Dubravka Ilić | Elena Troubitsyna |
Linas Laibinis | Colin Snook

Formal Development of Mechanisms for Tolerating Transient Faults

Turku Centre *for* Computer Science

TUCS Technical Report
No 763, April 2006

Formal Development of Mechanisms for Tolerating Transient Faults

Dubravka Ilić

Elena Troubitsyna

Linas Laibinis

Åbo Akademi University, Department of Computer Science, Lemminkäisenkatu 14A, 20520 Turku, FIN

Colin Snook

School of Electronics and Computer Science, University of Southampton, SO17 1BJ, UK

TUCS Technical Report
No 763, April 2006

Abstract

Transient faults belong to a wide-spread class of faults typical in control systems. These are faults that appear for a short period of time and might reappear later. However, even by appearing for a short time, they might cause dangerous system errors. Hence, designing mechanisms for tolerating and recovering from transient faults is an acute issue, especially in the development of safety-critical control systems. In this paper we propose a formal development (based on the B Method) of a software-based mechanisms for tolerating transient faults. The mechanism relies on a specific architecture of error detection actions called evaluating tests. These tests are executed (with different frequencies) on predefined subsets of analyzed data. Our formal model allows us to formally express and verify interdependencies between tests as well as to define the test scheduling. Application of the proposed approach ensures proper damage confinement caused by transient faults. We use our approach in the avionics domain, focusing on a formal development of the engine Failure Management System. The proposed specification and refinement patterns can be applied in the development of control systems in other application domains as well.

Keywords: transient fault, control system, FMS, B Method, refinement

TUCS Laboratory
Distributed Systems Design

1. Introduction

Software is nowadays a crucial part of many safety-critical applications. To guarantee *dependability* [1] of such systems, we should ensure that software is not only fault-free but also is able to cope with faults of other system components. In this paper we focus on designing controllers able to withstand transient physical faults of the system components. Transient faults are temporal defects within the system [2]. They frequently occur in hardware functioning. However, design of mechanisms for tolerating transient faults is inherently complex. On the one hand, controlling software (further referred to as a controller) should not over-react on an isolated transient fault. On the other hand, it should ensure that even the isolated transient faults are not propagated further into the system. Moreover, if the fault persists, the controller should initiate the appropriate recovery actions. The algorithm for ensuring this was proposed in [3,4].

In complex fault-tolerant control systems, a controller largely consists of mechanisms for supporting fault tolerance. This is often perceived as a separate subsystem dedicated to fault tolerance. In avionics, such a subsystem is traditionally called *Failure Management System* (further referred to as FMS). The major role of FMS is to mask faulty readings obtained from sensors and hereby provide the controller with the correct information about the system state.

The requirements imposed on FMS are often changed as a result of simulation of the system behaviour under failure conditions. These changes occur at the later development stages, which complicates the design of FMS [4]. To overcome this difficulty, we propose a generic formal pattern for specifying and developing FMS. The proposed pattern can be used in the product-line development.

Obviously, correctness of FMS is essential for ensuring dependability of the overall system. Formal methods are traditionally used for reasoning about software correctness. In this paper we demonstrate how to develop FMS by stepwise refinement in the B Method [5,6]. The B Method is a formal framework for the development of dependable systems correct by construction. AtelierB [7] – the tool supporting the method – provides a high degree of automation of the verification process, which facilitates a better acceptance of the method in the industrial practice.

The work reported in this paper is conducted within EU project RODIN [8].

The paper is structured as follows: in Section 2 we describe FMS by presenting its structure, the behaviour and the error detection mechanism. In this section we also give the graphical representation of the FMS relying on data from a single sensor. In the Section 3 we give a short introduction to our modelling framework – the B Method. Section 4 demonstrates the process of developing FMS formally. We start from an abstract specification of the system and obtain the detailed specification by a number of correctness preserving refinement steps. In Section 5 we discus the proposed approach and outline the future work.

2. Failure Management System

2.1 Structure and Behaviour

Failure Management System (FMS) [3,4,9] is a part of the embedded control system as shown on Fig. 1.

The control system regularly reads data from its sensors. These readings are considered as the inputs to FMS. The outputs from FMS are forwarded to the controller. The role of FMS is to detect erroneous inputs and prevent their propagation into the controller. Hence the main purpose of FMS is to supply the controller of the system with fault-free inputs from the system environment.

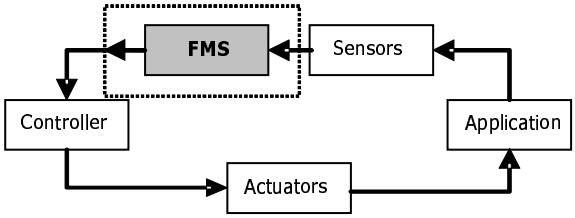


Fig. 1. Structure of an embedded control system

We assume that initially the system operates without any errors. The operating cycle starts with obtaining sensor readings which become the inputs to the FMS. FMS tests the inputs by applying a certain detection procedure. As a result, the inputs are categorized as fault-free or faulty. Then, FMS analyses the input and takes corresponding remedial action. The remedial actions can be classified as healthy, temporary or confirmation. This classification is adopted from [4].

In Fig. 2 we illustrate the general behaviour of FMS, as proposed in [3].

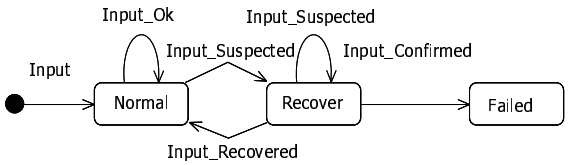


Fig. 2. Specification of the FMS behaviour

Healthy action. If FMS is in the `Normal` state (i.e., there is no suspected inputs) and the received input is fault-free, then the input is forwarded unchanged to the controller and FMS continues the operating cycle by accepting another input from the environment.

Temporary action. If FMS is in the `Normal` state and detects the first faulty input, it marks the status of that input as suspected and changes the operating state from `Normal` to `Recover` (Fig. 2). In the `Recover` state FMS starts to count the number of faulty inputs. In this state the input can get recovered during a certain number of operating cycles. One of the requirements imposed on FMS is to give a fault-free output even when the input is faulty. Hence, while operating in the `Recover` state, FMS returns the

last good value of the input obtained before entering the state `Recover`. Once a temporary action is triggered, it will keep the system in the state `Recover` as long as the status of the input coming from the environment is *suspected*. The counting mechanism determines whether the input gets recovered. If this is the case, the system changes its state from `Recover` to `Normal`.

Confirmation. If the system has been operating in the state `Recover` and the input fails to recover then the counting mechanism triggers the confirmation action. The input is confirmed failed and the system changes the operating state to `Failed`. After this the system proceeds with the control actions defined for the state `Failed`.

Next we describe error detection mechanism in detail.

2.2 Error Detection

The detection mechanism is the most important part of FMS. Its role is to determine whether the input is faulty or fault-free. In Fig. 3 we propose the architecture of the detection mechanism.

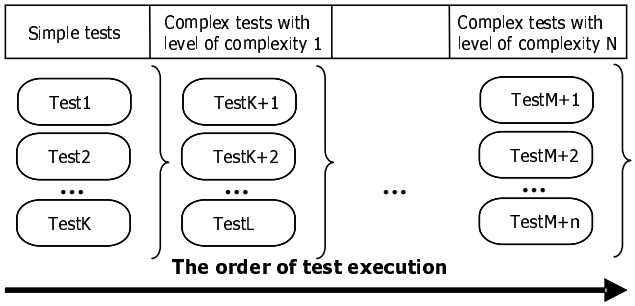


Fig. 3. Detection mechanism architecture

For each input we should define the corresponding tests required to detect whether that particular input is faulty. The detection procedure in FMS is based on applying the tests in the order defined by its architecture. The tests may vary depending on the application domain. For instance, the most commonly used tests on analogue signals in the avionics field are the magnitude test, the rate test and the predicted values test.

We differentiate between different kinds of tests. The basic category is a simple test. An input signal may pass through several simple tests, which can be applied in any order. When triggered, a simple test runs based solely on the input reading from the sensor. After the test is executed, it is marked as passed for the current input, which in turn may trigger the execution of some other test.

The second test category is complex test with the level of complexity 1. The execution of this kind of test depends on the execution of several simple tests. Input may go through several complex tests of this level. Tests can be executed in random order. However, in order to perform complex tests over a certain input, the system should first execute all required simple tests for that particular complex test as shown in Fig. 3.

In general, there might be N test categories, where the last test category is the complex test with the level of complexity N. The execution of this kind of test depends

not only on the previous execution of simple tests, but also on the execution of the complex tests with the level of complexity up to N-1. If the input requires several tests of this kind, then they are executed in random order. However, all the tests of the lower levels of complexity should be already passed. Hence, detection operates in stages, first executing all simple tests associated with the certain input and then all complex tests with ascending complexity levels as shown in Fig. 3.

In general, sensors can be classified as continuous (digital or analogue) and binary (switches). Hence, inputs to FMS can be represented as numerical or Boolean values correspondingly. For both types the detection template holds in general, but different tests can be applied on each of them.

After executing all required tests on a particular input, FMS analyses the results and finally classifies the input as faulty or fault-free.

2.3 FMS Pattern

The actions of FMS described in Fig. 2 and the detection template presented on Fig. 3 constitute the generic FMS structure and behaviour pattern as summarized in Fig. 4.

