COMPARAISON DES LANGAGES DE MODELISATION ALTARICA ET FIGARO

COMPARISON OF THE MODELING LANGUAGES ALTARICA AND FIGARO

Marc BOUISSOU
EDF R&D
1, avenue du Général de Gaulle
92141 CLAMART Cedex
et CNRS –UMR 8050
Marc.Bouissou@edf.fr

Christel SEGUIN
ONERA
2, av. E. Belin
31055 Toulouse Cedex

Christel.Seguin@onera.fr

Résumé
Le but de cette communication est de présenter de la manière la plus exhaustive possible, et en l'illustrant par des exemples, les différences entre les langages de modélisation AltaRica et FIGARO. La comparaison portera sur les objectifs poursuivis par ces deux langages, leurs contextes d'utilisation, et leurs caractéristiques intrinsèques. L'objectif est de comparer les langages eux-mêmes, et non les outils qui ont été développés en s'appuyant sur ces langages. On se place donc résolument dans une optique "méthode" plutôt que "outil".

Summary
The purpose of this communication is to present as exhaustively as possible, and with illustrative examples, the differences between the modelling languages AltaRica and FIGARO. The comparison will deal with the objectives pursued by these two languages, their contexts of use, and their intrinsic characteristics. The objective is to compare the languages themselves, and not the tools which were developed on the basis of these languages. The viewpoint of this paper is thus determinedly about "methods" rather than "tools".

Introduction

Les langages FIGARO et AltaRica sont deux langages de modélisation de systèmes qui ont été explicitement créés pour faciliter la réalisation d'études de sûreté de fonctionnement de systèmes. Par "langage", nous entendons des représentations textuelles de haut niveau dont le pouvoir de description va bien au delà de ce que l'on peut représenter avec un formalisme purement graphique, si sophistiqué qu'il soit.
L'existence même de ces langages montre qu'il existe un besoin qui n'est pas couvert par les langages de programmation, ni même par les langages de modélisation exclusivement orientés vers la simulation de Monte-Carlo tels que SIMSCRIPT [1].
L'enjeu est la possibilité de créer **automatiquement** des modèles fiabilistes tels que les arbres de défaillances, les graphes de Markov, ou des modèles se prêtant à la simulation de Monte-Carlo, à partir de représentations qui peuvent être **capitalisées** dans des bases de connaissances, faisant ainsi gagner un temps précieux lors de l'étude de nouveaux systèmes.
A première vue, donc, les langages AltaRica et FIGARO répondent aux mêmes objectifs. Il est donc intéressant d'examiner leurs points communs, car ils constituent sans doute des "incontournables", mais aussi leurs différences. On peut se demander s'il serait possible de créer un nouveau langage qui ferait la synthèse des apports de ces deux façons d'aborder la modélisation. A défaut, peut on imaginer un traducteur automatique permettant de passer de l'un à l'autre ?
Pour tenter de répondre à ces questions, nous allons commencer par présenter les deux langages puis nous les comparerons suivant différents critères, allant de leur contexte d'utilisation à une comparaison de leurs sémantiques, c'est à dire des modèles théoriques sous-jacents. Pour rendre le propos plus concret, nous donnerons aussi la modélisation complète dans ces deux langages d'un petit système physique, qui pose différents problèmes de modélisation.

Objectifs des deux langages

Face à la complexification des systèmes à étudier, et à la croissance des exigences de rapidité et exhaustivité des études de sûreté de fonctionnement, les fiabilistes ont dû se doter de nouvelles techniques de modélisation. En effet, la construction manuelle de modèles classiques tels que les arbres de défaillances, les graphes de Markov, les réseaux de Petri, etc. demande à la fois beaucoup de temps et d'expertise, comporte des risques d'erreur, et ne permet pas la capitalisation des connaissances autrement que dans la tête des analystes.

Les modèles classiques présentent l'inconvénient d'être peu robustes aux changements d'hypothèses car ils ne sont pas "compositionnels". Leur caractère monolithique fait qu'ils sont lisibles uniquement par des experts, et qu'ils sont très peu réutilisables.

Pour répondre à tous les besoins, un langage de modélisation idéal (pour la sûreté de fonctionnement des systèmes) devrait avoir les caractéristiques suivantes : être compositionnel, pouvoir être partagé par ceux qui font les analyses de sûreté de fonctionnement et les concepteurs de systèmes, permettre des études qualitatives et quantitatives de même type que celles que l'on fait avec les méthodes classiques : recherche exhaustive de causes, d'effets, démonstration de propriétés qualitatives, quantification probabiliste. Un tel langage devrait permettre une bonne structuration des connaissances de façon à permettre la capitalisation et la réutilisation des modèles, posséder une syntaxe et une sémantique claires. Il devrait être facile à documenter et à associer à des visualisations graphiques variées.
Les deux langages FIGARO et AltaRica possèdent toutes ces caractéristiques à des degrés divers, comme nous allons le montrer dans la suite.

Présentation rapide des deux langages

Contexte de création et évolutions ultérieures

Le langage FIGARO a été créé à EDF en 1989-1990 [2]. Sa définition a pu être rapide, car il s'appuyait sur une expérience de plusieurs années acquise à EDF dans l'écriture de modèles de sûreté de fonctionnement et de systèmes experts en logique des propositions et logique du premier ordre. Le besoin initial pour la création de ce langage a été le remplacement de deux outils très différents par une approche unifiée. Ces outils étaient [3] :

- un système expert de génération automatique d'arbres de défaillances, utilisant le moteur d'inférence d'ordre 1 ALOUETTE, et deux bases de règles : l'une pour les systèmes thermohydrauliques, et l'autre pour les systèmes électriques,
- le prédécesseur de l'outil actuel FIGSEQ de recherche et quantification de séquences à partir d'un modèle écrit dans un langage de modélisation très proche de l'actuel FIGARO d'ordre 0 [4].

