# LES CONSTRUCTIONS STRUCTURELLES DE ALTARICA 3.0

# THE STRUCTURAL CONSTRUCTIONS OF ALTARICA 3.0

PROSVIRNOVA Tatiana
LIX, Ecole Polytechnique
Bât. Alain Turing
Route de Saclay
91120 Palaiseau
+33 (0)1 77 57 80 91
prosvirnova@lix.polytechnique.fr

RAUZY Antoine
Chaire Blériot-Fabre
LGI – Ecole Centrale de Paris
Grande voie des vignes
92295 Châtenay-Malabry
+33 (0)1 41 13 10 25
Antoine.Rauzy@ecp.fr

**Résumé :**

Le langage AltaRica est un langage de modélisation de haut-niveau pour la sûreté de fonctionnement. La nouvelle version du langage, AltaRica 3.0, améliore la version précédente selon deux axes : d'un côté son modèle d'exécution est désormais basé sur le formalisme des Systèmes de Transitions Gardées, de l'autre côté un nouveau paradigme de structuration des modèles a été adopté : la modélisation orientée prototypes. L'objectif de cette communication est de présenter à l'aide d'exemples et de justifier les nouvelles constructions structurelles du langage AltaRica 3.0.

**Summary:**

AltaRica is a high level modeling language dedicated to safety analyses. The new version of the language, AltaRica 3.0, improves the previous version along two axes: first, the execution model is now based on the Guarded Transition Systems formalism; second, a new paradigm to structure models has been chosen, namely prototype orientation. The aim of this communication is to present by means of examples and to discuss the rational for the new structural constructs introduced in AltaRica 3.0.

## Introduction

La nouvelle version du langage AltaRica, AltaRica 3.0 (Prosvirnova, et al., 2013), met en œuvre de nouveaux mécanismes de structuration des modèles s'inspirant du paradigme de programmation-modélisation orientée prototypes (Noble, Taivalsaari, & Moore, 1999). L'objectif de cette communication est de présenter à l'aide d'exemples et de justifier ces nouvelles constructions structurelles.

La complexité croissante des systèmes industriels va de pair avec une complexité croissante des modèles de ces systèmes. Pour maîtriser cette complexité, il faut pouvoir structurer les modèles. En 1976, Niklaus Wirth, un des pionniers de l'informatique et le créateur des langages de programmation Pascal et Modula-2, a publié un livre qui a fait date et dont le titre exprimait la nécessité de structurer les programmes (Wirth, 1976) :

*« Programs = Algorithms + Data Structures »*

Comme les programmes, les modèles sont des artefacts. La formule de Wirth peut leur être appliquée, une fois adaptée. Les langages de modélisation de haut-niveau utilisés dans les différents domaines de l'ingénierie résultent en effet tous de la combinaison :

- d'un domaine mathématique (par exemple, systèmes d'équations différentielles et algébriques pour le langage Modelica (Fritzson, 2003), automates de Mealy pour le langage Lustre (Halbwachs, Caspi, Raymond, & Pilaud, 1991), Systèmes de Transitions Gardées pour AltaRica 3.0 (Prosvirnova, et al., 2013));
- d'un paradigme de structuration, c'est-à-dire d'un ensemble de constructions permettant d'organiser les modèles.

Le choix d'un paradigme de structuration est intimement lié à la méthode de modélisation mise en œuvre. En effet, implicitement au moins, tout formalisme de modélisation appuie et définit une telle méthode. De fait, on en observe deux grandes familles :

- Les méthodes qui construisent le modèle via une analyse descendante (« top-down ») du système. Parmi les formalismes supportant cette approche, on trouve typiquement SysML (Friedenthal, Moore, & Steiner, 2011), les BPMN (White & Miers, 2008), les StateCharts (Harel, 1987), mais aussi nos bons vieux arbres de défaillances, blocs diagrammes de fiabilité et réseaux de Petri stochastiques (Ajmone-Marsan, Balbo, Conte, Donatelli, & Franceschinis, 1995).
- Les méthodes qui construisent le modèle par assemblage (« bottom-up ») de composants (éventuellement regroupés dans des bibliothèques de composants sur étagère). Parmi les formalismes supportant cette approche, on trouve typiquement Modelica et Lustre déjà cités.

Cette distinction semble aller au-delà des problématiques de modélisation. Elle a par exemple été conceptualisée par Hatchuel dans le cadre de sa théorie C-K de l'innovation (Hatchuel & Weill, 2009). Hatchuel interprète cette distinction en termes de connaissance en train de s'élaborer et connaissance stabilisée. Dans le cadre des formalismes d'ingénierie système, il semble

qu'elle soit aussi et même plutôt liée au niveau d'abstraction auquel on considère le système : les méthodes « top-down » sont privilégiées quand le système est analysé globalement alors que les méthodes « bottom-up » sont privilégiées à un niveau d'abstraction moindre. Quoi qu'il en soit, à ces méthodes correspondent des mécanismes de structuration différents des modèles : paradigme orienté prototypes pour les méthodes « top-down » et paradigme orienté objets pour les méthodes « bottom-up ». Nous expliciterons ces concepts dans la suite. Le lecteur intéressé par des développements plus théoriques sur les langages de programmations peut se référer aux travaux de Abadi et Cardelli (Abadi & Cardelli, 1998).

De ce point de vue, les modélisations de sûreté de fonctionnement, celles que vise le langage AltaRica, sont ambivalentes : parce qu'elles considèrent les systèmes à un haut niveau d'abstraction, elles ressortent naturellement du paradigme orienté prototypes ; mais parce qu'elles réutilisent les composants (ne serait-ce que pour prendre en compte les redondances), elles ressortent aussi du paradigme orienté objets. Pour AltaRica 3.0, nous avons donc choisit de ne pas choisir et nous avons introduit deux types de composants : les blocs (qui sont des prototypes) et les classes. Ces deux constructions structurelles donnent au langage une grande richesse, que nous allons illustrer ici à l'aide d'exemples.

