Ilya Lopatkin | Yuliya Prokhorova |
Elena Troubitsyna | Alexei Iliasov |
Alexander Romanovsky

Patterns for Representing FMEA in Formal Specification of Control Systems

Turku Centre for Computer Science

TUCS Technical Report
No 1003, March 2011

Patterns for Representing FMEA in Formal Specification of Control Systems

Ilya Lopatkin
Newcastle University, UK
Ilya.Lopatkin@ncl.ac.uk

Yuliya Prokhorova
Turku Centre for Computer Science, Åbo Akademi University, Finland
Yuliya.Prokhorova@abo.fi

Elena Troubitsyna
Åbo Akademi University, Finland
Elena.Troubitsyna@abo.fi

Alexei Iliasov
Newcastle University, UK
Alexei.Iliasov@ncl.ac.uk

Alexander Romanovsky
Newcastle University, UK
Alexander.Romanovsky@ncl.ac.uk

TUCS Technical Report
No 1003, March 2011

Abstract

Failure Modes and Effect analysis (FMEA) is a widely used technique for inductive safety analysis. FMEA provides the engineers with the 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 the requirements derived from FMEA in formal system model in Event-B. These patterns facilitate traceability of requirements and allow us to increase automation of formal system development by refinement. Our approach is illustrated by an example - a sluice system.

Keywords: formal specification, Event-B, FMEA, patterns, safety, control systems

TUCS Laboratory
Distributed Systems Laboratory

1. 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 negative impact on requirements traceability and leaves the developers without the guidance on how to represents 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 the engineers systematically study of the causes of components 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 [1]. 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 to develop 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 from excerpts from the automated development of sluice gate system [7].

Formal system development by refinement in Event-B allows us to verify (by proofs) preservation of safety invariants event 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.

2. Modelling Control Systems in Event-B

2.1. Event-B Overview

The B Method is an approach for 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 *MachineName*. The state variables of the machine are declared in the **VARIABLES** clause and initialized in the *INITIALISATION* event. The variables are strongly typed by constraining predicates of invariants given in the **INVARIANTS** clause. Usually the invariant also defines 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 **CONTEXT**. The behaviour of the system is defined by a number of atomic events specified in the **EVENTS** clause. An event is defined as follows:

$$E = \textbf{WHEN } g \textbf{ THEN } S \textbf{ END}$$

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 $E(v)$ expression over the state variables v. The non-deterministic assignment can be denoted as $x :\in S$ *or* $x :\mid Q(v, x')$, 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 *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].

2.2. 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.

```
machine Abs_M sees Abs_C
variables flag Failure Stop
invariants
    flag ∈ PHASE
    Failure ∈ BOOL
    Stop ∈ BOOL
    Failure=FALSE ⇒ Stop=FALSE
    Failure=TRUE ∧ flag≠CONT ⇒ Stop=TRUE
events
 event INITIALISATION
  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 :∈ BOOL
 end
 event Normal_Operation
  where
    flag = CONT
    Failure = FALSE
    Stop = FALSE
  then
    flag ≔ PRED
 end
 event Error_Handling
  any res
  where
    flag = CONT
    Failure = TRUE
    Stop = FALSE
    res∈BOOL
  then
    flag ≔ PRED
    Stop ≔ res
    Failure ≔ res
 end
 event Prediction
  where
    flag = PRED
    Failure = FALSE
    Stop = FALSE
  then
    flag ≔ ENV
 end
end
```

Fig. 1. An abstract specification of a control system.

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

In the model invariant we declare the types of the variables and define conditions when the system is operational or stopped.

The 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. These events will be defined in details in the consequent refinement steps. The event **Detection** non-deterministically models the outcome of error detection by assigning the value TRUE to the variable *Failure* in case of an error and FALSE otherwise. As a result of error recovery, abstractly modelled by the 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, the variable *Stop* obtains the value TRUE and the system is shut 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.

3. Refinement of Control Systems in Event-B

3.1. 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.

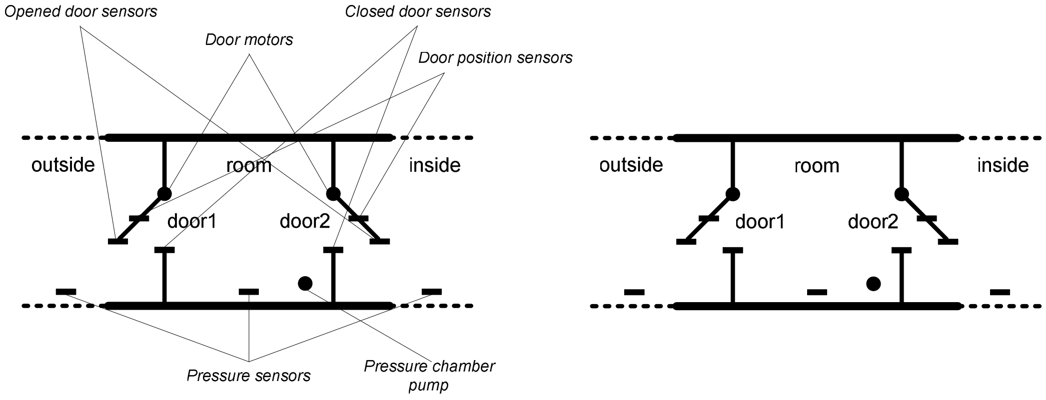


Fig. 2. Sluice gate system.

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

- three *pressure sensors* – they return the current pressure values in the room and in the two areas adjacent to the room;
- two *door position sensors* – they 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* attached to each door – they signal when the door is fully opened or closed;
- *pressure chamber pump actuator* – it changes the pressure inside the room
- two-way *door motors* - they open and close the doors

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

3.2. 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 effect of component failures, as well as detection and recovery procedures. For instance, below is an excerpt from FMEA of *Door1* component of our sluice system.

The *Door1* component is composed of several hardware units. Their failures correspond to the failure modes of *Door1* component. For the sake of brevity, we omit showing FMEA for all failure modes of *Door1* and next discuss how to specify error detection and recovery for the failure mode described in FMEA table in Fig.3.

Component	Door1
Failure mode	Door position sensor value is different from the door closed sensor value
Possible cause	Failure of position sensor or closed sensor
Local effects	Sensor readings are not equal in corresponding states
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of the values received from position and closed sensors
Remedial action	Retry three times. If failure persists then switch to redundant sensor, diagnose motor failure. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm.

Fig.3. FMEA table.

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 the event **Environment** we introduce the actions that change the values of *door1_position_sensor, door1_closed_sensor* and *door1_opened_sensor.* In the event **Normal_Operation** we define the action that non-deterministically changes the value of *door1_motor.*

We refine the 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 the 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. The event **Detection_door1_checks** included in this group contains the actual checks for value ranges and consistency:

```
event Detection_Door1_checks
  where
   grd1 flag = DET
   grd2 Stop = FALSE
  then
   act1 door1_position_sensor_pred ≔ bool((door1_position_sensor < d1_exp_min ∨
        door1_position_sensor > d1_exp_max) ∧ door1_sensor_disregard=FALSE)
   act2 door1_closed_sensor_inconsistent ≔ bool(¬(door1_closed_sensor=TRUE ⇔
        (door1_position=0 ∨ door1_sensor_disregard=TRUE)))
   <other checks>
end
```

The 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. The event **Detection_Door1** combines the results of the checks of the status of the *door1* component as shown below.

The failure of the 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 the similar manner, the system failure is detected if failure of any of system component – Door1, Door2 or *PressurePump* is detected, as specified in the event *Detection_Fault.*

```
event Detection_Doors1
  where
    grd1 flag = DET
    grd2 Stop = FALSE
  then
    act1 door1_fail ≔ bool(
        door1_position_sensor_pred=TRUE ∨
        door1_closed_sensor_inconsistent=TRUE ∨
     <other check statuses>)
end
```

```
event Detection_Fault refines Detection
  where
    grd1 flag = DET
    grd2 Stop = FALSE
    grd3 door1_fail=TRUE ∨
         door2_fail=TRUE ∨
         pressure_fail = TRUE
  with
    Failure' Failure'=TRUE
  then
    act1 flag ≔ CONT
end
```

Observe that by performing FMEA of all system components we obtain a systematic textual description of all procedures required to detect component errors and perform their recovery. We gradually by refinement introduce the specification of these requirements into the system model.

