = TRUE \\ \wedge unitES\_cond > 0 \wedge unitGPS\_cond > 0$$

This step finalises the development of the AOCS. It can be shown that the *control cycle pattern* is also applicable to the system, we omit the step due to space restriction. The full Rodin project containing the models and views can be downloaded from the Modal Views wiki page [2]. The refinement chain consists of 5 Event-B machines and 4 associated modal views; overall 381 proof obligations were proven, 359 of them automatically.

6 Conclusions

Development of correct fault tolerance is a major challenge in designing complex dependable systems as evidenced by major failures such as the crash of the Ariane 5 launcher and the August 2003 Blackout in the US and Canada. Analysis of these and more recent failures shows that a (typically substantial) support for tolerating faults in many modern systems often fails or has a lower quality than the rest of the systems. In this paper, we described a top-down development method for formal modelling of fault tolerant systems focusing on high levels of abstraction. The method allows developers to refine the abstract fault tolerant systems into control systems. The method facilitates modelling fault tolerance formally starting from the early stages of development. The early consideration of fault tolerance in refinement-based methods can reduce the modelling efforts, and helps to ensure the overall dependability of the resulting systems.

The method proposed incorporates a separate viewpoint for modelling modal and fault tolerance features of systems. This viewpoint adds rigour to the formal development process, contributes to readability of formal models by engineers,

and bridges the gap between requirements and formal models. The method ensures the reuse of formal modelling by supporting patterns typical for modelling fault tolerance.

The method is tool supported. The modal viewpoint is implemented as a plug-in for the Rodin environment which includes a diagram editor and a smooth integration with prover facilities [2].

Acknowledgements

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

References

1. The RODIN platform: http://rodin-b-sharp.sourceforge.net/.
2. Wiki page for Modal and Fault Tolerance Views language and tool support: http://wiki.event-b.org/index.php/mode/ft_views.
3. J.-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
4. A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.
5. F. L. Dotti, A. Iliasov, L. Ribeiro, and A. Romanovsky. Modal systems: Specification, refinement and realisation. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, pages 601–619. Springer-Verlag, 2009.
6. D. Ilic et al. *DEPLOY Deliverable D3.2: Report on Enhanced Deployment in the Space Sector*. August 2011.
7. L. Laibinis and E. Troubitsyna. Fault Tolerance in a Layered Architecture: A General Specification Pattern in B. In *Proceedings of the 2nd International Conference on Software Engineering and Formal Methods*, SEFM'04, pages 346–355. IEEE Computer Society, September 2004.
8. L. Laibinis and E. Troubitsyna. Refinement of Fault Tolerant Control Systems in B. In *Proceedings of the 23rd International Conference on Computer Safety, Reliability, and Security (SAFECOMP'04)*, pages 254–268, Potsdam, Germany, 2004.
9. P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., 1990.
10. I. Lopatkin, A. Iliasov, and A. Romanovsky. Rigorous Development of Dependable Systems using Fault Tolerance Views. In *Proceedings of the 22nd International Symposium on Software Reliability Engineering, ISSRE'11*, pages 180–189, Hiroshima, Japan, December 2011.
11. I. Lopatkin, A. Iliasov, A. Romanovsky, Y. Prokhorova, and E. Troubitsyna. Patterns for Representing FMEA in Formal Specification of Control Systems. In *The 13th IEEE International High Assurance Systems Engineering Symposium (HASE'11)*, pages 146–151, Boca Raton, FL, USA, November 2011.
12. J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):19:1–19:36, October 2009.