INTRODUCTION AU NOUVEAU LANGAGE DE MODELISATION POUR LA SURETE DE FONCTIONNEMENT : ALTARICA NOUVELLE GENERATION

INTRODUCTION TO HIGH-LEVEL MODELING LANGUAGE FOR SAFETY ANALYSIS: ALTARICA NEXT GENERATION

PERROT Benoit, PROSVIRNOVA Tatiana, RAUZY Antoine, SAHUT D'IZARN Jean-Philippe
Dassault Systèmes
10, rue Marcel Dassault,
78140 Vélizy-Villacoublay
Tel. : (+33)1 61 62 41 32
Tatiana.Prosvirnova@3ds.com

Résumé

Cette communication a pour but de présenter une évolution directe d'AltaRica ([4]), langage de modélisation pour la sûreté de fonctionnement. Cet AltaRica « nouvelle génération » augmente le pouvoir d'expression de son prédécesseur : son orientation objet améliore les capacités de réutilisation et de capitalisation de connaissances d'AltaRica ; son nouveau modèle d'exécution plus puissant permet notamment de traiter des réseaux bouclés.

Summary

This article presents a new version of AltaRica, a modeling language for safety analysis. This next generation of AltaRica improves the expressivity of its predecessor: its object orientation property improves AltaRica reusability and knowledge capitalization capacities; its new execution model, more powerful, allows looping system modeling and assessment.

Introduction

La complexité croissante des nouveaux systèmes produits en industrie requièrent le développement de nouveaux outils d'ingénierie adéquats. Cette problématique se retrouve aussi dans le domaine de la sûreté de fonctionnement qui doit aussi s'adapter et se doter de nouvelles techniques de modélisation. En effet, l'expérience montre que les méthodes classiques utilisées par les fiabilistes, telles que les arbres de défaillance, les réseaux de Petri ou les chaînes de Markov ne permettent pas une bonne intégration des études de sûreté de fonctionnement lors du processus de développement d'un produit. En effet, la réalisation de ces études demande à la fois beaucoup de temps et d'expertise. Le caractère monolithique des méthodes employées, leur spécialisation et leur manque de structuration mènent souvent à la création de modèles difficiles à réutiliser, à maintenir, à adapter aux changements d'hypothèses et à partager avec des ingénieurs n'ayant pas une formation suffisante en sûreté de fonctionnement. Il est enfin difficile d'assurer la traçabilité entre les modèles fiabilistes et les architectures systèmes.

Pour résoudre ces problèmes, le langage de modélisation AltaRica a été créé en 1996. Il permet d'écrire des modèles pour la sûreté de fonctionnement à un haut niveau d'abstraction, très proche de l'architecture fonctionnelle. Ses constructions syntaxiques permettent la structuration de modèles en hiérarchie de composants et la description rapide de modèles réutilisables. Grâce aux interfaces graphiques Cecilia OCAS (Dassault Aviation) et son successeur Safety Designer (Dassault Systèmes, www.3ds.com), il est en outre possible d'associer aux modèles des descriptions graphiques, les rendant visuellement très proches des P&ID (Process and Instrumentation Diagram) [8]. Le modèle d'exécution d'AltaRica, basé sur le formalisme des automates de mode, a également permis de développer un ensemble d'outils de traitement automatique: un compilateur vers les arbres de défaillance [7], un générateur de séquences et un moteur de simulation stochastique.

Cependant, les besoins en création de modèles réutilisables, en capitalisation des connaissances, en association de représentations graphiques aux modèles, en partage avec des non-spécialistes du domaine et en traitements automatiques se retrouvent aussi dans d'autres domaines de l'ingénierie système. L'approche orientée modèle à base des langages de haut-niveau d'abstraction est aussi utilisée. Par exemple, le langage de modélisation Modelica [5] est utilisé pour modéliser le comportement des systèmes dynamiques à base d'équations différentielles et algébriques, et le langage Lustre [6] est employé pour modéliser le control-commande à base des automates d'états finis. Tous ces langages ont des propriétés communes: ils partagent à un certain niveau la manière dont ils structurent les modèles.

Pour être compatible (au sens de la compatibilité des modèles de structure) avec les langages de description, tels que Lustre ou Modelica, les constructions permettant de décrire la structure du langage AltaRica ont été changées dans la nouvelle version que nous proposons et la syntaxe a évolué pour s'approcher plus de la syntaxe de Modelica.

D'autre part, certaines limitations du langage ont pu être identifiées. Parmi celles-ci, l'impossibilité de traiter des réseaux bouclés, l'absence de la dérivation (au sens de la programmation orientée objet) et la description compliquée des propriétés des événements (via la clause *extern*) lui sont le plus souvent reprochées. Pour pouvoir supporter les systèmes bouclés le modèle d'exécution d'AltaRica a aussi été amélioré dans notre nouvelle version.

Dans cet article nous présentons ces évolutions apportées au langage AltaRica. Dans la première partie de cet article nous donnerons un exemple introductif qui illustrera par la suite les évolutions apportées au langage. Nous exposerons ensuite les différents concepts du langage avant de présenter ses différentes applications.

Exemple introductif

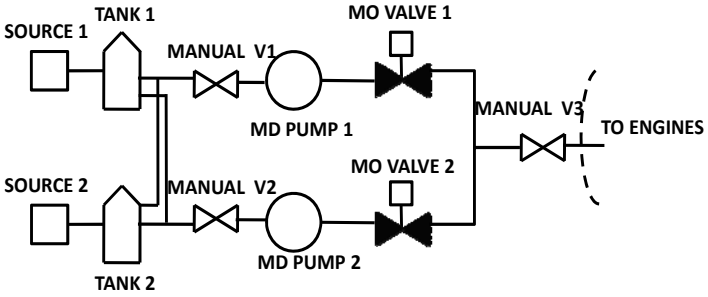
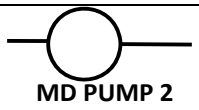


Figure 1 : Système d'alimentation de moteurs.

