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

Anton Tarasyuk[1,2], Inna Pereverzeva[1,2], Elena Troubitsyna[1],
Timo Latvala[3], and Laura Nummila[3]

[1] Åbo Akademi University, Turku, Finland
[2] Turku Centre for Computer Science, Turku, Finland
[3] Space Systems Finland, Espoo, Finland
{inna.pereverzeva,anton.tarasyuk,elena.troubitsyna}@abo.fi,
{timo.latvala,laura.nummila}@ssf.fi

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. One of the essential means to achieve it is redundancy. However, the use of (hardware) component redundancy in spacecraft is restricted by the weight and volume constraints. Thus, the system developers need to perform a careful cost-benefit analysis to minimise the use of spare modules yet achieve the required level of reliability.

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 consists of two identical modules. When one of the first module subcomponents 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 a solution that relied on healthy subcomponents of both modules and a complex communication mechanism to restore system functioning. Obviously, a certain amount of data has been lost before a 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 Space Systems Finland and propose an approach to formal development and assessment of fault tolerant satellite systems. The essence of our modelling approach is to start from abstract modelling functional goals that the system should achieve to remain operational, and to derive reconfigurable architecture by refinement in the Event-B formalism [1]. 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 the reconfiguration procedure requires additional inter-component communication, the developers should also verify that system performance remains acceptable. Quantitative evaluation of reliability and performance of probabilistically augmented Event-B models is performed using the PRISM model checker [8].

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 the design space and helps to build redundancy-frugal systems that meet the desired reliability and performance requirements.

2 Reconfigurable Fault Tolerant Systems

2.1 Case Study: Data Processing Unit

As mentioned in the previous section, our work is inspired by a solution proposed to circumvent the double failure occurred in a currently operational on-board satellite system. The architecture of that system is similar to Data Processing Unit (DPU) – a subsystem of the European Space Agency (ESA) mission BepiColombo [2]. Space Systems Finland is one of the providers for BepiColombo. The main goal of the mission is to carry out 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 circular buffer (TC pool). With a predefined rate, DPU periodically polls the buffer, decodes a TC and performs the required actions. Processing of each TC results in producing telemetry (TM). Both TC and TM packages follow the syntax defined by the ESA Packet Utilisation Standard [12]. As a result of TC decoding, DPU might produce a housekeeping report, switch to some mode or initiate/continue production of *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 the system behaviour. Hence, in the rest of the paper, TC will correspond to the telecommands requiring production of scientific data, while TM will designate packages containing scientific data.

2.2 Goal-Oriented Reasoning about Fault Tolerance

We use the notion of a 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 [7,9].

Let $\mathcal{G}$ be a predicate that defines a desired goal and $\mathcal{M}$ be a system model. Ideally, the system design should ensure that the goal can be reached "infinitely often". Hence, while verifying the system, we should establish that

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

The main idea of a 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"*.

Generally, the goal $\mathcal{G}$ can be decomposed into a finite set of tasks:

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

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

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

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. Fault tolerance mechanisms employed to mitigate results of component failures rely on various forms of component redundancy. 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, 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 a TC and producing the TM. When some 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, even though none of the DPUs can accomplish $\mathcal{G}$ on its own, it might be the case that the operational components of both DPUs can together 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 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*. Specifically, it reactivates DPU_A and tries to assign the failed task to its corresponding component. 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:

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

Each module relies on its components to achieve the tasks required to accomplish $\mathcal{G}$. 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 remains reachable while there exists at least one *operational* component associated with each task. Formally, it can be specified as:

$$\mathcal{M} \models \Box \mathcal{O}_s, \quad \text{where} \;\; \mathcal{O}_s \equiv \forall t \in \mathcal{T} \cdot (\exists c \in \mathcal{C} \cdot \Phi(t,c) \wedge \mathcal{O}(c))$$

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 a fault tolerance strategy, it is important to consider not only reachability of functional goals but also their performance and reliability aspects.

In engineering, reliability is usually measured by the probability that the system 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, it is 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. 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 [6]:

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

The formulas above are specified using PRISM notation. 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 modelling in Event-B to formally define the architecture of a dynamically reconfigurable system, and on the probabilistic extension of Event-B to create models for assessing system reliability and performance. The next section briefly describes Event-B and its probabilistic extension.

3 Modelling in Event-B and Probabilistic Analysis

3.1 Modelling and Refinement in Event-B

Event-B 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* [1], which encapsulates the model state, represented as a collection of variables, and defines operations on the state, i.e., it describes the behaviour of a modelled system. Usually, a machine has an accompanying component, called *context*, which includes user-defined sets, constants and their properties given as a list of model axioms. 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*.

The dynamic behaviour of the system is defined by a 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 event is *enabled*. If several events are enabled at the same time, any of them can be chosen for execution nondeterministically.

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract 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, is demonstrated by discharging the relevant proof obligations generated by the Rodin platform [11]. 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 of an Event-B machine into continuous time Markov chain – CTMC (the details can be found in [15]). To achieve this, we augment all events of the machine with 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. Let also $\mathcal{I}$ be the model invariant. 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 $\rightarrow$:*

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

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 denote by $\lambda_e(\sigma,\sigma')$ 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 modify Definition 1 as follows.

