Anton Tarasyuk | Inna Pereverzeva | Elena Troubitsyna |
Timo Latvala | Laura Nummila

Formal Development and Assessment of a Reconfigurable On-board Satellite System

Turku Centre *for* Computer Science

TUCS Technical Report
No 1038, February 2012

Formal Development and Assessment of a Reconfigurable On-board Satellite System

Anton Tarasyuk

Åbo Akademi University, Department of Computer Science

Turku Centre for Computer Science

anton.tarasyuk@abo.fi

Inna Pereverzeva

Åbo Akademi University, Department of Computer Science,

Turku Centre for Computer Science

inna.pereverzeva@abo.fi

Elena Troubitsyna

Åbo Akademi University, Department of Computer Science

elena.troubitsyna@abo.fi

Timo Latvala

Space Systems Finland, Espoo, Finland

timo.latvala@ssf.fi

Laura Nummila

Space Systems Finland, Espoo, Finland

laura.nummila@ssf.fi

TUCS Technical Report

No 1038, February 2012

Abstract

Ensuring fault tolerance of satellite systems is critical for achieving goals of the space mission. Since the use of redundancy is restricted by the size and the weight of the on-board equipments, the designers need to rely on dynamic reconfiguration in case of failures of some components. In this paper we propose a formal approach to development of dynamically reconfigurable systems in Event-B. Our approach allows us to build the system that can discover possible reconfiguration strategy and continue to provide its services despite failures of its vital components. We integrate probabilistic verification to evaluate reconfiguration alternatives. Our approach is illustrated by a case study from aerospace domain.

Keywords: Formal modelling, fault tolerance, Event-B, refinement, probabilistic verification.

1 Introduction

Fault tolerance is an important characteristics of on-board satellite systems. Usually fault tolerance is achieved via redundancy. However, the use of (component) redundancy in spacecraft is restricted by the weight and volume constraints. System developers perform a careful cost-benefit analysis to minimise the use of spare modules yet achieve the required level of system reliability.

Nevertheless, despite such an analysis, Space System Finland has recently experienced a double-failure problem with a system that samples and packages scientific data in one of the operating satellites. The system consisted of two identical modules. When one of the subcomponents of the first module failed the system switched to the use of the second module. However, after a while a subcomponent of the spare has also failed, so it became impossible to produce scientific data. To not lose the entire mission, the company has invented solution that relied on healthy subcomponents of both modules and complex communication mechanism to restore functioning and resume production of scientific data. Obviously, a certain amount of data has been lost before the repair was deployed. This motivated our work on exploring *proactive* solutions for fault tolerance, i.e., planning and evaluating of scenarios implementing a seamless reconfiguration using a fine-grained redundancy.

In this paper we propose a formal approach to modelling and assessment of on-board reconfigurable systems. We generalise the ad-hoc solution created by the Space Systems Finland and propose an approach to formal development and assessment of the fault tolerant satellite systems. The essence of the modelling side of our approach is to start from abstract modelling functional goals that the system should achieve to remain operational and derive reconfigurable architecture by refinement in Event-B.

Event-B is a formal top-down development approach to correct-by-construction system development [2]. Currently, it is actively used within the EU project Deploy [10] to model dependable systems from various domain including space, automation, rail-ways and business information systems. The rigorous refinement process allows us to establish the precise relationships between component failures and goal reachability. The derived system architecture should not only satisfy functional requirements but also achieve its reliability objective. Moreover, since reconfiguration procedure requires additional inter-component communication, the developers should also verify that system performance remains acceptable. At the assessment site of our approach, we rely on the probabilistic extension of Event-B to verify reliability and performance properties. It is performed using PRISM model checker [12].

The main novelty of our work is in proposing an integrated approach to formal derivation of reconfigurable system architectures and probabilistic assessment of their reliability and performance. We believe that the proposed approach facilitates early exploration of design space and helps to build more redundancy-frugal systems that yet meet the desired reliability and performance requirements.

2 Reconfigurable Fault Tolerant Systems

2.1 Case Study: Data Processing Unit

As we have already mentioned in the previous section, our work is inspired by a solution proposed to circumvent double failure occurred in a currently operational on-board satellite system. The architecture of this system is similar the Data Processing Unit (DPU) – a subsystem of European Space Agency mission BepiColombo [3] that is under development now. Space Systems Finland is one of the providers for BepiColombo. The main goal of the mission is to carry various scientific measures to explore the planet Mercury. DPU is an important part of the Mercury Planetary Orbiter. It consists of four independent components (computers) responsible for receiving and processing data from four sensor units: SIXS-X (X-ray spectrometer), SIXS-P (particle spectrometer), MIXS-T (telescope) and MIXS-C (collimator).

The behaviour of DPU is managed by telecommands (TCs) received from the spacecraft and stored in a TC pool, which is structured as a circular buffer. With a predefined rate DPU periodically polls the buffer, decodes TC and performs the required actions. Processing of each TC results in producing telemtery (TM) that is stored in a TM pool. Both TC and TM packages follow a strict syntax defined by the European Space Agency's Packet Utilisation Standard [14]. As a result of decoding TC, DPU might produce a housekeeping report, switch to some mode or initiate/continue production of *scientific data*. Correspondingly, TM would constrain either housekeeping data, or acknowledgement of mode transition or scientific data. The main purpose of DPU is to ensure a required rate of producing TM containing scientific data. In this paper we focus on analysing this particular aspect of system behaviour. Hence in the rest of the paper TCs will correspond to telecommands requiring production of scientific data, while TM will designate packages containing scientific data.

2.2 Goal-Oriented Reasoning about Fault Tolerance

In this paper we use the notion of goal as a basis for reasoning about fault tolerance. Goals – the functional and non-functional objectives that the system should achieve – are often used to structure the requirements of dependable systems [11, 17].

Let $\mathcal{G}$ be a predicate that defines a goal and $\mathcal{M}$ be a system model. The system design should ensure that eventually goal is reached. Hence, while verifying the system, we should establish that

$$\mathcal{M} \models \Diamond \mathcal{G}.$$

The main idea of goal-oriented development is to decompose the high-level system goals into a set of subgoals. Essentially, subgoals define the intermediate stages of achieving a high-level goal. In the process of goal decomposition we associate system components with tasks – the lowest-level subgoals. A component is associated with a task if its functionality enables establishing the goal defined by the corresponding task.

For instance, in this paper we consider *"Produce Scientific TM"* as a goal of DPU. DPU sequentially enquires each of its four components to produce its part of scientific data. Each component acquires fresh scientific data from the corresponding sensor unit (SIXS-X, SIXS-P, MIXS-T or MIXS-C), preprocesses it and makes available to DPU that eventually forms the entire TM package. Thus, the goal can be decomposed into four similar tasks *"Sensor data production"*.

In general, we say that the goal $\mathcal{G}$ can be decomposed into a finite set of tasks:

$$\mathcal{T} = \{task_j \mid j \in 1..n \land n \in \mathbb{N}_1\},$$

Let also $\mathcal{C}$ be a finite set of components capable of performing some tasks form $\mathcal{T}$:

$$\mathcal{C} = \{comp_j \mid j \in 1..m \land m \in \mathbb{N}_1\},$$

where $\mathbb{N}_1$ is the set of positive integers. Then the relation Φ defined below associates components with the tasks:

$$\Phi \in \mathcal{T} \leftrightarrow \mathcal{C}, \text{ such that } \forall t \in \mathcal{T} \cdot \exists c \in \mathcal{C} \cdot \Phi(t, c),$$

where $\leftrightarrow$ designates a binary relation.

To reason about fault tolerance, we should take into account component unreliability. A failure of a component means that it cannot perform its associated task. Obviously, this might prevent system from reaching the targeted goals and as a consequence achieving the required level of reliability. Fault tolerance mechanisms employed to mitigate results of component failures rely on various forms of component redundancy. Usually, spacecraft have stringent limitations on the size and weight of the on-board equipment, hence high degree of redundancy is rarely present. Typically, components are either duplicated or triplicated. Let us consider a duplicated system that consists of two identical DPUs – DPU_A and DPU_B. As it was explained above, each DPU contains four components responsible for controlling the corresponding sensor.

Traditionally, the satellite systems are designed to implement the following simple redundancy scheme. Initially DPU_A is active, while DPU_B is a cold spare. DPU_A allocates tasks on its components to achieve the system goal $\mathcal{G}$ – processing of TC and producing TM. When some lower-level component of DPU_A fails, DPU_B is activated to achieve the goal $\mathcal{G}$. Failure of DPU_B results in failure of the overall system. However, let us observe that even though none of the DPUs can accomplish the overall goal $\mathcal{G}$ on its own, it might be the case that the components that remained operational can perform the entire set of tasks required to reach $\mathcal{G}$. This observation allows us to define the following dynamic reconfiguration strategy.

Initially DPU_A is active and is assigned to reach the goal $\mathcal{G}$. If some of its components fails resulting in a failure to execute one of four scientific tasks (let it be $task_j$), the spare DPU_B is activated and DPU_A is deactivated. DPU_B performs the $task_j$ and the consecutive tasks required to reach $\mathcal{G}$. It becomes fully responsible for achieving the goal $\mathcal{G}$ until some of its component fails. In this case, to remain operational, the system performs *dynamic reconfiguration*. Namely, it reactivates DPU_A and tries to assign the failed task to the corresponding component of DPU_A. If such a component is operational then DPU_A continues to execute the subsequent tasks until it encounters a failed component. Then the control is passed to DPU_B again. Obviously, the overall system stays operational until two identical components of both DPUs have failed.

