On Fault Tolerance Reuse during Refinement

Ilya Lopatkin, Alexei Iliasov, Alexander Romanovsky

{ilya.lopatkin, alexei.iliasov, alexander.romanovsky}@ncl.ac.uk

Newcastle University, CSR, Newcastle upon Tyne, UK, NE1 7RU

ABSTRACT

Complex modern applications have to be developed to be dependable to meet their requirements and expectations of their users. An important part of this is their ability to deal with various threats (such as faults in the system environment, operator's mistakes, underlying hardware and software support problems). Development of modern applications is complicated by the need for systematic and rigorous integration of fault tolerance measures. The paper focuses on reuse of fault tolerance modelling. First, it introduces the idea of general modelling templates reflecting abstract views on system behaviour with respect to faults. These templates are used during system detalisation (refinement) to capture the user's view on system external behaviour. Secondly, it proposes to use a library of concrete modelling patterns allowing developers to systematically integrate specific fault tolerance mechanisms (e.g. recovery blocks, checkpoints, exception handling) into the models. The proposed solutions are linked to the Event-B method and demonstrated using a case study.

Categories and Subject Descriptors

I.6.5 [**Computing methodologies**]: Model Development—*modeling methodologies*; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, refinement, reliability*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Formal methods, Reuse, Reliability, Modelling

Keywords

fault tolerance, reuse, patterns, formal methods, refinement

SERENE '10 13-16 April 2010, London, United Kingdom

1. INTRODUCTION

Our society is becoming increasingly dependent on computer-based systems. There is a class of such systems, called *critical*, that provide services for fulfilling essential aspects of our life. Insufficient reliability of a critical system may result in significant losses of time, money, resources, or even lives. Critical systems have to be dependable [3], so that they can be justifiably trusted to provide the required services. Using formal methods is one of the solutions for ensuring system dependability by fault prevention and/or fault removal. Even though formal methods are not always used in developing industrial systems, their use in development of dependable systems is increasing and proven to be cost-effective [15]. If such system is formally developed, its key properties are ensured during the formal stage. However, it is well-known that we cannot produce a perfect system without a single fault, which functions in the perfect fault-free environment, this is due to many reasons including changing environmental conditions, hardware failures, and inevitable mistakes during the development process. In order to keep reliability at a sufficient level, we need to mitigate faults during system execution time. This is exactly the target of *fault tolerance* (FT): providing a required service in presence of faults.

There is a considerable amount of FT-related requirements within any critical system project[1]. Conditions causing erroneous transitions, error detection and recovery mechanisms, data and time redundancy requirements are all to be addressed during development of such systems. We believe that fault tolerance must be formally and explicitly developed starting from the earlier engineering steps with the purpose of improving requirements traceability, development discipline and to allow developers to evaluate the fault tolerance decisions earlier in the development.

In this paper, we propose solutions for systematic integration of fault tolerance during the refinement-based formal stage of software development in the Event-B method. Refinement provides a consistent way for stepwise development during which we have a correct model of a system at each step. The prevailing practice in system modelling is in focusing modeller's attention on the functional part of system behaviour, i.e. the normal one, with an (unsubstantiated) assumption that no errors can arise. This often results in difficulty with (or non-optimal) addition of the fault-tolerant behaviour at the later steps. With the refinement process and typically not accurate and evolving requirements, this

[1]See, for example, our ongoing work within the ICT Deploy project - http://www.deploy-project.eu/

problem is exacerbated by the inherent principles of formal system detalisation: in particular, by the inability to formally extend the behaviour of the abstract models with the new (e.g. abnormal) behaviour that was not anticipated. There are several solutions, all of which modify the way the formal refinement is defined. One of such solutions is a formal definition of partial refinement [11]. However, in our work we would like to be practical: we use the existing methods widely used in practice and minimize changes of the existing tool suites. More generally, we would like to encourage early FT development by providing corresponding top-level abstractions.

It is widely accepted that it is beneficial to support multiple views on the model, so that each of the views can focus on a particular concern of the model/system. This facilitates the development by explicitly bounding the modeller into a specific context without cluttering the model (some examples of this are multiple views provided by UML, or different views on the model supported in AADL). We intend to exploit this idea by developing an FT modelling approach, orthogonal to the existing method and supported by an appropriate extension of the existing tool chain.

Our proposal consists of several parts. We describe a modelling approach to assisting development of the fault tolerance part of the models. We provide a set of *abstractions* for system modelling from the FT point of view, which can be further refined using basic *templates*. We are developing a modelling environment for the FT view operating with such abstractions and templates and orthogonal to the existing Event-B model view. A formal link with Event-B is based on our previous work on modal system modelling [6]. We further investigate possible solutions for systematic Event-B model transformations to support specific FT templates with automatic refinement.

The rest of the paper is organised as follows. Section 2 briefly introduces the Event-B method and the Rodin development environment. Section 3 demonstrates the abstract templates, defining general classes of the systems with respect to their fault tolerance behaviour. Section 4 shows how a fault tolerance view can be supported by Event-B model transformation patterns. A case study is used in Section 5 to illustrate our approach. Section 6 outlines existing work in the area.

