Modeling patterns for the assessment of maintenance policies with AltaRica 3.0

Michel Batteux, Tatiana Prosvirnova, Antoine Rauzy

Modeling patterns for the assessment of maintenance policies with AltaRica 3.0

Michel Batteux[1], Tatiana Prosvirnova[2,3], and Antoine Rauzy[4]

[1] IRT SystemX, Palaiseau, France
michel.batteux@irt-systemx.fr
[2] Laboratoire Genie Industriel, CentraleSupélec, Gif-sur-Yvette, France
[3] ONERA/DTIS, UFTMiP, Toulouse, France
tatiana.prosvirnova@onera.fr
[4] Norwegian University of Science and Technology, Trondheim, Norway
antoine.rauzy@ntnu.no

Abstract. In this article, we present modeling patterns dedicated to the assessment of maintenance policies with AltaRica 3.0. From the analyst's perspective, these modeling patterns make models easier to design, to understand by stakeholders and to maintain. From a technical point of view, their design involves advanced features of AltaRica 3.0 that are worth presenting.

Keywords: Assessment of maintenance policies · AltaRica 3.0 · modeling patterns

1 Introduction

AltaRica 3.0 is an object-oriented modeling language dedicated to probabilistic risk and safety analyses of complex technical systems [5]. It is of primary importance, in order to make the modeling process efficient (in AltaRica 3.0 as with any other modeling formalism), to reuse as much as possible modeling components. In AltaRica 3.0, reuse is mostly achieved by the design of modeling patterns, i.e. examples of models representing remarkable features of the system under study. Once identified, patterns can be duplicated and adjusted for specific needs. Patterns are actually pervasive in engineering, see e.g. [8, 7]. Patterns are not only a mean to organize and to document models, but also and more fundamentally a way to reason about systems under study.

In this article we present modeling patterns to represent and to assess maintenance policies with AltaRica 3.0. We focus actually on corrective maintenance policies (components are repaired only when they are failed) taking into account that resources required to perform them (such as the number of repairmen or spare parts) may be limited. We show different modeling approaches, involving advanced features of AltaRica 3.0, such as the synchronization of events or the aggregation of prototypes.

The contribution of this article is thus twofold: first, it provides effective modeling patterns for the assessment of maintenance policies; second, it demonstrates the interest of AltaRica 3.0 advanced modeling constructs.

The remainder of this article is organized as follows. Section 2 introduces a case study that we use throughout the article to illustrate the presentation. Section 3 makes a brief description of the AltaRica 3.0 modeling language. Section 4 presents the maintenance policy modeling, according to three different modeling patterns. Section 5 provides some results on the three corresponding models. Finally, section 6 concludes the article and discusses future works.

2 Illustrative example

Fig. 1 shows a system made of two subsystems: an equipment under control and a control system.We focus our study on the latter. This system is made of three sensors, a controller and two actuators. The controller is made of three data acquisition units (one per sensor) and a voter, also called a logic solver, which works according to a 2-out-of-3 logic. Each actuator is made of two components.

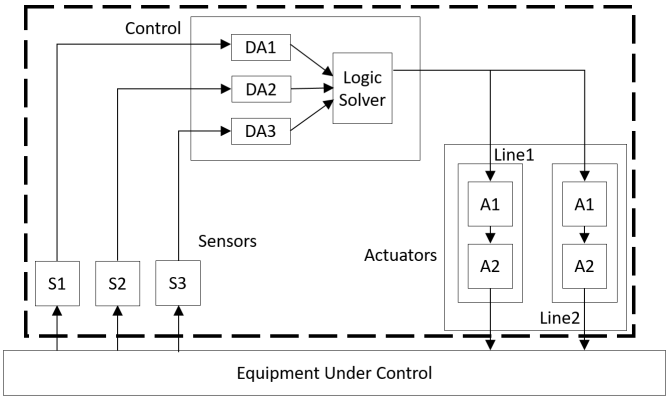

Fig. 1. An equipment under control and its control part.