Ces deux outils (dépourvus de toute interface graphique) étaient utilisés dans le contexte des EPS (Etudes Probabilistes de Sûreté) des centrales nucléaires pour automatiser la construction de modèles **quantitatifs**.
En plus des objectifs communs cités dans le paragraphe précédent, le langage FIGARO avait un objectif de généralisation des formalismes classiques (arbres de défaillances, graphes de Markov, diagrammes et réseaux de fiabilité, réseaux de Petri stochastiques, files d'attente...) en permettant de les manipuler **avec leurs représentations graphiques habituelles** [2] [5]. Ainsi les développements effectués sur les outils de saisie graphique et de traitement pourraient servir non seulement pour des modèles construits par une approche de type base de connaissances (comme pour les systèmes thermohydrauliques et électriques), mais aussi pour les modèles graphiques classiques des fiabilistes sans que cela implique des développements spécifiques.

Le langage FIGARO a bénéficié d'une grande stabilité à la fois syntaxique et sémantique, ce qui a permis de réutiliser les types FIGARO de certaines bases de connaissances plus de dix ans après leur création, pratiquement sans adaptation, avec l'outil KB3 v3 [6] [18].

Les premières études sur le langage de modélisation AltaRica [23] ont débuté au milieu des années 1990 et ont abouti à une première version du langage et des outils associés définie dans la thèse de Gérald Point au Labri [7]. Dans cette version, les modèles AltaRica sont constitués d'automates de mode (ou à contraintes) hiérarchisés interfacés quelconques [8].
Au départ, les développeurs d'AltaRica ont cherché uniquement à décrire des comportements qualitatifs, ne faisant intervenir aucune loi de probabilité.
Ils ont avant tout recherché les concepts minimaux permettant une projection aisée et rigoureuse des modèles AltaRica vers les modèles traditionnels de fiabilité (arbre de défaillances, réseaux de Petri stochastique, automate à états finis, ...). Ainsi, ils ont choisi de représenter la dynamique d'un composant par un automate de modes. Dans un automate de modes, des transitions d'automate qui modélisent les enchaînements des modes de fonctionnement et défaillance, cohabitent avec des formules booléennes qui contraignent les valeurs que peuvent prendre les paramètres du système dans chaque mode.
L'environnement d'édition et de calcul Cecilia OCAS a été ensuite développé par Dassault Aviation au début des années 2000. Cet environnement a été notamment utilisé pour modéliser les commandes de vol électrique du Falcon 7X et générer des arbres de défaillances qui ont contribué au dossier de certification de l'avion. Il s'est avéré que pour développer leurs modèles, les ingénieurs ont utilisé un fragment d'AltaRica appelé AltaRica DataFlow. Dans ce fragment, les flux sont orientés et ceci permet de tester plus aisément la complétude de chaque description. D'autre part des outils d'évaluation plus optimisés ont été développés pour ce fragment du langage et appliqués non seulement chez Dassault mais aussi validés dans différents projets de recherche dont ESACS [9] et ISAAC [10]. Le fragment AltaRica DataFlow devrait aussi être intégré à l'atelier SIMFIA d'APSYS.
Parallèlement, différentes extensions d'AltaRica ont été proposées ou font l'objet de travaux de recherche. Dans sa thèse, Claire Pagetti a introduit des variables d'horloges pour pouvoir modéliser des contraintes de durées [11]. Christophe Kehren a proposé de généraliser les contraintes en considérant non seulement des invariants d'états mais aussi des formules de logique temporelle linéaire quelconque [12]. Enfin, le LaBri étudie comment introduire le concept de type abstrait algébrique.

Modes d'utilisation

Les modes d'utilisation des deux langages sont très différents.
Pour le langage FIGARO il existe clairement deux classes d'utilisateurs : ceux qui développent des bases de connaissances et ceux qui les utilisent. Les premiers sont des fiabilistes capables d'extraire de la connaissance sur les systèmes à étudier à partir de documents techniques ou d'experts. Les seconds peuvent être des fiabilistes chargés d'études, ou des concepteurs de systèmes n'ayant qu'une formation minimale en fiabilité. Ces deux modes d'utilisation correspondent d'ailleurs à deux niveaux du langage. Le langage FIGARO "d'ordre 1" sert à écrire des modèles très génériques dans des bases de connaissances. Il permet d'utiliser des constructions élaborées, notamment grâce à l'existence des quantificateurs. Pour construire un modèle de système à partir d'une base de connaissances, il suffit de déclarer la liste des objets du système, les relations qui les lient, et les surcharges éventuelles de valeurs de paramètres (lorsque les valeurs par défaut définies dans la base de connaissances ne conviennent pas). En pratique, cela se fait toujours par une saisie graphique.
Le langage FIGARO dit "d'ordre 0" est un sous langage très simplifié du langage FIGARO. Il ne permet d'écrire que des modèles spécifiques à un système donné. Le rôle premier du langage FIGARO0 est de servir d'interface entre l'outil de construction graphique de modèles et les outils de traitement. Sa simplicité le rend particulièrement stable, ce qui permettra d'ajouter de nouvelles caractéristiques au langage FIGARO1 sans avoir à modifier les outils de traitement. L'article [6] donne un exemple complet de base de connaissances, de description de système sous forme de liste d'objets, et un extrait de son instanciation en FIGARO0.
Les utilisateurs de bases de connaissances matures ont rarement besoin d'en adapter les modèles. Si cette nécessité existe, ils peuvent faire quelques ajouts simples d'objets, attributs, règles dans le modèle FIGARO0. Pour des modifications plus profondes, ils s'adressent au concepteur de la base de connaissances qu'ils utilisent.

