Anton Tarasyuk | Elena Troubitsyna | Linas Laibinis

Reliability Assessment in Event-B

Turku Centre *for* Computer Science

TUCS Technical Report
No 932, June 2009

Reliability Assessment in Event-B

Anton Tarasyuk
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, FIN-20520 Turku, Finland
anton.tarasyuk@abo.fi
Elena Troubitsyna
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, FIN-20520 Turku, Finland
elena.troubitsyna@abo.fi
Linas Laibinis
Åbo Akademi University, Department of Information Technologies
Joukahaisenkatu 3-5 A, FIN-20520 Turku, Finland
linas.laibinis@abo.fi

TUCS Technical Report
No 932, June 2009

Abstract

Formal methods are indispensable for ensuring dependability of complex software-intensive systems. In particular, the B Method and its recent extension Event-B have been successfully used in the development of several complex safety-critical systems. However, they are currently not supporting quantitative assessment of dependability attributes that is often required for certifying safety-critical systems. In this paper we demonstrate how to integrate reliability assessment into Event-B development. This work shows how to conduct probabilistic assessment of system reliability at the development stage rather than at the implementation level. This allows the developers to chose the design alternative that offers the most optimal solution from the reliability point of view.

Keywords: Reliability assessment; formal modelling; Markov processes; refinement; probabilistic model checking

TUCS Laboratory
Distributed Systems Laboratory

1 Introduction

Formal verification techniques provide us with rigorous and powerful methods for establishing correctness of complex systems. The advances in expressiveness, usability and automation of these techniques enable their use in the design of wide range of complex dependable systems. For instance, the B Method [2] and its extension Event-B [1] provide us with a powerful framework for developing systems correct-by-construction. The top-down development paradigm based on stepwise refinement adopted by these frameworks has proven its worth in several industrial projects [16, 4].

While developing system by refinement, we start from an abstract system specification and, in a number of refinement steps, introduce the desired implementation decisions. While approaching the final implementation, we decrease the abstraction level and reduce non-determinism inherently present in the abstract specifications. In general, an abstract specification can be refined in several different ways because usually there are several ways to resolve its non-determinism. These refinement alternatives are equivalent from the correctness point of view, i.e., they faithfully implement functional requirements. Yet they might be different from the point of view of non-functional requirements, e.g., reliability, performance etc. Early quantitative assessment of various design alternatives is certainly useful and desirable. However, within the current refinement frameworks we cannot perform it. In this paper we propose an approach to overcoming this problem.

We propose to integrate stepwise development in Event-B with probabilistic model checking [11] to enable reliability assessment already at the development stage. Reliability is a probability of system to function correctly over a given period of time under a given set of operating conditions [19, 20, 14]. Obviously, to assess reliability of various design alternatives, we need to model their behaviour stochastically. In this paper we demonstrate how to augment (non-deterministic) Event-B models with probabilistic information and then convert them into the form amenable to probabilistic verification. Reliability is expressed as a property that we verify by probabilistic model checking. To illustrate our approach, we assess reliability of refinement alternatives that model different fault tolerance mechanisms.

We believe that our approach can facilitate the process of developing dependable systems by enabling evaluation of design alternatives at early development stages. Moreover, it can also be used to demonstrate that the system adheres to the desired dependability levels, for instance, by proving statistically that the probability of a catastrophic failure is acceptably low. This application is especially useful for certifying safety-critical systems.

The remainder of the paper is structured as follows. In Section 2 we give a brief overview of our modelling formalism – the Event-B framework. In Section 3 we give an example of refinement in Event-B. In Section 4 we demonstrate how to augment Event-B specifications with probabilistic information and convert them

into specifications of the PRISM model checker [15]. In Section 5 we define how to assess reliability via probabilistic verification and compare the results obtained by model checking with algebraic solutions. Finally, in Section 6 we discuss the obtained results, overview the related work and propose some directions for the future work.

2 Modelling and Refinement in Event-B

The B Method is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications [16, 4]. Event-B [1] is an extension of the B Method [2] to model parallel, distributed and reactive systems. The Rodin platform [18] provides automated tool support for modelling and verification (by theorem proving) in Event-B. Currently Event-B is used in the EU project Deploy [6] to model several industrial systems from automotive, railway, space and business domains.

Event-B uses the Abstract Machine Notation [17] for constructing and verifying system models. An abstract machine encapsulates the state (the variables) of a model and defines operations on its state. A simple abstract machine has the following general form:

MACHINE AM
VARIABLES v
INVARIANTS I
EVENTS
$init$
evt_1
$\ldots$
evt_N

The machine is uniquely identified by its name AM. The state variables of the machine, v, are declared in the **VARIABLES** clause and initialised in $init$ event. The variables are strongly typed by constraining predicates of invariants I given in the **INVARIANTS** clause. The invariant is usually defined as a conjunction predicates and the predicates defining the properties of the system that should be preserved during system execution.

The dynamic behaviour of the system is defined by the set of atomic events specified in the **EVENTS** clause. An event is defined as follows:

$$evt \mathrel{\widehat{=}} \textbf{when } g \textbf{ then } S \textbf{ end}$$

where the guard g is conjunction of predicates over the state variables v, and the action S is an assignment to the state variables.

The guard defines the conditions under which the action can be executed, i.e., when the event is *enabled*. If several events are enabled then any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks.

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

The semantics of Event-B events is defined using so called before-after predicates [17]. It is a variation of the weakest precondition semantics [5]. A before-after predicate describes a relationship between the system states before and after execution of an event. The formal semantics provides us with a foundation for establishing correctness of Event-B specifications. To verify correctness of a specification we need to prove that its initialization and all events preserve the invariant.

The formal semantics also establishes a basis for system refinement – the process of developing systems correct by construction. The basic idea underlying formal stepwise development by refinement is to design the system implementation gradually, by a number of correctness preserving steps, called *refinements*. The refinement process starts from creating an abstract, albeit unimplementable, specification and finishes with generating executable code. The intermediate stages yield the specifications containing a mixture of abstract mathematical constructs and executable programming artifacts.