2. EVENT-B

Event-B is a state-based formalism closely related to Classical B [1] and Action Systems [4]. The step-wise refinement approach is the corner stone of the Event-B development method. A combination of model elaboration, atomicity refinement and data refinement helps to formally transition from high-level architectural models to very detailed, executable specifications ready for code generation. An extensive tool support makes Event-B especially attractive. An integrated Eclipse-based development environment[2] is under active development now and is well-supported. Importantly for us, it is open for extension using the Eclipse plug-in mechanism. The main verification technique is theorem proving and the development is supported by a collection of powerful theorem provers while there is also a capable model checker.

[2]The Rodin platform
http://www.event-b.org/platform.html

An Event-B model is defined by a tuple (c, s, P, v, I, R_I, E) where c and s are constants and sets known in the model; v is a vector of model variables; $P(c, s)$ is a collection of axioms constraining c and s. I is a model invariant limiting the possible states of v: $I(c, s, v)$. The combination of P and I should characterise a non-empty collection of suitable constants, sets and model states: $\exists c, s, v \cdot P(c, s) \wedge I(c, s, v)$. The purpose of an invariant is to express model safety properties (that is, unsafe states may not be reached). In Event-B an invariant is also used to deduce model variable types. R_I is an initialisation action computing initial values for the model variables; it is typically given in the form of a predicate constraining next values of model variables without, however, referring to previous values - $R_I(c, s, v')$. Finally, E is a set of model *events*.

An event is a guarded command: $H(c, s, v) \rightarrow S(c, s, v, v')$, where $H(c, s, v)$ is an event guard and $S(c, s, v, v')$ is a before-after predicate. The general form of an event in Event-B notation is

$$
\begin{array}{lcl}
\mathit{name} & = & \texttt{any } p \texttt{ where} \\
 & & \quad H(c, s, p, v) \\
 & & \texttt{then} \\
 & & \quad S(c, s, p, v, v') \\
 & & \texttt{end}
\end{array}
$$

where p is a vector of event parameters.

An event may fire as soon as the condition of its guard is satisfied and no other event executes at the same time. In case there is more than one enabled event at a certain state, the demonic choice semantics is applied. The result of an event execution is some new model state v'. The semantics of an Event-B model is usually given in the form of proof semantics, based on Dijkstra's work on weakest precondition. A collection of proof obligations is generated from the definition of the model and these must be discharged in order to demonstrate that the model is correct. For an abstract model (a model that is not a refinement of another model) two such proof obligations are the invariant satisfaction and event feasibility. A new state produced by an event must satisfy the model invariant:

$$I(c, s, v) \wedge P(c, s) \wedge H(c, s, v) \wedge S(c, s, v, v') \Rightarrow I(c, s, v')$$

An event must also be feasible, in a sense that an appropriate new state v' must exist for some given current state v:

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

There are also proof obligations to establish deadlock freeness, enabledness conditions and a collection of proof obligations for demonstrating Event-B forward simulation refinement [14].

3. FAULT TOLERANCE DETALISATION TEMPLATES

An FT view is a document developed alongside an Event-B model. It describes the design of the fault tolerance features in a compact and concise manner. It also offers simple detalisation rules that assist a user in constructing models with a corresponding fault tolerance part. There is a set of rules for formally checking the consistency of an FT view and its Event-B model.

The FT view is treated as a special case of a mode diagram, developed in our previous work on modelling modal systems; so that the consistency conditions for fault tolerance are borrowed from the mode conditions [6].

In his/her modelling activity the user is assisted with two/three simple refinement templates.

3.1 Overview

The two basic concepts of the FT view are *activity* and *error*. An activity is an abstract description of a system behaviour. An error, followed by an error detection always leads to switching from one activity to another. An error originates in a *normal activity* and leads to switching to a *degraded activity* or a *recovery activity*. Note that activity attribution to the specific type is relative and depends on the scope of discussion: what is a degraded mode activity in respect to one activity may be a normal activity in respect to another. It is important that a degraded activity is also a normal activity with respect to any error originating from it. In Fig. 1, activity A is a normal activity; there is an error $e1$ leading to alternative activity B from which the system could arrive at activity C upon an occurrence of another error $e2$.

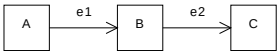

Figure 1: FT view basic concepts

There are certain restrictions to the ways FT diagram may be drawn. For instance, it must not contain cycles formed entirely from error transitions.

The building blocks of a diagram are primitives describing the initiation of a degraded activity and a transition into a recovery activity (Fig. 2). The principle distinction between the two is that recovery activity is obliged to terminate and pass control back to the activity from which the initiating error originated.

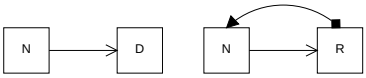

Figure 2: Degraded and recovery activities

Safe-stop is regarded as a special case of a degraded activity.

3.2 General Classes of Fault Tolerant Systems

Diagrams are built in a step-wise manner, starting from the most primitive case and introducing details with a number of predefined templates. The reason for a template-based approach is to match the step-wise development method of Event-B and avoid verification of FT diagrams since diagrams built in a step-wise manner are always well-formed.

There are just two possible initial diagrams, defining two broad system classes. The first class does not have an unrecoverable error: all errors are recoverable and, at a sufficiently abstract level, there are no errors at all. In the other case, there are errors that cannot be masked and system necessarily transitions into a differing activity after an error occurrence. What is considered to be an error is a design choice: the same functionality may be implemented by either system class. Fig. 3 illustrates the two possible initial diagrams.