While analysing the refined specification it is easy to note that there are several typical specification solutions called patterns that represent certain groups of requirements. This prompts the idea of creating an automated tool support that would automatically transform a specification by applying the patterns chosen and instantiated by the developer. In the next section we describe the essence and usage of such a tool.

4. Patterns and Tool for Representing results of FMEA in Event-B

4.1. 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 model.

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 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 patterns. The safety patterns define how safety conditions can be introduced into the model. The gluing 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 patterns, let us consider FMEA of an abstract sensor. We assume that our sensor is a value type sensor. We analyse the failure mode of providing incorrect data. To detect such a fault, we compare received value with the predicted one (*Expected value detection pattern*). The remedial action in this case 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, it so 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*). Moreover, we have to apply *Gluing invariant pattern* to establish a correspondence between the refinement step introducing a model of unreliable sensor and the abstract specification. Fig. 4 shows how patterns are instantiated by the requirements defined in FMEA.

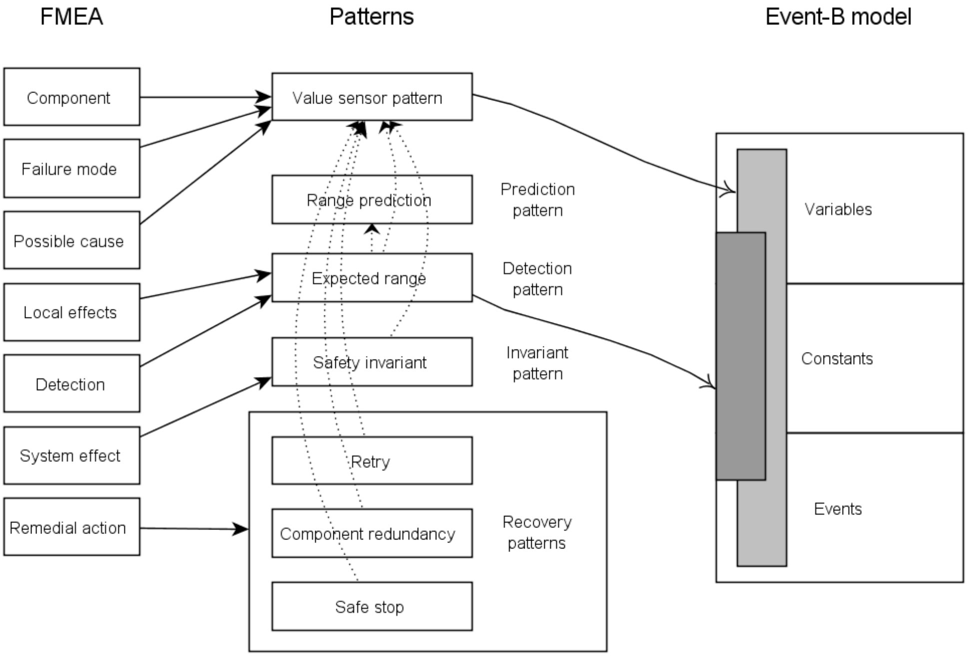


Fig. 4. FMEA representation patterns.

Each FMEA field is mapped to one or more patterns. Patterns have interdependencies between them and hence 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.

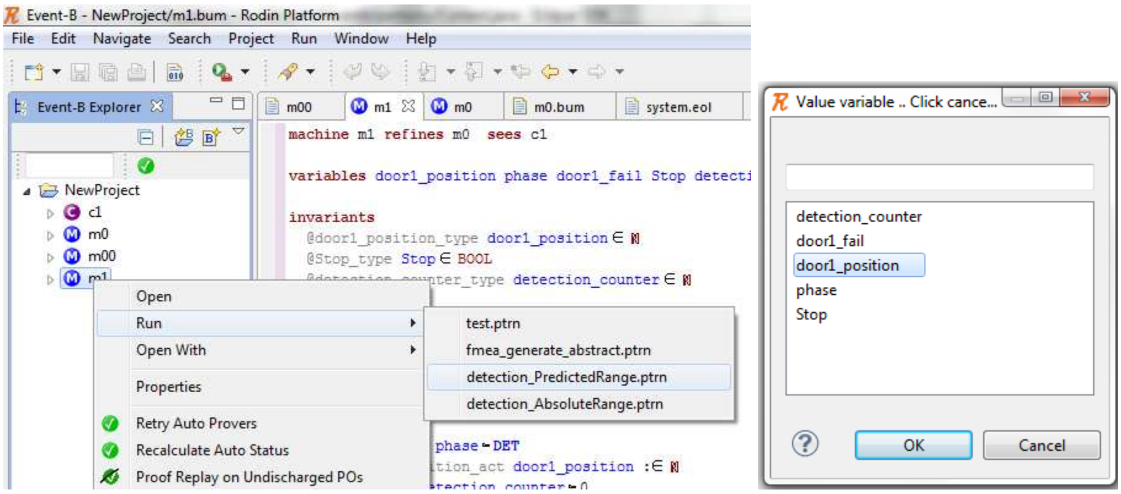
Each pattern creates Event-B elements specific to the pattern, and requires elements created by other patterns. The illustrative example on Fig. 4 shows that instantiating the *Expected range pattern* would create new constants and variables (dark grey rectangle) and will instantiate the *Value sensor pattern* to create the elements it depends on (light grey rectangle).

4.2. Automation of Patterns Implementation

The automation of the pattern instantiation is implemented as a tool plugin 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 as shown in Fig. 5. A pattern application may require user input, e.g., to variable names or types, define 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.



```
machine m1 refines m0  sees c1

variables door1_position phase door1_fail Stop detecti

invariants
  @door1_position_type door1_position ∈ ℕ
  @Stop_type Stop ∈ BOOL
  @detection_counter_type detection_counter ∈ ℕ
phase ≔ DET
tion_act door1_position :∈ ℕ
```

Fig.5. Screenshots of plug-in tool.

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:

```
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 a new model event with several guards and actions. Next we illustrate the use of tool in the refinement of our sluice gate case study.

5. Automated Refinement Process

5.1. 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.6 shows FMEA table for the “out of predicted range” failure mode of the door position sensor.

Component	Door1
Failure mode	Door position sensor value out of expected range
Possible cause	Loss of precision of sensor or motor failure
Local effects	Sensor reading is out of expected range
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of received value with the predicted one
Remedial action	The same as for Fig.3

Fig. 6. FMEA table for “out of predicted range” failure mode of positioning sensor.

Below we show an excerpt from a model obtained automatically via instantiation and application of several patterns.

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 the 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.

```
variables door1_position_sensor
          door1_fail
          door1_position_sensor_pred
          d1_exp_max
          d1_exp_min



event RetryPosition
  where
    grd1 flag = CONT
    grd_pos door1_position_sensor_abs = TRUE ∨
            door1_position_sensor_pred = TRUE
    grd_retry retry<3
  then
    act1 door1_position_sensor_abs ≔ FALSE
    act2 door1_position_sensor_pred ≔ FALSE
    act3 door1_fail_masked ≔ bool(
         door1_opened_sensor_inconsistent=TRUE ∨
         door1_closed_sensor_inconsistent=TRUE)
    act4 retry ≔ retry + 1
end
```

```
event Detection_Door1_checks
  where
    grd1 flag = DET
    grd2 Stop = FALSE
  then
    act1 door1_position_sensor_pred ≔ bool(
         (door1_position_sensor < d1_exp_min
        ∨ door1_position_sensor > d1_exp_max)
        ∧ door1_sensor_disregard=FALSE)
    <other checks>
end

event SafeStop refines ErrorHandling
  where
    grd1 flag = CONT
    grd2 (door1_fail=TRUE ∧
          door1_fail_masked=TRUE) ∨
          door2_fail=TRUE ∨
          pressure_fail=TRUE
    grd3 Stop = FALSE
  with
    res=TRUE
  then
    act1 flag ≔ PRED
    act2 Stop ≔ TRUE
end
```

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.