La contribution de cette communication est donc double : d'une part, nous y introduisons le langage AltaRica 3.0 avec un focus particulier sur les mécanismes de structuration des modèles ; d'autre part nous y ouvrons une discussion plus générale sur ces derniers, discussion qui s'applique au-delà de la sûreté de fonctionnement à tous les langages de modélisation de systèmes.

Le reste de l'article est organisé comme suit : nous introduisons dans la prochaine section les concepts de prototype et de classe. Nous montrons comment ils sont mis en œuvre dans AltaRica 3.0 et en quoi ils sont distincts et utiles. Nous présentons ensuite les mécanismes permettant à différentes branches d'une structure hiérarchique d'être en relation et de partager des composants. Finalement, nous consacrons une longue section à la discussion de divers points touchant à la structure des modèles.

## Prototypes et classes

### Pourquoi les deux concepts de prototype et de classe sont distincts et utiles ?

Supposons pour commencer que nous souhaitions étudier le système de pompage représenté Figure 1. Ce système est composé d'un réservoir, d'un réacteur et de deux trains eux-mêmes composés d'une pompe et d'une vanne.

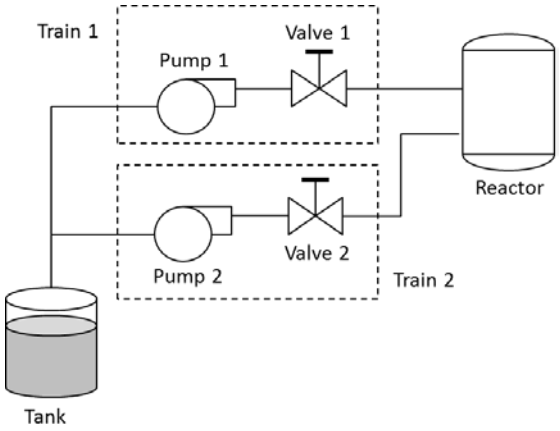


**Figure 1.** Un système de pompage

Nous avons donc à représenter cette structure hiérarchique. La Figure 2 donne la partie structurelle d'un code AltaRica permettant de faire cela. Ce modèle (partiel) est constitué d'une hiérarchie de blocs, c'est-à-dire de prototypes, imbriqués. Chaque bloc est unique. Le comportement du composant correspondant doit donc être défini individuellement.

Le comportement de la pompe n°1 pourrait par exemple être défini par le système de transitions gardées représenté Figure 3. Cet automate a deux états (« WORKING » et « FAILED »). Il change d'état sous l'effet d'un événement (« failure » ou « repair »). Il a d'autre part une entrée et une sortie booléennes (« inflow » et « outflow »). Dans l'état de marche, il y a un flux de sortie quand il y a un flux d'entrée. Dans l'état de panne, rien ne sort de la pompe. Le code AltaRica correspondant est donné Figure 4. Il est suffisamment explicite pour que nous ne le détaillions pas davantage ici.

Le comportement de la pompe n°2 devrait être défini de façon similaire. Cependant, si les deux pompes sont identiques, ce qui est probablement le cas, dupliquer les modèles est à la fois fastidieux et sujet à erreurs (de copier-coller, de mise à jour…). L'idée est donc de définir un composant « Pump » générique qui pourra ensuite être instancié à plusieurs endroits dans le modèle. La définition d'un tel composant générique est réalisée via une classe, comme illustré par le code donné Figure 5 (où le même principe a été appliqué aux vannes).

Certains langages de modélisation, comme Modelica, ne mettent en œuvre que la notion de classe (c'était aussi le cas de la version précédente de AltaRica, appelée AltaRica Data-Flow, voir par exemple (Boiteau, Dutuit, Rauzy, & Signoret, 2006)). D'autres, comme SysML, que la notion de prototype. AltaRica 3.0 met en œuvre les deux, pour plusieurs raisons. Si l'intérêt de la notion de classe est clair (et vient d'être illustré), il est important de comprendre pourquoi la notion de prototype est tout aussi nécessaire.

```
block PumpingSystem
  block Tank
    ...
  end
  block Train1
    block Pump1
      ...
    end
    block Valve1
      ...
    end
  end
  block Train2
    block Pump1
      ...
    end
    block Valve1
      ...
    end
  end
  block Reactor
    ...
  end
end
```

**Figure 2.** Partie structurelle d'un code AltaRica pour le système de pompage

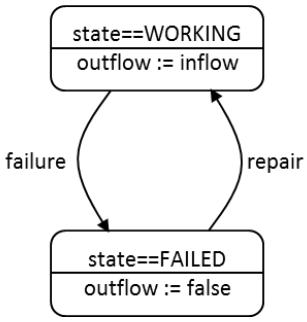


**Figure 3.** Un système de transitions gardées représentant le comportement de la pompe n°1

```
domain PumpState {WORKING, FAILED}

block Pump1
  PumpState state (init = WORKING);
  Boolean inflow, outflow (reset = false);
  event failure, repair;
transition
  failure: state==WORKING -> state := FAILED;
  repair: state==FAILED -> state := WORKING;
assertion
  outflow := state==WORKING and inflow;
end
```

**Figure 4.** Le code AltaRica pour le système de transitions gardées représentant le comportement de la pompe n°1

```
class Pump
  // same code here as on Figure 4
end

class Valve
  // behavior of valves
end

block PumpingSystem
  ...
  block Train1
    Pump P;
    Valve V;
    ...
  end
  block Train2
    Pump P;
    Valve V;
    ...
  end
  ...
end
```

**Figure 5.** Code AltaRica 3.0 pour le système de pompage avec les comportements de pompes et des vannes définis via des classes

Tout d'abord, dans un modèle de sûreté de fonctionnement, de nombreux composants sont uniques. Dans un système comme celui de notre exemple, c'est le cas bien sûr du système lui-même, mais aussi probablement des trains. En effet, même si les deux trains sont similaires, il y a tout à parier qu'il existe entre eux de légères différences (de raccordement, de localisation…) qui nécessitent de les modéliser individuellement. De plus, les trains sont beaucoup plus des entités organisationnelles ou fonctionnelles que des composants physiques. Même si formellement il est possible de décrire le comportement de tels composants uniques via des classes, il est de loin préférable de les distinguer des composants génériques. On obtient ainsi une séparation claire entre la connaissance en train de s'élaborer (le « bac à sable ») et la connaissance stabilisée (les composants génériques sur étagères).

