COMPUTING SCIENCE

Rigorous Development of Fault-Tolerant Systems through
Co-Refinement

Ilya Lopatkin and Alexander Romanovsky

TECHNICAL REPORT SERIES

No. CS-TR-1405

January 2014

Rigorous Development of Fault-Tolerant Systems through Co-Refinement

I. Lopatkin and A. Romanovsky

Abstract

With our increasing dependency on computer-based systems, ensuring their dependability becomes one of 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 is used to demonstrate the proposed method.

Bibliographical details

LOPATKIN, I., ROMANOVSKY, A.

Rigorous Development of Fault-Tolerant Systems through Co-Refinement
[By] I. Lopatkin and A. Romanovsky

Newcastle upon Tyne: Newcastle University: Computing Science, 2014.

(Newcastle University, Computing Science, Technical Report Series, No. CS-TR-1405)

Added entries

NEWCASTLE UNIVERSITY

Computing Science. Technical Report Series. CS-TR-1405

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 is used to demonstrate the proposed method.

About the authors

Ilya Lopatkin was awarded his PhD degree by Newcastle University in 2012. During his PhD study he contributed to the PF7 DEPLOY Integrated Project. In 2012-2014 he was an RA on the EPSRC/RSSB SafeCap and the EPSRC SafeCap Impact Projects.

Alexander (Sascha) Romanovsky is a Professor in the Centre for Software and Reliability, Newcastle University. 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 PhD degree in Computer Science from St. Petersburg State Technical University and has worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland and at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993 he became a postdoctoral fellow in Newcastle University, and worked on the ESPRIT projects on Predictable Dependable Computing Systems (PDCS), Design for Validation (DeVa) and on UK-funded projects on the Diversity, both in Safety Critical Software using Off-the-Shelf components. He was a member of the executive board of EU Dependable Systems of Systems (DSoS) Project, and between 2004 and 2012 headed projects on the development of a Rigorous Open Development Environment for Complex Systems (RODIN), and latterly was coordinator of the major FP7 Integrated Project on Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity (DEPLOY). He now leads work on fault tolerance in Systems of Systems within the COMPASS project and is Principal Investigator of Newcastle's Platform Grant on Trustworthy Ambient Systems.

Suggested keywords

FAULT TOLERANCE

ERROR RECOVERY

FORMAL METHODS

VERIFICATION

CORRECT BY CONSTRUCTION

Rigorous Development of Fault-Tolerant Systems through Co-Refinement

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 of 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 is used to demonstrate the proposed method.

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 and defects of which 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.

It is well-known that one cannot produce a faultless system functioning in a perfect fault-free environment [8]. 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 [8] in both hardware and software.

Furthermore, the design complexity of modern systems requires additional means for reducing the number and criticality of design faults. 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.

The development method proposed in this work builds on our study of the requirements descriptions and formal models produced by deployment partners of the FP7 DEPLOY project [2]. The study showed that up to 35-40% of requirements to critical systems can be devoted to fault tolerance. However, typically, formal models do not adequately represent the fault tolerant behaviour due to the fact that the fault tolerance component of requirements is intertwined with the functional one and is difficult to address during the modelling phase.

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 [6]. Other techniques provide patterns and modelling styles for modelling fault tolerance within the formal semantics of a particular formalism [7].

We follow a pattern-based approach and propose a method for modelling high-level fault tolerant system behaviour. In contrast to the above-mentioned studies, we focus on reactive style of system-level fault tolerant behaviour, and provide support for explicit reasoning about safety properties.

The paper is organised as follows. We give an overview of the proposed method in Section 2. Then we describe the basic modelling principles behind the method and the proposed refinement strategy in Sections 3 and 4 accordingly. We describe the method by applying it to a case study in Section 5, and draw conclusions in Section 6.

2 Overview

In this paper we demonstrate a method for top-down development of fault tolerant systems with a focus on abstract levels of modelling. The method focuses on verification of safety properties of fault tolerant systems and ensures traceability of the relevant safety and fault tolerance requirements.

The method accommodates two formalisms: a traditional state-based formalism and a diagrammatic formalism from our previous work [10]. Any state-based formalism with interleaving semantics can be used with the proposed method such as Action Systems, B, Event-B, Z, and VDM with refinement-based formalisms benefiting the most. In this paper we exemplify our approach on the Event-B formalism [3]. The second formalism used in the method constitutes an additional viewpoint called Fault Tolerance (FT) Views for modelling modal and fault tolerant behaviour [10]. The FT viewpoint is also refinement-based. The refinement chains for the two formalisms coexist in a single development and formally relate to each other. At each refinement level, the two formal models essentially represent the same system at the appropriate level of abstraction (Figure 1).