We generalise the architecture of DPU by stating that essentially a system consists of a number of modules and each module consists of n components:

$$\mathcal{C} = \mathcal{C}_a \cup \mathcal{C}_b, \quad \text{where} \quad \mathcal{C}_a = \{a\_comp_j \mid j \in 1..n \land n \in \mathbb{N}_1\} \quad \text{etc.}$$

Each module relies on its components to achieve the tasks required to accomplish $\mathcal{G}$. Hence we associate the corresponding tasks from the set $\mathcal{T}$ with these components.

An introduction of redundancy allows us to associate not a single but several components with each task. We reformulate the goal reachability property as follows: a goal is reachable if there is at least one *operational* component associated with each task. Formally, it can be specified as:

$$\mathcal{M} \models \Box\,(\mathcal{O}_s \Rightarrow \Diamond\,\mathcal{G}), \quad \text{where} \quad \mathcal{O}_s \equiv \forall t \in \mathcal{T} \cdot (\exists c \in \mathcal{C} \cdot \varPhi(t,c) \land \mathcal{O}(c))$$

and $\mathcal{O}$ is a predicate over the set of components $\mathcal{C}$ such that $\mathcal{O}(c)$ evaluates to $TRUE$ if and only if the component c is operational.

2.3 Probabilistic Assessment

If a duplicated system with the dynamic reconfiguration achieves the desired reliability level, it might allow the designers to avoid module triplication. However, it also increases the amount of intercomponent communication that leads to decreasing the system performance. Hence, while deciding on fault tolerance strategy, it is important to consider not only reachability of functional goals but also their performance and reliability aspects.

In engineering, both reliability and performance are system quality attributes that are usually measured quantitatively . Specifically, reliability is the probability that the system/component remains operational under given conditions for a certain time interval. In terms of goal reachability the system remains operational until it is capable of reaching targeted goals. Hence to guarantee that system is capable of performing a required functions within a time interval t is it enough to verify that

$$\mathcal{M} \models \Box^{\leq t}\, \mathcal{O}_s. \tag{1}$$

However, due to possible component failures we usually cannot guarantee the absolute preservation of (1). Instead, to assess the reliability of a system, we need to show that the probability of preserving the property (1) is sufficiently high, i.e., it exceeds the required threshold. On the other hand, the system performance is a reward-based property that can be measured by the number of successfully achieved goals within a certain time period.

To quantitatively verify these quality attributes we formulate the following CSL (Continuous Stochastic Logic) formulas [9]:

$$\mathbf{P}_{=?}\{\mathbf{G} \leq t\ \mathcal{O}_s\} \quad \text{and} \quad \mathbf{R}(|goals|)_{=?}\{\mathbf{C} \leq t\,\}.$$

The formulas above are specified using PRISM notations. The operator **P** is used to refer to the probability of an event occurrence, **G** is an analogue of $\Box$, **R** is used to analyse the *expected values* of rewards specified in a model, while **C** specifies that the reward should be cumulated only up to a given time bound. Thus, the first formula is used to analyse how likely the system remains operational as time passes, while the second one is used to compute the expected number of achieved goals cumulated by the system over t time units.

In this paper we rely on formal modelling in Event-B to formally define the architecture of a dynamically reconfigurable system, and a probabilistic extension of Event-B to create models for assessing system reliability and performance. In the next section we briefly describe Event-B and its probabilistic extension.

3 Modelling in Event-B and Probabilistic Analysis

3.1 Modelling and Refinement in Event-B

The Event-B formalism – a variation of the B Method [1] – is a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. In Event-B, a system model is specified using the notion of an *abstract state machine* [2]. An abstract state machine encapsulates the model state represented as a collection of variables and defines operations on the state, i.e., it describes the *behaviour* of the modelled system. Usually, a machine has an accompanying component, called *context*, which may include user-defined carrier sets, constants and their properties given as a list of model axioms. In Event-B, the model variables are strongly typed by the constraining predicates. These predicates and the other important properties that must be preserved by the model constitute model *invariants* ($\mathcal{I}$).

The dynamic behaviour of the system is defined by the set of atomic *events*. Generally, an event has the following form:

$$e \mathrel{\widehat{=}} \textbf{any } a \textbf{ where } G_e \textbf{ then } R_e \textbf{ end},$$

where e is the event's name, a is the list of local variables, the *guard* G_e is a predicate over the local variables of the event and the state variables of the system. The body of the event is defined by the next-state relation R_e. In Event-B, R_e is defined by a *multiple* (possibly nondeterministic) assignment over the system variables. The guard defines the conditions under which the substitution can be performed, i.e., when the event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically. If an event does not have local variables, it can be described simply as:

$$e \mathrel{\widehat{=}} \textbf{when } G_e \textbf{ then } R_e \textbf{ end}.$$

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract system specification that nondeterministically models the most essential functional requirements. In a sequence of refinement steps we gradually reduce nondeterminism and introduce detailed design decisions. In particular, we can add new events, split events as well as replace abstract variables by their concrete counterparts, i.e., perform *data refinement*. When data refinement is performed, we should define *gluing invariants* as a part of the invariants of the refined machine. They define the relationship between the abstract and concrete variables. The proof of data refinement is often supported by supplying *witnesses* – the concrete values for the replaced abstract variables and parameters. Witnesses are specified in the event clause **with**.

The consistency of Event-B models, i.e., verification of well-formedness and invariant preservation as well as correctness of refinement steps are formally demonstrated by discharging the relevant proof obligations generated by the Rodin platform [13]. The platform provides an automated tool support for proving.

3.2 Augmenting Event-B Models with Probabilities

Next we briefly describe the idea behind translating Event-B into continuous time Markov chains – CTMC (the details can be found in [16]). To achieve this, we augment all events of the model with the information about the probability and duration of all the actions that may occur during their execution, and refine them by their probabilistic counterparts.

Let Σ be a state space of an Event-B model defined by all possible values of the system variables. We consider an event e as a binary relation on Σ, i.e., for any two states $\sigma, \sigma' \in \Sigma$:

$$e(\sigma, \sigma') \overset{def}{=} G_e(\sigma) \wedge R_e(\sigma, \sigma').$$

Definition 1. *The behaviour of an Event-B machine is fully defined by a transition relation $\rightarrow$:*

$$\frac{\sigma, \sigma' \in \Sigma \ \wedge \ \sigma' \in \bigcup\limits_{e \in \mathcal{E}_\sigma} \mathsf{after}(e)}{\sigma \rightarrow \sigma'},$$

where $\mathsf{before}(e) = \{\sigma \in \Sigma \mid \mathcal{I}(\sigma) \wedge G_e(\sigma)\}$, $\mathcal{E}_\sigma = \{e \in \mathcal{E} \mid \sigma \in \mathsf{before}(e)\}$ *and*
$\mathsf{after}(e) = \{\sigma' \in \Sigma \mid \mathcal{I}(\sigma') \wedge (\exists \sigma \in \Sigma \cdot \mathcal{I}(\sigma) \wedge G_e(\sigma) \wedge R_e(\sigma, \sigma'))\}$.

Furthermore, let us adopt the notation $\lambda_e(\sigma, \sigma')$ to denote the (exponential) transition rate from σ to σ' via the event e, where $\sigma \in \mathsf{before}(e)$ and $R_e(\sigma, \sigma')$. By augmenting all the event actions with transition rates, we can respectively modify Definition 3.2 as follows.

Definition 2. *The behaviour of a probabilistically augmented Event-B machine is defined by a transition relation $\xrightarrow{\Lambda}$:*

$$\frac{\sigma, \sigma' \in \Sigma \ \wedge \ \sigma' \in \bigcup\limits_{e \in \mathcal{E}_\sigma} \mathsf{after}(e)}{\sigma \xrightarrow{\Lambda} \sigma'},$$

where $\Lambda = \sum\limits_{e \in \mathcal{E}_\sigma} \lambda_e(\sigma, \sigma')$.

Definition 3.2 allows us to define the semantics of a probabilistically augmented Event-B model as a probabilistic transition system with the state space Σ, transition relation $\xrightarrow{\Lambda}$ and the initial state defined by model initialisation (for probabilistic models we require the initialisation to be deterministic). Clearly, such a transition system corresponds to a CTMC.

In the next section we demonstrate how to formally derive an Event-B model of the architecture of a reconfigurable system.

4 Deriving Fault Tolerant Architectures by Refinement in Event-B

The general idea behind our formal development is to start from an abstract goal modelling, decompose it into tasks and introduce an abstract representation of the goal execution flow. Such a model can be refined into different fault tolerant architectures. Subsequently these models are augmented with probabilistic data and used for the assessment.

