Yuliya Prokhorova | Elena Troubitsyna | Linas Laibinis | Dubravka Ilić | Timo Latvala

Formalisation of an Industrial Approach to Monitoring Critical Data

Turku Centre *for* Computer Science

TUCS Technical Report
No 1070, March 2013

Formalisation of an Industrial Approach to Monitoring Critical Data

Yuliya Prokhorova
TUCS – Turku Centre for Computer Science,
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
yuliya.prokhorova@abo.fi

Elena Troubitsyna
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
elena.troubitsyna@abo.fi

Linas Laibinis
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, 20520 Turku, Finland
linas.laibinis@abo.fi

Dubravka Ilić
Space Systems Finland
Kappelitie 6 B, 02200 Espoo, Finland
dubravka.ilic@ssf.fi

Timo Latvala
Space Systems Finland
Kappelitie 6 B, 02200 Espoo, Finland
timo.latvala@ssf.fi

TUCS Technical Report

No 1070, March 2013

Abstract

A large class of safety-critical control systems contains monitoring subsystems that display certain system parameters to (human) operators. Ensuring that the displayed data are sufficiently fresh and non-corrupted constitutes an important part of safety requirements. However, the monitoring subsystems are typically not a part of a safety kernel and hence often built of SIL1 and SIL2 components. In this paper, we formalise a recently implemented industrial approach to architecting dependable monitoring systems that ensures data freshness and integrity despite unreliability of their components. Moreover, we derive an architectural pattern that allows us to formally reason about data freshness and integrity. The proposed approach is illustrated by an industrial case study.

Keywords: Fault-tolerance, formal modelling, Event-B, data freshness, data integrity.

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

Data Monitoring Systems (DMSs) are typical for a wide range of safety-critical applications, spanning from nuclear power plant control rooms to individual healthcare devices. Data monitoring is usually not a part of the system safety kernel and hence DMSs are often developed using methods prescribed for SIL1 or SIL2 systems. However, data monitoring might have serious *indirect* safety implications. Indeed, based on the displayed data the operator should take appropriate and timely decisions. Therefore, we have to guarantee that a DMS outputs data that are sufficiently fresh and non-corrupted.

One possible solution would be to build a DMS from highly reliable components and formally verify its correctness. However, such a solution would be rather cost-inefficient. Instead, another practical solution has been recently proposed in the industrial setting[1]. The solution is based on building a networked DMS over (potentially unreliable) components and utilising diversity and redundancy to guarantee dependability of a DMS.

In this paper, we aim at giving a formal justification for the proposed industrial solution. We formally define the generic architecture of a networked DMS, formalise the expected data freshness and integrity properties, and derive the constraints that a DMS should satisfy to guarantee them. We use the Event-B [1] formalism and the associated RODIN platform [2] to formally specify the system architecture and its properties. The proposed specification can be seen as a pattern for designing a networked DMS. We believe that the presented work not only defines a formal basis for constructing a dependable DMS but also gives a good demonstration of how formal modelling can facilitate validation of an industrial solution.

The paper is organised as follows. In Section 2, we define a generic architecture of a DMS. In Section 3, we briefly introduce our modelling framework – Event-B. In Section 4, we derive a generic specification of a DMS and formally define the data freshness and integrity properties. In Section 5, we overview the industrial case study and lessons learnt. Finally, in Section 6, we discuss the proposed approach and the related work.

2 Industrial Solution to Monitoring Critical Data

In this section, we present a generalised version of the proposed industrial solution to data monitoring. The main purpose of the system is to display a certain system parameter (e.g., temperature, pressure, etc.). We start by defining a generic system architecture.

2.1 Overview of a Distributed DMS Architecture

The monitored parameter is measured by sensors. Each sensor is associated with the corresponding data processing unit (DPU) that periodically reads sensor data. The proposed industrial solution is to build a networked DMS to achieve reliable monitoring of data. The networked DMS contains two types of DPUs – the ones that are directly connected to the sensors and the others that are not. Both types of DPUs output data to the displays connected

[1] We omit a reference to the actual product due to confidentiality reasons.

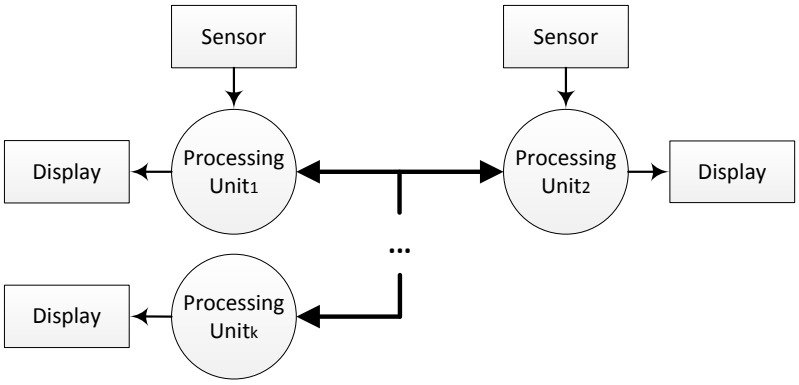


Figure 1: Distributed monitoring system

to them, i.e., the operator observes several versions of data (typically up to four). A generic architecture of the system is shown in Fig. 1.

A network built over DPUs allows them to communicate with each other. The DPUs that are connected to sensors periodically poll sensor data, process data received from sensors and other DPUs, and output the result to the display as well as broadcast the own processed data over the network. The DPUs that are not connected to sensors perform the same steps, except reading and processing sensor data. The DPUs run different versions of software, i.e., rely on software diversity to avoid common errors. The main goal of the system is to guarantee that each DPU displays only the data that are sufficiently fresh and non-corrupted. If a DPU cannot satisfy these properties, it should output a special predefined error value.

2.2 Data Freshness and Integrity

Let us now discuss the mechanism of achieving data freshness and integrity. Each DPU has a data pool. In this pool the DPU records the processed sensor data (if the DPU is connected to a sensor) as well as the data received from the other units. The DPU puts in the pool only the data that *have been checked to be non-erroneous*. For sensor data, this means that the obtained sensor reading has passed the reasonableness check and the sensor data processing has completed successfully, i.e., no failure flag was raised. For the data received from the other units, the check of their attached checksums has to be successful and the received data packet should not contain an error message.

Each data processing unit has its own local clock. (The system periodically sends a special clock adjusting signal to each DPU to prevent an unbounded local clock drift.) All the data that are processed by the system are timestamped. Each unit timestamps every data that it processes based on its local clock. To ensure freshness of the displayed data, before displaying data, the DPU analyses its data pool and filters out the data that are not fresh enough. To select or calculate the data item to be displayed, the DPU applies a predefined

function (e.g., maximum) to the set of fresh pool data.

The data are considered to be fresh if the difference between the current (local) DPU time and the data timestamp is less than δ time units. Globally, the freshness property can be formulated as follows: the displayed data are considered fresh if their timestamp differs by no more that $\delta + \epsilon$ time units from the imaginary global clock, where ϵ is the upper bound of the local clock drift.

Data freshness and correctness depend on several factors. If the DPU is connected to a sensor, processing sensor data might take excessive time (e.g., due to a software error) and hence the DPU's own data might not be fresh anymore. Due to network delays or slow processing in other DPUs, the received data might be old as well. Moreover, software errors might corrupt the DPU's own data. A received data packet might also get corrupted during transmission. However, despite a potentially large number of various faults, an occurrence of the system failure, making all DPUs to display an error message, is rather unlikely. Our modelling formally defines the link between data freshness and data integrity that allows us to validate this claim.

In the next section, we present our formal modelling framework – Event-B, while in Section 4 we demonstrate how to apply it to model a networked DMS.

3 Overview of Event-B

Event-B [1, 2] is a state-based formalism for system level modelling and verification. It is an extension of the B Method [3] that aims at facilitating modelling of parallel, distributed and reactive systems.

In Event-B, system models are defined using the notion of an *abstract state machine*. An abstract machine encapsulates the state (the variables) of a model and defines operations (events) on its state. Each machine is uniquely identified by its name *MachineName*. The state variables of the machine are declared in the *Variables* clause and initialised in the *INITIALISATION* event. The variables are strongly typed by the constraining predicates given in the *Invariants* clause. The data types and constants of the model are defined in *CONTEXT* that also postulated their properties as axioms. The behaviour of the system is determined by a number of atomic *EVENTS*. An event can be defined as follows:

$$evt \mathrel{\hat{=}} \textbf{any } lv \textbf{ where } g \textbf{ then } R \textbf{ end}$$

where lv is a list of local variables, the guard g is the conjunction of predicates defined over the model variables, and the action R is a parallel composition of assignments over the variables.

The guard defines when an 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 assignments executed simultaneously. Variable assignments can be either deterministic or non-deterministic. The deterministic assignment is denoted as $x := E(v)$, where x is a state variable and $E(v)$ is an expression over the state variables v. The non-deterministic

assignment can be denoted as $x :\in S$ or $x :| Q(v, x')$, 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.