Les modèles AltaRica peuvent être réalisés à différentes fins (analyse de sécurité, de fiabilité opérationnelle, …). Ils sont créés à partir de composants génériques élémentaires, que l'on instancie en nombre suffisant et que l'on interconnecte pour créer des modules plus complexes jusqu'à l'obtention du modèle du système complet. AltaRica ayant été défini à partir d'un ensemble minimum de concepts, le langage peut être aisément appris. Aussi, les utilisateurs peuvent aussi bien réutiliser des modèles que les adapter ou en créer de nouveaux, par exemple si le système comprend des composants technologiques nouveaux, avec des modes de défaillances propres.
Lorsque AltaRica est utilisé pour réaliser des évaluations de sécurité, l'ensemble des composants peuvent être regroupés en bibliothèques construites en suivant par exemple les principes présentés dans [13]. Cependant, l'aspect documentation et gestion des bibliothèques est propre à chaque entreprise et indépendant du langage.
Cette grande flexibilité est compatible avec différentes organisations, que les ingénieurs responsables des analyses de sécurité soient intégrés aux équipes de concepteurs ou bien qu'ils appartiennent à des équipes distinctes.
Enfin, dans le milieu aéronautique, cette flexibilité et la rapidité de construction des modèles semblent nécessaires pour pouvoir non seulement faciliter a posteriori la certification des avions et de leur variantes mais aussi, a priori, accompagner la conception des systèmes.

Principales caractéristiques

Les deux langages ont été conçus pour modéliser la propagation de pannes au sein d'un système. Les deux permettent donc de modéliser au premier chef des états d'erreur, les événements qui les ont causés, les modes de défaillances induits par ces états. Ils se distinguent dans les moyens fournis pour structurer ces informations et dans les primitives offertes pour décrire la dynamique du système.
Pour résumer en quelques mots les caractéristiques les plus saillantes des deux langages, on peut retenir que :
Le langage FIGARO est orienté objet : il permet la définition de types organisés grâce à des relations d'héritage (éventuellement multiple) et d'objets qui héritent de ces types. Il possède deux niveaux (l'ordre 1 et l'ordre 0). Un modèle FIGARO est composé d'une liste fixée (il n'y a pas de création ni destruction d'objets en cours d'évolution du système) d'objets en interaction. Ces objets ne sont pas organisés hiérarchiquement et encapsulés (il n'y a pas d'emboîtement entre objets), et, à l'ordre 0, tout objet peut interagir directement avec tous les autres objets du modèle, et ce, de diverses manières.
Enfin le langage FIGARO permet d'écrire des modèles probabilistes quantifiables pouvant être simplifiés en modèles qualitatifs.

Le langage AltaRica permet de définir des modèles composés d'une liste fixée (il n'y a pas de création ni destruction d'objets en cours d'évolution du système) d'instances d'objets (appelés "nodes") en interaction. Ces objets sont organisés de manière hiérarchique : chaque objet contient un certain nombre de caractéristiques propres (états internes d'erreur, événements cause, flux en interfaces) et la déclaration d'objets de niveau inférieur. Par défaut, les règles de visibilité et d'accès aux attributs des objets suivent la hiérarchie. Au niveau le plus abstrait, le système global est représenté par un objet unique. Chaque type d'objet peut être instancié plusieurs fois dans un modèle, mais il n'existe pas de relation d'héritage permettant de factoriser des informations communes à plusieurs types. Les interfaces d'un type sont partitionnées en un nombre fixé d'entrées et un nombre fixé de sorties. Les deux seuls moyens pour faire interagir des objets sont : soit identifier une sortie d'un objet avec une entrée d'un autre, soit définir un "vecteur de synchronisation" capable de provoquer un changement d'état simultané sur deux ou plusieurs objets.
Le langage AltaRica est orienté vers la création de modèles qualitatifs pouvant être enrichis par des informations de nature probabiliste.

Exemple : un petit modèle électrique dans les deux langages

Description du problème

Soit le système non réparable suivant, composé de deux sources électriques (S1 et S2), deux récepteurs (R1 et R2) et un interrupteur (I1).

Le récepteur R1 est en priorité alimenté par la source S1, et est réalimenté par la source S2 en cas de perte de S1, grâce à la fermeture de I1. Un fonctionnement symétrique est supposé pour l'alimentation de R2.

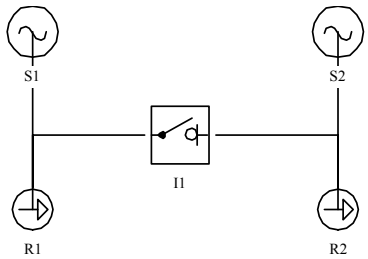

Figure 1 : système électrique simple

On suppose pour simplifier que les seuls modes de défaillance à prendre en compte sont les suivants :

Mode de défaillance →	*en fonctionnement*	*à la sollicitation*
Sources	*Destruction : DS*	
Récepteurs	*Court-circuit : CC*	
Interrupteur	*Fermeture intempestive : FI*	*Refus de fermeture : RF*

Le mode de défaillance DS des sources peut se produire "spontanément", ou bien de manière provoquée par d'autres dysfonctionnements. La destruction provoquée des deux sources se produit si elles sont mises en relation par la fermeture de l'interrupteur alors qu'aucune d'elles n'est perdue. Si un court-circuit se produit sur un récepteur, cela provoque également la destruction des sources auxquelles le récepteur est relié.
Ces hypothèses ne sont guère réalistes : dans un système réel, on aurait à la fois des modes de défaillances supplémentaires, à commencer par la coupure et le court-circuit sur les câbles, et des composants supplémentaires, tels que des disjoncteurs destinés à éviter la propagation de courts-circuits. Mais nous avons dû simplifier l'exemple à l'extrême pour pouvoir lister intégralement les modèles dans l'article.
Les modèles FIGARO et AltaRica que nous donnons ci-après **in extenso** ont été construits et testés à l'aide d'outils, seul moyen pour assurer leur complétude et leur validité.