4.1 Modelling Goal Reaching

Goal Modelling. Our initial specification abstractly models the process of reaching the goal. The progress of achieving the goal is modelled by the variable *goal* that obtains values from the enumerated set $STATUS = \{not\_reached, reached, failed\}$. Initially, the system is not assigned any goals to accomplish, i.e., the variable *idle* is equal $TRUE$. When the system becomes engaged in establishing the goal, *idle* obtains value $FALSE$ as modelled by the event *Activation*. In the process of accomplishing the goal, the variable *goal* might eventually change its value from *not_reached* to *reached* or $failed$, as modelled by the event *Body*. After the goal is reached the system becomes idle, i.e., a new goal can be assigned. The event $Finish$ defines such a behaviour. We treat the failure to achieve the goal as a permanent system failure. It is represented by the infinite stuttering defined in the event $Abort$.

```
Activation ≙
  when $idle = TRUE$
  then $idle := FALSE$
  end
Body ≙
  when $idle = FALSE \wedge goal = not\_reached$
  then $goal :\in STATUS$
  end
Finish ≙
  when $idle = FALSE \wedge goal = reached$
  then $goal, idle := not\_reached, TRUE$
  end
Abort ≙
  when $goal = failed$
  then $skip$
  end
```

Goal Decomposition. The aim of our first refinement step is to define the goal execution flow. We assume that the goal can be decomposed into n tasks. It can be achieved by a sequential execution one task after another. For simplicity, we assume that the id of each task is defined by the order of its execution. Initially, when a goal is chosen, none of the tasks is executed, i.e., the state of each task is "not defined" (designated by the constant value ND). After the execution, the state of a task might be changed to success or failure, represented by the constants OK and NOK correspondingly. Our refinement step is essentially data refinement that replaces the abstract variable *goal* with the new variable *task* defined as a total function between the task id and its state:

$$task \in 1..n \rightarrow STATE.$$

The events of the refined model can be found in Appendix A. They represent the process of sequential selection of one task after another until either all tasks are executed, i.e., the goal is reached, or execution of some task fails, i.e., goal is not achieved. Correspondingly, the guards ensure that either the goal reaching has not commenced yet or the execution of all previous task has been successful. The body of the events nondeterministically changes the state of the chosen task to OK or NOK. The following invariants define the properties of the task execution flow:

$$\forall l \cdot l \in 2 \,..\, n \wedge task(l) \neq ND \Rightarrow (\forall i \cdot i \in 1 \,..\, l-1 \Rightarrow task(i) = OK),$$
$$\forall l \cdot l \in 1 \,..\, n-1 \wedge task(l) \neq OK \Rightarrow (\forall i \cdot i \in l+1 \,..\, n \Rightarrow task(i) = ND).$$

They state that the goal execution can progress, i.e., a next task can be chosen for execution, only if none of the previously executed tasks failed and the subsequent tasks have not been executed yet.

From the requirements perspective, the refined model should guarantee that the system level goal remains achievable. This is ensured by the gluing invariants that establish the relationship between the abstract goal and tasks as shown below:

$$task[1\,..\,n] = \{OK\} \Rightarrow goal = reached,$$
$$(task[1\,..\,n] = \{OK, ND\} \vee task[1\,..\,n] = \{ND\}) \Rightarrow goal = not\_reached,$$
$$(\exists i \cdot i \in 1\,..\,n \wedge task(i) = NOK) \Rightarrow goal = failed.$$

Introducing Abstract Communication. The aim of our next refinement step is to introduce an abstract model of communication. We define a new variable *ct* that stores the id of the last achieved task. The value of *ct* is checked every time when a new task is to be chosen for execution. If task execution succeeds then *ct* is incremented. Failure to execute the task leaves *ct* unchanged and results only in the change of the status of the failed task to NOK. Essentially the refined model introduces an abstract communication via shared memory. The following gluing invariants allow us to prove the refinement:

$$ct > 0 \Rightarrow (\forall i \cdot i \in 1\,..\,ct \Rightarrow task(i) = OK),$$
$$ct < n \Rightarrow task(ct + 1) \in \{ND, NOK\},$$
$$ct < n - 1 \Rightarrow (\forall i \cdot i \in ct + 2\,..\,n \Rightarrow task(i) = ND).$$

They relate the values of *ct* with the id's and the statuses of the tasks.

As we discussed in Section 2, each task is executed by a separate component of a high-level module. Therefore, by substituting the id of a task with the id of the component executing it, i.e., performing a trivial data refinement with the gluing invariant

$$\forall i \in 1..n \cdot task(i) = comp(i),$$

we define a *non-redundant* architecture of the system. In other words, our current model implements the relation Φ as a function. Next we demonstrate how to introduce a fault tolerant architecture with different forms of redundancy. This corresponds to defining Φ as a relation. The use of formal modelling in Event-B allows us to mathematically prove that the system architecture specified by Φ is preserved by the system model. We demonstrate how to introduce either a triplicated architecture without a dynamic reconfiguration or duplicated architecture with a dynamic reconfiguration by refinement.

4.2 Introducing Reconfiguration Strategies

To define triplicated architecture with static reconfiguration, we define three identical modules A, B and C. Each module consists of n components executing corresponding tasks. We refine the abstract variable *task* by the three new variables *a_comp*, *b_comp* and *c_comp*:

$$a\_comp \in 1..n \rightarrow STATE,\ b\_comp \in 1..n \rightarrow STATE,\ c\_comp \in 1..n \rightarrow STATE.$$

To associate the tasks with the components of each module, we formulate a number of gluing invariants that essentially specify the relation Φ. Some of these invariants are shown below:

$$\begin{gathered}
\forall i \cdot i \in 1 \mathinner{..} n \land module = A \land a\_comp(i) = OK \Rightarrow task(i) = OK,\\
module = A \Rightarrow (\forall i \cdot i \in 1 \mathinner{..} n \Rightarrow b\_comp(i) = ND \land c\_comp(i) = ND),\\
\forall i \cdot i \in 1 \mathinner{..} n \land module = A \land a\_comp(i) \neq OK \Rightarrow task(i) = ND,\\
\forall i \cdot i \in 1 \mathinner{..} n \land module = B \land b\_comp(i) \neq OK \Rightarrow task(i) = ND,\\
\forall i \cdot i \in 1 \mathinner{..} n \land module = C \Rightarrow c\_comp(i) = task(i),\\
module = B \Rightarrow (\forall i \cdot i \in 1 \mathinner{..} n \Rightarrow c\_comp(i) = ND).
\end{gathered}$$

Here, a new variable $module \in \{A, B, C\}$ stores the id of the currently active module. The complete list of invariants can be found in Appendix A.

An alternative way to perform this refinement step is to introduce a duplicated architecture with dynamic reconfiguration. In this case, we assume that our system consists of two modules – A and B – defined in the same way as discussed above. We replace the abstract variable *task* with two new variables *a_comp* and *b_comp*. Below we give an excerpt from the definition of the gluing invariants:

$$\begin{gathered}
module = A \land ct > 0 \land a\_comp(ct) = OK \Rightarrow task(ct) = OK,\\
module = B \land ct > 0 \land b\_comp(ct) = OK \Rightarrow task(ct) = OK,\\
\forall i \cdot i \in 1 \mathinner{..} n \land a\_comp(i) = NOK \land b\_comp(i) = NOK \Rightarrow task(i) = NOK,\\
\forall i \cdot i \in 1 \mathinner{..} n \land a\_comp(i) = NOK \land b\_comp(i) = ND \Rightarrow task(i) = ND,\\
\forall i \cdot i \in 1 \mathinner{..} n \land b\_comp(i) = NOK \land a\_comp(i) = ND \Rightarrow task(i) = ND.
\end{gathered}$$

Essentially, the invariants define the behavioural patterns for executing the tasks according to dynamic reconfiguration scenario described in Section 2.

Since our goal is to study the fault tolerance aspect of the system architecture, in our modelling we have deliberately abstracted away from the representation of the details of the system behaviour. A significant number of functional requirements is formulated as gluing invariants in the Event-B model. As a result, to verify correctness of the models we had to discharge more than 500 proof obligations. Around 90% of them have been proved automatically by the Rodin platform and the rest have been proved manually in the Rodin interactive proving environment.

In this section, we have described the development for a generic system. However, such a development can be easily instantiated to formally derive fault tolerant architectures of DPU. The goal of DPU is to handle the scientific TCs by producing TM. This goal is decomposed into four tasks that define the production of data by the corresponding sensor units – SIXS-X, SIXS-P, MIXS-T and MIXS-C. Thus, for such a model we have four tasks ($n = 4$) and each task is independently handled by the corresponding computing component of DPU. Furthermore, the high-level modules A, B and C correspond to the three identical DPUs that control handling of scientific TCs – DPU_A, DPU_B and DPU_C, while functions *a_comp*, *b_comp* and *c_comp* represent statuses of their internal components.

From the functional point of view, both alternatives of the last refinement step are equivalent. Indeed, each of them models the process of reaching the goal by a fault tolerant system architecture. In the next section we will present a quantitative assessment of their reliability and performance aspects.

5 Quantitative Assessment of Reconfiguration Strategies

In this section we assess the proposed fault tolerant architectures – triplicated with static reconfiguration and duplicated with the dynamic reconfiguration. The scientific mission of BepiColombo on the orbit of the Mercury will last for one year with possibility to extend this period for another year. Therefore, we should assess the reliability of both architectural alternatives for this period of time. Clearly, the triplicated DPU is able to tolerate up to three DPU failures within the two-year period, while the use of a duplicated DPU with a dynamic reconfiguration allows the satellite to tolerate from one (in the worst case) to four (in the best case) failures of the components.

Obviously, the duplicated architecture with a dynamic configuration minimises volume and the weight of the on-board equipment. However, the dynamic reconfiguration requires additional inter-component communication that slows down the process of producing TM. Therefore, we need to carefully analyse the performance aspect as well. Essentially, we need to show that the duplicated system with the dynamic reconfiguration can also provide a sufficient amount of scientific TM within the two-year period.