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

$$\frac{\sigma, \sigma' \in \Sigma \;\wedge\; \sigma' \in \bigcup_{e \in \mathcal{E}_\sigma} \mathsf{after}(e)}{\sigma \xrightarrow{\Lambda} \sigma'}, \quad \textit{where } \Lambda = \sum_{e \in \mathcal{E}_\sigma} \lambda_e(\sigma, \sigma').$$

Definition 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 quantitative 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 to $TRUE$. 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 is decomposed into n tasks, and can be achieved by a sequential execution of one task after another. We also assume that the id of each task is defined by its execution order. Initially, when the goal is assigned, 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* that maps the id of a task to its state, i.e., $task \in 1..n \rightarrow \{OK, NOK, ND\}$.

We omit showing the events of the refined model (the complete development can be found in [13]). 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 the tasks:

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

Introducing Abstract Communication. In the second refinement step we 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 failed task status to NOK. Essentially, the refined model introduces an abstract communication via shared memory. The following gluing invariants allow us to prove the refinement:

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

As discussed in Section 2, each task is independently executed by a separate component of a high-level module. Hence, by substituting the id of a task with the id of the corresponding component, i.e., performing a data refinement with the gluing invariant

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

we specify a *non-redundant* system architecture. This invariant trivially defines the relation Φ. Next we demonstrate how to introduce either a triplicated architecture or duplicated architecture with a dynamic reconfiguration by refinement.

4.2 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:

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

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 [13]. Please note, that these invariants allows us to mathematically prove that the Event-B model preserves the desired system architecture.

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:

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

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 Event-B model 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. As a result, to verify correctness of the models we discharged 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.

Note that the described development for a generic system can be easily instantiated to formally derive fault tolerant architectures of DPU. The goal of DPU – handling the scientific TC by producing TM – is decomposed into four tasks that define the production of data by the satellite's 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 handled by the corresponding computing component of DPU. The high-level modules A, B and C correspond to three identical DPUs that control handling of scientific TC – 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

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 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 the probabilistic symbolic model checker PRISM to evaluate reliability and performance of the proposed models.

Due to the space constraints, we omit showing the PRISM specifications in the paper, they can be found in [13]. The guidelines for Event-B to PRISM model transformation can be found in our previous work [14].

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 relia bility and performance is optimal for the considered system.

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	$\frac{1}{6\cdot 10^7}$

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.

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.

[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.

A large variety of aspects of the dynamic reconfiguration has been studied in the last decade. For instance, Wermelinger et al. [17] 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 contrast, 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 [3] or integrity [16] 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 [4,10]. However, these approaches are characterised by a high degree of uncertainty in achieving fault tolerance that is unsuitable for the satellite systems. The work [5] 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 reconfiguration in the presence of parallelism and complex component interdependencies.

References

1. Abrial, J.-R.: Modeling in Event-B. Cambridge University Press (2010)
2. BepiColombo: ESA Media Center, Space Science, `http://www.esa.int/esaSC/SEMNEM3MDAF_0_spk.html`
3. Caporuscio, M., Di Marco, A., Inverardi, P.: Model-Based System Reconfiguration for Dynamic Performance Management. J. Syst. Softw. 80, 455–473 (2007)
4. de Castro Guerra, P.A., Rubira, C.M.F., de Lemos, R.: A Fault-Tolerant Software Architecture for Component-Based Systems. In: Architecting Dependable Systems, pp. 129–143. Springer (2003)
5. Goldsby, H.J., Sawyer, P., Bencomo, N., Cheng, B., Hughes, D.: Goal-Based Modeling of Dynamically Adaptive System Requirements. In: ECBS 2008, pp. 36–45. IEEE Computer Society (2008)
6. Grunske, L.: Specification Patterns for Probabilistic Quality Properties. In: ICSE 2008, pp. 31–40. ACM (2008)
7. Kelly, T.P., Weaver, R.A.: The Goal Structuring Notation – A Safety Argument Notation. In: DSN 2004, Workshop on Assurance Cases (2004)
8. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
9. van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: RE 2001, pp. 249–263. IEEE Computer Society (2001)
10. de Lemos, R., de Castro Guerra, P.A., Rubira, C.M.F.: A Fault-Tolerant Architectural Approach for Dependable Systems. IEEE Software 23, 80–87 (2006)
11. Rodin: Event-B Platform, `http://www.event-b.org/`

12. Space Engineering: Ground Systems and Operations – Telemetry and Telecommand Packet Utilization: ECSS-E-70-41A. ECSS Secretariat (January 30, 2003), http://www.ecss.nl/
13. Tarasyuk, A., Pereverzeva, I., Troubitsyna, E., Latvala, T., Nummila, L.: Formal Development and Assessment of a Reconfigurable On-board Satellite System. Tech. Rep. 1038, Turku Centre for Computer Science (2012)
14. Tarasyuk, A., Troubitsyna, E., Laibinis, L.: Quantitative Reasoning about Dependability in Event-B: Probabilistic Model Checking Approach. In: Dependability and Computer Engineering: Concepts for Software-Intensive Systems, pp. 459–472. IGI Global (2011)
15. Tarasyuk, A., Troubitsyna, E., Laibinis, L.: Formal Modelling and Verification of Service-Oriented Systems in Probabilistic Event-B. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) IFM 2012. LNCS, vol. 7321, pp. 237–252. Springer, Heidelberg (2012)
16. Warren, I., Sun, J., Krishnamohan, S., Weerasinghe, T.: An Automated Formal Approach to Managing Dynamic Reconfiguration. In: ASE 2006, pp. 18–22. Springer (2006)
17. Wermelinger, M., Lopes, A., Fiadeiro, J.: A Graph Based Architectural Reconfiguration Language. SIGSOFT Softw. Eng. Notes 26, 21–32 (2001)