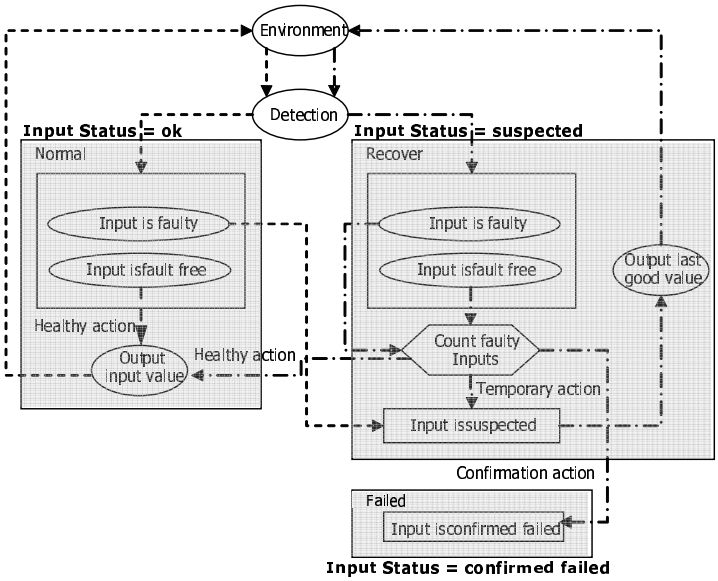


Fig. 4. FMS pattern

The figure shows the flow of the detection decisions and effect of remedial actions after the inputs are received from the system environment. Note that now three main FMS states (`Normal`, `Recover` and `Failed`) are connected with the input status. The additional component – the counting mechanism (described in detail later) – is introduced to distinguish between recoverable and unrecoverable transient faults. As a result, the system switches to the state `Normal` after the input has recovered or stays in the state `Recover` if the input is still suspected. The system enters a `Failed` state if the input failed to recover.

The given pattern can be applied in the controlling software product line [10] for creating a collection of similar control systems fault tolerant against transient faults, as

proposed in [9]. Although this pattern is created for one single sensor, it can be reused for handling N multiple sensors as well. However, when handling N multiple sensors, a system failure state might be executed when several or all sensors have failed. This system failure state corresponds to a frozen state or selection of a backup controller when the system becomes sufficiently degraded to be unsafe.

Next we give a short introduction into the B Method.

3. Formal Modelling in the B Method

In this paper we have chosen the B Method [5,6] as our formal modelling framework. The B Method is an approach for the industrial development of highly dependable software that has been successfully used in the development of several complex real-life applications [11]. The tool support available for B provides us with the assistance for the entire development process with a high degree of automation in verifying correctness. For instance, Atelier B [7], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. The high degree of automation in verifying correctness improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

In B, a specification is represented by a module or a set of modules, called Abstract Machines. The common pseudo-programming notation, called Abstract Machine Notation (AMN), is used to construct and formally verify them. An abstract machine encapsulates a state and operations of the specification and has the following general form:

```
MACHINE            name
SETS               Set
VARIABLES          v
INITIALISATION     Init
INVARIANT          I
OPERATIONS         Op
```

Each machine is uniquely identified by its name. The state variables of the machine are declared in the `VARIABLES` clause and initialized in the `INITIALISATION` clause. The variables in B are strongly typed by constraining predicates of the `INVARIANT` clause. The constraining predicates are conjoint by conjunction (denoted as `&`). All types in B are represented by non-empty sets and hence set membership (denoted as `:`) expresses typing constraint for a variable, e.g., `x:TYPE`. Local types can be introduced by enumerating the elements of the type, e.g., `TYPE = {element1, element2,…}` in the `SETS` clause. The operations of the machine are atomic and they are defined in `OPERATIONS` clause. To describe the computation in operations we use the B statements listed in the Table 1.

In this paper we adopt the event-based approach to system modelling [12]. The events are specified as the guarded operations of the form:

Event = SELECT cond THEN body END

Here cond is a state predicate, and body is a B statement describing how the state variables are affected by the operation. If cond is satisfied, the behaviour of the guarded operation corresponds to the execution of its body. If cond is false at the current state then the operation is disabled, i.e., cannot be executed. The event-based modelling is especially suitable for describing reactive systems, typical examples of which are the control systems. Then a SELECT operation describes the reaction of the system when particular event occurs.

Table 1. List of B statements used in our operations

Statement	Informal meaning
x := e	Assignment
x, y := e1, e2	Multiple assignment
IF P THEN S1 ELSE S2 END	If P is true then execute S1, otherwise S2
S1 ; S2	Sequential composition
S1 || S2	Parallel execution of S1 and S2
x :: T	Nondeterministic assignment – assigns variable x arbitrary value from given set T
ANY x WHERE Q THEN S END	Nondeterministic block – introduces a new local variable x according to the predicate Q, which is then used in S
CHOICE S OR T OR … OR U END	Nondeterministic choice – one of the statements S, T… U is arbitrarily chosen for execution

B also provides structuring mechanisms for modularization, which allows us to express machines as compositions of other machines. For instance, if in the specification of machine M1 we define that M1 SEES M2, where M2 is another machine, then the sets, the constants and the state of M2 are available to M1 for the use in its own initialization and within preconditions and the bodies of operations. In particular, this allows us to define widely used sets and constants in a separate machine and then make it "seen" by all other machines where these sets and constants are needed.

The development methodology adopted by B is based on stepwise refinement [13]. The result of a refinement step in B is a machine called REFINEMENT. Its structure coincides with the structure of the abstract machine. The refined machine contains an additional clause REFINES, which directly refers to the machine refined by the current machine. Moreover, besides typing of variables, the invariant of the refinement machine includes the refinement relation (linking invariant) that describes the connection between the state spaces of more abstract and refined machines.

Sometimes, both in MACHINE and REFINEMENT, it is useful to introduce user's own definitions as abbreviations for certain complex expressions. This can be formulated in the DEFINITION clause.

To ensure correctness of a specification or a refinement, we should verify that initialization and each operation preserve the machine invariant. Verification can be completely automatic or user-assisted. In the former case, the tool generates the required proof obligations and discards them without user's help. In the latter case, the user proves certain proof obligations using the interactive prover provided by the tool.

Next we demonstrate how to formally specify a failure management system described in the previous section.

4. Formal Development of FMS

4.1 FMS Specification Pattern

Control systems are usually cyclic, i.e., their behaviour is essentially interleaving between environment stimuli and the controller reaction on these stimuli. The controller reaction depends on whether FMS has detected error in the obtained inputs (i.e., stimuli). Hence, it is natural to consider the behaviour of FMS in the context of the overall system.

The abstract specification pattern given in Fig. 5 is obtained from the informal FMS description represented graphically in Fig. 4. The abstract specification defines the behaviour of FMS during one operating cycle. The stages of such a cycle are modelled using the variable `FMS_State`. The type `STATES` of `FMS_State` is defined in the machine `Global`, as follows:

$$STATES = \{env, det, detloop, anl, anlloop, act, out, stop\};$$

where the values of `FMS_State` define the phases of FMS execution in the following way:

- `env` – obtaining inputs from the environment,
- `detloop` and `det` – performing tests on inputs and detecting erroneous inputs,
- `anlloop` and `anl` – deciding upon the input status,
- `act` – setting the appropriate remedial actions,
- `out` – sending output to the controller either by simple forwarding of the obtained input or by calculating the output based on the last good values of inputs,
- `stop` – freezing the system.

The variable `FMS_State` models the evolution of the system behaviour in the operating cycle. At the end of the operating cycle the system finally reaches either the terminating (freezing) state or produces a fault-free output. In the latter case, the operating cycle starts again.

In our abstract specification the input values produced by the environment (i.e., sensor data) are assigned nondeterministically in the operation `Environment`. The input values produced by the sensors are modelled by the variable `InputN`. The variable represents the readings of N multiple sensors.

After obtaining the sensor readings from the environment, FMS changes its state to `Detection`. In the abstract specification we omit detailed representation of the error detection and model only its result, which is assigned to the variable `Input_In_ErrorN`. Its value is `TRUE` if the error is detected on the sensor reading on particular input and `FALSE` otherwise.

After the detection, FMS enters the `Analysis` state. Based on the results obtained at the previous state FMS decides upon the status of an input – fault-free, suspected or confirmed. The variable `Input_StatusN` is an array that for each of N inputs contains a value of the type

$$I\_STATUS = \{ok, suspected, confirmed\_failed\};$$

representing the status of this input. The nondeterministic assignment to `Input_StatusN` is bounded by the following condition: namely, if the input is not in error, its status can be either `ok` or `suspected`. If the input is in error, the assigned status is either `suspected` or `confirmed_failed`. The assignment is then written as:

$$Input\_StatusN :\in \{ ff \mid ff \in Indx \rightarrow I\_STATUS \wedge$$
$$\forall ee.(ee \in Indx \wedge Input\_In\_ErrorN(ee) = \textbf{FALSE} \Rightarrow ff(ee) \in \{ok, suspected\}) \wedge$$
$$\forall ee.(ee \in Indx \wedge Input\_In\_ErrorN(ee) = \textbf{TRUE} \Rightarrow ff(ee) \in \{suspected, confirmed\_failed\}) \}$$

Upon completing analysis, FMS applies the corresponding remedial action. A healthy action is executed if the input is fault-free; a temporary action if the input is suspected, and a confirmation action if the input is confirmed failed. While performing a healthy action, FMS forwards its input to the system controller. Then the operating cycle starts again. As a result of a temporary action FMS calculates the output based on the information about the last good input value. After this the operating cycle starts again. If FMS cannot properly function after the input has failed, the system goes into the freezing state. Otherwise, it removes the input which has been confirmed failed from further observations. In the latter case, the output is calculated based on the last good input value (similarly as in a temporary action).