Assume that the refinement machine AM' is a result of refinement of the abstract machine AM:

MACHINE AM **VARIABLES** v **INVARIANTS** I **EVENTS** $init$ evt_1 $\dots$ evt_N	$\sqsubseteq$	**MACHINE** AM' **VARIABLES** v' **INVARIANTS** I' **EVENTS** $init'$ evt'_1 $\dots$ evt'_K

The machine AM' might contain new variables and events as well as replace the abstract data structures of *AM* with the concrete ones. The invariants of AM' – I' – define not only the invariant properties of the refined model, but also the connection between the state spaces of AM and AM'. For a refinement step to be

valid, every possible execution of the refined machine must correspond (via I') to some execution of the abstract machine. To demonstrate this, we should prove that $init'$ is a valid refinement of $init$, each event of AM' is a valid refinement of its counterpart in AM and that the refined specification does not introduce additional deadlocks.

In the next section we illustrate modelling and refinement in Event-B by an example.

3 Example of Refinement in Event-B

Control and monitoring systems constitute a large class of dependable systems. Essentially, the behaviour of these systems is periodic. Indeed, a control system periodically executes a control cycle that consists of reading sensors and setting actuators. The monitoring systems periodically perform certain measurements. Due to faults (e.g., caused by random hardware failures) inevitably present in any system, the system can fail to perform its functions. In this paper we focus on modelling fail-safe systems, i.e., the systems that shut down upon occurrence of failure.

In general, the behaviour of such system can be represented as shown in the specification below.

MACHINE $System$
VARIABLES res
INVARIANTS
$inv_1 : res \in BOOL$
EVENTS
$init \mathrel{\widehat{=}}$
begin
$res := TRUE$
end
$output \mathrel{\widehat{=}}$
when
$res = TRUE$
then
$res :\in BOOL$
end

For the sake of simplicity, we omit the detailed modelling of the system functionality. The variable res abstractly models success or failure to perform the required functions at each iteration. Each iteration of the system corresponds to the execution of the event $output$. If no failure has occurred then, as a result of the non-deterministic assignment, the variable res obtains the value $TRUE$. In this case the next iteration can be executed. However, if a failure has occured then res obtains the value $FALSE$ and the system deadlocks.

In the initial specification we have deliberately abstracted away from modelling system components and their failures. In the next refinement step we introduce explicit representation of system components and introduce fault tolerance mechanisms. These mechanisms allow the system to perform its functions even in the presence of certain faults [19]. Fault tolerance is usually achieved by introducing redundancy into the system design. The redundancy can be either static or dynamic. When static redundancy is used, the redundant components work in parallel the main ones. In dynamic redundancy activation of the redundant components occurs only after the main ones have failed.

Refining a system by introducing the fault tolerance mechanisms is a rather standard model transformation frequently performed in the development of dependable systems. Next we show by examples how to introduce various fault tolerance mechanisms by refinement.

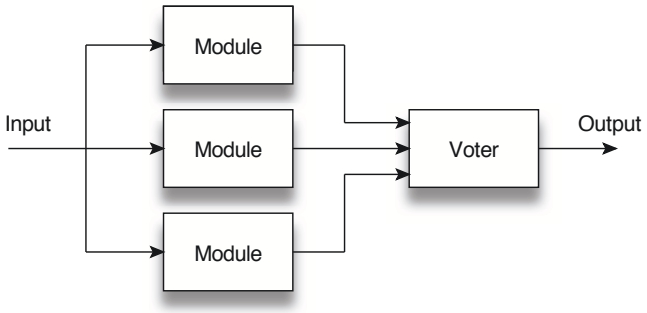


Figure 1: A Triple Modular Redundancy Arrangement

Triple Modular Redundancy (TMR) [19] is a well-known mechanism based on static redundancy. The general principle is to triplicate a system module and introduce the majority voting to obtain a single result of the module, as shown in Figure 1. Such an arrangement allows us to mask failures of a single module. TMR can be introduced into a system specification by refinement it as explained below. We introduce variables m_1, m_2 and m_3 to model the results produced by the redundant modules. The variable $phase$ models the phases of TMR execution – first reading the results produced by the modules and then voting.

MACHINE $System_{TMR}$
REFINES $System$
VARIABLES
$res, m_1, m_2, m_3, phase$
$flag_1, flag_2, flag_3$
INVARIANTS
$inv_{1..3} : m_1, m_2, m_3 \in \{0, 1\}$
$inv_4 : phase \in \{reading, voting\}$
$inv_{5..7} : flag_1, flag_2, flag_3 \in \{0, 1\}$
$inv_6 : m_1 + m_2 + m_3 > 1 \Rightarrow res = TRUE$
EVENTS
...
$module_{ok_1} \widehat{=}$
when
$m_1 = 1 \wedge flag_1 = 1 \wedge phase = reading$
then
$m_1 :\in \{0, 1\} \parallel flag_1 := 0$
end
$module_{failed_1} \widehat{=}$
when
$m_1 = 0 \wedge flag_1 = 1 \wedge phase = reading$
then
$flag_1 := 0$
end
...

$synchr \widehat{=}$
when
$flag_1 = 0 \wedge flag_1 = 0 \wedge flag_3 = 0 \wedge$
$phase = reading$
then
$phase := voting$
end
$voter_{ok} \widehat{=}$
refines $output$
when
$res = TRUE \wedge phase = voting \wedge$
$m_1 + m_2 + m_3 > 1$
then
$phase := reading \parallel$
$flag_1 := 1 \parallel flag_2 := 1 \parallel flag_3 := 1$
end
$voter_{nok} \widehat{=}$
refines $output$
when
$res = TRUE \wedge phase = voting \wedge$
$m_1 + m_2 + m_3 \leq 1$
then
$res := FALSE$
end

The modules work in parallel. In our specification it is reflected by the fact that all the events modelling module behaviour are enabled simultaneously. Each event disables itself after being executed once. When all the modules complete their execution, the event $synchr$ enables the events modelling voting. Let us observe that the invariant

$$m_1 + m_2 + m_3 > 1 \Rightarrow res = TRUE$$

relates the abstract and refined systems, i.e, it requires that the correct output can be produced only if no more than one module has failed.

