HAL open science

Schemas de modelisation de politiques de maintenance de composants en Altarica 3.0

Michel Batteux, Tatiana Prosvirnova, Antoine Rauzy

To cite this version:

Michel Batteux, Tatiana Prosvirnova, Antoine Rauzy. Schemas de modelisation de politiques de maintenance de composants en Altarica 3.0. Congrès Lambda Mu 21, " Maîtrise des risques et transformation numérique : opportunités et menaces ", Oct 2018, Reims, France. hal-02063641

HAL Id: hal-02063641

https://hal.science/hal-02063641v1

Submitted on 11 Mar 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

SCHEMAS DE MODELISATION DE POLITIQUES DE MAINTENANCE DE COMPOSANTS EN ALTARICA 3.0

MODELING PATTERNS TO DESIGN MAINTENANCE POLICY OF COMPONENTS WITH ALTARICA 3.0

Michel Batteux
IRT SystemX
Saclay, France

Tatiana Prosvirnova
CentraleSupélec
Gif-sur-Yvette, France

Antoine Rauzy
NTNU
Trondheim, Norway

Résumé

Cette communication présente comment modéliser efficacement, en AltaRica 3.0, des politiques de maintenance de plusieurs composants d'un système en prenant en compte différentes caractéristiques, telles que un nombre limité de réparateurs, des réparations conditionnelles, etc. Nous montrons comment certaines primitives évoluées du langage, peuvent être utilisées pour rendre l'activité de modélisation plus efficace et simple. Puis nous montrons comment rendre générique ces 'types' de modélisation afin de définir des schémas de modélisation. Ces schémas pouvant être ensuite repris pour modéliser d'autres systèmes.

Summary

This article presents an efficient way to design AltaRica 3.0 models with different features: maintenance policy, limited number of repairers, etc. We use some advanced primitives of the AltaRica 3.0 modeling language to simplify the design of models with such features. This making the design activity efficient and easier. We also show that these features can be defined as modeling patterns. These modeling patterns could be reused to design other systems.

Introduction

AltaRica 3.0 est un langage de modélisation événementiel et orienté objet, dédié aux analyses probabilistes du risque de systèmes complexes (Prosvirnova *et al.*, 2013). Il permet de modéliser les systèmes à plus haut niveau que les formalismes classiques d'analyse du risque (arbres de défaillance, chaînes de Markov, réseaux de Petri stochastiques, etc.), sans augmenter la complexité des calculs d'indicateurs du risque.

Dans cette communication, nous utilisons le langage AltaRica 3.0 pour modéliser une politique de maintenance corrective sur les composants d'un système, en incluant une contrainte donnée par un nombre limité de réparateurs. Nous nous intéresserons de ce fait beaucoup plus à la modélisation en AltaRica 3.0, et finalement que peu à la manière dont peuvent être ensuite obtenus des indicateurs sur les modèles produits. Le lecteur intéressé par l'obtention d'indicateurs pourra se référer à Aupetit et *al.*, 2015 pour la simulation stochastique de modèles AltaRica 3.0, ou à Brameret et al. 2015 et Prosvirnova et al. 2015 pour la génération de chaînes de Markov ou d'arbres de défaillance à partir de modèles AltaRica 3.0.

La suite de la communication se décompose comme suit. La deuxième partie introduit un exemple applicatif sur lequel nous réaliserons la modélisation. La troisième partie décrit le langage de modélisation AltaRica 3.0. La quatrième partie propose une première modélisation qui sera générique pour toutes les politiques de maintenance et schémas de modélisation. Les politiques de maintenance et schémas de modélisation seront présentés en cinquième partie. La sixième partie réalisera une expérimentation sur les différents modèles produits. Enfin la dernière partie conclura cette communication.

Exemple

Nous considérons un système tel que représenté en Figure 1. Le système est constitué d'un équipement à contrôler et de son système de contrôle. Nous ne nous intéressons qu'au système de contrôle. Ce système de contrôle est constitué de différents composants. Trois capteurs qui obtiennent les informations sur l'équipement. Un contrôleur contenant trois unités d'acquisition de données et une unité logique de décision. Les trois unités d'acquisition sont reliées aux capteurs. L'unité logique de décision est de type deux sur trois et reliée aux unités d'acquisition. Enfin, quatre actionneurs répartis en série sur deux trains parallèles ; chaque train contenant deux actionneurs.

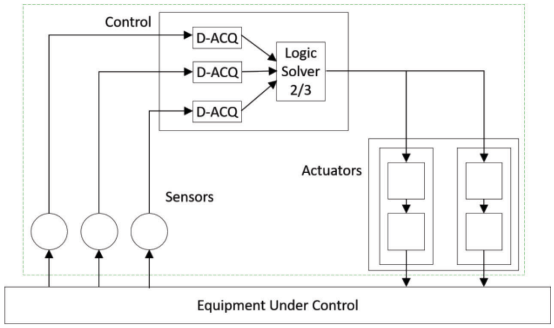

Figure 1. Système de contrôle d'un équipement

Tous ces composants peuvent tomber en panne et être réparés. Ces composants tombent en panne suivant une loi exponentielle de paramètres respectifs 10^{-5} pour les capteurs, 10^{-6} pour les unités d'acquisition de données, 10^{-8} pour l'unité logique de décision et 10^{-6} pour les actionneurs. Ils sont ensuite réparés, d'une part si un réparateur est disponible et d'autre part suivant une loi exponentielle de paramètres 10^{-2}.