Since the controller of the system relies only on the input it obtains from FMS, in our safety invariant we express error confining conditions:

$$Safety\ Invariant ==$$
$$((FMS\_State = act \Rightarrow \forall (ee).(ee \in Indx \wedge Input\_In\_ErrorN(ee) = \textbf{FALSE} \Rightarrow$$
$$Input\_StatusN(ee) \in \{ok, suspected\})) \wedge$$

$$(FMS\_State = act \Rightarrow \forall (ee).(ee \in Indx \wedge Input\_In\_ErrorN(ee) = \textbf{TRUE} \Rightarrow$$
$$Input\_StatusN(ee) \in \{suspected, confirmed\_failed\})) \wedge$$

$$(Indx = \varnothing \Rightarrow FMS\_State = stop))$$

Here `Indx` is a set of `ok` or `suspected` inputs. The first predicate states that whenever FMS is in the state `act` and some input `ee` is detected fault-free, the value assigned to the variable `Input_StatusN` is either `ok` or `suspected`. The second predicate expresses similarly that, whenever FMS is in the state `act` and the error is detected for some input `ee`, the value assigned to the variable `Input_StatusN` is either `suspected` or `confirmed_failed`. Finally, the last predicate states that whenever the variable `Indx` is empty, which means that all the inputs have failed, `FMS_State` becomes `stop` (i.e., the system goes into the freezing state).

```
MACHINE      FMS
SEES         Global
VARIABLES    Indx, InputN, Input_StatusN, Input_In_ErrorN, Last_Good_InputN, Output, FMS_State
INVARIANT
        … ∧ <safety invariant>
INITIALISATION
        …
OPERATIONS

Environment=
SELECT FMS_State=env
THEN
        InputN :∈ Indx → T_INPUT || FMS_State := det
END;

Detection=
SELECT FMS_State=det
THEN
        Input_In_ErrorN :∈ Indx → BOOL || FMS_State := anl
END;

Analysis=
SELECT FMS_State=anl
THEN
        Input_StatusN :∈ {ff | ff∈ Indx → I_STATUS ∧
        ∀ee.(ee∈Indx ∧ Input_In_ErrorN(ee)=FALSE ⇒ ff(ee)∈{ok,suspected}) ∧
        ∀ee.(ee∈Indx ∧ Input_In_ErrorN(ee)=TRUE ⇒
                        ff(ee)∈{suspected,confirmed_failed}) } || FMS_State := act
END;

Action=
SELECT FMS_State=act ∧ confirmed_failed ∈ ran(Input_StatusN)
THEN
        CHOICE
                <remove the inputs which are confirmed failed from further observation> ||
                FMS_State:=out
        OR      FMS_State:=stop
        END
WHEN
        FMS_State=act ∧ confirmed_failed ∉ ran(Input_StatusN)
THEN
        FMS_State:=out
END;

Return=
SELECT FMS_State=out
THEN
        <save the values of healthy inputs in Last_Good_InputN and set outputs to
          inputs value for healthy inputs and to last good values for non-healthy inputs> ||
        FMS_State:=env
END;

Freeze=
SELECT FMS_State=stop
THEN
        skip
END
END
```

Fig. 5. Excerpt from the abstract FMS specification pattern

Our initial specification of FMS abstractly describes the intended behaviour of FMS. However, it leaves the mechanism of detecting errors and the analysis of inputs underspecified. Next we demonstrate how to fill in these details by refinement.

4.2 Refining Input Analysis in FMS

In our first refinement step we introduce a detailed specification of the input analysis procedure.

REFINEMENT *FMSR1*
REFINES *FMS*
VARIABLES
… *Input_StatusN1*, *Processed*
INVARIANT
/* linking invariant */
(∀*ee*.(*ee*∈ *Indx* ∧ *Processed*(*ee*)=**TRUE** ∧ *Input_In_ErrorN*(*ee*)=**TRUE** ⇒
Input_StatusN1(*ee*)∈ {*suspected*,*confirmed_failed*})) ∧
(∀*ee*.(*ee*∈ *Indx* ∧ *Processed*(*ee*)=**TRUE** ∧ *Input_In_ErrorN*(*ee*)=**FALSE** ⇒
Input_StatusN1(*ee*)∈ {*ok*,*suspected*})) ∧ ...
OPERATIONS

Environment=…
Detection=…

AnalysisLoop=
SELECT *FMS_State=anlloop*
THEN
ANY *ii* **WHERE** *ii*∈ *Indx* ∧ *Processed*(*ii*)=**FALSE**
THEN
<decide upon the value of *Input_StatusN1* depending on
the previous status *Input_StatusN* and the information if
the error is detected *Input_In_ErrorN*> ||
Processed(*ii*):=**TRUE**
END;
IF ran(*Processed*)={**TRUE**} **THEN** *FMS_State*:=*anl* **ELSE** *FMS_State*:=*anlloop* **END**
END;

Analysis=
SELECT *FMS_State=anl*
THEN
Input_StatusN := *Input_StatusN1* || *FMS_State*:=*act*
END;

Action=…
Return=…
Freeze=…

END

Fig. 6. First FMS refinement – specifying input analysis

In the initial FMS specification the input analysis was modelled by the nondeterministic assignment to the variable `Input_StatusN` in the operation `Analysis`. In the refined specification we calculate the current value of the input status based on the value of `Input_In_ErrorN` and the value of input status obtained at the previous cycle of FMS. Namely, if the analysed input was `ok` (fault-free), after the error is detected it becomes `suspected` (faulty). If the input was already `suspected` and the error is detected again, it can either stay `suspected` or become `confirmed_failed`. This information is used to construct the linking invariant as shown in Fig. 6. In this refinement step we add the new operation `AnalysisLoop`. The operation gradually performs input analysis, considering inputs one by one until all the inputs are processed. The information about the input status is correspondingly accumulated in the variable `Input_StatusN1`. After the `AnalysisLoop` is completed, the value of `Input_StatusN1` is assigned to `Input_StatusN` in the operation `Analysis`.

REFINEMENT *FMSR2*
REFINES *FMSR1*
VARIABLES … *cc*, *num*
OPERATIONS

Environment=…
Detection=…
AnalysisLoop=
  SELECT *FMS_State=anlloop*
  THEN
    ANY *ii* **WHERE** *ii*∈ *Indx* ∧ *Processed*(*ii*)=FALSE …
    THEN
      IF *Input_In_ErrorN*(*ii*)=FALSE
      THEN
        IF *Input_StatusN*(*ii*)=*suspected*
        THEN <decrement cc_{ii} by *yy*>; <increment the num_{ii} >;
          IF <num_{ii} is less then defined *Limit* and
            cc_{ii} reached *zero*>
          THEN <input is recovered>; <reset num_{ii}>
          END
        END
      ELSE
        <increment cc_{ii} by *xx*>; <increment the num_{ii}>;
        IF <num_{ii} is equal to or greater then *Limit* or cc_{ii} reached *zz*>
        THEN <input is confirmed failed>
        ELSE <input is suspected>
        END
      END ‖ …
  END;
Analysis=…
Action=…
Return=…
Freeze=…

END

Fig. 7. Second FMS refinement – specifying error recovery

Our second refinement step (Fig. 7) aims at introducing a detailed procedure for determining the input status in the operation `AnalysisLoop`. The procedure is based on using a customisable counting mechanism which re-evaluate the status of a particular input at each cycle.

For each of N inputs, we introduce counters $cc_i (i \in 1..N)$, which contain accumulated values determining how trustworthy a particular input `i` is. If $cc_i = 0$ then the input `i` is `ok`. If $0 < cc_i < zz$, where `zz` is some predefined value, then the input `i` is `suspected`. Otherwise, the input `i` is considered failed.

At every cycle the counters cc_i are re-evaluated depending on the detection results. Each input `i` that was found in error increments the counter cc_i by a certain predefined value `xx`. Similarly, if the input `i` was found not in error by the detection procedure, the corresponding counter cc_i is decremented by another predefined value `yy`.

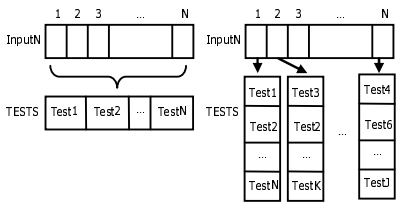
If at some point the value of cc_i reaches `0`, the input `i` is declared `ok`. Similarly, if the value of cc_i exceeds `zz`, the input `i` is declared `confirmed_failed` and should be removed from the set of inputs used by FMS.

The predefined values `zz`, `xx` and `yy` are set after observing the real performance of FMS. By setting the value of `xx` higher then the value of `yy`, the counter `cc` is biased towards failure. However, such a specification is insufficient for guaranteeing termination of recovery. Observe that the input may behave in such a way that the counter `cc` is practically oscillating between some values but never reaches the limit `zz` or zero. To overcome this problem we introduce the second counter $num_i (i \in 1..N)$, which counts the number of consequent recovering cycles on each suspected input (i.e., when $0 < cc_i < zz$). When a certain limit for num_i is exceeded, the recovery terminates and, if cc_i is different from zero, the input is confirmed failed.

4.3 Refining Error Detection in FMS

We continue the development of FMS by refining the error detection procedure. This third refinement step aims at introducing the architecture of tests which are then used by the refined error detection procedure.