Modèle AltaRica

Un fil électrique peut indifféremment véhiculer un courant dans un sens ou dans l'autre. Pour modéliser ceci, deux solutions sont envisageables : écrire des modèles AltaRica généraux dont les composants sont interfacés par des flux non orientés ou écrire des modèles AltaRica Dataflow dont les flux sont soit des entrées, soit des sorties, soit des variables locales. Nous avons choisi d'utiliser cette dernière approche car aujourd'hui, seul ce type de modèles peut être traité efficacement par les outils d'analyse en AltaRica.
Ainsi, un fil électrique permettant de connecter deux composants sera décomposé en deux variables, notées systématiquement `port_e` pour le courant entrant et `port_s` pour le courant sortant, comme illustré sur la figure ci dessous. De plus, le flux électrique est une donnée structurée, constituée de deux champs ^V et ^CC, vrais respectivement lorsque la direction du courant est celle de la variable et lorsqu'un court-circuit est présent. Dans la Figure 2, les couleurs matérialisent la valeur des champs ^V entrant et sortant et l'on voit que dans la configuration donnée, le courant circule des sources vers les récepteurs. Les composants du modèle AltaRica sont construits en respectant globalement cette philosophie et constituent une bibliothèque réutilisable.

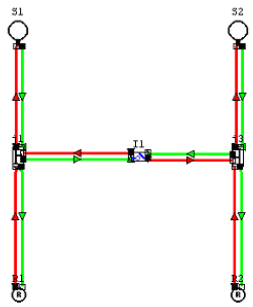

Figure 2 : représentation graphique du modèle AltaRica

Ce modèle est composé de quatre type de composants: source, recepteur, interrupteur et « t_jonction » explicitant comment les sorties d'un composant peuvent être divisées en deux et comment deux entrées peuvent être réunies en une.

```
node source
flow
    e^V:bool:in; e^CC:bool:in;
    s^V:bool:out; s^CC:bool:out;
state
    marche:bool; // la source est en état de marche //
    conflit_detecte:bool; // détec. conflit mémorisée //
event
    DS; // destruction spontanée//
    CC_recu; // destruction par court circuit //
    conflit_sources; // destruction par autre source //
    detec_conflit;   // détec. + mémorisation conflit //
trans
    marche |- DS -> marche := false;
    (marche and e^CC) |- CC_recu -> marche := false;
    (marche and e^V) |- detec_conflit ->
                              conflit_detecte := true;
    (conflit_detecte and marche) |- conflit_sources ->
                              marche := false;
assert
    s^V = marche;
    s^CC = false;
init
    marche := true,
    conflit_detecte := false;
extern
    law <event DS> = exponential(1e-5);
    priority <event CC_recu> = 1;
    priority <event conflit_sources> = 2;
    priority <event detec_conflit> = 1;
edon

node recepteur
flow
    e^V:bool:in; e^CC:bool:in;
    s^V:bool:out; s^CC:bool:out;
    r_CC:bool:out; // indique si le récepteur génère des
courts-circuits //
state
    emission_CC:bool; // le récepteur génère des courts-
circuits //
event
    CC_emis; // génération de courts-circuits //
trans
    (not emission_CC) |- CC_emis -> emission_CC := true;
assert
    s^V = false;
    s^CC = (emission_CC and e^V);
    r_CC = emission_CC;
init
    emission_CC := false;
extern
    law <event CC_emis> = exponential(1e-5);
edon

node interrupteur
flow
    a_e^V:bool:in; a_e^CC:bool:in;
    a_s^V:bool:out; a_s^CC:bool:out;
    b_e^V:bool:in; b_e^CC:bool:in;
    b_s^V:bool:out; b_s^CC:bool:out;
    r1_CC:bool:in; // test du récepteur 1 //
    r2_CC:bool:in; // test du récepteur 2 //
    fermeture_demande:bool:private; // calcul local //
state
    ouvert:bool; bloque:bool;
event
    ferm;  // fer meture normale //
    FI;    // fermeture intempestive //
    RF;    // refus de fermeture //
trans
    (fermeture_demande and ouvert and not bloque) |-
          ferm -> ouvert := false;
    (fermeture_demande and ouvert and not bloque) |-
          RF ->  ouvert := true, bloque :=true;
    ((not fermeture_demande) and ouvert)
          |- FI -> ouvert := false;
assert
    fermeture_demande = ((((not a_e^V) or (not b_e^V))
and (not r1_CC)) and (not r2_CC));
    a_s^V = (not ouvert and b_e^V);
    a_s^CC = (not ouvert and b_e^CC and a_e^V);
    b_s^V = (not ouvert and a_e^V);
    b_s^CC = (not ouvert and a_e^CC and b_e^V);
init
    ouvert := true;
extern
     bucket {<event ferm>, <event RF>}
    law <event ferm> = constant(0.999);
    law <event RF> = constant(0.001);
    law <event FI> = exponential(1e-5);
edon

node t_jonction
flow
    a_e^V:bool:in; a_e^CC:bool:in;
    a_s^V:bool:out; a_s^CC:bool:out;
    b_e^V:bool:in; b_e^CC:bool:in;
    b_s^V:bool:out; b_s^CC:bool:out;
    c_e^V:bool:in; c_e^CC:bool:in;
```

```
    c_s^V:bool:out; c_s^CC:bool:out;
assert
    a_s^V = (b_e^V or c_e^V);
    a_s^CC = ((b_e^CC or c_e^CC) and a_e^V);
    b_s^V = (a_e^V or c_e^V);
    b_s^CC = ((a_e^CC or c_e^CC) and b_e^V);
    c_s^V = (a_e^V or b_e^V);
    c_s^CC = ((a_e^CC or b_e^CC) and c_e^V);
edon

node main
sub
    S1:source; S2:source;
    R1:recepteur; R2:recepteur;
    I1:interrupteur;
    j1:t_jonction; j2:t_jonction;

assert  // connexions entre composants //
    S1.e^V = j1.a_s^V; S1.e^CC = j1.a_s^CC;
    S2.e^V = j2.b_s^V; S2.e^CC = j2.b_s^CC;
    R1.e^V = j1.b_s^V; R1.e^CC = j1.b_s^CC;
    R2.e^V = j2.a_s^V; R2.e^CC = j2.a_s^CC;
    I1.a_e^V = j1.c_s^V; I1.a_e^CC = j1.c_s^CC;
    I1.b_e^V = j2.c_s^V; I1.b_e^CC = j2.c_s^CC;
    I1.r1_CC = R1.r_CC; // non représenté sur dessin //
    I1.r2_CC = R2.r_CC; // non représenté sur dessin //
    j1.a_e^V = S1.s^V; j1.a_e^CC = S1.s^CC;
    j1.b_e^V = R1.s^V; j1.b_e^CC = R1.s^CC;
    j1.c_e^V = I1.a_s^V; j1.c_e^CC = I1.a_s^CC;
    j2.a_e^V = R2.s^V; j2.a_e^CC = R2.s^CC;
    j2.b_e^V = S2.s^V; j2.b_e^CC = S2.s^CC;
    j2.c_e^V = I1.b_s^V; j2.c_e^CC = I1.b_s^CC;
edon
```

