Patterns for Representing FMEA in Formal Specification of Control Systems

Ilya Lopatkin, Alexei Iliasov, Alexander Romanovsky, Yuliya Prokhorova and Elena Troubitsyna
Patterns for Representing FMEA in Formal Specification of Control Systems

I. Lopatkin, A. Iliasov, A. Romanovsky, Y. Prokhorova and E. Troubitsyna

Abstract

Failure Modes and Effect analysis (FMEA) is a widely used technique for inductive safety analysis. FMEA provides engineers with valuable information about failure modes of system components as well as procedures for error detection and recovery. In this paper we propose an approach that facilitates representation of FMEA results in formal Event-B specifications of control systems. We define a number of patterns for representing requirements derived from FMEA in formal system model in Event-B. The patterns help the developers to trace the requirements and allow them to increase automation of formal system development by refinement. Our approach is illustrated by an example - a sluice system.
Failure Modes and Effect analysis (FMEA) is a widely used technique for inductive safety analysis. FMEA provides engineers with valuable information about failure modes of system components as well as procedures for error detection and recovery. In this paper we propose an approach that facilitates representation of FMEA results in formal Event-B specifications of control systems. We define a number of patterns for representing requirements derived from FMEA in formal system model in Event-B. The patterns help the developers to trace the requirements and allow them to increase automation of formal system development by refinement. Our approach is illustrated by an example - a sluice system.

About the authors
Ilya Lopatkin graduated from KRSU, Kyrgyzstan in 2008. He is currently interested in formal modelling of fault tolerance for improving the resilience of critical systems.

Alexei Iliasov is a Researcher Associate at the School of Computing Science of Newcastle University, Newcastle-upon-Tyne, UK. He got his PhD in Computer Science in 2008 in the area of modelling artefacts reuse in formal developments. His research interests include agent systems, formal methods for software engineering and tools and environments supporting modelling and proof.

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 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, the 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. He has been the Coordinator of the Rigorous Open Development Environment for Complex Systems (RODIN) IST Project (2004-2007). He is now the Coordinator of the major FP7 DEPLOY Integrated Project (2008-2012) on Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity.

Yuliya Prokhorova is a PhD student at Åbo Akademi University, Department of Information Technologies, Turku, Finland. She got her M.Sc. in Computer Systems and Networks in 2008 at National Aerospace University "KhAI", Kharkiv, Ukraine. Her research interests include application of formal modelling and verification methods and also development and verification of safety-critical and fault-tolerant systems.

E. Troubitsyna - a Professor at Åbo Akademi, Turku, Finland. Her main research interests are Formal modelling and verification methods; Design and verification of safety-critical systems; Development and verification of fault tolerant systems; Probabilistic verification; Integrated modelling approaches; Service-oriented development and System architecture.

Suggested keywords
FORMAL SPECIFICATION
EVENT-B
FMEA
PATTERNS
SAFETY
CONTROL SYSTEMS
Patterns for Representing FMEA in Formal Specification of Control Systems

Ilya Lopatkin, Alexei Iliasov, Alexander Romanovsk
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
{Ilya.Lopatkin, Alexei.Iliasov, Alexander.Romanovsk}@ncl.ac.uk

Yuliya Prokhorova, Elena Troubitsyna
Turku Centre for Computer Science
Department of Information Technologies
Åbo Akademi University
Turku, Finland
{Yuliya.Prokhorova, Elena.Troubitsyna}@abo.fi

Abstract — Failure Modes and Effect analysis (FMEA) is a widely used technique for inductive safety analysis. FMEA provides engineers with valuable information about failure modes of system components as well as procedures for error detection and recovery. In this paper we propose an approach that facilitates representation of FMEA results in formal Event-B specifications of control systems. We define a number of patterns for representing requirements derived from FMEA in formal system model in Event-B. The patterns help the developers to trace the requirements and allow them to increase automation of formal system development by refinement. Our approach is illustrated by an example - a sluice system.