The nondeterministic assignment to the variable `Input_In_ErrorN` in the abstract `Detection` operation specifies only that each of N inputs can be either faulty or fault-free. This assignment is refined in the third refinement step by introducing evaluation tests, which are applied to all inputs to determine whether they are in error or not.



a) b)

Fig. 8. Defining tests for homogeneous and heterogeneous multiple sensors

Since we observe homogeneous multiple sensors measuring the same physical process in the environment, for each of N sensor readings the same series of tests can be applied (Fig. 8a). The pattern for heterogeneous multiple sensors could easily be adapted from the one presented here and it would require defining tests for each one of the N sensor readings separately, as shown in Fig. 8b.

The architecture of tests used for error detection follows the idea of test dependencies presented in Section 2.2. The set of all tests (modelled by deferred set `TESTS`) is partitioned into two subsets:

$$S\_TEST \subseteq TESTS \wedge C\_TEST \subseteq TESTS$$

where `S_TEST` is the set of all simple tests and `C_TEST` is the set of complex tests. Moreover, since each complex test depends on some simple tests, we define this dependency as the following constant function:

$$ComplexTest \in C\_TEST \rightarrow POW(S\_TEST)$$

To model the error detection, we add the new operation `DetectionLoop`. The operation guard, as given below, defines which tests are actually enabled for execution. We introduce the relation `TestExecuted` which contains only those pairs `(ii,te)`, where `ii` is the input and `te` is the test, such that input `ii` is tested by the test `te`.

DetectionLoop=
ANY *ii*,*te* **WHERE** $FMS\_State=detloop \wedge ii \in Indx \wedge te \in TESTS \wedge (ii,te) \notin TestExecuted \wedge$
$Input\_In\_ErrorN1(ii)=$**FALSE** $\wedge$
$(te \in C\_TEST \Rightarrow \forall mm.(mm \in ComplexTest(te) \Rightarrow (ii,mm) \in TestExecuted))$
THEN
CHOICE $TestPassed:=TestPassed \cup \{ii \mapsto te\}$ **OR** skip **END**;
IF $(ii,te) \notin TestPassed$
THEN
$Input\_In\_ErrorN1(ii):=$**TRUE** ||
<all the tests associated with the input *ii* are marked as executed,
i.e., after the input has failed no more tests on that input are executed>
ELSE <the test *te* for the input *ii* is marked as executed>
END || $FMS\_State :\in \{detloop, det\};$
END

This operation guard implements the following requirements imposed on evaluating tests:

[req1] each test can be executed at most once on a certain input, i.e., for some input `ii` and test `te` predicate $(ii,te) \notin TestExecuted$ should hold;

[req2] if the test is complex, then all the simple tests it depends on have to be already executed, i.e., predicate
$(te \in C\_TEST \Rightarrow \forall mm.(mm \in ComplexTest(te) \Rightarrow (ii,mm) \in TestExecuted))$
should hold and

[req3] if some input has failed, i.e., the error on input is detected, then no more tests on that input should be applied, or formally, only tests for which the predicate $Input\_In\_ErrorN1(ii)=$**FALSE** holds, are chosen for execution.

After executing the chosen test, we model the result of this execution as a variable `TestPassed`, which keeps all successfully passed tests on particular input. This value is afterwards checked and if the test on the input failed, the input is found in error.

Similarly as in the operation `AnalysesLoop`, `DetectionLoop` performs error detection on inputs one by one. The information about the inputs in error is correspondingly accumulated in the variable `Input_In_ErrorN1`. After the `DetectionLoop` is completed, the value of `Input_In_ErrorN1` is assigned to `Input_In_ErrorN`.

The invariant of this third refinement step guarantees that if any of the tests applied on a certain input failed, then the input is considered in error:

$$(\forall (ii,te).(ii \in Indx \wedge te \in TESTS \wedge (ii,te) \in TestExecuted \wedge (ii,te) \notin TestPassed \Rightarrow Input\_In\_ErrorN1(ii) = \textbf{TRUE}))$$

In other words, in order for some input to be error free, it should successfully pass all the required tests.

The process of error detection can be graphically represented as shown in Fig. 9.

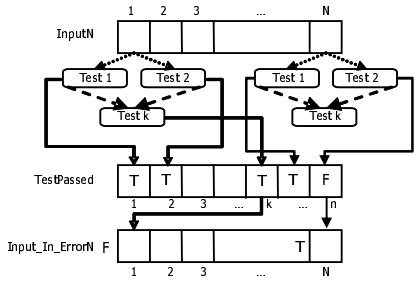


Fig. 9. Process of deciding upon the error detection

The mechanism of error detection can be further refined. Namely, which tests are enabled for execution depends not only on the requirements listed in req1-3 but also on some additional conditions on the required test frequencies and the internal state of the system:

[req4] every test is executed with a certain frequency; The test frequency can be different for different tests;

[req5] in order for some complex test to be executed, its frequency has to be divisible by the frequencies of all the simple tests required for its execution; This requirement is necessary in order to ensure the application of all required tests on the same data;

[req6] the execution of each test may depend on the current internal state of the system;

We introduce the constant function `Freq: TESTS --> NAT` which defines the frequency for each test. The state of the system is modelled as a variable `State`, whose values are assigned from the set `STATE`.

With the new requirements in mind, we develop the fourth FMS refinement step. In order to apply tests according to the given frequencies, we introduce *time scheduling*. There is one global clock guaranteeing that the tests with the same frequency are

executed at the same time instances. We model the real time by introducing the event `TickTime` which increments the current `Time` whenever the event is enabled. In addition, the operation `TickTime` models possible change of the internal system state by nondeterministically updating the variable `State`.

TickTime=
SELECT *Clock_Flag=enabled*
THEN
Time:=*Time*+1 || *State* :∈ *STATE*;
IF *Exist_Test_For_Execution* **THEN** *Clock_Flag*:=*disabled* **END**
END;

The progress of time is allowed in two situations:

- when one FMS operation cycle finishes and before the next one starts, or
- when there are no tests enabled for execution under given conditions.

In that case we allow time to progress and possibly update the internal system state until some tests become enabled.

The conditions under which the tests are enabled combine some condition on the internal state (modelled by the abstract function `Cond`) and checking the required frequency:

$$CONDITION(tt,ti,st)==(Cond(tt,st)=\mathbf{TRUE} \wedge (ti \ \mathbf{mod} \ Freq(tt)=0));$$

The above definition expresses that a particular test `tt` is enabled for execution at the time `ti` and at the system state `st`. With this definition we strengthen the guard of the operation `DetectionLoop` so that it implements requirements req4-6:

DetectionLoop=
ANY *ii,te* **WHERE** $FMS\_State=detloop \wedge ii \in Indx \wedge te \in TESTS \wedge (ii,te) \notin TestExecuted \wedge$
$Input\_In\_ErrorN1(ii)=\mathbf{FALSE} \wedge$
$(te \in C\_TEST \Rightarrow \forall mm.(mm \in ComplexTest(te) \Rightarrow$
$(ii,mm) \in TestExecuted \wedge (Freq(te) \ \mathbf{mod} \ Freq(mm)=0))) \wedge$
$CONDITION(te,Time,State)$
THEN
… || **IF** *StopCond* **THEN** *FMS_State*:=*det* **END**
END;

The termination of the detection procedure in the operation `DetectionLoop` is accomplished by explicitly checking the defined stopping condition, which becomes true only when all required tests for all inputs have been executed:

$$StopCond== \\ (\forall(ii,te).(ii \in Indx \wedge te \in TESTS \wedge CONDITION(te,Time,State) \Rightarrow (ii,te) \in TestExecuted))$$

We can summarize our formal development of FMS as follows. We start with a simple specification of the system together with its environment, which abstractly models the necessary stages of FMS execution (input reading, detection, input analysis,…). The first two refinements focus on introducing a detailed procedure for analyzing inputs and determining their current status in the system. The next refinement

step introduces the architecture of tests to be used by the refined error detection procedure. Finally, the last refinement elaborates on error detection by modelling the required time scheduling of the tests. The specification of the full development can be found in the Appendix.

5. Conclusion

In this paper we proposed a formal pattern for specifying and refining a part of the safety-critical control system – the Failure Management System. Our formal development of FMS aims at specifying and refining, firstly, the analysis of inputs by introducing a customisable counting mechanism and, secondly, error detection procedure on N multiple homogeneous sensors based on applying a certain architecture of tests. Moreover, in order to assure application of tests on the same data, i.e., data collected at the same time instances, we introduced test scheduling. The test scheduling is implemented by introducing one global clock and enabling the progression of time only when the whole FMS operating cycle finishes or when there are no enabled tests for execution. In this way, we prevent the deadlock which might occur while executing the detection operation.

Laibinis and Troubitsyna have proposed a formal approach to model-driven development of fault tolerant control systems in B [14]. However, they did not consider transient faults. Since we consider this type of faults our approach is an extension of the pattern proposed in their work.

Formal development of FMS has also been undertaken in [3,9]. This work is focused on reusability and portability of FMS modelled using UML-B [15]. However, the dependencies between tests are not explicitly addressed. The error detection mechanism we proposed is based on a hierarchical test architecture allowing us to tackle the input anomalies more efficiently.

A similar goal – design of software-implemented fault tolerance – was studied in [16,17,18]. This work focused on studying how to modify software at the code level to achieve fault tolerance. Our approach is complementary: we aimed at studying how to specify and develop software with fault tolerance mechanism integrated into it.

We verified our complete development with the automatic tool support – Atelier B. Around 70% of proof obligations have been proved automatically by the tool. The rest have been proved using the interactive prover.

The proposed FMS refinement pattern gives a template for the instantiation of a domain-specific reliable FMS. Hence, as a future work it would be interesting to study the instantiation of the developed pattern with realistic data on concrete multiple homogeneous sensors. Also, the pattern can be modified to handle multiple heterogeneous sensors and their instantiation.

