Institut pour la Maîtrise des Risques
Sûreté de Fonctionnement - Management - Cindyniques

Modélisation de combinaisons de maintenances en AltaRica 3.0

Modeling combination of maintenance policies with AltaRica 3.0

Michel BATTEUX
IRT SystemX
Palaiseau, France
michel.batteux@irt-systemx.fr

Tatiana PROSVIRNOVA
ONERA/DTIS
Université de Toulouse
Toulouse, France
tatiana.prosvirnova@onera.fr

Antoine RAUZY
NTNU-MTP
Trondheim, Norway
antoine.rauzy@ntnu.no

***Résumé*—Le maintien en condition opérationnelle de systèmes industriels est un facteur de compétitivité important ; et une bonne stratégie de maintenance du système assure une disponibilité élevée des équipements, tout en minimisant les coûts liés aux interventions. La modélisation et la simulation de tels systèmes, incluant leurs stratégies associées de maintenances, permet d'en étudier les indicateurs de performance sous-jacents (disponibilité, coûts, etc.). Dans cette communication, nous présentons un schéma de modélisation AltaRica 3.0 de combinaison de différentes politiques de maintenance de différents composants d'un système.**

Mots-clés*—*MCO, Maintenance, AltaRica 3.0, schéma de modélisation

***Summary*—operational condition strategies of complex systems is one of the major actual challenges. An appropriated maintenance strategy ensures a high availability of the system, while minimizing costs due to interventions. Modeling and simulation of such systems, with their strategies of maintenances, enables the study of their performance indicators, such as availability, costs, etc. This publication presents an AltaRica 3.0 modeling pattern for the combination of several maintenance policies.**

I. Introduction

Le maintien en condition opérationnelle (MCO) de systèmes industriels est un facteur de compétitivité important. Une bonne stratégie de maintenance du système assure en effet une disponibilité élevée des équipements, tout en minimisant les coûts liés aux interventions. Différentes stratégies de maintenances existent et sont classiquement catégorisées en deux grands groupes suivant la norme [1]. Les maintenances correctives, qui se déclinent en maintenances palliatives et maintenances curatives, qui consistent à intervenir uniquement après l'occurrence de la panne. Les maintenances préventives, qui se déclinent en maintenances systématiques, maintenances conditionnelles et maintenance prévisionnelles, qui consistent à intervenir avant l'occurrence de la panne. Dans ce cadre l'utilisation d'une bonne stratégie de maintenance peut combiner plusieurs de ces différentes maintenances suivant les types des composants du système considéré, et de la politique d'exploitation du système (disponibilité, rentabilité, criticité, etc.).

La modélisation et la simulation de tels systèmes, incluant leurs stratégies associées de maintenances, permet d'en étudier des indicateurs de performance sous-jacents, par exemple la disponibilité, le coût, etc. Dans cette communication, nous utilisons le langage AltaRica 3.0 pour modéliser différentes politiques de maintenance de différents composants d'un système. L'objectif étant de montrer comment, grâce à un langage de modélisation haut niveau, il est simple et efficace de modéliser des combinaisons de politiques de maintenances. L'approche utilisée de modélisation sera ainsi considérée comme un schéma de modélisation. Par ailleurs, 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 de ces indicateurs pourra se référer à [2] ou [3] pour la simulation stochastique de modèles AltaRica 3.0, ou à [8] et [9] 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 la modélisation en AltaRica 3.0 de l'exemple applicatif. La cinquième partie présente des études et expérimentations sur ce modèle. Enfin la dernière partie conclura cette communication.

II. EXEMPLE APPLICATIF

Nous considérons un système tel que représenté en Figure 1. Le système peut être envisager comme un système de production, par exemple de matières, contenant trois principales activités : l'extraction, la séparation et le conditionnement, à partir d'une source et allant jusque une cible (ces deux derniers éléments étant hors scope du système considéré). Plusieurs composants, répartis dans trois parties représentant ces activités, constituent ce système. Enfin un organe de supervision, qui n'est pas représenté sur la Figure 1, sera aussi pris en compte dans la modélisation.

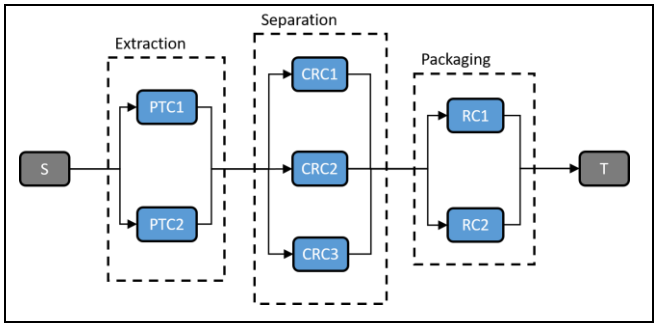

Figure 1 : schéma du système applicatif

La première partie d'extraction contient deux composants 'PTC1' et 'PTC2' qui fonctionnent en parallèles pour réaliser l'extraction. Lorsque l'un de ces deux composants passe dans un état dégradé, ou tombe en panne, l'autre composant passe alors dans un état de 'surcharge' afin de compenser la perte de production due au composant dégradé ou en panne. Concernant la maintenance, ces deux composants sont testés périodiquement. Cela signifie que l'organe de supervision effectue un test régulier de ces composants, tous les 9 mois pour notre exemple. Cela permet de représenter le cas où ces deux composants ne sont pas directement accessibles, par exemple dans un milieu hostile ou à haut risque. Lors des tests périodiques, les composants sont arrêtés durant une certaine période de temps, 6h pour notre exemple, et aucune extraction n'est réalisée durant cette période de temps. Si un composant est dégradé ou en panne, alors le test permet d'obtenir cette information, et une maintenance s'ensuit après un certain délai. Nous considérons que cette partie d'extraction est opérationnelle lorsqu'au moins un composant est en fonctionnement, même s'il est dans un état dégradé.