```
invariants
  @glue  flag≠DET ⇒ (Failure=TRUE ⇔ door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail=TRUE)

  @glue_door1_fail  flag≠CONT ⇒  (door1_fail=TRUE ⇔
                    door1_position_sensor_abs=TRUE ∨  door1_position_sensor_pred=TRUE ∨
                    door1_opened_sensor_inconsistent=TRUE ∨  door1_closed_sensor_inconsistent=TRUE)

  @safety  door1_fail=TRUE ∧ flag≠CONT ∧ flag≠DET ⇒  Stop=TRUE
```

5.2. 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 the 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:

failure = FALSE ∧ door1_position = door1_position ⇒ door1_position = 0
failure = FALSE ∧ (door1_position > 0 ∨ door1_motor=**MOTOR_OPEN**) ⇒
pressure_value = **PRESSURE_OUTSIDE**
failure = FALSE ∧ (door2_position > 0 ∨ 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 ∧ pump≠**PUMP_OFF** ⇒ (door1_position=0 ∧ door2_position=0)

These invariants formally define the safety requirements informally described in subsection 3.1. While verifying 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.

6. Discussion

6.1. 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.

6.2. 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 results in achieving even greater degree of development automation and knowledge reuse.

Acknowledgments

This work is supported by FP7 ICT DEPLOY.

References

[1] J.-R. Abrial, “Modeling in Event-B: System and Software Engineering”, Cambridge University Press, 2010.

[2] Deploy project. www.deploy-project.eu/

[3] Eclipse GMT – Generative Modeling Technology, http://www.eclipse.org/gmt

[4] Event-B and the Rodin Platform. Retrieved from http://www.event-b.org/, 2010.

[5] FMEA Info Centre. Retrieved from http://www.fmeainfocentre.com/, 2009.

[6] D. Hatebur and M. Heisel, “A Foundation for Requirements Analysis of Dependable Software”, Proceedings of the International Conference on Computer Safety, Reliability and Security (SAFECOMP), Springer, 2009, pp. 311-325.

[7] A.Iliasov. “Modularisation Plug-in Tutorial” http://wiki.event-b.org/index.php/Modularisation_Plug-in_Tutorial

[8] D. Ilic and E. Troubitsyna. Formal Development of Software for Tolerating Transient Faults. In Proc. of the 11th IEEE Pacific Rim International Symposium on Dependable Computing, IEEE Computer Society, Changsha, China, December 2005.

[9] Industrial use of the B method. Retrieved from ClearSy: http://www.clearsy.com/pdf/ClearSy-Industrial_Use_of_%20B.pdf, 2008.

[10] D. S. Kolovos. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon), Official Web-Site. http://www.cs.york.ac.uk/~dkolovos/epsilon.

[11] L. Laibinis, and E. Troubitsyna, "Refinement of fault tolerant control systems in B", SAFECOMP 2004, Springer, Potsdam, Germany, 2004.

[12] L. Laibinis and E. Troubitsyna. Fault Tolerance in a Layered Architecture: a General Specification Pattern in B. In Proc. of International Conference on Software Engineering and Formal Methods SEFM'2004. IEEE Computer Society Press, pp.346-355, Beijing, China, September 2004.

[13] N.G. Leveson, "Safeware: System Safety and Computers", Addison-Wesley, 1995.

[14] F. Ortmeier, M. Guedemann and W. Reif, "Formal Failure Models", Proceedings of the IFAC Workshop on Dependable Control of Discrete Systems (DCDS 07), Elsevier, 2007.

[15] K. Sere, and E. Troubitsyna, "Safety analysis in formal specification". In J. Wing, J. Woodcock, & J. Davies (Ed.), FM'99 – Formal Methods. Proceedings of World Congress on Formal Methods in the Development of Computing Systems, Lecture Notes in Computer Science 1709, II, 1999, pp. 1564-1583.

[16] N. Storey, "Safety-critical computer systems", Addison-Wesley, 1996.

[17] Thai Son Hoang, A, Furst, J.-R. Abrial: Event-B Patterns and Their Tool Support. SEFM 2009: 210-219. IEEE Computer Press 2009.

[18] E. Troubitsyna, "Elicitation and Specification of Safety Requirements", Proceedings of the Third International Conference on Systems (ICONS 2008), 2008, pp. 202-207.

[19] E. Troubitsyna, "Integrating Safety Analysis into Formal Specification of Dependable Systems", Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), 2003, p. 215b.

Appendix A. FMEA of the Sluice Gate System

Table A.1. Failure mode "contradictory sensor data" of Door1 component

Component	Door1
Failure mode	Door position sensor value is different from the door closed sensor value
Possible cause	Failure of position sensor or closed sensor
Local effects	Sensor readings are not equal in corresponding states
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of the values received from position and closed sensors
Remedial action	Retry three times. If failure persists then switch to redundant sensor, diagnose motor failure. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm.

Table A.2. Failure mode "out of predicted range" of Door1 component

Component	Door1
Failure mode	Door position sensor value out of expected range
Possible cause	Loss of precision of sensor or motor failure
Local effects	Sensor reading is out of expected range
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of received value with the predicted one
Remedial action	Retry three times. If failure persists then switch to redundant sensor, diagnose motor failure. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm.

Table A.3. Failure mode "contradictory sensor data" of Door2 component

Component	Door2
Failure mode	Door position sensor value is different from the door closed sensor value
Possible cause	Failure of position sensor or closed sensor
Local effects	Sensor readings are not equal in corresponding states
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of the values received from position and closed sensors
Remedial action	Retry three times. If failure persists then switch to redundant sensor, diagnose motor failure. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm.

Table A.4. Failure mode "out of predicted range" of Door2 component

Component	Door2
Failure mode	Door position sensor value out of expected range
Possible cause	Loss of precision of sensor or motor failure
Local effects	Sensor reading is out of expected range
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of received value with the predicted one
Remedial action	Retry three times. If failure persists then switch to redundant sensor, diagnose motor failure. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm.

Table A.5. Failure mode “out of predicted range” of Pressure chamber component

Component	Pressure chamber
Failure mode	Pressure out of expected range
Possible cause	Loss of precision of sensor or pump failure
Local effects	Sensor reading is out of expected range
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of received value with the predicted one
Remedial action	Retry three times. If failure persists then switch to redundant sensor, diagnose pump failure. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm.

Table A.6. Failure mode “out of predicted range” of Pressure sensor inside component

Component	Pressure sensor inside
Failure mode	Pressure out of expected range
Possible cause	Loss of precision of sensor
Local effects	Sensor reading is out of expected range
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of received value with the predicted one
Remedial action	Retry three times. If failure persists then switch to redundant sensor. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm.

Table A.7. Failure mode “out of predicted range” of Pressure sensor outside component

Component	Pressure sensor outside
Failure mode	Pressure out of expected range
Possible cause	Loss of precision of sensor
Local effects	Sensor reading is out of expected range
System effects	Switch to degraded or manual mode or shut down
Detection	Comparison of received value with the predicted one
Remedial action	Retry three times. If failure persists then switch to redundant sensor. If failure still persists, switch to manual mode and raise the alarm. If no redundant sensor is available then switch to manual mode and raise the alarm.

Appendix B. Patterns for Representing FMEA

Each FMEA table shown in Appendix A corresponds to a set of patterns.

Set of patterns for Table A.1:

Value sensor pattern
Retry recovery pattern
Component redundancy recovery pattern
Safe stop recovery pattern

