Formal Modelling and Safety Analysis of an Embedded Control System for Construction Equipment: an Industrial Case Study using VDM

Takayuki Mori[1,2]

[1] School of Computing Science, Newcastle University, UK
[2] Komatsu Ltd., Japan

Abstract. This paper reports on an industrial application of formal methods to develop an embedded control system for construction equipment. Informal specifications and safety requirements of the system are formalised using a formal modelling language VDM, and the derived model is used for safety analysis of the system. In our approach, we introduce a kind of modelling pattern: a fault framework, which abstracts the notion of faults and can be widely exploited for analysis and design of control systems dealing with faults. The results of validation and safety analysis of the model are presented, and it is revealed that our modelling approach is effectively applied to the analysis of a practical embedded control system in industry.

1 Introduction

A number of functions of modern construction equipment are realised by embedded control systems in order to achieve desired performance, e.g. low fuel consumption and emission as well as high productivity and comfort. The scale and complexity of control software are rapidly increasing. This makes it difficult to ensure the correctness of the software by conventional approaches such as testing and human review. Formal techniques are expected to be promising approaches to make the control software more reliable.

Safety is a critical factor in the control systems of construction equipment. In order to ensure safety, the Failure Mode and Effects Analysis (FMEA) method [5] has been used for decades. In FMEA, we identify all potential faults of the system to be developed and assess the effect of each fault. If the effect is not negligible for the system from the viewpoint of safety or functionality, a way of detecting the fault and a measure to be taken in case of the fault occurrence should be determined to guarantee a certain level of safety or functionality.

The FMEA process is usually carried out manually by system experts. However, the growth in the scale and complexity of the control system makes the task itself more complicated and difficult. For example, it could be possible that a measure against some fault would cause a side effect to another portion of the control system and lead to an

unpredictable behaviour. For the above reason, we aim to describe formally the specifications of fault detection and associated measures of the system using a formal modelling notation VDM [1, 2], and check if the system satisfies certain safety properties.

This paper reports on a case study of applying VDM to safety analysis of a transmission controller for a wheel loader (a digging and loading vehicle). The controller is responsible for gear change (including forward-reverse change) of the transmission, which transfers engine power to the wheels. The transmission consists of a number of gears and clutches hydraulically controlled by the system. By engaging the proper combination of clutches, the rotating direction of an axle (that is, the moving direction of the vehicle) and the gear ratio of the transmission are determined.[3] The moving direction of the vehicle is specified by a direction lever which is mounted on a steering column and manipulated by an operator of the vehicle. The proper gear is selected by a gear change algorithm implemented in the controller according to the vehicle speed and the engine revolution etc. In our current research, however, we simply focus on a part of the control system, a specification for detecting the direction lever position, and investigate if its safety properties are guaranteed when some fault occurs in the system. This is mainly because the wheel loader has characteristics that its moving direction is frequently switched by the operator for digging and loading work, and detecting the direction lever position is a crucial factor in the system. Moreover, the scale of the system seems to be moderate for our initial trial.

The rest of the paper is organised as follows. The next section describes informal specifications and safety requirements of the control system considered in the case study. In Section 3, we present a formal model of the system along with a kind of modelling pattern: a fault framework. Section 4 describes the results of validation and safety analysis of the system. Finally, Section 5 concludes the paper. The full VDM++ model for the case study is provided in Appendix A.

2 Informal Description of the System

In this section, we informally describe the specifications and the safety requirements of the control system under consideration.[4]

2.1 Control Specifications

The control system consists of the direction lever and the transmission controller. Figure 1 shows the system diagram. Each component is described as follows:

Direction Lever: It is an input device to the transmission controller, mounted on the steering column of the vehicle and manipulated by the operator. The lever has three positions, namely forward (F), neutral (N) and reverse (R), specifying the direction to go. It generates three digital and one analogue input signals. For redundancy, the

[3] The transmission is 4-speed in both forward and reverse directions.

[4] The example in this paper is simplified to some extent from the original control specifications. Furthermore, tangible data values, such as voltage etc., are not shown explicitly and denoted by symbols for confidentiality reasons.

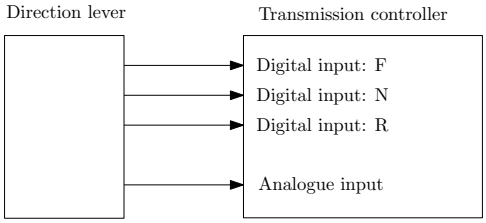

Fig. 1. System Diagram

former are used as primary signals and the latter is used as a backup. The electrical characteristics of the digital and the analogue input signals are illustrated in Figs. 2 and 3 respectively.

Digital Input Signals: Only one of the three signals F, N or R is "on" depending on the lever position. The signals do not overlap one another. That is, there need to exist areas in which no digital input signals are "on" between the lever positions F and N, and between N and R. The lever can be intentionally held in the middle of the lever positions. This means that we cannot distinguish open-circuit of the digital input from the case in which the lever is being held in the middle position.

Analogue Input Signal: It is a voltage signal which ranges depending on the lever position, indicating a value from v_{R1} to v_{R2} at the position R, v_{N1} to v_{N2} at the position N and v_{F1} to v_{F2} at the position F. It has some tolerance at each position.

The possible combination of the digital and the analogue input signals is specified in Table 1. For instance, when the lever is set to the position R, only the digital input signal R should be "on" and the analogue input signal should be between v_{R1} and v_{R2}. In case the lever is held in the middle of the positions R and N, the following are possible:

1. The digital input signal R is "on" and the analogue input signal is between v_{R1} and v_{N2}.
2. No digital input signals are "on" and the analogue input signal is between v_{R1} and v_{N2}.
3. The digital input signal N is "on" and the analogue input signal is between v_{R2} and v_{N2}.

This means it is possible that the lever positions detected by the digital and the analogue input are different from each other (though the possibility is low in reality) and this makes the control specifications complicated.

Notes. In older types of control system, the direction lever consisted of only digital input signals. In case a fault has occurred in the system, the lever position is simply regarded as N, which is allowed from a safety perspective. But from the viewpoint

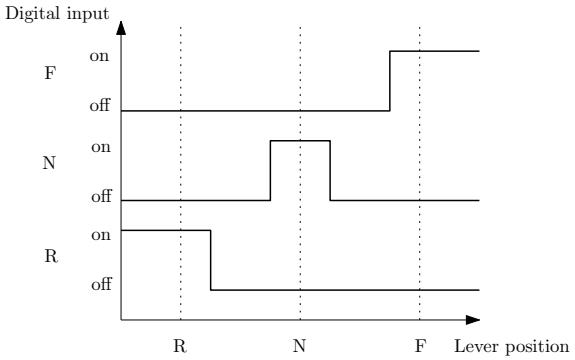

Fig. 2. Electrical Characteristics of the Digital Input Signals

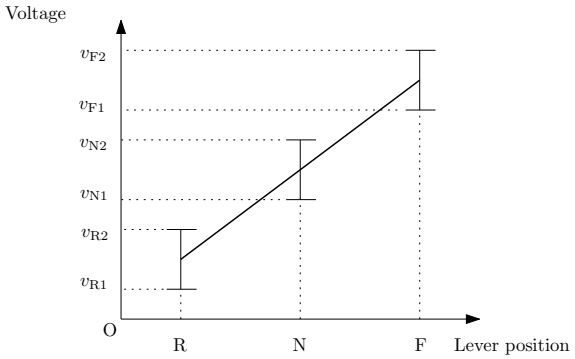

Fig. 3. Electrical Characteristics of the Analogue Input Signal

Table 1. Possible Combination of the Digital Input and the Analogue Input

Direction lever position	Digital input signal	Analogue input signal				
		R	Mid(RN)	N	Mid(FN)	F
		v_{R1}–v_{R2}	v_{R2}–v_{N1}	v_{N1}–v_{N2}	v_{N2}–v_{F1}	v_{F1}–v_{F2}
R	R	◯	–	–	–	–
Mid(RN)	R	×	×	×	–	–
	None	×	×	×	–	–
	N	–	×	×	–	–
N	N	–	–	◯	–	–
Mid(FN)	N	–	–	×	×	–
	None	–	–	×	×	×
	F	–	–	×	×	×
F	F	–	–	–	–	◯