Key words - formal specification; Event-B; FMEA; patterns; safety; control systems

I. INTRODUCTION

Formal modelling and verification are valuable for ensuring system dependability. However, often formal development process is perceived as being too complex to be deployed in industrial engineering process. Hence, there is a clear need for methods that facilitate adopting of formal modelling techniques and increase productivity of their use.

Reliance on patterns – the generic solutions for certain typical problems – facilitates system engineering because it allows the developers to document the best practices and reuse previous knowledge. However, patterns defined for formal system development, e.g., by Hoang et al. [17] focus on describing model manipulations only and do not provide the insight on how to derive a formal model from textual requirements description. The gap between requirements engineering and in particular safety analysis and formal development has a negative impact on requirements traceability and leaves the developers without the guidance on how to represent certain types of requirements in the formal model.

In this paper we propose an approach to automating formal system development by refinement in Event-B. We demonstrate how to connect formal modelling and refinement with Failure Modes and Effects Analysis (FMEA) via a set of patterns.

FMEA is a widely-used inductive technique for safety analysis [5,13,16]. It allows engineers to systematically study the causes of component faults, their global and local effects, and the means to cope with these faults. These requirements are invaluable for ensuring system dependability.

In this paper we propose a set of patterns formalising the requirements derived from FMEA and enabling automatic transformation of system specification to incorporate these results. Our formal modelling framework is Event-B – a state-based formalism for formal system development by refinement and proof-based verification [17]. Event-B has a mature tool support – Rodin platform [4]. Currently, the framework is actively used by several industrial partners of EU FP7 project Deploy for developing dependable systems from various domains.

The approach proposed in this paper allows us to automate the development process by requiring the user merely to choose the types of patterns corresponding to certain generic representation of FMEA results and instantiate these patterns with model-specific information. As a result of pattern application the model is automatically transformed to faithfully represent the desired requirements.

In this paper we illustrate our approach with excerpts from the automated development of a sluice gate system [7]. Formal system development by refinement in Event-B allows us to verify (by proofs) preservation of safety invariants in presence of component failures identified by FMEA. We believe that the proposed approach provides a good support for formal development and improves traceability of safety requirements.

II. MODELLING CONTROL SYSTEMS IN EVENT-B

A. Event-B Overview

The B Method is an approach to the industrial development of highly dependable control systems. The method has been successfully used in the development of several complex real-life applications [9]. Event-B [1] is a specialization of the B Method aimed at facilitating modelling parallel, distributed and reactive systems. The Rodin platform provides an automated support for modelling and verification in Event-B [4].

In Event-B system models are defined using the Abstract Machine Notation. An abstract machine encapsulates the state (the variables) of a model and defines operations on its state.
The machine is uniquely identified by its name \textbf{MACHINE Name}. The state variables of the machine are declared in the \textbf{VARIABLES} clause and initialized in the \textbf{INITIALISATION} event. The variables are strongly typed by constraining predicates of invariants given in the \textbf{INVARINANTS} clause. Usually the invariants also define the properties of the system that should be preserved during system execution. The data types and constants of the model are defined in a separate component called \textbf{CONTEXT}. The behaviour of the system is defined by a number of atomic events specified in the \textbf{EVENTS} clause. An event is defined as follows:

\[
E = \text{WHERE } g \text{ THEN } S \text{ END}
\]

where the guard \(g\) is a conjunction of predicates defined over the state variables, and the action \(S\) is an assignment to the state variables.

The guard defines when the event is enabled. If several events are enabled simultaneously then any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks.

In general, the action of an event is a composition of variable assignments executed simultaneously. Variable assignments can be either deterministic or non-deterministic. The deterministic assignment is denoted as \(x := E(v)\), where \(x\) is a state variable and \(E(v)\) is an expression over the state variables \(v\). The non-deterministic assignment can be denoted as \(x \in S\) or \(x \vdash Q(v, x')\), where \(S\) is a set of values and \(Q(v, x')\) is a predicate. As a result of the non-deterministic assignment, \(x\) gets any value from \(S\) or it obtains such a value \(x'\) that \(Q(v, x')\) is satisfied.