Nous supposons qu'une équipe de réparateurs vienne réparer les composants lorsqu'un nombre suffisant de ceux-ci sont en panne : au moins deux capteurs, ou au moins deux unités d'acquisition de données, ou au moins l'unité logique de décision, ou au moins un actionneur par train. Enfin nous fixons le nombre de réparateurs à deux individus.

L'objectif d'étude d'un tel modèle est de calculer, par exemple, la disponibilité du système sous-jacent durant une certaine période de temps, en incluant les pannes et réparations des composants suivant des politiques spécifiques de maintenance. Comme indiqué en introduction, nous ne nous intéresserons que peu aux méthodes et outils de simulation permettant d'obtenir de

tels résultats, mais plus aux différentes modélisations possibles en AltaRica 3.0 pour rendre compte de ce système.

Le langage de modélisation AltaRica 3.0

AltaRica 3.0 est un langage de modélisation événementiel et orienté objet, dédié aux analyses probabilistes du risque de systèmes complexes (Prosvirnova *et al.*, 2013). Il permet de modéliser les systèmes à plus haut niveau que les formalismes classiques d'analyse du risque (arbres de défaillance, chaînes de Markov, réseaux de Pétri stochastiques, etc.), sans augmenter la complexité des calculs d'indicateurs du risque.

AltaRica 3.0 est la combinaison de deux éléments :

- Le cadre mathématique GTS, pour Guarded Transitions System (Rauzy, 2008 et Batteux et *al.* 2017) qui permet de rendre compte des aspects comportementaux que l'on souhaite modéliser, c'est-à-dire la sémantique d'exécution.
- Le paradigme de structuration S2ML, pour System Structure Modeling Language (Batteux et *al.* 2015), permettant de structurer les modèles.

L'exécution des modèles AltaRica 3.0, donnée par le cadre mathématique GTS, est similaire aux autres formalismes à événements discrets dans le sens où chaque fois qu'une transition est tirable, elle est planifiée et sera potentiellement tirée après un certain délai (Cassandras *et al.*, 2008), (Zimmerman, 1976). Ces délais peuvent être déterministes ou stochastiques ; et dans ce dernier cas, AltaRica 3.0 fournit les délais usuels (du type exponentiel, Weibull, etc.) et des délais empiriques.

La structuration d'un modèle AltaRica 3.0, donnée par le paradigme S2ML, permet de construire les modèles de différentes manières. S2ML unifie les deux paradigmes (de structuration) dominants des langages de modélisation, i.e. l'orienté objet et l'orienté prototype. Il permet de modéliser suivant deux approches combinables. L'approche dite 'top-down', avec une vision au niveau système permettant la réutilisation de schémas de modélisation, et donc utilisant le paradigme orienté prototype. L'approche dite 'bottom-up', avec une vision au niveau composants permettant la réutilisation de composants, définis en bibliothèques, et donc utilisant le paradigme orienté objet.

Deux principales constructions structurelles peuvent être déclarées en AltaRica 3.0 : Les '`block`' et les '`class`'. Elles correspondent aux éléments des formalismes orientés objet et prototype.

- Un élément '`class`', que nous nommerons classe pour la suite, représente une construction structurelle définissant un composant générique. Une classe est d'abord déclarée, en dehors de tout autre élément, puis est ensuite utilisée par instanciation ou héritage.
- Un élément '`block`', que nous nommerons block dans la suite, représente une construction structurelle ayant une unique occurrence. Un block est déclaré et directement utilisé. Il n'est pas possible d'instancier un block, mais il est néanmoins possible de le cloner afin d'en faire une copie.

La Table 1 compare ces deux constructions structurelles block et classe et indique, notamment, les opérations possibles.

Modélisation en AltaRica 3.0

Pour modéliser en AltaRica 3.0 le système de la Figure 1, nous allons combiner différentes approches. Nous commencerons par décrire la partie principale du système, sans nous soucier de ce que peuvent précisément contenir les composants de cette partie principale. Nous imposerons uniquement la présence de certains éléments dans ces composants.