All the components may fail in operation and be repaired. Failure rates (h^{-1}) are respectively 10^{-5} for sensors, 10^{-6} for data acquisition units, 10^{-8} for the logic solver and 10^{-6} for actuators. A maintenance operation is launched when the system as a whole is failed, i.e. if either two or more sensors are failed or two or more data acquisition modules are failed or the voter is failed or the two actuators are failed. All failed components are repaired during the maintenance operation and can be considered as good as new after. Failed components are repaired one by one. Mean times to repair components are one shift, i.e. 8 hours for actuators, 4 hours for sensors, data acquisition units and the logic solver. To accelerate maintenance operations, two repairers are involved (and can thus repair components in parallel).

The objective of this study is to calculate, for example, the system operational availability during its mission time, taking into account maintenance policies of the components.

3 AltaRica 3.0 modeling

3.1 The AltaRica 3.0 modeling language

AltaRica 3.0 is an event-based and object-oriented modeling language dedicated to probabilistic risk and safety analyses of complex technical systems [9]. This language is the combination of two parts: the mathematical framework GTS, for Guarded Transition Systems ([10]-[4]) to describe the behavior of the system under study; the structuring paradigm S2ML, for System Structure Modeling Language ([3]), to organize the model.

The execution of an AltaRica 3.0 model, done by the mathematical framework GTS, is quite similar to other event-based formalisms. It means that when a transition is enabled, it is scheduled and will be potentially fired after its associated delay ([6]-[12]). These delays can be deterministic or stochastic. For stochastic delays, AltaRica 3.0 provides usual probability distributions: exponential, Weibull, uniform or user defined ones.

To structure an AltaRica 3.0 model, S2ML provides the appropriate primitives. S2ML unifies the two main structuring paradigms for modeling languages: object-oriented and prototype-oriented. With S2ML, one can design the model in two ways. The 'top-down' approach: the system is considered at its highest level and modeling patterns are mainly used; it is the realm of prototype-oriented. The "bottom-up' approach: the system is considered at its lowest level (the components) and libraries of components are mainly used: it is the realm of object-oriented.

Two main structural constructs can be used in AltaRica 3.0: a 'block' and a 'class'. A class is an "on-the-shelf", reusable modeling component. It is defined and then can be instanced in a model, or inherited by another class or block. A block is a modeling component with a unique instance, as opposed to a class which can have several instances. The definition of a block is also its (unique) instance. More information can be found in [3].

3.2 Modeling with AltaRica 3.0

To design the AltaRica 3.0 model of the system depicted Fig. 1, we start by modeling the main part. We only consider that this main part contains a set of hierarchically ordered components, without thinking about how these components are internally designed.

Main part of the AltaRica 3.0 model The main part is given Fig. 2. It is defined with the block `System`, which contains two parts: one with the declaration of the different structural elements, the other defining the behavior.

The main part represents the hierarchy of declared components and the links between them (the behavioral part). It is composed of:

- Three instances `S1`, `S2` and `S3`, of the class `Sensor`. We only assume that this class `Sensor` contains two flow variables `in` and `out` (we can see them in the second part defining the behavior).

```
block System
  // Declaration of elements
  Sensor S1, S2, S3;
  block Control
    DataAcquisition DA1, DA2, DA3;
    block LogicSolver
      extends RComponent (lambda = 1.0e-8, mu = 4);
      Boolean in1, in2, in3, out (reset = false);
      assertion
        out := if vs == WORKING
               then (in1 and in2) or (in1 and in3) or (in2 and in3)
               else false;
    end
    assertion
      LogicSolver.in1 := DA1.out;
      LogicSolver.in2 := DA2.out;
      LogicSolver.in3 := DA3.out;
  end
  block Actuators
    block Line1
      Actuator A1, A2;
      assertion
        A2.in := A1.out;
    end
    clones Line1 as Line2;
  end
  observer Boolean TE = (Actuators.Line1.A2.out == false) and
                        (Actuators.Line2.A2.out == false);
  // Definition of the behavior
  assertion
    S1.in := true; S2.in := true; S3.in := true;
    Control.DA1.in := S1.out;
    Control.DA2.in := S2.out;
    Control.DA3.in := S3.out;
    Actuators.Line1.A1.in := Control.LogicSolver.out;
    Actuators.Line2.A1.in := Control.LogicSolver.out;
end
```

Fig. 2. AltaRica 3.0 code for the main part.