Sémantique du langage AltaRica

Un modèle AltaRica est un automate qui génère un graphe des états atteignables à partir d'états initiaux, appelé structure de Kripke (W, W0,Var, Dom(Var), Evt, m, t) définie comme suit :

- W = ensemble des états atteignables et W0, partie de W représentant les états initiaux possibles
- Var = ensemble fini de variables dénotant les flux (entrées, sorties) ou les états internes des composants du système
- Dom(Var) = ensemble des valeurs que peuvent prendre les variables
- Evt = ensemble fini d'événements
- m: W × Var → Dom(Var) = fonction d'assignation de valeurs aux variables
- t:W × Evt → W = relation de transition entre états étiquetée par les événements.

Cette structure est construite à partir d'un modèle AltaRica de la manière suivante :

- Dans les états initiaux de W0, on alloue tout d'abord une valeur aux variables d'état internes initialisées par une clause « init ». Pour les autres variables de Var, on considère toutes les allocations compatibles avec les contraintes établies dans les clauses « assert » et l'on peut obtenir ainsi **plusieurs** états initiaux possibles.
- Dans un état (défini par l'ensemble des valeurs des variables de Var) donné, on examine l'ensemble des transitions des clauses « trans ». Elles sont de la forme «garde |- nom_evt -> post-conditions». Les *gardes* sont des expressions boléennes définies à partir de conditions sur les flux et les états internes. Les *post-conditions* affectent explicitement une nouvelle valeur à certaines variables d'état internes et implicitement laissent inchangées les autres. Si dans un état w les valeurs assignées aux flux et aux états internes vérifient les conditions exprimées dans la garde d'une transition t, t est une transition *candidate*. Les transitions peuvent être groupées par des *vecteurs de synchronisation* (ceci n'est pas illustré sur l'exemple). Selon le type de synchronisation, le groupe sera candidat si toutes ses transitions sont candidates ou si l'une d'elle au moins l'est. Des *priorités* peuvent être attribuées aux transitions (par exemple, on a attribué une priorité 1 aux événements detec_conflit, conflit_sources, CC_recu pour une source). Parmi les transitions ou groupes de transition synchronisées candidats, seuls les plus prioritaires sont *tirables*. Pour chaque transition tirable, on construira tous les successeurs possibles qui assignent aux flux et aux variables internes des valeurs compatibles avec l'effet de la post-condition et avec les assertions. Pour un groupe de transitions synchronisées, le successeur devra satisfaire simultanément les post-conditions de chacune des transitions du groupe
- Le processus est itéré indéfiniment dans le cas le plus général ou jusqu'à ce que l'on ne construise plus de nouveaux états de W dans la plupart des cas traités.

Modèle Figaro

Le modèle FIGARO se compose de deux parties : la base de connaissances (écrite sous une forme générique), et la liste des objets du système à étudier.