The development method includes the following three constituents:

Fig. 1: Refinement chain of a bi-model development

- the *modelling principles* stating the key rules and reasoning behind the modelling process of the method,
- 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.

3 Modelling Principles

The development method is based on a number of modelling principles. These postulate a set of terms and rules that are used in definitions of modelling patterns and the refinement strategy described later.

The first principle of the method defines the style for modelling fault tolerant systems. 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 the following 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. During its evolution, it can go from any state to any other state. In the proposed method, 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.

In our method, we assume that a system observes some part of its environment and reacts to its changes. To represent the environment adequately in the models, we require all state transitions occurring during system execution 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.

To structure the formal development in such a way that follows the implementable causality and the behaviour restriction principles, we offer a term *fault tolerant component*. A fault tolerant component is a structural system unit that is described by its *functional* and *error* state variables which are explicitly separate in the model. In this regard, the term is similar to the Idealised Fault Tolerant Component [8]. Note that an FT component is a modelling concept and does not necessarily represent a physical object or a design-level module.

4 Refinement Strategy

The development method prescribes a number of modelling steps that need to be performed to arrive at a correct and meaningful model of a fault tolerant system. The schematic procedure of the development method is shown on Figure 2. 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 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 5.2.

Steps 3, 4 and 5 form the core of the method. They are repeated iteratively until all the required properties of the reactive system behaviour are expressed and verified. Step 3 is a 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 step is described in Section 5.3. Step 5 is the *behaviour restriction* step that is used to restrict the functional behaviour with operational conditions dictated by the environment. The details of this step are given in Section 5.4.

The second part of the method refines the reactive model into a model of a control system. Here we reuse our previous work on modelling control cycles and incorporating FMEA into Event-B specifications [11] and omit description of this step due to space restriction. It is important to note that the first part

Fig. 2: The steps of the method

focuses on verification of safety properties whereas the second part of the method facilitates further implementation of the system.

5 Method Application

In this section we describe the steps of the proposed method applied to a running example. The example is an airlock system shown in Figure 3. We define requirements to the system as a set of informal statements that we then formalise using the proposed method.

5.1 Failure-Free Functionality

We start describing our system by defining its environment:

(ENV1) *The airlock system separates two different environments (external and internal). The pressure of the external environment is lower than that of the internal one. The internal environment is considered to be natural to humans.*

(ENV2) *The system has two doors and a chamber (Figure 3). Each door when closed separates the chamber from the appropriate environment.*

The primary function of the system can be expressed in the following form:

(FUN1) *When in operation, the airlock system must be able to let users pass safely between the two environments via the airlock.*

In order to allow a user to pass from the internal area through the airlock into the external area, the system needs to perform the following steps:

Fig. 3: The airlock system

1. equalise the chamber pressure to that of the internal environment,
2. open the second door to allow the user in the chamber,
3. close the second door,
4. equalize the pressure in the airlock to that of the external environment,
5. open the first door to allow the user out,

and vice versa for the opposite direction.

To provide such operations, the system is equipped with a number of sensors and actuators as shown on Figure 3. We assume that the details of such a low level are given in requirements on which we base our abstractions. In this paper, we omit the low-level modelling steps due to lack of space and focus on modelling and proving safety properties at a high level of abstraction. We define the following safety requirements for the airlock system:

(SAF1) *The pressure in the chamber must always be between the lower external pressure and the higher internal one*

(SAF2) *A door can only be opened if the pressure values in the chamber and the conjoined environment are equal*

(SAF3) *At most one door is allowed to be opened at any moment of time*

(SAF4) *The pressure in the chamber shall not be changed unless both doors are closed*