La deuxième partie du système concerne la séparation. Elle contient trois composants identiques en parallèles, dont l'un est en redondance froide des deux autres. Lorsqu'un composant est en marche, il peut tomber en panne. Lorsqu'il est en attente de mise en marche, soit il peut tomber en panne avant la sollicitation : c'est-à-dire une panne dormante ; soit lorsqu'il est sollicité pour démarrer, il démarre normalement ou il ne démarre pas : c'est-à-dire une panne à la sollicitation. Concernant la maintenance, chaque composant est réparé après un certain délai lorsqu'il est en panne, sauf la panne dormante qui n'est pas visible tant que le composant n'est pas sollicité. Nous considérons que cette partie de séparation est opérationnelle lorsqu'au moins un composant est en fonctionnement.

Enfin la troisième partie du système est la partie de conditionnement. Elle est constituée de deux composants réparables qui fonctionnent en parallèle. Concernant la maintenance, chaque composant est maintenu préventivement à intervalle régulier, tous les ans pour notre, et en alternance l'un de l'autre. Dans le cas où un composant tombe en panne il est réparé après un certain délai. Nous considérons que cette partie de conditionnement est opérationnelle lorsqu'au moins un composant est en fonctionnement.

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 les politiques de maintenance spécifiées. 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 possibilités du langage AltaRica 3.0 pour modéliser ce système et ses politiques de maintenance.

III. 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 [4]. 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 du cadre mathématique GTS, pour Guarded Transitions System ([10] et [6]), et du paradigme de structuration S2ML, pour System Structure Modeling Language ([5]).

Le cadre mathématique GTS permet de rendre compte des aspects comportementaux que l'on souhaite modéliser, c'est-à-dire la sémantique d'exécution. Cette exécution 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 ([7] et [11]). Ces délais peuvent être déterministes ou stochastiques ; et AltaRica 3.0 fournit les délais usuels (du type exponentiel, Weibull, etc.) et des délais empiriques.

Le paradigme de structuration S2ML permet de construire les modèles de différentes manières. S2ML unifie les deux paradigmes de structuration dominants des langages de modélisation : l'orienté objet à classe et l'orienté objet à prototype. Il permet de modéliser suivant deux approches combinables. L'approche dite 'top-down', avec une vision au niveau du système, qui permet la réutilisation de schémas de modélisation, et donc utilisant le paradigme orienté objet à prototype. L'approche dite 'bottom-up', avec une vision au niveau des composants, qui permet la réutilisation de composants définis en bibliothèques, et donc utilisant le paradigme orienté objet à classe. Les deux principales constructions structurelles pouvant être déclarées en AltaRica 3.0, et découlant de ces deux approches, sont les '`block`' et les '`class`'. 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. Les classes correspondent aux éléments des formalismes orientés objet à classe. Un élément '`block`', que nous nommerons bloc dans la suite, représente une construction structurelle ayant une unique occurrence. Un bloc est déclaré et directement utilisé. Les blocs correspondent aux éléments des formalismes orientés objet à prototype.

IV. MODÉLISATION DE L'EXEMPLE APPLICATIF

Conceptuellement le modèle AltaRica 3.0, du système représenté en Figure 1, suit un schéma de modélisation structuré en deux niveaux :

- Le niveau du processus physique (de production) du système, qui définit les différents composants du système. Ces composants sont organisés suivant les différentes parties de production. Ils intègrent, en propre, leurs comportements respectifs en termes de fonctionnements nominaux et anormaux, suites aux occurrences des défaillances. Le langage AltaRica 3.0 offre les mécanismes nécessaires pour définir efficacement des comportements génériques, qui sont ensuite hérités et spécialisés.
- Le niveau de supervision du système, lui-même structuré en plusieurs parties suivant les différentes parties du processus physique, qui permet de modéliser les modules nécessaires dédiés aux politiques de maintenance : modules de tests, qui permettent de détecter l'occurrence d'une défaillance, modules de mises en place des maintenances correctives ou préventives.

Pour construire le modèle AltaRica 3.0, nous allons utilisé une approche de conception dite 'top-down'. Nous commencerons par décrire la partie principale du système, le squelette, sans nous soucier de ce que peuvent précisément contenir ses éléments. Nous supposerons juste quelques caractéristiques sur ceux-ci. Nous construirons ensuite les différents composants et spécifierons les politiques de maintenances dans la supervision.

A. Partie principale du modèle AltaRica 3.0

La partie principale du modèle AltaRica 3.0 est donnée par la Figure 2. Elle représente le squelette de notre modèle. Elle déclare un bloc '`System`', qui est le bloc principal du modèle. Ce bloc principal est composé de deux sous-blocs '`Supervisor`' et '`Plant`', qui correspondent respectivement à la partie supervision du système et au processus physique. Le sous-bloc '`Supervisor`' est composé de trois sous-blocs correspondant aux supervisions respectives des trois parties du processus physique. Le sous-bloc '`Plant`' est composé, à son tour, de cinq sous-blocs constituant ce qui est représenté en Figure 1. L'assertion de ce sous-bloc, donné par le mot-clé '`assertion`' et composée d'instructions de mises à jour de variables de flux, permet de connecter les différentes parties du système. Enfin un observateur Booléen est déclaré : '`oSystemProduction`'. Il correspond à l'observation de la production du système, et il est défini par rapport à la variable d'entrée du sous-bloc cible '`Target`'. Cet observateur permet, par exemple, d'étudier la disponibilité du système.