Acknowledgments

This work is supported by EU funded research project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

References

[1] Laprie, J.-C., Dependability: Basic Concepts and Terminology, Springer-Verlag, Vienna, 1991

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

[3] Johnson, I., Snook, C., Edmunds, A., and Butler, M., "Rigorous development of reusable, domain-specific components, for complex applications", In Proceedings of 3rd International Workshop on Critical Systems Development with UML, Lisbon, 2004, pp. 115-129

[4] Johnson, I., Snook, C., Rodin Project Case Study 2: Requirements Specification Document, RODIN Deliverable D4 - Traceable Requirements Document for Case Studies, Section 3, 2005, pp. 24-52

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

[6] Schneider, S., The B Method. An introduction, Palgrave, 2001

[7] Atelier B - User Manual, Version 3.6, ClearSy, Aix-en-Provence, France, 2003

[8] RODIN web page: http://rodin.cs.ncl.ac.uk/index.htm

[9] Snook, C., Poppleton, M., and Johnson, I., "The engineering of generic requirements for failure management", In Proceedings of 11th International Workshop on Requirements Engineering: Foundation for Software Quality, Oporto, 2005

[10] Bosch, J., Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach, Addison-Wesley, 2000

[11] MATISSE Handbook for Correct Systems Construction, EU-project MATISSE: Methodologies and Technologies for Industrial Strength Systems Engineering, IST-199-11345, 2003

[12] Abrial, J.-R., Event Driven Sequential Program Construction, 2001, http://www.atelierb.societe.com/ressources/articles/seq.pdf

[13] Back, R.J., and von Wright, J., Refinement Calculus: A Systematic Introduction, Springer-Verlag, 1998

[14] Laibinis, L., and Troubitsyna, E., "Refinement of fault tolerant control systems in B", In ComputerSafety, Reliability, and Security - Proceedings of SAFECOMP 2004 Lecture Notes in Computer Science, Num: 3219, Springer-Verlag, September 2004, pp. 254-268

[15] Snook, C., and Walden, M., "Use of U2B for specifying B action systems", In Proceedings of RCS'02 – International workshop on refinement of critical systems: methods, tools and experience, Grenoble, France, 2002

[16] Rebaudengo, M., Reorda, M.S., Torchiano, M., and Violante, M., "A Source-to-Source Compiler for Generating Dependable Software", IEEE International Workshop on Source Code Analysis and Manipulation, 2001, pp. 33–42

[17] Reis, G.A., Chang, J., Vachharajani, N., Rangan, R., and August, D.I., "SWIFT: Software Implemented Fault Tolerance", Proceedings of the Third International Symposium on Code Generation and Optimization, March 2005, pp. 243–254

[18] Oh, N., Mitra, S., and McCluskey, E.J., "ED4I: Error Detection by Diverse Data and Duplicated Instructions", IEEE Transactions on Computers, Vol.51, No.2, 2002, pp. 180–199

Appendix

MACHINE *Global*

SETS $STATES = \{env, det, detloop, anl, anlloop, act, out, stop\}$;
$I\_STATUS = \{ok, suspected, confirmed\_failed\}$;
$PARAMETERS = \{xx,yy,zz\}$;
$TESTS$;
$STATE$;
$CLOCK\_STATES = \{enabled, disabled\}$

ABSTRACT_CONSTANTS
T_INPUT, max_int, max_indx , $Good\_Input$, $Init\_Output$, $Config$, $Limit$, S_TEST, C_TEST, $ComplexTest$, $Cond$, $Freq$

PROPERTIES
$T\_INPUT \subseteq \mathbf{NAT} \wedge$
$\forall nn.(nn \in \mathbf{NAT} \Rightarrow nn < 2147483645) \wedge$
$max\_int = 214748364 \wedge$
$max\_indx \in \mathbf{NAT} \wedge max\_indx \geq 2 \wedge$
$Good\_Input \in T\_INPUT \wedge$
$Init\_Output \in T\_INPUT \wedge$
$Config \in PARAMETERS \rightarrow \mathbf{NAT} \wedge$
$Config(zz) \geq Config(xx) \wedge Config(xx) \geq Config(yy) \wedge (Config(yy) \in 0..1) \wedge$
$Limit \in \mathbf{NAT} \wedge$
$S\_TEST \subseteq TESTS \wedge C\_TEST \subseteq TESTS \wedge$
$S\_TEST \cap C\_TEST = \varnothing \wedge S\_TEST \cup C\_TEST = TESTS \wedge$
$ComplexTest \in C\_TEST \rightarrow POW(S\_TEST) \wedge$
$Cond : (TESTS \times STATE) \rightarrow \mathbf{BOOL} \wedge$
$Freq \in TESTS \rightarrow \mathbf{NAT}$

END

MACHINE *FMS*

SEES *Global*

VARIABLES

Indx, *InputN*, *Input_StatusN*, *Input_In_ErrorN*, *Last_Good_InputN*, *Output*, *FMS_State*

INVARIANT

$Indx \subseteq 1..max\_indx \wedge InputN \in Indx \rightarrow T\_INPUT \wedge$
$Input\_StatusN \in Indx \rightarrow I\_STATUS \wedge Input\_In\_ErrorN \in Indx \rightarrow \mathbf{BOOL} \wedge$
$Last\_Good\_InputN \in Indx \rightarrow T\_INPUT \wedge Output \in T\_INPUT \wedge FMS\_State \in STATES \wedge$
$(Indx=\varnothing \Rightarrow FMS\_State=stop) \wedge$
$(FMS\_State=act \Rightarrow \forall(ee).(ee \in Indx \wedge Input\_In\_ErrorN(ee)=\mathbf{FALSE} \Rightarrow$
$Input\_StatusN(ee) \in \{ok,suspected\})) \wedge$
$(FMS\_State=act \Rightarrow \forall(ee).(ee \in Indx \wedge Input\_In\_ErrorN(ee)=\mathbf{TRUE} \Rightarrow$
$Input\_StatusN(ee) \in \{suspected,confirmed\_failed\}))$

INITIALISATION

$Indx := 1..max\_indx \parallel$
$InputN := (1..max\_indx) \times \{Good\_Input\} \parallel$
$Input\_StatusN := (1..max\_indx) \times \{ok\} \parallel$
$Input\_In\_ErrorN := (1..max\_indx) \times \{\mathbf{FALSE}\} \parallel$
$Last\_Good\_InputN := (1..max\_indx) \times \{Good\_Input\} \parallel$
$Output := Init\_Output \parallel FMS\_State := env$

OPERATIONS

Environment=
SELECT *FMS_State=env*
THEN
$InputN :\in Indx \rightarrow T\_INPUT \parallel$
$FMS\_State := detloop$
END;

DetectionLoop=
SELECT *FMS_State=detloop*
THEN
$FMS\_State :\in \{detloop, det\}$
END;

Detection=
SELECT *FMS_State=det*
THEN
$Input\_In\_ErrorN :\in Indx \rightarrow \mathbf{BOOL} \parallel$
$FMS\_State := anlloop$
END;

AnalysisLoop=
SELECT *FMS_State=anlloop*
THEN
$FMS\_State :\in \{anlloop, anl\}$
END;

Analysis=

SELECT *FMS_State=anl*
THEN
Input_StatusN :∈ {*ff*|*ff*∈ *Indx* → *I_STATUS* ∧
∀*ee*.(*ee*∈ *Indx* ∧ *Input_In_ErrorN*(*ee*)=**FALSE** ⇒ *ff*(*ee*)∈ {*ok*,*suspected*}) ∧
∀*ee*.(*ee*∈ *Indx* ∧ *Input_In_ErrorN*(*ee*)=**TRUE** ⇒ *ff*(*ee*)∈ {*suspected*,*confirmed_failed*}) } ||
FMS_State := *act*
END;

Action=

SELECT *FMS_State=act* ∧ *confirmed_failed* ∈ **ran**(*Input_StatusN*)
THEN **CHOICE**
IF $Input\_StatusN^{-1}[\{ok,suspected\}] \neq \emptyset$
THEN
Indx := $Input\_StatusN^{-1}[\{ok,suspected\}]$ ||
InputN := $Input\_StatusN^{-1}[\{ok,suspected\}] \lhd InputN$ ||
Input_StatusN := $Input\_StatusN \rhd \{ok,suspected\}$ ||
Input_In_ErrorN := $Input\_StatusN^{-1}[\{ok,suspected\}] \lhd Input\_In\_ErrorN$ ||
Last_Good_InputN := $Input\_StatusN^{-1}[\{ok,suspected\}] \lhd Last\_Good\_InputN$ ||
FMS_State:=*out*
ELSE *FMS_State*:=*stop*
END
OR
FMS_State:=*stop*
END
WHEN
FMS_State=act ∧ *confirmed_failed* ∉ **ran**(*Input_StatusN*)
THEN
FMS_State:=*out*
END;

Return=

SELECT *FMS_State=out*
THEN
ANY *in* **WHERE** *in*=(*Last_Good_InputN* $\Leftarrow$ ($Input\_StatusN^{-1}[\{ok\}] \lhd InputN$))
THEN
Last_Good_InputN := *in* ||
Output:∈ **ran**(*in*)
END ||
Input_In_ErrorN := *Indx* × {**FALSE**} ||
FMS_State:=*env*
END;

TickTime=

BEGIN
skip
END;

Freeze=

SELECT *FMS_State=stop*
THEN
skip
END
END

REFINEMENT *FMSR1*

REFINES *FMS*

SEES *Global*

VARIABLES

Indx, *InputN*, *Input_StatusN*, *Input_StatusN1*, *Input_In_ErrorN*, *Last_Good_InputN*, *Output*, *FMS_State*, *Processed*