Dans un langage ne proposant que des classes, une représentation et l'édition graphique du système via un schéma de type « *Process & Instrumentation Diagram* » tel que celui de la Figure 1 est impossible ou en tout cas extrêmement dangereux. En effet, ce schéma a trois niveaux de profondeur : le système, les trains à l'intérieur du système, les pompes et les vannes à l'intérieur des trains. Or, autoriser à modifier une classe (comme « Train1 ») via son instance dans une autre classe (ici « PumpingSystem ») est une aberration conceptuelle, en particulier parce que cela peut avoir des impacts non maîtrisés sur d'autres instances de cette classe. Les langages ne proposant que le mécanisme de classes pour structurer les modèles ne supportent donc que des représentations graphiques « plates », c'est-à-dire à deux niveaux de profondeur. On pourra nous objecter ici que, l'analyste sachant que cette classe a une unique instance, il peut se permettre de tels écarts. Outre le fait que, d'un point de vue méthodologique, ce n'est jamais une bonne idée d'enfreindre les règles, une classe avec une instance unique est un prototype et doit être désignée en tant que tel.

Il existe une troisième distinction importante entre les classes et les prototypes que nous discuterons dans la section « Partage de composants ».

**Agrégation et héritage**

Dans notre exemple, le bloc « Train1 » contient une instance de la classe « Pump ». On dit qu'il agrège cette instance. De la même façon, le bloc « PumpingSystem » agrège le bloc « Train1 ». Un bloc ou une classe peuvent donc agréger des blocs et des instances de classes, permettant ainsi de construire des modèles hiérarchiques.

Il existe cependant un autre type de relation hiérarchique, celle de spécialisation. Dans notre exemple, une pompe (et sans doute aussi une vanne) est un composant réparable. Le modèle de pompe doit donc spécialiser le modèle de composant réparable, mais bien sûr pas l'agréger. Le mécanisme d'héritage (que l'on retrouve dans tous les langages de programmation et de modélisation orientés objets) permet de rendre compte de ce type de relation. Sa mise en œuvre en AltaRica 3.0 via la clause « extends » est illustrée Figure 6. La classe « Pump » étend donc la classe « RepairableComponent » en lui adjoignant des variables de flux d'entrée et de sortie et l'assertion correspondante.

L'héritage multiple est possible en AltaRica 3.0, même si peu recommandé (comme dans tous les langages orientés objets).

D'un point de vue technique, il y a moins de différence entre agrégation et héritage qu'il ne peut y paraître à première vue. Les composants d'une instance de classe agrégée sont ajoutés au bloc ou à la classe agrégeant en les préfixant par le nom de l'instance. Dans le cas d'une classe héritée, ils sont ajoutés sans préfixe. Ainsi, il est possible de référer dans le bloc « PumpingSystem » l'état de la pompe du train n°1 via l'identificateur « Train1.P.state ». Les préfixes « Train1 » et « P » ont été ajoutés car ils résultent d'agrégations. En revanche, la classe « Pump » héritant de la classe « RepairableComponent », la variable d'état « state » de la seconde peut être référencée telle quelle dans la première.

```
domain RepairableComponentState {WORKING, FAILED}

class RepairableComponent
  RepairableComponentState state (init = WORKING);
  event failure, repair;
transition
  failure: state==WORKING -> state := FAILED;
  repair: state==FAILED -> state := WORKING;
end

class Pump
  extends RepairableComponent;
  Boolean inflow, outflow (reset = false);
assertion
  outflow := state==WORKING and inflow;
end
```

**Figure 6.** Illustration de la mise en œuvre du mécanisme d'héritage via la clause « extends »

**Relations entre composants**

A côté des liens « verticaux » entre composants (agrégation, héritage), il y a leurs liens « horizontaux », c'est-à-dire les moyens par lesquels ils interagissent. En AltaRica, ces moyens sont au nombre de deux : la propagation de valeurs via les variables de flux et les assertions et les synchronisations d'événements. Nous présentons brièvement ces deux notions.

AltaRica 3.0 comporte deux types de variables : les variables dites d'état et les variables dites de flux.

Les variables d'état, comme leur nom l'indique, servent à modéliser l'état des composants. Elles sont repérées par l'attribut « init ». La variable « state » du composant réparable présenté Figure 6 est une variable d'état. Les variables d'état sont initialisées une fois pour toutes. Leur valeur n'est modifiée que par le tir d'une transition (via l'action après la flèche).

Les variables de flux servent à modéliser les propagations d'information, de matières, d'énergie dans le réseau de composants. Elles sont repérées par l'attribut « reset ». La variable « inflow » des pompes est ainsi une variable de flux. Les variables de flux sont recalculées (au moins en théorie) après chaque tir de transition via les assertions. Dans notre exemple, le flux d'entrée du train n°1 est connecté via deux équations (appelée assertions) d'une part au flux de sortie du réservoir, d'autre part au flux d'entrée de la pompe n°1.