B. Partie du processus physique du modèle AltaRica 3.0

Pour les descriptions des composants de production, nous supposerons qu'ils ne maîtrisent que leurs comportements internes propres : généralement que des pannes. Tous les comportements de supervision (maintenance, tests, contrôle-commande) seront réalisés par la partie supervision, et cette partie pourra modifier directement les états des composants. Cette manière de modéliser est doublement justifiée. D'une part elle allège le modèle d'éléments redondants : généralement des variables et des transitions permettant de synchroniser les composants avec la supervision. D'autre part elle représente plus finement la réalité car c'est bien un opérateur qui, lors d'une maintenance, modifie physiquement l'état du composant.

```
block System
  block Supervisor
    block Extraction
      // ...
    end
    block Separation
      // ...
    end
    block Packaging
      // ...
    end
  end

  block Plant
    block Source
      Boolean out (reset=false);
      assertion
        out := true;
    end
    block Extraction
      // ...
      Boolean in, out (reset=false);
    end
    block Separation
      // ...
      Boolean in, out (reset=false);
    end
    block Packaging
      // ...
      Boolean in, out (reset=false);
    end
    block Target
      Boolean in (reset=false);
    end
    assertion
      Extraction.in := Source.out;
      Separation.in := Extraction.out;
      Packaging.in := Separation.out;
      Target.in := Packaging.out;
  end
  observer Boolean oSystemProduction
                    = Plant.Target.in;
end
```

Figure 2 : partie principale du modèle AltaRica 3.0

1) Partie extraction

Pour la partie extraction du système, nous définissons d'abord, en Figure 3, les éléments nécessaires dans le bloc '`Extraction`', du sous-bloc '`Plant`' du bloc principal '`System`'. Ce bloc est constitué de deux instances '`EC1`' et '`EC2`' d'une classe nommée '`ExtractionComponent`', que nous définirons après. Nous supposons que cette classe contient deux variables de flux '`in`' et '`out`', qui sont reliées aux variables de flux '`in`' et '`out`' du bloc '`Extraction`', qui ont déjà été déclarées dans la présentation du bloc principale '`System`'. Par exemple l'instruction '`EC2.in := in;`' de l'assertion exprime le fait que la variable de flux '`in`' du composant '`EC2`', noté donc ici '`EC2.in`', est mise à jour par la valeur de la variable de flux '`in`' du bloc '`Extraction`'.

```
block Extraction
  ExtractionComponent EC1, EC2;
  Boolean in, out (reset=false);
  assertion
    EC1.in := in; EC2.in := in;
    out := in and (EC1.out or EC2.out);
end
```

Figure 3 : bloc extraction du modèle AltaRica 3.0

La définition de la classe '`ExtractionComponent`', pour laquelle les deux instances '`EC1`' et '`EC2`' ont été déclarées, se réalise à l'extérieur de la partie principale du modèle (i.e. à du

bloc principal ‘System’). Pour rappel, nous souhaitons modéliser d’une part que chaque composant peut compenser, en partie, la dégradation ou la panne de l’autre composant, et qu’ils sont de plus testés périodiquement. La Figure 4 présente la définition de deux domaines et de la classe permettant de décrire des composants testés périodiquement. Nous reprenons, en partie, ce qui est donné dans [4] pour modéliser de tels composants testés périodiquement. Les deux domaines permettent de représenter les états de ces composants testés périodiquement : ‘PTCState’ pour définir les états de production interne du composant, et ‘PTCMode’ pour définir les modes opérationnels. La classe ‘PeriodicallyTestedComponent’ utilise deux variables d’état ‘_State’ et ‘_Mode’ de types respectifs ceux décris par les deux domaines. Différents événements, et transitions associées, sont définis pour spécifier les comportements de défaillances ou de surcharge du composant : ‘failure’, ‘degradation’ et ‘overload’. Ces trois événements utilisent des délais qui leurs sont propres : ‘exponential’ ou ‘Dirac’. Chaque transition contient trois éléments : un label, une garde et une action. Le label est un événement préalablement défini. La garde est une formule Booléenne qui permet d’activer, ou désactiver, le tirage de la transition. L’action est composée d’instructions de mises à jour de variables d’état. Enfin une variable de flux ‘toOther’ envoie l’information de l’état de production du composant à l’extérieur via l’assertion.

La Figure 5 présente la définition de la classe ‘ExtractionComponent’, qui permet de décrire les composants d’extraction (c’est-à-dire ceux instanciés en Figure 3). Elle hérite de la classe ‘PeriodicallyTestedComponent’, présentée en Figure 4, et y ajoute les variables de flux permettant de décrire la production du composant, en fonction de son mode opérationnel et de son état de production.

```
domain PTCState {OVERLOADED, NOMINAL, DEGRADED,
                                    FAILED, STOP}
domain PTCMode {OPERATION, TEST}

class PeriodicallyTestedComponent
  PTCState _State (init=NOMINAL);
  PTCMode _Mode (init=OPERATION);
  PTCState fromSibling (reset=NOMINAL);
  parameter Real pF = 1.0e-5;
  parameter Real pD = 1.0e-3;
  event failure (delay=exponential(pF));
  event degradation (delay=exponential(pD));
  event overload (delay=Dirac(0.0));
  transition
    failure: _Mode == OPERATION and
    (_State == NOMINAL or _State == DEGRADED
     or _State == OVERLOADED)
    -> _State := FAILED;
    degradation: _Mode == OPERATION and
     (_State == NOMINAL or _State == OVERLOADED)
    -> _State := DEGRADED;
    overload: _Mode == OPERATION and
                _State == NOMINAL and
                (fromSibling == DEGRADED
                 or fromSibling == FAILED)
    -> _State := OVERLOADED;

  PTCState toOther (reset=NOMINAL);
  assertion
    toOther := _State;
end
```

Figure 4 : définition des composants testés périodiquement

```
class ExtractionComponent
  extends PeriodicallyTestedComponent;
  Boolean in, out (reset=false);
  assertion
    out := if _Mode == TEST or _State == FAILED
           or _State == STOP then false else in;
end
```