```
variables door1_position_sensor
          door1_fail
          door1_opened_sensor
          door1_closed_sensor

event Detection_Doors
  where
    @grd1 flag = DET
    @grd3 Stop = FALSE
  then
    act1 door1_position_sensor_abs ≔ bool((
          door1_position_sensor < 0 ∨
          door1_position_sensor > 100) ∧
          door1_sensor_disregard=FALSE)
    act2 door1_position_sensor_pred ≔ bool((
          door1_position_sensor < d1_exp_min ∨
          door1_position_sensor > d1_exp_max) ∧
          door1_sensor_disregard=FALSE)
    act3 door1_opened_sensor_inconsistent ≔ bool(
          ¬(door1_opened_sensor=TRUE ⇔
          (door1_position=100 ∨ door1_sensor_disregard=TRUE)))
    act4 door1_closed_sensor_inconsistent ≔ bool(
          ¬(door1_closed_sensor=TRUE ⇔
          (door1_position=0 ∨ door1_sensor_disregard=TRUE)))
    <other checks>
end

event RetryPosition
  where
    grd1 flag = CONT
    grd_pos door1_position_sensor_abs = TRUE ∨
              door1_position_sensor_pred = TRUE
    grd_retry retry<3
  then
    act1 door1_position_sensor_abs ≔ FALSE
    act2 door1_position_sensor_pred ≔ FALSE
    act3 door1_fail_masked ≔ bool(
         door1_opened_sensor_inconsistent=TRUE ∨
         door1_closed_sensor_inconsistent=TRUE)
    act4 retry ≔ retry + 1
end
```

```
event EnableRedundant
  where
    grd1 flag = CONT
    grd2 retry_done=TRUE
    grd3 door1_sensor_redundant_done=FALSE
    grd4 door1_position_sensor_abs = TRUE ∨
         door1_position_sensor_pred = TRUE
    grd5 door1_sensor_redundant = TRUE
  then
    act1 door1_position_sensor_abs ≔ FALSE
    act2 door1_position_sensor_pred ≔ FALSE
    act3 door1_fail_masked ≔ bool(
         door1_opened_sensor_inconsistent=TRUE ∨
         door1_closed_sensor_inconsistent=TRUE)
    act4 door1_sensor_redundant ≔ TRUE
    act5 door1_sensor_redundant_done≔TRUE
end

event SafeStop refines ErrorHandling
  where
    grd1 flag = CONT
    grd2 (door1_fail=TRUE ∧
          door1_fail_masked=TRUE) ∨
          door2_fail=TRUE ∨
          pressure_fail=TRUE
    grd3 Stop = FALSE
  with
    res=TRUE
  then
    act1 flag ≔ PRED
    act2 Stop ≔ TRUE
end
```

Safety invariant pattern
Gluing invariant pattern

```
invariants
  @glue  flag≠DET ⇒ (Failure=TRUE ⇔ door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail=TRUE)
  @glue_door1_fail  flag≠CONT ⇒  (door1_fail=TRUE ⇔
                    door1_position_sensor_abs=TRUE ∨  door1_position_sensor_pred=TRUE ∨
                    door1_opened_sensor_inconsistent=TRUE ∨  door1_closed_sensor_inconsistent=TRUE)
  @glue_door1_masking flag=CONT ∧ retry_done=TRUE ∧ door1_sensor_redundant_done=TRUE ⇒
                      (door1_fail_masked=TRUE ⇔ door1_position_sensor_abs=TRUE ∨
                       door1_position_sensor_pred=TRUE ∨ door1_opened_sensor_inconsistent=TRUE ∨
                       door1_closed_sensor_inconsistent=TRUE)
  @safety  door1_fail=TRUE ∧ flag≠CONT ∧ flag≠DET ⇒  Stop=TRUE
```

Set of patterns for Table A.2:

Expected value detection pattern
Value sensor pattern
Range prediction pattern
Retry recovery pattern
Component redundancy recovery pattern
Safe stop recovery pattern

```
variables door1_position_sensor
          door1_fail
          door1_position_sensor_pred
          d1_exp_max
          d1_exp_min

event Detection_Door1_checks
 where
  grd1 flag = DET
  grd2 Stop = FALSE
 then
  act1 door1_position_sensor_pred ≔ bool(
     (door1_position_sensor < d1_exp_min
    ∨ door1_position_sensor > d1_exp_max)
    ∧ door1_sensor_disregard=FALSE)
  <other checks>
end

event RetryPosition
 where
  grd1 flag = CONT
  grd_pos door1_position_sensor_abs = TRUE ∨
          door1_position_sensor_pred = TRUE
  grd_retry retry<3
 then
  act1 door1_position_sensor_abs ≔ FALSE
  act2 door1_position_sensor_pred ≔ FALSE
  act3 door1_fail_masked ≔ bool(
       door1_opened_sensor_inconsistent=TRUE ∨
       door1_closed_sensor_inconsistent=TRUE)
  act4 retry ≔ retry + 1
end
```

```
event EnableRedundant
 where
  grd1 flag = CONT
  grd2 retry_done=TRUE
  grd3 door1_sensor_redundant_done=FALSE
  grd4 door1_position_sensor_abs = TRUE ∨
       door1_position_sensor_pred = TRUE
  grd5 door1_sensor_redundant = TRUE
 then
  act1 door1_position_sensor_abs ≔ FALSE
  act2 door1_position_sensor_pred ≔ FALSE
  act3 door1_fail_masked ≔ bool(
       door1_opened_sensor_inconsistent=TRUE ∨
       door1_closed_sensor_inconsistent=TRUE)
  act4 door1_sensor_redundant ≔ TRUE
  act5 door1_sensor_redundant_done≔TRUE
end

event SafeStop refines ErrorHandling
 where
  grd1 flag = CONT
  grd2 (door1_fail=TRUE ∧
        door1_fail_masked=TRUE) ∨
        door2_fail=TRUE ∨
        pressure_fail=TRUE
  grd3 Stop = FALSE
 with
  res=TRUE
 then
  act1 flag ≔ PRED
  act2 Stop ≔ TRUE
end

event Prediction refines Prediction
 where
  grd1 flag = PRED
  grd2 door1_fail=FALSE ∧ door2_fail=FALSE
       ∧ pressure_fail = FALSE
   @grd3 Stop = FALSE
 then
  act1 flag ≔ ENV
  act2 d1_exp_min≔min_door(door1_position↦door1_motor)
  act3 d1_exp_max≔max_door(door1_position↦door1_motor)
  <other predictions>
end
```

Safety invariant pattern
Gluing invariant pattern

```
invariants
 @glue  flag≠DET ⇒ (Failure=TRUE ⇔ door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail=TRUE)

 @glue_door1_fail  flag≠CONT ⇒  (door1_fail=TRUE ⇔
                   door1_position_sensor_abs=TRUE ∨  door1_position_sensor_pred=TRUE ∨
                   door1_opened_sensor_inconsistent=TRUE ∨  door1_closed_sensor_inconsistent=TRUE)

 @safety  door1_fail=TRUE ∧ flag≠CONT ∧ flag≠DET ⇒  Stop=TRUE
```

Set of patterns for Table A.3:

Value sensor pattern
Retry recovery pattern
Component redundancy recovery pattern
Safe stop recovery pattern

```
variables door2_position_sensor
          door2_fail
          door2_opened_sensor
          door2_closed_sensor

event Detection_Doors
  where
    @grd1 flag = DET
    @grd3 Stop = FALSE
  then
   act5 door2_fail ≔ bool((door2_position <
        d2_exp_min ∨ door2_position > d2_exp_max) ∨
        (door2_position < 0 ∨ door2_position > 100 ∨
        door2_fail=TRUE) ∨
        ¬(door2_opened_sensor=TRUE ⇔ door2_position=100) ∨
        ¬(door2_closed_sensor=TRUE ⇔  door2_position=0))
   <other checks>
end

event RetryPosition
  where
   grd1 flag = CONT
   grd_pos door2_position_sensor_abs = TRUE ∨
           door2_position_sensor_pred = TRUE
   grd_retry retry<3
  then
   act1 door2_position_sensor_abs ≔ FALSE
   act2 door2_position_sensor_pred ≔ FALSE
   act3 door2_fail_masked ≔ bool(
        door2_opened_sensor_inconsistent=TRUE ∨
        door2_closed_sensor_inconsistent=TRUE)
   act4 retry ≔ retry + 1
end
```