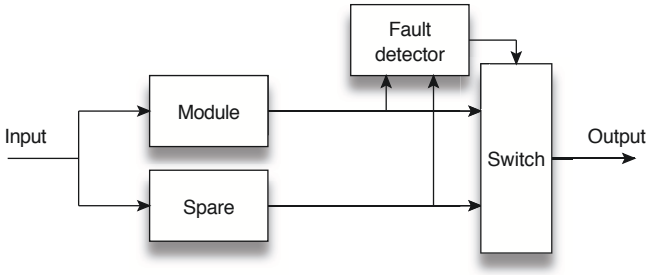


Figure 2: A Standby Spare Arrangement

In general, we can introduce any fault tolerance mechanism by refinement. Below we show other alternatives. For instance, instead of the TMR arrangement we can introduce a standby spare mechanism shown in Figure 2. In this mechanism, every result produced by an active (main) module is checked by a fault detector. If an error is detected then the result produced by the failed module is ignored and the system switches to accepting the results produced by the spare. The spare can be *hot* meaning that the main module and spare work in parallel. In this case the switch to spare happens almost instantly. The spare also can be *cold*, i.e., the spare is in the standby mode and is activated only after the main module fails.

Below we present an excerpt from the specification that refines the *System* specification to model dynamic redundancy. Here the values in and out of the variable *phase* correspond to the values $reading$ and $voting$ in the TMR specification. The additional execution phase det is introduced to model failure detection. The events that model the behaviour at this phase for the hot spare arrangement are presented below.

EVENTS
$\ldots$
$detection_{ok_1} \mathrel{\widehat{=}}$
when
$m_1 = 1 \wedge phase = det$
then
$phase := out \wedge m := m_1$
$flag_1 := 1 \wedge flag_2 := 1$
end
$detection_{ok_2} \mathrel{\widehat{=}}$
when
$m_1 = 0 \wedge m_2 = 1 \wedge phase = det$
then
$phase := out \wedge m := m_2 \wedge flag_2 := 1$
end
$detection_{nok} \mathrel{\widehat{=}}$
when
$m_1 = 0 \wedge m_2 = 0 \wedge phase = det$
then
$phase := out \wedge m := 0$
end
$\ldots$

The output can be produced successfully if at least one module functions correctly. If an error is detected then the system switches the failed module off.

Finally, we can also introduce a hybrid arrangement, which combines static and dynamic redundancy, as shown in Figure 3. The system works as TMR until a failure of a module occurs. Then the system activates the spare to ”replace” the

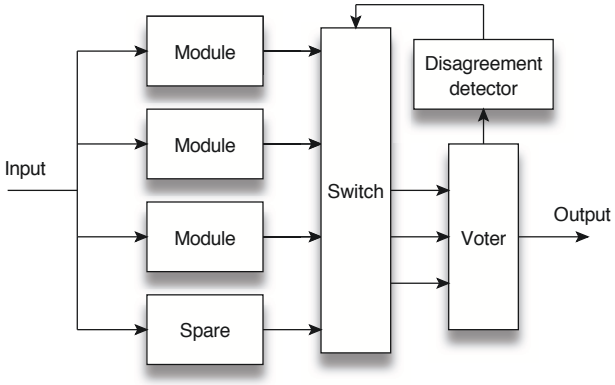


Figure 3: TMR with a Spare Arrangement

failed module. The full Event-B specifications of this and the previous arrangements can be found in Appendix.

Let us observe that any specification described above is a valid refinement of our abstract specification $System$. However, even though the fault tolerance mechanisms were introduced to increase system reliability, we cannot evaluate which of the specifications is more optimal from the point of view of reliability. This problem is caused by the non-deterministic modelling of the failure occurrence – the only possible modelling currently available in Event-B. To evaluate reliability, we need to replace the non-deterministic modelling of failure occurrence by the probabilistic ones and use the suitable techniques for reliability evaluation. Next we present our approach for achieving this.

4 From Event-B Modelling to Probabilistic Model Checking

To enable formal, probabilistic analysis of reliability in Event-B we can choose several options. The first and the most powerful is to rely on probabilistic weakest precondition semantics [12] and use probabilistic refinement technique [13] to evaluate reliability. This technique allows us to express algebraically the reliability of the system as a function of reliabilities of its components. However, for complex industrial-size systems finding this function might be very complex or even analytically intractable. A simpler and more scalable solution is to use probabilistic model checking to obtain numeric solution. To achieve this we need to augment Event-B models with probabilities in such way that they would become amenable for probabilistic verification. Then we need to establish connection between probabilistic verification and reliability assessment.

To tackle the first problem let us observe that Event-B is a state-based formal-

ism. The state space of the system specified in Event-B is formed by the values of the state variables. The transitions between states are determined by the actions of the system events. The states that can be reached as a result of event execution are defined by the current state. If we augment Event-B specification with the probabilities of reaching the next system state from the current one then we obtain a probabilistic automaton [3]. In case the events are mutually exclusive, i.e., only one event is enabled at each system state then the specification can be represented by a Markov chain. Otherwise, it corresponds to a Markov Decision process [7, 10, 21]. More specifically, it is a discrete time Markov process since we can use it to describe the states at certain instances of time.

The probabilistic model checking framework developed by Kwiatkowska et al. [11] supports verification of Discrete-Time Markov Chains (DTMC) and Markov Decision Processes (MDP). The framework has a mature tool support – the PRISM model checker [15]

The PRISM modelling language is a high-level state-based language. It relies on the Reactive Modules formalism of Alur and Henzinger [3]. PRISM supports the use of constants and variables that can be integers, doubles (real numbers) and Booleans. Constans are used, for instance, to define the probabilities associated with variable updates. The variables in PRISM are finite-ranged and strongly typed. They can be either local or global. The definition of an initial value of a variable is usually attached to its declaration. The state space of a PRISM model is defined by the set of all variables, both global and local.

In general, a PRISM specification looks as follows:

dtmc
const double $p_{11} = \ldots;$
$\ldots$
global $v : Type$ **init** $\ldots;$
$\ldots$
module M_1
$v_1 : Type$ **init** $\ldots;$
$[]\ g_{11} \rightarrow p_{11} : act_{11} + \cdots + p_{1n} : act_{1n};$
$[sync]\ g_{12} \rightarrow q_{11} : act'_{11} + \cdots + q_{1m} : act'_{1m};$
$\ldots$
endmodule
module M_2
$v_2 : Type$ **init** $\ldots;$
$[sync]\ g_{21} \rightarrow p_{21} : act_{21} + \cdots + p_{2k} : act_{2k};$
$[]\ g_{22} \rightarrow q_{21} : act'_{21} + \cdots + q_{2l} : act'_{2l};$
$\ldots$
endmodule
$\ldots.$

A system specification in PRISM is constructed as a parallel composition of modules. Modules work in parallel. They can be independent of each other or interact with each other. Each module has a number of local variables $v_1,\ v_2$ and a set of guarded commands that determine its dynamic behaviour. The guarded commands can have names. Similarly to the events of Event-B, a guarded command can be executed if its guard evaluates to $TRUE$. If several guarded commands are enabled then the choice between them can be non-deterministic in case of MDP or probabilistic (according to the uniform distribution) in case of DTMC. In general, the body of a guarded command is a probabilistic choice between deterministic assignments.

