MODELISATION EN ALTARICA DES SYSTEMES AVEC DES COMPOSANTS MOBILES

MOBILITY MODELING WITH ALTARICA

PROSVIRNOVA Tatiana et RAUZY Antoine
Ecole Polytechnique
Laboratoire d'Informatique (LIX)
91128 Palaiseau
tél: (+33) 1 69 33 60 42
fax: (+33) 1 69 33 30 14
prosvirnova@lix.polytechnique.fr, rauzy@lix.polytechnique.fr

Résumé

Dans cet article, nous montrons que le langage AltaRica peut être utilisé efficacement pour modéliser les systèmes à composants mobiles, tels que les chaînes de production et les réseaux de communication, et évaluer leurs performances (fiabilistes). Nous introduisons un nouveau concept du langage : la notion de synchronisation gardée. Nous illustrons notre propos à l'aide d'exemples.

Summary

The goal of this article is to demonstrate that AltaRica modelling language can be efficiently used to represent systems with mobile components, such as production chains or communication networks, and to evaluate their performance indicators. A new concept is introduced to the language: the notion of guarded synchronization. A case study of a production system is used to illustrate AltaRica concepts.

Introduction

Dans cet article, nous montrons que le langage AltaRica [1] peut être utilisé efficacement pour modéliser les systèmes à composants mobiles et évaluer leurs performances (fiabilistes).
De nombreux systèmes complexes comportent des composants mobiles. C'est le cas par exemple des chaînes de production ou des réseaux de communication. La modélisation et l'évaluation des performances de tels systèmes posent des problèmes spécifiques. En effet les formalismes de modélisation ont pour la plupart été conçus pour des systèmes dont l'architecture ne varie pas au cours de la mission.
Les systèmes à composants mobiles se décrivent assez naturellement suivant un paradigme place/composant. Dans ce paradigme, il y a deux types d'objets :
- une topologie, c'est-à-dire un certain nombre de places et une relation de voisinage entre ces places;
- des composants statiques ou mobiles ayant chacun son comportement propre. Tout composant se trouve dans une place et une seule. Les composants mobiles peuvent changer de place. La majorité des interactions entre composants se fait entre des composants se trouvant dans la même place.

Par exemple, une chaîne de production comporte un certain nombre de lieux où sont placées les machines (composants statiques). Les produits transformés par ces machines (composants mobiles) se déplacent (ou sont déplacés) d'un lieu à l'autre.
On trouve dans la littérature plusieurs formalismes plus ou moins dédiés à la modélisation de ce type de systèmes, parmi lesquels on peut citer : le π-calcul [6], PEPA (Performance Evaluation Process Algebra) Nets [4], les réseaux de Petri colorés [5]...
En dépit de leurs qualités ces formalismes s'accommodent mal d'un passage à l'échelle industrielle. Leur élégance, leur concision mathématique sont aussi leur faiblesse : ils sont de trop bas niveau pour être déployés lors de la conception des systèmes complexes réels.
Le langage AltaRica est un langage de modélisation de haut-niveau pour la sûreté de fonctionnement. Il permet de décrire des modèles à un haut niveau d'abstraction, très proche de l'architecture fonctionnelle des systèmes. Ses constructions syntaxiques autorisent la description rapide de modèles réutilisables. Il est entre autres possible d'associer aux modèles AltaRica des représentations graphiques, les rendant très proches d'un "Process & Instrumentation Diagram". Le langage AltaRica est utilisé par plusieurs ateliers de sûreté de fonctionnement : Cecilia OCAS (Dassault Aviation), Simfia (EADS Apsys) et Safety Designer (Dassault Systèmes). De nombreuses expériences industrielles réussies ont été menées avec le langage AltaRica [2, 3].
La sémantique formelle du langage, basée initialement sur les automates de modes [7] et ensuite sur les systèmes de transitions gardées [8], a permis de développer des algorithmes de traitement efficaces :
- compilation vers les arbres de défaillance,
- génération de séquences critiques,
- simulation stochastique, ou encore
- outils de model-checking.

Parmi les modèles AltaRica, on distingue deux grandes classes de modèles :
- Les modèles des systèmes statiques. Ces modèles simples mais de taille conséquente sont traités par génération automatique des arbres de défaillance pour ensuite en déduire des indicateurs fiabilistes (probabilité de l'événement sommet, facteurs d'importance...) et les scénarios de pannes (coupes minimales).
- Les modèles des systèmes dynamiques. Ils sont traités par la génération des séquences pour trouver les scénarios de pannes et par la simulation stochastique pour calculer les indicateurs fiabilistes [3].