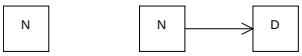

Figure 3: Abstract classes of FT systems

In the first diagram, the most abstract system is a normal activity. Further detalisation of the diagram may introduce only maskable errors. In the second diagram, in addition to the normal activity there is an error leading to a degraded mode activity. Both normal and degraded activities may be explained in further details by introducing new maskable errors. The error originally present in the initial diagram may also be explained in the terms of a number of new errors.

To assist in the construction of FT diagrams, two detalisation templates are offered. The first one is concerned with detalisation of an error occurrence in a diagram. The idea here is to replace an abstract depiction of an error with two or more concrete errors. This process may be repeated as many times as needed and the result is a whole family of errors derived from a single abstract error.

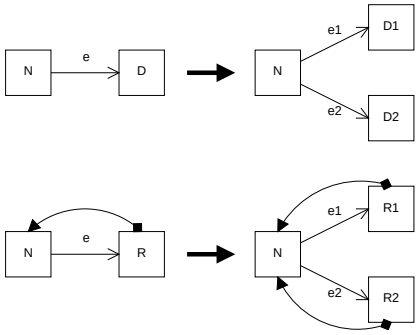

Figure 4: First template

As shown in the diagram on Fig. 4, there are two versions of this template. One for the case when an error leads to a degraded activity and another when there is also a subsequent recovery. This distinction is due to the fact that an obligation of successful recovery must be preserved during detalisation.

The second template introduces a new error that was not observed at a previous abstraction layer (Fig. 5). This error, of course, must be masked and thus an error may only switch into a recovery activity.

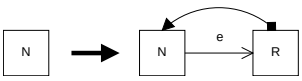

Figure 5: Second template

A diagram may be considered final once for each error condition in a system requirements document, there is a corresponding error arrow in the diagram.

3.3 Event-B Link

Our intention is to offer a modelling assistant environment to Event-B developers. For the FT diagrams approach to be truly useful, there need to be a formal relationship between a diagram and an Event-B model establishing that a model agrees with a diagram. Thus, an FT diagram alone would be enough to grasp the design of fault-tolerance in a model.

The formalisation approach is based on a more general notion of formal modal systems [10]. There is a study on linking mode diagrams and Event-B [6] and some of the results are reused here.

Activity is a general characterisation of a system behaviour. To match this notion in terms of Event-B models, activities are mapped into non-overlapping event groups. Likewise, an error is mapped into a single Event-B event.

For a stronger notion of a diagram - model relationship, we consider an FT view as a set of activities providing different functionality under differing operating conditions. We use the terms *assumption* to denote the different operating conditions and *guarantee* to denote the functionality ensured by the system under the corresponding assumption. With assumption and guarantee of an activity being predicates expressed on the same variables as an Event-B machine, we are able to impose restrictions on the way activities and errors are mapped into model events and thus cross-check design decisions in either part.

Formally, an activity is characterised by pair A/G where:

- $A(v)$ is an assumption - a predicate over the current system state;
- $G(v, v')$ is the guarantee, a relation over the current and next states of the system; and
- vector v is the set of model variables.

It is required to show that the assumptions exhaust the invariant and thus cover all the safe system states:

$$I(v) \Rightarrow A_1 \vee A_2 \vee \cdots \vee A_n \tag{1}$$

Here $I(v)$ is a model invariant characterising valid states of v. One other important property of an activity is that it is possible for some state transition to take place within an activity. We do not care here to give a precise definition of such transition, - this information would be later filled in by an Event-B machine - it is only necessary to show that there exists at least one such transition and thus activity characterisation makes sense:

$$\exists v, v' \cdot I(v) \wedge A(v) \Rightarrow G(v, v') \tag{2}$$

Thus, G can never be *false* everywhere while, under certain circumstances, this would be allowed for A. Note that from above it follows that an activity assumption is satisfiable: $\exists v \cdot A(v)$. It is required that no two activities may be running at the same time. This translates into demonstrating that each activity has an assumption that does not overlap with an assumption of any other activity:

$$I(v) \Leftarrow A_1(v) \oplus \cdots \oplus A_n(v) \tag{3}$$

In addition to activities, a diagram also includes two kind of activity connectors: errors and recovery. Their purpose is to define possible activity changes. We formalise the notion of activity connectors by modelling possible transitions between activities.

A system switches from one activity into another through an activity transition that non-deterministically updates the state of v in such a way that the assumption of the source activity becomes false while assumption of the target activity becomes true. Let us consider two activities, i and j. An activity transition is required to establish a new state v' such that $\neg A_i(v')$ and $A_j(v')$ while it is not under the obligation to respect $G_i(v, v')$.

It is required that all the activities are reachable. Although we could give a formal test for this property, there is no need to check it as long as a diagram is constructed using the predefined development templates discussed above.

3.3.1 Detalisation Conditions

FT diagrams are built by incrementally adding new activities, errors and recoveries using the two development templates. There are certain constrains to applying development templates. First, we should not allow a modeller to arbitrarily change the activity assumption and guarantee during detalisation. Second, in some cases of adding a new activity we effectively 'split' an abstract one and thus there is a relationship between the attributes of the abstract and concrete activities.

The first condition states that during detalisation the assumption of an activity may be weakened while its guarantee may be strengthened. For a discussion why it is like this see [6].

$$\begin{array}{l} A(v)/G(v, v') \sqsubseteq A'(u)/G'(u, u') \\ \quad \text{iff } \begin{cases} J(v, u) \wedge A(v) \Rightarrow A'(u) \\ J(v, u) \wedge G'(u, u') \Rightarrow G(v, v') \end{cases} \end{array} \tag{4}$$

Here $J(v, u)$ is a gluing invariant relating a concrete state u to an abstract state v. The assumption and guarantee of a refined activity are stated on the concrete state.