The main development methodology of Event-B is refinement. Refinement formalises model-driven development and allows us to develop systems correct-by-construction. Each refinement transforms the abstract specification to gradually introduce implementation details. For a refinement step to be valid, every possible execution of the refined machine must correspond to some execution of the abstract machine.

The formal semantics of Event-B [1] provides us with a foundation for rigorous reasoning about system correctness. The consistency (invariant preservation) and well-definedness of Event-B models as well as correctness of refinement steps is demonstrated by discharging \textit{proof obligations}. The Rodin platform [4], a tool supporting Event-B, automatically generates the required proof obligations and attempts to automatically prove them. Sometimes it requires user assistance by invoking its interactive prover. However, in general the tool achieves high level of automation (usually over 90%) in proving.

Next we describe specification and refinement of control systems in Event-B. It follows the specification pattern proposed earlier [11].

\textbf{B. Modelling Control Systems}

The control systems are usually cyclic, i.e., at periodic intervals they get input from sensors, process it and output the new values to the actuators. In our specification the sensors and actuators are represented by the corresponding state variables. We follow the systems approach, i.e., model the controller together with its environment – plant. This allows us to explicitly state the assumptions about environment behaviour. At each cycle the plant assigns the variables modelling the sensor readings. They depend on the physical process of the plant and the current state of the actuators. In its turn, the controller reads the variables modelling sensors and assigns the variables modelling the actuators. We assume that the reaction of the controller takes negligible amount of time and hence the controller can react properly on changes of the plant state.

In this paper, we focus on modelling failsafe control systems. A system is failsafe if it can be put into a safe but non-operational state to preclude an occurrence of a hazard.

The general specification pattern for modelling a failsafe control system in Event-B is shown in Fig. 1.

\begin{verbatim}
machine Abs_M
sees Abs_C
variables flag Failure Stop
invariants
flag \in PHASE
Failure \in BOOL
Stop \in BOOL
Failure=FALSE \Rightarrow Stop=FALSE
Failure=TRUE \land flag=PRED
End=STOP

events
event INITILSIATION
then flag = ENV
Failure = FALSE
Stop = FALSE
end

event Environment
where flag = ENV
Failure = FALSE
Stop = FALSE
then flag = DET
end

event Detection
where flag = DET
Failure = FALSE
Stop = FALSE
then flag = CONT
Failure :\in BOOL
end

event Normal_Operation
where flag = PRED
Failure = FALSE
Stop = FALSE
then flag = PRED
Failure = TRUE
Stop = FALSE
end

end

Failure = FALSE
Stop = FALSE
\end{verbatim}

Figure 1. An abstract specification of a control system.

Abstract model \textbf{Abs_M} represents the overall behaviour of the system as an interleaving between the events.
modelling the plant and the controller. The behaviour of the controller has the following stages: Detection; Control (Normal Operation or Error Handling); Prediction. The stages are defined in enumerated set PHASE: [ENV, DET, CONT, PRED]. Variable flag of type PHASE models the current stage.

In the model invariant we declare the types of the variables and define the conditions when the system is operational or stopped. The system must be in operation if it did not fail, and it must stop at the end of the current cycle if a failure occurred.

Events Environment, Normal Operation and Prediction are the very abstract specifications of events (essentially placeholders) modelling environment behaviour, controller reaction and computation of the next expected states of system components correspondingly. These events are defined in details in the consequent refinement steps. Event Detection non-deterministically models the outcome of error detection by assigning value TRUE to variable Failure in case of an error and FALSE otherwise. As a result of error recovery, abstractly modelled by event Error Handling, the normal system operation can be resumed. In this case, the value of Failure is changed to FALSE. However, if the error recovery is unsuccessful, variable Stop obtains value TRUE and the system shuts down, i.e., the specification deadlocks.

In the next section we demonstrate how to arrive at a detailed specification of a control system by refinement in Event-B. We use the sluice gate control system to exemplify the refinement process.