Il est important de noter que la direction des flux en AltaRica 3.0 est déterminée à l'exécution, c'est-à-dire qu'elle peut dépendre de l'état des composants (au contraire par exemple de Modelica où la direction des flux est déterminée une fois pour toutes à la compilation). C'est la raison pour laquelle AltaRica 3.0 permet de traiter des systèmes bouclés.

L'autre mécanisme permettant aux différents composants d'un modèle d'interagir est la synchronisation d'événements. AltaRica 3.0 offre un mécanisme de synchronisation générique et très puissant. Supposons par exemple que nous souhaitions introduire une défaillance de cause commune entre les deux pompes. Il nous suffira pour ce faire d'introduire un nouvel événement « CCFPumps » ainsi qu'une transition au niveau du bloc « PumpingSystem » qui est l'ancêtre commun des deux pompes. Cette transition prendra la forme suivante :

```
CCFPumps : ?Train1.P.failure & ?Train2.P.failure ;
```

Cette transition est tirable si au moins une des transitions de panne des pompes l'est (c'est le sens des modalités « ? » préfixant les deux événements locaux de panne).

## Partage de composants

Les constructions structurelles proposées dans la section précédente permettent de construire des modèles hiérarchiques en arbre : si un bloc ou une classe peut contenir plusieurs blocs ou instances de classes (par agrégation ou héritage), en revanche chaque bloc ou instance de classes n'est contenu que dans un seul bloc ou instance de classe. Ainsi, la pompe n°1 de notre exemple n'est contenu que de dans le train n°1 qui n'est lui-même contenu que dans le système de pompage.

Si l'on considère l'architecture physique du système et uniquement son architecture physique, une telle décomposition hiérarchique peut suffire. Si l'on veut décrire aussi son architecture fonctionnelle, cela pose des problèmes délicats à surmonter. Ce problème n'apparait pas dans les langages de simulation physique, comme Matlab/Simulink ou Modelica, précisément parce qu'ils ne sont destinés qu'à la simulation multi-physique (ce qui est déjà beaucoup). Les études de sûreté de fonctionnement se placent à un niveau d'abstraction plus élevé. Elles prennent en compte les aspects physiques et les aspects fonctionnels d'un système. En pratique, l'événement sommet d'un arbre de défaillance est pratiquement toujours fonctionnel, du type « perte de la capacité à faire ceci ou cela ». Dans notre exemple, ce serait typiquement : « perte de l'alimentation en liquide du réacteur ». Les événements de base de l'arbre sont en revanche pratiquement toujours des défaillances de composants physiques (si l'on accepte de considérer les opérateurs comme des composants physiques). Une conséquence naturelle de ce mélange est que la défaillance d'un composant peut impacter différentes fonctions. Les arbres de défaillance n'ont d'arbre que le nom : leur structure est en fait plutôt un graphe orienté acyclique (« Directed Acyclic Graph »). Un événement de base ou une porte peuvent avoir plusieurs parents dans l'arbre.

Dans le domaine de l'architecture système, cette question a été discutée via la notion de chaîne fonctionnelle (voir par exemple (Voirin, 2008)). L'idée est que l'architecture du système (fonctionnelle et physique) n'est jamais édité globalement, via des chaînes de fonctions ou de composants participant à la réalisation d'une des missions du système. Non seulement ces

chaînes peuvent partager des composants, mais elles ne recoupent en général pas la décomposition physique du système. Un composant peut contribuer à plusieurs chaînes et chaque chaîne requiert plusieurs composants.

La question est donc comment créer des modèles hiérarchiques dont la structure ne soit pas un arbre mais un graphe orienté acyclique ou, pour le dire autrement, comment différentes branches de la hiérarchie peuvent-elles partager des composants.

Dans notre exemple, on pourrait avoir un dispositif d'arrêt d'urgence en cas de surpression dans le réservoir, avec les capteurs, le contrôleur et les interrupteurs (pour stopper les pompes) correspondants. Un des intérêts des modèles de haut niveau, comme ceux réalisés avec AltaRica, est qu'ils permettent d'étudier plusieurs événements redoutés à partir du même modèle. Un de ces événements redoutés pourrait être ici « perte du dispositif d'arrêt d'urgence ». Ce dispositif incorporerait les deux pompes qui de ce fait se trouveraient faire à la fois partie des trains et du dispositif.

AltaRica 3.0 permet à des blocs appartenant à différentes branches d'un modèle hiérarchique via la clause « embeds ». L'utilisation de cette clause est illustrée sur notre exemple par le code (partiel) donné Figure 7. Le dispositif d'arrêt d'urgence est modélisé par un bloc (dans ce cas au même niveau hiérarchique que les deux trains). Il embarque les pompes en les renommant. « Pump1 » dans le système d'arrêt d'urgence dénote bien le même composant que « P » dans le train n°1. Ce modèle nécessiterait bien sûr un comportement un peu plus élaboré pour les pompes (permettant notamment de les arrêter et de les démarrer).

```
block PumpingSystem
  ...
  block Train1
    Pump P;
    ...
  end
  block Train2
    Pump P;
    ...
  end
  block EmergencyShutdownSystem
    embeds Train1.P as Pump1;
    embeds Train2.P as Pump2;
    ...
  end
  ...
end
```

**Figure 7.** Illustration de l'utilisation de la clause « embeds » qui permet de partager des composants

Il est important de comprendre ici que seule le paradigme orienté prototypes permet ce partage de composants. En effet, une classe définit un composant sur étagère. Le concepteur de la classe ne peut pas présumer de l'utilisation qui en sera faite. Il ne peut donc y avoir au sein d'une classe des références à des objets définis en dehors de la classe. Un prototype au contraire est toujours localisé. Il peut donc faire référence à des objets situés dans le même modèle, c'est-à-dire soit dans le même bloc de plus haut niveau (ici le bloc « PumpingSystem ») soit dans la classe dans laquelle il apparait.