◯ : normal position, × : possible in the middle position, – : impossible

of functionality, it is desirable that the vehicle can move even if some fault occurs in the system. For the above reason, we have worked on adding an analogue input signal to the system to improve redundancy. This not only makes the control specifications complicated, but also makes safety analysis of the system difficult. This motivates us to apply formal methods to our safety analysis process.

Transmission Controller: It detects the position of the direction lever using the digital and the analogue input signals. In detection, it also diagnoses each input signal and takes the proper measures if some fault has occurred. The specifications for detecting the direction lever position are described as follows. Table 2 shows how to detect the lever position by the digital input signals. If only one signal is "on", the signal indicates the lever position. If no or multiple signals are "on", the lever position is determined by a fault measure.
Table 3 specifies the lever position detection by the analogue input signal. In case the signal indicates a voltage of a middle position of the lever, the position is recognised as N. If it is out of range (too low or high), the lever position is determined by a fault measure.

We identify the following six fault modes possible in the system.

F1: Digital input: open-circuit or short-circuit to ground (minor fault)
F2: Digital input: open-circuit or short-circuit to ground (severe fault)
F3: Digital input: short-circuit to power
F4: Analogue input: open-circuit or short-circuit to ground
F5: Analogue input: short-circuit to power
F6: Analogue input: internal circuit fault

A way of detecting each fault and a measure which should be taken in case the fault is detected are specified. As an example, we show the specification of F1 in Table 4. The table instructs how to detect the fault and what to do in case of the fault. Specifically, if "error state" holds, the system starts to detect the fault and takes a "measure before fault confirmation". If the error state has continued for "fault detecting time", it is confirmed that the fault has occurred, and a "measure after fault confirmation" is taken. After that, if "recovery state" holds continuously for "recovery detecting time", it is confirmed that the fault has recovered, and a "measure after fault recovery" is taken.
Some fault modes deserve comment. Both F1 and F2 indicate open-circuit or short-circuit to ground of the digital input signals. For the reason that we cannot distinguish the fault from the case in which the lever is being held in the middle position (as mentioned above), the fault is detected in two-stage manner. The fault detecting time of F1 (minor fault) is set to a value less than that of F2 (severe fault) so that F1 is detected earlier than F2. In case the occurrence of F1 has been confirmed, the system gives an alarm. If the operator puts the lever back to the normal position (if possible) and the error state no longer holds, the fault F1 recovers and the alarm stops. If the operator does not put the lever back or open-circuit has actually occurred, the fault F2 is confirmed eventually.[5]

[5] Strictly speaking, F1 is not regarded as a fault in the system, though this is not directly relevant to our case study.

Table 2. Detection of the Direction Lever Position by the Digital Input Signals

No.	Digital input signals			Detected lever position
	R	N	F	
1	◯			R
2		◯		N
3			◯	F
4				Undefined. Obey fault detection and measure.
5	◯	◯		Undefined. Obey fault detection and measure.
6	◯		◯	
7		◯	◯	
8	◯	◯	◯	

◯ : on, blank: off

Table 3. Detection of the Direction Lever Position by the Analogue Input Signal

No.	Analogue input voltage (A_{in})	Detected lever position
1	$A_{\mathrm{in}} < v_{\mathrm{R1}}$	Undefined. Obey fault detection and measure.
2	$v_{\mathrm{R1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{R2}}$	R
3	$v_{\mathrm{R2}} < A_{\mathrm{in}} < v_{\mathrm{N1}}$	N (middle position between R and N)
4	$v_{\mathrm{N1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{N2}}$	N (normal position)
5	$v_{\mathrm{N2}} < A_{\mathrm{in}} < v_{\mathrm{F1}}$	N (middle position between F and N)
6	$v_{\mathrm{F1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{F2}}$	F
7	$v_{\mathrm{F2}} < A_{\mathrm{in}}$	Undefined. Obey fault detection and measure.

Table 4. An Example of Fault Detection and Measure

Fault mode	Digital input: open-circuit or short-circuit to ground
Error state	All digital input signals F, N and R are “off”.
Fault detecting time	t_{1f} seconds
Measure before fault confirmation	Keep the detected lever position before the error state.
Measure after fault confirmation	Obey the detected lever position by the analogue input.
Recovery state	Only one digital input signal F, N or R is “on”.
Recovery detecting time	t_{1r} seconds
Measure after fault recovery	Keep obeying the detected lever position by the analogue input until it becomes consistent with that by the digital input. After the consistency, obey the detected lever position by the digital input.

The fault F6 indicates impossible combination of the digital and the analogue input signals specified in Table 1. The system detects the situation as a fault of an internal circuit.

In short, the specifications are summarised as follows:

1. If the digital input signals are normal, the position detected by the digital input signals is valid.
2. If the digital input signals have a fault, the position detected by the analogue input signal is valid.
3. Even if the digital input signals have recovered from the fault, the position detected by the analogue input signal is still valid until the detected positions by the digital and the analogue input signals are consistent with each other. Once the consistency is reached, the position detected by the digital input signals becomes valid.
4. If both the digital and the analogue input signals respectively have a fault, the lever position is recognised as N.

2.2 Safety Requirements

We informally describe the safety requirements to be satisfied by the control system as follows:

R1: If any fault occurs in the system, the detected position of the direction lever must be consistent with the actual lever position or recognised as neutral (N), i.e. if the actual position is F, the detected position must be F or N; if the actual position is N, the detected position must be N; and if the actual position is R, the detected position must be R or N.

R2: If any fault occurs in the system, the detected position of the direction lever must not change to F or R without lever manipulation by the operator of the vehicle.

The above requirement R1 inhibits the vehicle from moving in the opposite direction of the lever position or moving while the lever is set to N. It is allowable from the safety viewpoint that the lever position is recognised as N by a fault measure while the actual position is not N. The requirement R2 inhibits the vehicle from moving suddenly as opposed to the operator's intention.

3 Formal Modelling of the System

We model the control system described informally in the previous section using an object-oriented formal modelling notation VDM++ [2], because the notions of the object-oriented method, e.g. inheritance, encapsulation etc., seem to be useful also in formal specification description. The overview of the model (class diagram) is illustrated in Fig. 4. In the following subsections, we explain the characteristics of the model and describe each class in detail. The full VDM++ model is given in Appendix A.