Event-B enables development of systems correct-by-construction. It allows the designers to create and verify formal specifications of complex industrial-scale systems without encountering state explosion problem. Event-B relies on the top-down refinement-based approach to formal development. The development starts from an abstract specification of the system that defines essential behaviour and properties of the system. In a number of correctness-preserving transformations, refinements, we introduce implementation details and arrive at the detailed system specification closely resembling an eventual implementation. Usually refinement steps result in introducing new variables and events into the model. We can also perform data refinement that allowing us to replace some abstract variables of the model with their concrete counterparts. In this case, the invariant of a refined model formally defines the relationship between the abstract and concrete variables.

Event-B relies on theorem proving to verify correctness. Via discharging proof obligations we formally verify that the events preserve the invariant, the model is well-formed and refinement does not introduce additional deadlocks. The detailed discussion of proof obligations can be found in [1]. The Rodin platform [2] provides an integrated modelling environment that among others supports automatic generation and proving of proof obligation. It also provides facilities for interactive proving. In general the Rodin-platform achieves a high degree of automation – usually over 80% of proof obligations are discharged automatically. In the next section, we present our approach to modelling DMSs in Event-B.

4 Formal Generic Development of Distributed Monitoring Systems in Event-B

Let us observe that the generic architecture of DMS described in Section 2 is a composition of loosely coupled asynchronous components. Indeed, each DPU has its own display and relies not only on the data received asynchronously from the other DPUs but also on its own data to produce the displayed data. The system is modular and behaviour of its modules, DPUs, follows the same generic pattern. Therefore, to reason about the overall system, it is sufficient to model the behaviour of its single module and define its interactions with the other modules as a part of the environment specification. One of the obvious benefits of such an approach is clear reduction of the model complexity.

4.1 Abstract Model

Next we will present an Event-B development of a DPU – a generic module of a DMS. Since it is based on the generic architecture of the system discussed above, the presented development is also generic and thus can be instantiated to accommodate for specific details of a concrete monitoring system.

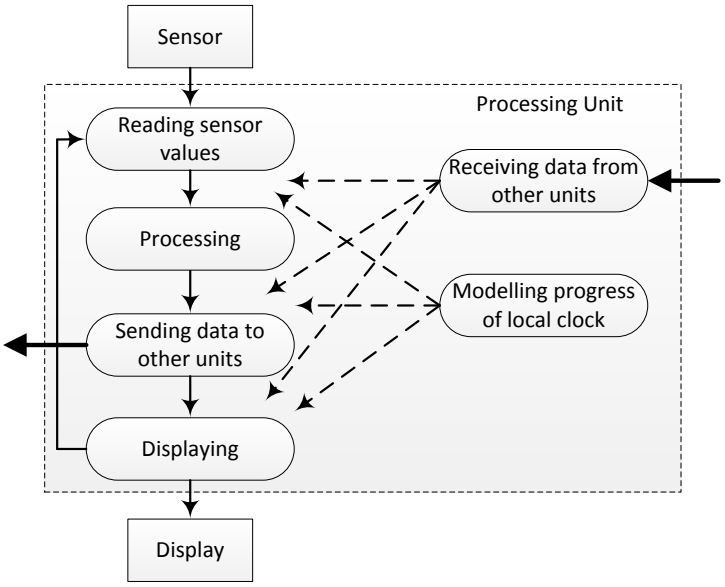


Figure 2: Dynamics of a DPU

We employ the following refinement strategy. The initial abstract specification formally describes the essential functional behaviour of DPU. Nevertheless, this allows us to formulate (as model invariants) and verify the desired freshness and correctness properties for the displayed data. The next model (first refinement) introduces fault-tolerance mechanisms and allows us to formulate and prove the required data integrity properties. Finally, the second refinement step deals with the local clock adjustment.

Essentially, the behaviour of a DPU is cyclic. At each cycle, it reads and processes sensor data, broadcasts the processed data to the other DPUs, possibly receives data from the other units, and finally produces the value to display. These activities are modelled by the events *Environment*, *Processing*, *Sending_Packet*, and *Displaying* respectively. The event *Receiving_Packets* models interaction with the environment – asynchronous receiving of data packets from the other DPUs. The event *Time_Progress* models a progress of the local clock and is also executed asynchronously. The dynamic behaviour of DPU is graphically presented in Fig. 2. The solid lines show the passage of control between the cyclically executed events. Enabledness of asynchronous events is depicted with the dashed lines. The overall structure of the initial specification – the **machine** DPU – is shown in Fig. 3, while Fig. 4 presents the specifications of the main events.

In the model, the variable *main_phase* stores the current phase of DPU execution. The type of *main_phase* is defined the enumerated set *MAIN_PHASES* of elements $\{$*ENV, PROC, DISP*$\}$. Here, the *ENV* phase stands for environment (sensor readings), *PROC* – for data processing, and *DISP* – for data displaying. Broadcasting data to the other units is modelled as a part of the *DISP* phase.

machine DPU

variables main_phase, monitored_value, processed_value, timestamp, displayed_value, curr_time, time_progressed, packet_sent_flag

invariants

main_phase ∈ MAIN_PHASES	*// phases of the unit cyclic behaviour*
monitored_value ∈ ℕ	*// raw sensor readings*
processed_value ∈ 0 .. UNIT_NUM → MIN_VAL .. MAX_VAL	*// collected processed data from all DPUs*
timestamp ∈ 0 .. UNIT_NUM → ℕ	*// collected timestamp values from all DPUs*
displayed_value ∈ ℕ	*// the output data*
curr_time ∈ ℕ	*// the current value of the local unit clock*
time_progressed ∈ BOOL	*// the flag to determine time progress*
packet_sent_flag ∈ BOOL	*// the flag to determine sending of a packet*
// Freshness1 ∧ Freshness2 ∧ Correctness	

events

INITIALISATION	*// initialising variables*
Environment	*// reading sensor values*
Processing	*// processing sensor data*
Sending_Packet	*// broadcasting data packet to other DPUs*
Displaying	*// outputting data to a display*
Receiving_Packets	*// receiving packets from other DPUs*
Time_Progress	*// modelling progress of local clock*

end

Figure 3: Outline of the abstract specification

The *Environment* event models sensor reading. As a result, it updates the variable *monitored_value*. As a part of the environment action, we also model a possible adjustment (synchronisation) of the local clock, the value of which is stored in the variable $curr\_time$.

As shown in Fig. 4, the *Processing* event specifies a conversion of the sensor data. We use the abstract function $Convert$ to model generic conversion process. The result of the conversion is then used to update the DPU data pool. Implicitly, the event also models a possibility of conversion failure. In this case, the corresponding data pool value remains unchanged (i.e., the last good value is used instead).

To avoid unnecessary complex data structures, we represent the DPU's data pool by two array variables – $processed\_value$ and $timestamp$. For each $i \in 0..$*UNIT_NUM*, the data item $processed\_value(i)$ contains the data produced or received from the DPU$_i$, while $timestamp(i)$ contains the corresponding data timestamp. Here the abstract constant *UNIT_NUM* stands for the maximal index value of these arrays (i.e., the number of the DPUs of the system). The value of *UNIT_NUM* can vary for different DMSs. Another generic constants, *MIN_VAL* and *MAX_VAL*, specify the minimal and maximal valid values for the processed measurements respectively.

The *Sending_Packet* event models broadcasting the processed DPU data as data packets to the other DPUs. Each DPU cycle finishes with the execution of the *Displaying* event that calculates the value to be displayed. First it filters the data pool for fresh data and then applies the abstract function *Output_Fun* on the filtered data to produce the DPU output value to be displayed. If there are no fresh data in the pool, a pre-defined error value (modelled by the abstract constant *ERR_VAL*) is displayed.

event Receiving_Packets
  any *p*
  where
    p ∈ PACKET
    packet_time(*p*) > timestamp(packet_unit_id(*p*))
    packet_data(p) ∈ MIN_VAL .. MAX_VAL
    main_phase ≠ ENV
    time_progressed = TRUE
  then
    time_progressed ≔ FALSE
    timestamp(packet_unit_id(*p*)) ≔ packet_time(*p*)
    processed_value(packet_unit_id(*p*)) ≔ packet_data(*p*)
end

event Sending_Packet
  any *p*
  where
    main_phase = DISP
    time_progressed = TRUE
    packet_sent_flag = FALSE
    p ∈ PACKET
    packet_unit_id(*p*) = 0
    packet_time(*p*) = curr_time
    packet_data(*p*) = Convert(monitored_value)
  then
    time_progressed ≔ FALSE
    packet_sent_flag ≔ TRUE
end

event Processing
  where
    main_phase = PROC
    time_progressed = TRUE
  then
    main_phase ≔ DISP
    time_progressed ≔ FALSE
    timestamp, processed_value :|
      timestamp' ∈ 0 .. UNIT_NUM → ℕ ∧
      processed_value' ∈ 0 .. UNIT_NUM → MIN_VAL .. MAX_VAL ∧
      ((timestamp'(0) = curr_time ∧
        processed_value'(0) = Convert(monitored_value)) ∨
       (timestamp'(0) = timestamp(0) ∧
        processed_value'(0) = processed_value(0)))
end

event Displaying
  any *ss, DATA_SET*
  where
    main_phase = DISP
    time_progressed = TRUE
    packet_sent_flag = TRUE
    DATA_SET ⊆ ℕ
    ss = {x↦y | ∃i · i ∈ dom(timestamp) ∧ x = timestamp(i) ∧
          y = processed_value(i)}
          [curr_time−Fresh_Delta .. curr_time]
    (*ss* ≠ ∅ ⇒ *DATA_SET* = *ss*)
    (*ss* = ∅ ⇒ *DATA_SET* = {ERR_VAL})
  then
    main_phase ≔ ENV
    time_progressed ≔ FALSE
    packet_sent_flag ≔ FALSE
    displayed_value ≔ Output_Fun(*DATA_SET*)
end

Figure 4: Events of the abstract model

Obviously, to reason about data freshness, we should model progress of time. The event *Time_Progress* forcefully alternates between any cyclic events of the model and changes the value of the variable *curr_time* modelling the local clock. Event alternation is enforced by using the boolean variable *time_progressed*.