Le mécanisme de partage de composants proposé ci-dessus est, à notre connaissance, original. Nous sommes convaincus, sur la base de diverses modélisations, qu'il représente un vrai avantage en termes de clarté et de simplicité des modèles.

## Discussion

Les deux sections précédentes nous ont permis de présenter les constructions structurelles de AltaRica 3.0. L'objectif de la présente section est de discuter plusieurs aspects importants de la modélisation de systèmes complexes qui font appel à des constructions structurelles et notamment de recenser un certain nombre de questions ouvertes.

### Réutilisation : bibliothèques de composants ou bibliothèques de schémas ?

Une des clés de la productivité du processus de modélisation est la réutilisation de composants, voire de modèles ou autrement dit la capitalisation de connaissances. Il existe en fait deux types de réutilisation assez distincts, la réutilisation de composants et la réutilisation de schémas qui correspondent aux forces respectives des paradigmes orienté-objets et orientés prototypes.

Réutiliser des composants veut simplement dire que l'on définit des bibliothèques de composants qui sont destinés à être intégrés tels quels dans les modèles. Cette notion de bibliothèque de composants vient directement des langages de programmation avec de très imposantes bibliothèques comme Qt en C++ (voir par exemple (Blanchette & Summerfield, 2008)). Elle fait le succès de langages de modélisation comme Matlab/Simulink et Modelica pour lesquels des dizaines de bibliothèques dédiées à tel ou tel domaine sont disponibles. La réutilisation de composants est au cœur du paradigme orienté objets.

La réutilisation de schémas (« *patterns* » en anglais) procède d'un autre principe. L'idée est plutôt de partir d'un code existant, de le dupliquer et de l'adapter à ses besoins. Les fameux « *design patterns* » sont l'archétype de cette approche ( (Gamma, Helm, Johnson, & Vlissides, 1994). Elle correspond naturellement au paradigme orienté prototypes.

L'expérience montre que pour le type de modélisation que vise AltaRica, c'est plutôt la réutilisation de schémas qui est utile (voir par exemple (Kehren, et al., 2004)), même si la définition de bibliothèques de composants peut s'avérer utile. Un des objectifs du projet Open AltaRica (qui s'appuie sur le langage AltaRica 3.0) est précisément de concevoir de telles bibliothèques.

**Modèles paramétriques et scripts**

On a souvent besoin d'adapter un composant à un besoin particulier. Par exemple, dans le modèle de composant réparable défini Figure 6, on peut vouloir associer un taux de défaillance et un taux de réparation au composant (qui s'appelleront comme il se doit lambda et mu). Ces taux seront fixés à l'instanciation du composant. AltaRica 3.0 offre la notion de paramètre pour ce faire (directement inspirée de celle de Modelica). Un paramètre a une valeur par défaut qui peut être modifiée à l'instanciation de la classe. Dans le cas des pompes de notre exemple, cela donnerait le code de la Figure 8.

```
class RepairableComponent
  ...
  event failure (delay = exponential(lambda));
  parameter Real lambda = 1.0e-3;
  ...
end

block PumpingSystem
  ...
  block Train1
    Pump P(lambda = 2.34e-5)
    ...
  end
  ...
end
```

**Figure 8.** Illustration de l'utilisation de paramètres

Un autre besoin de généricité, auquel ne répond pas la notion de paramètre, porte sur le nombre de flux d'entrée (ou de sortie) d'un composant. Imaginons par exemple que l'on ait besoin de composants réparables avec un nombre variable d'entrées booléennes et une sortie booléenne qui est égale à la disjonction des entrées si le composant fonctionne et faux s'il est en panne. Il est évidemment préférable d'éviter de définir un tel composant avec deux entrées, un autre avec trois et ainsi de suite. Idéalement, on aimerait définir un composant générique avec un nombre variables d'entrées et fixer ce nombre à l'instanciation. C'est ce qui a été fait dans le formalisme Figaro où des quantificateurs ont été introduits pour concevoir ce type de composants génériques (voir par exemple (Bouissou, Bouhadana, Bannelier, & Villatte, 1991)). Le problème c'est que l'introduction de mécanismes ad-hoc tend à alourdir le développement des outils de traitement pour un gain finalement assez faible. En effet, il ne faut pas non plus surestimer le besoin en composants génériques, ni les inconvénients d'avoir à dupliquer un peu de code.

Une solution plus générale est d'utiliser un langage de script, de préférence intégré au langage hôte, pour générer automatiquement les modèles. C'est plus ou moins ce qui est fait dans Modelica avec les tableaux. C'est aussi la conclusion à laquelle nous étions arrivés lors de notre comparaison entre AltaRica et PEPAnets pour la modélisation de composants mobiles (Kloul, Prosvirnova, & Rauzy, 2013). Aucune décision n'a encore été prise à ce sujet pour AltaRica 3.0.

**Polymorphisme**

En informatique, le polymorphisme désigne la capacité d'un objet (typiquement une liste) à contenir des objets de types différents. Des langages comme Scheme (Abelson, Sussman, & Sussman, 1996) sont nativement polymorphes puisqu'ils ne sont pas typés. Certains informaticiens leur reprochent précisément cette absence de typage. Quoi qu'il en soit, ce n'est pas exactement à cette définition du polymorphisme que nous faisons référence, mais plutôt à la capacité de substituer un composant à un autre dans un modèle, permettant à ce dernier d'être « polymorphe » puisque pouvant contenir des composants de différents types. Pour comprendre de quoi il s'agit, le mieux est de prendre un exemple.