INVARIANT

$Input\_StatusN1 \in Indx \rightarrow I\_STATUS \wedge$
$Processed \in Indx \rightarrow \mathbf{BOOL} \wedge$

$(FMS\_State \in \{env, detloop, det\} \wedge Indx \neq \varnothing \Rightarrow \mathbf{ran}(Processed) = \{\mathbf{FALSE}\}) \wedge$
$(FMS\_State \in \{anl, act, out\} \Rightarrow \mathbf{ran}(Processed) = \{\mathbf{TRUE}\}) \wedge$
$(FMS\_State = det \Rightarrow Indx \neq \varnothing) \wedge$

$(\forall ee.(ee \in Indx \wedge Processed(ee) = \mathbf{TRUE} \wedge Input\_In\_ErrorN(ee) = \mathbf{TRUE} \Rightarrow$
$Input\_StatusN1(ee) \in \{suspected, confirmed\_failed\})) \wedge$

$(\forall ee.(ee \in Indx \wedge Processed(ee) = \mathbf{TRUE} \wedge Input\_In\_ErrorN(ee) = \mathbf{FALSE} \Rightarrow$
$Input\_StatusN1(ee) \in \{ok, suspected\})) \wedge$

$(FMS\_State \in \{act, out, env, detloop, det\} \Rightarrow Input\_StatusN = Input\_StatusN1) \wedge$
$(FMS\_State \in \{out, env, detloop, det\} \Rightarrow \mathbf{ran}(Input\_StatusN) \subseteq \{ok, suspected\}) \wedge$

$(\forall ee.(ee \in Indx \wedge FMS\_State = anlloop \wedge Processed(ee) = \mathbf{FALSE} \Rightarrow$
$Input\_StatusN(ee) = Input\_StatusN1(ee))) \wedge$

$(\forall ee.(ee \in Indx \wedge FMS\_State = anlloop \wedge Processed(ee) = \mathbf{FALSE} \Rightarrow$
$\mathbf{ran}(Input\_StatusN) \subseteq \{ok, suspected\}))$

INITIALISATION

$Indx := 1..max\_indx \parallel$
$InputN := (1..max\_indx) \times \{Good\_Input\} \parallel$
$Input\_StatusN := (1..max\_indx) \times \{ok\} \parallel Input\_StatusN1 := (1..max\_indx) \times \{ok\} \parallel$
$Input\_In\_ErrorN := (1..max\_indx) \times \{\mathbf{FALSE}\} \parallel$
$Last\_Good\_InputN := (1..max\_indx) \times \{Good\_Input\} \parallel$
$Output := Init\_Output \parallel FMS\_State := env \parallel$
$Processed := (1..max\_indx) \times \{\mathbf{FALSE}\}$

OPERATIONS

Environment=

SELECT *FMS_State=env*
THEN
$InputN :\in Indx \rightarrow T\_INPUT \parallel$
$FMS\_State := detloop$
END;

DetectionLoop=
SELECT *FMS_State=detloop*
THEN
FMS_State :∈ {*detloop*, *det*}
END;

Detection=
SELECT *FMS_State=det*
THEN
Input_In_ErrorN :∈ *Indx* → **BOOL** ||
FMS_State := *anlloop*
END;

AnalysisLoop=
SELECT *FMS_State=anlloop*
THEN
ANY *ii* **WHERE** *ii*∈*Indx* ∧ *Processed*(*ii*)=**FALSE**
THEN
IF *Input_In_ErrorN*(*ii*)=**FALSE**
THEN
IF *Input_StatusN*(*ii*)=*suspected*
THEN
ANY *ch* **WHERE** *ch*∈{*ok*,*suspected*}
THEN *Input_StatusN1*(*ii*):=*ch* **END**
END
ELSE
ANY *ch* **WHERE** *ch*∈{*suspected*,*confirmed_failed*}
THEN *Input_StatusN1*(*ii*):=*ch* **END**
END ||
Processed(*ii*):=**TRUE**
END;
IF ran(*Processed*)={**TRUE**}
THEN *FMS_State*:=*anl*
ELSE *FMS_State*:=*anlloop*
END
END;

Analysis=
SELECT *FMS_State=anl*
THEN
Input_StatusN := *Input_StatusN1* ||
FMS_State:=*act*
END;

Action=
SELECT *FMS_State=act* ∧ *confirmed_failed* ∈ **ran**(*Input_StatusN*)
THEN **CHOICE**
IF $Input\_StatusN^{-1}[\{ok,suspected\}]\neq\emptyset$
THEN
$Indx := Input\_StatusN^{-1}[\{ok,suspected\}]$ ||
$InputN := Input\_StatusN^{-1}[\{ok,suspected\}] \lhd InputN$ ||
$Input\_StatusN := Input\_StatusN \rhd \{ok,suspected\}$ ||
$Input\_In\_ErrorN := Input\_StatusN^{-1}[\{ok,suspected\}] \lhd Input\_In\_ErrorN$ ||

Last_Good_InputN := *Input_StatusN*$^{-1}$[{*ok*,*suspected*}] $\lhd$ *Last_Good_InputN* ||
Input_StatusN1:= *Input_StatusN1* $\rhd$ {*ok*,*suspected*} ||
Processed:= *Input_StatusN*$^{-1}$[{*ok*,*suspected*}] $\lhd$ *Processed* ||
FMS_State:=*out*
ELSE *FMS_State*:=*stop* **END**
OR
FMS_State:=*stop*
END
WHEN
FMS_State=*act* $\wedge$ *confirmed_failed* $\notin$ **ran**(*Input_StatusN*)
THEN
FMS_State:=*out*
END;

Return=
SELECT *FMS_State*=*out*
THEN
Last_Good_InputN := *Last_Good_InputN* $\Leftarrow$ (*Input_StatusN*$^{-1}$[{*ok*}]$\lhd$*InputN*);
Output:$\in$ **ran**(*Last_Good_InputN*) ||
Input_In_ErrorN := *Indx* $\times$ {**FALSE**} ||
Processed := *Indx* $\times$ {**FALSE**} ||
FMS_State:=*env*
END;

TickTime=
BEGIN
skip
END;

Freeze=
SELECT *FMS_State*=*stop*
THEN
skip
END

END

REFINEMENT *FMSR2*

REFINES *FMSR1*

SEES *Global*

VARIABLES

Indx, *InputN*, *Input_StatusN*, *Input_StatusN1*, *Input_In_ErrorN*, *Last_Good_InputN*, *Output*, *FMS_State*, *Processed*, *cc*, *num*

INVARIANT

$cc \in Indx \mathrel{+\!\!\!\rightarrow} \mathbf{NAT} \wedge num \in Indx \mathrel{+\!\!\!\rightarrow} \mathbf{NAT}$

INITIALISATION

$Indx := 1..max\_indx \parallel$
$InputN := (1..max\_indx) \times \{Good\_Input\} \parallel$
$Input\_StatusN := (1..max\_indx) \times \{ok\} \parallel Input\_StatusN1 := (1..max\_indx) \times \{ok\} \parallel$
$Input\_In\_ErrorN := (1..max\_indx) \times \{\mathbf{FALSE}\} \parallel$
$Last\_Good\_InputN := (1..max\_indx) \times \{Good\_Input\} \parallel$
$Output := Init\_Output \parallel FMS\_State := env \parallel$
$Processed := (1..max\_indx) \times \{\mathbf{FALSE}\} \parallel$
$cc := (1..max\_indx) \times \{0\} \parallel num := (1..max\_indx) \times \{0\}$

OPERATIONS

Environment=

SELECT *FMS_State=env*
THEN
$InputN :\in Indx \rightarrow T\_INPUT \parallel$
$FMS\_State := detloop$
END;

DetectionLoop=

SELECT *FMS_State=detloop*
THEN
$FMS\_State :\in \{detloop, det\}$
END;

Detection=

SELECT *FMS_State=det*
THEN
$Input\_In\_ErrorN :\in Indx \rightarrow \mathbf{BOOL} \parallel$
$FMS\_State := anlloop$
END;

AnalysisLoop=

SELECT *FMS_State=anlloop*
THEN
ANY *ii* **WHERE** $ii \in Indx \wedge Processed(ii)=\mathbf{FALSE} \wedge Config(yy) \leq cc(ii) \wedge cc(ii)+Config(xx) \leq max\_int \wedge num(ii)+1 \leq max\_int$
THEN
IF *Input_In_ErrorN*(*ii*)=**FALSE**
THEN

```
                    IF Input_StatusN(ii)=suspected
                    THEN
                        cc(ii):=cc(ii)-Config(yy); num(ii):=num(ii)+1;
                        IF (num(ii)<Limit ∧ cc(ii)=0)
                        THEN Input_StatusN1(ii):=ok; num(ii):=0
                        END
                    END
                ELSE
                    cc(ii):=cc(ii)+Config(xx); num(ii):=num(ii)+1;
                    IF (num(ii)≥Limit ∨ cc(ii)≥Config(zz))
                    THEN
                        Input_StatusN1(ii):=confirmed_failed
                    ELSE
                        Input_StatusN1(ii):=suspected
                    END
                END ||
                Processed(ii):=TRUE
            END;
            IF ran(Processed)={TRUE}
            THEN FMS_State:=anl
            ELSE FMS_State:=anlloop
            END
        END;

Analysis=
        SELECT FMS_State=anl
        THEN
            Input_StatusN := Input_StatusN1 ||
            FMS_State:=act
        END;

Action=
        SELECT FMS_State=act ∧ confirmed_failed ∈ ran(Input_StatusN)
        THEN  CHOICE
                IF Input_StatusN^-1[{ok,suspected}]≠∅
                THEN
                    Indx := Input_StatusN^-1[{ok,suspected}] ||
                    InputN := Input_StatusN^-1[{ok,suspected}] ⩤ InputN ||
                    Input_StatusN := Input_StatusN ⩥ {ok,suspected} ||
                    Input_In_ErrorN := Input_StatusN^-1[{ok,suspected}] ⩤ Input_In_ErrorN ||
                    Last_Good_InputN := Input_StatusN^-1[{ok,suspected}] ⩤ Last_Good_InputN ||
                    Input_StatusN1:= Input_StatusN1 ⩥ {ok,suspected} ||
                    Processed:= Input_StatusN^-1[{ok,suspected}] ⩤ Processed ||
                    cc:= Input_StatusN^-1[{ok,suspected}] ⩤ cc ||
                    num:= Input_StatusN^-1[{ok,suspected}] ⩤ num ||
                    FMS_State:=out
                ELSE FMS_State:=stop
                END
            OR
                    FMS_State:=stop
            END
```