Voici tout d'abord la base de connaissances :

```
ORDRE_DES_ETAPES
    determination_isolants;
    propagation_cc;
    propagation_flux;
    reconfiguration;

TYPE arete_bi_dir ;
EFFET  V_e ; V_s ; cc_e ; cc_s ;

TYPE noeud ;
INTERFACE
    entree GENRE arete_bi_dir;
    sortie GENRE arete_bi_dir;
CONSTANTE
    fonction DOMAINE 'source' 'recepteur' 'autre'
        PAR_DEFAUT 'autre';
    lambda DOMAINE REEL PAR_DEFAUT 1e-5 ;
EFFET
    relie LIBELLE "%OBJET est alimenté et en état de
            marche";
    isolant LIBELLE "%OBJET ne transmet pas la tension";
INTERACTION
ETAPE determination_isolants
SI (fonction = 'source' OU fonction = 'recepteur')
ET PANNE ALORS isolant;

regle_tension_source
ETAPE propagation_flux
SI MARCHE ET fonction = 'source'
ALORS relie, POUR_TOUT x UNE sortie FAIRE V_s(x),
POUR_TOUT y UNE entree FAIRE V_e(y);

regle_tension_non_source
ETAPE propagation_flux
SI NON isolant ET fonction <> 'source' ET
((IL_EXISTE x UNE entree TELLE_QUE (V_s(x) ET NON
cc_s(x))) OU
(IL_EXISTE y UNE sortie TELLE_QUE (V_e(y) ET NON
cc_e(y))))
ALORS relie, POUR_TOUT x UNE sortie FAIRE V_s(x),
POUR_TOUT y UNE entree FAIRE V_e(y);

regle_CC_recepteur
ETAPE propagation_cc
SI fonction = 'recepteur' ET PANNE
ALORS POUR_TOUT x UNE sortie FAIRE cc_s(x),
POUR_TOUT y UNE entree FAIRE cc_e(y);

regle_CC_non_recepteur
ETAPE propagation_cc
SI NON isolant ET
((IL_EXISTE x UNE entree TELLE_QUE cc_s(x)) OU
(IL_EXISTE y UNE sortie TELLE_QUE cc_e(y)))
ALORS POUR_TOUT x UNE sortie FAIRE cc_s(x),
POUR_TOUT y UNE entree FAIRE cc_e(y);

TYPE source SORTE_DE noeud ;
CONSTANTE fonction PAR_DEFAUT 'source';
PANNE     DS LIBELLE "destruction de %OBJET";

OCCURRENCE
SI MARCHE IL_PEUT_SE_PRODUIRE
DEFAILLANCE DS LOI EXP(lambda);

INTERACTION
destruction_par_CC_recu
ETAPE propagation_cc
SI MARCHE ET
((IL_EXISTE x UNE entree TELLE_QUE cc_s(x)) OU
(IL_EXISTE y UNE sortie TELLE_QUE cc_e(y)))
ALORS DS;

TYPE recepteur SORTE_DE noeud ;
CONSTANTE fonction PAR_DEFAUT 'recepteur';
PANNE     CC LIBELLE "court-circuit de %OBJET" ;

OCCURRENCE
SI relie IL_PEUT_SE_PRODUIRE
DEFAILLANCE CC LOI EXP(lambda);

TYPE interrupteur SORTE_DE noeud ;
INTERFACE recept GENRE recepteur ;
CONSTANTE gamma DOMAINE REEL PAR_DEFAUT 1e-3 ;
ATTRIBUT  position DOMAINE 'ferme' 'ouvert'
          PAR_DEFAUT 'ouvert';
EFFET     demande_fermeture ;
PANNE
      FI LIBELLE "fermeture intempestive de %OBJET";
      RF LIBELLE "refus de fermeture de %OBJET";

OCCURRENCE
SI position = 'ouvert' ET NON RF
IL_PEUT_SE_PRODUIRE DEFAILLANCE FI
PROVOQUE position <-- 'ferme' LOI EXP(lambda);
```

```
SI position = 'ouvert' ET demande_fermeture
IL_PEUT_SE_PRODUIRE
DEFAILLANCE RF LOI INS(gamma)
OU_BIEN TRANSITION ferm
PROVOQUE position <-- 'ferme';

INTERACTION
ETAPE determination_isolants
SI position = 'ouvert' ALORS isolant;

ETAPE reconfiguration
SI (IL_EXISTE x UN recept TEL_QUE NON relie(x)) ET
(QQSOIT y UN recept ON_A MARCHE(y))) ET
position='ouvert'
ALORS demande_fermeture ;

TYPE chemin ;
INTERFACE
src GENRE source CARDINAL 2 JUSQUA INFINI ;
int GENRE interrupteur ;
INTERACTION
destruction_sources
ETAPE propagation_cc
SI QQSOIT x UN int ON_A position(x) = 'ferme'
ET QQSOIT y UNE src ON_A MARCHE(y)
ALORS POUR_TOUT z UNE src FAIRE DS(z);
```

Voici maintenant la liste des objets : cette liste équivaut strictement à la Figure 3. Dans cette figure, le sens de tracé des liens est indiqué par le sens des flèches. Grâce à la symétrie des règles écrites en FIGARO, le sens du tracé n'a pas d'incidence sur le fonctionnement du modèle.

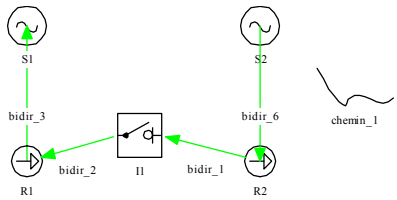

Figure 3 : représentation graphique du modèle FIGARO

```
OBJET I1 EST_UN interrupteur;
 INTERFACE
   entree = bidir_1; sortie = bidir_2; recept = R1 R2;
OBJET R1 EST_UN recepteur;
 INTERFACE  entree = bidir_2; sortie = bidir_3;
OBJET R2 EST_UN recepteur;
 INTERFACE  entree = bidir_6; sortie = bidir_1;
OBJET S1 EST_UN source;
 INTERFACE  entree = bidir_3;
OBJET S_2 EST_UN source;
 INTERFACE  sortie = bidir_6;
OBJET chemin_1 EST_UN chemin;
 INTERFACE  src = S1 S2; int = I1;
OBJET bidir_1 EST_UN arete_bi_dir;
OBJET bidir_2 EST_UN arete_bi_dir;
OBJET bidir_3 EST_UN arete_bi_dir;
OBJET bidir_6 EST_UN arete_bi_dir;
```