Figure 5 : classe des composants d'extraction

2) Partie séparation du modèle AltaRica 3.0

Pour la partie séparation, et comme pour la partie extraction présentée précédemment, nous définissons d’abord, en Figure 6, les éléments nécessaires dans le bloc ‘Separation’. Ce bloc est constitué de trois instances ‘SC1’, ‘SC2’ et ‘SC3’ d’une classe nommée ‘SeparationComponent’, que nous définirons après. Comme ces trois composants sont en parallèles, dont l’un en redondance froide des deux autres, nous spécifions que l’état initial de ‘SC3’ est en attente, via la directive ‘_State.init=STANDBY’. Ici aussi, nous supposons que cette classe contient deux variables de flux ‘in’ et ‘out’, qui sont reliées au variable de flux ‘in’ et ‘out’ du bloc ‘Separation’. La dernière instruction de l’assertion, qui met à jour la variable ‘out’, utilise l’opérateur de comptage ‘#’ qui permet de compter le nombre de valeurs à vrai dans les paramètres. Cela permet bien de représenter que la production se réalise si au moins deux composants produisent.

```
block Separation
  SeparationComponent SC1, SC2;
  SeparationComponent SC3 (vsState.init=STANDBY);
  Boolean in, out (reset=false);
  assertion
    SC1.in := in; SC2.in := in; SC3.in := in;
    out := #(SC1.out, SC2.out, SC3.out) >= 2;
end
```

Figure 6 : bloc séparation du modèle AltaRica 3.0

La définition de la classe ‘SeparationComponent’, se réalise, ici aussi, à l’extérieur du bloc principal ‘System’. Pour rappel, nous souhaitons modéliser que les trois composants sont en parallèles avec toujours l’un en redondance froide des deux autres, ce qui implique différents type de pannes, et que chaque composant est réparé lorsqu’il est en panne. La Figure 7 montre la définition de ces composants en redondance froide. Elle spécifie préalablement le domaine ‘CRCState’ pour définir les différents états possibles. Ensuite nous spécifions la classe ‘ColdRedundantComponent’. Elle utilise une variable d’état de type ‘CRCState’ et définie l’ensemble des événements, et transitions associée, permettant de spécifier les comportements de pannes et de démarrages par sollicitation. À noter la panne dormante qui se réalise lorsque le composant est en attente de mise en marche, et qui n’est donc pas visible tant que le composant n’est pas sollicité. Ici aussi, plus d’informations sur la modélisation de tels composants en redondance froide peuvent être trouvées dans [4]. Enfin une variable de flux ‘isFailed’ envoie l’information de l’état de panne du composant (pas la panne dormante).

```
domain CRCState {STANDBY, WORKING, FAILED,
                                    HIDDEN_FAILED}
class ColdRedundantComponent
  CRCState _State (init=WORKING);
  Boolean isDemanded (reset=false);
  parameter Real pF = 1.0e-5;
  parameter Real pFoD = 1.0e-6;
  parameter Real pDF = 1.0e-6;
  event failure (delay=exponential(pF));
  event start (delay=Dirac(0.0),
                      expectation=1 - pFoD);
  event failureOnDemand (delay=Dirac(0.0),
                         expectation=pFoD);
  event dormantFailure
                    (delay=exponential(pDF));
  transition
    failure: _State == WORKING
              -> _State := FAILED;
    start: _State == STANDBY and isDemanded
              -> _State := WORKING;
    failureOnDemand: (_State == STANDBY or
       _State == HIDDEN_FAILED) and isDemanded
            -> _State := FAILED;
    dormantFailure: _State == STANDBY
              -> _State := HIDDEN_FAILED;

  Boolean isFailed (reset=false);
  assertion
    vfIsFailed := vsState == FAILED;
end
```

Figure 7 : définition des composants en redondance froide

La Figure 8 présente la définition de la classe ‘SeparationComponent’, qui permet de décrire les composants de séparation, (ceux instanciés en Figure 6). Elle hérite de la classe ‘ColdRedundantComponent’, présentée en Figure 7, et y ajoute les variables de flux permettant de décrire la production du composant en fonction de son état.

```
class SeparationComponent
  extends ColdRedundantComponent;
  Boolean in, out (reset=false);
  assertion
    out := _State == WORKING and in;
end
```

Figure 8 : classe des composants de séparation

3) Partie conditionnement du modèle AltaRica 3.0

Pour la partie conditionnement, et comme pour les parties précédentes extraction et séparation, nous définissons d’abord, en Figure 9, les éléments nécessaires dans le bloc ‘Packaging’. Ce bloc est constitué de deux instances ‘PC1’ et ‘PC2’ d’une classe nommée ‘PackagingComponent’, que nous définirons après. Ici encore, nous supposons que cette classe contient deux variables de flux ‘in’ et ‘out’, qui sont reliées au variable de flux ‘in’ et ‘out’ du bloc.

```
block Packaging
  PackagingComponent PC1, PC2;
  Boolean in, out (reset=false);
  assertion
    PC1.in := in; PC2.in := in;
    out := PC1.out or PC2.out;
end
```

Figure 9 : bloc conditionnement du modèle AltaRica 3.0

Pour rappel, pour la définition de la classe ‘PackagingComponent’, nous souhaitons modéliser des composants maintenus préventivement. Pour cela nous utilisons de simples composants réparables. La Figure 10 montre la définition de tels composants. Elle spécifie préalablement le domaine ‘RCState’ pour définir les différents états possibles. Ensuite est spécifiée la classe ‘RepairableComponent’. Elle utilise une variable d’état de type ‘RCState’ et définie l’événement, et la transition associée, permettant de spécifier la panne. Ici aussi, plus d’informations sur la modélisation de tels composants réparables peuvent être trouvées dans [4]. Enfin une variable de flux ‘isFailed’ envoie l’information de l’état de panne du composant.