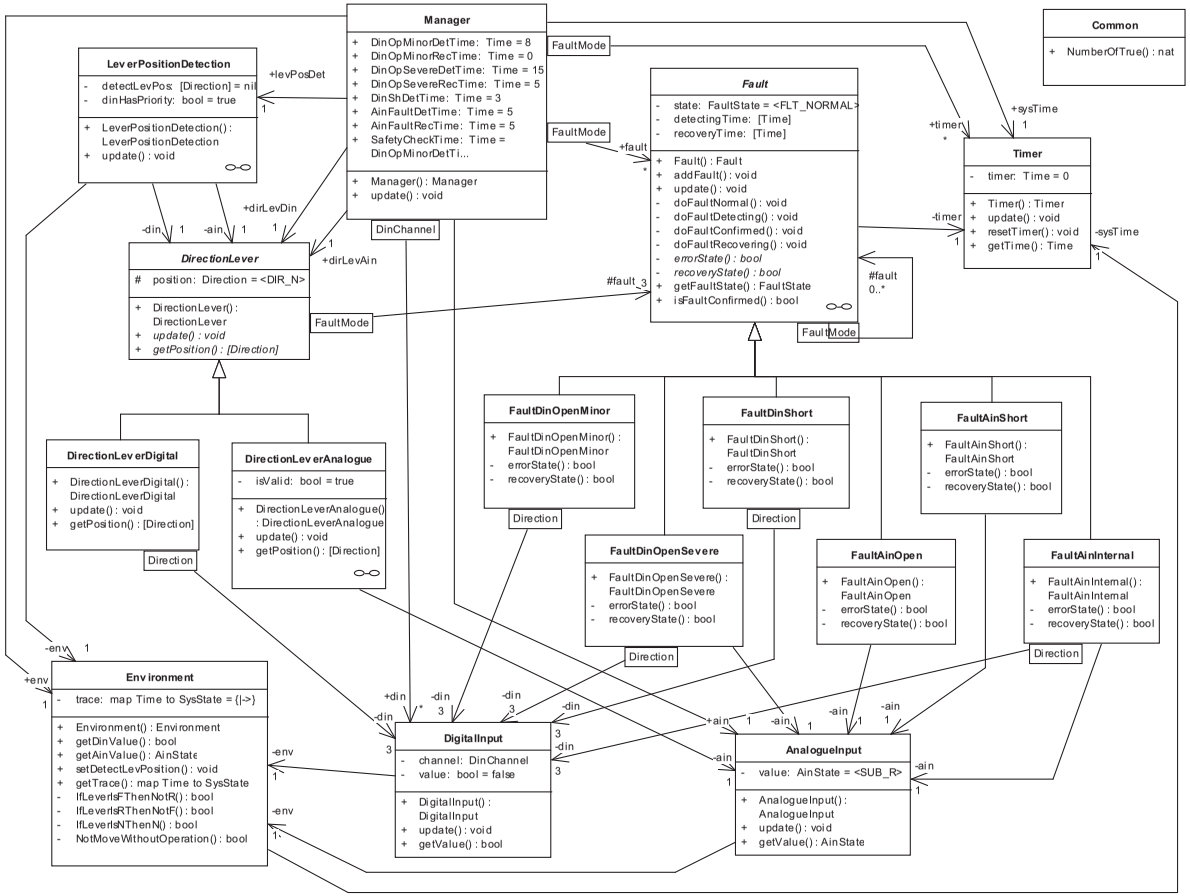

Fig. 4. Overview of the Model

3.1 Periodic Execution Architecture

The actual transmission controller is a periodic real-time system with a certain period, that is, a specific program is executed repeatedly every time unit. In order to reflect such a mechanism into the model, we introduce a periodic execution architecture, referred to as "time-triggered object-oriented model" in [6]. In this architecture, each class has a method `update`, in which its attributes are updated.[6] The `Manager` class, which controls the model execution, calls the `update` method of each class in a specified order as it increments the system timer by one time unit. That is, each class is updated once per time unit.

3.2 Fault Framework

In the model, we abstract the notion of faults (e.g. open- or short-circuit etc.) as a class containing a state represented by the state transition diagram of Fig. 5. The figure says:

1. As long as the device is normally working, the state stays in `NORMAL`.
2. If the error state holds, the state goes into `DETECTING`.
3. If the error state continues for a specified time (`detectingTime`), the fault is confirmed (`CONFIRMED`). If the error state no longer holds while in `DETECTING`, the state goes back to `NORMAL`.
4. If the recovery state holds while in `CONFIRMED`, the state goes into `RECOVERING`.
5. If the recovery state continues for a specified time (`recoveryTime`), the fault recovers (`NORMAL`). If the recovery state no longer holds while in `RECOVERING`, the state goes back to `CONFIRMED`.

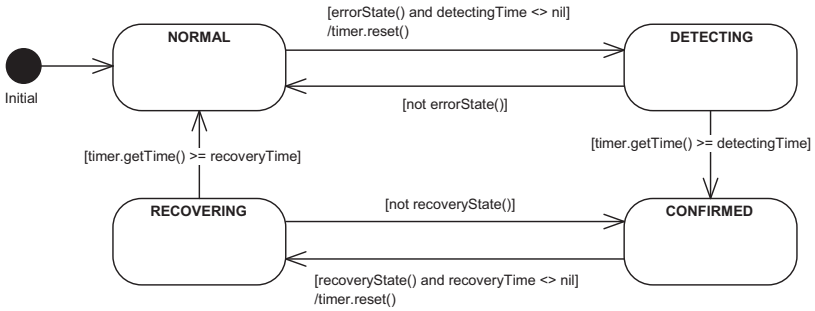

Fig. 5. State Transition Diagram of Fault

[6] The method corresponds to a `Step` in a VDM-RT context [3, 4]. But we use the name `update` in this paper according to [6] and our convention.

3.3 Detailed Description of Each Class

Common: Types and a function commonly used by various classes are defined in this class. It is inherited by all the other classes to make the model description simple, though in Fig. 4, inheritance arrows are hidden for legibility. Some of principal types are illustrated here.

```
types
public Time = nat;

public Direction = <DIR_F> | <DIR_N> | <DIR_R>;

public AinState = Direction | <SUB_R>  | <MID_RN>
                            | <MID_FN> | <SUPER_F>;
```

`Time` is defined as natural number (`nat`), denoting discrete time steps in the control system. The notion of direction (F, N and R) is defined as a union type of three quote types. We abstract the analogue input signal using a union type `AinState` instead of expressing it in voltage as follows:

$$
\begin{array}{ll}
\texttt{<SUB\_R>}: & A_{\mathrm{in}} < v_{\mathrm{R1}} \\
\texttt{<DIR\_R>}: & v_{\mathrm{R1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{R2}} \\
\texttt{<MID\_RN>}: & v_{\mathrm{R2}} < A_{\mathrm{in}} < v_{\mathrm{N1}} \\
\texttt{<DIR\_N>}: & v_{\mathrm{N1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{N2}} \\
\texttt{<MID\_FN>}: & v_{\mathrm{N2}} < A_{\mathrm{in}} < v_{\mathrm{F1}} \\
\texttt{<DIR\_F>}: & v_{\mathrm{F1}} \leq A_{\mathrm{in}} \leq v_{\mathrm{F2}} \\
\texttt{<SUPER\_F>}: & v_{\mathrm{F2}} < A_{\mathrm{in}}
\end{array}
$$

Manager: This class controls the overall model. It is responsible for constructing all instances of the model and executing them. All instances are created in the constructor `Manager`, and the operation `update` calls the same operations of all the other classes in a specified order (usually from lower to upper level classes). Specific values of fault and recovery detecting times, which are used to instantiate each fault class, are defined in this class. Note that they do not express the actual values of the control system. They are properly chosen to make the model feasible in practicable time and to simplify creation of test data. In our case, magnitude relation between the values is significant, and the value itself is not.