Voici un exemple simplifié d'une partie du système d'alimentation en essence de moteurs. Deux sources SOURCE1 et SOURCE2 alimentent les réservoirs TANK1 et TANK2 respectivement. Les réservoirs sont interconnectés et alimentent les moteurs à travers une chaîne de vannes et de pompes. On cherche à calculer la probabilité de panne de ce système. L'événement redouté survient si les moteurs ne sont pas alimentés, i.e. le flux en sortie de la vanne V3 est nul.

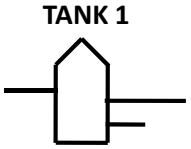
La méthode classique de résolution de ce problème est la construction d'un arbre de défaillance "à la main". Mais ici, nous utiliserons l'approche orientée modèle: nous modéliserons ce système avec la nouvelle version du langage AltaRica ce qui nous permettra ensuite de générer automatiquement l'arbre de défaillance. Cet exemple nous aidera à illustrer les concepts de la nouvelle version du langage, détaillés dans la suite de l'article.

<table>
<tr><td> 
</td><td>

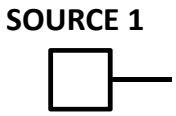
```
type Status = enumeration(working, failed, under_repair);
class Pump
  Boolean input(reset = false), output(reset = false);
  Status s(init = working);

transitions
  fail: s == working -> s := failed;
  start_repair: s == failed -> s := under_repair;
  end_repair: s == under_repair -> s := working;

assertions
  output := if working then input else false;
end;
```

</td></tr>
<tr><td> 
</td><td>

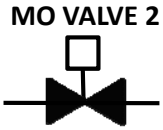
```
class ManualValve
  Boolean isClosed(init = false);
  Boolean input(reset = false), output(reset = false);
transitions
  open: isClosed -> isClosed := false;
  close : not isClosed -> isClosed := true;
assertions
  output := if isClosed then input else false;
end;
```

</td></tr>
<tr><td> 
</td><td>

```
class Tank
  Boolean in1(reset = false), in2(reset = false),
out(reset = false);
assertions
  out := in1 or in2;
end;
```

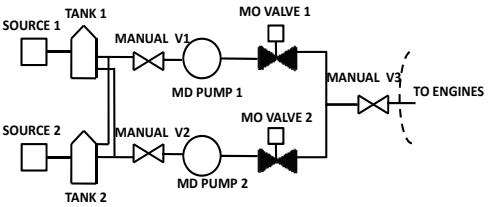
</td></tr>
<tr><td> 
</td><td>

```
class Source
  Boolean s(init = true);
  Boolean out(reset = false);
transitions
  failure: s -> s := false;
  repair: not s -> s:= true;
assertions
  out := s;
end;
```

</td></tr>
<tr><td> 
</td><td>

```
class MOValve
  Boolean isClosed(init = false);
  Boolean input(reset = false), output(reset = false);
  Status s(init = working);

transitions
  fail: s == working -> s := failed;
  start_repair: s == failed -> s := under_repair;
  end_repair: s == under_repair -> s := working;
  open: isClosed -> isClosed := false;
  close : not isClosed -> isClosed := true;
assertions
  output := if (isClosed and (s == working)) then input
```

</td></tr>
</table>




```
else false;
end;
```

```
class FuelSystem
  Source s1, s2;
  Tank t1, t2;
  ManualValve v1, v2, v3;
  Pump p1, p2;
  MOValve mov1, mov2;

transitions
  ccf_pumps : p1.fail || p2.fail;
assertions
  t1.in1 := s1.out;
  t1.in2 := t2.out;
  t2.in1 := t1.out;
  t2.in2 := s2.out;
  v1.input := t1.out;
  v2.input := t2.out;
  p1.input := v1.output;
  p2.input := v2.output;
  mov1.input := p1.output;
  mov2.input := p2.output;
  v3.input := mov1.output or mov2.output;
end;
```

Figure 2 : Model AltaRica "nouvelle génération" du système étudié.

Ci-dessus nous proposons une modélisation AltaRica "nouvelle génération" du système étudié. Chaque composant du système est modélisé par une classe. La modélisation du système lui-même n'est qu'un assemblage de composants déjà existants, c'est donc également une classe. Cette modélisation très proche de l'architecture logique du système permet de prendre en compte très facilement toutes les modifications de l'architecture du système étudié. L'état du composant est représenté par une variable et sa valeur; le changement d'état est décrit par des transitions auxquelles on associe un nom. Les assertions sont des instructions permettant de décrire comment les variables de sortie dépendent des variables d'entrée et de variables d'état. Ces concepts sont détaillés dans la suite de l'article.

Concepts du langage

La définition des améliorations à apporter au langage s'est articulée autour de deux axes. Le premier a consisté en la formalisation des constructions orientées objet nécessaires à une bonne réutilisabilité des modèles ainsi qu'à une compatibilité native avec d'autres langages de l'ingénierie système. Le second s'est attaché à la spécification d'un modèle d'exécution directement compatible avec celui d'AltaRica, et apte à traiter les systèmes bouclés.

1 Structure

Les langages de modélisation de l'ingénierie système, tels que Modelica [5] ou Lustre [6], ont été conçus pour répondre à des besoins spécifiques, et proposent de fait des syntaxes et des sémantiques d'exécution qui leurs sont propres. Pourtant ils partagent, jusqu'à un certain point, la manière dont ils structurent leurs modèles en hiérarchies de « boîtes » (e.g. paquets, classes). Ces « boîtes » sont connectées par des « fils » (e.g. opérations). Ces « fils » mettent en relation des éléments atomiques (e.g. variables, événements, constantes). La formalisation de ces « boîtes » et « fils » a abouti à la description de « S2ML » (System Structure Modeling Language) ([3]).