La capacité d'expression du langage AltaRica ne se limite cependant pas à ces deux classes de modèles. Nous avons donc cherché à déterminer s'il était adapté à la modélisation de systèmes avec des composants mobiles et si les algorithmes de traitement développés pour AltaRica pouvaient être utilisés pour évaluer les performances de ce type de systèmes.

Le reste de cet article est organisé comme suit : la section 2 présente un exemple de chaîne de production qui nous servira de fil rouge; la section 3 illustre les différents concepts du langage AltaRica; la section 4 présente la modélisation du système de production; la section 5 introduit un nouveau concept du langage - les synchronisations gardées - et explique comment ce concept facilite la modélisation des composants mobiles. La section 6 conclut l'article et donne quelques perspectives.

Exemple introductif : une chaîne de production

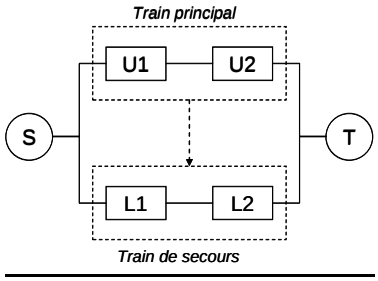

Figure 1. Système de production.

Le système de production Figure 1 est composé de deux chaînes de production en redondance froide. La chaîne de production principale est composée de deux unités de traitement en série U1 et U2. La chaîne de secours est composée de deux unités de traitement L1 et L2. Si U1 ou U2 tombe en panne (ou est arrêté pour effectuer une maintenance), la deuxième unité de traitement de la même chaîne est arrêtée et la chaîne de secours est sollicitée à démarrer. Quand U1 et U2 sont à nouveau mises en marche, la chaîne de secours est arrêtée.
On suppose que la Source S et la cible T ne sont pas défaillants et ne sont jamais arrêtées. Chaque unité de traitement X a un taux de défaillance λ, un taux de réparation μ et une probabilité de panne à la sollicitation γ. Le comportement de chaque unité de traitement obéit à l'hypothèse Markovienne.
Il faut calculer la disponibilité de production de ce système pour un temps de mission donné, par exemple un an. Il est possible d'estimer cette quantité en considérant les produits comme un flux continu. Cependant dans certains cas on aimerait pouvoir suivre la trajectoire de chaque produit traité par le système de production.
Dans la suite on suppose que:
- le taux de production de chaque unité de traitement est τ;
- chaque unité a une capacité limitée de stockage de produits κ;
- si l'unité tombe en panne ou doit être arrêtée avec un produit à l'intérieur, le produit doit être traitée à nouveau.

Le comportement de l'unité de traitement est représenté par l'automate Figure 2.
Chaque produit peut être localisé dans l'une des 6 places : S, U1, U2, L1, L2 et T.
Si le produit se trouve dans l'une des unités traitements il peut être soit non traité, soit en traitement, soit traité et donc prêt à être chargé par l'unité de traitement suivante.

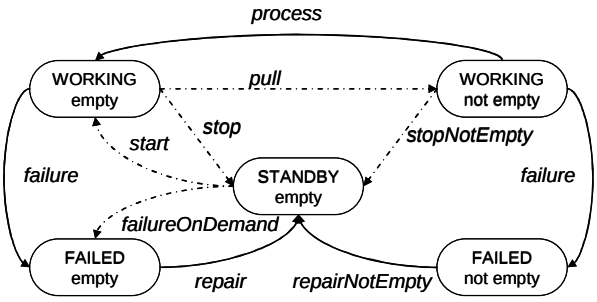

Figure 2. Unité de traitement: machine à états.

Le langage AltaRica

La nouvelle version du langage AltaRica intègre des notions des langages orientés objet - l'héritage et les objets globaux - ce qui améliore la réutilisation des modèles et la capitalisation des connaissances. La sémantique de la nouvelle version est basée sur le formalisme des Système de Transitions Gardées(GTS), défini par l'un des auteurs de cette communication dans [8], ce qui permet de modéliser des systèmes bouclés. Cette section présente les différents concepts du langage AltaRica, illustrés par la modélisation de l'exemple donné au début cette communication.