```
event EnableRedundant
  where
   grd1 flag = CONT
   grd2 retry_done=TRUE
   grd3 door2_sensor_redundant_done=FALSE
   grd4 door2_position_sensor_abs = TRUE ∨
        door2_position_sensor_pred = TRUE
   grd5 door2_sensor_redundant = TRUE
  then
   act1 door2_position_sensor_abs ≔ FALSE
   act2 door2_position_sensor_pred ≔ FALSE
   act3 door2_fail_masked ≔ bool(
        door2_opened_sensor_inconsistent=TRUE ∨
        door2_closed_sensor_inconsistent=TRUE)
   act4 door2_sensor_redundant ≔ TRUE
   act5 door2_sensor_redundant_done≔TRUE
end

event SafeStop refines ErrorHandling
  where
   grd1 flag = CONT
   grd2 (door2_fail=TRUE ∧
         door2_fail_masked=TRUE) ∨
         door1_fail=TRUE ∨
         pressure_fail=TRUE
   grd3 Stop = FALSE
  with
   res=TRUE
  then
   act1 flag ≔ PRED
   act2 Stop ≔ TRUE
end
```

Safety invariant pattern
Gluing invariant pattern

```
invariants
  @glue  flag≠DET ⇒ (Failure=TRUE ⇔ door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail=TRUE)
  @glue_door2_fail  flag≠CONT ⇒  (door2_fail=TRUE ⇔
                    door2_position_sensor_abs=TRUE ∨ door2_position_sensor_pred=TRUE ∨
                    door2_opened_sensor_inconsistent=TRUE ∨ door2_closed_sensor_inconsistent=TRUE)
  @glue_door2_masking flag=CONT ∧ retry_done=TRUE ∧ door2_sensor_redundant_done=TRUE ⇒
                       (door2_fail_masked=TRUE ⇔ door2_position_sensor_abs=TRUE ∨
                        door2_position_sensor_pred=TRUE ∨ door2_opened_sensor_inconsistent=TRUE ∨
                        door2_closed_sensor_inconsistent=TRUE)
  @safety  door2_fail=TRUE ∧ flag≠CONT ∧ flag≠DET ⇒  Stop=TRUE
```

Set of patterns for Table A.4:

Expected value detection pattern
Value sensor pattern
Range prediction pattern
Retry recovery pattern
Component redundancy recovery pattern
Safe stop recovery pattern

```
variables door2_position_sensor
          door2_fail
          door2_position_sensor_pred
          d2_exp_max
          d2_exp_min


event Detection_Door2_checks
  where
    grd1 flag = DET
    grd2 Stop = FALSE
  then
    act1 door2_position_sensor_pred ≔ bool(
         (door2_position_sensor < d2_exp_min
       ∨ door2_position_sensor > d2_exp_max)
       ∧ door2_sensor_disregard=FALSE)
    <other checks>
end

event RetryPosition
  where
    grd1 flag = CONT
    grd_pos door2_position_sensor_abs = TRUE ∨
            door2_position_sensor_pred = TRUE
    grd_retry retry<3
  then
    act1 door2_position_sensor_abs ≔ FALSE
    act2 door2_position_sensor_pred ≔ FALSE
    act3 door2_fail_masked ≔ bool(
         door2_opened_sensor_inconsistent=TRUE ∨
         door2_closed_sensor_inconsistent=TRUE)
    act4 retry ≔ retry + 1
end
```

```
event EnableRedundant
  where
    grd1 flag = CONT
    grd2 retry_done=TRUE
    grd3 door2_sensor_redundant_done=FALSE
    grd4 door2_position_sensor_abs = TRUE ∨
         door2_position_sensor_pred = TRUE
    grd5 door2_sensor_redundant = TRUE
  then
    act1 door2_position_sensor_abs ≔ FALSE
    act2 door2_position_sensor_pred ≔ FALSE
    act3 door2_fail_masked ≔ bool(
         door2_opened_sensor_inconsistent=TRUE ∨
         door2_closed_sensor_inconsistent=TRUE)
    act4 door2_sensor_redundant ≔ TRUE
    act5 door2_sensor_redundant_done≔TRUE
end

event SafeStop refines ErrorHandling
  where
    grd1 flag = CONT
    grd2 (door2_fail=TRUE ∧
          door2_fail_masked=TRUE) ∨
          door1_fail=TRUE ∨
          pressure_fail=TRUE
    grd3 Stop = FALSE
  with
    res=TRUE
  then
    act1 flag ≔ PRED
    act2 Stop ≔ TRUE
end

event Prediction refines Prediction
  where
    grd1 flag = PRED
    grd2 door1_fail=FALSE ∧ door2_fail=FALSE ∧ pressure_fail
          = FALSE
    grd3 Stop = FALSE
  then
    act1 flag ≔ ENV
    act4 d2_exp_min≔min_door(door2_position↦door1_motor)
    act5 d2_exp_max≔max_door(door2_position↦door1_motor)
    <other predictions>
end
```

Safety invariant pattern
Gluing invariant pattern

```
invariants
  @glue  flag≠DET ⇒ (Failure=TRUE ⇔ door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail=TRUE)

  @glue_door1_fail  flag≠CONT ⇒  (door2_fail=TRUE ⇔
                    door2_position_sensor_abs=TRUE ∨  door2_position_sensor_pred=TRUE ∨
                    door2_opened_sensor_inconsistent=TRUE ∨  door2_closed_sensor_inconsistent=TRUE)

  @safety  door2_fail=TRUE ∧ flag≠CONT ∧ flag≠DET ⇒  Stop=TRUE
```

Set of patterns for Table A.5:

Expected value detection pattern
Value sensor pattern
Range prediction pattern
Retry recovery pattern
Component redundancy recovery pattern
Safe stop recovery pattern

```
variables pressure_value
          pressure_fail
          pressure_pred
          pressure_exp_max
          pressure_exp_min

event Detection_Door1_checks
 where
  grd1 flag = DET
  grd2 Stop = FALSE
 then
  act1 pressure_pred ≔ bool(
      (pressure_value < pressure_exp_min
       ∨ pressure_value > pressure_exp_max)
       ∧ pressure_disregard=FALSE)
  <other checks>
end

event RetryPressure
 where
  grd1 flag = CONT
  grd_pos pressure_abs = TRUE ∨
          pressure_pred = TRUE ∧ retry<3
 then
  act1 pressure_abs ≔ FALSE
  act2 pressure_pred ≔ FALSE
  act3 retry ≔ retry + 1
end
```

```
event SafeStop refines ErrorHandling
 where
  grd1 flag = CONT
  grd2 pressure_fail=TRUE
  grd3 Stop = FALSE
 with
  res=TRUE
 then
  act1 flag ≔ PRED
  act2 Stop ≔ TRUE
end

event Prediction refines Prediction
 where
  grd1 flag = PRED
  grd2 door1_fail=FALSE ∧ door2_fail=FALSE ∧ pressure_fail
       = FALSE
  grd3 Stop = FALSE
 then
  act1 flag ≔ ENV
  act6 pressure_exp_min ≔
       min_pressure_exp(pressure_value↦pump)
  act7 pressure_exp_max ≔
       max_pressure_exp(pressure_value↦pump)
  <other predictions>
end
```

Safety invariant pattern
Gluing invariant pattern

```
invariants
 @glue  flag≠DET ⇒ (Failure=TRUE ⇔ door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail=TRUE)

 @safety  pressure_fail =TRUE ∧ flag≠CONT ∧ flag≠DET ⇒  Stop=TRUE
```

Set of patterns for Table A.6:

Expected value detection pattern
Value sensor pattern
Range prediction pattern
Retry recovery pattern
Component redundancy recovery pattern
Safe stop recovery pattern
Safety invariant pattern
Gluing invariant pattern

Set of patterns for Table A.7:

Expected value detection pattern
Value sensor pattern
Range prediction pattern
Retry recovery pattern
Component redundancy recovery pattern
Safe stop recovery pattern
Safety invariant pattern
Gluing invariant pattern

Appendix C. Formal Development of the Sluice Gate System

Context c0

```
context c0
constants ENV DET CONT PRED
sets PHASE
axioms
  @axm1 partition(PHASE, {ENV}, {DET}, {CONT}, {PRED})
end
```

Abstract Machine m0