WHEN

FMS_State=*act* ∧ *confirmed_failed* ∉ **ran**(*Input_StatusN*)

THEN

FMS_State:=*out*

END;

Return=

SELECT *FMS_State*=*out*

THEN

Last_Good_InputN := *Last_Good_InputN* $\Leftarrow$ (*Input_StatusN*$^{-1}$[{*ok*}]$\lhd$*InputN*);

Output:∈ **ran**(*Last_Good_InputN*) ||

Input_In_ErrorN := *Indx* × {**FALSE**} ||

Processed := *Indx* × {**FALSE**} ||

FMS_State:=*env*

END;

TickTime=

BEGIN

skip

END;

Freeze=

SELECT *FMS_State*=*stop*

THEN

skip

END

END

REFINEMENT *FMSR3*

REFINES *FMSR2*

SEES *Global*

VARIABLES

Indx, *InputN*, *Input_StatusN*, *Input_StatusN1*, *Input_In_ErrorN*, *Input_In_ErrorN1*, *Last_Good_InputN*, *Output*, *FMS_State*, *Processed*, *cc*, *num*, *TestExecuted*, *TestPassed*

INVARIANT

$Input\_In\_ErrorN1 \in Indx \rightarrow \mathbf{BOOL} \wedge$
$TestExecuted \in Indx \leftrightarrow TESTS \wedge$
$TestPassed \in Indx \leftrightarrow TESTS \wedge$

$(FMS\_State=env \Rightarrow \mathbf{ran}(Input\_In\_ErrorN1)=\{\mathbf{FALSE}\}) \wedge$

$(\forall(ii,te).(ii \in Indx \wedge te \in TESTS \wedge (ii,te) \in TestExecuted \wedge (ii,te) \notin TestPassed \Rightarrow Input\_In\_ErrorN1(ii)=\mathbf{TRUE})) \wedge$

$(\forall(ii,te).(ii \in Indx \wedge te \in TESTS \wedge (ii,te) \notin TestExecuted \Rightarrow (ii,te) \notin TestPassed)) \wedge$

$(\forall ee.(ee \in Indx \wedge ee \in Input\_StatusN^{-1}[\{ok,suspected\}] \Rightarrow ee \notin Input\_StatusN^{-1}[\{confirmed\_failed\}]))$

$I\_STATUS = \{ ok, suspected, confirmed\_failed \} \wedge$

$(\forall(ii,te).(ii \in Indx \wedge te \in TESTS \wedge (ii,te) \notin TestExecuted \wedge (\forall pp.(pp \in TESTS \wedge (ii,pp) \in TestExecuted \Rightarrow (ii,pp) \in TestPassed)) \Rightarrow Input\_In\_ErrorN1(ii)=\mathbf{FALSE})) \wedge$

$(\forall(ii,te).(ii \in Indx \wedge te \in TESTS \wedge (ii,te) \in TestExecuted \wedge (ii,te) \in TestPassed \wedge (\forall pp.(pp \in TESTS \wedge (ii,pp) \in TestExecuted \Rightarrow (ii,pp) \in TestPassed)) \Rightarrow Input\_In\_ErrorN1(ii)=\mathbf{FALSE}))$

INITIALISATION

$Indx := 1..max\_indx \parallel$
$InputN := (1..max\_indx) \times \{Good\_Input\} \parallel$
$Input\_StatusN := (1..max\_indx) \times \{ok\} \parallel$
$Input\_StatusN1 := (1..max\_indx) \times \{ok\} \parallel$
$Input\_In\_ErrorN := (1..max\_indx) \times \{\mathbf{FALSE}\} \parallel$
$Input\_In\_ErrorN1 := (1..max\_indx) \times \{\mathbf{FALSE}\} \parallel$
$Last\_Good\_InputN := (1..max\_indx) \times \{Good\_Input\} \parallel$
$Output := Init\_Output \parallel FMS\_State := env \parallel$
$Processed := (1..max\_indx) \times \{\mathbf{FALSE}\} \parallel$
$cc := (1..max\_indx) \times \{0\} \parallel num := (1..max\_indx) \times \{0\} \parallel$
$TestExecuted := \varnothing \parallel TestPassed := \varnothing$

OPERATIONS

Environment=
SELECT *FMS_State=env*
THEN
InputN :∈ *Indx* → *T_INPUT* ||
FMS_State := *detloop*
END;

DetectionLoop=
SELECT *FMS_State=detloop*
THEN
ANY *ii*,*te* **WHERE** *ii*∈*Indx* ∧ *te*∈*TESTS* ∧ (*ii*,*te*)∉*TestExecuted* ∧
Input_In_ErrorN1(*ii*)=**FALSE** ∧
(*te*∈*C_TEST* ⇒ ∀*mm*.(*mm*∈*ComplexTest*(*te*)⇒ (*ii*,*mm*)∈*TestExecuted*))
THEN
CHOICE
TestPassed:=*TestPassed* ∪{*ii*↦*te*}
OR
skip
END;
IF (*ii*,*te*)∉*TestPassed*
THEN
Input_In_ErrorN1(*ii*):=**TRUE** ||
TestExecuted:=*TestExecuted*∪({*ii*}×*TESTS*)
ELSE
TestExecuted:=*TestExecuted*∪{*ii*↦*te*}
END
END ||
FMS_State :∈ {*detloop*, *det*}
END;

Detection=
SELECT *FMS_State=det*
THEN
Input_In_ErrorN := *Input_In_ErrorN1* ||
FMS_State:= *anlloop*
END;

AnalysisLoop=
SELECT *FMS_State=anlloop*
THEN
ANY *ii* **WHERE** *ii*∈*Indx* ∧ *Processed*(*ii*)=**FALSE** ∧ *Config*(*yy*)≤*cc*(*ii*) ∧
cc(*ii*)+*Config*(*xx*)≤*max_int* ∧ *num*(*ii*)+1≤*max_int*
THEN
IF *Input_In_ErrorN*(*ii*)=**FALSE**
THEN
IF *Input_StatusN*(*ii*)=*suspected*
THEN
cc(*ii*):=*cc*(*ii*)-*Config*(*yy*); *num*(*ii*):=*num*(*ii*)+1;
IF (*num*(*ii*)<*Limit* ∧ *cc*(*ii*)=0)
THEN *Input_StatusN1*(*ii*):=*ok*; *num*(*ii*):=0
END

```
                    END
                ELSE
                    cc(ii):=cc(ii)+Config(xx); num(ii):=num(ii)+1;
                    IF (num(ii)≥Limit ∨ cc(ii)≥Config(zz))
                    THEN
                        Input_StatusN1(ii):=confirmed_failed
                    ELSE
                        Input_StatusN1(ii):=suspected
                    END
                END ||
                Processed(ii):=TRUE
            END;
            IF ran(Processed)={TRUE}
            THEN FMS_State:=anl
            ELSE FMS_State:=anlloop
            END
        END;

Analysis=
        SELECT FMS_State=anl
        THEN
            Input_StatusN := Input_StatusN1 ||
            FMS_State:=act
        END;

Action=
        SELECT FMS_State=act ∧ confirmed_failed ∈ ran(Input_StatusN)
        THEN  CHOICE
                IF Input_StatusN~¹[{ok,suspected}]≠∅
                THEN
                    Indx := Input_StatusN~¹[{ok,suspected}] ||
                    InputN := Input_StatusN~¹[{ok,suspected}] ⩤ InputN ||
                    Input_StatusN := Input_StatusN ⩥ {ok,suspected} ||
                    Input_In_ErrorN := Input_StatusN~¹[{ok,suspected}] ⩤ Input_In_ErrorN ||
                    Last_Good_InputN := Input_StatusN~¹[{ok,suspected}] ⩤ Last_Good_InputN ||
                    Input_StatusN1:= Input_StatusN1 ⩥ {ok,suspected} ||
                    Processed:= Input_StatusN~¹[{ok,suspected}] ⩤ Processed ||
                    cc:= Input_StatusN~¹[{ok,suspected}] ⩤ cc ||
                    num:= Input_StatusN~¹[{ok,suspected}] ⩤ num ||
                    Input_In_ErrorN1:= Input_StatusN~¹[{ok,suspected}] ⩤ Input_In_ErrorN1 ||
                    TestPassed := Input_StatusN~¹[{ok,suspected}] ⩤ TestPassed ||
                    TestExecuted := Input_StatusN~¹[{ok,suspected}] ⩤ TestExecuted ||
                    FMS_State:=out
                ELSE FMS_State:=stop
                END
            OR
                FMS_State:=stop
            END
        WHEN
            FMS_State=act ∧ confirmed_failed ∉ ran(Input_StatusN)
        THEN
            FMS_State:=out
        END;
```