Concept	Classe	Block
Définition	Composant générique	Composant ayant une unique occurrence
Utilisation	Instances multiples d'un composant stocké en bibliothèque	Occurrence unique
Réutilisation	Instanciation et héritage	Clonage et modification
Opérations	Composition héritage	Composition Héritage Agrégation Clonage

Table 1. Comparaison entre les blocks et les classes

1 Partie principale du modèle AltaRica 3.0

La partie principale du modèle AltaRica 3.0 du système de la Figure 1 est donnée par la Figure 2.

```
Block System
  // Declaration of elements

  Sensor S1, S2, S3 (lambda = 1.0e-5);

  block Control
    DataAcquisition DA1, DA2, DA3
                              (lambda = 1.0e-6);
    block LogicSolver
      extends Component (lambda = 1.0e-8);
      Boolean in1, in2, in3 (reset = false);
      Boolean 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 Actuator
    block Line1
      Actuator A1, A2 (lambda = 1.0e-6);
      assertion
        A2.in := A1.out;
    end
    clones Line1 as Line2;
  end

  observer Boolean TE =
              (Actuator.Line1.A2.out == false)
          and (Actuator.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;
    Actuator.Line1.A1.in :=
                        Control.LogicSolver.out;
    Actuator.Line2.A1.in :=
                        Control.LogicSolver.out;
end
```

Figure 2. Modélisation de la partie principale

Cette partie principale est donnée par un block nommé 'System'. Ce block principal contient ensuite deux parties : une partie de déclaration des éléments et une partie de définition du comportement. Ce schéma est repris à tous les niveaux dans un modèle AltaRica 3.0. Cette partie principale représente la hiérarchie des composants et les liens associés tels que dans la Figure 1.
La première partie de déclaration spécifie :

- Trois instances, nommées 'S1', 'S2' et 'S3', d'une classe 'Sensor'. Comme indiqué précédemment, nous ne supposons que très peu de choses sur cette classe 'Sensor' : elle contient un paramètre nommé 'lambda' qui est surchargé avec la valeur $1.0e^{-5}$, elle a deux variables de flux nommées 'in' et 'out' (visibles dans la seconde partie de définition du comportement).
- Un block nommé 'Control'. Ce block déclare trois instances 'DA1', 'DA2' et 'DA3' d'une classe 'DataAcquisition', en surchargeant la valeur d'un paramètre nommé 'lambda'. Elle déclare aussi un block, qui est donc un sous-block de 'Control' et un sous-sous-block de 'System', nommé 'LogicSolver'. Ce block 'LogicSolver' hérite d'une classe nommée 'Component', que nous supposons être une classe représentant des composants réparables. Cet héritage signifie donc que ce block 'LogicSolver' est un composant réparable, il en a donc les caractéristiques telles que définies dans la classe 'Component'. Outre cet héritage, ce 'LogicSolver' déclare quatre variables de flux 'in1', 'in2', 'in3' et 'out'. Ces quatre variables sont utilisées dans la seconde partie de ce 'LogicSolver' : l'assertion qui définit le comportement externe du composant, c'est-à-dire comment se met à jour la variable 'out' en fonction des autre variables 'in1', 'in2', 'in3' et d'une variable d'état 'vs' issue de la classe héritée 'Component'. Enfin le block 'Control' spécifie le comportement externe de ses différentes sous-parties au travers de l'assertion, c'est-à-dire comment sont reliés ses composants au travers de leurs variables de flux.
- Un block nommé 'Actuator' qui déclare un sous-block 'Line1' représentant la première ligne d'actionneurs. Cette ligne est, elle-même, constituée de deux instances 'A1' et 'A2' d'une classe 'Actuator' pour lesquelles un paramètre 'lambda' est surchargé. Ces deux actionneurs sont reliés entre eux via l'assertion. Ensuite au sein de ce block 'Actuator', le block 'Line1' est cloné, c'est-à-dire qu'une copie de ce block 'Line1' est réalisée et nommée 'Line2'. Il est à noter qu'à partir de cette copie, les deux blocks vont respectivement « mener leur propre vie » de manière indépendante, c'est-à-dire que les changements, par exemple des valeurs des variables, de l'un ne se feront pas sur l'autre.
- Enfin un observateur Booléen nommé 'TE' qui va, comme son nom l'indique, observer lorsque les deux variables 'out', des deux actionneurs 'A2' des deux lignes d'actionneurs, sont toutes les deux à faux.

Suite à cette première partie de déclaration, ce block principal 'System' définit l'assertion, c'est-à-dire comment sont reliées les sous-parties (capteurs, contrôleur et actionneurs), d'une part entre-elles et d'autre part avec l'environnement : on suppose que l'équipement sous contrôle fonctionne bien et que les capteurs reçoivent une valeur correcte en entrée (i.e. 'true' car nous avons considéré des variables Booléennes).

2 Composants en bibliothèque

La partie principale du modèle AltaRica 3.0, décrite précédemment fait appel à différents composants qui sont des instances de classes définies en bibliothèque. Nous commençons par définir en Figure 3 une classe générique de composants réparables avec une entrée et une sortie nommé 'ComponentIO'. Ce composant hérite d'une classe nommée 'Component', la même que celle héritée par le composants 'LogicSolver' du block 'Controller'. Il a donc les caractéristiques d'un composant réparable, telles que définies dans la classe 'Component', que nous définirons plus loin dans le document. Outre cet héritage, ce composant 'ComponentIO' déclare deux variables de flux 'in', et 'out', qui sont utilisées dans la seconde partie : l'assertion qui définit le comportement externe, c'est-à-dire comment se met à jour la variable 'out' en fonction de la variable 'in' et d'une variable d'état 'vs' issue de la classe héritée 'Component'.