Enfin la Figure 10 présente la définition de la classe ‘PackagingComponent’, permettant de décrire les composants de conditionnement, c’est-à-dire les composants instanciés en Figure 9. Elle hérite de la classe ‘RepairableComponent’ et y ajoute les variables de flux permettant de décrire la production du composant en fonction de son état.

```
domain RCState {WORKING, FAILED, MAINTENANCE}

class RepairableComponent
  RCState _State (init=WORKING);
  parameter Real pF = 1.0e-5;
  event failure (delay=exponential(pF));
  transition
    failure: _State == WORKING
               -> _State := FAILED;

  Boolean isFailed (reset=false);
  assertion
    isFailed := _State == FAILED;
end

class PackagingComponent
  extends RepairableComponent;
  Boolean in, out (reset=false);
  assertion
    out := _State == WORKING and in;
end
```

Figure 10 : définition des composants réparables et de conditionnement

C. Partie supervision du modèle AltaRica 3.0

Concernant la supervision, et comme présenté en Figure 2, elle est décomposée en trois sous-blocs en correspondance aux trois sous-blocs de la production : extraction, séparation et conditionnement. Chacun de ces sous-blocs de supervision va donc spécifier les interactions de contrôle entre les composants, ainsi que les politiques de maintenances spécifiques par parties. Ainsi, et même si ces interactions de contrôle entre composants se réalisent physiquement du point de vue du système, c’est-à-dire à l’endroit où se trouvent les composants, nous les considérons pour le modèle dans la partie supervision ; il s’agit bien sûr d’un point de vue conceptuel.

1) Supervision de l’extraction

Pour la supervision de l’extraction, et comme le montre la Figure 11, il y a quatre parties. La première partie qui importe, via la directive ‘embeds’ les composants d’extraction ‘EC1’ et ‘EC2’ définis dans la partie du processus physique (voir Figure 3). Cela permettra de directement utiliser les éléments comportementaux liés au contrôle et à la maintenance de ces composants. La deuxième partie spécifie les interactions de contrôle directes entre les composants. Enfin, les deux dernières parties spécifient, respectivement, les tests périodiques, et les maintenances.

```
block Supervision
  block Extraction
    embeds main.Plant.Extraction.EC1 as EC1;
    embeds main.Plant.Extraction.EC2 as EC2;
    // Interactions between components
    assertion
      EC1.vcFromSibling := EC2.vcToOther;
      EC2.vcFromSibling := EC1.vcToOther;
    // State variables for tests and maintenances
    // ...
    // Periodic tests
    // ...
    // Maintenances
    // ...
  end
end
```

Figure 11 : bloc de supervision de l'extraction

Pour définir les tests périodiques et les maintenances, un nouveau domaine de valeurs est préalablement défini et donné ci-dessous :

```
domain ExtractionSMode {OPERATION, TEST,
                        MAINTENANCE}
```

De plus deux variables sont définies, afin de rendre compte l'état du superviseur de cette partie extraction, et d'autre part pour capturé l'état des composants. Leur définition est donnée ci-dessous :

```
// State variables for tests and maintenances
ExtractionSMode _Mode (init=OPERATION);
PTCState _CState (init=NOMINAL);
```

Les tests périodiques sont décrits en Figure 12. De manière simple, deux événements sont définis et spécifiés avec leurs transitions associées. Dans les actions de ces transitions, la supervision va directement modifier les variables états des composants pour les mettre en modes 'TEST' ou 'OPERATION'. De plus la transition liée à l'événement 'completeTest' met à jour, dans l'action, l'état observé des composants au travers de la variable '_CState'.

```
// Periodic tests
parameter Real pST = 6570.0;
parameter Real pCT = 6.0;
event startTest (delay=Dirac(pST));
event completeTest (delay=Dirac(pCT));
transition
  startTest: _Mode == OPERATION -> {_Mode := TEST;
          EC1._Mode := TEST; EC2._Mode := TEST;}
  completeTest: _Mode == TEST -> {
        _Mode := OPERATION;
        _CState := switch {
          case EC1.toOther == DEGRADED or
               EC2.toOther == DEGRADED: DEGRADED
          case EC1.toOther == FAILED or
               EC2.toOther == FAILED: FAILED
          default: NOMINAL};
        EC1._Mode := OPERATION;
        EC2._Mode := OPERATION;}
```

Figure 12 : tests périodiques pour l'extraction

Les maintenances sont décrites en Figure 13. Ici aussi, et de manière simple, deux événements sont définis et spécifiés avec leurs transitions associées. Notons que les délais de ces maintenances sont donnés par des lois uniformes, caractérisant des délais, plus ou moins maitrisés, des débuts et fins de maintenances. Comme pour les tests, les actions de ces transitions vont directement modifier les variables états des composants pour les mettre en états 'STOP ou 'NOMINAL'. D'autre part elles utilisent l'état observé des composants, au travers de la variable '_CState'.

```
// Maintenances
parameter Real pStartMin = 24.0;
parameter Real pStartMax = 48.0;
parameter Real pFinishMin = 12.0;
parameter Real pFinishMax = 24.0;
event startMaintenance
        (delay=uniform(pStartMin,pStartMax));
event finishMaintenance
        (delay=uniform(pFinishMin,pFinishMax));
transition
  startMaintenance: _Mode == OPERATION and
       (_CState == DEGRADED or _CState == FAILED)
                -> { _Mode := MAINTENANCE;
        EC1._State := STOP; EC2._State := STOP;}
  finishMaintenance: _Mode == MAINTENANCE -> {
    _Mode := OPERATION; _CState := NOMINAL;
    EC1._State := NOMINAL; EC2._State := NOMINAL;}
```

Figure 13 : maintenances pour l'extraction

Cette partie de supervision des tests et des maintenances se réalisent en coordination. C'est-à-dire que les tests des composants sont réalisés à des intervalles réguliers afin d'obtenir l'état observé de ceux-ci. Lorsque cet état observé lors du test est dégradé ou en panne, alors une maintenance est mise en place, et nous supposons que suite à la maintenance, les deux composants redeviennent en fonctionnement normal.