Finally, let us discuss communication between DPUs. It is organised via sending and receiving packets of data. At this level of abstraction, we assume that each packet includes the following fields: (1) an id (i.e., the identification number) of DPU that sent the packet; (2) a timestamp, indicating when the packet was sent; (3) the actual data. We further elaborate of the packet structure, i.e., extend it with new fields, at the next refinement steps.

To access the packet fields, the introduce the following abstract functions:

$$packet\_unit\_id \in PACKET \to 0\,..\,UNIT\_NUM,$$
$$packet\_time \in PACKET \to \mathbb{N},$$
$$packet\_data \in PACKET \to MIN\_VAL\,..\,MAX\_VAL.$$

They allow us to extract the corresponding packet fields. The incoming packets are modelled as parameters of the event *Receiving_Packets*. The extractor functions are then used to decompose these packets. As a result of the event, the pool values $processed\_value(j)$ and $timestamp(j)$ may get updated, where j is the index of the DPU that sent the data. However, the update occurs only if the received data are fresher than the previously stored values and the packet did not contain an error flag.

Note that the outgoing packets are constructed (using the same functions to enforce the correctness of the contained information) as the local variables of the event *Sending_Packet*.

Our modelling allows us to formally define and verify the data freshness and integrity properties. We define them as model invariants as follows:

Freshness 1: $main\_phase = ENV \ \wedge \ displayed\_value = ERR\_VAL \Rightarrow$
$(\forall i \cdot i \in dom(timestamp) \Rightarrow$
$timestamp(i) \notin curr\_time - Fresh\_Delta \mathinner{..} curr\_time)$

Freshness 2: $main\_phase = ENV \ \wedge \ displayed\_value \neq ERR\_VAL \Rightarrow$
$\neg\{x \mid \exists j \cdot j \in dom(timestamp) \ \wedge \ x = timestamp(j)\} \cap$
$curr\_time - Fresh\_Delta \mathinner{..} curr\_time = \varnothing$

Correctness: $main\_phase = ENV \ \wedge \ displayed\_value \neq ERR\_VAL \Rightarrow$
$displayed\_value = Output\_Fun(\{x \mapsto y \mid$
$\exists i \cdot i \in dom(timestamp) \ \wedge x = timestamp(i) \wedge$
$y = processed\_value(i)\}[curr\_time - Fresh\_Delta \mathinner{..} curr\_time])$

where *dom* and [...] are respectively the relational domain and image operators, while *Fresh_Delta* is the pre-defined constant standing for the maximum time offset while the data is still considered to be fresh.

The first invariant states that the unit displays the pre-defined error value only when there are no fresh data produced by at least one unit. The second invariant formulates the opposite case, i.e., it requires that, if some data other than the pre-defined error value are displayed, they are based on the fresh data from at least one unit. The third invariant formulates the correctness of the displayed data – these data are always calculated by applying the pre-defined function (modelled by *Output_Fun*) to the filtered fresh data from the unit data pool. The invariant properties are proved as a part of the model verification process.

4.2 Model Refinements

The first refinement. The aim of our first refinement step is to introduce modelling of failures. The result of the fault tree analysis performed for the considered DMS is given in Appendix A. We explicitly specify the effect of three types of failures: sensor failures, sensor data processing failures, and communication errors. If the DPU experiences sensor or sensor data processing failures, it does not update the value of its own data in the data pool. Similarly, if the DPU detects a communication error, it does not update the data of the

sending process in the data pool. We also abstractly model the presence of software faults, although do not introduce explicit mechanisms for diagnosing them. In all these cases, the mechanism for tolerating the faults is the same: the DPU neglects erroneous or corrupted data and relies on the last good values from the respective DPUs stored in the data pool to calculate the displayed value (provided it is still fresh at the moment of displaying).

To implement these mechanisms, first we extend the data packet structure with two new fields: the one containing the information about the status of the DPU that sent the packet, and the other one storing a checksum for determining whether the packet was corrupted during the transmission. The corresponding extractor functions are added to the model:

$$\begin{aligned} &\mathit{packet\_status} \in \mathit{PACKET} \to \mathit{STATUS}, \\ &\mathit{packet\_checksum} \in \mathit{PACKET} \to \mathbb{N}, \end{aligned}$$

where the set $STATUS$ consists of two subsets NO_FLT and FLT modelling the absence or presence of faults of a unit, respectively. To calculate a checksum, we define the function

$$\mathit{Checksum} \in \mathbb{N} \times \mathit{MIN\_VAL} \mathinner{..} \mathit{MAX\_VAL} \to \mathbb{N}.$$

The function takes as the input the transmitted timestamp and measurement data.

The communication between units is modelled by the event *Receiving_Packets* shown in Fig. 5. The event specifies a successful receiving of packets, i.e., when the sending DPU has succeeded in producing fresh data and the corresponding packet was not corrupted during the transmission. If it is not the case, the data pool of the receiving DPU is not getting updated, i.e., this behaviour corresponds to *skip*.

The detection of sensor faults is modelled by the new event *Pre_Processing*. An excerpt from the specification of this event, shown in Fig. 5, illustrates detection of the sensor fault *"Value is out of range"*. The event *Pre_Processing* also introduces an implicit modelling of the effect of software faults by non-deterministic update of the variable $unit\_status$.

In this refinement step, we split the abstract event *Processing* into two events: *Processing_OK* and *Processing_NOK*. The event *Processing_OK* models an update of the DPU's data pool with the new processed measurements, i.e., it is executed when no failure occurred. Correspondingly, the event *Processing_NOK* is executed when errors have been detected. In this case, the DPU relies on the last good value in its further computations.

The performed refinement step allows us to formulate the data integrity property as the following model invariants:

Integrity 1: $\forall j \cdot j \in 0 \mathinner{..} \mathit{UNIT\_NUM} \Rightarrow$
$\mathit{Checksum}(\mathit{timestamp}(j) \mapsto \mathit{processed\_value}(j)) = \mathit{checksum}(j)$

Integrity 2: $\forall j \cdot j \in 0 \mathinner{..} \mathit{UNIT\_NUM} \Rightarrow \mathit{status}(j) \in \mathit{NO\_FLT}$

These invariants guarantee that the displayed data are based only on the valid data stored in the DPU data pool. In other words, neither corrupted nor faulty data are taken into account to compute the data to be displayed.

event Pre_Processing
where
main_phase = PROC
time_progressed = TRUE
pre_proc_flag = TRUE
then
pre_proc_flag ≔ FALSE
sensor_fault :| sensor_fault' ∈ BOOL ∧
((monitored_value ≥ Sens_Lower_Threshold ∧
monitored_value ≤ Sens_Upper_Threshold) ⇒
sensor_fault' = FALSE) ∧
(¬(monitored_value ≥ Sens_Lower_Threshold ∧
monitored_value ≤ Sens_Upper_Threshold) ⇒
sensor_fault' = TRUE)
unit_status :∈ STATUS
end

event Receiving_Packets **refines** Receiving_Packets
any *p*
where
// other guards as in the abstract event
p ∈ PACKET
packet_status(*p*) ∈ NO_FLT
Checksum(packet_time(*p*)↦packet_data(*p*)) =
packet_checksum(*p*)
then
// other actions as in the abstract event
status(packet_unit_id(*p*)) ≔ packet_status(*p*)
checksum(packet_unit_id(*p*)) ≔
packet_checksum(*p*)
end

event Processing_OK **refines** Processing
where
// other guards as in the abstract event
pre_proc_flag = FALSE
sensor_fault = FALSE ∧ unit_status ∈ NO_FLT
then
// other actions as in the abstract event
pre_proc_flag ≔ TRUE
processed_value (0) ≔ Convert(monitored_value)
checksum(0) ≔
Checksum(curr_time↦Convert(monitored_value))
status :| status' ∈ 0 .. UNIT_NUM → STATUS ∧
(∃x · x ∈ NO_FLT ∧ status' = status ⩤ {0↦x})
end

event Processing_NOK **refines** Processing
where
// other guards as in the abstract event
pre_proc_flag = FALSE
¬(sensor_fault = FALSE ∧
unit_status ∈ NO_FLT)
then
// other actions as in the abstract event
pre_proc_flag ≔ TRUE
end

Figure 5: Events of the first refinement model

The second refinement. The aim of our last refinement step is to refine the mechanism of local clock adjustment. Every k cycles, the DPU receives the reference time signal and adjust its local clock according to it. This prevents an unbounded local clock drift and allows the overall system guarantee "global" data freshness as discussed in Section 2. For brevity, we omit showing the details of this specification. The complete development can be found in Appendix B.