Supposons que nous voulions spécialiser le modèle de nos vannes pour, par exemple, prendre en compte le fait que ce sont des vannes opérées avec un moteur. Nous pourrions faire cela sans problème grâce au mécanisme d'héritage. Mais supposons maintenant que, plutôt que d'utiliser des blocs, nous ayons défini une classe « Train » pour modéliser nos trains. Comment ferions-nous alors pour spécialiser notre classe « Train » en une classe « TrainWithMotorOperatedValve », c'est-à-dire pour substituer le modèle générique de vanne utilisé dans la classe « Train » par le modèle spécifique « ModelOperatedValve » dérivé du modèle générique de vanne « Valve » ? Plus embêtant encore, supposons que nous voulions remplacer la vanne par un mécanisme de dérivation. S'il est à peu près clair qu'une vanne opérée par un moteur possède toute les propriétés d'une vanne générique, il n'en va pas de même pour un mécanisme de dérivation. Mais alors comment garantir que les autres composants du système qui interagissent normalement avec la vanne, vont pouvoir interagir de la même façon avec le mécanisme de dérivation ?

Deux approches ont été proposées pour traiter ce problème. Tout d'abord, on peut considérer chaque classe comme une fonction prenant des arguments. Autrement dit, on ne définit pas une classe complète, mais plutôt comme un composant auquel il faut connecter d'autres composants pour obtenir la classe complète. Cette méthode est séduisante d'un point de vue mathématique (elle a par exemple été explorée par Milner (Milner, The Space and Motion of Communicating Agents, 2009)). Elle a cependant à notre sens deux inconvénients majeurs : d'une part les modèles deviennent difficilement lisibles et donc maintenables, d'autre part, elle ne permet pas vraiment de définir des composants par défaut ou standards. L'autre solution, adoptée par exemple dans Modelica, consiste à proposer une construction permettant de substituer un composant par un composant d'une classe dérivée. Dans Modelica, seulement les composants explicitement déclarés remplaçables peuvent être ainsi substitués, sans doute pour améliorer l'efficacité du compilateur, et ils ne peuvent être remplacés que par des instances d'une classe dérivée de leur propre classe.

A ce jour, nous n'avons pas encore pris de décision définitive sur ce sujet pour AltaRica 3.0. Comme précédemment, l'utilisation de script est peut être suffisante et préférable à l'introduction de mécanismes ad-hoc, complexes à mettre en œuvre et à maintenir.

**Représentations graphiques**

Aujourd'hui, pour être adopté largement, un outil de modélisation doit avoir une interface graphique. Les ateliers intégrés de modélisation mettant en œuvre des langages comme AltaRica n'échappent pas à cette règle. Il faut cependant avoir conscience qu'une représentation graphique ne peut pas être en correspondance biunivoque avec le modèle sauf à réduire considérablement la puissance d'expression du langage de modélisation ou à complexifier considérablement les représentations graphiques au point de leur faire perdre leur intérêt, à savoir être des outils de communication. La confusion entre le modèle et sa ou ses représentations graphiques est, à notre humble avis, une faiblesse constitutive d'approches comme le « *Eclipse Modeling Framework* » (Steinberg, Budinsky, Paternostro, & Merks, 2008), quel que soit leur intérêt par ailleurs. Il faut accepter que les représentations graphiques ne soient que des vues partielles du modèle (AltaRica ou autre). Dans le cadre des réflexions autour de la spécification du langage AltaRica 3.0, nous cherchons à pousser jusqu'au bout ce raisonnement.

Tout d'abord, nous pensons qu'il faut distinguer l'édition de l'animation graphique du modèle. Nous avons récemment montré que cette dernière peut avantageusement être mise en œuvre grâce à un petit langage graphique et le simulateur correspondant (Prosvirnova, Batteux, Maarouf, & Rauzy, 2013). L'idée est que le simulateur graphique et le simulateur AltaRica communiquent via l'échange de valeur de variables. Les animations graphiques sont donc totalement indépendantes du modèle AltaRica et le langage d'animation graphique peut être utilisé par d'autres outils que les outils AltaRica. Le bénéfice immédiat est de ne pas « polluer » le modèle comportemental avec des considérations graphiques et de pouvoir donner plusieurs vues graphiques animées d'un même modèle.

Reste la question de l'édition graphique des modèles. Il faut tout d'abord remarquer que les ateliers AltaRica ne représentent graphiquement que la partie structurelle des modèles, avec des schémas de type « Process & Instrumentation Diagram » comme celui représenté Figure 1. La même remarque s'applique d'ailleurs aux ateliers Modelica ou à Simulink. La partie comportementale, en l'occurrence les systèmes de transitions gardées, est éditée textuellement. Il serait sans doute intéressant de générer automatiquement les automates comme celui représenté Figure 3 à partir du code donné Figure 4. Cela n'a pas encore été fait, probablement en raison du coût de développement d'un tel outil et du bénéfice relativement faible attendu, dans la mesure où le code AltaRica est suffisamment explicite.

En ce qui concerne la partie structurelle, plusieurs remarques peuvent être faites qui s'inspirent beaucoup des réflexions de Furhmann sur le sujet (dans un autre contexte que l'édition de modèles AltaRica (Fuhrmann, 2011)). Tout d'abord, il serait bien sûr là aussi souhaitable de générer automatiquement la représentation graphique du modèle à partir du code, comme cela se fait par exemple avantageusement pour les arbres de défaillance. Une telle génération automatique est tout à fait possible pour des représentations arborescentes de la structure du modèle, comme celle présentée Figure 9.