1 Automates d'états finis

Le langage AltaRica représente le comportement des composants du système en tant qu'automates d'états finis.
La machine de traitement de l'exemple décrit ci-dessus peut être modélisée par un automate d'états finis. Cet automate est décrit en AltaRica comme suit:

```
Enumeration Status {WORKING, FAILED, STANDBY};
node Machine
    Status status(init = STANDBY);
    bool empty(init = true);

    event start(delay = 0, expectation = 1 - gamma);
```

```
    event failureOnDemand (delay = 0, expectation = gamma);
    event stop (delay = 0);
    event stopNotEmpty(delay = 0);
    event pull(delay = 0);
    event process(delay = exponential(tau));
    event failure(delay = exponential(lambda));
    event repair(delay = exponential(mu));
    event repairNotEmpty(delay = exponential(mu));

    parameter float lambda    = 0.001;
    parameter float mu        = 0.1;
    parameter float tau       = 0.2;
    parameter float gamma     = 0.02;

  transitions
    start:           (status==STANDBY)              -> status := WORKING;
    failureOnDemand: (status==STANDBY)              -> status := FAILED;
    stop:            (status==WORKING) and empty    -> status := STANDBY;
    stopNotEmpty:    (status==WORKING) and not empty -> {status := STANDBY; empty := true;}
    failure:         (status==WORKING)               -> status := FAILED;
    repair:          (status==FAILED) and empty      -> status := STANDBY;
    repairNotEmpty:  (status==FAILED) and not empty  -> {status := STANDBY; empty := true;}

    pull:            (status==WORKING) and empty     -> empty := false;
    process:         (status==WORKING) and not empty -> empty := true;
end;
```

Tableau 1. Modèle AltaRica de la machine de traitement

Etats : L'état de la machine de traitement est représenté par deux variable d'état : la variable **status** de type **Status**, représentant l'état interne de la machine (si elle est en arrêt, en marche ou en panne) et la variable Booléenne **empty**, indiquant si la machine est en train de traiter un produit ou non. L'état initial du composant, i.e. les valeurs initiales de ses variables d'état, est précisé par l'attribut **init**.

Evénements : L'état de la machine de traitement change si certains événements se produisent. Les événements sont déclarés par le mot clé **event**. Un délai est associé avec chaque événement via l'attribut **delay**. Dans l'exemple ci-dessus les délais associés aux événements **failure**, **repair** and **process** sont des variables aléatoires ayant des distributions de probabilité exponentielles avec des taux **lambda**, **mu** et **tau**. Autrement dit, ses variables respectent l'hypothèse Markovienne. Les événements **start**, **failureOnDemand** sont immédiats. Les deux peuvent se produire si la machine est arrêté et vide. L'événement **start** a la probabilité **1-gamma** et l'événement **failureOnDemand** a la probabilité **gamma** de se produire dans cet état. Cette probabilité est précisée par l'attribut **expectation**. Les événements **repairNotEmpty** et **stopNotEmpty** sont introduits pour modéliser le fait que la machine doit se vider si elle est réparée ou arrêtée avec un produit en cours de traitement. Ils ont les mêmes propriétés que les événements **repair** et **stop**.

Transitions : Une transition est un triplet <e, G, P>, aussi noté e: G -> P, où e est un événement, G est une expression Booléenne, appelée la garde ou la pré-condition de la transition, et P est une instruction, appelée aussi l'action ou la post-condition de la transition. Les transitions sont décrites dans la clause **transitions**. Dans l'exemple ci-dessus si l'état de la machine est WORKING et not empty, deux transitions temporisées sont tirables : la transition associée à l'événement **failure** et la transition associée à l'événement **process**. Si le délai tiré pour l'événement **failure** est le plus petit, la transition correspondante est tirée; son action est exécutée : la valeur de la variable **status** devient **FAILED** et la variable **empty** garde sa valeur.
Sur la Figure 2, les transitions immédiates sont représentées par des lignes pointillées, tandis que les transitions temporisées sont représentées par des lignes pleines.

Paramètres : Les paramètres ont des valeurs constantes, ils peuvent être définis dans chaque nœud AltaRica. Quand un nœud est instancié, leur valeurs peuvent être changées. Dans l'exemple ci-dessus, quatre paramètres sont utilisés : **gamma**, **lambda**, **mu** et **tau**. Ils définissent respectivement la probabilité de défaillance à la sollicitation (événement **failureOnDemand**) et le taux de panne, de réparation et de traitement (événements **failure**, **repair**, **process**) de la machine.