To perform the probabilistic assessment of reliability and performance, we rely on two types of data:

- probabilistic data about lengths of time delays required by DPU components and sensor units to produce the corresponding parts of scientific data
- data about occurrence rates of possible failures of these components

It is assumed that all time delays are exponentially distributed. We refine the Event-B specifications obtained at the final refinement step by their

Table 1: Rates (time is measured by minutes)

TC access rate when the system is idle	λ	$\frac{1}{12\cdot 60}$	SIXS-P work rate	α_2	$\frac{1}{30}$
TM output rate when a TC is handled	μ	$\frac{1}{20}$	SIXS-P failure rate	β_2	$\frac{1}{10^6}$
Spare DPU activation rate (power on)	δ	$\frac{1}{10}$	MIXS-T work rate	α_3	$\frac{1}{30}$
DPUs "communication" rate	τ	$\frac{1}{5}$	MIXS-T failure rate	β_3	$\frac{1}{9\cdot 10^7}$
SIXS-X work rate	α_1	$\frac{1}{60}$	MIXS-C work rate	α_4	$\frac{1}{90}$
SIXS-X failure rate	β_1	$\frac{1}{8\cdot 10^7}$	MIXS-C failure rate	β_4	$\frac{1}{6\cdot 10^7}$

probabilistic counterparts. This is achieved via introducing probabilistic information into events and replacing all the local nondeterminism with the (exponential) race conditions. Such a refinement relies on the model transformation presented in Section 3. As a result, we represent the behaviour of Event-B machines by CTMCs. This allows us to use probabilistic symbolic model checker PRISM to evaluate reliability and performance of the proposed models.

The PRISM specifications are presented in Appendix B. Due to limitations of the PRISM modelling language (e.g., it does not provide support for defining relations), they slightly differ form the corresponding Event-B machines. The guidelines for Event-B to PRISM model transformation can be found in our previous work [15].

The results of quantitative verification performed by PRISM show that with probabilistic characteristics of DPU presented, in Table 1[1], both reconfiguration strategies lead to a similar level of system reliability and performance with insignificant advantage of the triplicated DPU. Thus, the reliability levels of both systems within the two-year period are approximately the same with the difference of just 0.003 at the end of this period (0.999 against 0.996). Furthermore, the use of two DPUs under dynamic reconfiguration allows the satellite to handle only 2 TCs less after two years of work – 1104 against 1106 returned TM packets in the case of the triplicated DPU. Clearly, the use of the duplicated architecture with dynamic reconfiguration to achieve the desired levels of reliability and performance is optimal for the considered system.

Finally, let us remark that the goal-oriented style of the reliability and performance analysis has significantly simplified the assessment of the architectural alternatives of DPU. Indeed, it allowed us to abstract away from the configuration of input and output buffers, i.e., to avoid modelling of the circular buffer as a part of the analysis.

[1]Provided information may differ form the characteristics of the real components. It is used merely to demonstrate how the required comparison of reliability/performance can be achieved

6 Conclusions and Related Work

In this paper we proposed a formal approach to development and assessment of fault tolerant satellite systems. We made two main technical contributions. On the one hand, we defined the guidelines for development of the dynamically reconfigurable systems. On the other hand, we demonstrated how to formally assess reconfiguration strategy and evaluate whether the chosen fault tolerance mechanism fulfils reliability and performance objectives. The proposed approach was illustrated by a case study – development and assessment of the reconfigurable DPU. We believe that our approach not only guarantees correct design of complex fault tolerance mechanisms but also facilitates finding suitable trade-offs between reliability and performance.

A large variety of aspects of the dynamic reconfiguration has been studied in the last decade. For instance, Wermelinger et al. [19] proposed a high-level language for specifying the dynamically reconfigurable architectures. They focus on modifications of the architectural components and model reconfiguration by the algebraic graph rewriting. In our work, we focused on the functional rather than structural aspect of reasoning about reconfiguration.

Significant research efforts are invested in finding suitable models of triggers for run-time adaptation. Such triggers monitor performance [5] or integrity [18] of the application and initiate reconfiguration when the desired characteristics are not achieved. In our work we perform the assessment of reconfiguration strategy at the development phase that allows us to rely on existing error detection mechanisms to trigger dynamic reconfiguration.

A number of researchers investigate self* techniques for designing adaptive systems that autonomously achieve fault tolerance, e.g., see [6, 7]. However, these approaches are characterised by a high degree of uncertainty in achieving fault tolerance that is unsuitable for the satellite systems.

An extensive body of research investigates quality of service characteristics of dynamically reconfigurable service-oriented systems. Among the most prominent works in the area is the approach proposed by Calinescu et al. [4]. It aims at defining the optimal configuration in terms of quality of service by assessing quality of service attributes of various service component that are available at run-time. We use the similar probabilistic verification techniques to assess reliability and performance. However, the satellite systems are more deterministic, i.e., the set of component is predefined. This allows us to assess the optimal reconfiguration strategy at the development phase and simplify reconfiguration process.

The work [8] proposes an interesting conceptual framework for establishing a link between changing environmental conditions, requirements and system-level goals. In our approach we were more interested in studying a formal aspect of dynamic reconfiguration.

In our future work we are planning to further study the properties of dynamic reconfiguration. It particular, it would be interesting to investigate dynamic reconfiguration in the presence of parallelism and complex component interdependencies.

References

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

[2] J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.

[3] BepiColombo. ESA Media Center, Space Science, online at http://www.esa.int/esaSC/SEMNEM3MDAF_0_spk.html.

[4] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS Management and Optimization in Service-Based Systems. volume 37, pages 387–409. IEEE Computer Society, 2011.

[5] M. Caporuscio, A. Di Marco, and P. Inverardi. Model-Based System Reconfiguration for Dynamic Performance Management. *J. Syst. Softw.*, 80:455–473, 2007.

[6] P.A. de C. Guerra, C.M.F. Rubira, and R. de Lemos. A Fault-Tolerant Software Architecture for Component-Based Systems. In *Architecting Dependable Systems*, pages 129–143. Springer, 2003.

[7] R. de Lemos, P.A. de Castro Guerra, and C.M.F. Rubira. A Fault-Tolerant Architectural Approach for Dependable Systems. *Software, IEEE*, 23(2):80–87, 2006.

[8] H.J. Goldsby, P. Sawyer, N. Bencomo, B.H.C. Cheng, and D. Hughes. Goal-Based Modeling of Dynamically Adaptive System Requirements. In *ECBS 2008, Engineering of Computer Based Systems*, pages 36–45, 2008.

[9] L. Grunske. Specification Patterns for Probabilistic Quality Properties. In *ICSE 2008, International Conference on Software Engineering*, pages 31–40. ACM, 2008.

[10] Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity (DEPLOY). IST FP7 IP Project, online at http://www.deploy-project.eu/.

[11] T. P. Kelly and R. A. Weaver. The Goal Structuring Notation – A Safety Argument Notation. In *The Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[12] G. Norman M. Kwiatkowska and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *CAV'11, International Conference on Computer Aided Verification*, pages 585–591. Springer, 2011.

[13] Rodin. Event-B Platform, online at http://www.event-b.org/.

[14] Space Engineering: Ground Systems and Operations – Telemetry and Telecommand Packet Utilization. ECSS-E-70-41A. ECSS Secretariat, 30.01.2003, online at http://www.ecss.nl/.

[15] A. Tarasyuk, E. Troubitsyna, and L. Laibinis. Quantitative Reasoning about Dependability in Event-B: Probabilistic Model Checking Approach. In *Dependability and Computer Engineering: Concepts for Software-Intensive Systems*, pages 459–472. IGI Global, 2011.

[16] Anton Tarasyuk, Elena Troubitsyna, and Linas Laibinis. Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B. In *IFM 2012, Integrated Formal Methods*. Springer-Verlag, 2012, to appear.

[17] Axel van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. In *RE*, pages 249–263, 2001.

[18] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe. An Automated Formal Approach to Managing Dynamic Reconfiguration. In *ASE 2006*, pages 18–22. Springer-Verlag, 2006.

[19] M. Wermelinger, A. Lopes, and J.L. Fiadeiro. A Graph Based Architectural Reconfiguration Language. *SIGSOFT Softw. Eng. Notes*, 26:21–32, 2001.

Appendix A

Event-B Development

CONTEXT ReconfigSystem_ctx

SETS

```
    STATE
    MODES
    MODULES
    STATUS
```

CONSTANTS

```
    ND
    OK
    NOK
    n
    main
    spare
    reconf
    A
    B
    C
    reached
    not_reached
    failed
    ss_trans
```

AXIOMS

axm1 : $partition(STATE, \{ND\}, \{OK\}, \{NOK\})$

axm2 : $n \in \mathbb{N}_1$

axm3 : $partition(MODES, \{main\}, \{spare\}, \{reconf\})$

axm4 : $partition(MODULES, \{A\}, \{B\}, \{C\})$

axm5 : $partition(STATUS, \{reached\}, \{not\_reached\}, \{failed\})$

axm6 : $ss\_trans \in \mathbb{P}(STATE) \rightarrow STATUS$

axm7 : $ss\_trans(\{OK\}) = reached$

axm8 : $ss\_trans(\{OK, ND\}) = not\_reached$

axm9 : $ss\_trans(\{ND\}) = not\_reached$

axm10 : $\forall S \cdot S \in \mathbb{P}(STATE) \wedge NOK \in S \Rightarrow ss\_trans(S) = failed$

END

MACHINE AbstractGoal
SEES ReconfigSystem_ctx
VARIABLES

```
    goal
    idle
```

INVARIANTS

inv1 : $goal \in STATUS$
inv2 : $idle \in BOOL$

EVENTS
Initialisation

```
    begin
        act1 : $goal := not\_reached$
        act2 : $idle := TRUE$
    end
Event  Activation $\widehat{=}$
    when
        grd1 : $idle = TRUE$
    then
        act1 : $idle := FALSE$
    end
Event  Body $\widehat{=}$
    when
        grd2 : $idle = FALSE$
        grd1 : $goal = not\_reached$
    then
        act1 : $goal :\in STATUS$
    end
Event  Finish $\widehat{=}$
    when
        grd1 : $goal = reached$
    then
        act1 : $goal := not\_reached$
        act2 : $idle := TRUE$
    end
Event  Abort $\widehat{=}$
    when
        grd1 : $goal = failed$
    then
        skip
    end
END
```