– A block `Control`, which is a sub-block of the main block `System`. This block declares three instances `DA1`, `DA2` and `DA3` of the class `DataAcquisition`. It declares an internal block `LogicSolver`, which inherits from the class `RComponent`. We assume that the class `RComponent` represents reparable components and contains two parameters `lambda` and `mu` that we overload with the values 10^{-8} and 4. This inheritance means that the block `LogicSolver` is a reparable component: it takes the features of the class `RComponent`. Furthermore, the block `LogicSolver` declares four flow variables `in1`, `in2`, `in3` and `out`. These variables are used in the second part defining the behavior of the `LogicSolver`: the assertion defines the external behavior according to the internal behavior, i.e. the update of the variable

`out` according to the other variables `in1`, `in2`, `in3` and a state variable `vs` (an internal variable) coming from the inherited class `RComponent`. Finally the block `Control` specifies the external behavior of its sub-parts in the assertion.
- A block `Actuator` declaring a sub-block `Line1`. `Line1` represents the first line of actuators. It is composed of two instances `A1` and `A2` of the class `Actuator`. These actuators are linked thanks to the assertion. The sub-block `Line1` is cloned: a copy of `Line1` is made and named `Line2`. One can notice that these two blocks `Line1` and `Line2` will independently live their own lives: changes into one block (e.g. the update of a variable, or the firing of a transition) has no impact on the other.
- Finally, a Boolean observer `TE` (Top Event) is declared. This observer observes if the two flow variables `out`, coming from the two actuators `A2` of the two lines, are false.

After the first declarative part, the main block `System` defines the assertion, which describes how the sub-parts (the sensors, the control and the actuators) are linked together and with the environment. We assume here that the equipment under control cannot fail and the sensors always receive a correct value as input (i.e. the value `true` because we consider Boolean variables).

Library of components The main part of the AltaRica 3.0 model contains different components, which are instances of classes. These classes are defined in a dedicated library and are depicted Fig. 3. The class `RComponentIO` implements a generic reparable component with one input and one output. It inherits from another class `RComponent`, the same as the one inherited by the component `LogicSolver` of the block `Control`. This inheritance means that `RComponentIO` takes the features of `RComponent`. `RComponentIO` declares two flow variables `in` and `out`, which are used in the second part: the assertion defining the external behavior by updating the variable `out`, according to the variable `in` and the state (internal) variable `vs`, coming from the inherited class `RComponent`. Finally, the classes corresponding to the components `Sensor`, `DataAcquistion` and `Actuator` are defined. They inherit from the class `RComponentIO` and overload the values of the parameters `lambda` and `mu`.

4 Modeling pattern for maintenance

The (part of the) AltaRica 3.0 model, presented previously, does not integrate the behavioral description of a reparable component, as well as the maintenance policy according to the limited number of repairers. We start with the definition of two AltaRica 3.0 elements in Fig. 4: a domain and an operator. The domain `SDomain` defines four values, `WORKING`, `FAILED`, `WAITING_REPAIR` and `REPAIR`, which will be used after to define types of elements (e.g. variables, parameters or observers). The operator `IsNotFailed` returns an integer value (`1` or `0`) according to the value of the argument `aState`, of type `SDomain`.

```
class RComponentIO
  extends RComponent;
  Boolean in, out (reset= false);
  assertion
    out := vs == WORKING and in;
end

class Sensor
  extends RComponentIO (lambda = 1.0e-5, mu = 4);
end

class DataAcquisition
  extends RComponentIO (lambda = 1.0e-6, mu = 4);
end

class Actuator
  extends RComponentIO (lambda = 1.0e-6, mu = 8);
end
```

Fig. 3. AltaRica 3.0 library of components.

```
domain SDomain {WORKING, FAILED, WAITING_REPAIR, REPAIR}

operator Integer IsNotFailed(SDomain aState)
  if (aState != FAILED) then 1 else 0
end
```

Fig. 4. AltaRica 3.0 code for the domain and operator.