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

Figure 3. Modélisation d'un composant réparable à une entrée et une sortie

Nous pouvons ensuite définir nos différents composants nécessaires 'Sensor', 'DataAcquistion' et 'Actuator' par héritage de ce composant 'ComponentIO', comme présenté en Figure 4.

```
class Sensor
  extends ComponentIO;
end

class DataAcquisition
  extends ComponentIO;
end

class Actuator
  extends ComponentIO;
end
```

Figure 4. Composants réparables pour la partie principale

Schémas de modélisation de politiques de maintenance

La partie précédente présentant le modèle AltaRica 3.0 n'intégrait pas la partie comportement d'un composant réparable, ainsi que la configuration liée à la politique de maintenance et au nombre de réparateurs.
Pour ce faire, nous commençons par définir en Figure 5 deux éléments AltaRica 3.0 nécessaires pour la suite.

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

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

Figure 5. Eléments AltaRica 3.0

La définition du domaine 'SDomain' permet de spécifier un type pour de futurs éléments (potentiellement variables, paramètres ou observateurs). Les quatre valeurs associées à ce type sont, elles aussi, définies (i.e. 'WORKING', 'FAILED', etc.). L'opérateur 'IsNotFailed' permet de définir une opération sur un paramètre 'aState', de type 'SDomain'. Cette opération renvoie l'entier 1 si ce paramètre 'aState' ne vaut pas 'FAILED' et 0 sinon. Cette opération permettra d'alléger les opérations arithmétiques par la suite.

1 Les différentes politiques de maintenance

Comme explicité dans la norme NF X 60-000, il y deux grandes catégories de maintenances :

- Les maintenances correctives dont l'objectif est de remettre le système dans un état lui permettant de fournir les qualités nécessaires à son utilisation (à noter qu'un mode dégradé peut être considéré). Ce type de

maintenances entraîne une indisponibilité de tout ou partie du système.

- Les maintenances préventives, dont l'objectif est de réduire la probabilité d'occurrence de défaillances du système. Elles sont effectuées selon des critères prédéterminés et doivent permettre d'éviter les défaillances du système durant son utilisation.

De plus, différents types de maintenances préventives existent. Les maintenances systématiques qui sont exécutée à des intervalles de temps préétablis, mais sans contrôle préalable de l'état du système. Les maintenances conditionnelles qui sont basées sur une surveillance du fonctionnement du système, en intégrant les actions correctives qui découlent des maintenances. Enfin les maintenances prévisionnelles qui sont exécutées en suivant les prévisions extrapolées de l'analyse et de l'évaluation de paramètres significatifs de la dégradation du système.

Il est tout à fait possible de modéliser chacune de ces politiques de maintenance en AltaRica 3.0. Pour la suite, nous allons nous intéresser et des maintenances correctives au niveau des composants, nous ajouterons néanmoins des conditions de mise en place d'une maintenance lorsque le système arrive dans un état trop critique, par exemple un nombre de composants en panne trop important. Nous ajouterons de plus une contrainte sur le nombre de réparateurs : limité à deux individus.

Le composant réparable, nommé '`Component`' sera décrit suivant le graphe d'états représenté Figure 6. Les nœuds du graphe représentent les différents états que pourra prendre le composant. Ces états sont bien sûr issus du domaine '`SDomain`' déclaré en Figure 5. Les arrêtes représentent les transitions entre états. Il s'agit d'un comportement générique qui sera plus ou moins adapté en fonction du schéma de modélisation présenté.

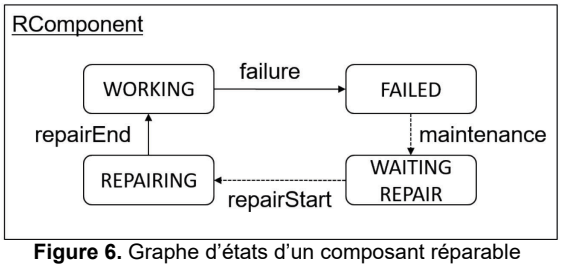

Figure 6. Graphe d'états d'un composant réparable

Par ailleurs en Figure 7, nous rajoutons, au sein de la partie principale du modèle, deux transitions qui représentent la politique de maintenance.

Les deux événements respectifs '`maintenanceReq`' et '`maintenance`' sont définis. Ils ont des délais donnés par une loi '`Dirac(0.0)`', signifiant que dès que la transition labélisée devient tirable, elle est instantanément tirée. Nous verrons après la signification de l'attribut '`hidden`'. La transition labélisée par l'événement '`maintenanceRequired`' spécifie la politique de maintenance au travers de sa garde. Cette transition est masquée (attribut '`hidden`' de son événement labélisant ayant la valeur '`true`'), ce qui signifie qu'elle ne pourra jamais être tirée toute seule, elle devra être synchronisée. De plus, aucune action n'est associée (instruction '`skip`'). Enfin la politique est complètement définie dans la Table 2. Une action de maintenance sera lancée sur le système si, par exemple, chacune des lignes d'actionneurs à au moins un actionneur en panne ; ou alors si au moins deux capteurs sont en panne.