2) Supervision de la séparation

Pour la supervision de la séparation, et comme le montre la Figure 14, il y a trois parties. La première partie importe les composants de séparation, définis dans la partie du système (voir Figure 6), La deuxième partie spécifie les interactions directes de contrôle entre les composants. La troisième partie spécifie les maintenances.

```
block Supervision
  block Separation
    embeds main.Plant.Separation.SC1 as SC1;
    embeds main.Plant.Separation.SC2 as SC2;
    embeds main.Plant.Separation.SC3 as SC3;
    // Interactions between components
    assertion
      SC1.isDemanded := SC2.isFailed or
                                SC3.isFailed;
      SC2.isDemanded := SC1.isFailed or
                                SC3.isFailed;
      SC3.isDemanded := SC1.isFailed or
                                SC2.isFailed;
    // Maintenances
    // ...
  end
end
```

Figure 14 : bloc de supervision de la séparation

Pour définir les maintenances de cette partie séparation, un nouveau domaine de valeurs est, ici aussi, préalablement défini et donné ci-dessous :

```
domain SeparationSMode {OPERATION, MAINTENANCE}
```

Les maintenances sont décrites en Figure 15. Ici encore, et de manière simple, deux événements sont définis et spécifiés avec leurs transitions associées. Elles se réalisent en fonction de l'information des composants qui sont en panne, via la variable de flux 'areFailed' qui est mise à jour dans l'assertion. Leurs actions vont directement modifier les variables d'états des composants pour les mettre en états 'STANDBY', si le composant est en panne, ou 'WORKING', pour les remettre en marche. Il est à noter que ces remises en marche des composants se réalisent suivant une logique spécifiant que si deux composants sont déjà en marche, alors le troisième, qui vient justement d'être réparé, est laissé en attente.

```
// Maintenance part
SeparationSMode _Mode (init=OPERATION);
Boolean areFailed (reset=false);
parameter Real pStartMin = 2.0;
parameter Real pStartMax = 3.0;
parameter Real pFinishMin = 6.0;
parameter Real pFinishMax = 12.0;
event startMaintenance
          (delay = uniform(pStartMin,pStartMax));
event finishMaintenance
        (delay = uniform(pFinishMin,pFinishMax));
transition
  startMaintenance: areFailed and
              _Mode == OPERATION -> {
                   _Mode := MAINTENANCE;
                   if SC1._State == FAILED
                     then SC1._State := STANDBY;
                   if SC2._State == FAILED
                     then SC2._State := STANDBY;
                   if SC3._State == FAILED
                     then SC3._State := STANDBY;}
  finishMaintenance: _Mode == MAINTENANCE
                     -> {
                           _Mode := OPERATION;
                           switch {
                 case SC1._State == WORKING and
                      SC2._State == STANDBY and
                      SC3._State == STANDBY:
                          SC2._State := WORKING;
                 case SC1._State == STANDBY and
                      SC2._State == WORKING and
                      SC3._State == STANDBY:
                          SC1._State := WORKING;
                 case SC1._State == STANDBY and
                      SC2._State == STANDBY and
                      SC3._State == WORKING:
                          SC1._State := WORKING;
                 case SC1._State == STANDBY and
                      SC2._State == STANDBY and
                      SC3._State == STANDBY: {
                          SC1._State := WORKING;
                          SC2._State := WORKING; }
                 default: skip; }
                          }
assertion
  areFailed := SC1.isFAiled or SC2.isFAiled or
               SC3.isFAiled;
```

Figure 15 : maintenances pour la séparation

3) Supervision du conditionnement

Pour la supervision du conditionnement, et comme le montre la Figure 16, il y a deux parties. La première partie importe les composants de conditionnement, définis dans la partie du système (voir Figure 9). La seconde partie spécifie les maintenances.

```
block Supervision
  block Packaging
    embeds main.Plant.Packaging.PC1 as PC1;
    embeds main.Plant.Packaging.PC2 as PC2;
    // Maintenances
    // ...
  end
end
```

Figure 16 : bloc de supervision du conditionnement

Le domaine de valeurs ci-dessous, préalablement défini, permettra de spécifier les différents modes de maintenance des composants de conditionnement :

```
domain PackagingSMode {OPERATION, MAINTENANCE_C1,
                       MAINTENANCE_C2}
```

Les maintenances sont décrites en Figure 17. Les variables d'état '_Mode' et '_Next' permettent de définir le mode actuel de la maintenance, suivant que ce soit le composant 'PC1' ou 'PC2' pour lequel il faut réaliser une maintenance préventive, ainsi que le prochain composant qui sera à maintenir. Les maintenances préventives sont modélisées par les événements 'startMaintanceC1' et 'startMaintenanceC2', avec leurs transitions associées. Elles spécifient les démarrages des maintenances de chacun des composants avec un délai de un an. L'événement 'finishMaintenance' spécifie les fins de maintenance. Enfin l'événement 'startRequiredMaintenance' spécifie le démarrage d'une maintenance corrective, qui est requise lorsqu'un composant tombe en panne avant sa période de maintenance.

```
// Maintenance part
PackagingSMode _Mode (init=OPERATION);
PackagingSMode _Next (init=MAINTENANCE_C1);
parameter Real pStart = 8760.0;
parameter Real pFinishMin = 6.0;
parameter Real pFinishMax = 12.0;
parameter Real pRequiredMin = 24.0;
parameter Real pRequiredMax = 48.0;
event startMaintenanceC1,
      startMaintenanceC2 (delay=Dirac(pStart));
event finishMaintenance
        (delay=uniform(pFinishMin,pFinishMax));
event startRequiredMaintenance
    (delay = uniform(pRequiredMin,pRequiredMax));
transition
  startMaintenanceC1: _Mode == OPERATION and
                  _Next == MAINTENANCE_C1 -> {
                        _Mode := MAINTENANCE_C1;
                        PC1._State := MAINTENANCE;
                        _Next := MAINTENANCE_C2;}
  startMaintenanceC2: _Mode == OPERATION and
                  _Next == MAINTENANCE_C2 -> {
                        _Mode := MAINTENANCE_C2;
                        PC2._State := MAINTENANCE;
                        _Next := MAINTENANCE_C1;}
  finishMaintenance: _Mode == MAINTENANCE_C1 or
                     _Mode == MAINTENANCE_C2 -> {
                           _Mode := RCCM_OPERATION;
                  if PC1._State == MAINTENANCE
                     then PC1._State := WORKING;
                  if PC2._State == MAINTENANCE
                     then PC2._State := _WORKING;}
  startRequiredMaintenance: (PC1.isFailed or
      PC2.isFailed) and _Mode == RCCM_OPERATION
      -> switch {
    case PC1.isFailed: {
       _Mode := MAINTENANCE_C1;
       PC1._State := MAINTENANCE;
       _Next := MAINTENANCE_C2; }
    case PC2.vfIsFailed: {
       _Mode := MAINTENANCE_C2;
       PC2._State := MAINTENANCE;
       _Next := MAINTENANCE_C1; }
    default: skip; }
```