At this stage, we only define the *failure-free functionality* of the system. That is, no failures are considered and the system is assumed to work in a flawless manner forever. We formalise the given requirements in an Event-B model M0 (see Snippet 1). In the model, section **axioms** contains axiomatic definitions such as a given set of door states (*opened*, *closed*, etc). We represent the two environments described in (ENV1) by their pressure values *LOW_PRESSURE* and *HIGH_PRESSURE* that we assume to be constant. Section **invariants** contains variable type definitions and safety properties that must always hold during system evolution. For example, we represent the physical components

axioms

axm1: $\text{partition}(\text{DOOR_STATE}, \{\text{OPENED}\}, \{\text{CLOSED}\}, \{\text{OPENING}\},$
 $\{\text{CLOSING}\}, \{\text{STOPPED}\})$
 axm2: $\text{LOW_PRESSURE} = 0$
 axm3: $\text{HIGH_PRESSURE} = 2$

invariants

inv1: $\text{door1} \in \text{DOOR_STATE}$
 inv2: $\text{door2} \in \text{DOOR_STATE}$
 inv3: $\text{pressure} \in \mathbb{N}$
 inv4: $\text{door1} \neq \text{CLOSED} \Rightarrow \text{pressure} = \text{LOW_PRESSURE}$
 inv5: $\text{door2} \neq \text{CLOSED} \Rightarrow \text{pressure} = \text{HIGH_PRESSURE}$
 inv6: $\text{door1} = \text{CLOSED} \vee \text{door2} = \text{CLOSED}$
 inv7: $\text{pressure} > \text{LOW_PRESSURE} \Rightarrow \text{door1} = \text{CLOSED}$
 inv8: $\text{pressure} < \text{HIGH_PRESSURE} \Rightarrow \text{door2} = \text{CLOSED}$
 inv9: $\text{pressure} \geq \text{LOW_PRESSURE} \wedge \text{pressure} \leq \text{HIGH_PRESSURE}$

events

event **open1** $\triangleq$

when

grd1: $\text{door1} = \text{CLOSED} \vee \text{door1} = \text{STOPPED}$
 grd2: $\text{pressure} = \text{LOW_PRESSURE}$
 grd3: $\text{door2} = \text{CLOSED}$

then

act1: $\text{door1} := \text{OPENING}$

end

Snippet 1: Definitions, invariants, and behaviour of M0

from (ENV2) by variables *door1* and *door2* for the two doors correspondingly, and the current value of pressure in the chamber by variable *pressure*. The rest of the invariants correspond to the safety requirements. Namely, *inv9* ensures the pressure limits required by (SAF1). Invariants *inv4* and *inv5* correspond to requirement (SAF2). Requirement (SAF3) is ensured by *inv6*. Invariants *inv7* and *inv8* together represent (SAF4).

The **events** section formalises the behaviour of the system. It consists of a set of events each of which represents a guarded labelled transition. The guard of each event is given as a set of predicates in section **when**. When the guard becomes true, the event fires thus atomically making a transition described in section **then** by a set of assignments. The system evolves by making transitions in an interleaving fashion. During evolution, invariants must always hold, and the modeller is obliged to prove that the action of each event preserves all of the invariants. In model M0, there are five events for each door that ensure the safe traversal of the corresponding door through its set of possible states, and two events for changing the level of pressure in the chamber. For brevity, we only show an event of a door behaviour (*open1*) in Snippet 1. Event *open1*

starts opening the first door if it is either closed or stopped at some intermediate position.

5.2 Abstract Class of System Fault Tolerance

The next step is to identify the class of the system from the fault tolerance perspective. In the proposed method, 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.

At this stage, we include an additional type of formal models into the development process, Modal Views, and define a formal relationship with Event-B models. A modal view consists of *modes* and *mode transitions*. A mode describes the functionality of a system by its *guarantee* predicate, and the operating conditions under which the system provides this functionality by its *assumption* predicate. A mode is mapped into a set of Event-B events and represents a superstate of a system. A mode transition represents an instantaneous switch of the system between two modes. More details on formal definitions of Modal Views and relationship with Event-B can be found in [10,5,1,9]. Here it is important to understand that such a relationship between two different types of models produces formal consistency conditions in a form of additional proof obligations. These require developers to focus more on the modelling activity and, thus, add rigour to the development process.

We associate the two classes of system fault tolerance with two possible initial modal views accordingly (Figure 4). Systems of the first class can mask all internal errors and operate indefinitely long. This is represented by a single mode *Normal*. Systems of the second class cannot tolerate certain errors and can eventually stop. The errors that can cause a system stop are called *unrecoverable* and are collectively represented by the abstract transition to mode *Stop*.

Fig. 4: Two abstract classes of fault tolerant systems: a) Failure-free, b) Safe-stop

The airlock system so far contained only failure-free functionality reflecting the absence of failures in requirements. Now we assume that some components of the system can fail, and the whole system may eventually stop due to such errors.