La transition labélisée par l'événement '`maintenance`' synchronise obligatoirement (symbole '`!`') la transition '`maintenanceReq`' définie précédemment, et synchronise optionnellement (symbole '`?`') toute les transitions des composants labélisées par leur propre événement '`maintenance`'. Cela signifie d'une part que pour pouvoir tirer cette transition, il faut que la garde de la transition '`maintenanceReq`' soit à vrai (suivant la politique de maintenance définie précédemment). D'autre part, chaque composant ayant sa garde aussi à vrai pourra aussi tirer sa transition '`maintenance`'. Il passera alors de l'état '`FAILED`' à l'état '`WAITING_REPAIR`'.

```
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
```

Figure 7. Partie principale comportant la politique de maintenance

Composant	Nombre minimum en marche
Capteur (`Sensor`)	1
Unités d'acquisition (`DataAcquisition`)	1
Unité logique de décision (`LogicSolver`)	1
Actionneur (`Actuator`)	Au moins 1 ligne avec 1 actionneur

Table 2. Politique de maintenance

Le composant qui vient de passer dans l'état '`WAITING_REPAIR`' devra ensuite attendre qu'un réparateur soit disponible pour le réparer. C'est ce que nous allons modéliser de trois manières différentes dans ce qui suit. La première manière consistera à réaliser les réparations par synchronisation des variables de flux. La deuxième consistera à réaliser les réparations par synchronisation des événements. Enfin la dernière utilisera une notion avancée du langage : l'agrégation virtuelle.

2 Réparations par synchronisation des variables de flux

Pour synchroniser par variables de flux, le composant générique réparable '`Component`' est défini en Figure 8.

Ce composant est initialisé à l'état '`WORKING`' : la variable d'état '`vs`' a son attribut '`init`' (spécifiant que c'est une variable d'état) à la valeur '`WORKING`'. Les quatre événements labélisant les transitions, telles qu'indiquées en Figure 6, sont définies. L'événement maintenance a son attribut '`hidden`' à la valeur '`true`' car il sera synchronisé dans la transition globale de maintenance, spécifiant la politique de maintenance.

Deux variables de flux *ad-hoc* ont été définies, '`rUsed`' et '`rAvailable`', afin de réaliser les synchronisations par variables de flux. La variable '`rUsed`' fournira l'information

si un réparateur s'occupe du composant et sa valeur sera donc définie dans l'assertion. La variable 'rAvailable' récupérera l'information de disponibilité d'un réparateur et permettra de tirer la transition labélisée par l'événement 'repairStart', i.e. de lancer la réparation du composant.

```
class Component
  SDomain vs (init = WORKING);
  Integer rUsed (reset = 0);
  Boolean rAvailable (reset = false);
  parameter Real lambda = 1.0e-5;
  parameter Real mu = 1.0e-2;
  event failure (delay = exponential(lambda));
  event maintenance (delay = Dirac(0.0),
                     hidden = true);
  event repairStart (delay = Dirac(0.0));
  event repairEnd (delay = exponential(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
```

Figure 8. Composant réparable pour synchronisation par variables de flux

La Figure 9 suivante montre les ajouts nécessaires à la partie principale du modèle afin de réaliser cette synchronisation par variables de flux.

```
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
```

Figure 9. Partie principale comportant la synchronisation par variables de flux

Le paramètre 'repairer' spécifie le nombre de réparateurs du système considéré. La variable de flux 'rUsed' permet d'obtenir le nombre de réparateurs qui sont utilisés. Elle est mise à jour dans l'assertion en additionnant toutes les valeurs des variables 'rUsed' des composants. Enfin la variable de flux 'rAvailable' indique si un réparateur est disponible. Elle est mise à jour par rapport au nombre total 'repairer' de réparateurs et au nombre 'rUsed' de réparateurs utilisés. Cette variable 'rAvailable' est ensuite utilisée pour mettre à jour toute les variables 'rAvailable' associées aux composants.
La Figure 10 représente le graphe d'états d'un composant synchronisé, par variables de flux, avec l'équipe de réparateurs.

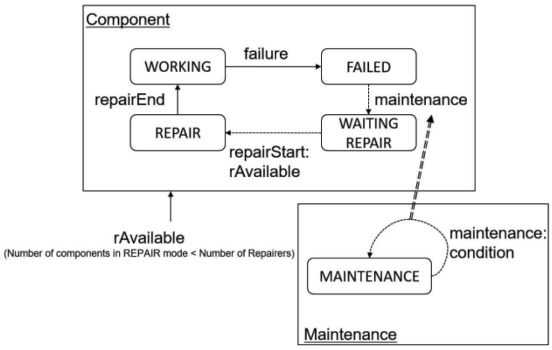

Figure 10. Graphe d'états de la synchronisation par variables de flux

3 Réparations par synchronisation des événements

Pour synchroniser par événements, le composant générique réparable 'Component' est défini en Figure 11.

```
class Component
  SDomain vs (init = WORKING);
  parameter Real lambda = 1.0e-5;
  parameter Real mu = 1.0e-2;
  event failure (delay = exponential(lambda));
  event maintenance (delay = Dirac(0.0),
                     hidden = true);
  event repairStart (delay = Dirac(0.0) ,
                     hidden = true);
  event repairEnd (delay = exponential(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
```

Figure 11. Composant réparable pour synchronisation par événements

Cette classe représente le graphe d'état d'un composant réparable de la Figure 6. Les événements 'maintenance', 'repairStart' et 'repairEnd' ont leur attribut 'hidden' à la valeur 'true' car ils seront synchronisés dans la partie principale, spécifiant la politique de maintenance.