Discussion of the development. Let us point out that the proposed approach is also applicable to formal modelling and verification of DPUs that are not connected to a sensor directly (e.g., *Processing Unit*$_k$ in Fig. 1). In this case, we we can assume that the DPU operates in the presence of a permanent sensor fault and, therefore, only relies on the data received from the other units. The phases related to sensor reading and processing then could be excluded from the model of such a DPU.

In our development we have focused on the logical aspects of data monitoring – the data freshness and integrity properties. Implicitly, we assume that the time-related constraints have been obtained by the corresponding real-time analysis. The real-time analysis allows us to derive the constraints on how often sensor data should be read, the DPU worst case execution time, the upper bound of network delay and how often the local clocks should

be adjusted. Usually, this kind of analysis is performed when the system is implemented, i.e., with hardware in the loop. In our previous work, we have also experimented with the verification of real-time properties in Event-B [4] and demonstrated how to assess interdependencies between timing constraints at the abstract specification level.

In the next section, we overview the industrial case study and then present the lessons learnt from the development.

5 Validating an Industrial Solution

5.1 Overview of the Industrial Case Study

Our development presented in Section 4 generalises the architecture of a Temperature Monitoring System (TMS). The TMS is a part of the data monitoring system typical for nuclear power plants. The TMS consists of three DPUs connected to the operator's display in the control room (Fig. 6).

The TMS is an instantiation of the generic architecture described in Section 2 and formally modelled in Section 4. DPUs of the TMS monitor readings of the temperature sensors installed in a certain module of the plant.

The system is redundant with the architecture 1oo3 (one out of three). The actual temperature signals are generated by two temperature sensors – Resistance Temperature Detectors

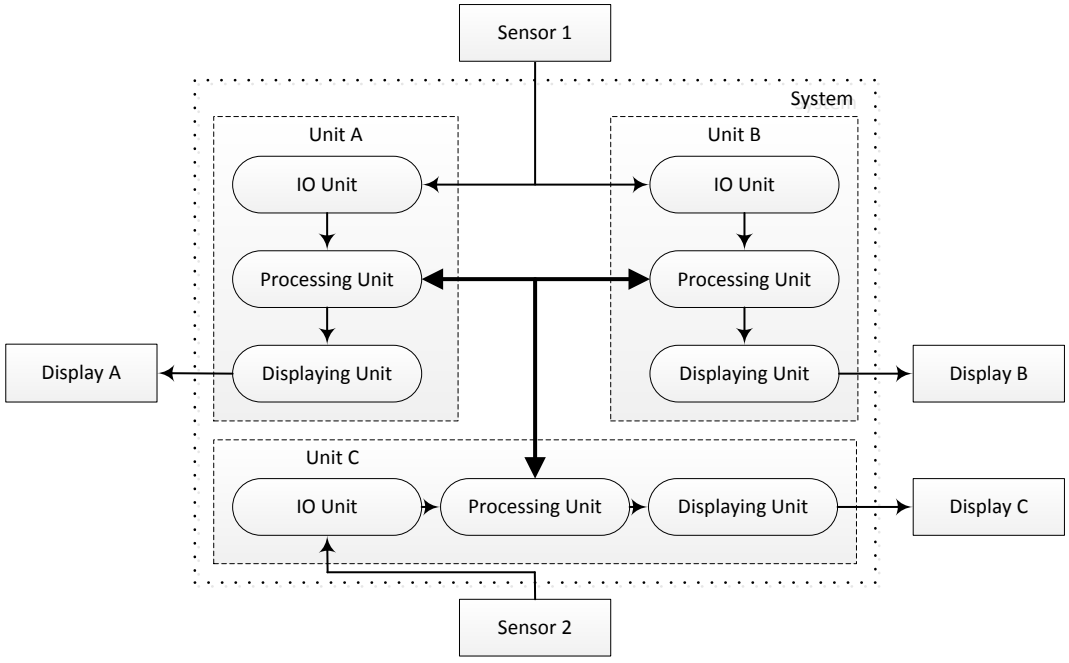


Figure 6: Temperature monitoring system

(RTDs). RTD is a thermal device containing a resistance element. The change of resistance of this element indicates the change of the temperature. Therefore, by measuring the resistance, the system can determine the temperature [5]. The temperature signal from the first sensor is transmitted to two different DPUs – Unit A and Unit B. The temperature signal from the second sensor is transmitted to the third DPU – Unit C.

After obtaining a temperature signal from the sensor, the DPU processes it and sends the temperature data further to the other DPUs. Then, the trusted temperature is communicated to the operator displays. Usually, each DPU gets all three temperature values. The temperature to be shown to the operator is then chosen as the maximum of the valid temperature values obtained. If no valid data is available, then the error message is shown to the operator warning him about a TMS error.

To guarantee that the trusted temperature data is shown to the operator, the system has to ensure integrity of the temperature data as well as its freshness. To model and verify the described system, we instantiate the proposed generic models as follows:

- ***Variables:*** *monitored_value* becomes *temp_sensor_value*, *processed_value – temperature*, and *displayed_value – output*. The rest of variables may remain unchanged, since they are not application-specific.

- ***Constants:*** all constants are assigned the values specific for the TMS.

- ***Functions:*** we instantiate the function *Output_Fun* with the function *max* because the temperature to be displayed should be the highest among the valid measurements. Moreover, the function *Convert* can be defined precisely, i.e., the actual physical law can be provided to convert a raw reading into the temperature.

- ***Invariants:*** we instantiate the invariants that guarantee the preservation of data freshness and data integrity as follows:

Freshness 1: $\mathit{main\_phase} = \mathit{ENV} \ \wedge \ \mathit{output} = \mathit{ERR\_VAL} \Rightarrow$
$(\forall i \cdot i \in \mathit{dom(timestamp)} \Rightarrow$
$\mathit{timestamp}(i) \notin \mathit{curr\_time} - \mathit{Fresh\_Delta} \mathinner{..} \mathit{curr\_time})$

Freshness 2: $\mathit{main\_phase} = \mathit{ENV} \ \wedge \ \mathit{output} \neq \mathit{ERR\_VAL} \Rightarrow$
$\neg\{x \mid \exists j \cdot j \in \mathit{dom(timestamp)} \ \wedge \ x = \mathit{timestamp(j)}\} \ \cap$
$\mathit{curr\_time} - \mathit{Fresh\_Delta} \mathinner{..} \mathit{curr\_time} = \varnothing$

Integrity 1: $\forall j \cdot j \in 0 \mathinner{..} \mathit{UNIT\_NUM} \Rightarrow$
$\mathit{Checksum(timestamp}(j) \mapsto \mathit{temperature}(j)) = \mathit{checksum}(j)$

Integrity 2: $\forall j \cdot j \in 0 \mathinner{..} \mathit{UNIT\_NUM} \Rightarrow \mathit{status}(j) \in \mathit{NO\_FLT}$

5.2 Lessons Learnt

Next we discuss our experience in applying the proposed generic development pattern to an industrial case study and describe the constraints that should be satisfied by the implementation to guarantee dependability of data monitoring.

Instantiating the generic development. Since the formal development proposed in Section 4 is generic, to model and verify the TMS, we merely had to instantiate it. This simplified the overall modelling task and reduced it to renaming the involved variables and providing correct instances for generic constants and functions, while afterwards getting the proved essential properties practically for free, i.e., without any additional proof effort. This illustrates the usefulness of involved genericity, where the used abstract data structures (constants and functions) become the parameters of the whole formal development.

In our case, this allowed us to model and verify a distributed system with an arbitrary number of units and sensors. Moreover, the introduced constants became the parameters of the system that may vary from one application to another. For instance, different sensors may have different valid thresholds, while the error value to be displayed may also depend on a particular type of a display. Furthermore, the software functions used to calculate the temperature from the raw sensor readings as well as the functions utilised to calculate the output value may differ from each other even within the same system. Nonetheless, the derived formal proofs of the data freshness and data integrity properties for each unit hold for the whole system for any valid values of the generic parameters. We believe that the presented approach can also be used in other domains without major modifications.

Validating architectural solution. The proposed generic development approach has allowed us not only to formally define two main properties of data monitoring systems – data freshness and integrity but also gain a better insight on the constraints that the proposed architecture should satisfy to guarantee dependability. Below we discuss them.

Compositionality and elasticity. Since DPUs should not produce one common reading, a DMS can be designed by composing independent DPUs, that significantly simplifies system design and verification. Firstly, the architecture enables an independent development and verification of each particular DPU. Secondly, it facilitates reasoning about the overall system behaviour, since interactions between the components can be verified at the interface level. Finally, the proposed solution allows the system to achieve elasticity – since each DPU has a pool of data, it can seamlessly adapt to various situations (errors, delays) without requiring system-level reconfiguration.