The guarded commands define not only the dynamic behaviour of a standalone module but can also be used to define syncronisation between modules. If several modules synchronise then each of them should contain a command defining the syncronisation condition. These commands should have identical names. For instance, in our general PRISM specification shown above, the modules M_1 and M_2 synchronise. They contain the corresponding guarded commands labelled with the name $sync$. The guarded commands that provide synchronisation with other modules cannot modify the global variables. This allows to avoid read-write and write-write conflicts on the global variables.

Converting Event-B model into a PRISM model is rather straightforward. When converting Event-B model into its counterpart, we need to restrict the types of variables and constants to the types supported by PRISM. The invariants that describe system properties can be represented as a number of temporal logic formulas in a list of properties of the model and then can be verified by PRISM if needed. While converting events into the PRISM guarded commands, we identify four classes of events: initilisation events, events with parallel deterministic assignment, non-deterministic assignment and parallel non-deterministic assignment. The conversion of an Event-B event to a PRISM guarded command depends on its class:

- The initialisation events are used to form the initialisation part of the corresponding variable declaration. Hence the initialisation does not have a corresponding guarded command in PRISM;

- An event with a parallel deterministic assignment

  $$evt \mathrel{\widehat{=}} \textbf{when } g \textbf{ then } x := x_1 \parallel y := y_1 \parallel z := z_1 \textbf{ end}$$

  can be represented by the following guarded command in PRISM:

  $$[]\ g \rightarrow (x' = x_1)\ \&\ (y' = y_1)\ \&\ (z' = z_1)$$

  Here $\&$ denotes the parallel composition;

- An event with a non-deterministic assignment

$$evt \mathrel{\widehat{=}} \textbf{when } g \textbf{ then } x :\in \{x_1, \ldots x_n\} \textbf{ end}$$

can be represented as

$$[]\ g \rightarrow p_1 : (x' = x_1) + \cdots + p_n : (x' = x_n)$$

where $p_1, ..., p_n$ are defined according to a certain probability distribution;

- An event with a parallel non-deterministic assignment

$$evt \mathrel{\widehat{=}} \textbf{when } g \textbf{ then } x :\in \{x_1, \ldots x_n\} \parallel y :\in \{y_1, \ldots y_m\} \parallel z :\in \{z_1, \ldots z_k\} \textbf{ end}$$

can be represented using the PRISM synchronisation mechanism. It corresponds to a set of the guarded commands modelling syncronisation. These commands have the identical guards. Their bodies are formed from the assignments used in the parallel composition of the Event-B action.

module X

$x : Type$ **init** $\ldots$;

$[name]\ g \rightarrow p_1 : (x' = x_1) + \cdots + p_n : (x' = x_n)$;

endmodule

module Y

$y : Type$ **init** $\ldots$;

$[name]\ g \rightarrow q_1 : (y' = y_1) + \cdots + q_m : (y' = y_m)$;

endmodule

module Z

$z : Type$ **init** $\ldots$;

$[name]\ g \rightarrow r_1 : (z' = z_1) + \cdots + z_k : (z' = z_k)$;

endmodule.

To demonstrate the convertion of an Event-B specification into a PRISM specification, below we present an excerpt from the PRISM counterpart of the TMR specification. Here we assume that at each iteration step a module successfully produces a result with a constant probability p.

$System_{TMR}$
module $module_1$

$m_1 : [0..1]$ **init** 1;
$f : [0..1]$ **init** 0;

$[m]\ (phase = 0)\ \&\ (m_1 = 1)\ \&\ (f = 0) \rightarrow$
$p : (m_1' = 1)\ \&\ (f' = 1) +$
$+ (1 - p) : (m_1' = 0)\ \&\ (f' = 1);$

$[m]\ (phase = 0)\ \&\ (m_1 = 0)\ \&\ (f = 0) \rightarrow (f' = 1);$
$[]\ (phase = 0)\ \&\ (f = 1) \rightarrow (phase' = 1)\ \&\ (f' = 0);$

endmodule

module $module_2$ ...

module $module_3$ ...

module $voter$

res : ***bool*** **init** $true$;
$[]\ (phase = 1)\ \&\ (m_1 + m_2 + m_3 > 1) \rightarrow (phase' = 0);$
$[]\ (phase = 1)\ \&\ (m_1 + m_2 + m_3 \leq 1) \rightarrow (res' = false);$

endmodule

While converting an Event-B model into PRISM we could have modelled the parallel work of the system modules in the same way as we have done it in the Event-B specifications, i.e., using non-determinism to represent parallel behaviour and explicitly modelling the phases of system execution. However, we can also directly use the synchronisation mechanism of PRISM because all the modules update only their local variables and no read-write conflict can occur. This solution is presented in the excerpt above. In the $System_{TMR}$ specification, the guarded commands of the modules $module_1$, $module_2$ and $module_3$ are synchronised (as designated by the m label). In the $module_1$ we additionally update the global variable $phase$ to model transition of the system to the voting phase.

5 Reliability Assessment via Probabilistic Model Checking

In engineering, reliability [20, 14] is generally measured by the probability that an entity $\mathcal{E}$ can perform a required function under given conditions for the time interval $[0, t]$:

$$R(t) = P[\mathcal{E} \textit{ not failed over time } [0, t]].$$

The analysis of the abstract and refined specification shows that we can clearly distinguish between two classes of systems states: operating and failed. In our case the operating states are the states where the variable res has the value $TRUE$. Correspondingly, the failed states are the states where the variable res has the value $FALSE$. While the system is in an operating state, it continues to iterate. When the system fails, it deadlocks. Therefore, we define *reliability of the system as a probability of staying opertaional for a given number of iterations*.