La Figure 12 suivante montre les ajouts nécessaires à la partie principale du modèle afin de réaliser cette synchronisation par événements.

```
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
```

Figure 12. Partie principale comportant la synchronisation par événements

Nous ajoutons un block 'Repairer' qui va définir le comportement de l'équipe des deux réparateurs via des événements. Nous définissons ensuite, pour chaque composant du système deux nouveaux événements 'repairStart' et 'repairEnd' qui sont utilisés, dans la partie transition, pour synchroniser les événements 'repairStart' et 'repairEnd' propres à chaque composant et les événements 'repairStart' et 'repairEnd' du block 'Repairer'.
La Figure 13 représente le graphe d'états d'un composant synchronisé, par événements, avec l'équipe de réparateurs.

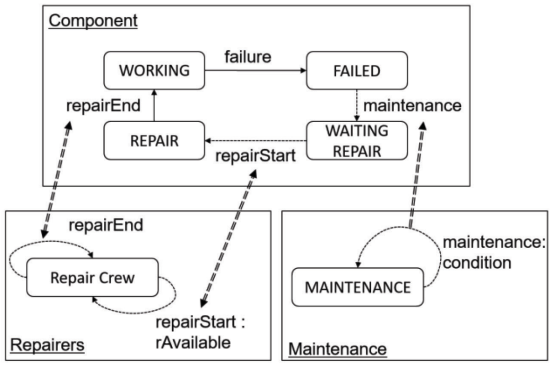

Figure 13. Graphe d'états de la synchronisation par événements

4 Réparations par agrégation virtuelle

Pour l'agrégation virtuelle, nous commençons par définir classe représentant le composant générique réparable 'Component' défini en Figure 14.

Cette classe est comme la classe définie en Figure 11 pour celle avec synchronisation par événements. En complément, elle agrège virtuellement un élément de type 'T' et qui sera utilisé, dans cette classe, avec l'alias 't'. Ce sera à l'instanciation d'une telle classe 'Component' qu'il faudra résoudre cette agrégation virtuelle en indiquant une instance d'une classe ou un block, compatible avec l'utilisation que nous en faisons dans celle-ci. L'utilisation de cette agrégation virtuelle se fait dans les deux transitions 'repairStart' et 'repairEnd' en synchronisant avec deux événements de 't'.

```
class Component
  embeds virtual T as t;
  SDomain vs (init = WORKING);
  parameter Real lambda = 1.0e-5;
  parameter Real mu = 1.0e-2;
  event failure (delay = exponential(lambda));
  event maintenance (delay = Dirac(0.0),
                     hidden = true);
  event repairStart (delay = Dirac(0.0));
  event repairEnd (delay = exponential(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
```

Figure 14. Composant réparable pour agrégation virtuelle

La Figure 15 suivante montre les ajouts nécessaires à la partie principale du modèle afin de réaliser cette synchronisation par agrégation virtuelle.

```
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 (lambda = 1.0e-5,
                     virtual T = main.Repairer);

  block Control
    DataAcquisition DA1, DA2, DA3
                          (lambda = 1.0e-6,
                     virtual T = main.Repairer);
    block LogicSolver
      extends Component (lambda = 1.0e-8,
                     virtual T = main.Repairer);
      ...
    end
    ...
  end

  block Actuator
    block Line1
      Actuator A1, A2 (lambda = 1.0e-6,
                     virtual T = main.Repairer);
      ...
    end
```

```
    ...
  end

  ...

  // Definition of the behavior

  ...
end
```

Figure 15. Partie principale avec l'agrégation virtuelle

Nous commençons par définir un block 'Repairer'. Ce block est utilisé à l'instanciation des classes héritant de la classe 'Component' pour résoudre le lien d'agrégation virtuelle. Ce block 'Repairer' est assez basique. Il permet de définir quand démarrer ou arrêter une réparation par un réparateur si un au moins est disponible.
Suite à cette création du block 'Repairer', pour chacun des composants nous résolvons l'agrégation virtuelle, au sein de la liste d'attributs, en indiquant que le type 'T' est égale au block 'Repairer' (virtual T = main.Repairer). Le mot clé 'main' indique que nous considérons l'élément principal dans la hiérarchie, c'est-à-dire le block Système ; donc 'main.Repairer' signifie l'élément fils 'Repairer' du block parent principal 'System'.
La Figure 16 représente le graphe d'états d'un composant partageant virtuellement l'équipe de réparateurs.