Diversity and fault tolerance. The system has several layers of fault tolerance – operator level, system level and unit level. Since the operator obtains several variants of data, (s)he can detect anomalies and initiate manual error recovery (e.g., reconfiguration). At the system level, the system exceeds its fault tolerance limit only if all N modules fail at once. Finally, at the DPU level, even if all DPUs fail to produce fresh data, DPU keeps displaying data based on the last good value until it remains fresh. At the same time, software diversity significantly contributes to achieving data integrity – it diminishes the possibility of a common processing error.

Constraints. Our formal analysis has allowed us to uncover a number of the constraints that should be satisfied to guarantee dependability. Firstly, let us observe that if a DPU keeps receiving data packets with corrupted or old data then after time δ it will start to rely only on its own data, i.e., no redundant data would be available. Therefore, potentially the system architecture can reduce itself to a single module. To avoid this, the designers should guarantee that the WCET of each module is sufficiently short for the processed data to be considered fresh by the other DPUs. Moreover, we also should guarantee that the network delays are sufficiently short and the data do not become outdated while being transmitted over the network. Furthermore, it should be verified that the successful transmission rate is high enough for a sufficient number of packets to reach their destinations non-corrupted. Finally, to guarantee "global" freshness, we have to ensure that the local clock drift is kept within the limit, i.e., does not allow DPUs to display old data.

6 Related Work and Conclusions

Related work. Traditionally, the problem of data integrity is one of the main concerns in the security domain, while data freshness is much sought after in the replicated databases. However, for our work, a more relevant is the research that focuses on achieving data integrity "from input to output", i.e., ensuring that a system does not inject faults in the data flow.

Data freshness has been studied by Sakurai et al. [6] in the context of time-triggered architectures. They propose to introduce an additional communication layer that aligns data between different replicas of an operation and define system properties using SAL. Unlike our approach, their solution relies on a tightly fixed job schedule.

Hoang at al. [7] propose a set of Event-B design patterns including a pattern for asynchronous message communication between a sender and a receiver. Each message is assigned a sequence number that is checked by a receiver. Though Hoang et al. rely on the similar technique, timestamps. The goal of their modelling – ensuring correct order of packet receiving – is different from ours. The packet ordering problem was insignificant for our study, because DPU always checks freshness of received data irrespectively of the order in which data packets are received. Westerlund and Plosila [8] treat data freshness as packet ordering problem in the Timed Action Systems framework. Though the Action System framework is similar to Event-B, it lacks an automated tool support that and hence would make verification of complex industrial systems cumbersome.

Umezawa and Shimizu [9] explore the benefits of hybrid verification methodology for ensuring data integrity. They focus on finding techniques that would be most suitable for verifying error detection, soundness of system internal states and output data integrity. This work can be seen as complementary to ours – it identifies techniques that can be used to verify freshness and integrity properties that we have formulated.

Conclusions. In this paper, we have generalised an industrial architectural solution to data monitoring and proposed a formal generic model of a data monitoring system. We formally defined and verified (by proofs) the data freshness and integrity properties. We applied the generic development pattern to verify an industrial implementation of a temperature monitoring system. Our formal modelling has also allowed us to derive the constraints that the system should satisfy to ensure that trusted (fresh and correct) data are displayed. These constraints can be seen as a guidelines facilitating design of data monitoring systems. The proposed generic development pattern can be easily instantiated to verify data monitoring systems from different domains. As a result of our modelling, we received formally grounded assurance of dependability of the proposed industrial solution.

As a future work, it would be interesting to experiment with quantitative verification of the system and propose a solution to optimising the performance-reliability ratio.

References

[1] Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, New York, NY, USA (2010)

[2] Event-B and the Rodin platform. [online] `http://www.event-b.org/` (accessed 10 February 2013)

[3] Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA (1996)

[4] Iliasov, A., Laibinis, L., Troubitsyna, E., Romanovsky, A., and Latvala, T. Augmenting Event-B Modelling with Real-Time Verification. In Proceedings of *Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA 2012).*

[5] Hashemian, H.M.: Measurement of Dynamic Temperatures and Pressures in Nuclear Power Plants. University of Western Ontario – Electronic Thesis and Dissertation Repository (2011). [online] `http://ir.lib.uwo.ca/etd/189` (accessed 19 February 2013)

[6] Sakurai, K., Bokor, P., and Suri, N.: Aiding Modular Design and Verification of Safety-Critical Time-Triggered Systems by Use of Executable Formal Specifications. In: Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium (HASE'08), pp. 261-270 (2008)

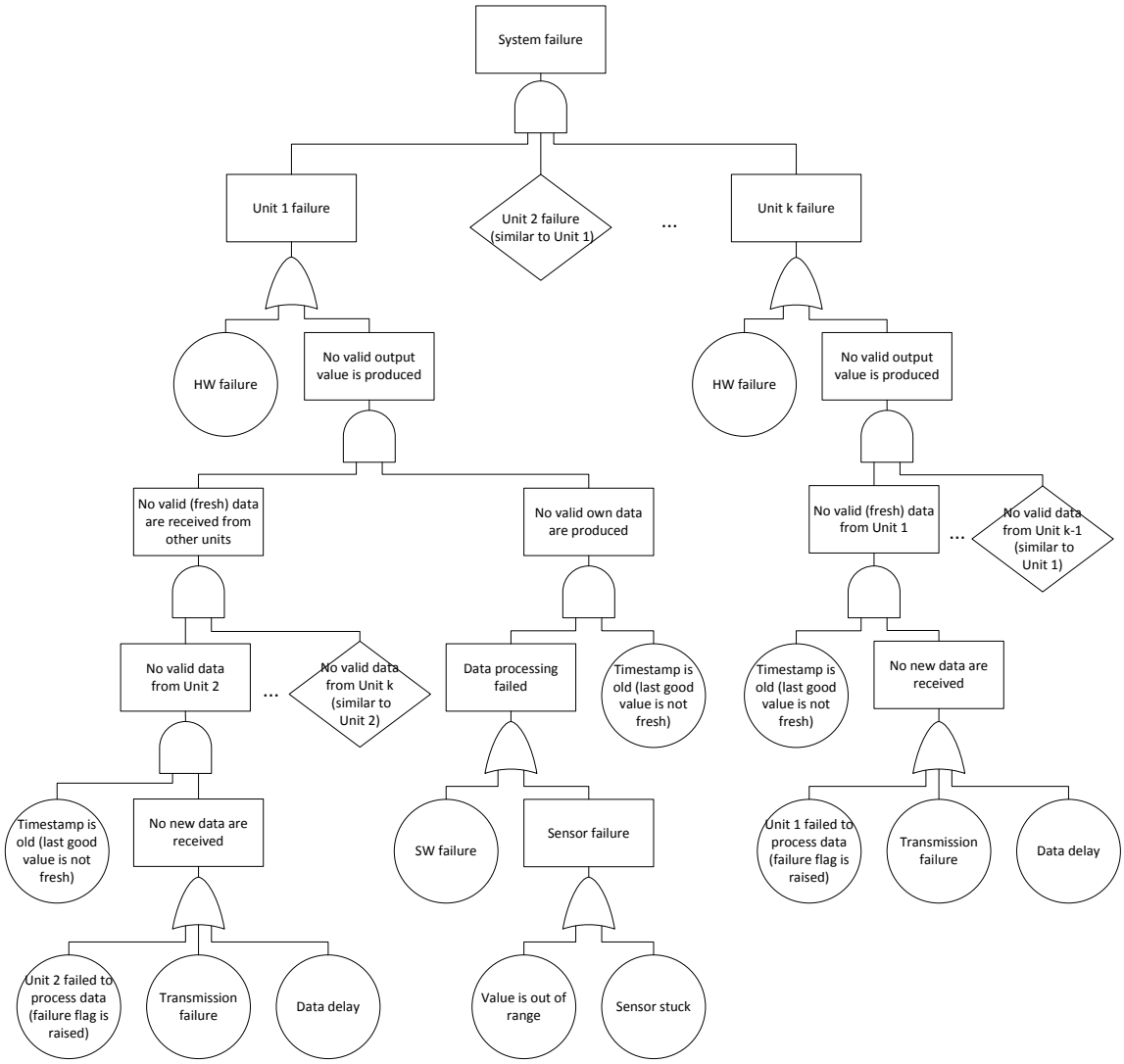
[7] Hoang, T.S., Furst, A., and Abrial, J.-R.: Event-B patterns and their tool support. In: Software and Systems Modeling, Springer-Verlag, pp. 1-16 (2011)

[8] Westerlund, T., Plosila, J.: Formal Modelling of Multiclocked SoC Systems. In: Proceedings of International Symposium on System-on-Chip, pp.1-4 (2006)

[9] Umezawa, Y. and Shimizu, T.: A Formal Verification Methodology for Checking Data Integrity. In: Proceedings of the conference on Design, Automation and Test in Europe (DATE'05), pp.284-289 (2005)

Appendix A

Fault Tree of a DMS



Appendix B

Formal Generic Development of a DMS in Event-B

CONTEXT C0

SETS

$MAIN\_PHASES$
$PACKET$

CONSTANTS

ENV
$PROC$
$DISP$
$Fresh\_Delta$
MIN_VAL
MAX_VAL
$UNIT\_NUM$
ERR_VAL
$packet\_unit\_id$
$packet\_time$
$packet\_data$
$Convert$
$Output\_Fun$