III. REFINEMENT OF CONTROL SYSTEMS IN EVENT-B

A. The Sluice Gate Control System

The general specification pattern given in Fig.1 defines the initial abstract specification for any typical control system, including the sluice gate control system that we describe next. The sluice gate system shown in Fig.2 is a sluice connecting areas with dramatically different pressures [7]. The pressure difference makes it unsafe to open a door unless the pressure is levelled between the areas connected by the sluice door. The purpose of the system is to adjust the pressure in the sluice area. Such a system can be deployed, e.g., on a submarine to allow divers to get into the sea when the submarine is submerged. The sluice gate system consists of two doors - door1 and door2 that can be operated independently of each other and a pressure chamber pump that changes the pressure in the sluice area. There are the following safety requirements imposed on the system. A door may be opened only if the pressure in the locations it connects is equalized. Since the pressure of two environments is different, at most one door can be opened at any moment. The pressure chamber pump can only be switched on when both doors are closed.

The sluice gate system is equipped with the following sensors and actuators:

- three pressure sensors return the current pressure values in the room and in the two areas adjacent to the room;
- two door position sensors give the current positions of two doors respectively. Each sensor has a cold spare – a redundant sensor to which the system can automatically switch;
- two switch sensors are attached to each door – these signal when the door is fully opened or closed;
- pressure chamber pump actuator changes the pressure inside the room;
- two-way door motors open and close the doors.

The system has physical redundancy (the door position sensors have spares) and information redundancy (when the doors are fully opened or closed, the door position sensor readings should be in accordance with the switch sensors).

B. Introducing Error Detection and Recovery by Refinement

At the first refinement step we aim at introducing models of system components, error detection procedures for their failure modes, as well as error masking and recovery actions. We postpone refinement of the normal functional behaviour of the system until the next refinement step.

To systematically define failure modes, detection and recovery procedures, for each component we conduct Failure Modes and Effect Analysis, FMEA [5,13,16] is a well-known inductive safety analysis technique. For each system component it defines its possible failure modes, local and system effects of component failures, as well as detection and recovery procedures. For instance, on Fig.3 is an excerpt from FMEA of the Door1 component of our sluice system.

The Door1 component is composed of several hardware units. Their failures correspond to the failure modes of the Door1 component. For the sake of brevity, we omit showing FMEA of all failure modes of Door1 and next discuss how to specify error detection and recovery for the failure mode described in the FMEA table in Fig.3.
In the refined specification we introduce the variables representing the units of Door1: door position sensor - door1_position_sensor, motor - door1_motor and door opened and closed sensors - door1_opened_sensor, door1_closed_sensor. In event Environment we introduce the actions that change the values of door1_position_sensor, door1_closed_sensor and door1_opened_sensor. In event Normal_Operation we define the action that non-deterministically changes the value of door1_motor.

We refine event Detection by splitting it into a group of events responsible for the detection of each mode of failures of all system components. We introduce variable door1_fail to designate a failure of the door component. This failure is assigned TRUE when any failure mode of Door1 component is detected. Event Detection_door1_checks included in this group contains the actual checks for value ranges and consistency:

\[
\text{event Detection_Door1_checks}
\begin{align*}
\text{where} & \quad \text{flag} = \text{DET} \\
\text{Stop} & = \text{FALSE} \\
\text{then} & \\
\text{door1_position_sensor_pred} & = \text{bool}(\text{door1_position_sensor} < d1\_exp\_min \lor \text{door1_position_sensor} > d1\_exp\_max) \\
\text{door1_sensor_disregard} & = \text{FALSE} \\
\text{door1_closed_sensor_inconsistent} & = \text{bool}(\text{door1_closed_sensor} \neq \text{TRUE}) \\
\text{flag} & = \text{CONT}
\end{align*}
\]

Variables d1_exp_min and d1_exp_max are the new variables introduced to model the next expected sensor readings. These variables are updated in the Prediction event. Event Detection_Door1 combines the results of the checks of the status of the door1 component.