MACHINE ReconfigSystem
REFINES AbstractGoal
SEES ReconfigSystem_ctx
VARIABLES

    idle
    task

INVARIANTS

inv1 : $task \in 1 \,..\, n \rightarrow STATE$
inv2 : $(\exists i \cdot i \in 1 \,..\, n \wedge task(i) \neq ND) \Rightarrow idle = FALSE$
inv3 : $task[1 \,..\, n] = \{OK\} \Rightarrow goal = reached$
inv4 : $(task[1 \,..\, n] = \{OK, ND\} \vee task[1 \,..\, n] = \{ND\}) \Rightarrow goal = not\_reached$
inv5 : $(\exists i \cdot i \in 1 \,..\, n \wedge task(i) = NOK) \Rightarrow goal = failed$
inv6 : $\forall l \cdot l \in 2 \,..\, n \wedge task(l) \neq ND \Rightarrow (\forall i \cdot i \in 1 \,..\, l-1 \Rightarrow task(i) = OK)$
inv7 : $\forall l \cdot l \in 1 \,..\, n-1 \wedge task(l) \neq OK \Rightarrow (\forall i \cdot i \in l+1 \,..\, n \Rightarrow task(i) = ND)$

EVENTS
Initialisation

    begin
        act1 : $task := 1 \,..\, n \times \{ND\}$
        act2 : $idle := TRUE$
    end

Event *Activation* $\widehat{=}$
extends *Activation*

    when
        grd1 : idle = TRUE
    then
        act1 : idle := FALSE
    end

Event *Start* $\widehat{=}$
refines *Body*

    any
        res
    where
        grd1 : $res \in \{OK, NOK\}$
        grd2 : $idle = FALSE$
        grd3 : $task(1) = ND$
    with
        goal' : $goal' = ss\_trans(task'[1 \,..\, n])$

```
        then
            act1 : $task(1) := res$
        end
Event  Progress ≙
refines  Body
        any
            $j$
            $res$
        where
            grd1 : $j > 0$
            grd2 : $j < n$
            grd3 : $res \in \{OK, NOK\}$
            grd4 : $task(j) = OK$
            grd5 : $task(j + 1) = ND$
        with
            goal' : $\mathtt{goal'} = \mathtt{ss\_trans}(\mathtt{task'}[1 \,..\, \mathtt{n}])$
        then
            act1 : $task(j + 1) := res$
        end
Event  Finish ≙
refines  Finish
        when
            grd1 : $task[1 \,..\, n] = \{OK\}$
        then
            act1 : $idle := TRUE$
            act2 : $task := 1 \,..\, n \times \{ND\}$
        end
Event  Abort ≙
refines  Abort
        any
            $j$
        where
            grd1 : $j > 0$
            grd2 : $j \leq n$
            grd3 : $task(j) = NOK$
        then
            skip
        end
END
```

MACHINE ReconfigSystem_Ref
REFINES ReconfigSystem
SEES ReconfigSystem_ctx
VARIABLES

```
    idle
    task
    ct
```

INVARIANTS

inv1 : $ct \in 0 \mathrel{..} n$
inv2 : $idle = TRUE \Rightarrow ct = 0$
inv3 : $ct > 0 \Rightarrow (\forall i \cdot i \in 1 \mathrel{..} ct \Rightarrow task(i) = OK)$
inv4 : $ct < n \Rightarrow task(ct + 1) \in \{ND, NOK\}$
inv5 : $ct < n - 1 \Rightarrow (\forall i \cdot i \in ct + 2 \mathrel{..} n \Rightarrow task(i) = ND)$

EVENTS
Initialisation

extended
begin
act1 : task := 1 .. n × {ND}
act2 : idle := TRUE
act3 : $ct := 0$
end

Event *Activation* $\widehat{=}$
extends *Activation*
when
grd1 : idle = TRUE
then
act1 : idle := FALSE
end

Event *Start* $\widehat{=}$
refines *Start*
when
grd1 : $idle = FALSE$
grd2 : $ct = 0$
grd3 : $task(1) = ND$
with
res : res = OK
then
act1 : $task(1) := OK$
act2 : $ct := 1$

```
        end
Event   FailStart ≙
refines Start
        when
            grd1 : $idle = FALSE$
            grd2 : $ct = 0$
            grd3 : $task(1) = ND$
        with
            res : res = NOK
        then
            act1 : $task(1) := NOK$
        end
Event   Progress ≙
refines Progress
        when
            grd1 : $ct > 0$
            grd2 : $ct < n$
            grd3 : $task(ct + 1) = ND$
        with
            res : res = OK
            j : j = ct
        then
            act1 : $task(ct + 1) := OK$
            act2 : $ct := ct + 1$
        end
Event   FailProgress ≙
refines Progress
        when
            grd1 : $ct > 0$
            grd2 : $ct < n$
            grd3 : $task(ct + 1) = ND$
        with
            res : res = NOK
            j : j = ct
        then
            act1 : $task(ct + 1) := NOK$
        end
Event   Finish ≙
refines Finish
        when
            grd1 : $ct = n$
        then
            act1 : $idle := TRUE$
            act2 : $task := task \lhd\!\!\!- (1 \,..\, n \times \{ND\})$
            act3 : $ct := 0$
        end
Event  $Abort$ $\widehat{=}$
refines  $Abort$
        when
            grd1 : $ct < n$
            grd2 : $task(ct + 1) = NOK$
        with
            j : j = ct + 1
        then
            skip
        end
END
```

MACHINE ReconfigSystem_Ref_Str1
REFINES ReconfigSystem_Ref
SEES ReconfigSystem_ctx
VARIABLES

```
    idle
    ct
    module
    a_comp
    b_comp
    c_comp
```

INVARIANTS

- inv1 : $module \in \{A, B, C\}$
- inv2 : $a\_comp \in 1 \,..\, n \rightarrow STATE$
- inv3 : $b\_comp \in 1 \,..\, n \rightarrow STATE$
- inv4 : $c\_comp \in 1 \,..\, n \rightarrow STATE$
- inv5 : $\forall i \cdot i \in 1 \,..\, n \land module = A \land a\_comp(i) = OK \Rightarrow task(i) = OK$
- inv6 : $module = A \Rightarrow (\forall i \cdot i \in 1 \,..\, n \Rightarrow b\_comp(i) = ND \land c\_comp(i) = ND)$
- inv7 : $\forall i \cdot i \in 1 \,..\, n \land module = A \land a\_comp(i) \neq OK \Rightarrow task(i) = ND$
- inv8 : $module = A \land ct < n-1 \Rightarrow (\forall i \cdot i \in ct+2 \,..\, n \Rightarrow a\_comp(i) = ND)$
- inv9 : $module = A \land ct > 0 \Rightarrow (\forall i \cdot i \in 1 \,..\, ct \Rightarrow a\_comp(i) = OK)$
- inv10 : $\forall i \cdot i \in 1 \,..\, n \land module = B \land b\_comp(i) \neq OK \Rightarrow task(i) = ND$
- inv11 : $\forall i \cdot i \in 1 \,..\, n \land module = C \Rightarrow c\_comp(i) = task(i)$
- inv12 : $module = B \Rightarrow (\exists i \cdot i \in 1 \,..\, n \land a\_comp(i) = NOK)$
- inv13 : $module = C \Rightarrow (\exists i \cdot i \in 1 \,..\, n \land b\_comp(i) = NOK)$
- inv19 : $module = B \Rightarrow (\forall i \cdot i \in 1 \,..\, n \Rightarrow c\_comp(i) = ND)$
- inv14 : $module = B \land ct > 0 \Rightarrow (\forall i \cdot i \in 1 \,..\, ct \Rightarrow b\_comp(i) = OK)$
- inv15 : $module = C \land ct > 0 \Rightarrow (\forall i \cdot i \in 1 \,..\, ct \Rightarrow c\_comp(i) = OK)$
- inv16 : $module = A \land ct < n \Rightarrow a\_comp(ct + 1) \in \{ND, NOK\}$
- inv17 : $module = B \land ct < n \Rightarrow b\_comp(ct + 1) \in \{ND, NOK\}$
- inv18 : $module = B \land ct < n-1 \Rightarrow (\forall i \cdot i \in ct+2 \,..\, n \Rightarrow b\_comp(i) = ND)$