AXIOMS

axm1 : $partition(MAIN\_PHASES, \{ENV\}, \{PROC\}, \{DISP\})$
axm2 : $Fresh\_Delta = 10$
axm3 : $MIN\_VAL \in \mathbb{N}$
axm4 : $MAX\_VAL \in \mathbb{N}$
axm5 : $MIN\_VAL < MAX\_VAL$
axm6 : $ERR\_VAL = 999$
axm7 : $MAX\_VAL < ERR\_VAL$
axm8 : $UNIT\_NUM > 0$
axm9 : $PACKET \neq \varnothing$
axm10 : $packet\_unit\_id \in PACKET \rightarrow 0 \mathinner{..} UNIT\_NUM$
axm11 : $packet\_time \in PACKET \rightarrow \mathbb{N}$
axm12 : $packet\_data \in PACKET \rightarrow MIN\_VAL \mathinner{..} MAX\_VAL$
axm13 : $Convert \in \mathbb{N} \rightarrow MIN\_VAL \mathinner{..} MAX\_VAL$
axm14 : $Output\_Fun \in \mathbb{P}_1(\mathbb{N}) \rightarrow \mathbb{N}$

END

MACHINE M0

SEES C0

VARIABLES

$main\_phase$
$time\_progressed$
$packet\_sent\_flag$
$monitored\_value$
$displayed\_value$
$curr\_time$
$processed\_value$
$timestamp$

INVARIANTS

inv1 : $main\_phase \in MAIN\_PHASES$
inv2 : $time\_progressed \in BOOL$
inv3 : $packet\_sent\_flag \in BOOL$
inv4 : $monitored\_value \in \mathbb{N}$
inv5 : $displayed\_value \in \mathbb{N}$
inv6 : $curr\_time \in \mathbb{N}$
inv7 : $processed\_value \in 0 \mathinner{..} UNIT\_NUM \rightarrow MIN\_VAL \mathinner{..} MAX\_VAL$
inv8 : $timestamp \in 0 \mathinner{..} UNIT\_NUM \rightarrow \mathbb{N}$
Freshness 1 : $main\_phase = ENV \land displayed\_value = ERR\_VAL \Rightarrow (\forall i \cdot i \in dom(timestamp) \Rightarrow timestamp(i) \notin curr\_time - Fresh\_Delta \mathinner{..} curr\_time)$
Freshness 2 : $main\_phase = ENV \land displayed\_value \neq ERR\_VAL \Rightarrow \neg\{x \mid \exists j \cdot j \in dom(timestamp) \land x = timestamp(j)\} \cap curr\_time - Fresh\_Delta \mathinner{..} curr\_time = \varnothing$
Correctness : $main\_phase = ENV \land displayed\_value \neq ERR\_VAL \Rightarrow displayed\_value = Output\_Fun(\{x \mapsto y \mid \exists i \cdot i \in dom(timestamp) \land x = timestamp(i) \land y = processed\_value(i)\}[curr\_time - Fresh\_Delta \mathinner{..} curr\_time])$

EVENTS

Initialisation

begin

act1 : $main\_phase := ENV$
act2 : $time\_progressed := FALSE$
act3 : $packet\_sent\_flag := FALSE$
act4 : $monitored\_value := 0$

act5 : $displayed\_value := MIN\_VAL$
act6 : $curr\_time := 0$
act7 : $processed\_value := 0 \mathinner{..} UNIT\_NUM \times \{MIN\_VAL\}$
act8 : $timestamp := 0 \mathinner{..} UNIT\_NUM \times \{0\}$

end

Event *Environment* $\widehat{=}$

any

$sync\_t$

where

grd1 : $main\_phase = ENV$
grd2 : $sync\_t \in \mathbb{N}$

then

act1 : $main\_phase := PROC$
act2 : $monitored\_value :\in \mathbb{N}$
act3 : $curr\_time := sync\_t$

end

Event *Receiving_Packets* $\widehat{=}$

any

p

where

grd1 : $p \in PACKET$
grd2 : $packet\_time(p) > timestamp(packet\_unit\_id(p))$
grd3 : $packet\_data(p) \in MIN\_VAL \mathinner{..} MAX\_VAL$
grd4 : $main\_phase \neq ENV$
grd5 : $time\_progressed = TRUE$

then

act1 : $time\_progressed := FALSE$
act2 : $timestamp(packet\_unit\_id(p)) := packet\_time(p)$
act3 : $processed\_value(packet\_unit\_id(p)) := packet\_data(p)$

end

Event *Processing* $\widehat{=}$

when

grd1 : $main\_phase = PROC$
grd2 : $time\_progressed = TRUE$

then

act1 : $main\_phase := DISP$
act2 : $time\_progressed := FALSE$

act3 : $timestamp, processed\_value :|$
$timestamp' \in 0 \mathinner{..} UNIT\_NUM \to \mathbb{N} \land$
$processed\_value' \in 0 \mathinner{..} UNIT\_NUM \to$
$MIN\_VAL \mathinner{..} MAX\_VAL \land$
$((timestamp'(0) = curr\_time \land$
$processed\_value'(0) = Convert(monitored\_value)) \lor$
$(timestamp'(0) = timestamp(0) \land$
$processed\_value'(0) = processed\_value(0)))$

end

Event *Sending_Packet* $\widehat{=}$

any

p

where

grd1 : $main\_phase = DISP$
grd2 : $time\_progressed = TRUE$
grd3 : $packet\_sent\_flag = FALSE$
grd4 : $p \in PACKET$
grd5 : $packet\_time(p) = curr\_time$
grd6 : $packet\_data(p) = Convert(monitored\_value)$
grd7 : $packet\_unit\_id(p) = 0$

then

act1 : $time\_progressed := FALSE$
act2 : $packet\_sent\_flag := TRUE$

end

Event *Displaying* $\widehat{=}$

any

ss
$DATA\_SET$

where

grd1 : $main\_phase = DISP$
grd2 : $time\_progressed = TRUE$
grd3 : $packet\_sent\_flag = TRUE$
grd4 : $DATA\_SET \subseteq \mathbb{N}$
grd5 : $ss = \{x \mapsto y \mid \exists i \cdot i \in dom(timestamp) \land x = timestamp(i) \land$
$y = processed\_value(i)\}[curr\_time - Fresh\_Delta \mathinner{..} curr\_time]$
grd6 : $(ss \neq \varnothing \Rightarrow DATA\_SET = ss)$
grd7 : $(ss = \varnothing \Rightarrow DATA\_SET = \{ERR\_VAL\})$

then

act1 : $main\_phase := ENV$
act2 : $time\_progressed := FALSE$

act3 : $packet\_sent\_flag := FALSE$
act4 : $displayed\_value := Output\_Fun(DATA\_SET)$

end

Event *Time_Progress* $\widehat{=}$

any

t

where

grd1 : $t \in \mathbb{N}$
grd2 : $t > curr\_time$
grd3 : $main\_phase \neq ENV$

then

act1 : $time\_progressed := TRUE$
act2 : $curr\_time := t$

end

END

CONTEXT C1

EXTENDS C0

SETS

$STATUS$

CONSTANTS

NO_FLT

FLT

$Sens\_Lower\_Threshold$

$Sens\_Upper\_Threshold$

$packet\_status$

$packet\_checksum$

$Checksum$

AXIOMS

axm1 : $partition(STATUS, NO\_FLT, FLT)$

axm2 : $\neg NO\_FLT = \varnothing$

axm3 : $\neg FLT = \varnothing$

axm4 : $Sens\_Lower\_Threshold = 4$

axm5 : $Sens\_Upper\_Threshold = 20$

axm6 : $packet\_status \in PACKET \rightarrow STATUS$

axm7 : $packet\_checksum \in PACKET \rightarrow \mathbb{N}$

axm8 : $Checksum \in \mathbb{N} \times MIN\_VAL \mathinner{..} MAX\_VAL \rightarrow \mathbb{N}$

axm9 : $Checksum(0 \mapsto MIN\_VAL) = MIN\_VAL$

END

MACHINE M1

REFINES M0

SEES C1

VARIABLES

$main\_phase$
$time\_progressed$
$packet\_sent\_flag$
pre_proc_flag
$monitored\_value$
$displayed\_value$
$curr\_time$
$processed\_value$
$timestamp$
$displayed\_mess$
$unit\_status$
$sensor\_fault$
$status$
$checksum$

INVARIANTS

inv1 : $pre\_proc\_flag \in BOOL$
inv2 : $displayed\_mess \in STATUS$
inv3 : $unit\_status \in STATUS$
inv4 : $sensor\_fault \in BOOL$
inv5 : $status \in 0 \mathinner{..} UNIT\_NUM \rightarrow STATUS$
inv6 : $checksum \in 0 \mathinner{..} UNIT\_NUM \rightarrow \mathbb{N}$
Integrity 1 : $\forall j \cdot j \in 0 \mathinner{..} UNIT\_NUM \Rightarrow Checksum(timestamp(j) \mapsto processed\_value(j)) = checksum(j)$
Integrity 2 : $\forall j \cdot j \in 0 \mathinner{..} UNIT\_NUM \Rightarrow status(j) \in NO\_FLT$