We accommodate this behaviour in model M1 (Snippet 2). We define variable *stopped* representing the operational availability of the system, and separate the functional behaviour from the stopped state by using this variable. We refine all functional events by strengthening their guards and add two new events as shown in Snippet 2. The changes we made to the model are generic in that they are applicable to any safe stop system. These constitute the *safe stop pattern* that should be applied as the first refinement step for modelling fault tolerant behaviour in case of a safe stop system.

```

event open1  $\hat{=}$  extends open1
  when grd_stopped: stopped = FALSE

event stop  $\hat{=}$ 
  when grd_stopped: stopped = FALSE
  then act_stopped: stopped := TRUE

event stopped  $\hat{=}$ 
  when grd: stopped = TRUE
  then skip

```

Snippet 2: Part of M1 after applying the safe stop pattern

We also associate the modal view *b* on Figure 4 with Event-B model M1. This produces additional formal obligations to prove for M1 and ensures that the model indeed contains the safe stop behaviour.

The purpose of this refinement step is to "reserve" an abstract representation of the overall system fault tolerant behaviour for further refinements. Event *stop* represents an abstraction of all unrecoverable errors that will be introduced later.

5.3 Fault Tolerant Component Refinement

On top of the functional requirements to the system, we also introduce a "fragile" environment where the physical components of the system may fail:

(ENV3) *Sensors and actuators may fail to provide a correct function.*

From this fault assumption, description of sensors (which we omit due to lack of space), and description of available redundancy in the system we can construct an adequate abstraction for modelling. We define the three possible error states of the two doors in model M2:

$$door1_cond, door2_cond : \{BROKEN, DEGRADED, OK\}$$

Such a definition of error states constitutes the *error state variable pattern*.

Fault assumption (ENV3) raises a number of requirements that concern system fault tolerance:

(FT1) *The system shall disallow opening a degraded door.*

(FT2) *The system shall stop if at least one of the doors is broken.*

(FT3) *If both doors are degraded, the system shall stop unless there is a user in the chamber. If the user is present in the chamber, the system shall allow opening the inner door.*

Fig. 5: Modal view of the airlock M2 model

In order to represent these requirements formally, we refine the modal view shown in Figure 4b by splitting the normal behaviour of the system into four modes: the normal operation mode and three degraded modes (Figure 5). The system stays in mode *Door1* when the first door is degraded and the second door is fully operational, and vice versa for mode *Door2*. When both doors are degraded and there is a user present in the chamber, the system stays in mode *Trapped* until the user leaves the chamber. The new mode *Normal* together with the three degraded modes formally refine abstract mode *Normal* as shown by a dashed area. Such a refinement of a mode by a chain of degraded modes constitutes the *mode split template*. The degraded modes in Figure 5 represent different sets of available components and the associated subsets of system behaviour. The assumption predicates of the modes split the possible combinations of the components' error states into disjoint sets. The mode assumptions cover all the system states which must be demonstrated through the well-definedness proof obligation COVER [1].

We refine the abstract system failure transition by four concrete transitions depicting the sources of failure. Transitions *Stop on degrade* and *User leaves* initiate at the new modes. Transition *Break* can initiate at any of the four modes within the dashed area. All three formally refine the abstract error transition, and this is an example of an application of the *transition split template*. The two

templates shown here (mode split and transition split templates) provide a means for step-wise construction of complex modal behaviour and ensure traceability of modal views into fault tolerance requirements.

Modal view M2 represents fault tolerant behaviour of the system in terms of its *fault tolerant components*, two doors and the chamber. In order to ensure formal refinement between M1 and M2, we define a relationship between the abstract error state variable *stopped* and the newly defined door error state variables as a gluing invariant *inv4*:

$$\begin{aligned} & \text{door1_cond} = \text{BROKEN} \vee \text{door2_cond} = \text{BROKEN} \vee \\ & (\text{door1_cond} = \text{DEGRADED} \wedge \text{door2_cond} = \text{DEGRADED} \wedge \\ & \text{obj_presence} = \text{FALSE}) \Leftrightarrow \text{stopped} = \text{TRUE} \end{aligned}$$

Definition of gluing invariants over error state variables constitutes the *error state invariant pattern*. This pattern is necessary for establishing the refinement relation between each two subsequent modelling steps. Note that the gluing invariant also refers to functional variable *obj_presence* that we introduced to meet requirement (FT3). This variable shows whether a user is present in the chamber. Such reference to a functional variable highlights the point that the fault tolerance properties of the system are inevitably tied to the functional state and both need to be taken into account during refinement.

```

event break  $\hat{=}$  extends stop
event degrade  $\hat{=}$ 
event stop_on_degrade  $\hat{=}$  extends stop
  when
    grd1: door1_cond = DEGRADED  $\vee$  door2_cond = DEGRADED
    grd2: obj_presence = FALSE
    grd4: door1_cond = OK  $\vee$  door2_cond = OK
  then
    act1: door1_cond := DEGRADED
    act2: door2_cond := DEGRADED