Timer: A generic timer class containing one instance variable `timer`, which is incremented by one step time in the operation `update`. The operation `resetTimer` sets the `timer` to zero. Instances of the class are used as a system timer, which denotes time evolution in the model, and a timer for each fault class, which is used to detect the fault.

Environment: As advocated in [3, 4], the components outside the controller are modelled as a class `Environment`. It provides input to the controller, i.e. the digital input signals F, N, R and the analogue input signal, and receives the detected lever

position from the controller as output of the system. It also has the actual lever position used for safety requirement check. This information is put into one record type `SysState`, and `trace`, a mapping from `Time` to `SysState`, is defined as an instance variable of the class, indicating time series of input and output data of the system.

```
types
public SysState :: dinF : bool
                   dinN : bool
                   dinR : bool
                   ain : AinState
                   levPos : LeverPosition
                   detectLevPos : [Direction];

instance variables
private trace : map Time to SysState := {|->};
```

The operations `getDinValue` and `getAinValue` respectively return the digital and the analogue input value at the current system time, and another operation `setDetectLevPosition` is called to set the detected lever position to `trace`.

DigitalInput and AnalogueInput: These two classes represent input channels of the controller, serving as interface to `Environment`. In the operation `update`, each class gets its current value from `Environment` and stores it in its instance variable `value`, which is used by the other upper level classes, that is, `Fault` and `DirectionLever` (described below). Three instances of the `DigitalInput` class, namely the digital input F, N and R, are created by the `Manager` class.

Fault: A superclass of the following six subclasses. It implements the state transition of fault described in Sect. 3.2, and provides the other classes with the state information. Fault detecting time and recovery time are respectively declared as an instance variable of optional type `[Time]`. The value `nil` means that the fault is undetectable or unrecoverable, respectively. The operations `errorState` and `recoveryState` are implemented in each subclass representing each fault mode (F1 to F6 in Sect. 2.1), because they are different depending on the fault modes. Note that these operations are declared as abstract methods.

- **FaultDinOpenMinor:**
  Digital input: open-circuit or short-circuit to ground (F1)
- **FaultDinOpenSevere:**
  Digital input: open-circuit or short-circuit to ground (F2)
- **FaultDinShort:**
  Digital input: short-circuit to power (F3)
- **FaultAinOpen:**
  Analogue input: open-circuit or short-circuit to ground (F4)

- **FaultAinShort:**
  Analogue input: short-circuit to power (F5)
- **FaultAinInternal:**
  Analogue input: internal circuit fault (F6)

DirectionLever: A superclass of the following two subclasses. In the `update` operation (implemented in the subclass), it updates its instance variable `position`, which denotes the position of the direction lever detected by the digital or the analogue input depending on its subclass. Users of this class do not need to take into account by which input the position is detected.

- **DirectionLeverDigital:**
  This class detects the direction lever position using the digital input and its fault information according to the control specifications described in Sect. 2.1. The operation `getPosition` returns `nil` if at least one digital input fault is confirmed, otherwise it returns the detected lever position.
- **DirectionLeverAnalogue:**
  This class detects the direction lever position using the analogue input and its fault information according to the control specifications described in Sect. 2.1. It has an instance variable `isValid`, which is used to realise the measures after fault recovery of F4 and F5. Briefly speaking, the variable is set to `false` if an analogue input fault is confirmed, and held `false` unless the normal N position is detected after fault recovery. As long as the variable is `false`, the lever position is regarded as N.

LeverPositionDetection: In the operation `update` in this class, the conclusive direction lever position is determined using the positions detected by the digital and the analogue input respectively, and the result is set to `Environment`. This class has an instance variable `dinHasPriority`, which is used to realise the measures after fault recovery of F1 and F2. The variable is set to `false` if a digital input fault is confirmed, and held `false` unless the detected lever positions by the digital and the analogue input are consistent with each other after fault recovery. As long as the variable is `false`, the lever position detected by the analogue input is valid.

3.4 Safety Requirements

Safety requirements are described in the `Environment` class because they require to access the lever position information enclosed in the class. Specifically, the requirements are formalised as postconditions of the operation `setDetectLevPosition`, which the `LeverPositionDetection` class calls to set the conclusive detected lever position to the `Environment` class. The postconditions are divided into four sub-operations (see Appendix A.4). We comment on the first and the last ones for illustration:

```
private IfLeverIsFThenNotR: () ==> bool
IfLeverIsFThenNotR() ==
```

```
  let curTime = sysTime.getTime()
  in
    return
    (((curTime >= Manager`SafetyCheckTime) and
      (forall t in set
          {curTime - Manager`SafetyCheckTime,..., curTime} &
           trace(t).levPos = <DIR_F>))
    => trace(curTime).detectLevPos <> <DIR_R>)
post RESULT;
...
private NotMoveWithoutOperation: () ==> bool
NotMoveWithoutOperation() ==
  let curTime = sysTime.getTime()
  in
    return
    (((curTime >= Manager`SafetyCheckTime) and
      (forall t in set
          {curTime - Manager`SafetyCheckTime,..., curTime-1} &
          (trace(t).levPos = trace(curTime).levPos and
           trace(t).detectLevPos = <DIR_N>)))
    => trace(curTime).detectLevPos = <DIR_N>)
post RESULT;
```

The operation `IfLeverIsFThenNotR` insists: if the lever has been set in the position F for a specified time, the detected lever position at the current time should not be R, which corresponds to the safety requirement R1 in Sect. 2.2. The operation `NotMoveWithoutOperation` insists: if the lever has not been manipulated and the detected lever position has been N for a specified time (until one step time before), the detected lever position at the current time should also be N, corresponding to the safety requirement R2.

4 Validation and Safety Analysis

In our approach, validation process is composed of two phases: unit testing and system testing. The former tests each class of the model, while the latter deals with the whole system. A testing framework VDMUnit [2, Chap. 9] is used for both testing phases. Various time series of input data for the `Environment` class (so called test scenarios) are elaborated. Using assert functions of VDMUnit, we check if a return value of each method of each class (in the case of unit testing) or the detected lever position (in the case of system testing) is consistent with the expected value at the time as we execute the model periodically. An example of the test cases is given below:

```
class SystemTest1 is subclass of TestCase, Environment

operations
...
```

```
public runTest : () ==> ()
runTest() ==
(
    let testInData = {t |-> testData(t).inData |
                       t in set dom testData}
    in (
        dcl mgr : Manager := new Manager(testInData);
        for t = 0 to (card dom testData - 1)
        do (
            mgr.update();
            assertTrue("t=" ^ VDMUtil`val2seq_of_char[nat](t) ^
                       ", failed.",
                       mgr.env.getTrace()(t).detectLevPos =
                       testData(t).expectVal)
        )
    )
);

types
private TestData :: inData : SysState
                    expectVal : [Direction];
values
-- time to (input data, expected value of trace(t).detectLevPos)
private testData: map Time to TestData =
{
   0 |-> mk_TestData(mk_SysState(false, true, false,
                      <DIR_N>, <DIR_N>, nil), <DIR_N>),
   1 |-> mk_TestData(mk_SysState(false, false, false,
                      <MID_FN>, <MID_FN_>, nil), <DIR_N>),
   2 |-> mk_TestData(mk_SysState(true, false, false,
                      <DIR_F>, <DIR_F>, nil), <DIR_F>),
...
};

end SystemTest1
```

The value `testData` denotes the time series of input data and expected values of the test scenario. We considered various scenarios: for example, normal lever operation without faults, a case in which open-circuit of the digital input signal F occurs and then it recovers, and so on. As for the system testing, we executed 14 test scenarios in total.