Pour représenter la structure d'un système, S2ML offre les éléments suivants:

- Atomes;
- Relations;
- Boîtes.

Les atomes représentent les composants élémentaires: ports, événements, variables, constantes, etc. Les relations permettent de décrire les fonctions, les équations, les instructions, i.e. les dépendances entre les atomes. Les boîtes sont utilisées pour représenter les classes, les nœuds, les blocs, etc. Une boîte peut contenir des atomes, des relations et des instances d'autres boîtes (notion de composition).

Ces éléments offrent déjà une grande capacité de modélisation. S2ML introduit également la notion d'héritage au sens de la programmation orientée objet. Si une boîte D dérive d'une boîte C, alors D contient tous les atomes, toutes les relations et toutes les boîtes contenus par C. L'héritage multiple est supporté par S2ML.

S2ML propose une manière élégante de définir et manipuler les objets globaux: atomes ou boîtes. Un objet peut être déclaré global à n'importe quel niveau hiérarchique. A partir du niveau auquel il a été déclaré global, l'objet n'appartient plus à la boîte englobant mais devient partagé entre toutes les boîtes.

S2ML permet de décrire la structure d'un système et ne spécifie aucune sémantique d'exécution. Ainsi, il est possible d'exprimer en S2ML des modèles Modelica, Lustre ou AltaRica, par exemple, dans un formalisme commun ne faisant pas intervenir leurs sémantiques d'exécution respectives.

La sémantique du modèle Modelica est un ensemble d'équations différentielles et algébriques. La sémantique du programme Lustre est un automate fini. La sémantique du modèle Altarica est un système de transitions gardées. Toutes ces sémantiques demandent de mettre à plat la hiérarchie de boîtes pour n'en avoir qu'une seule. En S2ML, un système est une instance de boîte et un système mis à plat est un ensemble d'atomes et de relations.

Dans la suite nous présenterons les concepts structurels de cette nouvelle version du langage AltaRica et leur relation avec les concepts introduits par S2ML : nous montrerons comment le modèle AltaRica "nouvelle génération" peut être exprimé naturellement en S2ML.

1.1 Classes et types

La nouvelle version du langage AltaRica manipule les types primitifs: Boolean, Integer, Real, String. Elle permet aussi de définir des énumérations et des types structurés.

Une classe peut représenter un composant, un sous-système, un équipement ou un système.

La définition des énumérations, des types structurés et des classes est illustré par la figure 3.

```
type Status =
enumeration(working, failed, under_repair);
```

```
record Channel
  Boolean b1;
  Boolean b2;
  Boolean b3;
end;
```

```
class Pump
  …
end;
```

Figure 3 : Définition des types et classes en AltaRica "nouvelle génération".

En S2ML les classes, les types structurés et les énumérations sont représentés par des boîtes.

1.2 Composition

Une classe peut contenir

- des variables d'un certain type,
- des instances d'autres classes,
- des transitions et
- des assertions.

En S2ML, les variables sont représentées par les atomes, les instances d'autres classes par des instances de boîtes, les transitions et les assertions par les relations entre les atomes (constantes et variables).

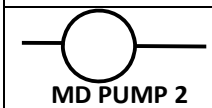
Reprenons l'exemple du système d'alimentation en essence décrit au début de cet article. La classe `Pump` ne contient que des atomes et des relations: les variables `s` de type prédéfini `Status`, `input` et `output` de type `Boolean`, ainsi que des assertions et des transitions qui expriment les dépendances entre les variables de la classe et les constantes. La classe `FuelSystem` est composée d'instances de boîtes et de relations: elle contient les classes `Tank`, `Source`, `Pump`, `ManualValve`, `MOValve`, ainsi que des assertions et des transitions. Le modèle du système est l'instance de la classe `FuelSystem`.

1.3 Héritage

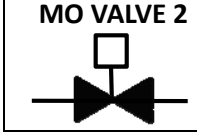
La nouvelle version du langage AltaRica est orientée objet. Elle se base sur les mécanismes d'héritage introduits par S2ML et décrits plus haut. Une classe peut donc dériver d'une autre. L'exemple ci-dessous illustre ce concept. Dans cet exemple nous proposons une autre modélisation des composants `Pump` et `Valve` du système d'alimentation en essence, en utilisant le concept d'héritage. Nous définissons un composant abstrait `RepairableComponent` qui décrit le comportement d'un composant réparable. La classe `RepairablePump` décrit juste les variables d'entrée et de sortie de la pompe et hérite du comportement du composant abstrait. De même, la classe `RepairableValve` hérite le comportement du composant abstrait et rajoute des transitions pour modéliser les aspects fonctionnels du comportement de la vanne.

Exemple :

```
type Status = enumeration(working, failed, under_repair);
class RepairableComponent
  Status s(init = working);

transitions
  fail: s == working -> s := failed;
  start_repair: s == failed -> s := under_repair;
  end_repair: s == under_repair -> s := working;

end;
```




```
class RepairablePump
  extends RepairableComponent

  Boolean input(reset = false);
  Boolean output(reset = false);

assertions
  output := if (s == working) then input else false;
end;
```




```
class RepairableValve
  extends RepairableComponent

  Boolean isClosed(init = true);
  Boolean input(reset = false), output(reset = false);
```

```
transitions
  open: isClosed -> isClosed := false;
  close: not isClosed -> isClosed := true;

assertions
  output := if ((s == working) and isClosed) then input
else false;
end;
```

Figure 4 : Héritage.