In a more general case, an activity is refined into two or more concrete activities:

$$\begin{array}{l} A(v)/G(v, v') \sqsubseteq \begin{array}{l} A_1(u)/G_1(u, u') \\ A_2(u)/G_2(u, u') \end{array}, \\ \quad \text{iff } \begin{cases} J(v, u) \wedge A(v) \Rightarrow A_1(u) \vee A_2(u) \\ J(v, u) \wedge G_1(u, u') \vee G_2(u, u') \Rightarrow G(v, v') \end{cases} \end{array} \tag{5}$$

With this rule new activities appear by splitting a fictitious activity *false/true*. This activity can be refined into a copy of itself and a new activity $A(w)/G(w, w')$.

After adding a new activity one has to connect it to the rest of a diagram by placing a new activity transition: an error or a recovery. A simple rule applies: removing new activities and new activity transitions from a detalised diagram results in the original diagram, apart from possible change in assumption and guarantee predicates. A more general modal diagram defines a number of additional constraints on transitions that are not needed here due to very restricted manner of a diagram evolution.

3.3.2 Reconciling Event-B and FT View Diagram

With the basic formal framework of activities in place, it is possible to define a consistency condition for an FT view diagram and Event-B machine. The core principle is seeing the diagram as a source of further obligations for a machine. We are not going to translate activities into Event-B or Event-B into activities. Instead, we are going to add additional proof obligations to a machine that establish the consistency with a given FT diagram. Of course, it does not matter where the proof obligations are added - we could flip this around and prove that a machine is consistent with a diagram by adding some theorems to diagrams. It is, however, more natural to deal with additional constraints in a machine and the intuition is that a simpler diagram should lead the development of a machine. Although we do not discuss this case, nothing prevents one from proving that the same machine is consistent with more than one diagram.

The first step is to relate activities to machine elements. A diagram is linked with an Event-B model by attributing a list of Event-B model events to each activity:

$$\begin{array}{l} A_1/G_1 \mapsto E_1 \\ A_2/G_2 \mapsto E_2 \\ \quad \cdots \\ A_n/G_n \mapsto E_n \end{array}$$

Event sets $E_1, \ldots, E_n$ may overlap but should not be identical. The latter is due to the fact that two activities $A_i/G_i \mapsto E$ and $A_j/G_j \mapsto E$ are equivalent to a single activity $A_i \vee A_i/G_i \wedge G_j \mapsto E$ and thus there is no advantage in allowing configurations where activities have identical event sets.

The result of mapping activities into event sets is that there are now additional requirements to machine events: an event related to some activity must respect the activity guarantee provided the activity assumption holds. Execution cannot progress if there is no suitable enabled event for an activity. From the above the following conditions are derived.

All the events of an activity must satisfy the activity guarantee provided the assumption holds:

$$I(v) \wedge A(v) \wedge H(v) \wedge R(v, v') \Rightarrow G(v, v') \tag{6}$$

The partitioning of the events into activities must be in agreement with the event guards. When an event is enabled then the assumption of its activity must hold. Since an event is potentially associated with multiple activities, the disjunction of all the relevant assumptions must hold:

$$\begin{array}{l} H(v) \Rightarrow A_1(v) \vee \cdots \vee A_k(v) \\ A_{k+1}(v) \vee \cdots \vee A_n(v) \Rightarrow \neg H(v) \end{array} \tag{7}$$

where $A_1, \ldots, A_k$ are the assumptions of the activities containing an event with guard $H(v)$ and $A_{k+1}, \ldots, A_n$ are those not containing the event.

4. FACILITATING FT VIEW WITH EVENT-B SUPPORT

The FT view and detalisation templates are orthogonal to Event-B model development. For more FT modelling support, diagrams can be coupled with a model transformation mechanism. Here, we give general ideas and a view on such mechanism which is an ongoing work and is described informally based on our intuition and a number of related works.

The essential entity of the mechanism as we see is a single model transformation - a *pattern*. A pattern takes an input model and produces an output model which should be a correct refinement. Such pattern introduces a comprehensible detalisation into the model focusing on a particular aspect. The refinement relation is to be formally established in either an offline proof of transformation correctness, or left to user as a number of proof obligations. We can clearly see three main aspects to be addressed for a well-designed pattern approach:

- applicability conditions that input model must satisfy in order to be transformed;
- transformation itself; and
- formal proof of transformation correctness

By providing model transformations, it is possible to reduce the total number of proofs done by user and generally simplify the whole development process. The basic idea for gluing patterns with the FT view is to create a simple transformation pattern for each of FT detalisation templates. Thus, when refining an FT view, a user automatically gets a refined Event-B model which is consistent with the new FT view. A reasonable part of necessary proofs may be carried out without involving user which is a subject for further investigation.

The FT view is a structuring mechanism which encourages a disciplined approach to modelling fault tolerance. Transformation patterns operate on a meta-model level and therefore are expressive in describing particular changes into the model. We believe it is feasible to create transformations which add specific FT mechanisms. They may be domain-specific or suitable for a wider application. Transformation mechanism together with detalisation templates can allow to create libraries of patterns which can be extensively reused within formal modelling community.