2 Produit synchronisé

Considérons maintenant l'unité de traitement de l'exemple donné au début de cet article. Cette unité de traitement charge des produits, les traite avec sa machine et ensuite une fois traités les renvoie vers l'unité suivante. Une fois chargé dans l'unité de traitement, le produit peut alors être dans trois états différents: non traité, en cours de traitement et traité. Le produit peut être représenté en AltaRica comme suit:

```
enumeration ProductStatus {NOT_PROCESSED, IN_PROCESS, PROCESSED};

node Product
    ProductStatus status(init = NOT_PROCESSED);

    event pulled;
    event processed;
    event pushedBack;

  transitions
    pulled:          status==NOT_PROCESSED  -> status := IN_PROCESS;
```

```
    process:          status==IN_PROCESS     -> status := PROCESSED;
    pushedBack:       status==IN_PROCESS     -> status := NOT_PROCESSED;
end;
```

Tableau 2. Modèle AltaRica du produit

Composition : Le modèle de l'unité de traitement est obtenu par agrégation d'une instance de Machine avec un certain nombre d'instances de Product. Pour deux produits le code AltaRica est comme suit.

```
node Unit
    Machine M;
    Product P1, P2;
  transitions
    pull:             <!M.pull, !P1.pulled>,
                      <!M.pull, !P2.pulled>;
    process:          <!M.process, !P1.processed>,
                      <!M.process, !P2.processed>;
    repairNotEmpty:   <!M.repairNotEmpty, !P1.pushedBack>,
                      <!M.repairNotEmpty, !P2.pushedBack>;
    stopNotEmpty:     <!M.stopNotEmpty, !P1.pushedBack>,
                      <!M.stopNotEmpty, !P2.pushedBack>;
end;
```

Tableau 3. Modèle AltaRica de la machine traitant un produit

Synchronisation : La synchronisation consiste à obliger plusieurs événements à se produire simultanément. La synchronisation définit un nouveau macro-événement, et les événements synchronisées cessent d'exister indépendamment de la synchronisation. Dans l'exemple ci-dessus les événements **process** de la machine et **processed** du produit sont synchronisés. Le nouveau macro-événement **process** est créé. Les événements **M.process** et **P1.processed** ou **P2.processed** ne peuvent plus se produire indépendamment. Les événements **failure** ou **repair** peuvent toujours se produire indépendamment. Le macro-événement **process** hérite des propriétés des événements synchronisés, en particulier il hérite le délai stochastique de l'événement **M.process** de la machine.
Les événements **M.process** et **P1.processed** ou **P2.processed** sont préfixés par le point d'exclamation !. Cela signifie que tous les événements doivent être tirables pour que le macro-événement soit tirable. AltaRica permet aussi de représenter des synchronisations d'événements qui ne sont pas forcément tirables en même temps. Ceci permet de modéliser d'autres types de phénomènes, par exemple broadcast ou défaillance de cause commune.

Mise à plat : La définition du nœud **Unit** est le produit synchronisé de l'instance du nœud **Machine** et de deux instances du nœud **Product** Ce produit synchronisé est équivalent à un seul nœud AltaRica obtenu par l'opération de mise à plat. Le nœud **Unit** après l'opération de mise à plat serait comme suit.

```
node Unit
    Status M.status(init = WORKING);
    bool M.empty(init = true);
    ProductStatus P1.status(init = NOT_PROCESSED), P2.status(init = NOT_PROCESSED);
    event M.start(delay = 0, expectation = 1-M.gamma);
    event M.failureOnDemand(delay = 0, expectation = M.gamma);
    event M.stop(delay = 0);
    event M.failure(delay = exponential(M.lambda));
    event M.repair(delay = exponential(M.mu));
    event pull(delay = 0);
    event process(delay = exponential(M.tau));
    event repairNotEmpty(delay = exponential(M.mu));
    event stopNotEmpty(delay = 0);
    parameter float M.gamma = 0.02;
    parameter float M.lambda = 0.001;
    parameter float M.mu = 0.1;
    parameter float M.tau = 0.2;

  transitions
    M.failure: M.status==WORKING -> M.status := FAILED;
    M.repair:  (M.status==FAILED) and M.empty -> M.status := STANDBY;
    M.start:   (M.status==STANDBY) and M.empty -> M.status := WORKING;
    M.failureOnDemand: (M.status==STANDBY) and M.empty -> M.status := FAILED;
    M.stop:    (M.status==WORKING) and M.empty -> M.status := STANDBY;

    pull:            (M.status==WORKING) and M.empty and (P1.status==NOT_PROCESSED) ->
                                          {M.empty := false; P1.status := IN_PROCESS;}
    pull:            (M.status==WORKING) and M.empty and (P2.status==NOT_PROCESSED) ->
                                          {M.empty := false; P2.status := IN_PROCESS;}

    process:         (M.status==WORKING) and not M.empty and (P1.status==IN_PROCESS) ->
                                          {M.empty := true; P1.status := PROCESSED;}
    process:         (M.status==WORKING) and not M.empty and (P2.status==IN_PROCESS) ->
                                          {M.empty := true; P2status := PROCESSED;}

    repairNotEmpty: (M.status==FAILED) and not M.empty and (P1.status==IN_PROCESS)  ->
                          {M.status := STANDBY; M.empty := true; P1.status := NOT_PROCESSED;}
    repairNotEmpty: (M.status==FAILED) and not M.empty and (P2.status==IN_PROCESS)  ->
                          {M.status := STANDBY; M.empty := true; P2.status := NOT_PROCESSED;}

    stopNotEmpty:   (M.status==WORKING) and not M.empty and (P1.status==IN_PROCESS) ->
                           {M.status := STANDBY; M.empty := true; P1.status := NOT_PROCESSED;}
    stopNotEmpty:   (M.status==WORKING) and not M.empty and (P2.status==IN_PROCESS) ->
                           {M.status := STANDBY; M.empty := true; P2.status := NOT_PROCESSED;}
end;
```