1.4 Objets globaux

Parfois il est utile de définir des objets globaux. La nouvelle version du langage AltaRica utilise les mécanismes de définition des objets globaux, spécifiés par S2ML. L'exemple ci-dessous illustre la définition et l'utilisation des objets globaux. La classe `Repairman` contient un objet global e de type `Environment`. Si un système contient plusieurs réparateurs, l'objet e de type `Environment` sera partagé entre tous les réparateurs.

```
type DayNight = enumeration(day, night);
class Environment
  DayNight t(init = day);

transitions
  sunset : t == day -> t := night;
  sunrise : t == night -> t := day;
end;
```

```
class Repairman
  global Environment e;
  Boolean isWorking(init = true);

assertions
  // A repairmen shall go to work at day
  isWorking := (e.t == day);
end;
```

Figure 5 : Objets globaux.

La nouvelle version du langage AltaRica conserve le caractère "compositionnel" de la version précédente – i.e. sa capacité à structurer un modèle en hiérarchie de composants - mais elle y ajoute une composante orientée objet qui améliore notablement ses capacités de réutilisation et de capitalisation des connaissances. En outre, cette nouvelle version se base sur le formalisme S2ML, ce qui la rend naturellement compatible avec les autres langages de modélisation dans le domaine de l'ingénierie système et lui confère ainsi un atout important pour faciliter l'intégration interdisciplinaire des modèles.

2 Modèle d'exécution

Le modèle d'exécution du langage se base désormais sur un nouveau formalisme mathématique – les Systèmes de Transitions Gardées – introduit par A. RAUZY dans [2]. Un Système de Transitions Gardées est un formalisme d'états et de transitions stochastiques qui généralise les formalismes classiques existants tels que diagrammes-blocs de fiabilité, réseaux de Petri et modèle de parallélisme d'Arnold-Nivat. Ce formalisme reprend les concepts d'état et de transition des réseaux de Petri, la notion de flux qui circule à travers un réseau de diagrammes-blocs de fiabilité, la notion de composition et de synchronisation du modèle d'Arnold-Nivat. Il enrichit le formalisme des automates de modes avec le mécanisme de calcul des assertions par point fixe après chaque tir de transition, ce qui offre une manière simple et élégante pour le traitement des réseaux bouclés.

Dans la nouvelle version du langage AltaRica, les variables sont divisées en deux catégories: celles qui reçoivent leurs valeurs initiales une fois au début de la simulation et celles qui reçoivent leurs valeurs initiales à chaque tir de transition. Ces dernières sont ensuite recalculées en fonction des premières.

Pour spécifier les valeurs initiales, ainsi que la catégorie de la variable, des mots-clés sont utilisés. Le mot clé "init" spécifie que la variable reçoit sa valeur initiale une seule fois au début de simulation. Le mot clé "reset" signifie que la variable reçoit sa valeur initiale à chaque tir de transition.


```
class ManualValve
  Boolean isClosed(init = false);
  Boolean input(reset = false), output(reset = false);
  …
end;
```

Figure 6 : Variables init et reset.

Dans l'exemple ci-dessus la variable `isClosed` représente l'état de la vanne, sa valeur initiale lui est affectée au début de la simulation. Les variables `input` et `output` représentent l'entrée et la sortie de la vanne, leurs valeurs sont remises à `false` à chaque tir de transition et sont recalculées en fonction de l'état du système.

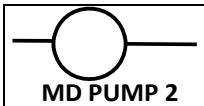
2.1 Transitions

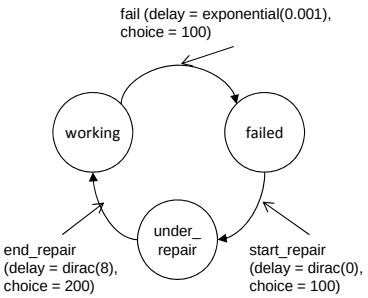
Une transition est un triplet <g, e, P>, où

- g est une garde, une expression booléenne,
- e est un événement auquel on peut associer une loi de délai et une loi de choix,
- P est une post-condition, une séquence d'instructions à exécuter après le tir de la transition.

Les transitions sont décrites dans la clause `transitions`. Une transition est représentée par son nom, sa garde et le bloc d'instructions de la post-condition. La transition peut être décorée par des données stochastiques. Le nom de la transition définit aussi l'événement associé à la transition.

Exemple:






```
type Status = enumeration(working, failed, under_repair);
class Pump
  Status s (init = working);

transitions
  fail(delay = exponential(0.001),
      choice = 100):
      s == working -> s := failed;
  start_repair (delay = dirac(0), choice = 100):
      s == failed -> s := under_repair;
  end_repair (delay = dirac(8), choice = 200):
    s == under_repair -> s := working;
end;
```

Figure 7 : Transitions.

Dans l'exemple ci-dessus les transitions `fail`, `start_repair` et `end_repair` définissent les trois événements. La garde de la transition `fail` est l'expression booléenne `(s == working)`, la post-condition est l'instruction `(s := failed)`. `Delay` et `Choice` sont des données stochastiques associées à l'événement.

Pendant la simulation, supposons que le système est dans un état S au temps t. Dans cet état plusieurs gardes sont vérifiées, donc plusieurs transitions peuvent être tirées. Mais il est impossible de tirer plusieurs transitions simultanément. La valeur de délai permet d'identifier l'événement qui se produit en premier parmi tous les événements possibles et ainsi choisir la transition qui doit être tirée. Le délai est une variable aléatoire qui détermine le temps entre le moment où la garde de la transition devient vérifiée et la survenue de l'événement. Il est représenté par sa distribution de probabilité qui peut dépendre du temps.