EVENTS

Initialisation

begin

- act1 : $module := A$
- act2 : $idle := TRUE$
- act3 : $ct := 0$

```
            act4 : $a\_comp := 1 \,..\, n \times \{ND\}$
            act5 : $b\_comp := 1 \,..\, n \times \{ND\}$
            act6 : $c\_comp := 1 \,..\, n \times \{ND\}$
      end
Event  Activation ≙
extends  Activation
      when
            grd1 : idle = TRUE
      then
            act1 : idle := FALSE
      end
Event  StartA ≙
refines  Start
      when
            grd1 : $module = A$
            grd2 : $idle = FALSE$
            grd3 : $ct = 0$
            grd4 : $a\_comp(1) = ND$
      then
            act1 : $a\_comp(1) := OK$
            act2 : $ct := 1$
      end
Event  StartB ≙
refines  Start
      when
            grd1 : $module = B$
            grd2 : $idle = FALSE$
            grd3 : $ct = 0$
            grd4 : $b\_comp(1) = ND$
      then
            act1 : $b\_comp(1) := OK$
            act2 : $ct := 1$
      end
Event  StartC ≙
refines  Start
      when
            grd1 : $module = C$
            grd2 : $idle = FALSE$
            grd3 : $ct = 0$
            grd4 : $c\_comp(1) = ND$
```

```
        then
            act1 : $c\_comp(1) := OK$
            act2 : $ct := 1$
        end
Event   $FailStartA \mathrel{\widehat{=}}$
        when
            grd1 : $module = A$
            grd2 : $idle = FALSE$
            grd3 : $ct = 0$
            grd4 : $a\_comp(1) = ND$
        then
            act1 : $a\_comp(1) := NOK$
        end
Event   $FailStartB \mathrel{\widehat{=}}$
        when
            grd1 : $module = B$
            grd2 : $idle = FALSE$
            grd3 : $ct = 0$
            grd4 : $b\_comp(1) = ND$
        then
            act1 : $b\_comp(1) := NOK$
        end
Event   $FailStartC \mathrel{\widehat{=}}$
refines $FailStart$
        when
            grd1 : $module = C$
            grd2 : $idle = FALSE$
            grd3 : $ct = 0$
            grd4 : $c\_comp(1) = ND$
        then
            act1 : $c\_comp(1) := NOK$
        end
Event   $ProgressA \mathrel{\widehat{=}}$
refines $Progress$
        when
            grd1 : $module = A$
            grd2 : $ct > 0$
            grd3 : $ct < n$
            grd4 : $a\_comp(ct + 1) = ND$
        then
            act1 : $a\_comp(ct + 1) := OK$
            act2 :  $ct := ct + 1$
        end
Event  $ProgressB \widehat{=}$
refines  $Progress$
        when
            grd1 :  $module = B$
            grd2 :  $ct > 0$
            grd3 :  $ct < n$
            grd4 :  $b\_comp(ct + 1) = ND$
        then
            act1 :  $b\_comp(ct + 1) := OK$
            act2 :  $ct := ct + 1$
        end
Event  $ProgressC \widehat{=}$
refines  $Progress$
        when
            grd1 :  $module = C$
            grd2 :  $ct > 0$
            grd3 :  $ct < n$
            grd4 :  $c\_comp(ct + 1) = ND$
        then
            act1 :  $c\_comp(ct + 1) := OK$
            act2 :  $ct := ct + 1$
        end
Event  $FailProgressA \widehat{=}$
        when
            grd1 :  $module = A$
            grd2 :  $ct > 0$
            grd3 :  $ct < n$
            grd4 :  $a\_comp(ct + 1) = ND$
        then
            act1 :  $a\_comp(ct + 1) := NOK$
        end
Event  $FailProgressB \widehat{=}$
        when
            grd1 :  $module = B$
            grd2 :  $ct > 0$
            grd3 :  $ct < n$
            grd4 :  $b\_comp(ct + 1) = ND$
        then
            act1 :  $b\_comp(ct + 1) := NOK$
        end
Event  FailProgressC ≙
refines  FailProgress
        when
            grd1 :  $module = C$
            grd2 :  $ct > 0$
            grd3 :  $ct < n$
            grd4 :  $c\_comp(ct + 1) = ND$
        then
            act1 :  $c\_comp(ct + 1) := NOK$
        end
Event  FinishA ≙
refines  Finish
        when
            grd1 :  $ct = n$
            grd2 :  $module = A$
        then
            act1 :  $idle := TRUE$
            act2 :  $a\_comp := 1 .. n \times \{ND\}$
            act3 :  $ct := 0$
        end
Event  EnableSpareB ≙
        when
            grd1 :  $module = A$
            grd2 :  $ct < n$
            grd3 :  $a\_comp(ct + 1) = NOK$
        then
            act1 :  $module := B$
            act2 :  $b\_comp := b\_comp \mathbin{<\!\!+} (1 .. ct \times \{OK\})$
        end
Event  EnableSpareC ≙
        when
            grd1 :  $module = B$
            grd2 :  $ct < n$
            grd3 :  $b\_comp(ct + 1) = NOK$
        then
            act1 :  $module := C$
            act2 :  $c\_comp := c\_comp \mathbin{<\!\!+} (1 .. ct \times \{OK\})$
        end
Event  Abort ≙
```

```
refines Abort
    when
        grd1 : $ct < n$
        grd2 : $c\_comp(ct + 1) = NOK$
    then
        skip
    end
Event FinishB $\widehat{=}$
refines Finish
    when
        grd1 : $ct = n$
        grd2 : $module = B$
    then
        act1 : $idle := TRUE$
        act2 : $b\_comp := 1 \mathrel{..} n \times \{ND\}$
        act3 : $ct := 0$
    end
Event FinishC $\widehat{=}$
refines Finish
    when
        grd1 : $ct = n$
        grd2 : $module = C$
    then
        act1 : $idle := TRUE$
        act2 : $c\_comp := 1 \mathrel{..} n \times \{ND\}$
        act3 : $ct := 0$
    end
END
```

MACHINE ReconfigSystem_Ref_Str2

REFINES ReconfigSystem_Ref

SEES ReconfigSystem_ctx

VARIABLES

idle

a_comp

b_comp

ct

module

INVARIANTS

inv1 : $module \in \{A, B\}$

inv2 : $a\_comp \in 1 \,..\, n \rightarrow STATE$

inv3 : $b\_comp \in 1 \,..\, n \rightarrow STATE$

inv4 : $ct = 0 \wedge module = A \wedge a\_comp(1) = ND \Rightarrow task(1) = ND$

inv5 : $ct = 0 \wedge module = B \wedge b\_comp(1) = ND \Rightarrow task(1) = ND$

inv6 : $module = A \wedge ct > 0 \wedge a\_comp(ct) = OK \Rightarrow task(ct) = OK$

inv7 : $module = B \wedge ct > 0 \wedge b\_comp(ct) = OK \Rightarrow task(ct) = OK$

inv8 : $\forall i \cdot i \in 1 \,..\, n \wedge a\_comp(i) = NOK \wedge b\_comp(i) = NOK \Rightarrow task(i) = NOK$

inv9 : $\forall i \cdot i \in 1 \,..\, n \wedge a\_comp(i) = NOK \wedge b\_comp(i) = ND \Rightarrow task(i) = ND$

inv16 : $\forall i \cdot i \in 1 \,..\, n \wedge b\_comp(i) = NOK \wedge a\_comp(i) = ND \Rightarrow task(i) = ND$

inv17 : $\forall i \cdot i \in 1 \,..\, n \wedge a\_comp(i) = ND \wedge b\_comp(i) = ND \Rightarrow task(i) = ND$

inv20 : $module = A \wedge ct > 0 \wedge ct < n \wedge a\_comp(ct + 1) = ND \Rightarrow task(ct + 1) = ND$

inv21 : $module = B \wedge ct > 0 \wedge ct < n \wedge b\_comp(ct + 1) = ND \Rightarrow task(ct + 1) = ND$

EVENTS

Initialisation

begin

act1 : $idle := TRUE$

act2 : $ct := 0$

act3 : $module := A$

act5 : $b\_comp := 1 \,..\, n \times \{ND\}$

act4 : $a\_comp := 1 \,..\, n \times \{ND\}$

end

```
Event  Activation ≙
extends  Activation
      when
         grd1 : idle = TRUE
      then
         act1 : idle := FALSE
      end
Event  StartA ≙
refines  Start
      when
         grd1 : idle = FALSE
         grd2 : module = A
         grd3 : ct = 0
         grd4 : a_comp(1) = ND
      then
         act1 : a_comp(1) := OK
         act2 : ct := 1
      end
Event  StartB ≙
refines  Start
      when
         grd1 : idle = FALSE
         grd2 : module = B
         grd3 : ct = 0
         grd4 : b_comp(1) = ND
      then
         act1 : b_comp(1) := OK
         act2 : ct := 1
      end
Event  FailStartA_1 ≙
refines  FailStart
      when
         grd1 : idle = FALSE
         grd2 : module = A
         grd3 : ct = 0
         grd4 : a_comp(1) = ND
         grd5 : b_comp(1) = NOK
      then
         act1 : a_comp(1) := NOK
      end
```

```
Event  FailStartB_1 $\widehat{=}$
refines  FailStart
    when
        grd1 :  $idle = FALSE$
        grd2 :  $module = B$
        grd3 :  $ct = 0$
        grd4 :  $b\_comp(1) = ND$
        grd5 :  $a\_comp(1) = NOK$
    then
        act1 :  $b\_comp(1) := NOK$
    end
Event  ProgressA $\widehat{=}$
refines  Progress
    when
        grd1 :  $ct > 0$
        grd2 :  $ct < n$
        grd3 :  $module = A$
        grd4 :  $a\_comp(ct + 1) = ND$
    then
        act1 :  $a\_comp(ct + 1) := OK$
        act2 :  $ct := ct + 1$
    end
Event  ProgressB $\widehat{=}$
refines  Progress
    when
        grd1 :  $ct > 0$
        grd2 :  $ct < n$
        grd3 :  $module = B$
        grd4 :  $b\_comp(ct + 1) = ND$
    then
        act1 :  $b\_comp(ct + 1) := OK$
        act2 :  $ct := ct + 1$
    end
Event  FailProgressA_1 $\widehat{=}$
refines  FailProgress
    when
        grd1 :  $ct > 0$
        grd2 :  $ct < n$
        grd5 :  $module = A$
        grd4 :  $a\_comp(ct + 1) = ND$
        grd6 :  $b\_comp(ct + 1) = NOK$
    then
        act1 : $a\_comp(ct + 1) := NOK$
    end
Event  $FailProgressB\_1 \widehat{=}$
refines  $FailProgress$
    when
        grd1 : $ct > 0$
        grd2 : $ct < n$
        grd3 : $module = B$
        grd4 : $b\_comp(ct + 1) = ND$
        grd5 : $a\_comp(ct + 1) = NOK$
    then
        act1 : $b\_comp(ct + 1) := NOK$
    end
Event  $Finish \widehat{=}$
refines  $Finish$
    when
        grd1 : $ct = n$
    then
        act1 : $idle := TRUE$
        act2 : $ct := 0$
        act3 : $a\_comp := a\_comp \mathbin{\lhd\!\!\!+} (dom(a\_comp \triangleright \{OK\}) \times \{ND\})$
        act4 : $b\_comp := b\_comp \mathbin{\lhd\!\!\!+} (dom(b\_comp \triangleright \{OK\}) \times \{ND\})$
    end
Event  $Abort \widehat{=}$
refines  $Abort$
    when
        grd1 : $ct < n$
        grd2 : $a\_comp(ct + 1) = NOK \wedge b\_comp(ct + 1) = NOK$
    then
        skip
    end
Event  $SwitchAB \widehat{=}$
    when
        grd1 : $module = A$
        grd2 : $ct < n$
        grd3 : $a\_comp(ct + 1) = NOK$
        grd4 : $b\_comp(ct + 1) = ND$
    then
        act1 : $module := B$
```

```
    end
Event   SwitchBA ≙
    when
        grd1 : $module = B$
        grd2 : $ct < n$
        grd3 : $b\_comp(ct + 1) = NOK$
        grd4 : $a\_comp(ct + 1) = ND$
    then
        act1 : $module := A$
    end
Event   FailStartA_2 ≙
    when
        grd1 : $idle = FALSE$
        grd2 : $module = A$
        grd3 : $ct = 0$
        grd4 : $a\_comp(1) = ND$
        grd5 : $b\_comp(1) = ND$
    then
        act1 : $a\_comp(1) := NOK$
    end
Event   FailStartB_2 ≙
    when
        grd1 : $idle = TRUE$
        grd2 : $module = B$
        grd3 : $ct = 0$
        grd4 : $b\_comp(1) = ND$
        grd5 : $a\_comp(1) = ND$
    then
        act1 : $b\_comp(1) := NOK$
    end
Event   FailProgressA_2 ≙
    when
        grd1 : $ct > 0$
        grd2 : $ct < n$
        grd3 : $module = A$
        grd4 : $a\_comp(ct + 1) = ND$
        grd5 : $b\_comp(ct + 1) = ND$
    then
        act1 : $a\_comp(ct + 1) := NOK$
    end
Event   FailProgressB_2 ≙
```

```
    when
        grd1 : $ct > 0$
        grd2 : $ct < n$
        grd3 : $module = B$
        grd4 : $b\_comp(ct + 1) = ND$
        grd5 : $a\_comp(ct + 1) = ND$
    then
        act1 : $b\_comp(ct + 1) := NOK$
    end
END
```