As a result, we have confirmed that the model behaved as expected for all the input data series elaborated. The test coverage information generated by Overture indicates that almost all statements of the model are tested, except for a part of the following two operations: `DirectionLeverDigital`update` (coverage is 98.6%) and `Fault`doFaultNormal` (89.4%). But these statements can never been executed under the current specifications of fault detection and the data settings. Therefore we conclude that virtually every part of the model is tested.

In the validation process, however, we realised that one of the safety requirements was not satisfied (a postcondition was violated) for certain input data series. This occurs in the following manner:

1. The direction lever is in the middle of the positions F and N, and no digital input signals are "on".
2. The analogue input signal indicates the position F (this meets the specifications of Table 1).
3. The digital input signal N periodically short-circuits to power with a period less than the fault detecting time, that is, the signal N alternates between "on" and "off" in a short period of time. The controller is not able to detect the fault (because the time in which the signal N remains "on" or "off" respectively is too short for the controller to detect the fault) and recognises the lever position as N.
4. In these situations, if the short-circuit of the digital input signal N recovers, that is, the signal N settles down to "off", the analogue input signal (recognised as F) becomes valid by a fault measure. This indicates that the detected lever position changes from N to F without manipulation by the operator, which violates the safety requirement R2 in Sect. 2.2.

However, the above case could never happen in reality because it is caused by nothing but a coincidence of several rare accidents. Nevertheless, it seems to be one of the advantages of formal modelling that the above phenomenon which could hardly be predicted in a manual fashion has been discovered.

5 Conclusion

In this paper, we have reported on a case study of applying a formal modelling technique to safety analysis of an embedded control system for construction equipment, and we have also presented a fault framework, which makes it possible to encapsulate a fault detection mechanism into the `Fault` class and separate it from the other control logics.

The validation of the model revealed that, under particular conditions, the exemplified system failed to satisfy certain safety requirement which had been considered to be satisfied, though it could rarely happen in reality. This demonstrates the advantage of the formal modelling and validation techniques.

The control system treated in this paper is only a part of the entire system. In future work, we will apply the technique described above to a larger scale system. On the other hand, in our test scenario based approach, the result considerably depends on the quality of the scenarios. It might be sheer luck that we discovered the violation of the safety requirements. We will challenge formal verification of the model with the help of another verification tool, e.g. UPPAAL, in order to investigate if there exists another case which violates the safety requirements.

Acknowledgements: The author would like to thank John Fitzgerald and Ken Pierce for fruitful discussions. The work has been supported by Komatsu Ltd. Especially, the author is grateful to Shuuki Akushichi and Yasunori Ohkura for their valuable comments on a draft.

References

1. Fitzgerald, J., Larsen, P.G.: Modelling Systems: Practical Tools and Techniques in Software Development, Second Edition, Cambridge University Press (2009)
2. Fitzgerald, J., Larsen, P.G., Mukherjee, P., Plat, N., Verhoef, M.: Validated Designs for Object-Oriented Systems, Springer, London (2005)
3. Larsen, P.G., Fitzgerald, J., Wolff, S.: Methods for the Development of Distributed Real-Time Embedded Systems using VDM, Int J Software Informatics, vol. 3, no. 2-3, pp. 305–341 (2009)
4. Larsen, P.G., Wolff, S., Battle, N., Fitzgerald, J., Pierce, K.: Development Process of Distributed Embedded Systems using VDM, Overture Technical Report Series, no. TR-006 (2010)
5. McDermott, R.E., Mikulak, R.J., Beauregard, M.R.: The Basics of FMEA, Productivity Press (1996)
6. Yokoyama, T., Naya, H., Narisawa, F., Kuragaki, S., Nagaura, W., Imai, T., Suzuki, S.: A Development Method of Time-Triggered Object-Oriented Software for Embedded Control Systems (in Japanese), IEICE Trans. D-I, vol. J84-D-I, no. 4, pp. 338–349 (2001)

A VDM++ Model for the Control System

A.1 The Common class

```
class Common

types
public Time = nat;

-- Digital input channel
public DinChannel = <CH_LEV_F>  -- Direction lever F
                  | <CH_LEV_N>  -- Direction lever N
                  | <CH_LEV_R>; -- Direction lever R

public Direction = <DIR_F> | <DIR_N> | <DIR_R>;

-- Analogue input state
public AinState = Direction
                | <SUB_R>    -- Lower than minimum
                | <MID_RN>   -- Middle between R & N
                | <MID_FN>   -- Middle between F & N
                | <SUPER_F>; -- Higher than maximum

-- Physical position of the direction lever
public LeverPosition =
                  Direction
                | <MID_RNR>  -- Middle between R & N with Din R
                | <MID_RN_>  -- Middle between R & N without Din
                | <MID_RNN>  -- Middle between R & N with Din N
                | <MID_FNN>  -- Middle between F & N with Din N
```

```
                | <MID_FN_>  -- Middle between F & N without Din
                | <MID_FNF>; -- Middle between F & N with Din F

public FaultMode = <DIN_OPEN_MINOR>
                 | <DIN_OPEN_SEVERE>
                 | <DIN_SHORT>
                 | <AIN_OPEN>
                 | <AIN_SHORT>
                 | <AIN_INTERNAL>;

functions
-- Count the number of 'true' in a sequence of boolean values
public NumberOfTrue : seq of bool -> nat
NumberOfTrue(args) ==
    len [args(i) | i in set inds args & args(i)];

end Common
```