Let $\mathcal{T}$ be the random variable measuring the number of iterations before the deadlock is reached and $F(t)$ its cumulative distribution function. Then clearly $R(t)$ and $F(t)$ are related as follows:

$$R(t) = P[\mathcal{T} > t] = 1 - P[\mathcal{T} \leq t] = 1 - F(t).$$

It is straightforward to see that our definition corresponds to the standard definition of reliability given above. Now let us discuss how to employ PRISM model checking to assess system reliability.

While analysing a PRISM model, we define a number of temporal logic properties and systematically check the model to verify them. Properties of discrete-time PRISM models, i.e, DTMC and MDP, are expressed formally in the probabilistic computational tree logic [9]. The PRISM property specification language supports a number of different types of properties. For example, the $\mathbf{P}$ operator is used to refer to the probability of a certain event occurrence.

Since we are interested in assessment of system reliability, we have to verify *invariant* properties, i.e., properties maintained by the system globally. In the PRISM property specification language, the operator $\mathbf{G}$ is used inside the operator $\mathbf{P}$ to express properties of such type. In general, the property

$$\mathbf{P}_{=?}[\mathbf{G} \leq t\ prop]$$

returns a probability that the predicate $prop$ remains $TRUE$ in all states within the period of time t.

To evaluate reliability of a system, we have to assess a probability of system staying operational within time t. We define a predicate $\mathcal{OP}$ that defines a subset of all system states where the system is operational. Then, the PRISM property

$$\mathbf{P}_{=?}[\mathbf{G} \leq T\ \mathcal{OP}] \tag{1}$$

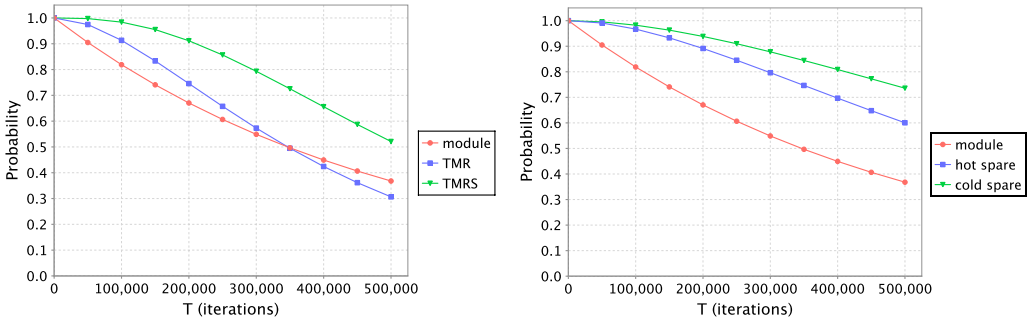
gives us the probability that the system will stay operational during the first T iterations, i.e, it is a probability that any state in which the system will be during this time belongs to the subset of operational states. In other words, the property (1) defines the reliability function of the system.

Let us return to our examples. As we discussed previously, the operational states of our systems are defined by the predicate $res = true$, i.e., $\mathcal{OP} \mathrel{\widehat{=}} res = true$. Then the PRISM property

$$\mathbf{P}_{=?}[\mathbf{G} \leq T\ (res = true)] \tag{2}$$

denotes the reliability of our systems within time T.

To evaluate reliability of our refinement alternative, let us assume that a module produces a result successfully with the probability p equal to 0.999998. In Figure 4 we present the results of analysis of reliability up to 500000 iterations. Figure 4 (a) shows the comparative results between single-module and both of TMR systems. The results show that the triple modular redundant system with a spare always gives better reliability. Note that using the simple TMR arrangement is better comparing to a single-module only up to approximately 350000 iterations. In Figure 4 (b) we compare single-module and standby spare arrangements. The results clearly indicate that the better reliability is provided by the dynamic redundancy systems and that using of the cold spare arrangement is always more reliable.



(a) (b)

Figure 4: Resulting Reliabilities

It would be interesting to evaluate precision of the results obtained by the model checking with PRISM. For our case study it is possible to derive analytical representations of reliability functions, which can be used for comparison with verification results of property (2). It is well-known that the reliability of a single-module system is $R_M(t) = p^t$ and it is easy to show that the reliability of TMR system, consits of three identical modules, is

$$R_{TMR}(t) = R_M^3(t) + 3R_M^2(t)(1 - R_M(t)) = \\ = 3R_M^2(t) - 2R_M^3(t) = 3p^{2t} - 2p^{3t}.$$

Indeed, we can also calculate that the standby spare arrangement with a faulty detector has the resulting reliability

$$R_{HSS} = 1 - (1 - p^t)^2$$

for the hot spare, and the reliability

$$R_{CSS} = p^t(1 + t(1 - p))$$

for the cold spare module. Finally, for the TMR arrangement, with a spare, the resulting reliability is

$$R_{TMRS} = (6t - 8)p^{3t} - 6tp^{3t-1} + 9p^{2t}.$$

It is easy to verify that the results obtained by the model checking are identical to those can be calculated from the formulas presented above. This fact demonstrates the feasibility of using the PRISM model checker for reliability assessment.

6 Conclusion

In this paper we have proposed an approach to integrating reliability assessment into the refinement process. The proposed approach enables reliability assessment at early design phases that allows the designers to evaluate reliability of different design alternatives already at the development phase.

Our approach integrates two frameworks: refinement in Event-B and probabilistic model checking. Event-B supported by the RODIN tool platform provides us with a suitable framework for development of complex industrial-size systems. By integrating probabilistic verification supported by PRISM model checker we open a possibility to reason about non-functional system requirements in the refinement process.

The Event-B framework has been extended by Hallerstede and Hoang [8] to take into account probabilistic behaviour. They introduce qualitative probabilistic choice operator to reason about almost certain termination. This operator attempts to bound demonic non-determinism that, for instance, allows to demonstrate convergence of certain protocols. However, this approach is not suitable for reliability assessment since explicit quantitative representation of reliability would not be supported.

Kwiatkowska et al. [11] proposed an approach to assessing dependability of control systems using continuous time Markov chains. The general idea is similar to ours – to formulate reliability as a system property to be verified. However, this approach aims at assessing reliability of already developed system. In our approach reliability assessment proceeds hand-in-hand with system development.