The failure of component Door1 is detected if any check of the error detection events for any of its failure modes finds a discrepancy between a fault free and the observed states. In a similar manner, the system failure is detected if a failure of any of system components – Door1, Door2 or PressurePump is detected, as specified in event Detection_Fault.

### Patterns and Tool for Representing Results of FMEA in Event-B

#### A. Patterns for Representing FMEA results

Our approach aims at structuring and formalising FMEA results via a set of generic patterns. These patterns serve as a middle hand between informal requirements description and their formal Event-B terms.

While deriving the patterns we assume that the abstract system specification adheres to the generic pattern given in Fig.1 and components can be represented by the corresponding state variables. Our patterns establish a correspondence between the results of FMEA and the Event-B terms.

We distinguish four groups of patterns: detection, recovery, prediction and invariants. The detection patterns reflect such generic mechanisms for error detection as discrepancy between the actual and expected component state, sensor reading outside of the feasible range etc. The recovery patterns include retry of actions or computations, switch to redundant components and safe shutdown. The prediction patterns represent the typical solutions for computing estimated states of components, e.g., using the underlying physical system dynamics or timing constraints. Finally, the invariant patterns are usually used in combination with other types of patterns to postulate how a model transformation affects the model invariant. This type contains safety and gluing invariant patterns. The safety
invariant patterns define how safety conditions can be introduced into the model. The gluing invariant patterns depict the correspondence between the states of refined and abstract model.

A pattern is a model transformation that upon instantiation adds or modifies certain elements of Event-B model. By *elements* we mean the terms of Event-B mathematical language such as variables, constants, invariants, events, guards etc. A pattern can add or modify several elements at once. Moreover, it can be composed of several other patterns.

To illustrate how FMEA results can be interpreted according to the proposed types of patterns, let us consider FMEA of a door1 position sensor in Fig.4.

<table>
<thead>
<tr>
<th>Component</th>
<th>Door1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure mode</td>
<td>Door position sensor value is different from the expected range of values</td>
</tr>
<tr>
<td>Possible cause</td>
<td>Failure of the position sensor</td>
</tr>
<tr>
<td>Local effects</td>
<td>Sensor reading is out of expected range</td>
</tr>
<tr>
<td>System effects</td>
<td>Switch to degraded or manual mode or shut down</td>
</tr>
<tr>
<td>Detection</td>
<td>Comparison of the received value with the predicted range of values</td>
</tr>
<tr>
<td>Remedial action</td>
<td>The same as for Fig.3</td>
</tr>
</tbody>
</table>

Figure 4. FMEA table for “out of predicted range” failure mode of a positioning sensor

We map the FMEA table to a set of patterns which together represent the desired phenomena in the model. The patterns are shown in a declarative form for illustration purposes. The identifiers shown in brackets will be substituted by those given by a user during the pattern instantiation (see next sections).

Firstly, our sensor is a value type sensor. Such an assumption leads to an introduction of a variable representing the value of the sensor by the *Value sensor* pattern:

```plaintext
variables [sensor]_value
invariants [sensor]_value : NAT
events event INITIALISATION
then [sensor]_value := 0
end
```

The value pattern creates a new variable, its typing invariant, and an initialisation action. To detect the failure mode, we use the *Expected range pattern*:

```plaintext
variables [component]_[sensor]_[error] [sensor]_fail [sensor]_exp_min [sensor]_exp_max
invariants [component]_[sensor]_[error] : BOOL
[component]_fail : BOOL
[sensor]_exp_min : NAT
[sensor]_exp_max : NAT
```

```plaintext
events event Detection_[component]_checks
where flag = DET
Stop = FALSE
then [component]_[sensor]_[error] := bool([sensor]_value < [sensor]_exp_min ∨ [sensor]_value > [sensor]_exp_max)
<other checks> end
event Detection_[component]
where flag = DET
Stop = FALSE
then [component]_fail := bool([component]_[sensor]_[error] ∨ <other check statuses>) end
```