4.1 Library of Fault Tolerance Patterns

Any non-trivial development in the Event-B method is a chain of refinements. Each refinement usually targets a single aspect of a model that needs to be detailed and keeps the rest fixed. It is obvious that such narrowed changes to the model cannot be unique among all developments. The modelling process might be significantly simplified by having a mechanism of reuse of decisions made for another model. The intuitive analogue of similar reuse in software engineering are design patterns which have had considerable effect on engineering process.

The library of fault tolerance patterns might ease the modelling process of new systems and provide a seamless way to add fault tolerance to existing systems. Examples of fault tolerance techniques which can be added to the model are: exception handling, N-version programming, checkpoints, recovery blocks, etc [3]. Each of these techniques can be decomposed into a number of finer-grained transformations. Such elementary patterns are semantically self-contained and introduce a particular functionality into a model. They might not represent specific fault tolerance techniques on their own. However, by combining elementary patterns, a modeller may get a wide variety of existing fault tolerance

techniques and effectively mix them to create more complex recovery. Examples of such small elements are: creation of replica, state save, voting mechanism, etc.

Fig. 6 shows a simple detalisation template for adding a spare sensor activation. *Activity* is an abstract depiction of a system normal execution, *Stop* is a state of system when its operation is no longer possible. On this level of detalisation, it is not important what exactly constitutes each of those activities. The input diagram essentially shows a separation of system execution into its normal activity and termination. We assume existence of a sensor within an input model which can fail. An output model introduces a maskable error recovery activity which fires when the active sensor has failed. This recovery is obliged to return the system to its normal operation or results in termination if there are no spare sensors available. We can see several explicit requirements to an input model: existence of a stop state, and of a sensor with its spares. These requirements need to be formally defined in applicability conditions of an underlying transformation pattern.

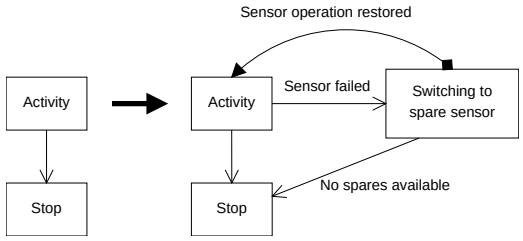

Figure 6: Example template for adding a spare sensor

Another example can be the checkpointing capability for tackling transient faults. The template for the mechanism (Fig. 7) adds two activities: taking a *Checkpoint* and making a *Rollback* action. The checkpoint can be taken regularly with an interval of time, or based on a certain condition on a state of the normal activity. The underlying transformation pattern can be composed of fine-grained elements mentioned above: creation of variables replica and state save. Such details lie on a lower level than the FT view diagram.

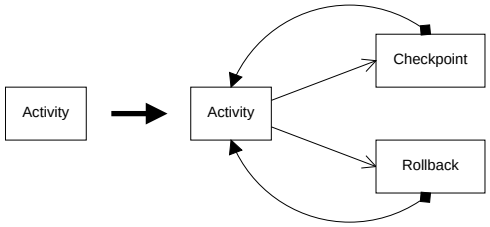

Figure 7: Example template for introducing checkpoints

Obviously, these examples are not tied to any particular domain and can be applied to various developments.

4.2 Possible Implementations

We considered a number of possible transformational approaches to adding FT by refining Event-B models, and drew a general picture of properties necessary for such mechanism. These properties are: usability, generality, and proof support. Usability is an informal property telling how easy it is to create and read transformations. Generality consists of the language expressiveness, and how broad is the context of applying transformations. Proof support is a formal necessity for proving the correctness of a refinement step.

In [9], a transformational approach for Event-B is proposed by the name of *refinement patterns*. A refinement pattern is a generic model transformer which takes a higher level abstract model as its input and produces a correct refined model on output. The pattern is composed from finer-grained model transformation rules. Each such rule is a basic operation which can be performed upon a model, e.g. addition of a variable or event to a model. The rule consists exactly of three parts described above with the formal proof of correctness relying on both offline proving and obligations for user. There is a language for describing sequential, parallel, and conditional rule compositions. The mechanism realises all three aspects we defined although there are certain issues with usability which need to be tackled.

Another possible solution is to rely on the Eclipse Modelling Framework (EMF) [5] and its support for meta-modelling. There are currently two languages we are considering: Epsilon Transformation Language (ETL) and ATLAS Transformation Language (ATL). ETL [12] is built on top of Epsilon Object Language which is an imperative programming language operating over EMF models. ATL [2] is another, more declarative, language also operating on EMF models. Given an existing Event-B meta-model, we can use these languages to transform Event-B models. Both languages are general and provide usability and rich functionality necessary for constructing sufficient transformations. But in order to use it for automating refinements, we need to formally connect it with Event-B by either making formal model of a language and proving its correctness, or generating appropriate proof obligations for a particular transformation.

5. CASE STUDY

The example is a sluice with two doors connecting areas with dramatically different pressures. The pressure difference makes it unsafe to open a door unless the pressure is levelled between the areas connected by the door. The purpose of the system is to adjust the pressure in the sluice area and control the door locks to allow a user to get safely inside or outside. Such system can be deployed, for example, on a submarine to allow divers to get out while submerged.

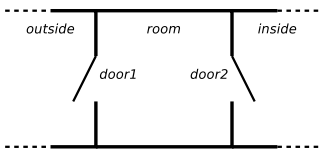

Figure 8: Sluice doors system

The diagram in Figure 8 shows the system elements.