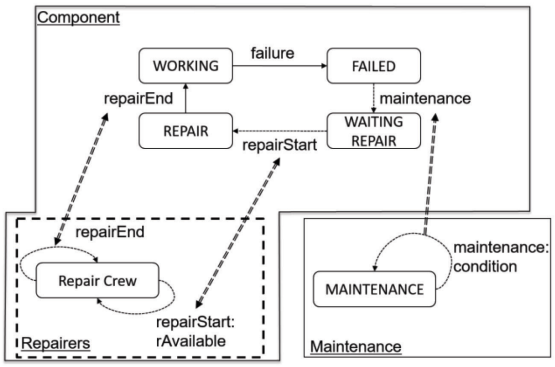

Figure 16. Graphe d'états partageant virtuellement l'équipe de réparateurs

Expérimentations

Nous venons de proposer une modélisation générique pour une politique de maintenance corrective au niveau des composants, c'est-à-dire que nous considérons que le système n'est en panne que lorsqu'un nombre critique de composants est en panne, lié en plus avec le type des composants. Nous avons de plus ajouté une contrainte sur le nombre de réparateurs : limité à deux individus. Nous avons réalisé trois modélisations différentes pour cette contrainte de réparateurs limités. La Table 3 suivante indique différentes caractéristiques quantitatives selon les trois modélisations Mf pour la synchronisation par variables de flux, Me pour la synchronisation par événements et Mv pour l'agrégation virtuelle. Les deux premières caractéristiques sont établies par rapport aux modèles tels que conçus ; alors que les autres sont établies par rapport aux modèles compilés, c'est-à-dire pour ce qui nous intéresse que toutes les instances des classes et toutes les synchronisations sont créées.

Caractéristiques	Mf	Me	Mv
Nombre de lignes	143	177	130
Nombre de lignes du block principal	84	124	66
Nombre de variables d'états	11	12	12
Nombre de variables de flux	48	24	24
Nombre d'événements	34	34	34
Nombre de ligne de l'assertion	48	24	24
Nombre de ligne	224	265	265

Table 3. Caractéristiques quantitatives selon les trois modélisations

Nous pouvons remarquer que entre les modélisations Mf et Me, l'écart se situe sur le nombre de variables de flux et leurs mises à jour dans l'assertion, beaucoup plus important dans Mf. Or ce résultat masque le fait que pour Me nous ayons dû déclarer deux événements supplémentaires par composants dans le modèle tel que conçus, ce qui n'est pas indiqué dans le tableau mais que l'on peut retrouver dans le nombre de lignes du modèle.
Par ailleurs, les modèles Me et Mv considérés compilés semblent identiques. Cela est normale car l'agrégation virtuelle reprend le schéma de synchronisation par événements mais l'intègre génériquement et directement à l'intérieur des composants. La différence notoire se situe donc dans la taille des deux modèles tels que conçus.
Enfin il est préférable d'utiliser une synchronisation par événements avec agrégation virtuelle car d'une part elle est moins source d'erreurs potentielles de modélisation. D'autre part, l'ajout de variable de flux à un coût à l'exécution car il est ensuite nécessaire de les mettre à jour au sein de l'assertion. Conceptuellement il y a une sorte de doublon entre la mise à jour de beaucoup de variables de flux, une par composants, et les transitions à tirer.

Nous avons aussi réalisé une expérimentation afin d'évaluer l'observateur Booléen 'TE' qui observe lorsque les deux variables 'out', des deux actionneurs 'A2' des deux lignes d'actionneurs, sont toutes les deux à la valeur 'false' Cela signifie qu'une certaine combinaison des composants en panne est valide. Nous l'avons réalisé pour les deux modèles Mf et Mv. Nous ne l'avons pas réalisé pour le modèle Me car compilé, il correspond au modèle Mv.
Nous avons utilisé le simulateur stochastique de la plateforme OpenAltaRica (Aupetit et Al., 2015). Nous avons d'abord comparé les moyennes des nombres de transitions tirées sur les histoires pour des simulation de 10^5, 10^6 et 10^7 histoires avec une durée de 20 ans (175200 unité de temps) pour chacune. La Table 4 ci-dessous repend les résultats.

		10^5	10^6	10^7
Transitions tirées (moyenne)	Mv	22.9015	22.9089	22.9002
	Mf	22.9133	22.8936	22.8997

Table 4. Moyennes des transitions tirées par simulation stochastique

Nous n'avons pas considéré les temps d'exécution nécessaire pour simuler ces histoires car d'une part pour 10^7 histoires, c'est de l'ordre de 1 à 2 minutes sur une machine portable de bureau ; d'autre part nous ne cherchons pas à réaliser des évaluations de performance. Le lecteur intéressé pourra se référer à Aupetit et Al., 2016 ou Aupetit et Al., 2017.