Figure 17 : maintenances pour le conditionnement

V. ANALYSES ET EXPÉRIMENTATIONS

La partie IV précédente a présenté le modèle AltaRica 3.0 complet du système de production, considéré en Figure 1. Bien que ce modèle ait l'air d'être conséquent et/ou compliqué, il n'en est rien en rapport aux concepts sous-jacents appréhendés, qui sont eux-mêmes compliqués, car en quantité et diversité importante : comportements dysfonctionnels multiples des composants, comportements fonctionnels multiples entre les composants, et les différentes politiques de maintenance. Nous allons présenter dans cette partie des éléments permettant d'analyser le modèle, puis nous présenterons des résultats d'expérimentations via la simulation stochastique.

A. Analyse de la modélisation

Le Tableau 1 présente différentes quantités du modèle à la conception, c'est-à-dire tel que construit par le concepteur, et

à l'exécution, c'est-à-dire lorsque le modèle est compilé pour être ensuite évalué par des outils de calculs. Nous pouvons observer que les phénomènes appréhendés ne sont modélisés que par peu d'éléments atomiques dans le modèle : les variables, les transitions et les assertions.

Tableau 1 : Données quantitatives du modèle complet

	conception	exécution
Nombre de lignes du modèle	≈ 350	257
Nombre de variables d'états	9	14
Nombre de variables de flux	20	35
Nombre de transitions	18	30
Nombre d'assertions	27	35

En ne considérant par exemple que la partie de séparation du système, avec sa propre partie associée de supervision, la Figure 18 représente les deux machines à états/transitions des composants (en haut) et de la supervision (en bas). Ces deux machines sont très simples : quatre états et cinq transitions pour la première, deux états et deux transitions pour la seconde. Cependant en les combinant comme nous l'avons réalisé, c'est-à-dire en manipulant les changements d'états des machines des composants via les transitions de la machine de la supervision comme montré en Figure 19, nous avons complexifié la machine issue de cette opération de combinaison de manière très simple.

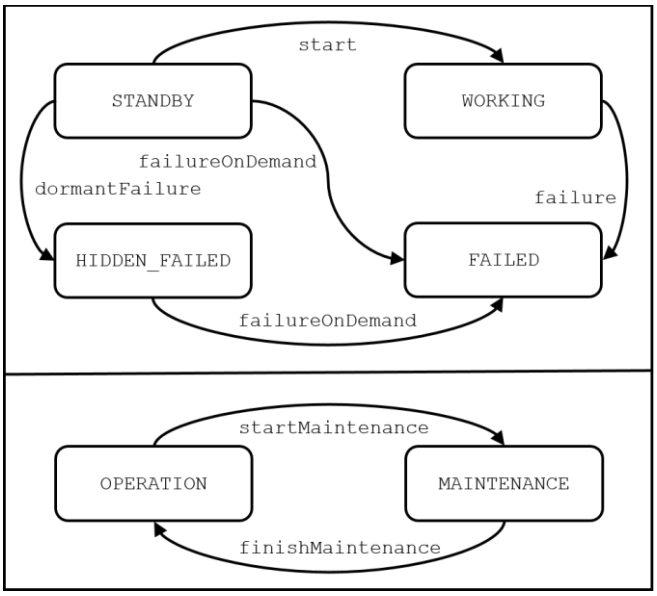

Figure 18 : machines à états/transitions de la partie séparation

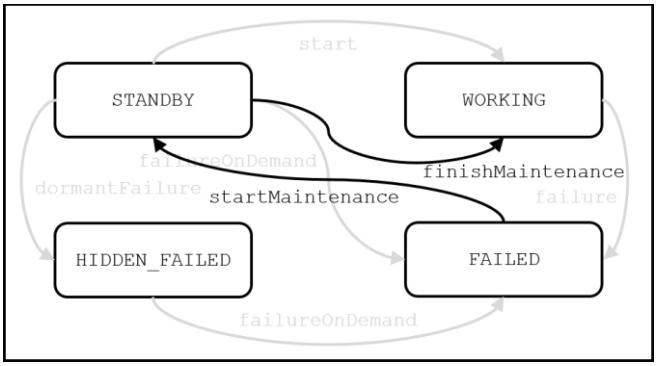

Figure 19 : modification des états des composants de séparation

Cette manière de modélisée permet d'optimiser le modèle par rapport aux nombres de transitions et de variables. En effet dans une modélisation classique, d'une part des variables de flux auraient été créées afin de faire transiter l'information des états des composants de et vers la supervision. D'autre part des transitions auraient aussi été créées afin de mettre à jour les différents états des éléments (composants et supervision), en fonction des changements d'états des autres éléments. Conceptuellement avec une telle modélisation, 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. Notre modélisation nous a permis de retirer ces doublons, et donc d'avoir un modèle plus optimisé.

Comme la majorité des analyses sur ce type de modèles se réalisent via la simulation stochastique, il est donc nécessaire d'avoir un modèle optimisé. Cela se justifie par le fait que, comme montré en [2] et [3], la simulation stochastique repose principalement sur deux mécanismes : la gestion d'un échéancier des transitions qui sont tirables en fonctions des valeurs des variables du modèle, et la mise à jour des variables du modèle suite au tirage d'une transition. L'efficacité de la simulation stochastique, au sens rapidité d'exécution, va donc dépendre d'une part de l'efficacité du simulateur, mais aussi de l'optimisation du modèle.