A.2 The Manager class

```
class Manager is subclass of Common

instance variables
-- Environment is declared as public
-- because it is referred to by test cases
public  env : Environment;
private sysTime : Timer;
private timer : map FaultMode to Timer;
private din : map DinChannel to DigitalInput;
private ain : AnalogueInput;
private fault : map FaultMode to Fault;
private dirLevDin : DirectionLever;
private dirLevAin : DirectionLever;
private levPosDet : LeverPositionDetection;

values
-- Detecting or recovery time of faults
-- Declared as public because they are referred to by test cases
public DinOpMinorDetTime:  Time = 8;
public DinOpMinorRecTime:  Time = 0;
public DinOpSevereDetTime: Time = 15;
public DinOpSevereRecTime: Time = 5;
public DinShDetTime:       Time = 3;
public AinFaultDetTime:    Time = 5;
public AinFaultRecTime:    Time = 5;
```

```
-- Time for safety requirements
public SafetyCheckTime: Time = (DinOpMinorDetTime +
                                AinFaultDetTime);

operations
public Manager : map Time to Environment`SysState ==> Manager
Manager(mTrace) ==
(
    -- Instantiate all objects
    sysTime := new Timer();
    timer := {fMode |-> new Timer() |
              fMode in set {<DIN_OPEN_MINOR>, <DIN_OPEN_SEVERE>,
                            <DIN_SHORT>, <AIN_OPEN>,
                            <AIN_SHORT>, <AIN_INTERNAL>}};
    env := new Environment(mTrace, sysTime);
    din := {ch |-> new DigitalInput(ch, env) |
            ch in set {<CH_LEV_F>, <CH_LEV_N>, <CH_LEV_R>}};
    ain := new AnalogueInput(env);
    let mapDin = {<DIR_F> |-> din(<CH_LEV_F>),
                  <DIR_N> |-> din(<CH_LEV_N>),
                  <DIR_R> |-> din(<CH_LEV_R>)}
    in
    (
        fault := {<DIN_OPEN_MINOR> |->
                        new FaultDinOpenMinor(
                                DinOpMinorDetTime,
                                DinOpMinorRecTime,
                                timer(<DIN_OPEN_MINOR>),
                                mapDin),
                  <DIN_OPEN_SEVERE> |->
                        new FaultDinOpenSevere(
                                DinOpSevereDetTime,
                                DinOpSevereRecTime,
                                timer(<DIN_OPEN_SEVERE>),
                                mapDin, ain),
                  <DIN_SHORT> |->
                        new FaultDinShort(
                                DinShDetTime,
                                nil,
                                timer(<DIN_SHORT>),
                                mapDin),
                  <AIN_OPEN> |->
                        new FaultAinOpen(
                                AinFaultDetTime,
                                AinFaultRecTime,
                                timer(<AIN_OPEN>),
                                ain),
                  <AIN_SHORT> |->
                        new FaultAinShort(
                                AinFaultDetTime,
```

```
                                AinFaultRecTime,
                                timer(<AIN_SHORT>),
                                ain),
                        <AIN_INTERNAL> |->
                             new FaultAinInternal(
                                AinFaultDetTime,
                                nil,
                                timer(<AIN_INTERNAL>),
                                mapDin, ain)};

        -- Add association from <AIN_INTERNAL> to <DIN_SHORT>
        fault(<AIN_INTERNAL>).addFault(
               {<DIN_SHORT> |-> fault(<DIN_SHORT>)});

        dirLevDin := new DirectionLeverDigital(
                            {fMode |-> fault(fMode) |
                             fMode in set {<DIN_OPEN_MINOR>,
                                           <DIN_OPEN_SEVERE>,
                                           <DIN_SHORT>}},
                            mapDin)
    );
    dirLevAin := new DirectionLeverAnalogue(
                            {fMode |-> fault(fMode) |
                             fMode in set {<AIN_OPEN>,
                                           <AIN_SHORT>,
                                           <AIN_INTERNAL>}},
                            ain);
    levPosDet := new LeverPositionDetection(
                            dirLevDin, dirLevAin, env);
);

public update : () ==> ()
update() ==
(
    for all x in set dom din do din(x).update();
    ain.update();
    for all x in set dom fault do fault(x).update();
    dirLevDin.update();
    dirLevAin.update();
    levPosDet.update();
    sysTime.update();
    for all x in set dom timer do timer(x).update();
);

end Manager
```

A.3 The Timer class

```
class Timer is subclass of Common

instance variables
private timer : Time := 0;

operations
public Timer : () ==> Timer
Timer() ==
    skip;

public update : () ==> ()
update() ==
    timer := timer + 1;

public resetTimer : () ==> ()
resetTimer() ==
    timer := 0;

public getTime : () ==> Time
getTime() ==
    return timer;

end Timer
```

A.4 The Environment class

```
class Environment is subclass of Common

types
public SysState :: dinF : bool     -- Digital input F
                   dinN : bool     -- Digital input N
                   dinR : bool     -- Digital input R
                   ain : AinState -- Analogue input
                   levPos : LeverPosition
                   -- Physical lever position
                   detectLevPos : [Direction];
                   -- Detected lever position

instance variables
-- Time series of input/output data
private trace : map Time to SysState := {|->};
private sysTime : Timer;
```

```
operations
public Environment : map Time to SysState * Timer
                     ==> Environment
Environment(inData, pTimer) ==
(
    trace := inData;
    sysTime := pTimer
);

public getDinValue : DinChannel ==> bool
getDinValue(ch) ==
(
    let currentTime = sysTime.getTime()
    in
        cases ch:
            <CH_LEV_F> -> return trace(currentTime).dinF,
            <CH_LEV_N> -> return trace(currentTime).dinN,
            <CH_LEV_R> -> return trace(currentTime).dinR
        end
)
pre
    sysTime.getTime() in set dom trace;

public getAinValue : () ==> AinState
getAinValue() ==
    return trace(sysTime.getTime()).ain
pre
    sysTime.getTime() in set dom trace;

public setDetectLevPosition : Direction ==> ()
setDetectLevPosition(dir) ==
    trace(sysTime.getTime()).detectLevPos := dir
pre
    sysTime.getTime() in set dom trace
post
    -- Safety requirements
    IfLeverIsFThenNotR() and
    IfLeverIsRThenNotF() and
    IfLeverIsNThenN() and
    NotMoveWithoutOperation();

public getTrace : () ==> map Time to SysState
getTrace() ==
    return trace;

-- Safety requirements
private IfLeverIsFThenNotR: () ==> bool
IfLeverIsFThenNotR() ==
    let curTime = sysTime.getTime()
    in
```

```
        return
        (((curTime >= Manager`SafetyCheckTime) and
          (forall t in set
             {curTime - Manager`SafetyCheckTime,..., curTime} &
              trace(t).levPos = <DIR_F>))
         => trace(curTime).detectLevPos <> <DIR_R>)
post RESULT;

private IfLeverIsRThenNotF: () ==> bool
IfLeverIsRThenNotF() ==
    let curTime = sysTime.getTime()
    in
        return
        (((curTime >= Manager`SafetyCheckTime) and
          (forall t in set
             {curTime - Manager`SafetyCheckTime,..., curTime} &
              trace(t).levPos = <DIR_R>))
         => trace(curTime).detectLevPos <> <DIR_F>)
post RESULT;

private IfLeverIsNThenN: () ==> bool
IfLeverIsNThenN() ==
    let curTime = sysTime.getTime()
    in
        return
        (((curTime >= Manager`SafetyCheckTime) and
          (forall t in set
             {curTime - Manager`SafetyCheckTime,..., curTime} &
              trace(t).levPos = <DIR_N>))
         => trace(curTime).detectLevPos = <DIR_N>)
post RESULT;

private NotMoveWithoutOperation: () ==> bool
NotMoveWithoutOperation() ==
    let curTime = sysTime.getTime()
    in
        return
        (((curTime >= Manager`SafetyCheckTime) and
          (forall t in set
            {curTime - Manager`SafetyCheckTime,..., curTime-1} &
            (trace(t).levPos = trace(curTime).levPos and
             trace(t).detectLevPos = <DIR_N>)))
         => trace(curTime).detectLevPos = <DIR_N>)
post RESULT;

end Environment
```

A.5 The DigitalInput class

```
class DigitalInput is subclass of Common

instance variables
private channel : DinChannel;
private value : bool := false;
private env : Environment;

operations
public DigitalInput : DinChannel * Environment ==> DigitalInput
DigitalInput(ch, pEnv) ==
(
    channel := ch;
    env := pEnv
);

public update : () ==> ()
update() ==
    value := env.getDinValue(channel);

public getValue : () ==> bool
getValue() ==
    return value;

end DigitalInput
```

A.6 The AnalogueInput class

```
class AnalogueInput is subclass of Common

instance variables
private value : AinState := <SUB_R>;
private env : Environment;

operations
public AnalogueInput : Environment ==> AnalogueInput
AnalogueInput(pEnv) ==
    env := pEnv;

public update : () ==> ()
update() ==
    value := env.getAinValue();

public getValue : () ==> AinState
```

```
getValue() ==
    return value;

end AnalogueInput
```