Sémantique du langage FIGARO 0

Le langage FIGARO d'ordre 1 ne sert qu'à permettre la création de modèles en FIGARO 0, à partir desquels sont faits les traitements. C'est pourquoi il est inutile de définir une sémantique à l'ordre 1. Un modèle en FIGARO0 décrit un processus stochastique X_t, où X_t est un vecteur fini de variables d'état dont la valeur définit l'état du modèle à l'instant t.
Afin de garder un exposé simple, nous écartons dans la présente description la possibilité, offerte par le langage FIGARO, d'associer des transitions temporisées à des lois probabilistes[1] dont le support est un sous-ensemble strict de $[0,+\infty[$. Nous ne détaillons pas non plus le comportement temporel de X_t : nous nous contenterons de définir les états vers lesquels le processus peut évoluer à partir d'un état donné, et si le changement d'état est instantané ou a lieu après un temps non nul, autrement dit, **l'automate représentant le comportement qualitatif du processus**. Ces simplifications nous éviteront d'avoir à définir un échéancier, une notion relativement complexe qui encombrerait quelque peu le propos. La distinction entre états instantanés et états non instantanés ou "tangibles" est une **nécessité** pour donner au langage un pouvoir de modélisation suffisant. En effet, ces deux types d'états ne sont pas traités de la même manière, ainsi qu'on le verra dans le paragraphe définissant comment l'automate peut évoluer à partir d'un état donné.
Soit L un ensemble de lois probabilistes, appartenant à l'une des deux catégories suivantes : lois continues sur $[0,+\infty[$, lois discrètes associant des probabilités à un nombre fini de modalités.

[1] Comme les lois de type Dirac, qui modélisent des délais déterministes, ou des lois dont le support est un intervalle [a, b]. En effet, l'existence de telles lois dans le modèle peut en compliquer sérieusement le comportement **qualitatif**

Un modèle en langage FIGARO0 est un nuplet $(\Xi, O, T, I, \sigma, Y_0, V_0)$ composé des éléments suivants :
Ξ est le produit cartésien $E_1 \otimes E_2 ... \otimes E_n$ des domaines des composantes de X, un vecteur fini de variables d'état $(x_1, x_2, ... x_n)$ dont la valeur définit l'état du modèle à l'instant t. X est en fait la concaténation V, Y de deux vecteurs V et Y. V regroupe les variables dites "**essentielles**" et Y les variables dites "**déduites**".
V_0 est la valeur initiale de V, et Y_0 est la valeur dite "de réinitialisation" de Y. **Y est une fonction de V et de Y_0 définie à l'aide de la fonction I décrite plus loin**.
T est l'ensemble des **groupes de transitions** du modèle. Une **transition** est une application de Ξ dans Ξ, qui à tout vecteur d'état fait correspondre un autre vecteur d'état, en général obtenu par la modification d'un faible nombre de variables essentielles. Un groupe de transitions, quant à lui, est un couple dont les éléments sont un ensemble de transitions dites "liées", et une loi de probabilité de L. Le groupe de transitions contient une seule transition si sa loi n'est pas de type loi discrète. Si la loi est de type discret, le groupe de transitions contient autant de transitions que la loi a de modalités. Un tel groupe de transitions sert à modéliser un choix aléatoire instantané entre plusieurs transitions (par exemple, le résultat d'un lancer de dé). Par abus de langage, on confondra la notion de groupe de transitions et de transition pour les groupes ne contenant qu'une transition.
O est une fonction de Ξ dans 2^T. En pratique, O est définie par l'ensemble des **règles dites "d'occurrence"**. A tout état du modèle, ces règles permettent d'associer un ensemble (éventuellement vide) de groupes de transitions.
I est une fonction de Ξ dans Ξ, qui à tout vecteur d'état X **pour lequel I est définie** fait correspondre un autre vecteur d'état. La fonction I est en pratique définie par les **règles dites "d'interaction"** du modèle FIGARO (et éventuellement un **système d'équations linéaires**), et cette fonction dépend **éventuellement** de l'ordre σ des règles. La fonction I est définie comme la composée d'un ensemble fini de fonctions de Ξ dans Ξ, notées $I_0, I_1, ... I_P$. Autrement dit, $I(X) = I_p(I_{p-1}(...I_0(X)...))$.
Les règles d'interaction du modèle sont regroupées en "étapes", correspondant aux différentes fonctions $I_1, ... I_P$. Quant à I_0, c'est une fonction particulière de réinitialisation des variables déduites :
$I_0(V, Y) = V, Y_0$

Inférence réalisée dans une étape
La fonction correspondant à chaque étape est définie par le fonctionnement d'un moteur d'inférence simple. Soit $\{R_k\}$ l'ensemble des règles d'une étape donnée (pour alléger les notations, nous omettons l'indice d'étape). La règle R_k est une fonction de Ξ dans Ξ, qui à tout vecteur d'état X fait correspondre un autre vecteur d'état $R_k(X)$. Chaque règle est en pratique composée d'une **condition** calculée par une fonction booléenne de l'état X, permettant son déclenchement, et **d'actions**, qui sont le plus souvent des instructions d'affectation de variables d'état, par des valeurs constantes ou par des valeurs calculées à partir de X. Il existe un autre type d'action possible, employé uniquement pour l'instant dans la base de connaissances TOPASE [22] : la résolution d'un système d'équations linéaires, qui permet de calculer une nouvelle valeur de X.
Le moteur d'inférence applique cycliquement les règles (R_k) *ordonnées suivant l'ordre σ* jusqu'à obtenir la même valeur pour X à la fin de deux "tours" successifs des règles.
Appliquer une règle consiste à évaluer sa condition, et, si celle ci est vraie, réaliser l'action correspondante. En particulier, si la condition n'est pas réalisée, le vecteur X n'est pas modifié.

Les différents éléments d'un modèle FIGARO0 étant décrits, il est facile de définir la sémantique du langage FIGARO0. Cette sémantique équivaut au fonctionnement de l'automate suivant :

Initialisation :
Un état initial **incomplet** est défini par l'utilisateur par la donnée de V_0 (seule contrainte : respect du domaine de V). Puis calcul de $X_0 = I(V_0, Y_0)$ qui est l'état initial complet du modèle.