Cependant, il existe des situations où les événements peuvent se produire au même moment, ce qui veut dire que plusieurs transitions peuvent être tirées simultanément. Il est nécessaire alors de choisir la transition à tirer en premier. La loi de choix permet de choisir l'événement qui doit se produire en priorité. La loi de choix associe à chaque événement un entier non-négatif.

Si, toutefois, plusieurs transitions doivent être tirées simultanément, elles peuvent être synchronisées. Le mécanisme de synchronisation consiste à contraindre plusieurs transitions à être tirées simultanément. Une synchronisation définit une nouvelle transition à partir des transitions déjà existantes en utilisant les opérateurs suivants:

- ? – opérateur unaire "si possible",
- && - opérateur binaire "et",
- || - opérateur binaire "ou".

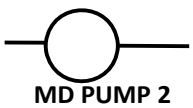
Les transitions qui sont impliquées dans la définition de la synchronisation peuvent être masquées, i.e. elles ne pourront plus être tirées indépendamment de la synchronisation.

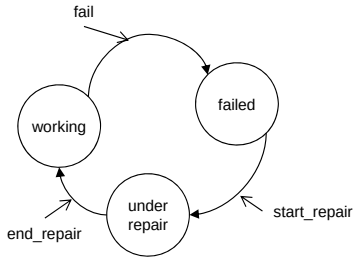
Les synchronisations sont typiquement utilisées pour modéliser les défaillances de cause commune, les réparations et la diffusion de l'information.

Exemple:

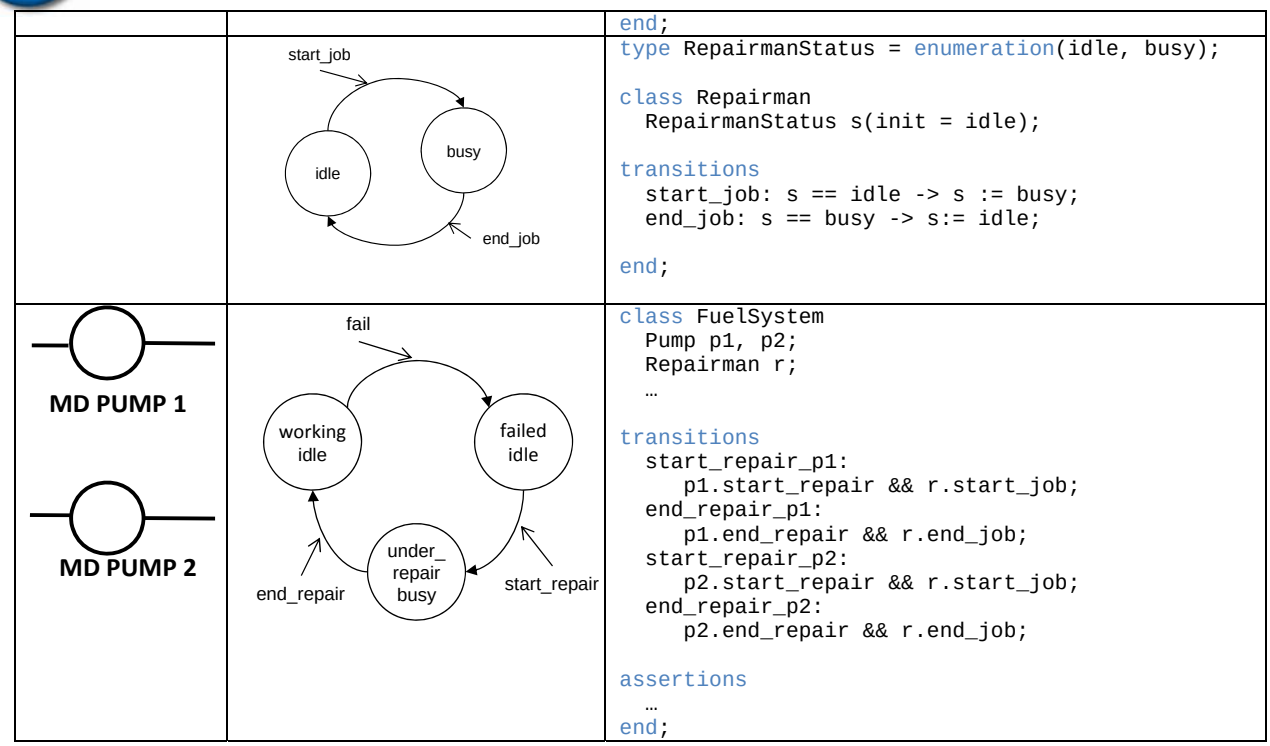
Réparation

Reprenons l'exemple donné au début de l'article. Supposons que les deux pompes MD PUMP 1 et MD PUMP 2 sont maintenues par une équipe de réparateurs qui ne peut intervenir que sur une pompe à la fois. Le système contient deux pompes et une équipe de réparateurs. Le code décrit le comportement de la pompe et de l'équipe de réparateurs. Au niveau système nous définissons deux transitions pour chaque pompe : start_repair_p1, end_repair_p1, start_repair_p2, end_repair_p2. Ces transitions synchronisent les transitions internes à la pompe et à l'équipe de réparateurs avec l'opérateur &&. Ceci signifie que la réparation ne peut avoir lieu que si la pompe est en panne et l'équipe de réparateurs est disponible. Les transitions impliquées dans les synchronisations doivent être masquées.






```
type Status = enumeration(working, failed, under_repair);

class Pump
  Status s(init = working);
  …
transitions
  fail: s == working -> s := failed;
  start_repair:
      s == failed -> s := under_repair;
  end_repair:
      s == under_repair -> s := working;

assertions
  …
```