EVENTS

Initialisation

extended

begin

act1 : $main\_phase := ENV$
act2 : $time\_progressed := FALSE$
act3 : $packet\_sent\_flag := FALSE$
act4 : $pre\_proc\_flag := TRUE$

act5 : $monitored\_value := 0$
act6 : $displayed\_value := MIN\_VAL$
act7 : $curr\_time := 0$
act8 : $processed\_value := 0 \mathinner{..} UNIT\_NUM \times \{MIN\_VAL\}$
act9 : $timestamp := 0 \mathinner{..} UNIT\_NUM \times \{0\}$
act10 : $displayed\_mess :\in NO\_FLT$
act11 : $sensor\_fault := FALSE$
act12 : $unit\_status :\in NO\_FLT$
act13 : $status :\in 0 \mathinner{..} UNIT\_NUM \rightarrow NO\_FLT$
act14 : $checksum := 0 \mathinner{..} UNIT\_NUM \times \{MIN\_VAL\}$

end

Event *Environment* $\widehat{=}$

refines *Environment*

any

$sync\_t$

where

grd1 : $main\_phase = ENV$
grd2 : $sync\_t \in \mathbb{N}$

then

act1 : $main\_phase := PROC$
act2 : $monitored\_value :\in \mathbb{N}$
act3 : $curr\_time := sync\_t$

end

Event *Receiving_Packets* $\widehat{=}$

refines *Receiving_Packets*

any

p

where

grd1 : $p \in PACKET$
grd2 : $packet\_time(p) > timestamp(packet\_unit\_id(p))$
grd3 : $packet\_data(p) \in MIN\_VAL \mathinner{..} MAX\_VAL$
grd4 : $main\_phase \neq ENV$
grd5 : $time\_progressed = TRUE$
grd6 : $packet\_status(p) \in NO\_FLT$
grd7 : $Checksum(packet\_time(p) \mapsto packet\_data(p)) = packet\_checksum(p)$

then

act1 : $time\_progressed := FALSE$
act2 : $timestamp(packet\_unit\_id(p)) := packet\_time(p)$

act3 : $processed\_value(packet\_unit\_id(p)) := packet\_data(p)$
act4 : $status(packet\_unit\_id(p)) := packet\_status(p)$
act5 : $checksum(packet\_unit\_id(p)) := packet\_checksum(p)$

end

Event *Pre_Processing* $\widehat{=}$

when

grd1 : $main\_phase = PROC$
grd2 : $time\_progressed = TRUE$
grd3 : $pre\_proc\_flag = TRUE$

then

act1 : $pre\_proc\_flag := FALSE$
act2 : $unit\_status :\in STATUS$
act3 : $sensor\_fault :| \ sensor\_fault' \in BOOL \wedge$
$((monitored\_value \geq Sens\_Lower\_Threshold \wedge$
$monitored\_value \leq Sens\_Upper\_Threshold) \Rightarrow$
$sensor\_fault' = FALSE) \wedge$
$(\neg(monitored\_value \geq Sens\_Lower\_Threshold \wedge$
$monitored\_value \leq Sens\_Upper\_Threshold)$
$\Rightarrow sensor\_fault' = TRUE)$

end

Event *Processing_OK* $\widehat{=}$

refines *Processing*

when

grd1 : $main\_phase = PROC$
grd2 : $time\_progressed = TRUE$
grd3 : $pre\_proc\_flag = FALSE$
grd4 : $sensor\_fault = FALSE \wedge unit\_status \in NO\_FLT$

then

act1 : $main\_phase := DISP$
act2 : $time\_progressed := FALSE$
act3 : $pre\_proc\_flag := TRUE$
act4 : $timestamp(0) := curr\_time$
act5 : $processed\_value(0) := Convert(monitored\_value)$
act6 : $status :| \ status' \in 0 \mathinner{..} UNIT\_NUM \rightarrow STATUS \wedge$
$(\exists x \cdot x \in NO\_FLT \wedge status' = status \Leftarrow \{0 \mapsto x\})$
act7 : $checksum(0) := Checksum(curr\_time \mapsto$
$Convert(monitored\_value))$

end

Event *Processing_NOK* $\widehat{=}$

refines *Processing*

when

grd1 : $main\_phase = PROC$
grd2 : $time\_progressed = TRUE$
grd3 : $pre\_proc\_flag = FALSE$
grd4 : $\neg(sensor\_fault = FALSE \land unit\_status \in NO\_FLT)$

then

act1 : $main\_phase := DISP$
act2 : $time\_progressed := FALSE$
act3 : $pre\_proc\_flag := TRUE$

end

Event *Sending_Packet* $\widehat{=}$

refines *Sending_Packet*

any

p
$st0$

where

grd1 : $main\_phase = DISP$
grd2 : $time\_progressed = TRUE$
grd3 : $packet\_sent\_flag = FALSE$
grd4 : $p \in PACKET$
grd5 : $st0 \in STATUS$
grd6 : $(sensor\_fault = FALSE \land unit\_status \in NO\_FLT) \Rightarrow st0 \in NO\_FLT$
grd7 : $\neg(sensor\_fault = FALSE \land unit\_status \in NO\_FLT) \Rightarrow st0 \in FLT$
grd8 : $packet\_unit\_id(p) = 0$
grd9 : $packet\_time(p) = curr\_time$
grd10 : $packet\_data(p) = Convert(monitored\_value)$
grd11 : $packet\_status(p) = st0$
grd12 : $packet\_checksum(p) = Checksum(curr\_time \mapsto Convert(monitored\_value))$

then

act1 : $time\_progressed := FALSE$
act2 : $packet\_sent\_flag := TRUE$

end

Event *Displaying* $\widehat{=}$

refines *Displaying*

any

ss
$DATA\_SET$

where

grd1 : $main\_phase = DISP$
grd2 : $time\_progressed = TRUE$
grd3 : $packet\_sent\_flag = TRUE$
grd4 : $DATA\_SET \subseteq \mathbb{N}$
grd5 : $ss = \{x \mapsto y \mid \exists i \cdot i \in dom(timestamp) \land x = timestamp(i) \land y = processed\_value(i)\}[curr\_time - Fresh\_Delta \mathinner{..} curr\_time]$
grd6 : $(ss \neq \varnothing \Rightarrow DATA\_SET = ss)$
grd7 : $(ss = \varnothing \Rightarrow DATA\_SET = \{ERR\_VAL\})$

then

act1 : $main\_phase := ENV$
act2 : $time\_progressed := FALSE$
act3 : $packet\_sent\_flag := FALSE$
act4 : $displayed\_value := Output\_Fun(DATA\_SET)$
act5 : $displayed\_mess :| \ displayed\_mess' \in STATUS \land (sensor\_fault = FALSE \land unit\_status \in NO\_FLT \land ss \neq \varnothing \Rightarrow displayed\_mess' \in NO\_FLT) \land (\neg(sensor\_fault = FALSE \land unit\_status \in NO\_FLT \land ss \neq \varnothing) \Rightarrow displayed\_mess' \in FLT)$

end

Event *Time_Progress* $\widehat{=}$

refines *Time_Progress*

any

t

where

grd1 : $t \in \mathbb{N}$
grd2 : $t > curr\_time$
grd3 : $main\_phase \neq ENV$

then

act1 : $time\_progressed := TRUE$
act2 : $curr\_time := t$

end

END

CONTEXT C2
EXTENDS C1
CONSTANTS

k the period of time synchronisation

AXIOMS

axm1 : $k \in \mathbb{N}$

END

MACHINE M2
REFINES M1
SEES C2
VARIABLES

$main\_phase$
$time\_progressed$
$packet\_sent\_flag$
pre_proc_flag
$monitored\_value$
$displayed\_value$
$displayed\_mess$
$curr\_time$
$processed\_value$
$timestamp$
$unit\_status$
$sensor\_fault$
$status$
$checksum$
$global\_time$
$cycle\_count$

INVARIANTS

inv1 : $global\_time \in \mathbb{N}$
inv2 : $cycle\_count \in \mathbb{N}$

thm1 : $\forall p' \cdot p' \in \{main\_phase' \mapsto curr\_time' \mapsto cycle\_count' \mid$
$main\_phase' \in MAIN\_PHASES \wedge$
$curr\_time' \in \mathbb{N} \wedge cycle\_count' \in \mathbb{N} \wedge$
$(\exists main\_phase, cycle\_count \cdot$
$main\_phase \in MAIN\_PHASES \wedge cycle\_count \in \mathbb{N} \wedge$
$(main\_phase = ENV \wedge cycle\_count < k) \wedge$
$(main\_phase' = PROC \wedge (cycle\_count' = cycle\_count + 1) \wedge$
$curr\_time' = curr\_time))\} \Rightarrow$
$p' \in \{main\_phase' \mapsto curr\_time' \mapsto cycle\_count' \mid$
$main\_phase' \in MAIN\_PHASES \wedge$
$curr\_time' \in \mathbb{N} \wedge cycle\_count' \in \mathbb{N} \wedge (curr\_time' = curr\_time)\}$

thm2 : $\forall p' \cdot p' \in \{main\_phase' \mapsto curr\_time' \mapsto cycle\_count' \mid$
$main\_phase' \in MAIN\_PHASES \wedge$
$curr\_time' \in \mathbb{N} \wedge cycle\_count' \in \mathbb{N} \wedge$
$(\exists main\_phase, cycle\_count \cdot$
$main\_phase \in MAIN\_PHASES \wedge cycle\_count \in \mathbb{N} \wedge$
$(main\_phase = ENV \wedge cycle\_count = k) \wedge$
$(main\_phase' = PROC \wedge cycle\_count' = 0 \wedge$
$curr\_time' = curr\_time + (global\_time - curr\_time)))\} \Rightarrow$
$p' \in \{main\_phase' \mapsto curr\_time' \mapsto cycle\_count' \mid$
$main\_phase' \in MAIN\_PHASES \wedge$
$curr\_time' \in \mathbb{N} \wedge cycle\_count' \in \mathbb{N} \wedge$
$(curr\_time' = curr\_time + (global\_time - curr\_time))\}$