The system description is summarised in the following set of requirements:

1. the system allows a user to get inside or outside by levelling the pressure between the room and the destination area
2. the system has three locations - *outside*, *sluice* and *inside*

3. the system has two doors - *door1*, connecting *outside* and *sluice*, and *door2*, connecting *sluice* and *inside*;

4. there is a device to change the pressure in the *sluice*;

5. a door may be opened only if the pressures in the locations it connects are equalised;

6. at most one door is open at any moment;

7. the pressure can only be changed when the doors are closed.

Requirements 1-4 characterise the system by stating its goal and its major parts. The last three characterise the system behaviour. More precisely, they are the safety properties of the system.

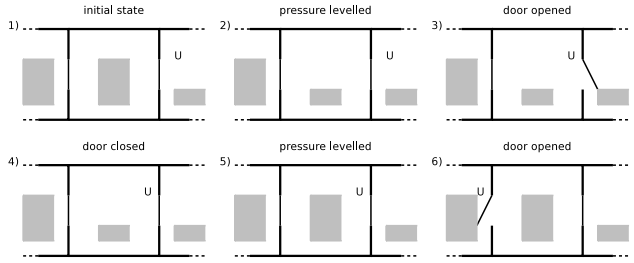

Figure 9: The sequence of steps required to let a sluice user get from inside to outside

Figure 9 shows the stages of the system operation that let a user to get outside starting in the inside area. The shaded rectangles denote a pressure level, high pressure area corresponds to a higher rectangle. The "u" label marks the current position of a system user. Initially, a user is inside and the sluice pressure is levelled with the outside pressure. Before the door connecting to the sluice is opened, the pressure is decreased to level it with the inside pressure. Once the door is open, the user moves in, the door is sealed again. Finally, the sluice pressure is set to match the outside pressure and the door leading outside may be safely opened.

Here we show how such system can be formally modelled with the help of detalisation templates discussed in the paper. All the Event-B models mentioned in this section were modelled in the Rodin tool and proved to be correct. We omit showing them in whole due to lack of space, and focus mainly on the FT view of the system.

By choosing one of abstractions discussed in section 3.2, we start with the most abstract view of a system behaviour by splitting it into *normal* and *degraded* activities (Fig. 10). A degraded activity represents a state when it is no longer possible for a system to operate normally.

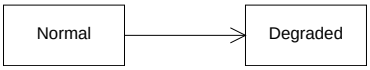

Figure 10: Sluice abstract FT diagram

Let us denote such separation by a variable $normal \in BOOL$ with normal and degraded activities having respective <assumption, guarantee> pairs:

$$A_n/G_n : normal = TRUE \,/\, TRUE$$
$$A_d/G_d : normal = FALSE \,/\, normal' = FALSE$$

An Event-B model conforming to this FT view consists of three events:

```
  normal = when normal = TRUE then skip end
degraded = when normal = FALSE then skip end
error_detected = when
 normal = FALSE
then
 normal := TRUE
end
```

The first two represent activities, and *error_detected* is an abstract depiction of an error detection mechanism which transitions system into its degraded state:

$$A_n/G_n \mapsto \{normal,\ error\_detected\}$$
$$A_d/G_d \mapsto degraded$$

Consistency between the model and the view can be easily established by showing that necessary conditions (1, 2, 3, 6, 7) trivially hold.

Now we refine the normal behaviour to meet the system requirements stated above. First, we define variables representing physical phenomena:

$$v : \begin{array}{c} door1 \in BOOL \\ door2 \in BOOL \\ pressure \in \{0, 1\} \end{array}$$

Next, we cover requirements 5 and 6 by the following invariant statements:

$$I(c, v) : \begin{array}{c} error = FALSE \wedge door1 = TRUE \\ pressure = 0 \\ error = FALSE \wedge door2 = TRUE \\ pressure = 1 \\ error = FALSE \wedge door1 = door2 \\ door1 = FALSE \end{array}$$

The system evolution is described by the following events: *open*1 and *close*1 control the first door, *open*2 and *close*2 control the second, *pressure_low* and *pressure_high* operate the pressure device. All of them refine an abstract event *normal*. The guards and actions of these events are specified to meet the safety requirements 5, 6, and 7. This step refines the normal behaviour of the system and does not introduce any changes into the FT view at the current level of abstraction.

Let us now refine the degraded activity into its more concrete counterparts. By using the first template from section 3.2, we split it into three possible activities from which there is no return to normal operation (Fig. 11).

The *Safe stop* activity triggers when the system can terminate with a state considered to be safe for a user and environment. For example, the doors operation can be further

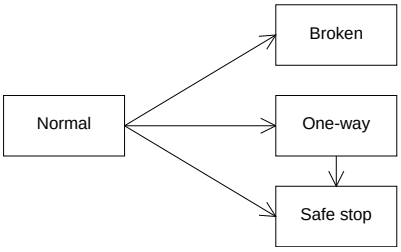

Figure 11: Detalisation of the degraded activity

detalised by adding sensors which might fail. If a sensor of a door fails while the door is opened, we consider this state safe since the pressures on two sides of the sluice are conserved. However, there is a chance of detecting an opened position of a door which must be closed at the moment. Obviously, it is a hazardous situation, possibly caused by the environment conditions, which is not manageable by the system. We define a *Broken* activity for such cases, which could fire an alarm for instance.