```
machine m0
sees c0

variables flag Failure Stop
invariants
  @inv1 flag ∈ PHASE
  @inv2 Failure ∈ BOOL
  @inv3 Stop ∈ BOOL
  @inv4 Failure=FALSE ⇒  Stop=FALSE
  @inv5 Failure=TRUE ∧ flag≠CONT  ⇒  Stop=TRUE

events
  event INITIALISATION
    then
      @act1 flag ≔ ENV
      @act2 Failure ≔ FALSE
      @act3 Stop ≔  FALSE
  end

  event Environment
    where
      @grd1 flag = ENV
      @grd2 Failure = FALSE
      @grd3 Stop = FALSE
    then
      @act1 flag ≔ DET
  end

  event Detection
    where
      @grd1 flag = DET
      @grd2 Failure = FALSE
      @grd3 Stop = FALSE
    then
      @act1 flag ≔ CONT
      @act2 Failure :∈ BOOL
  end

  event NormalOperation
    where
      @grd1 flag = CONT
      @grd2 Failure = FALSE
      @grd3 Stop = FALSE
    then
      @act1 flag ≔ PRED
  end

  event ErrorHandling
    any res
    where
      @grd1 flag = CONT
      @grd2 Failure = TRUE
      @grd3 Stop = FALSE
      @grdres res∈BOOL
    then
      @act1 flag ≔ PRED
      @act3 Stop ≔ res
      @act4 Failure ≔ res
  end

  event Prediction
    where
      @grd1 flag = PRED
      @grd2 Failure = FALSE
      @grd3 Stop = FALSE
    then
      @act1 flag ≔ ENV
  end
end
```

Context c1

```
context c1
extends c0

constants min_door max_door
OPEN_DOOR1 OPEN_DOOR2 CLOSE_DOOR1 CLOSE_DOOR2 NULL_CMD
POSITION
MOTOR_OPEN MOTOR_CLOSE MOTOR_OFF
min_pressure_exp max_pressure_exp
PRESSURE_INSIDE PRESSURE_OUTSIDE
PUMP_INC PUMP_DEC PUMP_OFF
FAULT_TYPES

sets CMD MOTOR PUMP

axioms
  @axm1 ∀x·(x∈ℕ ∧ x≥0 ∧ x≤100 ⇔ x∈POSITION) //door position: 0-closed, 100-opened
  @axm2 partition(MOTOR, {MOTOR_OFF},{MOTOR_OPEN},{MOTOR_CLOSE})
  @axm3 min_door ∈ POSITION × MOTOR → POSITION //lesser expectation limit of
          an opening door
  @axm4 ∀x·x∈POSITION ⇒ min_door(x↦MOTOR_OFF)=x  //if the motor is off, we expect
           our door to be stable
  @axm5 ∀x·x∈POSITION ⇒ min_door(x↦MOTOR_OPEN)=x  //during opening the door
           should at least stay the same
  @axm6 ∀x·x∈POSITION  ∧ x>0 ⇒ min_door(x↦MOTOR_CLOSE)<x  //closing
  @axm7 min_door(0↦MOTOR_CLOSE)=0
  @axm10 max_door ∈ POSITION × MOTOR → POSITION
  @axm11 ∀x·x∈POSITION ⇒ max_door(x↦MOTOR_OFF)=x
  @axm12 ∀x·x∈POSITION ∧ x<100 ⇒ max_door(x↦MOTOR_OPEN)>x
  @axm13 ∀x·x∈POSITION ∧ max_door(x↦MOTOR_CLOSE)=x
  @axm14 max_door(100↦MOTOR_OPEN)=100

  theorem @thm1 ∀x,a·x∈POSITION ∧ a∈MOTOR ∧ min_door(x↦a)≤max_door(x↦a)

  @axm20 partition(CMD, {NULL_CMD}, {OPEN_DOOR1}, {OPEN_DOOR2},
{CLOSE_DOOR1}, {CLOSE_DOOR2})

  @axm30 partition(PUMP, {PUMP_OFF},{PUMP_INC},{PUMP_DEC})
  @axm31 min_pressure_exp ∈ℕ ×  PUMP → ℕ
  @axm32 max_pressure_exp ∈ℕ ×  PUMP → ℕ //the same as for the doors
  @axm33 PRESSURE_INSIDE = 100
  @axm34 PRESSURE_OUTSIDE = 0

  @axm40 FAULT_TYPES = 2
end
```

Refinement 1. Machine m1

```
machine m1 refines m0 sees c1

variables
  door1_position
  door1_position_sensor
  door2_position
  d1_exp_min
  d1_exp_max
  d2_exp_min d2_exp_max
  door1_fail
  door1_fail_masked
  door1_position_sensor_abs
  door1_position_sensor_pred
  door1_opened_sensor_inconsistent
  door1_closed_sensor_inconsistent
  door2_fail door1_motor
  door2_motor
  pressure_value
  pressure_exp_min
  pressure_exp_max
  pump
  pressure_fail
  cmd
  flag
  Stop
  door1_opened_sensor
  door1_closed_sensor
  door2_opened_sensor
  door2_closed_sensor
  faults_detected
  retry
  door1_sensor_redundant
  retry_done
  door1_sensor_redundant_done
  door1_sensor_disregard

invariants
  @inv1 door1_position ∈ ℕ
  @inv2 door1_position_sensor ∈ ℕ // 0-closed, 100-open
  @inv3 door1_position_sensor_abs ∈ BOOL
  @inv4 door1_position_sensor_pred ∈ BOOL
  @inv5 door1_opened_sensor_inconsistent∈BOOL
  @inv6 door1_closed_sensor_inconsistent∈BOOL
  @inv7 d1_exp_max∈POSITION
  @inv8 d1_exp_min∈POSITION
  @inv9 door1_fail∈BOOL
  @inv10 door1_fail_masked∈BOOL
  @inv11 door1_sensor_redundant_done∈BOOL
  @inv12 door1_sensor_disregard∈BOOL
  @inv13 door2_position ∈ ℕ
  @inv14 d2_exp_max∈POSITION
```

```
  @inv15 d2_exp_min∈POSITION
  @inv16 door2_fail∈BOOL
  @inv17 faults_detected ∈ ℕ
  @inv18 retry_done ∈ BOOL
  @inv19 door1_fail_masked∈BOOL
  @inv20 door1_motor∈MOTOR
  @inv21 door2_motor∈MOTOR
  @inv22 pressure_value ∈ ℕ
  @inv23 pressure_exp_min ∈ ℕ
  @inv24 pressure_exp_max ∈ ℕ
  @inv25 pressure_fail∈BOOL
  @inv26 door1_sensor_redundant∈BOOL
  @inv27 pump∈PUMP
  @inv28 retry∈ℕ
  @inv29 door1_opened_sensor ∈ BOOL
  @inv30 door1_closed_sensor ∈ BOOL
  @inv31 door2_opened_sensor ∈ BOOL
  @inv32 door2_closed_sensor ∈ BOOL

  @glue flag≠DET ⇒ (Failure=TRUE ⇔ door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail=TRUE)

  @safety1 door1_fail=TRUE ∧ flag≠CONT ∧ flag≠DET ⇒ Stop=TRUE
  @safety2 door2_fail=TRUE ∧ flag≠CONT ∧ flag≠DET ⇒ Stop=TRUE
  @safety3 pressure_fail=TRUE ∧ flag≠CONT  ∧ flag≠DET ⇒ Stop=TRUE