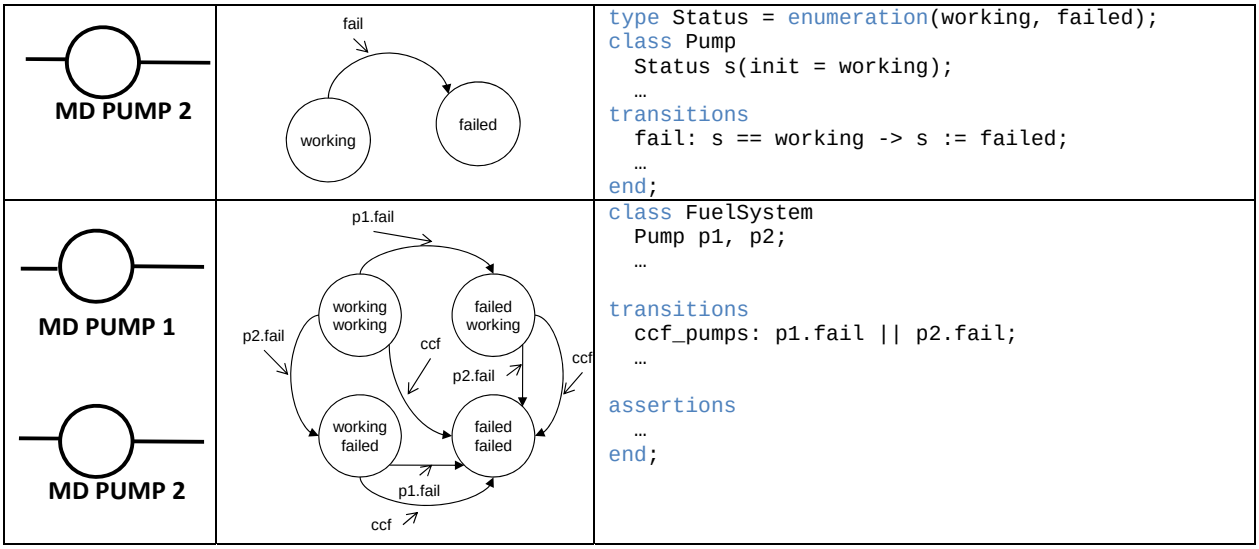



```
end;
type RepairmanStatus = enumeration(idle, busy);

class Repairman
  RepairmanStatus s(init = idle);

transitions
  start_job: s == idle -> s := busy;
  end_job: s == busy -> s:= idle;

end;

class FuelSystem
  Pump p1, p2;
  Repairman r;
  …

transitions
  start_repair_p1:
     p1.start_repair && r.start_job;
  end_repair_p1:
     p1.end_repair && r.end_job;
  start_repair_p2:
     p2.start_repair && r.start_job;
  end_repair_p2:
     p2.end_repair && r.end_job;

assertions
  …
end;
```

Figure 8 : Réparation.

Défaillance de cause commune

Dans l'exemple donné au début de cet article, le système contient plusieurs pompes de même nature. Supposons qu'en plus de leurs défaillances individuelles, ces pompes peuvent avoir une défaillance de cause commune. Pour modéliser cet événement, l'opérateur || peut être utilisé. Au niveau système nous définissons une transition `ccf_pumps` qui synchronisent les transitions `p1.fail`, `p2.fail`. Les transitions impliquées dans la synchronisation ne sont pas masquées.

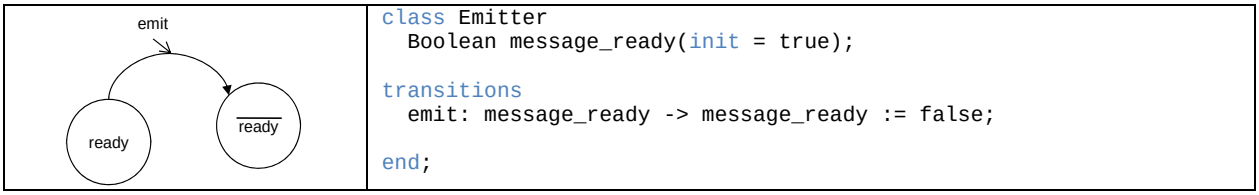



```
type Status = enumeration(working, failed);
class Pump
  Status s(init = working);
  …
transitions
  fail: s == working -> s := failed;
  …
end;
class FuelSystem
  Pump p1, p2;
  …

transitions
  ccf_pumps: p1.fail || p2.fail;
  …

assertions
  …
end;
```

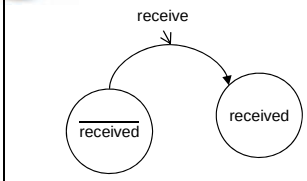
Figure 9 : Défaillance de cause commune.

Diffusion de l'information

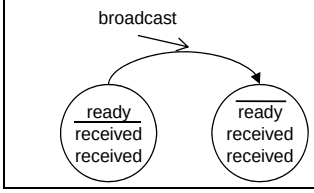
Dans l'exemple ci-dessous le système est composé d'un émetteur et de deux récepteurs. Nous modélisons la diffusion de l'information en utilisant les opérateurs de synchronisation && et ?. La transition `broadcast` peut être tirée même si l'un des deux récepteurs ne peut pas recevoir d'information.




```
class Emitter
  Boolean message_ready(init = true);

transitions
  emit: message_ready -> message_ready := false;

end;
```




```
class Receiver
  Boolean message_received(init = false);

transitions
  receive:
    not message_received -> message_received := true;

end;
```




```
class MyBroadcastSystem
  Emitter e;
  Receiver r1, r2(message_received.init = true);
transitions
  broadcast: e.emit && ?r1.receive && ?r2.receive;
end;
```

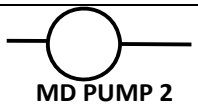
Figure 10 : Diffusion de l'information.