Appendix B

PRISM Specifications

```
**ctmc** // *Single DPU*

**const double** $\lambda = 1/(12 \cdot 60)$ // *TC access rate*
**const double** $\mu = 1/20;$ // *TM output rate*
**const double** $\alpha_1 = 1/60;$ // *DPU's component 1 service rate*
**const double** $\beta_1 = 0.0000008;$ // *DPU's component 1 failure rate*
**const double** $\alpha_2 = 1/30;$
**const double** $\beta_2 = 0.000001;$
**const double** $\alpha_3 = 1/30;$
**const double** $\beta_3 = 0.0000009;$
**const double** $\alpha_4 = 1/(90);$
**const double** $\beta_4 = 0.0000006;$
**global** $idle$ : **bool init** $true$;

**module** *Activation*
  $[]\ idle \rightarrow \lambda : (idle' = false);$
**endmodule**

**module** *DPU*
  $a_1 : [0..2]$ **init** $0$; // *0=ND, 1=OK, 2=NOK*
  $a_2 : [0..2]$ **init** $0$;
  $a_3 : [0..2]$ **init** $0$;
  $a_4 : [0..2]$ **init** $0$;
  $[]\ idle \rightarrow \lambda : (idle' = false);$
  $[]\ !idle \,\&\, a_1 = 0 \rightarrow \alpha_1 : (a_1' = 1) + \beta_1 : (a_1' = 2);$
  $[]\ a_2 = 0 \,\&\, a_1 = 1 \rightarrow \alpha_2 : (a_2' = 1) + \beta_2 : (a_2' = 2);$
  $[]\ a_3 = 0 \,\&\, a_2 = 1 \rightarrow \alpha_3 : (a_3' = 1) + \beta_3 : (a_3' = 2);$
  $[]\ a_4 = 0 \,\&\, a_3 = 1 \rightarrow \alpha_4 : (a_4' = 1) + \beta_4 : (a_4' = 2);$
  $[]\ a_4 = 1 \rightarrow \mu : (a_1' = 0) \,\&\, (a_2' = 0) \,\&\, (a_3' = 0) \,\&\, (a_4' = 0) \,\&\, (idle' = true);$
**endmodule**

**module** *Abort*
  $[]\ fail \rightarrow true;$
**endmodule**

**formula** $fail = (a_1 = 2 \,|\, a_2 = 2 \,|\, a_3 = 2 \,|\, a_4 = 2);$
```

rewards "*goals*"
  [] $a_4 = 1\ :\ 1$;
endrewards

```
ctmc // Triplicated DPU with static reconfiguration
const double $\lambda = 1/(12 \cdot 60)$ // TC access rate
const double $\mu = 1/20$; // TM output rate
const double $\delta = 1/10$ // spare DPU activation rate
const double $\tau = 1/5$; // DPUs' communication rate
const double $\alpha_1 = 1/60$; // DPU's component 1 service rate
const double $\beta_1 = 0.0000008$; // DPU's component 1 failure rate
const double $\alpha_2 = 1/30$;
const double $\beta_2 = 0.000001$;
const double $\alpha_3 = 1/30$;
const double $\beta_3 = 0.0000009$;
const double $\alpha_4 = 1/(90)$;
const double $\beta_4 = 0.0000006$;
global $idle$ : bool init $true$;
global $ct$ : [0..4] init 0;
module $Activation$
  [] $idle \rightarrow \lambda : (idle' = false)$;
endmodule
module $DPU_A$
  $a_1$ : [0..2] init 0; // 0=ND, 1=OK, 2=NOK
  $a_2$ : [0..2] init 0;
  $a_3$ : [0..2] init 0;
  $a_4$ : [0..2] init 0;
  [] $!idle \,\&\, dpu = 1 \,\&\, a_1 = 0 \rightarrow \alpha_1 : (a_1' = 1) \,\&\, (ct' = ct + 1) + \beta_1 : (a_1' = 2)$;
  [] $dpu = 1 \,\&\, a_2 = 0 \,\&\, ct = 1 \rightarrow \alpha_2 : (a_2' = 1) \,\&\, (ct' = ct + 1) + \beta_2 : (a_2' = 2)$;
  [] $dpu = 1 \,\&\, a_3 = 0 \,\&\, ct = 2 \rightarrow \alpha_3 : (a_3' = 1) \,\&\, (ct' = ct + 1) + \beta_3 : (a_3' = 2)$;
  [] $dpu = 1 \,\&\, a_4 = 0 \,\&\, ct = 3 \rightarrow \alpha_4 : (a_4' = 1) \,\&\, (ct' = ct + 1) + \beta_4 : (a_4' = 2)$;
  [] $dpu = 1 \,\&\, ct = 4 \rightarrow \mu$ :
       $(a_1' = 0) \,\&\, (a_2' = 0) \,\&\, (a_3' = 0) \,\&\, (a_4' = 0) \,\&\, (idle' = true) \,\&\, (ct' = 0)$;
endmodule
module $DPU_B$
  $b_1$ : [0..2] init 0; // 0=ND, 1=OK, 2=NOK
  $b_2$ : [0..2] init 0;
  $b_3$ : [0..2] init 0;
  $b_4$ : [0..2] init 0;
  [] $!idle \,\&\, dpu = 2 \,\&\, b_1 = 0 \rightarrow \alpha_1 : (b_1' = 1) \,\&\, (ct' = ct + 1) + \beta_1 : (b_1' = 2)$;
  [] $dpu = 2 \,\&\, b_2 = 0 \,\&\, ct = 1 \rightarrow \alpha_2 : (b_2' = 1) \,\&\, (ct' = ct + 1) + \beta_2 : (b_2' = 2)$;
  [] $dpu = 2 \,\&\, b_3 = 0 \,\&\, ct = 2 \rightarrow \alpha_3 : (b_3' = 1) \,\&\, (ct' = ct + 1) + \beta_3 : (b_3' = 2)$;
  [] $dpu = 2 \,\&\, b_4 = 0 \,\&\, ct = 3 \rightarrow \alpha_4 : (b_4' = 1) \,\&\, (ct' = ct + 1) + \beta_4 : (b_4' = 2)$;
  [] $dpu = 2 \,\&\, ct = 4 \rightarrow \mu$ :
       $(b_1' = 0) \,\&\, (b_2' = 0) \,\&\, (b_3' = 0) \,\&\, (b_4' = 0) \,\&\, (idle' = true) \,\&\, (ct' = 0)$;
**endmodule**

**module** $DPU_C$
  $c_1 : [0..2]$ **init** 0; *// 0=ND, 1=OK, 2=NOK*
  $c_2 : [0..2]$ **init** 0;
  $c_3 : [0..2]$ **init** 0;
  $c_4 : [0..2]$ **init** 0;
  [] $!idle \,\&\, dpu = 3 \,\&\, c_1 = 0 \rightarrow \alpha_1 : (c_1' = 1) \,\&\, (ct' = ct + 1) + \beta_1 : (c_1' = 2)$;
  [] $dpu = 3 \,\&\, c_2 = 0 \,\&\, ct = 1 \rightarrow \alpha_2 : (c_2' = 1) \,\&\, (ct' = ct + 1) + \beta_2 : (c_2' = 2)$;
  [] $dpu = 3 \,\&\, c_3 = 0 \,\&\, ct = 2 \rightarrow \alpha_3 : (c_3' = 1) \,\&\, (ct' = ct + 1) + \beta_3 : (c_3' = 2)$;
  [] $dpu = 3 \,\&\, c_4 = 0 \,\&\, ct = 3 \rightarrow \alpha_4 : (c_4' = 1) \,\&\, (ct' = ct + 1) + \beta_4 : (c_4' = 2)$;
  [] $dpu = c \,\&\, ct = 4 \rightarrow \mu$ :
       $(c_1' = 0) \,\&\, (c_2' = 0) \,\&\, (c_3' = 0) \,\&\, (c_4' = 0) \,\&\, (idle' = true) \,\&\, (ct' = 0)$;
**endmodule**

**module** $Switcher$
  $dpu : [1..3]$ **init** 1; *// 1=A, 2=B, 3=C*
  $spareB$ : **bool init** $false$;
  $spareC$ : **bool init** $false$;
*// switching to the first spare DPU (A to B)*
  [] $failA \,\&\, !spareB \rightarrow \delta : (spareB' = true)$;
  [] $spareB \,\&\, dpu = 1 \rightarrow \tau : (dpu' = 2)$;
*// switching to the second spare DPU (B to C)*
  [] $failB \,\&\, !spareC \rightarrow \delta : (spareC' = true)$;
  [] $spareC \,\&\, dpu = 2 \rightarrow \tau : (dpu' = 3)$;
**endmodule**

**module** $Abort$
  [] $failC \rightarrow true$;
**endmodule**

**formula** $failA = (a_1 = 2 \,|\, a_2 = 2 \,|\, a_3 = 2 \,|\, a_4 = 2)$;
**formula** $failB = (b_1 = 2 \,|\, b_2 = 2 \,|\, b_3 = 2 \,|\, b_4 = 2)$;
**formula** $failC = (c_1 = 2 \,|\, c_2 = 2 \,|\, c_3 = 2 \,|\, c_4 = 2)$;

**rewards** “$goals$”
  [] $ct = 4$ : 1;
**endrewards**
```

```
ctmc // Duplicated DPU with dynamic reconfiguration

const double $\lambda = 1/(12 \cdot 60)$ // TC access rate
const double $\mu = 1/20$; // TM output rate
const double $\delta = 1/10$ // spare DPU activation rate
const double $\tau = 1/5$; // DPUs' communication rate
const double $\alpha_1 = 1/60$; // DPU's component 1 service rate
const double $\beta_1 = 0.0000008$; // DPU's component 1 failure rate
const double $\alpha_2 = 1/30$;
const double $\beta_2 = 0.000001$;
const double $\alpha_3 = 1/30$;
const double $\beta_3 = 0.0000009$;
const double $\alpha_4 = 1/(90)$;
const double $\beta_4 = 0.0000006$;
global $idle$ : bool init $true$;
global $ct$ : [0..4] init 0;

module Activation
  [] $idle \rightarrow \lambda : (idle' = false)$;
endmodule

module $DPU_{AB}$
  $a_1$ : [0..2] init 0; // 0=ND, 1=OK, 2=NOK
  $a_2$ : [0..2] init 0;
  $a_3$ : [0..2] init 0;
  $a_4$ : [0..2] init 0;
  $b_1$ : [0..2] init 0;
  $b_2$ : [0..2] init 0;
  $b_3$ : [0..2] init 0;
  $b_4$ : [0..2] init 0;
  $na_1$ : [0..2] init 0; // next-step states of the components
  $na_2$ : [0..2] init 0;
  $na_3$ : [0..2] init 0;
  $na_4$ : [0..2] init 0;
  $nb_1$ : [0..2] init 0;
  $nb_2$ : [0..2] init 0;
  $nb_3$ : [0..2] init 0;
  $nb_4$ : [0..2] init 0;
  [] $!idle \,\&\, dpu = 1 \,\&\, a_1 = 0 \,\&\, ct = 0 \rightarrow$
        $\alpha_1 : (a_1' = 1) \,\&\, (na_1' = 0) \,\&\, (ct' = ct + 1) + \beta_1 : (a_1' = 2) \,\&\, (na_1' = 2)$;
  [] $dpu = 1 \,\&\, a_2 = 0 \,\&\, ct = 1 \rightarrow$
        $\alpha_2 : (a_2' = 1) \,\&\, (na_2' = 0) \,\&\, (ct' = ct + 1) + \beta_2 : (a_2' = 2) \,\&\, (na_2' = 2)$;
```