Another interesting situation is when the system can still provide its service but in a limited form. Imagine a user is in a sluice during a change of pressure, and both door sensors fail. The system is not fully operational anymore, however it can still equalize pressure levels and unlock one of the doors. After such activity, which we name *One-way* here, the system execution safely stops.

We model these three new activities by a variable $degraded \in \{SAFESTOP, BROKEN, ONEWAY, NONE\}$ and a number of events. The gluing invariant for this refinement is:

$$J(c,v): \begin{array}{c} degraded \neq NONE \Leftrightarrow normal = FALSE \\ degraded = NONE \Leftrightarrow normal = TRUE \end{array}$$

Assumption/guarantees and relation to Event-B events are the following:

$$\begin{aligned} A_n/G_n &: degraded = NONE \,/\, TRUE \\ A_{ss}/G_{ss} &: degraded = SAFESTOP \,/ \\ &\quad degraded' = SAFESTOP \\ A_{br}/G_{br} &: degraded = BROKEN \,/ \\ &\quad degraded' = BROKEN \\ A_{one}/G_{one} &: degraded = ONEWAY \,/ \\ &\quad degraded' = ONEWAY \vee \\ &\quad degraded' = SAFESTOP \end{aligned}$$

$$\begin{aligned} A_n/G_n &\mapsto open1,\ close1,\ open2,\ close2, \\ &\quad pressure\_high,\ pressure\_low, \\ &\quad oneway\_transition,\ safestop\_transition, \\ &\quad broken\_detection \\ A_{ss}/G_{ss} &\mapsto safestop \\ A_{br}/G_{br} &\mapsto broken \\ A_{one}/G_{one} &\mapsto oneway,\ oneway\_to\_safestop \end{aligned}$$

Once again, in order to show consistency with the Event-B model, we need to prove that conditions (1, 2, 3, 6, 7) hold.

Since this FT view is a detalisation of a previous one, we also need to show correspondence (4) and (5) for the normal and degraded activities respectively.

The typical proof obligation generated by the FT view for this case study would look like the following:

$$\begin{aligned} &(degraded \neq NONE \Leftrightarrow normal = FALSE) \wedge \\ &(degraded = NONE \Leftrightarrow normal = TRUE) \wedge \\ &(normal = FALSE) \Rightarrow \\ &(degraded = SAFESTOP) \vee (degraded = BROKEN) \vee \\ &(degraded = ONEWAY \vee degraded = SAFESTOP) \end{aligned}$$

$$\begin{aligned} &(degraded \neq NONE \Leftrightarrow normal = FALSE) \wedge \\ &(degraded = NONE \Leftrightarrow normal = TRUE) \wedge \\ &((degraded' = SAFESTOP) \vee (degraded' = BROKEN) \vee \\ &(degraded' = ONEWAY \vee degraded' = SAFESTOP)) \Rightarrow \\ &normal' = FALSE \end{aligned}$$

It shows a refinement relation (5) between an abstract degraded activity and its three concrete detalisations. Obviously, it holds. In more complex situations, guarantees would contain relations among variables. However, we expect a significant part of proof obligations to be discharged by existing provers within the Rodin platform.

As a simple example of reuse, now we introduce a notion of a masked error into our model. By using our second basic template from section 3.2, we add an abstract recovery activity. Then we add a notion of sensors to our model. A pressure room now has an active sensor along with a hot spare. As soon as we detect a failure of a sensor, we activate a spare one. When the last working sensor fails, the system safely stops (Fig. 12).

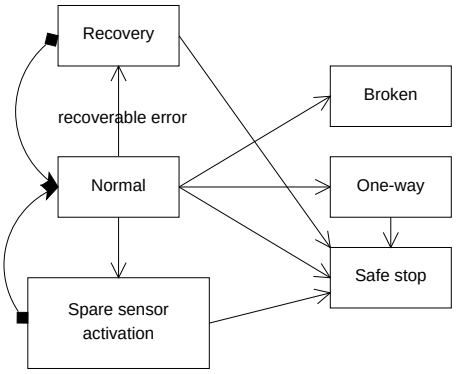

Figure 12: Introducing a masked error and a sensor failure tolerance

We need to proceed with two model refinement steps to achieve correspondence to the final FT view. It can be done manually as for previous refinements. Or the template and the corresponding Event-B model transformation pattern could be borrowed from conceptually existing library as discussed in section 4.1.

6. RELATED WORK

There are a number of works targeting provision of development facilities for FT in formal methods.

Gärtner [7] gives a formal background and a classification of transformational approaches to modelling FT. Three axes are defined for classification:

- level of transformation - system or subsystem;
- object of transformation - program or specification;
- what is introduced by transformation - "good" properties (FT components) or "bad" properties (effects of faults).

Many ideas surveyed in the paper are relevant to our work. E.g., concept of provision of FT along with targeted masked errors, checkpointing mechanism. Another interesting concept discussed - abstract detector/corrector - can be modelled using our approach with further detalisation into specific mechanisms. Attention is also paid to formal modelling of graceful degradation in presence of faults.

One of such approaches is discussed in [11]. It presents a model-based approach to modelling normal and fault-tolerant parts of system behaviour separately, and formal foundations of composition of the two. A concept of partial refinement is introduced and formally described to relate FT behaviour to the whole system behaviour. In this approach, a modeller is given two sets of requirements: for normal part and for FT part with system properties being weakened and faults added. The former is then called to be partially refined by the latter, where the term "partially" formally covers the weakened part of required properties. With such approach applied to Event-B, augmentation of existing refinement relation and the whole tool chain would be inevitable.