```

Snippet 3: The airlock M2 model after refining fault tolerant components

The defined error states are used in definition of fault tolerant behaviour using the *fault tolerant behaviour pattern*: we refine the events that represent the reactions of the system to errors by error detection events. In this way we associate the causes of failures with system reactions. Snippet 3 shows three events *break*, *degrade* and *stop_on_degrade* changing the door error states in three different situations. Events *break* and *stop_on_degrade* extend abstract event *stop* with actions putting the doors into degraded and broken states to satisfy gluing invariant *inv4*. This shows how an abstract fault tolerant reaction is refined into more specific component failures. Event *degrade* is new at M2, it

changes the doors' error states and continues the system operation. It represents the tolerance of the system to certain errors. The three events represent a part of requirements (FT1), (FT2), (FT3). Two events *degrade* and *stop_on_degrade* depict the same abstract detection of a door failure but they lead to different reactions, and the choice depends on the current system state. We have to have both events in the model to cover all relevant system states at the moment of component failure to satisfy the implementable causality rule.

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.

5.4 Behaviour Restriction

During the previous steps, we refined the error states of the airlock system into error states of its components and defined events that provide transitions between those states. However, the functional behaviour of the system does not take the current error states into account. For example, a broken door can still “operate” in our model M2 as if it was in a normal condition. In this step, we remedy such an omission and restrict the functional behaviour to satisfy the fault tolerance requirements.

Firstly, we refine the M2 modal view by applying the behavioural split template to modes *Door1*, *Door2* and *Trapped* (Figure 6). For example, mode *Door2_closing* now restricts the system to only operate with the second door, and only contains events that close or stop the door but do not open it. Upon the door closure, the system switches to mode *Door2* that guarantees that the pressure is set to low and the door is closed, and thus only allows operating the first door.

Secondly, in order to satisfy the consistency conditions enforced by the modal view, we restrict the system behaviour expressed in the Event-B model. Specifically, to satisfy proof obligation *EVT_A* [1], we strengthen the guard of event *open1* by the following condition:

$$door1_cond = OK$$

We strengthen every event in model M3 in a similar manner thus ensuring that (FT1) is satisfied. Such a restriction constitutes the *behaviour restriction pattern*: we restrict the functional transitions that are not allowed with respect to the error states of components. In this regard, we treat modal views as diagrammatic specifications of the system fault tolerant behaviour, and their assumption/guarantee pairs as one of the two formalisations of fault tolerance requirements.

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 and fault tolerance properties.

Fig. 6: Modal view of airlock M3 model

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 starting from the early stages of abstract modelling and following to modelling control systems. 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. We demonstrated the application of the method by modelling an airlock system. The refinement chain of the airlock case study consists of 5 Event-B machines and 3 associated modal views; overall the development produced 417 proof obligations, 356 of which were proven automatically. Most of the rest 61 proof obligations that required interactive proof were generated by the Rodin tools. The full Rodin project containing the models and views can be downloaded from the Modal Views wiki page [1].

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 [1].

Acknowledgements

This work is supported by the ICT DEPLOY IP and the EPSRC/UK TrAmS-2 platform grant. We are grateful to Alexei Iliasov for fruitful discussions and support.

References

1. Wiki page for Modal and Fault Tolerance Views language and tool support. http://wiki.event-b.org/index.php/Mode/FT_Views.
2. FP7 DEPLOY Project: Industrial deployment of system engineering methods providing high dependability and productivity. <http://www.deploy-project.eu/>, 2008 - 2012.
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. R. Jeffords, C. Heitmeyer, M. Archer, and E. Leonard. A Formal Method for Developing Provably Correct Fault-Tolerant Systems using Partial Refinement and Composition. *Lecture Notes in Computer Science*, 5850:173–+, 2009.
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. P. A. Lee and T. Anderson. *Fault Tolerance: Principles and Practice*. Springer-Verlag New York, Inc., 1990.
9. I. Lopatkin. *A Method for Rigorous Development of Fault-Tolerant Systems*. PhD thesis, School of Computing Science, Newcastle University, 2013.
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.