Le simulateur stochastique nous a de plus fourni des statistiques sur cet observateur 'TE'. Sur un temps de mission de 20 ans, nous avons réalisé des statistiques sur la valeur de cet observateur à la valeur vrai suivant différents indicateurs et à différents instants de mission. La Table 5 ci-dessous repend les résultats pour 10^7 histoires. Les résultats obtenus sont proches entre les deux modèles.

		Mf	Mv
5 years	had-value	0.315536	0.315438
	number-of-occurrences	0.343454	0.343362
10 years	had-value	0.646376	0.646189
	number-of-occurrences	0.849723	0.849419
15 years	had-value	0.829989	0.829901
	number-of-occurrences	1.35967	1.3594
20 years	had-value	0.920057	0.920068
	number-of-occurrences	1.86752	1.86752

Table 5. Evaluation des modèles par simulation stochastique

Encore une fois, l'objectif n'était pas, dans cette communication, de produire des résultats quantitatifs sur ces différents modèles. Des travaux en cours s'intéressent justement à l'évaluation de ces schémas de modélisation et aux politiques de maintenance. Ils seront publiés dans de prochaines communications.

Conclusion

Au travers de cette communication, nous avons présenté différents schémas de modélisation AltaRica 3.0 permettant de représenter une politique de maintenance corrective au niveau des composants d'un système, en incluant une contrainte supplémentaire donnée par un nombre limité de réparateurs pouvant intervenir sur le système.

Nous avons d'abord proposé une partie générique du modèle dans le sens où elle est commune aux trois schémas de modélisation étudiés. Cette partie générique fut présentée avec une approche 'top-down' et seul le comportement générique des composants ne fut pas défini. Nous avons aussi spécifié la politique de maintenance dans cette partie générique.

Concernant le nombre limité de réparateurs, c'est-à-dire la mise à disposition des réparateurs aux composants en fonction de leurs disponibilités, nous avons proposé trois schémas de modélisation différents. Chacun de ces schémas permet d'utiliser des constructions différentes du langage AltaRica 3.0. En réalisant une synchronisation par variables de flux, nous avons établi un modèle relativement simple en taille, néanmoins sujet à erreurs ainsi que potentiellement conséquent à l'exécution. En réalisant une synchronisation par événements, nous avons établi un modèle plus conséquent en taille, mais conceptuellement plus simple à l'interprétation et potentiellement à l'exécution. Enfin en réalisant une agrégation virtuelle, nous avons établi un modèle simple d'un point de vue taille, mais aussi moins propice aux erreurs.

Ces schémas de modélisation de politiques de maintenance en AltaRica 3.0 ouvrent la voie vers de nouveaux horizons. D'une part la définition de schémas de modélisation permettant de décrire simplement et efficacement des caractéristiques classiques, par exemples redondances froides, composants testés périodiquement, ressources partagées, défaillances de causes communes, etc. Il est en effet possible de définir des outils permettant de mettre en œuvre ces schémas simplement en paramétrant par des composants de bases définis en bibliothèques. D'autre part l'utilisation du langage AltaRica 3.0 pour des problématiques de performance sous incertitudes, typiquement les politiques de maintenance.

Références

Aupetit, B., M. Batteux, A. Rauzy, and J.-M. Roussel, 2015, Improving performance of the AltaRica 3.0 stochastic simulator, Proceedings of Safety and Reliability of Complex Engineered Systems: ESREL 2015, pp. 1815–1824. CRC Press.

Aupetit, B., M. Batteux, A. Rauzy, and J.-M. Roussel, 2016, Vers la définition d'un kit d'évaluation pour les simulateurs stochastiques. 20ième Congrès de Maîtrise des Risques et Sûreté de Fonctionnement (LambdaMu 20), Oct 2016, Saint_Malo, France. 2016.

Batteux, M., T. Prosvirnova, and A. Rauzy, 2017, Altarica 3.0 assertions: the why and the wherefore. Journal of Risk and Reliability.

Batteux, M., T. Prosvirnova, and A. Rauzy, 2015, System Structure Modeling Language (S2ML). AltaRica Association, 2015. archive hal-01234903, version 1.

Brameret, P.-A., A. Rauzy, J.-M. Roussel, 2015, Automated generation of partial markov chain from high level descriptions. Reliability Engineering and System Safety, vol. 139, pp. 179–187.

Cassandras, C. G, S. Lafortune, 2008, Introduction to Discrete Event Systems. New-York, NY, USA: Springer.

Prosvirnova, T., M. Batteux, P.-A. Brameret, A. Cherfi, T. Friedlhuber, J.-M. Roussel, A. Rauzy, 2013, The altarica 3.0 project for model-based safety ssessment. In Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS'2013, York, Great Britain, pp. 127–132.

Prosvirnova, T., A. Rauzy, 2015, Automated generation of minimal cutsets from altarica 3.0 models. International Journal of Critical Computer-Based Systems 6(1), 50–79.

Rauzy, A, 2008, Guarded transition systems: a new states/events formalism for reliability studies, Journal of Risk and Reliability, vol. 222(4), pp. 495–505.

Zimmermann, A, 1976, Stochastic Discrete Event Systems. Berlin, Heidelberg, Germany: Springer