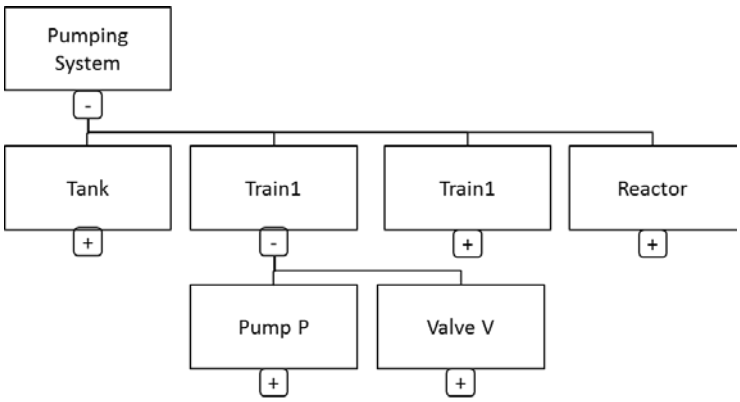


**Figure 9.** Une représentation arborescence de la structure du modèle du système de pompage

En revanche, il semble difficile de générer automatiquement des schémas de type « Process & Instrumentation Diagram ». De plus, comme le fait remarquer fort justement Furhmann, une génération automatique risque de faire changer du tout au tout la représentation graphique si l'on ajoute ou retire un élément au modèle, ce qui peut être troublant pour l'analyste. Il semble donc qu'il faut se résoudre à rendre les informations de représentation graphique persistantes. Le plus raisonnable est sans doute de définir un langage de feuilles de style graphiques qui serait à AltaRica (et aux autres langages de modélisation) ce que sont les « Cascading Style Sheets » à HTML. Les annotations Modelica, qui s'intègrent plutôt mal que bien au langage et qui alourdissent considérablement les modèles, ne sont à notre avis pas un exemple à suivre. Quoi qu'il en soit, ce travail reste à faire.

**Relations inter-modèles**

Un des problèmes majeurs de l'ingénierie système aujourd'hui est la synchronisation des dizaines de modèles conçus pour un même système. Comment, par exemple, s'assurer que les modèles d'architecture et les corpus d'exigences conçus par les architectes sont en phase avec les modèles d'analyse du risque conçus par les fiabilistes ? Cette synchronisation, qui aujourd'hui repose quasiment exclusivement sur des processus organisationnels, demande à être outillée et donc pensée en termes de langage (ne serait-ce qu'un langage de liens de traçabilité et d'impact). La question est dans quelle mesure ces liens inter-modèles doivent être intégrés aux langages de modélisation comme AltaRica 3.0 ? Cette question mérite d'être examinée avec attention.

Parmi les liens inter-modèles, il en est qui jouent un rôle particulier : les liens entre modèles du même type, typiquement entre modèles AltaRica. Dans le cadre d'une démarche de sûreté de fonctionnement « dirigée par les modèles », sous-entendu de

haut niveau, on imagine que le donneur d'ordre et ses sous-traitants seront amenés à concevoir des modèles (AltaRica dans notre cas). Le sous-traitant modélisera typiquement un composant ou un sous-système du système modélisé par le donneur d'ordre. Idéalement, ce dernier n'aurait qu'à brancher le modèle du composant dans son propre modèle. En pratique, cela risque de s'avérer délicat pour deux raisons : d'une part, le donneur d'ordre devra travailler sans attendre le modèle de son fournisseur. Il devra donc réaliser son propre modèle du composant. Ce dernier sera sans doute une version très simplifiée du modèle du fournisseur. Des désalignements entre les deux modèles sont donc à prévoir qu'il faudra savoir gérer. D'autre part, il ne sera probablement de toute façon pas possible d'intégrer le modèle détaillé du composant dans le modèle du système, en raison de la trop grande complexité des calculs que cela engendrerait. Tous les problèmes sous-jacents aux modélisations de sûreté de fonctionnement sont en effet difficiles au sens de la théorie de la complexité (Valiant, 1979). La mise en œuvre de langages de haut niveau ne change rien à cet état de fait. Notre sentiment est qu'il faut outiller la relation entre le modèle simplifié et le modèle complet du composant. L'interprétation abstraite (Cousot & Cousot, 1977) semble le bon cadre conceptuel pour cela. Reste encore à définir concrètement les outils pour mettre en œuvre ce cadre dans le cas particulier des modèles AltaRica et à déterminer si cela nécessite de faire évoluer les constructions structurelles du langage.

**Systèmes de systèmes**

Les constructions structurelles de AltaRica 3.0 sont adaptées pour modéliser des systèmes dont l'architecture ne change pas, ou pas significativement, au cours de la mission. Le nombre de composants et les liens entre les composants sont fixes. Il est bien sûr possible de simuler la création et la disparition de composants et l'activation ou la désactivation de liens, mais cela reste un pis-aller et cela n'est pas très ergonomique (voir par exemple l'article déjà cité (Kloul, Prosvirnova, & Rauzy, 2013)).

Prendre en compte la création et la destruction dynamique de composants fait faire un bond redoutable à la complexité des traitements. Certains problèmes, comme l'accessibilité d'un état, qui sont décidables même si difficiles dans le cadre d'architectures statiques, deviennent indécidables dans le cadre d'architectures dynamiques. Les formalismes de modélisation permettant la prise en compte d'architectures dynamiques, comme le pi-calcul (Milner, Communicating and Mobile Systems: The pi-calculus, 1999), restent très théoriques. Il reste qu'il serait sans doute intéressant de pouvoir s'attaquer à ce type d'architectures, notamment parce que ce sont celles que l'on rencontre dans les fameux systèmes de systèmes (une définition conceptuelle solide des systèmes de systèmes est peut-être précisément le fait d'avoir une architecture qui varie au cours de la mission). Nous sommes en tout cas convaincus qu'une réflexion conceptuelle et pragmatique doit être menée sur ce sujet, avec comme objectif d'intégrer des primitives de changement d'architecture dans des langages comme AltaRica 3.0.

## Conclusion

Dans cet article, nous avons présenté les constructions structurelles du langage AltaRica 3.0. Nous sommes convaincus que ces constructions étendent et améliorent significativement celles des versions précédentes du langage et par là la clarté des modèles. Le paradigme orienté prototypes nous parait le plus adapté pour le niveau d'abstraction auquel se placent les modèles de sûreté de fonctionnement.