$[]\ dpu = 1\ \&\ a_3 = 0\ \&\ ct = 2 \rightarrow$
$\alpha_3 : (a_3' = 1)\ \&\ (na_3' = 0)\ \&\ (ct' = ct + 1) + \beta_3 : (a_3' = 2)\ \&\ (na_3' = 2);$
$[]\ dpu = 1\ \&\ a_4 = 0\ \&\ ct = 3 \rightarrow$
$\alpha_4 : (a_4' = 1)\ \&\ (na_4' = 0)\ \&\ (ct' = ct + 1) + \beta_4 : (a_4' = 2)\ \&\ (na_4' = 2);$
$[]\ !idle\ \&\ dpu = 2\ \&\ b_1 = 0 \rightarrow$
$\alpha_1 : (b_1' = 1)\ \&\ (nb_1' = 0)\ \&\ (ct' = ct + 1) + \beta_1 : (b_1' = 2)\ \&\ (nb_1' = 2);$
$[]\ dpu = 2\ \&\ b_2 = 0\ \&\ b_1 = 1 \rightarrow$
$\alpha_2 : (b_2' = 1)\ \&\ (nb_2' = 0)\ \&\ (ct' = ct + 1) + \beta_2 : (b_2' = 2)\ \&\ (nb_2' = 2);$
$[]\ dpu = 2\ \&\ b_3 = 0\ \&\ b_2 = 1 \rightarrow$
$\alpha_3 : (b_3' = 1)\ \&\ (nb_3' = 0)\ \&\ (ct' = ct + 1) + \beta_3 : (b_3' = 2)\ \&\ (nb_3' = 2);$
$[]\ dpu = 2\ \&\ b_4 = 0\ \&\ b_3 = 1 \rightarrow$
$\alpha_4 : (b_4' = 1)\ \&\ (nb_4' = 0)\ \&\ (ct' = ct + 1) + \beta_4 : (b_4' = 2)\ \&\ (nb_4' = 2);$
$[]\ ct = 4 \rightarrow \mu : (a_1' = na_1)\ \&\ (a_2' = na_2)\&$
$(a_3' = na_3)\ \&\ (a_4' = na_4)\ \&\ (b_1' = nb_1)\ \&\ (b_2' = nb_2)\&$
$(b_3' = nb_3)\ \&\ (b_4' = nb_4)\ \&\ (idle' = true)\ \&\ (ct' = 0);$

endmodule

module *Switcher*

$dpu : [1..2]$ **init** 1; *// 1=A, 2=B*
// counter of switches between DPUs, 2 = "two or more times"
$swc : [0..2]$ **init** 1;

$[]\ failA\ \&\ swc = 0 \rightarrow \delta : (swc' = 1);$ *// spare DPU activation*
$[]\ failB\ \&\ swc = 1 \rightarrow \delta : (swc' = 2);$ *// main DPU reactivation*
// switching to the spare DPU (A to B)
$[]\ a_1 = 2\ \&\ b_1 = 0\ \&\ ct = 0\ \&\ swc > 0\ \&\ dpu = 1 \rightarrow \tau : (dpu' = 2);$
$[]\ a_2 = 2\ \&\ b_2 = 0\ \&\ ct = 1\ \&\ swc > 0\ \&\ dpu = 1 \rightarrow \tau : (dpu' = 2);$
$[]\ a_3 = 2\ \&\ b_3 = 0\ \&\ ct = 2\ \&\ swc > 0\ \&\ dpu = 1 \rightarrow \tau : (dpu' = 2);$
$[]\ a_4 = 2\ \&\ b_4 = 0\ \&\ ct = 3\ \&\ swc > 0\ \&\ dpu = 1 \rightarrow \tau : (dpu' = 2);$
// switching to the main DPU (B to A)
$[]\ a_1 = 0\ \&\ b_1 = 2\ \&\ ct = 0\ \&\ swc = 2\ \&\ dpu = 2 \rightarrow \tau : (dpu' = 1);$
$[]\ a_2 = 0\ \&\ b_2 = 2\ \&\ ct = 1\ \&\ swc = 2\ \&\ dpu = 2 \rightarrow \tau : (dpu' = 1);$
$[]\ a_3 = 0\ \&\ b_3 = 2\ \&\ ct = 2\ \&\ swc = 2\ \&\ dpu = 2 \rightarrow \tau : (dpu' = 1);$
$[]\ a_4 = 0\ \&\ b_4 = 2\ \&\ ct = 3\ \&\ swc = 2\ \&\ dpu = 2 \rightarrow \tau : (dpu' = 1);$

endmodule

module *Abort*

$[]\ fail \rightarrow true;$

endmodule

formula $fail = ((a_1 = 2\ \&\ b_1 = 2)\ |$
$(a_2 = 2\ \&\ b_2 = 2)\ |\ (a_3 = 2\ \&\ b_3 = 2)\ |\ (a_4 = 2\ \&\ b_4 = 2));$

rewards "*goals*"
  [] $ct = 4\ :\ 1$;
endrewards

ISBN 978-952-12-2719-6
ISSN 1239-1891