The similar topic in the context of refinement calculus has been explored previously by Morgan et al. [13, 12]. In this approach the probabilistic refinement was used to assess system dependability. However, this work does not have the corresponding tool support, so the use of this approach in industrial practice might be cumbersome. In our approach we see a great benefit in integrating frameworks that have mature tool support [18, 15].

When using model checking we need to validate whether the analysed model represents the behaviour of the real system accurately enough. For example, the validation can be done if we demonstrate that model checking provides a good approximation of the corresponding algebraic solutions. In this paper we deliberately chosen the examples for which algebraic solutions can be provided. The experiments have demonstrated that the results obtained by model checking accurately match the algebraic solutions.

In our future work it would be interesting to further explore the connection between Event-B modeling and dependability assessment. In particular, an additional study are required to establish a formal basis for converting all types of non-deterministic assignments into the probabilistic ones. Furthermore, it would be interesting to explore the topic of probabilistic data refinement in connection with dependability assessment.

References

[1] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habiras, editor, *First Conference on the B method*, pages 169–190. IRIN Institut de recherche en informatique de Nantes, 1996.

[2] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[3] R. Alur and T. Henzinger. Reactive modules. In *Formal Methods in System Design*, pages 7–48, 1999.

[4] D. Craigen, S. Gerhart, and T.Ralson. Case study: Paris metro signaling system. In *IEEE Software*, pages 32–35, 1994.

[5] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[6] EU-project DEPLOY. `http://www.deploy-project.eu/`.

[7] W. Feller. *An Introduction to Probability Theory and its Applications*, volume 1. John Wiley & Sons, 1967.

[8] S. Hallerstede and T. S. Hoang. Qualitative probabilistic modelling in Event-B. In J. Davies and J. Gibbons, editors, *IFM 2007, LNCS 4591*, pages 293–312, 2007.

[9] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. In *Formal Aspects of Computing*, pages 512–535, 1994.

[10] J. G. Kemeny and J. L. Snell. *Finite Markov Chains*. D. Van Nostrand Company, 1960.

[11] M. Kwiatkowska, G. Norman, and D. Parker. Controller dependability analysis by probabilistic model checking. In *Control Engineering Practice*, pages 1427–1434, 2007.

[12] A. K. McIver and C. C. Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Springer, 2005.

[13] A. K. McIver, C. C. Morgan, and E. Troubitsyna. The probabilistic steam boiler: a case study in probabilistic data refinement. In J. Grundy, M. Schwenke, and T. Vickers, editors, *Proc. International Refinement Workshop, ANU, Canberra*. Springer-Verlag, 1998.

[14] P. D. T. O'Connor. *Practical Reliability Engineering, 3rd ed.* John Wiley & Sons, 1995.

[15] PRISM probabilistic model checker. http://www.prismmodelchecker.org.

[16] Rigorous Open Development Environment for Complex Systems (RODIN). IST FP6 STREP project, http://rodin.cs.ncl.ac.uk/.

[17] Rigorous Open Development Environment for Complex Systems (RODIN). Deliverable D7, Event-B Language, http://rodin.cs.ncl.ac.uk/.

[18] RODIN Event-B platform. http://www.event-b.org/.

[19] N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[20] A. Villemeur. *Reliability, Availability, Maintainability and Safety Assessment*. John Wiley & Sons, 1995.

[21] D. J. White. *Markov Decision Processes*. John Wiley & Sons, 1993.

Appendix

MACHINE System

VARIABLES

`res`

INVARIANTS

`inv1` : $res \in BOOL$

EVENTS

Initialisation

begin

`act1` : $res := TRUE$

end

Event *output* $\widehat{=}$

when

`grd1` : $res = TRUE$

then

`act1` : $res :\in BOOL$

end

END

MACHINE System_TMR

REFINES System

SEES cnt

VARIABLES

m1

m2

m3

phase

flag1

flag2

flag3

res

INVARIANTS

inv1 : $m1 \in \{0, 1\}$

inv2 : $m2 \in \{0, 1\}$

inv3 : $m3 \in \{0, 1\}$

inv4 : $phase \in PHASES$

inv5 : $flag1 \in \{0, 1\}$

inv6 : $flag2 \in \{0, 1\}$

inv7 : $flag3 \in \{0, 1\}$

inv8 : $m1 + m2 + m3 > 1 \Rightarrow res = TRUE$

EVENTS

Initialisation

begin

act1 : $m1 := 1$

act2 : $m2 := 1$

act3 : $m3 := 1$

act4 : $phase := reading$

act5 : $flag1 := 1$

act6 : $flag2 := 1$

act7 : $flag3 := 1$

act8 : $res := TRUE$

end

Event *module_ok1* $\widehat{=}$

when

grd1 : $m1 = 1$
grd2 : $flag1 = 1$
grd3 : $phase = reading$

then

act1 : $m1 :\in \{0, 1\}$
act2 : $flag1 := 0$

end

Event *module_failed1* $\widehat{=}$

when

grd1 : $m1 = 0$
grd2 : $flag1 = 1$
grd3 : $phase = reading$

then

act1 : $flag1 := 0$

end

Event *module_ok2* $\widehat{=}$

when

grd1 : $m2 = 1$
grd2 : $flag2 = 1$
grd3 : $phase = reading$

then

act1 : $m2 :\in \{0, 1\}$
act2 : $flag2 := 0$

end

Event *module_failed2* $\widehat{=}$

when

grd1 : $m2 = 0$
grd2 : $flag2 = 1$

grd3 : $phase = reading$

then

act1 : $flag2 := 0$

end

Event *moudle_ok3* $\widehat{=}$

when

grd1 : $m3 = 1$

grd2 : $flag3 = 1$

grd3 : $phase = reading$

then

act1 : $m3 :\in \{0, 1\}$

act2 : $flag3 := 0$

end

Event *module_failed3* $\widehat{=}$

when

grd1 : $m3 = 0$

grd2 : $flag3 = 1$

grd3 : $phase = reading$

then

act1 : $flag3 := 0$

end

Event *synchr* $\widehat{=}$

when

grd1 : $flag1 = 0$

grd2 : $flag2 = 0$

grd3 : $flag3 = 0$

grd4 : $phase = reading$

then

act1 : $phase := voting$

end

Event *voter_ok* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $m1 + m2 + m3 > 1$