4.1 Maintenance policies

According to the European standard NF EN 13306 X 60-319, there are two main kinds of maintenance. The first one is the corrective maintenance, which is carried out after failure detection and is aimed at restoring an asset to a condition in which it can perform its intended function. This kind of maintenance implies an unavailability either of the overall or of a part of the system. The second one is the preventive maintenance, which aims at performing an intervention before the occurrence of a failure. Different kinds of preventive maintenance also exist. Planned maintenance is realized according to a specific bound reached by the system (e.g. date, time of running, distance travelled, etc.). Condition-based maintenance is realized according to a monitoring of the system. Finally, predictive maintenance uses sensor data to monitor a system, then continuously evaluates it against historical trends to predict failure before it occurs.

AltaRica 3.0 is a flexible and versatile tool. Maintenance policies can be taken into account with AltaRica 3.0, of course by realizing some kinds of abstraction. For example, [11] presents a modelling methodology for the assessment of preventive maintenance on a compressor drive system. In the following, we focus on the corrective maintenance policy at the component level. We introduce some condition at system level, to realize maintenance actions, i.e. when a sufficient

number of components are failed: at least two sensors are failed, or two data acquisition units or the logic solver unit or one actuator per lines. Furthermore we consider only two repairers to repair failed components.

In Fig. 5, two transitions representing the maintenance policy are added to the main part of the AltaRica 3.0 model.

```
block System
  // Declaration of elements
  ...
  event maintenanceReq (delay = Dirac(0.0), hidden = true);
  event maintenance (delay = Dirac(0.0));
  // Definition of the behavior
  transition
    maintenanceReq: (IsNotFailed(S1.vs) + IsNotFailed(S2.vs)
                              + IsNotFailed (S3.vs)) <= 1
                     or ...
                    // The maintenance policy involving all components
                    -> skip;
    maintenance: !maintenanceReq
                & ?S1.maintenance & ?S2.maintenance & ? ...
                // All transitions  maintenance  of all components
  assertion
    ...
end
```

Fig. 5. Main part of the AltaRica 3.0 model with the maintenance policy.

The two events `maintenanceReq` and `maintenance` are defined with an instantaneous delay, represented by the distribution `Dirac(0.0)`. It means that the transitions, labelled by these instantaneous events, are fired as soon as they are enabled. In the following, we will associate the transition and its label, if there is no ambiguity.

The transition `maintenanceReq` specifies the maintenance policy in the guard, by using the operator `IsNotFailed` with the states of the components: at least two sensors are failed. This transition is hidden (the attribute `hidden` of the labelling event is set to `true`), meaning it will never be fired alone, it must be synchronized with other transitions. Furthermore, no action is associated to this transition: the instruction `skip`. By defining the two events, and their associated transitions, `maintenanceReq` and `maintenance`, we totally separate the definition of the maintenance policy (in the guard of the transition `maintenanceReq`) and the action of maintenance (in the transition `maintenance`).

The transition `maintenance` synchronizes the transition `maintenanceReq` and all the transitions `maintenance` of the components. The symbol `!` means that the transition `maintenanceReq` is mandatory: to fire this transition `maintenance`, the guard of the transition `maintenanceReq` must be true. Conversely, the symbol `?` means that the transitions `maintenance` of the components are optional: the transition can be fired if its guard is true, but it is not required. When the

transition `maintenance` of a component is fired, it changes its state from `FAILED` to `WAITING_REPAIR`: it has to wait the availability of a repairer.

The behavior of a reparable component `RComponent` is described according the state machine depicted Fig. 6. Nodes represent the different states of the component, which are from the domain `SDomain`. Edges represent transitions between states. It is a generic behavior which will be adapted according to the considered modeling pattern.

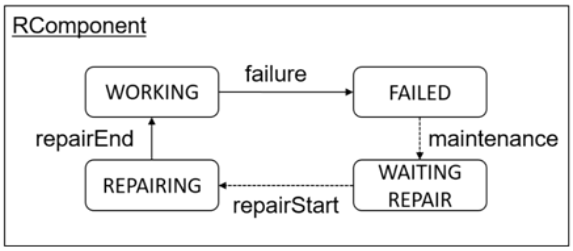

Fig. 6. State machine of a reparable component.