This pattern adds the detection events, the necessary variables, and ensures that the detection checks added previously by other patterns are preserved (informally shown in angle brackets). The expected range of values used by this pattern must be assigned by some event on the previous control cycle. To ensure that such assignment exists in the model, the *Expected range pattern* instantiates the *Range prediction pattern*:

```plaintext
variables [sensor]_exp_min [sensor]_exp_max
invariants [sensor]_exp_min : NAT
[sensor]_exp_max : NAT
events event Prediction extends Prediction
then [sensor]_exp_min := NAT
[sensor]_exp_max := NAT
<other sensor predictions> end
```

This pattern leaves the prediction non-deterministic for further refinement because the actual prediction depends on the functionality of the system under development.

Note how two previous patterns work with the same variables. In such case, only the first pattern to be instantiated actually creates variables as one would expect. The same applies to events, actions, guards etc.

To connect the model created using patterns with the abstract level, we use the *Gluing invariant pattern*:

```plaintext
variables [component]_fail
invariants flag=DET ⇒ (Failure=TRUE ⇒ [component]_fail=TRUE ∨ <other component failures>)
flag=CONT ⇒ ([component]_fail=TRUE ⇒ [component]_[sensor]_[error]=TRUE ∨ <other sensor errors>)
```

```plaintext
```
which links the sensor error to the component failure, and contributes the component failure to the gluing invariant thus preserving the refinement relation.

For our example, the remedial action can be divided into three actions. The first action retries reading the sensor for a specified number of times (Retry recovery pattern). The second action disables the faulty component and enables its spare (Component redundancy recovery pattern). The third action, when the spare component is failed either, is to switch the system from operational state to non-operational one (Safe stop recovery pattern). The system effect can be represented as a safety property (Safety invariant pattern). We omit showing all the patterns due to the lack of space.

As shown in the example, each FMEA field is mapped to one or more patterns. Patterns have interdependencies between them and they are composable. For instance, the recovery patterns have to have references to the variables set by the sensor, and thus depend on the results of the Value sensor pattern, the Expected value detection pattern needs to instantiate the Range prediction pattern to have the values predicted from the previous control cycle. Each pattern creates Event-B elements specific to the pattern, and requires elements created by other patterns. Such interdependency and mapping to FMEA is schematically shown on Fig.5. Note how the Expected range pattern creates new constants and variables (dark grey rectangle, variable \([\text{sensor}]_{\exp \_\text{min}}\) from the example) and will instantiate the Value sensor pattern to create the elements it depends on (light grey rectangle, variable \([\text{sensor}]_{\text{value}}\) from the example).

B. Automation of Patterns Implementation

The automation of the pattern instantiation is implemented as a tool plug-in for the Rodin platform [4]. Technically, each pattern is a program written in a simplified Eclipse Object Language (EOL). It is a general purpose programming language in the family of languages of the Epsilon framework [10] which operates on EMF [3] objects. It is a natural choice for automating model transformations since Event-B is interoperable with EMF.

The tool extends the application of EOL to Event-B models: it adds simple user interface features for instantiation, extends the Epsilon user input facility with discovery of the Event-B elements, and provides a library of Event-B and FMEA-specific transformations.

To apply a pattern, a user chooses a target model and a pattern to instantiate. A pattern application may require user input: variable names or types, references to existing elements of the model etc. The input is performed through a series of simple dialogs.

The requested input comprises the applicability conditions of the pattern. In many cases it is known that instantiation of a pattern depends primarily on the results of a more basic pattern. In those cases the former directly instantiates the latter and reuses the user input. Also more generally, if several patterns require the same unit of user input then the composition of such patterns will ask for such input only once. Typically, a single pattern instantiation requires up to 3-4 inputs.