  @glue_door1 flag≠CONT ∧ faults_detected=2 ⇒ (door1_fail=TRUE ⇔
              door1_position_sensor_abs=TRUE ∨ door1_position_sensor_pred=TRUE ∨
              door1_opened_sensor_inconsistent=TRUE ∨ door1_closed_sensor_inconsistent=TRUE)
  @glue_door1_masking flag=CONT ∧ retry_done=TRUE ∧ door1_sensor_redundant_done=TRUE ⇒
                      (door1_fail_masked=TRUE ⇔ door1_position_sensor_abs=TRUE ∨
                       door1_position_sensor_pred=TRUE ∨
                      door1_opened_sensor_inconsistent=TRUE ∨
                       door1_closed_sensor_inconsistent=TRUE)

events
 event INITIALISATION
  then
   @act1 flag ≔ ENV
   @act2 Stop ≔  FALSE
   @act3 door1_position ≔ 0
   @act4 door1_position_sensor ≔ 0
   @act5 door2_position ≔ 0
   @act7 door1_motor≔MOTOR_OFF
   @act8 door2_motor≔MOTOR_OFF
   @act9 d1_exp_min≔0
   @act10 d1_exp_max≔0
   @act11 d2_exp_min≔0
   @act11 d2_exp_max≔0
   @act12 door1_fail≔FALSE
   @act13 door2_fail≔FALSE
   @act14 pressure_value ≔ PRESSURE_INSIDE
   @act15 pressure_fail≔FALSE
   @act16 pressure_exp_min ≔  PRESSURE_INSIDE
   @act17 pressure_exp_max ≔  PRESSURE_INSIDE
   @act18 pump≔PUMP_OFF
```

```
    @act19 door1_opened_sensor ≔ FALSE
    @act20 door1_closed_sensor ≔ TRUE
    @act21 door2_opened_sensor ≔ FALSE
    @act22 door2_closed_sensor ≔ TRUE
    @act23 faults_detected ≔ 0
    @act24 door1_position_sensor_abs ≔  FALSE
    @act25 door1_position_sensor_pred ≔  FALSE
    @act26 door1_opened_sensor_inconsistent ≔  FALSE
    @act27 door1_closed_sensor_inconsistent ≔ FALSE
    @act28 retry_done ≔ FALSE
    @act29 retry ≔  0
    @act30 door1_fail_masked≔ FALSE
    @act31 door1_sensor_redundant≔TRUE
    @act32 door1_sensor_redundant_done≔FALSE
    @act33 door1_sensor_disregard≔FALSE
end

event Environment refines Environment
  where
    @grd1 flag = ENV
    @grd2 Stop = FALSE
  then
    @act1 flag ≔ DET
    @act2 door1_position_sensor :∈ ℕ
    @act4 pressure_value :∈ ℕ
    @act5 door1_opened_sensor :∈ BOOL
    @act6 door1_closed_sensor :∈ BOOL
    @act7 door2_opened_sensor :∈ BOOL
    @act8 door2_closed_sensor :∈ BOOL
    @act9 faults_detected ≔ 0
end

event Detection_Doors
  where
    @grd1 flag = DET
    @grd3 Stop = FALSE
    @grd5 faults_detected = 0
  then
    @act1 door1_position_sensor_abs ≔ bool((door1_position_sensor < 0 ∨
          door1_position_sensor > 100) ∧ door1_sensor_disregard=FALSE)
    @act2 door1_position_sensor_pred ≔ bool((door1_position_sensor < d1_exp_min ∨
          door1_position_sensor > d1_exp_max) ∧ door1_sensor_disregard=FALSE)
    @act3 door1_opened_sensor_inconsistent ≔ bool(¬(door1_opened_sensor=TRUE ⇔
          (door1_position=100 ∨ door1_sensor_disregard=TRUE)))
    @act4 door1_closed_sensor_inconsistent ≔ bool(¬(door1_closed_sensor=TRUE ⇔
          (door1_position=0 ∨ door1_sensor_disregard=TRUE)))
    @act5 door2_fail ≔ bool((door2_position < d2_exp_min ∨ door2_position > d2_exp_max) ∨
          (door2_position < 0 ∨ door2_position > 100 ∨ door2_fail=TRUE) ∨
          ¬(door2_opened_sensor=TRUE ⇔   door2_position=100) ∨
          ¬(door2_closed_sensor=TRUE ⇔   door2_position=0))
    @act6 pressure_fail≔bool(pressure_value < pressure_exp_min ∨
           pressure_value > pressure_exp_max)
    @act10 faults_detected ≔ faults_detected+1
end
```

```
event Detection_Door1_fail
 where
   @grd1 flag = DET
   @grd3 Stop = FALSE
   @grd5 faults_detected = 1
 then
   @act1 door1_fail ≔  bool(door1_position_sensor_abs=TRUE ∨
           door1_position_sensor_pred=TRUE ∨
           door1_opened_sensor_inconsistent=TRUE ∨
           door1_closed_sensor_inconsistent=TRUE)
   @act2 faults_detected ≔ faults_detected+1
end

event Detection_NoFault refines Detection
 where
   @grd1 flag = DET
   @grd3 Stop = FALSE
   @grd4 faults_detected = 2
   @grd2 door1_fail=FALSE ∧ door2_fail=FALSE ∧ pressure_fail = FALSE
 with
   @Failure' Failure'=FALSE
 then
   @act1 flag ≔ CONT
   @act2 retry_done≔FALSE
   @act3 door1_sensor_redundant_done≔FALSE
end

event Detection_Fault refines Detection
 where
   @grd1 flag = DET
   @grd2 Stop = FALSE
   @grd4 faults_detected = 2
   @grd3 door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail = TRUE
 with
   @Failure' Failure'=TRUE
 then
   @act1 flag ≔ CONT
   @act2 retry_done≔FALSE
   @act3 door1_sensor_redundant_done≔FALSE
end

event NormalSkip
refines NormalOperation
 where
   @grd1 flag = CONT
   @grd4 ¬( door1_position_sensor_abs=TRUE ∨ door1_position_sensor_pred=TRUE ∨
            door1_opened_sensor_inconsistent=TRUE ∨ door1_closed_sensor_inconsistent=TRUE)
   @grd2 door1_fail=FALSE ∧ door2_fail=FALSE ∧ pressure_fail = FALSE
   @grd3 Stop = FALSE
 then
   @act1 flag ≔ PRED
   @act2 door1_motor :∈ MOTOR
   @act3 door2_motor :∈ MOTOR
   @act4 pump :∈ PUMP
end
```

```
event RetryPosition
  where
    @grd1 flag = CONT
    @grd2 retry_done = FALSE
    @grd3 door1_position_sensor_abs = TRUE ∨ door1_position_sensor_pred = TRUE
    @grd4 retry<3
  then
    @act1 door1_position_sensor_abs ≔ FALSE
    @act2 door1_position_sensor_pred ≔ FALSE
    @act3 door1_fail_masked ≔ bool( door1_opened_sensor_inconsistent=TRUE ∨
            door1_closed_sensor_inconsistent=TRUE)
    @act4 retry ≔ retry + 1 || @act5 retry_done≔TRUE
end

event RetryFailed
  where
    @grd1 flag = CONT
    @grd2 retry_done=FALSE
    @grd3 ((door1_position_sensor_abs = TRUE ∨ door1_position_sensor_pred = TRUE) ∧ retry=3) ∨
             (door1_position_sensor_abs = FALSE ∧ door1_position_sensor_pred = FALSE)
  then
    @act1 door1_fail_masked ≔ bool(door1_position_sensor_abs = TRUE ∨
            door1_position_sensor_pred = TRUE ∨ door1_opened_sensor_inconsistent=TRUE ∨
            door1_closed_sensor_inconsistent=TRUE) || @act2 retry_done≔TRUE
end

event EnableRedundant
  where
    @grd1 flag = CONT
    @grd2 retry_done=TRUE
    @grd3 door1_sensor_redundant_done=FALSE
    @grd4 door1_position_sensor_abs = TRUE ∨ door1_position_sensor_pred = TRUE
    @grd5 door1_sensor_redundant = TRUE
  then
    @act1 door1_position_sensor_abs ≔ FALSE
    @act2 door1_position_sensor_pred ≔ FALSE
    @act3 door1_fail_masked ≔ bool( door1_opened_sensor_inconsistent=TRUE ∨
            door1_closed_sensor_inconsistent=TRUE)
    @act4 door1_sensor_redundant ≔ TRUE
    @act5 door1_sensor_redundant_done≔TRUE
end

event NoRedundant
  where
    @grd1 flag = CONT
    @grd2 retry_done=TRUE
    @grd3 door1_sensor_redundant_done=FALSE
    @grd4 ((door1_position_sensor_abs = TRUE ∨ door1_position_sensor_pred = TRUE) ∧
              door1_sensor_redundant=FALSE) ∨ (door1_position_sensor_abs = FALSE ∧
              door1_position_sensor_pred = FALSE)
  then
    @act1 door1_fail_masked ≔  bool(door1_position_sensor_abs = TRUE ∨
            door1_position_sensor_pred = TRUE ∨ door1_opened_sensor_inconsistent=TRUE ∨
            door1_closed_sensor_inconsistent=TRUE)
    @act2 door1_sensor_redundant_done≔TRUE
end
```