In the following we present how to model the availability of a repairer to repair a component, with three different patterns: by propagation of flow variables, by synchronizing events, or by using the virtual aggregation.

4.2 Repair by propagation of flow variables

Fig. 7 represents the reparable component used for the pattern propagation of flow variables. The component is initialized to the state `WORKING`: the attribute `init` of the state variable `vs` is set to `WORKING`. The event `maintenance` is synchronized at system level: its attribute `hidden` is set to `true`. Two flow variables are defined. `rUsed` indicates that a repairer repairs the component. `rAvailable` takes the value of the availability of a repairer, and is used in the guard of the transition `repairStart`, i.e. to launch the repair of the component.

Fig. 8 shows the additions to the main part of the AltaRica 3.0 model, which are needed for the pattern by propagation of flow variables. The parameter `repairer` defines the number of repairers (i.e. 2). The flow variable `rUsed` provides the number of used repairers to repair components. This variable is updated in the assertion by adding all flow variables `rUsed` of the components. Finally the flow variable `rAvailable` provides the information that a repairer is available. It is updated according to the value of `rUsed` and `repairer`. Then, it is used to update all the variables `rAvailable` of the components.

```
class RComponent
  SDomain vs (init = WORKING);
  Integer rUsed (reset = 0);
  Boolean rAvailable (reset = false);
  parameter Real lambda = 1.0e-5;
  parameter Real mu = 2.0;
  event failure (delay = exponential(lambda));
  event maintenance (delay = Dirac(0.0), hidden = true);
  event repairStart (delay = Dirac(0.0));
  event repairEnd (delay = Dirac(mu));
  transition
    failure: vs == WORKING -> vs := FAILED;
    maintenance: vs == FAILED -> vs := WAITING_REPAIR;
    repairStart: vs == WAITING_REPAIR and rAvailable -> vs := REPAIR;
    repairEnd: vs == REPAIR -> vs := WORKING;
  assertion
    rUsed := if vs == REPAIR then 1 else 0;
end
```

Fig. 7. AltaRica 3.0 code of the reparable component for the pattern by propagation of flow variables.

```
block System
  // Declaration of elements
  ...
  parameter Integer repairer = 2;
  Integer rUsed (reset = 0);
  Boolean rAvailable (reset = false);
  // Definition of the behavior
  transition
    ...
  assertion
    ...
    rUsed := S1.rUsed + S2.rUsed + S3.rUsed + Control.DA1.rUsed + ...
    // All flow variables  rUsed  of components
    rAvailable := rUsed < repairer;
    S1.rAvailable := rAvailable;
    S2.rAvailable := rAvailable;
    S3.rAvailable := rAvailable;
    Control.DA1.rAvailable := rAvailable;
    ...
    // All flow variables  rAvailable  of all components
    // take the value of the variable  rAvailable
end
```

Fig. 8. Main part of the AltaRica 3.0 model with the pattern by propagation of flow variables.

4.3 Repair by synchronizing events

Fig. 9 represents the reparable component used for the pattern by synchronizing events. The three events `maintenance`, `repairStart` and `repairEnd` have their attributes `hidden` set to `true` in order to synchronize them at the system level.

```
class RComponent
  SDomain vs (init = WORKING);
  parameter Real lambda = 1.0e-5;
  parameter Real mu = 2.0;
  event failure (delay = exponential(lambda));
  event maintenance (delay = Dirac(0.0), hidden = true);
  event repairStart (delay = Dirac(0.0), hidden = true);
  event repairEnd (delay = Dirac(mu), hidden = true);
  transition
    failure: vs == WORKING -> vs := FAILED;
    maintenance: vs == FAILED -> vs := WAITING_REPAIR;
    repairStart: vs == WAITING_REPAIR -> vs := REPAIR;
    repairEnd: vs == REPAIR -> vs := WORKING;
end
```

Fig. 9. AltaRica 3.0 code of the reparable component for the pattern by synchronizing events.