If a pattern only requires user input and creates new elements then its imperative form is close to declarative as shown in the example below:

```java
var flag: Variable = chooseOrCreateVariable("Phase variable");
createTypingInvariant(flag, "PHASE");

var failure: Variable = chooseOrCreateVariable("Failure variable");
createTypingInvariant(failure, "BOOL");
newEvent("Detection")
  .addGuard("phase_grd", flag.name + " = DET")
  .addGuard("failure_grd", failure.name + " = FALSE")
  .addAction("phase_act", flag.name + " := CONT")
  .addAction("failure_act", failure.name + " ::= BOOL");
```

Here the tool will ask the user to select two variables (or creates new ones). It will create typing invariants and a new model event with several guards and actions. Next we illustrate the use of the tool in the refinement of our sluice gate case study.

V. AUTOMATED REFINEMENT PROCESS

A. Automated refinement step

In section 3 we presented an excerpt showing how to (manually) model unreliable positioning sensor and error recovery. In this section we demonstrate how to automate the first refinement step. Fig.4 shows FMEA table for the “out of predicted range” failure mode of the door position sensor.
Below we show an excerpt from a model obtained automatically via instantiation and application of several patterns.

### Variables

<table>
<thead>
<tr>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>door1_position_sensor</code>, <code>door1_fail</code>, <code>door1_position_sensor_pred</code>, <code>d1_exp_max</code>, <code>d1_exp_min</code>, <code>retry</code></td>
</tr>
</tbody>
</table>

### Event Definitions

**RetryPosition**

```plaintext
where
flag = CONT

`door1_position_sensor_abs` = TRUE v `door1_position_sensor_pred` = TRUE v retry < 3

then

`door1_position_sensor_abs` = FALSE
`door1_position_sensor_pred` = FALSE

`door1_fail_masked` = bool(
  `door1_opened_sensor_inconsistent` = TRUE v `door1_closed_sensor_inconsistent` = TRUE)

retry = retry + 1

end
```

**Detection_Door1_checks**

```plaintext
where
flag = DET

Stop = FALSE

then

`door1_position_sensor_pred` = bool(
  (`door1_position_sensor` < `d1_exp_min` v `door1_position_sensor` > `d1_exp_max`) & `door1_sensor_diagnose` = FALSE)

<other checks>

end
```

**SafeStop**

```plaintext
where
flag = DET

Stop = FALSE

then

`door1_position` = 0

<other checks>

end
```

Upon instantiation, the *Expected value detection* and *Value sensor patterns* ensure that the necessary variables exist, and the detection events are appropriately modified. The *Expected value detection pattern* also instantiates the *Range prediction pattern* which adds a non-deterministic assignment to event *Prediction*. The *Retry recovery pattern* adds the *RetryPosition* event. This event masks the sensor failure for the current control cycle, and counts the number of retries. Upon an occurrence of a sensor failure for a given number of times (3 in this example), the system has to shut down. This is achieved by the event *SafeStop*, which is generated by the pattern with the same name.

The *Gluing invariant* and *Safety invariant patterns* generate the gluing and safety invariants correspondingly. The gluing invariants establish correspondence between abstract and refined states. In particular, it stipulates the relationships between the failures of all system components and the overall system failure, as well as between component failure and the results of error detection of their constituent units. As shown below, the safety invariant states that a *door1* failure must lead to a safe stop.

### Gluing invariant

```plaintext
@glue flag=DET = (Failure=TRUE => door1_fail=TRUE v door2_fail=TRUE v pressure_fail=TRUE)
@glue door1_fail flag=CONT => (door1_FAIL=TRUE =>
  `door1_position_sensor_pred` = TRUE v
  `door1_position_sensor_abs` = TRUE v
  `door1_opened_sensor_inconsistent` = TRUE v
  `door1_closed_sensor_inconsistent` = TRUE)

@ safety door1_fail=TRUE & flag=CONT & flag=DET =>
  Stop=TRUE
```

### Further Refinement Steps

As the result of the first refinement step we have obtained a specification that contains the detailed description of the FMEA-derived detection and recovery procedures. However, the normal control operations are modelled non-deterministically. In the second refinement step we introduce the detailed specification of the normal control logic. This refinement step leads to refining event *Normal_Operation* into a group of events that model the actual control algorithm. These events model opening and closing the doors as well as activation of the pressure chamber pump.