Tableau 4. Modèle AltaRica de l'unité de traitement mise à plat

3 Variables de flux

Maintenant considérons un système un peu plus compliqué : dans le système décrit au début de cet article les unités de traitement U1 et L1 sont en redondance froide. Quand U1 est en panne, l'unité L1 doit démarrer. Pour modéliser ce phénomène nous utilisons des variables de flux. Le modèle de la machine donné dans la section 1 doit être modifié pour prendre en compte l'information venant de l'extérieur. Les différences par rapport au modèle de la section 1 sont soulignées.

```
node Machine
    Status status(init = STANDBY);
    bool empty(init = true);
    bool demanded(reset = false);

    event start(delay = 0, expectation = 1 - gamma);
    event failureOnDemand (delay = 0, expectation = gamma);
    event stop (delay = 0);
    event stopNotEmpty(delay = 0);
    event pull(delay = 0);
    event process(delay = exponential(tau));
    event failure(delay = exponential(lambda));
    event repair(delay = exponential(mu));
    event repairNotEmpty(delay = exponential(mu));

    parameter float lambda    = 0.001;
    parameter float mu        = 0.1;
    parameter float tau       = 0.2;
    parameter float gamma     = 0.02;

  transitions
```

```
    start:           (status==STANDBY) and demanded  -> status := WORKING;
    failureOnDemand: (status==STANDBY) and demanded  -> status := FAILED;
    stop:            (status==WORKING) and empty and not demanded -> status := STANDBY;
    stopNotEmpty:    (status==WORKING) and not empty and not demanded ->
                                                    {status := STANDBY; empty := true;}
    failure:         (status==WORKING)               -> status := FAILED;
    repair:          (status==FAILED) and empty      -> status := STANDBY;
    repairNotEmpty:  (status==FAILED) and not empty  -> {status := STANDBY; empty := true;}

    pull:            (status==WORKING) and empty     -> empty := false;
    process:         (status==WORKING) and not empty -> empty := true;
end;
```

Tableau 5. Modèle AltaRica de la machine de traitement

La variable **demanded** dans l'exemple est une variable de flux. Les variables de flux sont utilisées pour modéliser les flux d'information qui circulent à travers le système. Elles peuvent représenter des connections physiques entre composants, des actions de contrôle commande, l'alimentation électrique, etc. C'est une manière simple et élégante d'exprimer les dépendances provenant de l'extérieur. Dans l'exemple les événements **start**, **failureOnDemand** peuvent se produire si on essaye de démarrer l'unité de traitement; l'événement stop se produit immédiatement s'il n'est plus exigé à l'unité de fonctionner. Les variables de flux sont introduites par l'attribut **reset** à la place d'**init** qui caractérise les variables d'état.
Le système composé de deux machine de traitement en redondance froide est modélisé comme suit.

```
node TwoUnits
    Machine M1, M2(status::init = STANDBY);
  assertions
    M1.demanded := not (M1.status==FAILED);
    M2.demanded := (M1.status==FAILED) and not (M2.status==FAILED);
end;
```

Tableau 6. Modèle AltaRica de deux machines en redondance froide

Assertions : Les assertions sont un ensemble d'instructions qui expriment les dépendances entre les variables de flux et les variables d'état. Les assertions sont introduites dans la clause **assertions**. Dans l'exemple ci-dessus les assertions sont utilisées pour propager l'information du composant principal **M1** au composant de secours **M2** pour solliciter ce dernier à démarrer quand le composant **M1** tombe en panne.
Après chaque tir de transition les valeurs des variables d'état et de flux sont mises à jour en deux étapes. D'abord l'action de la transition est exécutée. Ensuite toutes les assertions sont exécutées en parallèle. L'ordre dans lequel les assertions sont exécutées ne doit pas influencer le résultat. Dans le cas où le résultat dépend de l'ordre de l'exécution des assertions, le modèle est considéré incorrect.
Dans l'exemple ci-dessus quand l'événement **M1.failure** se produit, d'abord **M1.status** devient **FAILED**, ensuite **M1.demanded** reçoit la valeur **false** et **M2.demanded** reçoit la valeur **true**.