In [13] authors introduce a general formal specification approach to be applied in development of dependable systems with a layered architecture. The approach adds exception handling mechanism to each layer of such systems and organizes communication between components within a hierarchical structure by means of exceptions. The exceptions generated by a component are treated as propagated at higher layer. A propagated exception is evaluated by a component as an acknowledgement of normal termination, an indicator of recoverable error occurrence, or an indicator of unrecoverable error occurrence. The representation of exceptions is used to introduce a model of a fault tolerant component in B and the process of unfolding architectural layers by refinement. Exception handling is a widely used mechanism and bringing it into formal development is a reasonable way to improve support for FT development. However, it is a single mechanism, and projects would potentially benefit more from modelling FT mechanisms closer to their domains.

Among transformation languages discussed in section 4.2, there is also a *design pattern* approach to reuse for Event-B [8]. The pattern is an Event-B machine which can be incorporated into the main development chain if they match each other. The main advantage is a reduce of proofs: the pattern is considered to be already modelled and its proof obligations discharged before inserting into the target model. However, it is not a transformational approach. One must explicitly drive the development towards the matching pattern which leads to certain applicability restrictions.

7. CONCLUSIONS AND FUTURE WORK

While it is widely accepted that faults within complex systems are inevitable, providing reliable and systematically-developed fault tolerance for such systems is crucial. In this paper, we introduce an approach to facilitating formal modelling of fault tolerance. A notion of an FT view, orthogonal to model development, is proposed and integrated into the Event-B method. By using the proposed FT view, we encourage the modellers to employ architectural abstractions of fault-tolerant systems at early development phases, and to refine them using the FT detalisation templates. We have also briefly discussed our ongoing work on supporting the FT templates with Event-B model transformation patterns.

We believe that the proposed templates and patterns together constitute a useful combination of abstractions for disciplined and step-wise development of dependable systems and ensure an expressive link with the underlying formalism. Such coupling facilitates creation of libraries of reusable development components supporting faster adoption of widely accepted fault tolerance techniques for formal system development.

Our future work will consist of:

- creating a flexible and open mechanism of model transformation patterns which would be reasonably balanced between usability, generality and provability of transformations,
- connecting it to the FT detalisation templates,
- developing a tool support (as a number of Rodin plugins) supporting both parts of our approach to encourage reuse.

8. ACKNOWLEDGEMENTS

This work is supported by the ICT Deploy Integrated Project on industrial deployment of system engineering methods providing high dependability and productivity. Ilya Lopatkin is partially supported by Newcastle University (UK). We are grateful to our colleagues in Deploy for all their suggestions for improving this work.

9. REFERENCES

[1] J.-R. Abrial. The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996.

[2] F. Allilaire, and T. Idrissi. ADT: Eclipse development tools for ATL. *Second European Workshop on Model Driven Architecture (EWMDA-2)*, 2004.

[3] 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*, 1:11–33, 2004.

[4] R.-J. Back and K. Sere. Stepwise refinement of action systems. *Proceedings of the International Conference on Mathematics of Program Construction*, pages 115–138, London, UK, 1989. Springer-Verlag.

[5] F. Budinsky, S. A. Brodsky, and E. Merks. Eclipse Modeling Framework. Pearson Education, 2003.

[6] F. L. Dotti, A. Iliasov, L. Ribeiro, and A. Romanovsky. Modal systems: specification, refinement and realisation. *International Conference on Formal Engineering Methods (ICFEM'09)*, December 9-12, 2009, Rio de Janeiro, Brazil.

[7] F. C. Gartner. Transformational approaches to the specification and verification of fault-tolerant systems: Formal background and classification. *Journal of Universal Computer Science*, 5(10):668–692, 1999.

[8] T. S. Hoang, A. Furst, and J.-R. Abrial. Event-B patterns and their tool support. *Seventh IEEE International Conference on Software Engineering and Formal Methods (SEFM'09)*, pages 210–219, 2009.

[9] A. Iliasov, and A. Romanovsky. Refinement patterns for fault tolerant systems. *Seventh European Dependable Computing Conference (EDCC'08)*, pages 167–176, 2008, Kaunas, Lithuania.

[10] A. Iliasov, A. Romanovsky, and F. L. Dotti. Structuring specifications with modes. *Fourth Latin-American Symposium on Dependable Computing (LADC'09)*, 0:81–88, 2009, Brazil.

[11] R. Jeffords, C. Heitmeyer, M. Archer, and E. Leonard. A formal method for developing provably correct fault-tolerant systems using partial refinement and composition. *Proceedings of the 2nd World Congress on Formal Methods*, 173–189, 2009.

[12] D. Kolovos, R. Paige, and F. Polack. The Epsilon transformation language. *Lecture Notes in Computer Science*, 5063:46–60. 2008.

[13] L. Laibinis and E. Troubitsyna. Fault tolerance in a layered architecture: a general specification pattern in B. *Second International Conference on Software Engineering and Formal Methods (SEFM'04)*, 346–355, 2004.

[14] C. Metayer, J. Abrial, and L. Voisin, editors. Rodin Deliverable D7: Event-B language. Project IST-511599 RODIN "Rigorous Open Development Environment for Complex Systems", School of Computing Science, Newcastle University, 2005.

[15] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. S. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys (CSUR)*, 41(4), 2009.