2.2 Assertions

Les assertions sont des instructions qui permettent de calculer les variables de reset en fonction des valeurs des variables d'init. Les assertions permettent de modéliser la propagation des erreurs à travers le système et de calculer l'impact d'une panne sur le fonctionnement global d'un système. Dans la nouvelle version du langage AltaRica, les assertions n'ont plus de notion de contrainte: ce sont des affectations.

Les assertions sont définies dans la clause *assertions*.

Exemple:



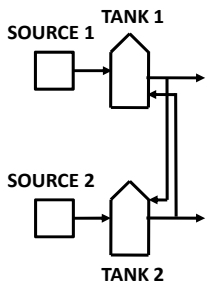

```
type Status = enumeration(working, failed, under_repair);
class Pump
  Boolean input(reset = false), output(reset = false);
  Status s(init = working);

transitions
  …
assertions
  output := if (s == working) then input else false;
end;
```

Figure 11 : Assertions.

A chaque tir de transition, les actions dans la post-condition de la transition sont exécutées en premier. Les valeurs des variables reset sont remises à leurs valeurs par défaut. Ensuite les assertions sont calculées par point fixe jusqu'à la stabilisation du système.

Reprenons l'exemple du début de l'article. Le système est composé de deux sources s1 et s2 qui alimentent les réservoirs t1 et t2 respectivement, le réservoir t1 alimente aussi le réservoir t2 et réciproquement. On suppose que seules les sources peuvent tomber en panne et être réparées. Ce système contient une boucle et illustre bien le calcul des assertions par point fixe**.**

Exemple




```
class Tank
  Boolean in1(reset = false), in2(reset = false), out(reset = false);
assertions
  out := in1 or in2;
end;

class Source
  Boolean s(init = true);
  Boolean out;
transitions
  failure: s -> s := false;
  repair: not s -> s:= true;
assertions
  out := s;
end;

class FuelSystem
  Source s1, s2;
  Tank t1, t2;
    …
assertions
  t1.in1 := s1.out;
  t1.in2 := t2.out;
  t2.in1 := t1.out;
  t2.in2 := s2.out;
    …
end;
```

Figure 12 : Système bouclé et sa modélisation.

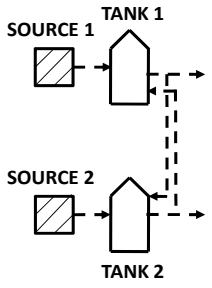
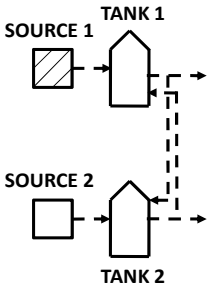
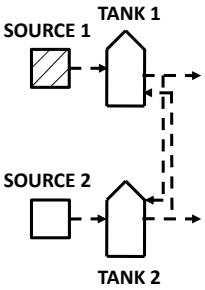
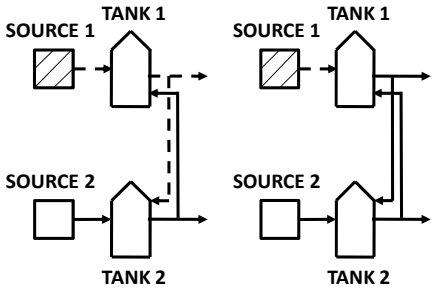
<table>
<tr>
<td> 
</td>
<td>0. On suppose que les sources <code>s1</code> et <code>s2</code> sont en panne. Il n'y a rien qui circule dans le système. On tire la transition <code>s2.repair</code>.</td>
<td> 
</td>
<td>1. L'action de la transition est exécutée en premier:
<code>s2.s := true;</code></td>
</tr>
<tr>
<td colspan="2"> 
</td>
<td colspan="2">2. Toutes les variables de reset sont initialisées à leurs valeurs par défaut:
<code>t1.in1 := false;
t1.in2 := false;
t2.in1 := false;
t2.in2 := false;
t1.out := false;
t2.out := false;
s1.out := false;
s2.out := false;</code></td>
</tr>
<tr>
<td colspan="2"> 
</td>
<td colspan="2">3. Exécution des assertions par point fixe:
<code>repeat
C1.in1 := S1.out;
C1.in2 := C2.out;
C2.in1 := C1.out;
C2.in2 := S2.out;
C1.out := C1.in1 or C1.in2;
C2.out := C2.in1 or C2.in2;
until nothing changes;</code></td>
</tr>
</table>

Figure 13 : Exécution des assertions par point fixe.

Tout en s'appuyant sur la structure événementielle de la précédente version, l'extension du langage AltaRica que nous proposons intègre plus naturellement dans les modèles les données stochastiques, telles que loi de délai et la loi de choix, grâce à sa nouvelle syntaxe. De plus, notre nouvelle version se base sur le formalisme mathématique des Systèmes de Transitions Gardées dont le modèle d'exécution intègre le calcul des assertions par point fixe. Ainsi, les systèmes bouclés peuvent désormais être traités, ce que ne permettait pas le formalisme mathématique de bas niveau choisi dans le langage AltaRica originel, qui s'appuyait sur les automates de mode.

Applications

Le projet CATIA Systèmes V6 de Dassault Systèmes a pour objectif de créer un atelier d'ingénierie système intégrant différentes disciplines : exigences, modélisation fonctionnelle et logique, description du comportement dynamique des systèmes, modélisation du control-commande, réalisation des études de sûreté de fonctionnement, tout en assurant la traçabilité et la cohérence entre tous les modèles.