Nous pensons que poser l'équation générale :

« Langage de modélisation = domaine mathématique + paradigme de structuration »

est une étape nécessaire pour l'ingénierie des systèmes. De ce point de vue, AltaRica 3.0 résulte de l'équation particulière :

« AltaRica 3.0 = systèmes de transitions gardées + modélisation orientée prototypes »

En fait, les mécanismes de structuration mis en œuvre dans AltaRica 3.0 peuvent être regroupés en un langage de structure que nous avons appelé S2ML, pour « *System Structure Modeling Language* ». Ce langage de structure pourrait être utilisé avec d'autres domaines mathématiques, des équations différentielles aux réseaux bayésiens en passant par les machines de Mealy.

L'étude des langages de modélisation en tant que tels n'en est qu'à ses débuts. Elle soulève de nombreuses questions scientifiques passionnantes et directement utiles pour l'industrie. Il faut penser la modélisation en termes de langage !

## Références


Abadi, M., & Cardelli, L. (1998). *A Theory of Objects.* New-York, USA: Springer-Verlag.

Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and Interpretation of Computer Programs.* Cambridge, MA 02142-1315, USA: MIT Press.

Ajmone-Marsan, M., Balbo, G., Conte, G., Donatelli, S., & Franceschinis, G. (1995). *Modelling with Generalized Stochastic Petri Nets.* Bognor Regis , West Sussex PO22 9SA, England: John Wiley and Sons.

Blanchette, J., & Summerfield, M. (2008, February). *C++ GUI Programming with Qt4.* Upper Saddle River, New Jersey, USA: Prentice Hall.

Boiteau, M., Dutuit, Y., Rauzy, A., & Signoret, J.-P. (2006, July). The AltaRica Data-Flow Language in Use: Assessment of Production Availability of a MultiStates System. *Reliability Engineering and System Safety, 91*(7), 747-755.

Bouissou, M., Bouhadana, H., Bannelier, M., & Villatte, N. (1991). Knowledge modelling and reliability processing: presentation of the FIGARO language and of associated tools. *Proceedings of SAFECOMP'91.* Trondheim.

Cousot, P., & Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (pp. 238-252). New York: ACM Press.

Friedenthal, S., Moore, A., & Steiner, R. (2011). *A Practical Guide to SysML: The Systems Modeling Language.* San Francisco, CA 94104, USA: Morgan Kaufmann. The MK/OMG Press.

Fritzson, P. (2003). *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1.* Hoboken, NJ 07030-5774, USA: Wiley-IEEE Press.

Fuhrmann, H. A. (2011). *On the Pragmatics of Graphical Modeling.* Norderstedt, Germany: Book on Demand.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994, October). *Design Patterns – Elements of Reusable Object-Oriented Software.* Boston, MA 02116, USA: Addison-Wesley.

Halbwachs, N., Caspi, P., Raymond, P., & Pilaud, D. (1991). The synchronous dataflow programming language Lustre. *Proceedings of the IEEE, 79*(9), 1305-1320.

Harel, D. (1987, June). Statecharts: a visual approach to complex systems. *Science of Computer Programming, 8*(3), 231-–274.

Hatchuel, A., & Weill, B. (2009). C-K design theory: an advanced formulation. *Research in Engineering Design, 19*(4), 181-192.

Kehren, C., Seguin, C., Bieber, P., Castel, C., Bougnol, C., Heckmann, J.-P., et al. (2004). Architecture Patterns for Safe Design. *AAAF 1st Complex and Safe Systems Engineering Conference (CS2E 2004).*

Kloul, L., Prosvirnova, T., & Rauzy, A. (2013, December). Modeling systems with mobile components: a comparison between AltaRica and PEPA nets. *Journal of Risk and Reliability, 227*(6), 599-613.

Milner, R. (1999). *Communicating and Mobile Systems: The pi-calculus.* Cambridge, CB2 8BS, United Kingdom: Cambridge University Press.

Milner, R. (2009). *The Space and Motion of Communicating Agents.* Cambridge, CB2 8BS, United Kingdom: Cambridge University Press.

Noble, J., Taivalsaari, A., & Moore, I. (1999). *Prototype-Based Programming: Concepts, Languages and Applications.* Berlin and Heidelberg, Germany: Springer-Verlag.

Prosvirnova, T., Batteux, M., Brameret, P.-A., Cherfi, A., Friedlhuber, T., Roussel, J.-M., et al. (2013, September). The AltaRica 3.0 project for Model-Based Safety Assessment. *Proceedings of 4th IFAC Workshop on Dependable Control of Discrete Systems, DCDS'2013* (pp. 127-132). York: International Federation of Automatic Control.

Prosvirnova, T., Batteux, M., Maarouf, A., & Rauzy, A. (2013, September-October). GraphXica: a Language for Graphical Animation of models. *Proceedings of the European Safety and Reliability conference, ESREL 2013.* Amsterdam: CRC Press.

Steinberg, D., Budinsky, F., Paternostro, M., & Merks, E. (2008, December). *EMF: Eclipse Modeling Framework.* Boston, MA 02116, USA: Addison Wesley.

Valiant, L. G. (1979). The Complexity of Enumeration and Reliability Problems. *SIAM Journal of Computing, 8*(3), 410-421.

Voirin, J.-L. (2008, June). Method and Tools for Constrained System Architecting. *Proceedings 18th Annual International Symposium of the International Council on Systems Engineering (INCOSE 2008)* (pp. 775-789). Utrecht: Curran Associates, Inc.

White, S., & Miers, D. (2008). *BPMN Modeling and Reference Guide: Understanding and Using BPMN.* Lighthouse Point, FL, USA: Future Strategies Inc.

Wirth, N. (1976). *Algorithms + Data Structures = Programs.* Upper Saddle River, New Jersey 07458, USA: Prentice-Hall.