Fig. 10 represents the additions to the main part of the AltaRica 3.0 model used for the pattern by synchronizing events. A new block `Repairer` is added. It defines the behavior of the repairer crew according to dedicated events. It is basic: it defines the start and stop of a repair if one of the two repairers is available. Then two new events `repairStartC` and `repairEndC` are defined for all components `C`. These events are used in the transition part to synchronize the own events `repairStart` and `repairEnd` of the component `C`, with the events `repairStart` and `repairEnd` of the block `Repairer`.

4.4 Repair by virtual aggregation

For the pattern with the virtual aggregation, the class defining the component is represented Fig. 11. This class is the same as the one defined in Fig. 9. In addition, it virtually aggregates a new element of type `T`, which is used in the class with the alias `t`. This virtually aggregated element is used in the class within the two transitions `repairStart` and `repairEnd`, by synchronizing them with two events of `t`. When the class is instantiated (i.e. when an object with this class as type is declared), this aggregation is resolved by indicating a real object (an instance of a class or a block to be used instead of `T`). This object must be compatible according to the use of it in the declared class.

Fig. 12 represents the additions to the main part of the AltaRica 3.0 model used for the pattern by virtual aggregation. A new block `Repairer` is defined. It is the same as for the previous pattern by synchronizing events. This block is used when all classes are instantiated, inheriting from the class `RComponent`, to resolve the virtually aggregated element. The resolution of the virtual aggregation is done by the attribute (`virtual T = main.Repairer`). It means that the virtual element `T`, used in the class with the alias `t`, is equal to the block `Repairer`. The keyword `main`, preceding the word `Repairer` with a dot between them, indicates

```
block System
  // Declaration of elements
  ...
  block Repairer
    parameter Integer repairer = 2;
    Integer rAvailable (init = repairer);
    event repairStart, evRepairEnd (delay = Dirac(0.0), hidden = true);
    transition
      repairStart: rAvailable > 0 -> rAvailable := rAvailable - 1;
      repairEnd: true -> rAvailable := rAvailable + 1;
  end
  event repairStartS1 (delay = Dirac(0.0));
  event repairEndS1 (delay=exponential(S1.mu));
  event repairStartS2 (delay = Dirac(0.0));
  event repairEndS2 (delay=exponential(S2.mu));
  ...
  // For all components, two new events 'repairStart' and 'repairEnd'
  // Definition of the behavior
  transition
    ...
    repairStartS1: !S1.repairStart & !Repairer.repairStart;
    repairEndS1: !S1.repairEnd & !Repairer.repairEnd;
    repairStartS2: !S2.repairStart & !Repairer.repairStart;
    repairEndS2: !S2.repairEnd & !Repairer.repairEnd;
    ...
    // All events previously defined are used to synchronize the events
    // of the considered component with the events of the block Repairer
  assertion
    ...
end
```

Fig. 10. Main part of the AltaRica 3.0 model with the pattern by synchronizing events.

```
class RComponent
  embeds virtual T as t;
  SDomain vs (init = WORKING);
  parameter Real lambda = 1.0e-5;
  parameter Real mu = 2.0;
  event failure (delay = exponential(lambda));
  event maintenance (delay = Dirac(0.0), hidden = true);
  event repairStart (delay = Dirac(0.0));
  event repairEnd (delay = Dirac(mu));
  transition
    failure: vs == WORKING -> vs := FAILED;
    maintenance: vs == FAILED -> vs := WAITING_REPAIR;
    repairStart: !t.repairStart & vs == WAITING_REPAIR -> vs := REPAIR;
    repairEnd: !t.repairEnd & vs == REPAIR -> vs := WORKING;
end
```

Fig. 11. AltaRica 3.0 code of the reparable component for the pattern by virtual aggregation.

that the object `Repairer` is declared at the main hierarchical level of the model, i.e. the block `System`.

```
block System
  // Declaration of elements
  block Repairer
    parameter Integer repairer = 2;
    Integer available (init = repairer);
    event repairStart (delay = Dirac(0.0), hidden=true);
    event repairEnd (delay = Dirac(0.0), hidden=true);
    transition
      repairStart: available > 0 -> available := available - 1;
      repairEnd: true -> available := available + 1;
  end
  Sensor S1, S2, S3 (virtual T = main.Repairer);
  block Control
    DataAcquisition DA1, DA2, DA3 (virtual T = main.Repairer);
    block LogicSolver
      extends RComponent (lambda = 1.0e-8, mu = 4.0,
                          virtual T = main.Repairer);
      ...
    end
    ...
  end
  block Actuator
    block Line1
      Actuator A1, A2 (virtual T = main.Repairer);
      ...
    end
    ...
  end
  ...
  // Definition of the behavior
  ...
end
```

Fig. 12. Main part of the AltaRica 3.0 model with the pattern by virtual aggregation.