A.7 The Fault class

```
class Fault is subclass of Common

types
public FaultState = <FLT_NORMAL>
                  | <FLT_DETECTING>
                  | <FLT_CONFIRMED>
                  | <FLT_RECOVERING>;

instance variables
private state : FaultState := <FLT_NORMAL>;
private detectingTime : [Time];
        -- nil means the fault is undetectable
private recoveryTime : [Time];
        -- nil means the fault is unrecoverable
private timer : Timer;
protected fault : map FaultMode to Fault := {|->};

operations
public Fault : [Time] * [Time] * Timer ==> Fault
Fault(detT, recT, pTimer) ==
(
    detectingTime := detT;
    recoveryTime := recT;
    timer := pTimer
);

-- Add association to another Fault to watch
public addFault : map FaultMode to Fault ==> ()
addFault(mFault) ==
    fault := fault ++ mFault;

public update : () ==> ()
update() ==
    cases state:
        <FLT_NORMAL> -> doFaultNormal(),
        <FLT_DETECTING> -> doFaultDetecting(),
        <FLT_CONFIRMED> -> doFaultConfirmed(),
        <FLT_RECOVERING> -> doFaultRecovering()
    end;
```

```
private doFaultNormal : () ==> ()
doFaultNormal() ==
(
    if errorState() and detectingTime <> nil
    then
    (
        timer.resetTimer();
        if detectingTime = 0
        then
            state := <FLT_CONFIRMED>
        else
            state := <FLT_DETECTING>
    )
    else
        skip
)
pre
    state = <FLT_NORMAL>;

private doFaultDetecting : () ==> ()
doFaultDetecting() ==
(
    if not errorState()
    then
        state := <FLT_NORMAL>
    else if timer.getTime() >= detectingTime
    then
        state := <FLT_CONFIRMED>
    else
        skip
)
pre
    state = <FLT_DETECTING>;

private doFaultConfirmed : () ==> ()
doFaultConfirmed() ==
(
    if recoveryState() and recoveryTime <> nil
    then
    (
        timer.resetTimer();
        if recoveryTime = 0
        then
            state := <FLT_NORMAL>
        else
            state := <FLT_RECOVERING>
    )
    else
        skip
```

```
)
pre
    state = <FLT_CONFIRMED>;

private doFaultRecovering : () ==> ()
doFaultRecovering() ==
(
    if not recoveryState()
    then
        state := <FLT_CONFIRMED>
    else if timer.getTime() >= recoveryTime
    then
        state := <FLT_NORMAL>
    else
        skip
)
pre
    state = <FLT_RECOVERING>;

private errorState : () ==> bool
errorState() == is subclass responsibility;

private recoveryState : () ==> bool
recoveryState() == is subclass responsibility;

public getFaultState : () ==> FaultState
getFaultState() ==
    return state;

public isFaultConfirmed : () ==> bool
isFaultConfirmed() ==
    return ((state = <FLT_CONFIRMED>) or
            (state = <FLT_RECOVERING>));

end Fault
```

A.8 The FaultDinOpenMinor class

```
class FaultDinOpenMinor is subclass of Fault

instance variables
private din : map Direction to DigitalInput;

operations
public FaultDinOpenMinor : [Time] * [Time] * Timer *
                           map Direction to DigitalInput
```

```
                            ==> FaultDinOpenMinor
FaultDinOpenMinor(detT, recT, pTimer, mDin) ==
(
    din := mDin;
    Fault(detT, recT, pTimer)
)
pre
    dom mDin = {<DIR_F>, <DIR_N>, <DIR_R>};

private errorState : () ==> bool
errorState() ==
    return NumberOfTrue([din(<DIR_F>).getValue(),
                         din(<DIR_N>).getValue(),
                         din(<DIR_R>).getValue()]) = 0;

private recoveryState : () ==> bool
recoveryState() ==
    return NumberOfTrue([din(<DIR_F>).getValue(),
                         din(<DIR_N>).getValue(),
                         din(<DIR_R>).getValue()]) = 1;

end FaultDinOpenMinor
```

A.9 The FaultDinOpenSevere class

```
class FaultDinOpenSevere is subclass of Fault

instance variables

private din : map Direction to DigitalInput;
private ain : AnalogueInput;

operations

public FaultDinOpenSevere : [Time] * [Time] * Timer *
                            map Direction to DigitalInput *
                            AnalogueInput
                            ==> FaultDinOpenSevere
FaultDinOpenSevere(detT, recT, pTimer, mDin, pAin) ==
(
    din := mDin;
    ain := pAin;
    Fault(detT, recT, pTimer)
)
pre
    dom mDin = {<DIR_F>, <DIR_N>, <DIR_R>};
```

```
private errorState : () ==> bool
errorState() ==
    let ainValue = ain.getValue()
    in
    (
        return (NumberOfTrue([din(<DIR_F>).getValue(),
                              din(<DIR_N>).getValue(),
                              din(<DIR_R>).getValue()]) = 0
                and
                 (ainValue = <DIR_F> or
                  ainValue = <DIR_N> or
                  ainValue = <DIR_R>))
    );

private recoveryState : () ==> bool
recoveryState() ==
    return NumberOfTrue([din(<DIR_F>).getValue(),
                         din(<DIR_N>).getValue(),
                         din(<DIR_R>).getValue()]) = 1;

end FaultDinOpenSevere
```

A.10 The FaultDinShort class

```
class FaultDinShort is subclass of Fault

instance variables
private din : map Direction to DigitalInput;

operations
public FaultDinShort : [Time] * [Time] * Timer *
                       map Direction to DigitalInput
                       ==> FaultDinShort
FaultDinShort(detT, recT, pTimer, mDin) ==
(
    din := mDin;
    Fault(detT, recT, pTimer)
)
pre
    dom mDin = {<DIR_F>, <DIR_N>, <DIR_R>};

private errorState : () ==> bool
errorState() ==
    return NumberOfTrue([din(<DIR_F>).getValue(),
                         din(<DIR_N>).getValue(),
```

```
                        din(<DIR_R>).getValue()]) > 1;

private recoveryState : () ==> bool
recoveryState() ==
    return false;

end FaultDinShort
```

A.11 The FaultAinOpen class

```
class FaultAinOpen is subclass of Fault

instance variables
private ain : AnalogueInput;

operations
public FaultAinOpen : [Time] * [Time] * Timer * AnalogueInput
                      ==> FaultAinOpen
FaultAinOpen(detT, recT, pTimer, pAin) ==
(
    ain := pAin;
    Fault(detT, recT, pTimer)
);

private errorState : () ==> bool
errorState() ==
    return ain.getValue() = <SUB_R>;

private recoveryState : () ==> bool
recoveryState() ==
    return ain.getValue() <> <SUB_R>;

end FaultAinOpen
```

A.12 The FaultAinShort class

```
class FaultAinShort is subclass of Fault

instance variables
private ain : AnalogueInput;

operations
public FaultAinShort : [Time] * [Time] * Timer * AnalogueInput
```

```
                        ==> FaultAinShort
FaultAinShort(detT, recT, pTimer, pAin) ==
(
    ain := pAin;
    Fault(detT, recT, pTimer)
);

private errorState : () ==> bool
errorState() ==
    return ain.getValue() = <SUPER_F>;

private recoveryState : () ==> bool
recoveryState() ==
    return ain.getValue() <> <SUPER_F>;

end FaultAinShort
```