Modèle AltaRica du système complet

Dans le système décrit au début de cette communication, en plus des pannes, des réparations et des sollicitations des unités de traitement, on cherche à modéliser les déplacements des produits à travers les différentes unités du système de production. Ainsi le produit peut se trouver dans la source S, dans chaque unité de traitement de la chaîne de production principale U1 et U2, dans chaque unité de traitement de la chaîne de secours L1 et L2 si la chaîne principale est en panne, et dans la cible T.
Pour modéliser le système de production complet il faut :
- modéliser la topologie du système, c'est-à-dire l'ensemble des places où se trouvent les composants statiques du système - unités de traitement - et où peuvent se trouver les composants mobiles du système - les produits;
- modéliser les déplacements des produits dans cette topologie;
- modéliser les interactions entre les composants mobiles (produits) et les composants statiques (unités de traitement);
- prendre en compte la capacité de stockage de chaque unité de traitement;
- modéliser la sollicitation de la chaîne de secours quand la chaîne principal tombe en panne.

Pour modéliser les systèmes à composants mobiles en AltaRica nous proposons la méthodologie suivante :
- La topologie du système est décrite par une énumération (exemple, énumération **Location** Tableau 9).
- Le composant mobile possède une variable qui représente sa localisation dans la topologie du système. Dans le cas de notre exemple, le produit circulant dans le système de production **MobileProduct** a une variable **location** qui prend ses valeurs dans l'ensemble des localités du système: *S, U1, U2, L1, L2, T*.
- Les déplacements possibles du composant dans le système sont modélisés par des événements et des transitions qui permettent au composant de changer de localité. Pour représenter le déplacement du produit dans le système de production des événements **moveU1**, **moveL1**, **moveU2**, **moveL2**, **moveT** sont introduits.
- Pour représenter les interactions des composants mobiles avec des composants statiques du système il faut synchroniser les événements de ces deux types de composants. Certains événements ne peuvent se produire que si le composant mobile se trouve dans une localité particulière du système.

Le modèle du produit qui circule dans le système de production peut être représenté comme suit.

```
enumeration Location {S, U1, U2, L1, L2, T};

node MobileProduct
          extends Product(status::init=PROCESSED);
```

```
    Location location(init = S);
    event moveU1, moveU2, moveL1, moveL2, moveT;
    event reset(delay = 0);
  transitions
    moveU1: (location==S) and (status==PROCESSED) -> {location:=U1; status:=NOT_PROCESSED;}
    moveU2: (location==U1) and (status==PROCESSED)-> {location := U2; status:=NOT_PROCESSED;}
    moveL1: (location==S) and (status==PROCESSED) -> {location := L1; status:=NOT_PROCESSED;}
    moveL2: (location==L1) and (status==PROCESSED)-> {location := L2; status:=NOT_PROCESSED;}
    moveT:  (location==U2) and (status==PROCESSED)-> location := T;
    moveT:  (location==L2) and (status==PROCESSED)-> location := T;
    reset:  (location==T)  -> location := S;
end;
```

Tableau 7. Modèle AltaRica d'un produit mobile

La variable **location** représente la localité du produit dans la topologie du système, les événements **moveU1**, **moveU2**, **moveL1**, **moveL2**, **moveT** modélisent les déplacements du produit. Le produit peut changer de localité une fois qu'il a été traité; à chaque fois que le produit change de localité il devient non traité.

Pour prendre en compte la capacité de stockage de produits κ de l'unité de traitement, on introduit une variable d'état Booléenne **loaded** et les événements **load** et **unload**. L'unité de traitement agrège une instance de **Machine**, dont le modèle AltaRica est donné dans la section 3. Le modèle AltaRica de l'unité de traitement est donné ci-dessous.

```
node Unit
    Machine M;
    int loaded (init = 0);
    bool demanded (reset = false), failed(reset = false);
    parameter int kappa = 2;
    event load, unload;
  transitions
    load: (loaded < kappa) and demanded -> loaded := loaded + 1;
    unload: (loaded > 0) and demanded -> loaded := loaded -1;
  assertions
    M.demanded := demanded;
    failed := (M.status == FAILED);
end;
```