grd3 : $phase = voting$

then

act1 : $flag1 := 1$

act2 : $flag2 := 1$

act3 : $flag3 := 1$

act4 : $phase := reading$

end

Event *voter_nok* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $m1 + m2 + m3 \leq 1$

grd3 : $phase = voting$

then

act1 : $res := FALSE$

end

END

MACHINE System_HSS

REFINES System

SEES cnt1

VARIABLES

`m1`

`m2`

`phase`

`flag1`

`flag2`

`res`

`m`

INVARIANTS

`inv1` : $m1 \in \{0, 1\}$

`inv2` : $m2 \in \{0, 1\}$

`inv3` : $phase \in PHASES$

`inv4` : $flag1 \in \{0, 1\}$

`inv5` : $flag2 \in \{0, 1\}$

`inv6` : $m \in \{0, 1\}$

`inv7` : $m1 + m2 > 0 \Rightarrow m = 1$

`inv8` : $m = 1 \Rightarrow res = TRUE$

EVENTS

Initialisation

begin

`act1` : $m1 := 1$

`act2` : $m2 := 1$

`act3` : $phase := in$

`act4` : $flag1 := 1$

`act5` : $flag2 := 1$

`act6` : $res := TRUE$

`act7` : $m := 1$

end

Event *module_ok1* $\widehat{=}$

when

grd1 : $m1 = 1$

grd2 : $flag1 = 1$

grd3 : $phase = in$

then

act1 : $m1 :\in \{0, 1\}$

act2 : $flag1 := 0$

end

Event *module_ok2* $\widehat{=}$

when

grd1 : $m2 = 1$

grd2 : $flag2 = 1$

grd3 : $phase = in$

then

act1 : $m2 :\in \{0, 1\}$

act2 : $flag2 := 0$

end

Event *module_failed2* $\widehat{=}$

when

grd1 : $m2 = 0$

grd2 : $flag2 = 1$

grd3 : $phase = in$

then

act1 : $flag2 := 0$

end

Event *synchr* $\widehat{=}$

when

grd1 : $flag1 = 0$

grd2 : $flag2 = 0$

grd3 : $phase = in$

then

act1 : $phase := det$

end

Event *detection_ok1* $\widehat{=}$

when

grd1 : $m1 = 1$

grd2 : $phase = det$

then

act1 : $m := m1$

act2 : $flag1 := 1$

act3 : $flag2 := 1$

act4 : $phase := out$

end

Event *detection_ok2* $\widehat{=}$

when

grd1 : $m1 = 0$

grd2 : $m2 = 1$

grd3 : $phase = det$

then

act1 : $m := m2$

act2 : $flag2 := 1$

act3 : $phase := out$

end

Event *detection_nok* $\widehat{=}$

when

grd1 : $m1 = 0$

grd2 : $m2 = 0$

grd3 : $phase = det$

then

act1 : $m := 0$

act2 : $phase := out$

end

Event *output_ok* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $m = 1$

grd3 : $phase = out$

then

act1 : $phase := in$

end

Event *output_nok* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $m = 0$

grd3 : $phase = out$

then

act1 : $res := FALSE$

end

END

MACHINE System_CSS

REFINES System

SEES cnt1

VARIABLES

m1

m2

phase

flag1

flag2

m

res

INVARIANTS

inv1 : $m1 \in \{0, 1\}$

inv2 : $m2 \in \{0, 1\}$

inv3 : $phase \in PHASES$

inv4 : $flag1 \in \{0, 1\}$

inv5 : $flag2 \in \{0, 1\}$

inv6 : $m \in \{0, 1\}$

inv7 : $m1 + m2 > 0 \Rightarrow m = 1$

inv8 : $m = 1 \Rightarrow res = TRUE$

inv10 : $flag1 = 1 \Rightarrow (flag2 = 0 \wedge m2 = 1)$

inv11 : $flag2 = 1 \Rightarrow (flag1 = 0 \wedge m1 = 0)$

EVENTS

Initialisation

begin

act1 : $m1 := 1$

act2 : $m2 := 1$

act3 : $flag1 := 1$

act4 : $flag2 := 0$

act5 : $phase := in$

act6 : $m := 1$

act7 : $res := TRUE$

end

Event *module1* $\widehat{=}$

when

grd1 : $m1 = 1$

grd2 : $flag1 = 1$

grd3 : $phase = in$

then

act1 : $m1 :\in \{0, 1\}$

act2 : $phase := det$

end

Event *module2* $\widehat{=}$

when

grd1 : $m2 = 1$

grd2 : $flag2 = 1$

grd3 : $phase = in$

then

act1 : $m2 :\in \{0, 1\}$

act2 : $phase := det$

end

Event *detection_ok1* $\widehat{=}$

when

grd1 : $m1 = 1$

grd2 : $phase = det$

then

act1 : $m := m1$

act2 : $phase := out$

end

Event *detection_nok1* $\widehat{=}$

when

grd1 : $m1 = 0$

grd2 : $flag1 = 1$

grd3 : $phase = det$

then

act1 : $flag1 := 0$

act2 : $flag2 := 1$

act3 : $phase := in$

end

Event *detection_ok2* $\widehat{=}$

when

grd1 : $m2 = 1$

grd2 : $flag2 = 1$

grd3 : $phase = det$

then

act1 : $m := m2$

act2 : $phase := out$

end

Event *detection_nok2* $\widehat{=}$

when

grd1 : $m2 = 0$

grd2 : $flag2 = 1$

grd3 : $phase = det$

then

act1 : $m := 0$

act2 : $flag2 := 0$

act3 : $phase := out$

end

Event *output_ok* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $m = 1$

grd3 : $phase = out$

then

act1 : $phase := in$

end

Event *output_nok* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $m = 0$

grd3 : $phase = out$

then

act1 : $res := FALSE$

end

END

MACHINE System_TMRS

REFINES System

SEES cnt

VARIABLES

m1

m2

m3

m4

phase

flag1

flag2

flag3

flag4

err

res

INVARIANTS

inv1 : $m1 \in \{0, 1\}$

inv2 : $m2 \in \{0, 1\}$

inv3 : $m3 \in \{0, 1\}$

inv4 : $m4 \in \{0, 1\}$

inv5 : $phase \in PHASES$

inv6 : $flag1 \in \{0, 1\}$

inv7 : $flag2 \in \{0, 1\}$

inv8 : $flag3 \in \{0, 1\}$

inv9 : $flag4 \in \{0, 1\}$

inv10 : $err \in 0 \,..\, 4$