A.13 The FaultAinInternal class

```
class FaultAinInternal is subclass of Fault

instance variables
private din : map Direction to DigitalInput;
private ain : AnalogueInput;

operations
public FaultAinInternal : [Time] * [Time] * Timer *
                          map Direction to DigitalInput *
                          AnalogueInput
                          ==> FaultAinInternal
FaultAinInternal(detT, recT, pTimer, mDin, pAin) ==
(
    din := mDin;
    ain := pAin;
    Fault(detT, recT, pTimer)
)
pre
    dom mDin = {<DIR_F>, <DIR_N>, <DIR_R>};

private errorState : () ==> bool
errorState() ==
    let dinValueF = din(<DIR_F>).getValue(),
        dinValueN = din(<DIR_N>).getValue(),
        dinValueR = din(<DIR_R>).getValue(),
        ainValue = ain.getValue()
    in
```

```
    (
        return ((dinValueF and not dinValueN and
                 not dinValueR and
                 (ainValue = <DIR_R> or ainValue = <MID_RN>))
                or
                (not dinValueF and not dinValueN and
                 dinValueR and
                 (ainValue = <DIR_F> or ainValue = <MID_FN>))
                or
                (not dinValueF and dinValueN and
                 not dinValueR and
                 (ainValue = <DIR_F> or ainValue = <DIR_R>)))
               and not fault(<DIN_SHORT>).isFaultConfirmed()
    )
pre
    <DIN_SHORT> in set dom fault;

private recoveryState : () ==> bool
recoveryState() ==
    return false;

end FaultAinInternal
```

A.14 The DirectionLever class

```
class DirectionLever is subclass of Common

instance variables
protected position : Direction := <DIR_N>;
protected fault : map FaultMode to Fault;

operations
public DirectionLever : map FaultMode to Fault
                        ==> DirectionLever
DirectionLever(mFault) ==
    fault := mFault;

public update : () ==> ()
update() == is subclass responsibility;

public getPosition : () ==> [Direction]
getPosition() == is subclass responsibility;

end DirectionLever
```

A.15 The DirectionLeverDigital class

```
class DirectionLeverDigital is subclass of DirectionLever

instance variables
private din : map Direction to DigitalInput;

operations
public DirectionLeverDigital : map FaultMode to Fault *
                               map Direction to DigitalInput
                               ==> DirectionLeverDigital
DirectionLeverDigital(mFault, mDin) ==
(
    din := mDin;
    DirectionLever(mFault)
)
pre
    dom mFault = {<DIN_OPEN_MINOR>, <DIN_OPEN_SEVERE>,
                  <DIN_SHORT>} and
    dom mDin = {<DIR_F>, <DIR_N>, <DIR_R>};

public update : () ==> ()
update() ==
(
    -- Detect the lever position by the digital input.
    -- In case the input has a fault,
    -- the position is not updated.
    if fault(<DIN_OPEN_MINOR>).getFaultState() = <FLT_NORMAL>
       and
       fault(<DIN_OPEN_SEVERE>).getFaultState() = <FLT_NORMAL>
       and
       fault(<DIN_SHORT>).getFaultState() = <FLT_NORMAL>
    then
    (
        let dinValueF = din(<DIR_F>).getValue(),
            dinValueN = din(<DIR_N>).getValue(),
            dinValueR = din(<DIR_R>).getValue()
        in
        (
            if dinValueF and
               not dinValueN and
               not dinValueR
            then
                position := <DIR_F>
            else if not dinValueF and
                    dinValueN and
                    not dinValueR
            then
                position := <DIR_N>
```

```
            else if not dinValueF and
                    not dinValueN and
                    dinValueR
            then
                position := <DIR_R>
            else
                skip
        )
    )
    else
        skip;
);

-- Return nil if at least one digital input fault is confirmed,
-- otherwise return detected lever position
public getPosition : () ==> [Direction]
getPosition() ==
    if fault(<DIN_OPEN_MINOR>).isFaultConfirmed() or
       fault(<DIN_OPEN_SEVERE>).isFaultConfirmed() or
       fault(<DIN_SHORT>).isFaultConfirmed()
    then
        return nil
    else
        return position;

end DirectionLeverDigital
```

A.16 The DirectionLeverAnalogue class

```
class DirectionLeverAnalogue is subclass of DirectionLever

instance variables
private isValid : bool := true;
private ain : AnalogueInput;

operations
public DirectionLeverAnalogue : map FaultMode to Fault *
                                AnalogueInput
                                ==> DirectionLeverAnalogue
DirectionLeverAnalogue(mFault, pAin) ==
(
    ain := pAin;
    DirectionLever(mFault)
)
pre
    dom mFault = {<AIN_OPEN>, <AIN_SHORT>, <AIN_INTERNAL>};
```

```
public update : () ==> ()
update() ==
(
    let ainValue = ain.getValue()
    in
    (
        -- Check if the analogue input is valid or not.
        -- If an analogue input fault is confirmed,
        -- then set to invalid.
        -- If the normal N position is detected after
        -- fault recovery, then set to valid.
        if fault(<AIN_OPEN>).isFaultConfirmed() or
           fault(<AIN_SHORT>).isFaultConfirmed() or
           fault(<AIN_INTERNAL>).isFaultConfirmed()
        then
            isValid := false
        else if ainValue = <DIR_N>
        then
            isValid := true
        else
            skip;

        -- Detect the lever position by the analogue input
        -- including fault measures
        if fault(<AIN_OPEN>).getFaultState() = <FLT_NORMAL>
           and
           fault(<AIN_SHORT>).getFaultState() = <FLT_NORMAL>
           and
           fault(<AIN_INTERNAL>).getFaultState() = <FLT_NORMAL>
           and isValid
        then
        (
            if ainValue = <DIR_F> or ainValue = <DIR_R>
            then
                position := ainValue
            else
                position := <DIR_N>
        )
        else
            position := <DIR_N>
    )
);

public getPosition : () ==> [Direction]
getPosition() ==
    return position;

end DirectionLeverAnalogue
```

A.17 The LeverPositionDetection class

```
class LeverPositionDetection is subclass of Common

instance variables
private detectLevPos : [Direction] := nil;
private dinHasPriority : bool := true;
private din : DirectionLever;
private ain : DirectionLever;
private env : Environment;

operations
public LeverPositionDetection : DirectionLever *
                                DirectionLever *
                                Environment
                                ==> LeverPositionDetection
LeverPositionDetection(pDin, pAin, pEnv) ==
(
    din := pDin;
    ain := pAin;
    env := pEnv
);

public update : () ==> ()
update() ==
(
    let dinPosition = din.getPosition(),
        ainPosition = ain.getPosition()
    in
    (
        -- Check which input has priority.
        -- If a digital input fault is confirmed,
        -- then analogue is valid.
        -- After fault recovery, if the lever positions by
        -- digital and analogue are consistent,
        -- then digital is valid.
        if dinPosition = nil
        then
            dinHasPriority := false
        else if dinPosition = ainPosition
        then
            dinHasPriority := true
        else
            skip;

        -- Get the lever position from the prior input
        if dinHasPriority
        then
            detectLevPos := dinPosition
```

```
        else
            detectLevPos := ainPosition
    );

    -- Set the detected lever position to Environment
    env.setDetectLevPosition(detectLevPos)
);

end LeverPositionDetection
```