Tableau 8. Modèle AltaRica de l'unité de traitement

Le modèle du système de production complet est obtenu par agrégation des modèles de l'unité de traitement **U1**, **U2**, **L1** et **L2** et d'un nombre quelconque de produits **P1**, **P2**, ..., **Pn**.
Pour représenter les déplacements des produits dans le système il est nécessaire de synchroniser les événements **moveU1**, **moveU2**, **moveL1**, **moveL2**, **moveT** de chaque produit avec les événements **load** et **unload** des unités de traitement **U1**, **U2**, **L1** et **L2**. Dans le code AltaRica ci-dessous, par exemple, l'événement **P1.moveU1** est remplacé par un événement synchronisé **<!U1.load, !P1.moveU1>**, i.e. le produit peut se déplacer de la source *S* dans l'unité *U1* que si la place dans **U1** est disponible et que si **U1** est en marche.
Pour modéliser les interaction des produits avec les machines de traitement il faut synchroniser les événements **pull**, **process**, **repairNotEmpty**, **stopNotEmpty** de chaque machine avec les événements **pulled**, **processed**, **pushedBack** de chaque produit. Mais ces événements synchronisés ne sont possibles que si le produit est localisé au bon endroit du système. Par exemple, la machine de l'unité de traitement **U1** ne peut charger le produit **P1** que si celui-ci est localisé dans la place *U1*. Pour modéliser cela on introduit des événements supplémentaires **checkP1inU1**, **checkP1inU2**, **checkP1inL1**, **checkP1inL2**, etc. et des transitions avec des post-conditions vides, par exemple

checkP1inU1: P1.location==U1 -> ;.

Ces événements sont ensuite synchronisés avec les événements des machines correspondantes et des produits. Par exemple, l'événement **checkP1inU1** est synchronisé avec les événements **U1.M.pull** et **P1.pulled**.
Ainsi, les interactions entre le produit **P1** et l'unité de traitement **U1** ne peuvent se produire que si le produit **P1** est localisé dans la place *U1*.

```
node ProductionSystem
    Unit U1, U2, L1(M.status::init = STANDBY), L2(M.status::init = STANDBY);
    MobileProduct P1, P2,..., Pn;
    event checkP1inU1, checkP1inU2, checkP1inL1, checkP1inL2, ...;
  transitions
    P1.moveU1: <!U1.load, !P1.moveU1>;
    P1.moveU2: <!U2.load, !P1.moveU2, !U1.unload>;
    P1.moveL1: <!L1.load, !P1.moveL1>;
    P1.moveL2: <!L2.load, !P1.moveL2, !L1.unload>;
    P1.moveT:  <!P1.moveT, !U2.unload>;
    P1.moveT:  <!P1.moveT, !L2.unload>;

    checkP1inU1: P1.location==U1 -> ;

    U1.M.pull:           <!checkP1inU1, !U1.M.pull, !P1.pulled>;
    U1.M.process:        <!checkP1inU1, !U1.M.process, !P1.processed>;
    U1.M.repairNotEmpty: <!checkP1inU1, !U1.M.repairNotEmpty, !P1.pushedBack>;
    U1.M.stopNotEmpty:   <!checkP1inU1, !U1.M.stopNotEmpty, !P1.pushedBack>;
```

```
    ...
  assertions
    U1.demanded := (not U1.failed) and (not U2.failed);
    U2.demanded := U1.demanded;
    L1.demanded := (not U1.demanded) and (not L1.failed) and (not L2.failed);
    L2.demanded := L1.demanded;
end;
```

Tableau 9. Modèle AltaRica du système de production

La sollicitation de la chaîne de secours est propagée par les assertions selon le mécanisme décrit dans la section 3.

Synchronisations gardées

Dans la section précédente pour modéliser les interactions entre les produits et les unités de traitement dans des localités spécifiques nous étions obligés d'introduire des événements et des transitions supplémentaires. Ces nouveaux événements ont été ensuite synchronisés avec des événements des machines et des produits.
Cette expérience de modélisation de systèmes à composants mobiles montre qu'il manque des concepts qui pourraient faciliter la modélisation de ce type de systèmes en AltaRica.