5 Experiments

Table 1 shows different quantitative features of the models for these three patterns: the model `Mf` for the pattern propagation of flow variables, the model `Me` for the pattern synchronizing events, and the model `Mv` for the pattern virtual aggregation. The first two features are done for the designed models, whereas the others are done for the compiled models. The difference between models `Mf` and `Me` concerns the number of flow variables and their updates in the assertion: more important in the model `Mf`. Nevertheless, this result hides the additional events, per components, defined in `Me`; which is not indicated in this table but

can be found thanks to the number of lines. Furthermore models `Me` and `Mv` seem to be equal when compiled. More precisely, they are equal and it is totally normal because the virtual aggregation pattern considers synchronization of events, but from a generic way: by including it directly into the class. The main difference is thus according to the size of the designed models. In the following, especially with the assessment tools, these two models are used equally. Finally it is recommended to use the pattern by virtual aggregation. On the one hand, fewer errors are made at the design phase. On the other hand, it defines fewer flow variables, than the pattern by propagation of flow variables, which has a cost when the model is evaluated by the assessment tools: these variables are updated in the assertion and there is a computational cost for that at runtime.

Table 1. Quantitative features for the three patterns.

Features	`Mf`	`Me`	`Mv`
Number of lines, *at design*	143	177	130
Number of lines of the main block (`System`), *at design*	84	124	66
Number of state variables	11	12	12
Number of flow variables	48	24	24
Number of events	34	34	34
Number of lines of the assertion	48	24	24

Some experiments are also realized in order to evaluate the Boolean observer `TE`. This observer indicates when the values of the two flow variables `out`, of the two actuators `A2` of the two lines, are false. It means that the system is failed. We used the AltaRica 3.0 stochastic simulator of the OpenAltaRica platform ([1]) to perform the experiments.

Table 2 shows the means of fired transitions for the following number of generated histories: 10^5, 10^6 and 10^7, for a mission time equal to 20 years (175200 units of time). The execution time, to generate these histories, has not been taken into account. On the one hand, it is quick: 1-2 minutes for 10^7 histories on a personal laptop. On the other hand, our interest does not focus on performance analysis of the tool. Elements can be found in [2] or [1].

Table 2. Means of fired transitions.

Number of histories	10^5	10^6	10^7
`Mv`	22.9015	22.9089	22.9002
`Mf`	22.9133	22.8936	22.8997

Table 3 shows statistics provided by the stochastic simulator, on the observer `TE`, for 10^7 generated histories. The considered mission time was 20 years and we

also considered different time instants: 5, 10 and 15 years. We focused on the two following statistics. '`had-value`' (denoted `h-v`) is equal to 1 if the observer took the value `true` for a non-null period at least once from time 0 to time d (with d equals to 5, 10, 15 or 20 years), and 0 otherwise. '`number-of-occurrences`' (denoted `n-o`) is equal to the number of dates the observer started taking the value `true` over the time period $[0, d]$ (with d equals to 5, 10, 15 or 20 years). The obtained results are quite similar.

Table 3. Statistics on the observer.

		`Mf`	`Mv`
5 years	`h-v`	0.315882	0.315438
	`n-o`	0.343274	0.343362
10 years	`h-v`	0.646718	0.646189
	`n-o`	0.848165	0.849419
15 years	`h-v`	0.830037	0.829901
	`n-o`	1.35688	1.3594
20 years	`h-v`	0.919748	0.920068
	`n-o`	1.86367	1.86752

6 Conclusion

In this article, we presented three different modeling patterns with the AltaRica 3.0 modeling language to represent a corrective maintenance policy on a set of components, with a limited number of repairers. A main modeling part of the system has first been proposed. This part is common to the three modeling patterns and was realized with a 'top-down' approach. Behaviors of the components were not directly defined. Furthermore we included the maintenance policy into this main part.