B. Expérimentations par simulation stochastique

Nous avons réalisé une expérimentation afin d'évaluer l'observateur Booléen '`oSystemProduction`' qui observe l'entrée de la cible du système. Nous pouvons utiliser cet observateur pour étudier la disponibilité de production du système, c'est-à-dire lorsqu'il est à la valeur '`true`'. Nous avons utilisé le simulateur stochastique de la plateforme OpenAltaRica (voir [2] pour une explication sur son fonctionnement) pour évaluer différents indicateurs sur cet observateur. Le Tableau 2 fourni les résultats obtenus pour 10^7 exécutions. Sur un temps de mission de 20 ans, nous avons réalisé des statistiques sur les valeurs de cet observateur à vrai ou faux suivant différents indicateurs et à différents instants de mission (5, 10, 15 et 20 ans).

Nous n'avons pas considéré les temps d'exécution nécessaires pour simuler ces histoires. En effet, nous ne cherchons pas à réaliser des évaluations de performance. Le lecteur intéressé pourra se référer à [2] ou [3].

Encore une fois, l'objectif n'était pas, dans cette communication, d'analyser des expérimentations, c'est-à-dire des résultats quantitatifs, sur ce modèle. En effet, pour pouvoir mener de telles analyses, il est nécessaire d'avoir à disposition d'une part les valeurs des différents taux que nous avons mis en paramètres : taux des différentes défaillances des composants, paramètres de durées de maintenances, etc. ; ce que nous n'avons pas précisément. D'autre part, il est nécessaire de pouvoir se comparer à d'autres études similaires. Pour ce second point, très peu d'éléments de comparaison peuvent être trouvés dans la littérature. Il faut en effet pouvoir comparer du point de vue de l'activité de modélisation, et du point de vue des résultats obtenus par les moteurs de calculs. Concernant la modélisation il n'existe pas, à notre connaissance, de langage supportant les primitives structurelles qui nous ont permis de modéliser efficacement ces combinaisons des différents types de politiques de maintenance. Concernant les résultats obtenus par les moteurs de calculs, de telles comparaisons ne sont pertinentes que si elles se basent sur les mêmes éléments modélisés en entrée. Cela ne nous a néanmoins pas été possible jusque maintenant.

De futurs travaux pourront s'intéresser à de telles comparaisons.

Tableau 2 : évaluation du modèle par simulation stochastique

Fired events		Mean: 251.919	Min: 223	Max : 312
5 years	true sojourn-time			43598.6
	false number-of-occurrences			11.9937
	false mean-time-between-occurrences			3021.10
10 years	true sojourn-time			87168.0
	false number-of-occurrences			25.9972
	false mean-time-between-occurrences			3185.03
15 years	true sojourn-time			130764.0
	false number-of-occurrences			38.0151
	false mean-time-between-occurrences			3226.27
20 years	true sojourn-time			174335.0
	false number-of-occurrences			52.0270
	false mean-time-between-occurrences			3249.52

VI. Conclusion

Au travers de cette communication, nous avons montré comment modéliser un système combinant différentes politiques de maintenances sur différents composants. Le modèle a été réalisé avec le langage AltaRica 3.0 et des expérimentations ont été produites par simulation stochastique sur ce modèle.

Pour la construction du modèle, nous avons suivi un schéma de modélisation structuré en deux niveaux : le niveau du processus physique et le niveau de supervision. Nous avons de plus utilisé une approche de conception 'top-down'. Nous avons d'abord construit le squelette du système avec sa partie de supervision et sa partie de processus physique (les composants de production). Ce squelette a été construit sans préciser les différents comportements des éléments de la supervision et des composants. Nous avons ensuite construit les différents composants et spécifié les politiques de maintenances dans la supervision. Cette approche de conception a été possible grâce aux constructions structurelles du langage AltaRica 3.0, notamment les éléments prototype et les éléments classes.

Nous avons réalisés des analyses et expérimentations sur le modèle AltaRica 3.0 construit. Nous avons montré que, malgré des concepts sous-jacents appréhendés qui sont compliqués et en quantité et diversité importante, le modèle est relativement simple en termes de nombre d'éléments atomiques nécessaires (variables et transitions). De plus, comme l'objectif n'était pas, dans cette communication, d'analyser des expérimentations, nous avons montré des résultats quantitatifs issus de simulations stochastiques sur ce modèle.

La présentation de ce schéma de modélisation, et du modèle associé, permettant de considérer différentes politiques de maintenance en AltaRica 3.0, continue la voie ouverte vers de nouveaux horizons depuis déjà quelques communications. 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. À terme, l'étude d'un tel modèle permettra de produire, ou de vérifier des scénarios spécifiques de politiques de maintenance du système, par exemple si l'on souhaite étudier les scénarios optimaux en termes de disponibilité ou de coûts d'exploitation.

References

[1] Afnor NF EN 13306 X 60-319

[2] B. Aupetit, M. Batteux, A. Rauzy, and J.-M. Roussel, "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, 2015

[3] B. Aupetit, M. Batteux, A. Rauzy, and J.-M. Roussel, "Vers la définition d'un kit d'évaluation pour les simulateurs stochastiques", Actes du Congrès Lambda-Mu 20 (actes électroniques). Saint-Malo, France, 2016

[4] M. Batteux, T. Prosvirnova, and A. Rauzy. "AltaRica 3.0 in 10 Modeling Patterns". In *International Journal of Critical Computer-Based Systems*. Inderscience Publishers. Vol. 9, Num. 1–2, pp 133–165, 2019.

[5] M. Batteux, T. Prosvirnova, and A. Rauzy. "From Models of Structures to Structures of Models". In *IEEE International Symposium on Systems Engineering (ISSE 2018)*. Roma, Italy. October, 2018.

[6] M. Batteux, T. Prosvirnova, and A. Rauzy. "Altarica 3.0 assertions: the why and the wherefore". Journal of Risk and Reliability. Professional Engineering Publishing. September, 2017.

[7] C. G. Cassandras, and S. Lafortune. « Introduction to Discrete Event Systems". New-York, NY, USA: Springer, 2008.

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

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

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

[11] A. Zimmermann. "Stochastic Discrete Event Systems". Springer-Verlag Berlin Heidelberg, 2008.