```
  event SafeStop
  refines ErrorHandling
    where
      @grd1 flag = CONT
      @grd2 (door1_fail=TRUE ∧ door1_fail_masked=TRUE) ∨ door2_fail=TRUE ∨ pressure_fail=TRUE
      @grd3 Stop = FALSE
      @grd4 retry_done=TRUE
      @grd5 door1_sensor_redundant_done=TRUE
    with
      @res res=TRUE
    then
      @act1 flag ≔ PRED
      @act2 Stop ≔  TRUE
      @act3 door1_fail ≔  door1_fail_masked
      @act4 door1_fail_masked≔FALSE
      @act5 retry_done≔FALSE
      @act6 door1_sensor_redundant_done≔FALSE
  end

  event ErrorHandling
  refines ErrorHandling
    where
      @grd1 flag = CONT
      @grd2 door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail=TRUE
      @grd3 Stop = FALSE
      @grd4 retry_done=TRUE
      @grd5 door1_sensor_redundant_done=TRUE
    with
      @res res=bool(door1_fail_masked=TRUE ∨ door2_fail=TRUE ∨ pressure_fail=TRUE)
    then
      @act1 flag ≔ PRED
      @act2 Stop ≔  bool(door1_fail_masked=TRUE ∨ door2_fail=TRUE ∨ pressure_fail=TRUE)
      @act3 door1_fail ≔  door1_fail_masked
      @act4 door1_fail_masked≔FALSE
      @act5 retry_done≔FALSE
      @act6 door1_sensor_redundant_done≔TRUE
  end

  event Prediction refines Prediction
    where
      @grd1 flag = PRED
      @grd2 door1_fail=FALSE ∧ door2_fail=FALSE ∧ pressure_fail = FALSE
      @grd3 Stop = FALSE
    then
      @act1 flag ≔ ENV
      @act2 d1_exp_min≔min_door(door1_position↦door1_motor)
      @act3 d1_exp_max≔max_door(door1_position↦door1_motor)
      @act4 d2_exp_min≔min_door(door2_position↦door1_motor)
      @act5 d2_exp_max≔max_door(door2_position↦door1_motor)
      @act6 pressure_exp_min ≔ min_pressure_exp(pressure_value↦pump)
      @act7 pressure_exp_max ≔ max_pressure_exp(pressure_value↦pump)
  end
end
```

Refinement 2. Machine m2

```
machine m2 refines m1 sees c1

variables
  failure
  flag
  Stop
  pressure_value
  door1_position
  door2_position
  door1_motor
  door2_motor
  pump
  door1_sensor_disregard

invariants
  @failure failure = bool(door1_fail=TRUE ∨ door2_fail=TRUE ∨ pressure_fail=TRUE)
  @safety1 failure = FALSE ∧ door1_position = door1_position ⇒ door1_position = 0 // only one door is
           open at any given moment
  @safety2 failure = FALSE ∧ (door1_position > 0 ∨ door1_motor=MOTOR_OPEN) ⇒
           pressure_value = PRESSURE_OUTSIDE // when the first door is open, the pressure must
           be set to OUTSIDE
  @safety3 failure = FALSE ∧ (door2_position > 0 ∨ door2_motor=MOTOR_OPEN) ⇒
            pressure_value = PRESSURE_INSIDE // when the second door is open, the pressure must
           be set toINSIDE
  @safety4 failure = FALSE ∧ pressure_value ≠ PRESSURE_INSIDE ∧ pressure_value ≠
           PRESSURE_OUTSIDE ⇒ door1_position=0 ∧ door2_position=0 //when the pressure differs
           from both sides - the doors must be closed
  @safety5 failure = FALSE ∧ pump≠PUMP_OFF ⇒ (door1_position=0 ∧ door2_position=0) //the doors
           must be closed when pump is working

events
  event INITIALISATION
    then
      @act1 flag ≔ ENV
      @act2 Stop ≔  FALSE
      @act3 door1_position ≔ 0
      @act4 door2_position ≔ 0
      @act5 door1_motor≔MOTOR_OFF
      @act6 door2_motor≔MOTOR_OFF
      @act7 pressure_value ≔ PRESSURE_INSIDE
      @act8 pump≔PUMP_OFF
      @act9 failure ≔ FALSE
  end

  event open1 refines NormalSkip
    where
      @grd1 pressure_value = PRESSURE_OUTSIDE
      @grd2 door1_position = 0
      @grd3 door2_position = 0
      @grd4 door1_sensor_disregard=FALSE
//do not allow opening the door when the position sensor is faulty
```

```
    @grd5 flag = CONT
    @grd6 failure=FALSE
    @grd7 Stop=FALSE
  then
    @act1 flag ≔ PRED
    @act2 door1_motor ≔  MOTOR_OPEN
end

event opened1 refines NormalSkip
  where
    @grd1 door1_position = 100
    @grd2 door1_motor = MOTOR_OPEN
    @grd3 flag = CONT
    @grd4 failure=FALSE
    @grd5 Stop=FALSE
  then
    @act1 flag ≔ PRED
    @act2 door1_motor ≔  MOTOR_OFF
end

event close1 refines NormalSkip
  where
    @grd1 door1_position = 100
    @grd2 flag = CONT
    @grd3 failure=FALSE
    @grd4 Stop=FALSE
  then
    @act1 flag ≔ PRED
    @act2 door1_motor ≔  MOTOR_CLOSE
end

event closed1 refines NormalSkip
  where
    @grd1 door1_position = 0
    @grd2 door1_motor = MOTOR_CLOSE
    @grd3 flag = CONT
    @grd4 failure=FALSE
    @grd5 Stop=FALSE
  then
    @act1 flag ≔ PRED
    @act2 door1_motor ≔  MOTOR_OFF
end

event pressure_high refines NormalSkip
  where
    @grd1 door1_position = 0
    @grd2 door2_position = 0
    @grd3 pressure_value = PRESSURE_OUTSIDE
    @grd0_1 flag = CONT
    @grd0_2 failure=FALSE
    @grd0_3 Stop=FALSE
  then
    @act1 flag ≔ PRED
    @act2 pump ≔  PUMP_INC
end
```

```
 event pressure_highed refines NormalSkip
  where
    @grd1 pump = PUMP_INC
    @grd2 pressure_value = PRESSURE_INSIDE
    @grd3 flag = CONT
    @grd4 failure=FALSE
    @grd5 Stop=FALSE
  then
    @act1 flag ≔ PRED
    @act2 pump ≔  PUMP_OFF
 end

 event pressure_low refines NormalSkip
  where
    @grd1 door1_position = 0
    @grd2 door2_position = 0
    @grd3 pressure_value = PRESSURE_INSIDE
    @grd4 flag = CONT
    @grd5 failure=FALSE
    @grd6 Stop=FALSE
  then
    @act1 flag ≔ PRED
    @act2 pump ≔  PUMP_DEC
 end

 event pressure_lowed refines NormalSkip
  where
    @grd1 pump = PUMP_DEC
    @grd2 pressure_value = PRESSURE_OUTSIDE
    @grd3 flag = CONT
    @grd4 failure=FALSE
    @grd5 Stop=FALSE
  then
    @act1 flag ≔ PRED
    @act2 pump ≔  PUMP_OFF
 end
end
```

At the resulting model we show all detection and recovery events for door1 only as they are identical to those for door2 and chamber pump and can be found in Appendix B.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-2571-0
ISSN 1239-1891