Regarding the limited number of repairers, meaning their assignment to failed components according to their availability, the three different patterns were presented. The first one uses the propagation of flow variables. This pattern is not difficult but error prone. In addition it could be less efficient during execution. In fact it duplicates the number of flow variables, thus the number of elements to update in the assertion. The second and third patterns use the synchronization of events. The second one defines these synchronizations by hand, for all the components. The third one uses the virtual aggregation to integrate these synchronizations into a generic class. This third pattern is more easy to design models, and thus less error prone.

These modeling patterns for maintenance policies with AltaRica 3.0 open the way to new opportunities. On the one hand, modeling patterns allow engineers to model simply and efficiently classical safety features: e.g. periodically tested component, (warm) redundancies, shared resources, common cause failures, etc.

Some of these patterns are defined in libraries. For the others, it is possible, in an easy way, to design tools helping engineers to create models with such patterns. On the other hand, it is possible to extend the use of AltaRica 3.0 modeling language to study other performance indicators than those for safety, e.g. scheduling maintenance policies.

References

1. Aupetit, B., Batteux, M., Rauzy, A., Roussel, J.M.: Improving performance of the AltaRica 3.0 stochastic simulator. In: Podofillini, L., Sudret, B., Stojadinovic, B., Zio, E., Kröger, W. (eds.) Proceedings of Safety and Reliability of Complex Engineered Systems: ESREL 2015. pp. 1815–1824. CRC Press (September 2015)
2. Aupetit, B., Batteux, M., Rauzy, A., Roussel, J.M.: Vers la définition d'un kit d'évaluation pour les simulateurs stochastiques. In: Actes du Congrès Lambda-Mu 20 (actes électroniques). Institut pour la Maîtrise des Risques (IMdR), Saint-Malo, France (2016). https://doi.org/10.4267/2042/61811
3. Batteux, M., Prosvirnova, T., A.Rauzy: From models of structures to structures of models. In: 4th IEEE International Symposium on Systems Engineering, ISSE 2018. Rome, Italy (October 2018)
4. Batteux, M., Prosvirnova, T., Rauzy, A.: Altarica 3.0 assertions: the why and the wherefore. Journal of Risk and Reliability (2017), article accepted
5. Batteux, M., Prosvirnova, T., Rauzy, A.: Altarica 3.0 in 10 modeling patterns. International Journal of Critical Computer-Based Systems **9**(1–2), 133–165 (2018). https://doi.org/10.1504/IJCCBS.2019.098809
6. Cassandras, C.G., Lafortune, S.: Introduction to Discrete Event Systems. Springer, New-York, NY, USA (2008)
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley professional computing series, Addison-Wesley, Boston, MA 02116, USA (October 1994)
8. Maier, M.W.: The art of systems architecting (2009)
9. Prosvirnova, T., Batteux, M., Brameret, P.A., Cherfi, A., Friedlhuber, T., Roussel, J.M., Rauzy, A.: The altarica 3.0 project for model-based safety assessment. In: Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS'2013. pp. 127–132. International Federation of Automatic Control, York, Great Britain (September 2013)
10. Rauzy, A.: Guarded transition systems: a new states/events formalism for reliability studies. Journal of Risk and Reliability **222**(4), 495–505 (2008). https://doi.org/10.1243/1748006XJRR177
11. Zhang, Y., Barros, A., Rauzy, A., Lunde, E.: A modelling methodology for the assessment of preventive maintenance on a compressor drive system. In: Haugen, S., Barros, A., van Gulijk, C., Kongsvik, T., Vinnem, J.E. (eds.) Safe Societies in a Changing World, Proceedings of European Safety and Reliability Conference (ESREL 2018). pp. 915–922. CRC Press, Trondheim, Norway (June 2018)
12. Zimmermann, A.: Stochastic Discrete Event Systems. Springer, Berlin, Heidelberg, Germany (2008)