La version précédente du langage AltaRica a déjà démontré ses capacités de traitement automatique. Le compilateur vers les arbres de défaillance [7], le générateur de séquences et le simulateur stochastique permettent de retrouver les outils classiques d'analyse de sûreté de fonctionnement. Il est aussi possible de créer des bibliothèques de composants réutilisables, ayant des représentations graphiques et des modèles très proches d'un P&ID grâce à l'éditeur de modèles Safety Designer (Dassault Systèmes). Cet outil permet en outre de réaliser la capitalisation des connaissances et le traitement automatique des modèles.

Les capacités d'interopérabilité entre modèles de différentes disciplines de l'ingénierie système de la nouvelle version du langage AltaRica offrent de nouvelles possibilités d'exploitation:

- Amélioration de la traçabilité entre les modèles;
- Partage de la structure jusqu'à un certain niveau entre les modèles de différentes disciplines;
- Capacité de connaître l'impact direct des modifications de l'architecture sur les modèles dans toutes les disciplines.

Ce nouveau langage offre ainsi des perspectives pour une compatibilité interdisciplinaire des modèles de l'ingénierie système dans CATIA Systèmes V6.

Après avoir modélisé le système, la compilation optimisée des modèles vers le formalisme de bas-niveau Systèmes de Transitions Gardées rend possible des traitements automatiques.

Dans la phase de design et de conception du produit, la compilation vers les arbres de défaillance pour traiter des systèmes statiques permet de réaliser les calculs des coupes minimales, des probabilités d'occurrence des événements, ainsi que des calculs sur la fiabilité des systèmes; la génération des séquences et la simulation stochastique permettent de faire des analyses des systèmes dynamiques; le simulateur pas-à-pas permet de rejouer des scénarios et de valider le comportement du système. Il est, entre autres, possible d'utiliser la nouvelle version d'AltaRica pour modéliser des arbres de défaillance dynamiques grâce à la bibliothèque de composants décrite dans [10] et ainsi profiter de son ensemble d'outils de traitement automatique pour faire des calculs.

En opérations, les modèles AltaRica nouvelle génération peuvent être utilisés pour calculer la disponibilité des systèmes, pour simuler les interventions de maintenance [9] et les planifier, ou même réaliser la gestion du système.

Conclusion

Dans cet article nous avons présenté la nouvelle version du langage AltaRica qui a conservé tous les avantages de la version précédente. Elle permet donc de modéliser les systèmes hiérarchiques, de décrire le comportement événementiel et conserve la simplicité de syntaxe, ainsi que le caractère formel.

Son orientation objet permet d'améliorer les capacités de réutilisation et de capitalisation des connaissances du langage AltaRica. Etant donné que la nouvelle version est basée sur S2ML, sa compatibilité avec les autres langages de l'ingénierie système en termes de structuration de modèles, par exemple avec Modelica, est assurée nativement. Ceci permet désormais de travailler sur la compatibilité interdisciplinaire des modèles de l'ingénierie système dans CATIA Systèmes V6 et offre de nouvelles possibilités pour la traçabilité entre les modèles et leur adaptation aux changements d'architecture.

La syntaxe du langage a aussi évolué pour pouvoir intégrer plus naturellement les données fiabilistes (stochastiques) aux modèles et les valeurs initiales des variables. Les capacités d'expression de la nouvelle version permettent d'exprimer plus simplement les différents types de synchronisations.

Grâce au nouveau modèle d'exécution, basé sur le formalisme des Systèmes de Transitions Gardées (GTS), la nouvelle version du langage AltaRica permet de traiter les réseaux bouclés.

La nouvelle version du langage offre la possibilité de créer des modèles de haut niveau très proches de l'architecture fonctionnelle (ou logique) du système. Ils peuvent être compilés en formalisme mathématique de bas niveau – les Systèmes de Transitions Gardées. Cette compilation permet de générer automatiquement des arbres de défaillance à partir des modèles de haut niveau, faire des simulations stochastiques et générer des séquences.

Références

[1] B. Perrot, T. Prosvirnova, A. Rauzy, J. P. Sahut d'Izarn, 2009, Towards a Modelica-Flavored Event-Centric Language, in publication interne de Dassault Systèmes.

[2] A. Rauzy, 2008, Guarded transition systems: A new states/events formalism for reliability studies, in Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability.

[3] B. Perrot, A. Rauzy, E. Tillaux, 2008, S2ML: a Box n' Ball Calculus, in Dassault Systèmes, publication interne de Dassault Systèmes.

[4] A. Arnold, A. Griffault, G. Point, A. Rauzy, 2000, The AltaRica Language and its Semantics. Fundamenta Informaticae, 34:109–124.

[5] 2010, Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling – Language Specification http://www.modelica.org

[6] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, 1991, The synchronous dataflow programming language Lustre. Proceedings of the IEEE, 79(9):1305-1320.

[7] A. Rauzy, 2002, Modes Automata and their compilation into Fault Trees, in Reliability Engineering and System Safety, 78:1-12.

[8] ftp://ftp.software.ibm.com/software/applications/plm/resources/PLB03008-USEN-00.pdf, www.3ds.com

[9] B. Perrot, T. Prosvirnova, A. Rauzy, J. P. Sahut d'Izarn, 2010, Expérience de couplage de modèles AltaRica avec des interfaces métiers, actes du congrès Lambda Mu 17, La Rochelle, octobre 2010.

[10] B. Perrot, T. Prosvirnova, A. Rauzy, J. P. Sahut d'Izarn, 2010, Arbres de défaillance dynamiques : une bibliothèque pour la nouvelle génération d'AltaRica , actes du congrès Lambda Mu 17, La Rochelle, octobre 2010.