Suite à cette expérience nous avons introduit la notion de synchronisation gardée dans le langage AltaRica. Une synchronisation gardée est un triplet <e, G, S>, aussi noté e: G -> S, où e est un événement, G est une expression Booléenne, aussi appelée une garde ou une pré-condition, et S est un vecteur de synchronisation. Un vecteur de synchronisation est composé de deux ou plus événements qui peuvent être préfixés par un point d'exclamation !.
En imposant une condition sur la synchronisation des composants, on peut s'assurer que cette dernière ne se produit que dans certaines "localités". Deux composants ne vont pouvoir interagir que s'ils sont dans la même zone. En apparence minime, cette évolution du langage simplifie grandement la modélisation.
Voici comment le système de production peut être modélisé en utilisant les synchronisations gardées (les différences par rapport au modèle précédent sont soulignées):

```
node ProductionSystem
    Unit U1, U2, L1(M.status::init = STANDBY), L2(M.status::init = STANDBY);
    MobileProduct P1, P2,..., Pn;

  transitions
    P1.moveU1: <!U1.load, !P1.moveU1>;
    P1.moveU2: <!U2.load, !P1.moveU2, !U1.unload>;
    P1.moveL1: <!L1.load, !P1.moveL1>;
    P1.moveL2: <!L2.load, !P1.moveL2, !L1.unload>;
    P1.moveT:  <!P1.moveT, !U2.unload>;
    P1.moveT:  <!P1.moveT, !L2.unload>;

    U1.M.pull:           P1.location==U1 -> <!U1.M.pull, !P1.pulled>;
    U1.M.process:        P1.location==U1 -> <!U1.M.process, !P1.processed>;
    U1.M.repairNotEmpty: P1.location==U1 -> <!U1.M.repairNotEmpty, !P1.pushedBack>;
    U1.M.stopNotEmpty:   P1.location==U1 -> <!U1.M.stopNotEmpty, !P1.pushedBack>;
    ...
  assertions
    U1.demanded := (not U1.failed) and (not U2.failed);
    U2.demanded := U1.demanded;
    L1.demanded := (not U1.demanded) and (not L1.failed) and (not L2.failed);
    L2.demanded := L1.demanded;
end;
```

Tableau 10. Synchronisations gardées

Les synchronisations gardées simplifient la modélisation des systèmes avec des composants mobiles et unifient les concepts de transitions et de synchronisations. L'introduction de ce nouveau concept ne modifie en revanche pas le modèle d'exécution du langage. Il est donc possible d'appliquer les algorithmes de traitement existants sur les modèles des systèmes avec des composants mobiles. Il est notamment possible d'utiliser la simulation stochastique pour évaluer les performances de ces systèmes.
Dans la suite de nos travaux nous allons utiliser le simulateur stochastique du langage AltaRica pour évaluer les indicateurs de performance du système de production suivants :
- La production moyenne de la chaîne de production;
- Le temps moyen de traitement pour chaque produit;
- Le taux d'utilisation des machines de traitement.

Ces indicateurs seront calculées pour différents temps de mission.

Conclusion

Dans cet article nous avons montré sur un exemple de chaîne de production que le langage AltaRica pouvait être utilisé pour modéliser les systèmes à composants mobiles. Nous avons aussi introduit un nouveau concept dans le langage - la notion de synchronisation gardée. Cette extension du langage consiste à « garder » les synchronisations, au même titre que les transitions individuelles de composants le sont.

L'introduction de synchronisations gardées nous a permis d'accroitre l'élégance et la puissance d'expression du langage, sans rien changer aux algorithmes de traitement. Cette extension permet de modéliser la notion de localité des composants, qui nous semble fondamentale pour la modélisation de systèmes à composants mobiles.
La suite de nos travaux porte sur l'application des moteurs de calcul d'AltaRica, notamment du simulateur stochastique, pour évaluer les indicateurs de performance des systèmes avec des composants mobiles.

4 Références

1. ARNOLD, A. GRIFFAULT, G. POINT, A. RAUZY, 2000, The AltaRica Language and its Semantics, Fundamenta Informaticae, 34:109–124.
2. R. Bernard, J.-J. Aubert, P. Bieber, C. Merlini, S. Metge, 2007, Experiments in model-based safety analysis: flight controls, in Proceedings of IFAC workshop on Dependable Control of Discrete Systems.
3. M. BOITEAU, Y. DUTUIT, A. RAUZY, J.-P. SIGNORET, 2006, The AltaRica Data-Flow language in use: Assessment of Production Availability of a MultiStates System, Reliability Engineering and System Safety, 91:747-755.
4. S. GILMORE, J. HILLSTON, L. KLOUL, M. RIBAUDO, 2003, Pepa Nets: a structured performance modelling formalism, Performance Evaluation, 54:79 – 104.
5. K. Jensen, L.M. Kristensen, L. Wells, 2007, Coloured Petri Nets and CPN Tools for Modelling and Validation of Concurrent Systems, International Journal on Software Tools for Technology Transfer, Springer Verlag, 213-254.
6. R. Milner, 1999, Communicating and Mobile systems: The π-calculus, Cambridge University Press.
7. A. RAUZY, 2002, Modes Automata and their compilation into Fault Trees. Reliability Engineering and System Safety, 78:1-12.
8. A. RAUZY, 2008, Guarded transition systems: a new state/events formalism for reliability studies. Reliability Engineering and System Safety, 222(4):495-505.