Return=

SELECT *FMS_State=out*

THEN

Last_Good_InputN := *Last_Good_InputN* $\Leftarrow$ (*Input_StatusN*$^{-1}$[{*ok*}]$\lhd$*InputN*);
Output:$\in$ **ran**(*Last_Good_InputN*) ||
Input_In_ErrorN:= *Indx* $\times$ {**FALSE**} ||
Processed := *Indx* $\times$ {**FALSE**} ||
TestExecuted := $\varnothing$ ||
TestPassed := $\varnothing$ ||
Input_In_ErrorN1:= *Indx* $\times$ {**FALSE**} ||
FMS_State:=*env*

END;

TickTime=

BEGIN

skip

END;

Freeze=

SELECT *FMS_State=stop*

THEN

skip

END

END

REFINEMENT *FMSR4*

REFINES *FMSR3*

SEES *Global*

VARIABLES

Indx, *InputN*, *Input_StatusN*, *Input_StatusN1*, *Input_In_ErrorN*, *Input_In_ErrorN1*, *Last_Good_InputN*, *Output*, *FMS_State*, *Processed*, *cc*, *num*, *TestExecuted*, *TestPassed*, *Time*, *Clock_Flag*, *State*

DEFINITIONS

Exist_Test_For_Execution ==
$(\exists(ii,te).(ii \in Indx \wedge te \in TESTS \wedge (ii,te) \notin TestExecuted \wedge Input\_In\_ErrorN1(ii)=\textbf{FALSE} \wedge$
$(te \in C\_TEST \Rightarrow \forall mm.(mm \in ComplexTest(te) \Rightarrow (ii,mm) \in TestExecuted \wedge (Freq(te) \textbf{ mod } Freq(mm)=0))) \wedge$
$(Time \textbf{ mod } Freq(te)=0)))$;

CONDITION(tt,ti,st) == $(Cond(tt,st)=\textbf{TRUE} \wedge (ti \textbf{ mod } Freq(tt)=0))$;

StopCond == $(\forall(ii,te).(ii \in Indx \wedge te \in TESTS \wedge CONDITION(te,Time,State) \Rightarrow (ii,te) \in TestExecuted))$

INVARIANT

$Time \in \textbf{NATURAL} \wedge$
$Clock\_Flag \in CLOCK\_STATES \wedge$
$State \in STATE$

INITIALISATION

$Indx := 1..max\_indx \parallel$
$InputN := (1..max\_indx) \times \{Good\_Input\} \parallel$
$Input\_StatusN := (1..max\_indx) \times \{ok\} \parallel$
$Input\_StatusN1 := (1..max\_indx) \times \{ok\} \parallel$
$Input\_In\_ErrorN := (1..max\_indx) \times \{\textbf{FALSE}\} \parallel$
$Input\_In\_ErrorN1 := (1..max\_indx) \times \{\textbf{FALSE}\} \parallel$
$Last\_Good\_InputN := (1..max\_indx) \times \{Good\_Input\} \parallel$
$Output := Init\_Output \parallel FMS\_State := env \parallel$
$Processed := (1..max\_indx) \times \{\textbf{FALSE}\} \parallel$
$cc := (1..max\_indx) \times \{0\} \parallel num := (1..max\_indx) \times \{0\} \parallel$
$TestExecuted := \varnothing \parallel TestPassed := \varnothing \parallel$
$Time := 0 \parallel Clock\_Flag := disabled \parallel$
$State :\in STATE$

OPERATIONS

Environment=

SELECT $FMS\_State=env \wedge Clock\_Flag=disabled$
THEN
$InputN :\in Indx \rightarrow T\_INPUT \parallel$
$FMS\_State := detloop$
END;

DetectionLoop=
SELECT *FMS_State=detloop*
THEN
ANY *ii*,*te*
WHERE $ii \in Indx \wedge te \in TESTS \wedge (ii,te) \notin TestExecuted \wedge$
$Input\_In\_ErrorN1(ii)=\mathbf{FALSE} \wedge CONDITION(te,Time,State) \wedge$
$(te \in C\_TEST \Rightarrow \forall mm.(mm \in ComplexTest(te) \Rightarrow$
$(ii,mm) \in TestExecuted \wedge (Freq(te) \bmod Freq(mm)=0)))$
THEN
CHOICE
$TestPassed := TestPassed \cup \{ii \mapsto te\}$
OR
skip
END;
IF $(ii,te) \notin TestPassed$
THEN
$Input\_In\_ErrorN1(ii) :=$ **TRUE** ||
$TestExecuted := TestExecuted \cup (\{ii\} \times TESTS)$
ELSE
$TestExecuted := TestExecuted \cup \{ii \mapsto te\}$
END
END ||
IF *StopCond* **THEN** *FMS_State*:=*det* **END**
END;

Detection=
SELECT *FMS_State=det*
THEN
$Input\_In\_ErrorN := Input\_In\_ErrorN1$ ||
FMS_State:= *anlloop*
END;

AnalysisLoop=
SELECT *FMS_State=anlloop*
THEN
ANY *ii* **WHERE** $ii \in Indx \wedge Processed(ii)=\mathbf{FALSE} \wedge Config(yy) \leq cc(ii) \wedge$
$cc(ii)+Config(xx) \leq max\_int \wedge num(ii)+1 \leq max\_int$
THEN
IF $Input\_In\_ErrorN(ii)=\mathbf{FALSE}$
THEN
IF $Input\_StatusN(ii)=suspected$
THEN
$cc(ii) := cc(ii)-Config(yy)$; $num(ii) := num(ii)+1$;
IF $(num(ii)<Limit \wedge cc(ii)=0)$
THEN $Input\_StatusN1(ii) := ok$; $num(ii) := 0$
END
END
ELSE
$cc(ii) := cc(ii)+Config(xx)$; $num(ii) := num(ii)+1$;
IF $(num(ii) \geq Limit \vee cc(ii) \geq Config(zz))$

```
                        THEN
                        Input_StatusN1(ii):=confirmed_failed
                        ELSE
                                Input_StatusN1(ii):=suspected
                        END
                END ||
                Processed(ii):=TRUE
        END;
        IF ran(Processed)={TRUE}
        THEN FMS_State:=anl
        ELSE FMS_State:=anlloop
        END
END;

Analysis=
        SELECT FMS_State=anl
        THEN
                Input_StatusN := Input_StatusN1 ||
                FMS_State:=act
        END;

Action=
        SELECT FMS_State=act ∧ confirmed_failed ∈ ran(Input_StatusN)
        THEN
                CHOICE
                   IF Input_StatusN^-1[{ok,suspected}]≠∅
                   THEN
                      Indx := Input_StatusN^-1[{ok,suspected}] ||
                      InputN := Input_StatusN^-1[{ok,suspected}] ⩤ InputN ||
                      Input_StatusN := Input_StatusN ⩥ {ok,suspected} ||
                      Input_In_ErrorN := Input_StatusN^-1[{ok,suspected}] ⩤ Input_In_ErrorN ||
                      Last_Good_InputN := Input_StatusN^-1[{ok,suspected}] ⩤ Last_Good_InputN ||
                      Input_StatusN1:= Input_StatusN1 ⩥ {ok,suspected} ||
                      Processed:= Input_StatusN^-1[{ok,suspected}] ⩤ Processed ||
                      cc:= Input_StatusN^-1[{ok,suspected}] ⩤ cc ||
                      num:= Input_StatusN^-1[{ok,suspected}] ⩤ num ||
                      Input_In_ErrorN1:= Input_StatusN^-1[{ok,suspected}] ⩤ Input_In_ErrorN1 ||
                      TestPassed := Input_StatusN^-1[{ok,suspected}] ⩤ TestPassed ||
                      TestExecuted := Input_StatusN^-1[{ok,suspected}] ⩤ TestExecuted ||
                      FMS_State:=out
                   ELSE FMS_State:=stop
                   END
                OR
                        FMS_State:=stop
                END
        WHEN
                FMS_State=act ∧ confirmed_failed ∉ ran(Input_StatusN)
        THEN
                FMS_State:=out
        END;
```

Return=

SELECT *FMS_State=out*

THEN

Last_Good_InputN := *Last_Good_InputN* $\Leftarrow$ (*Input_StatusN*$^{-1}$[{*ok*}]$\lhd$*InputN*);

Output:$\in$ **ran**(*Last_Good_InputN*) ||

Input_In_ErrorN:= *Indx* $\times$ {**FALSE**} ||

Processed := *Indx* $\times$ {**FALSE**} ||

TestExecuted := $\varnothing$ ||

TestPassed := $\varnothing$ ||

Input_In_ErrorN1:= *Indx* $\times$ {**FALSE**} ||

FMS_State:=*env* ||

Clock_Flag:=*enabled*

END;

TickTime=

SELECT *Clock_Flag=enabled*

THEN

Time:=*Time*+1 || *State* :$\in$ *STATE*;

IF *Exist_Test_For_Execution*

THEN *Clock_Flag*:=*disabled*

END

END;

Freeze=

SELECT *FMS_State=stop*

THEN

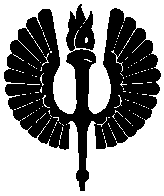
skip

END

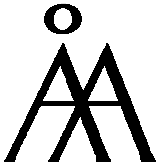
END

Turku
Centre *for*
Computer
Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi



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



Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 952-12-1709-X
ISSN 1239-1891