inv11 : $(m1 + m2 + m3 > 1) \wedge (err = 0) \Rightarrow res = TRUE$

inv12 : $(m1 + m2 + m3 + m4 > 1) \wedge (err \neq 0) \Rightarrow res = TRUE$

inv13 : $err = 1 \Rightarrow m1 = 0$

inv14 : $err = 2 \Rightarrow m2 = 0$

inv15 : $err = 3 \Rightarrow m3 = 0$

EVENTS

Initialisation

begin

act1 : $m1 := 1$

act2 : $m2 := 1$

act3 : $m3 := 1$

act4 : $m4 := 1$

act5 : $phase := reading$

act6 : $flag1 := 1$

act7 : $flag2 := 1$

act8 : $flag3 := 1$

act9 : $flag4 := 0$

act10 : $err := 0$

act11 : $res := TRUE$

end

Event *module_ok1* $\widehat{=}$

when

grd1 : $m1 = 1$

grd2 : $flag1 = 1$

grd3 : $phase = reading$

then

act1 : $m1 :\in \{0, 1\}$

act2 : $flag1 := 0$

end

Event *module_failed1* $\widehat{=}$

when

grd1 : $m1 = 0$

grd2 : $flag1 = 1$

grd3 : $phase = reading$

then

act1 : $flag1 := 0$

end

Event *module_ok2* $\widehat{=}$

when

grd1 : $m2 = 1$

grd2 : $flag2 = 1$

grd3 : $phase = reading$

then

act1 : $m2 :\in \{0, 1\}$

act2 : $flag2 := 0$

end

Event *module_failed2* $\widehat{=}$

when

grd1 : $m2 = 0$

grd2 : $flag2 = 1$

grd3 : $phase = reading$

then

act1 : $flag2 := 0$

end

Event *module_ok3* $\widehat{=}$

when

grd1 : $m3 = 1$

grd2 : $flag3 = 1$

grd3 : $phase = reading$

then

act1 : $m3 :\in \{0, 1\}$

act2 : $flag3 := 0$

end

Event *module_failed3* $\widehat{=}$

when

grd1 : $m3 = 0$

grd2 : $flag3 = 1$

grd3 : $phase = reading$

then

act1 : $flag3 := 0$

end

Event *module_ok4* $\widehat{=}$

when

grd1 : $m4 = 1$

grd2 : $flag4 = 1$

grd3 : $phase = reading$

then

act1 : $m4 :\in \{0, 1\}$

act2 : $flag4 := 0$

end

Event *module_failed4* $\widehat{=}$

when

grd1 : $m4 = 0$

grd2 : $flag4 = 1$

grd3 : $phase = reading$

then

act1 : $flag4 := 0$

end

Event *synchr* $\widehat{=}$

when

grd1 : $flag1 = 0$

grd2 : $flag2 = 0$

grd3 : $flag3 = 0$

grd4 : $flag4 = 0$

grd5 : $phase = reading$

then

act1 : $phase := voting$

end

Event *voter_ok* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $err = 0$

grd3 : $m1 + m2 + m3 = 3$

grd4 : $phase = voting$

then

act1 : $flag1 := 1$

act2 : $flag2 := 1$

act3 : $flag3 := 1$

act4 : $phase := reading$

end

Event *voter_ok1* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $err = 0$

grd3 : $m1 + m2 + m3 = 2$

grd4 : $m1 = 0$

grd5 : $phase = voting$

then

act1 : $err := 1$

act2 : $flag2 := 1$

act3 : $flag3 := 1$

act4 : $flag4 := 1$

act5 : $phase := reading$

end

Event *voter_ok2* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $err = 0$

grd3 : $m1 + m2 + m3 = 2$

grd4 : $m2 = 0$

grd5 : $phase = voting$

then

act1 : $err := 2$

act2 : $flag1 := 1$

act3 : $flag3 := 1$

act4 : $flag4 := 1$

act5 : $phase := reading$

end

Event *voter_ok3* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $err = 0$

grd3 : $m1 + m2 + m3 = 2$

grd4 : $m3 = 0$

grd5 : $phase = voting$

then

act1 : $err := 3$

act2 : $flag1 := 1$

act3 : $flag2 := 1$

act4 : $flag4 := 1$

act5 : $phase := reading$

end

Event *voter_nok* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $err = 0$

grd3 : $m1 + m2 + m3 \leq 1$

grd4 : $phase = voting$

then

act1 : $res := FALSE$

end

Event *voter'_ok1* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$
grd2 : $err = 1$
grd3 : $m2 + m3 + m4 > 1$
grd4 : $phase = voting$

then

act1 : $flag2 := 1$
act2 : $flag3 := 1$
act3 : $flag4 := 1$
act4 : $phase := reading$

end

Event *voter'_nok1* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$
grd2 : $err = 1$
grd3 : $m2 + m3 + m4 \leq 1$
grd4 : $phase = voting$

then

act1 : $res := FALSE$

end

Event *voter'_ok2* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$
grd2 : $err = 2$
grd3 : $m1 + m3 + m4 \geq 1$
grd4 : $phase = voting$

then

act1 : $flag1 := 1$
act2 : $flag3 := 1$

act3 : $flag4 := 1$

act4 : $phase := reading$

end

Event *voter'_nok2* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $err = 2$

grd3 : $m1 + m3 + m4 \leq 1$

grd4 : $phase = voting$

then

act1 : $res := FALSE$

end

Event *voter'_ok3* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $err = 3$

grd3 : $m1 + m2 + m4 \geq 1$

grd4 : $phase = voting$

then

act1 : $flag1 := 1$

act2 : $flag2 := 1$

act3 : $flag4 := 1$

act4 : $phase := reading$

end

Event *voter'_nok3* $\widehat{=}$

refines *output*

when

grd1 : $res = TRUE$

grd2 : $err = 3$

grd3 : $m1 + m2 + m4 \leq 1$

grd4 : $phase = voting$

then

act1 : $res := FALSE$

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 978-952-12-2263-4
ISSN 1239-1891