Refinement of the normal control operation results in restricting non-determinism. This allows us to formulate safety invariants that our system guarantees:

```plaintext
failure = FALSE & `door1_position` = `door1_position` =>
  `door1_position` = 0

failure = FALSE & (`door1_position` > 0 v
  `door1_motor` = MOTOR_OPEN) =>
  pressure_value = PRESSURE_OUTSIDE

failure = FALSE & (`door2_position` > 0 v
  `door2_motor` = MOTOR_OPEN) =>
  pressure_value = PRESSURE_INSIDE

failure = FALSE & pressure_value != PRESSURE_INSIDE &
  pressure_value != PRESSURE_OUTSIDE =>
  `door1_position`=0 &
  `door2_position`=0

failure = FALSE & pumps=PUMP_OFF =>
  (`door1_position`=0 &
  `door2_position`=0)
```

These invariants formally define the safety requirements informally described in subsection 3.A. While verifying the correctness of this refinement step we formally ensure (by proofs) that safety is preserved while the system is operational.

At the consequent refinement steps we introduce the error recovery procedures. This allows us to distinguish between criticality of failures and ensure that if a non-critical failure occurs then the system can still remain operational.

### VI. Discussion

#### A. Related Work

Integration of the safety analysis techniques with formal system modelling has attracted a significant research attention over the last few years. There are a number of approaches that aim at direct integration of the safety analysis techniques into formal system development. For instance, the work of Ortmeier et al. [14] focuses on using statecharts to formally represent the system behaviour. It
aims at combining the results of FMEA and FTA to model the system behaviour and reason about component failures as well as overall system safety. Our approach is different – we aim at automating the formal system development with the set of patterns instantiated by FMEA results. The application of instantiated patterns automatically transforms a model to represent the results of FMEA in a coherent and complete way. The available automatic tool support for the top-down Event-B modelling as well as for plug-in instantiation and application ensures better scalability of our approach.

In our previous work, we have proposed an approach to integrating safety analysis into formal system development within the Action System formalism [18]. Since Event-B incorporates the ideas of Action Systems into the B Method, the current work is a natural extension of our previous results.

The research conducted by Troubitsyna [19] aims at demonstrating how to use statecharts as a middle ground between safety analysis and formal system specifications in the B Method. This work has inspired our idea of deriving Event-B patterns.

Another strand of research aims at defining general guidelines for ensuring dependability of software-intensive systems. For example, Hatebur and Heisel [6] have derived patterns for representing dependability requirements and ensuring their traceability in the system development. In our approach we rely on specific safety analysis techniques rather than on the requirements analysis in general to derive guidelines for modelling dependable systems.

B. Conclusions

In this paper we have made two main technical contributions. Firstly, we derived a set of generic patterns for elicitation and structuring of safety and fault tolerance requirements from FMEA. Secondly, we created an automatic tool support that enables interactive pattern instantiation and automatic model transformation to capture these requirements in formal system development. Our methodology facilitates requirements elicitation as well as supports traceability of safety and fault tolerance requirements within the formal development process.

Our approach enables guided formal development process. It supports the reuse of knowledge obtained during formal system development and verification. For instance, while deriving the patterns we have analysed and generalised our previous work on specifying various control systems [8,11,12].

We believe that the proposed approach and tool support provide a valuable support for formal modelling that is traditionally perceived as too cumbersome for engineers. Firstly, we define a generic specification structure. Secondly, we automate specification of a large part of modelling decisions. We believe that our work can potentially enhance productivity of system development and improve completeness of formal models.

As a future work we are planning to create a library of domain-specific patterns and automate their application. This would result in achieving even greater degree of development automation and knowledge reuse.

ACKNOWLEDGMENT

The work reported in this paper is supported by FP7 ICT DEPLOY.

REFERENCES