EVENTS

Initialisation

extended

begin

act1 : $main\_phase := ENV$
act2 : $time\_progressed := FALSE$
act3 : $packet\_sent\_flag := FALSE$
act4 : $pre\_proc\_flag := TRUE$
act5 : $monitored\_value := 0$
act6 : $displayed\_value := MIN\_VAL$
act7 : $curr\_time := 0$
act8 : $processed\_value := 0 \mathinner{..} UNIT\_NUM \times \{MIN\_VAL\}$
act9 : $timestamp := 0 \mathinner{..} UNIT\_NUM \times \{0\}$
act10 : $displayed\_mess :\in NO\_FLT$
act11 : $sensor\_fault := FALSE$
act12 : $unit\_status :\in NO\_FLT$
act13 : $status :\in 0 \mathinner{..} UNIT\_NUM \rightarrow NO\_FLT$
act14 : $checksum := 0 \mathinner{..} UNIT\_NUM \times \{MIN\_VAL\}$
act15 : $global\_time := 0$

act16 : $cycle\_count := 0$

end

Event *Environment_1* $\widehat{=}$

refines *Environment*

where

grd1 : $main\_phase = ENV$
grd2 : $cycle\_count < k$

with

sync_t : $sync\_t = curr\_time$

then

act1 : $main\_phase := PROC$
act2 : $monitored\_value :\in \mathbb{N}$
act3 : $curr\_time := curr\_time$
act4 : $cycle\_count := cycle\_count + 1$

end

Event *Environment_2* $\widehat{=}$

refines *Environment*

any

$delta\_t$

where

grd1 : $main\_phase = ENV$
grd2 : $delta\_t = global\_time - curr\_time$
grd3 : $cycle\_count = k$

with

sync_t : $sync\_t = curr\_time + delta\_t$

then

act1 : $main\_phase := PROC$
act2 : $monitored\_value :\in \mathbb{N}$
act3 : $curr\_time := curr\_time + delta\_t$
act4 : $cycle\_count := 0$

end

Event *Receiving_Packets* $\widehat{=}$

extends *Receiving_Packets*

any

p

where

grd1 : $p \in PACKET$
grd2 : $packet\_time(p) > timestamp(packet\_unit\_id(p))$

grd3 : $packet\_data(p) \in MIN\_VAL \mathinner{..} MAX\_VAL$
grd4 : $main\_phase \neq ENV$
grd5 : $time\_progressed = TRUE$
grd6 : $packet\_status(p) \in NO\_FLT$
grd7 : $Checksum(packet\_time(p) \mapsto packet\_data(p)) = packet\_checksum(p)$

then

act1 : $time\_progressed := FALSE$
act2 : $timestamp(packet\_unit\_id(p)) := packet\_time(p)$
act3 : $processed\_value(packet\_unit\_id(p)) := packet\_data(p)$
act4 : $status(packet\_unit\_id(p)) := packet\_status(p)$
act5 : $checksum(packet\_unit\_id(p)) := packet\_checksum(p)$

end

Event *Pre_Processing* $\widehat{=}$

extends *Pre_Processing*

when

grd1 : $main\_phase = PROC$
grd2 : $time\_progressed = TRUE$
grd3 : $pre\_proc\_flag = TRUE$

then

act1 : $pre\_proc\_flag := FALSE$
act2 : $unit\_status :\in STATUS$
act3 : $sensor\_fault :| \ sensor\_fault' \in BOOL \wedge$
$((monitored\_value \geq Sens\_Lower\_Threshold \wedge$
$monitored\_value \leq Sens\_Upper\_Threshold) \Rightarrow$
$sensor\_fault' = FALSE) \wedge$
$(\neg(monitored\_value \geq Sens\_Lower\_Threshold \wedge$
$monitored\_value \leq Sens\_Upper\_Threshold) \Rightarrow$
$sensor\_fault' = TRUE)$

end

Event *Processing_OK* $\widehat{=}$

extends *Processing_OK*

when

grd1 : $main\_phase = PROC$
grd2 : $time\_progressed = TRUE$
grd3 : $pre\_proc\_flag = FALSE$
grd4 : $sensor\_fault = FALSE \wedge unit\_status \in NO\_FLT$

then

act1 : $main\_phase := DISP$

act2 : $time\_progressed := FALSE$
act3 : $pre\_proc\_flag := TRUE$
act4 : $timestamp(0) := curr\_time$
act5 : $processed\_value(0) := Convert(monitored\_value)$
act6 : $status :| \ status' \in 0 \mathinner{..} UNIT\_NUM \to STATUS \land (\exists x \cdot x \in NO\_FLT \land status' = status \lhd\!\!+ \{0 \mapsto x\})$
act7 : $checksum(0) := Checksum(curr\_time \mapsto Convert(monitored\_value))$

end

Event *Processing_NOK* $\widehat{=}$

extends *Processing_NOK*

when

grd1 : $main\_phase = PROC$
grd2 : $time\_progressed = TRUE$
grd3 : $pre\_proc\_flag = FALSE$
grd4 : $\neg(sensor\_fault = FALSE \land unit\_status \in NO\_FLT)$

then

act1 : $main\_phase := DISP$
act2 : $time\_progressed := FALSE$
act3 : $pre\_proc\_flag := TRUE$

end

Event *Sending_Packet* $\widehat{=}$

extends *Sending_Packet*

any

p
$st0$

where

grd1 : $main\_phase = DISP$
grd2 : $time\_progressed = TRUE$
grd3 : $packet\_sent\_flag = FALSE$
grd4 : $p \in PACKET$
grd5 : $st0 \in STATUS$
grd6 : $(sensor\_fault = FALSE \land unit\_status \in NO\_FLT) \Rightarrow st0 \in NO\_FLT$
grd7 : $\neg(sensor\_fault = FALSE \land unit\_status \in NO\_FLT) \Rightarrow st0 \in FLT$
grd8 : $packet\_unit\_id(p) = 0$
grd9 : $packet\_time(p) = curr\_time$
grd10 : $packet\_data(p) = Convert(monitored\_value)$
grd11 : $packet\_status(p) = st0$

grd12 : $packet\_checksum(p) = Checksum(curr\_time \mapsto Convert(monitored\_value))$

then

act1 : $time\_progressed := FALSE$
act2 : $packet\_sent\_flag := TRUE$

end

Event *Displaying* $\widehat{=}$

extends *Displaying*

any

ss
$DATA\_SET$

where

grd1 : $main\_phase = DISP$
grd2 : $time\_progressed = TRUE$
grd3 : $packet\_sent\_flag = TRUE$
grd4 : $DATA\_SET \subseteq \mathbb{N}$
grd5 : $ss = \{x \mapsto y \mid \exists i \cdot i \in dom(timestamp) \land x = timestamp(i) \land y = processed\_value(i)\}[curr\_time - Fresh\_Delta \mathinner{..} curr\_time]$
grd6 : $(ss \neq \varnothing \Rightarrow DATA\_SET = ss)$
grd7 : $(ss = \varnothing \Rightarrow DATA\_SET = \{ERR\_VAL\})$

then

act1 : $main\_phase := ENV$
act2 : $time\_progressed := FALSE$
act3 : $packet\_sent\_flag := FALSE$
act4 : $displayed\_value := Output\_Fun(DATA\_SET)$
act5 : $displayed\_mess :| \ displayed\_mess' \in STATUS \land (sensor\_fault = FALSE \land unit\_status \in NO\_FLT \land ss \neq \varnothing \Rightarrow displayed\_mess' \in NO\_FLT) \land (\neg(sensor\_fault = FALSE \land unit\_status \in NO\_FLT \land ss \neq \varnothing) \Rightarrow displayed\_mess' \in FLT)$

end

Event *Time_Progress* $\widehat{=}$

extends *Time_Progress*

any

t

where

grd1 : $t \in \mathbb{N}$
grd2 : $t > curr\_time$
grd3 : $main\_phase \neq ENV$

then

act1 : $time\_progressed := TRUE$
act2 : $curr\_time := t$

end

Event *Global_Time_Progress* $\widehat{=}$

any

t

where

grd1 : $t > global\_time$

then

act1 : $global\_time := t$

end

END

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 978-952-12-2864-3
ISSN 1239-1891