Evolution à partir d'un état courant X :

$O(X)$ définit les groupes de transitions applicables à partir de X.
- Si cet ensemble est vide, l'état courant est absorbant.
- S'il est réduit à un groupe contenant une seule transition t, l'unique état que peut prendre l'automate, à partir de l'état X, est $I(t(X))$.
- S'il contient plusieurs groupes de transitions, deux cas sont à considérer (la décomposition ci-après constitue une partition de l'ensemble des situations possibles) :

- *O(X)* contient uniquement des transitions de lois continues. Dans ce cas, l'état *X* est dit "tangible" et le prochain état que prendra l'automate sera *I(t(X)), t* étant l'une quelconque des transitions de lois continues.
- *O(X)* contient les groupes $G_1, G_2.., G_n$ de transitions de loi discrète. L'état *X* est alors dit "instantané", et les transitions de lois continues sont **ignorées**. Dans ce cas, le prochain état que prendra l'automate sera $I(t_1 \circ t_2 \circ ... t_k (X))$, expression dans laquelle t_i représente un choix quelconque (non déterminisme) de transition dans le groupe G_i : $t_i \in G_i$. L'ordre d'application des transitions (autrement dit le choix des indices pour les groupes) n'est pas précisé, car les modèles peuvent facilement être écrits de manière que cet ordre soit indifférent, ainsi que cela est expliqué dans [8].

Commentaires sur les différences entre les deux modèles

Afin de ne pas désorienter le lecteur par des modèles écrits suivant des philosophies trop différentes, nous avons écrit le modèle FIGARO dans un "style AltaRica", c'est à dire avec des liens qui n'ont aucun comportement propre. Le type arete_bi_dir ne sert qu'à contenir les variables décrivant des flux arrivant et partant des nœuds.
L'utilisation de l'héritage a permis de "factoriser" la plupart des règles d'interaction dans le modèle FIGARO. Ces règles, très génériques, expriment que les nœuds transmettent la tension qu'ils reçoivent[2] par un ou plusieurs liens aux autres liens, à condition de ne pas être isolants. Sont considérés comme isolants les sources et récepteurs en panne, et les interrupteurs ouverts. Les sources en état de marche sont des sources de tension. De manière à peu près symétrique, les nœuds propagent une information court-circuit, initiée par les récepteurs.
Pour les raisons expliquées dans le paragraphe "Robustesse", la modélisation AltaRica peut poser des problèmes en cas de topologie bouclée, alors que la base de connaissances FIGARO permet de modéliser aisément des systèmes bouclés, voire maillés. La topologie en H choisie pour notre exemple ne présente pas ce type de difficulté.
On retrouve en AltaRica (dans les assertions du node t_jonction) l'équivalent des deux règles FIGARO génériques du type noeud "regle_CC_non_recepteur" et "regle_tension_non_source", sous une forme spécialisée : 6 règles valides seulement pour un nœud de jonction à trois sorties.
La modélisation de la destruction mutuelle de deux sources lorsqu'elles sont reliées entre elles via une propagation locale d'information, uniquement par les objets physiques, aurait conduit à un modèle lourd en FIGARO. C'est pourquoi nous avons préféré utiliser une technique à laquelle ont souvent recours les concepteurs de bdc en FIGARO : la création d'objets dits "déduits" de la topologie du système, à savoir (ici) les chemins. Pour un cas aussi simple, cela n'a pas d'intérêt, mais signalons qu'un des points forts du langage FIGARO est de servir de base à un langage de haut niveau permettant de spécifier aisément des algorithmes de construction **automatique** d'objets déduits.

Comparaison des langages

Syntaxe et concepts de base

Sur ce sujet, deux points de vue s'opposent : les partis pris dans les deux langages sont très différents.
En AltaRica, le nombre de mots-clés est réduit, et la syntaxe rappelle celle d'un langage de programmation. Seule une version anglaise d'AltaRica a été définie.
Le langage FIGARO existe en Français et en Anglais, et fait appel à de nombreux mots-clés : on y trouve les équivalents de presque tous les mots-clés d'AltaRica, plus diverses primitives de haut niveau. Pour faciliter la lecture des modèles, les mots-clés ont été choisis très explicites[3], ce qui permet d'écrire des expressions proches du langage naturel.
Le petit nombre de concepts et de mots-clés définis en AltaRica fait que son apprentissage est beaucoup plus rapide que celui de FIGARO. Cet inconvénient de FIGARO est compensé par sa puissance de modélisation, sa capacité à décrire des modèles plus génériques, et le fait qu'il capitalise beaucoup d'expertise fiabiliste.

[2] "recevoir de la tension" signifie être à l'arrivée d'un lien dont le point de départ est alimenté ou être au départ d'un lien dont le point d'arrivée est alimenté. Grâce à cette double définition, on peut tirer un lien d'un objet A à un objet B, ou dans l'autre sens sans que cela change le fonctionnement simulé. C'est pourquoi les liens sont "bidirectionnels".
[3] C'est ainsi que le mot-clé IL_PEUT_SE_PRODUIRE de FIGARO a pour équivalent le mot-clé |- en AltaRica, bien plus court, mais peu parlant.

Puissance de modélisation

Par "puissance de modélisation", nous entendons facilité pour modéliser des interactions complexes entre objets d'un système. FIGARO n'est pas soumis aux contraintes liées à une description locale : en FIGARO0, tout objet peut avoir un impact direct sur n'importe quel attribut d'un autre (en FIGARO1, il faut passer par les interfaces). De plus FIGARO offre une grande concision et généricité permises par les quantificateurs pour décrire des interactions complexes. Ainsi, la puissance de modélisation de FIGARO est incontestablement meilleure. Précisons ceci.

Intérêts et contraintes d'une description locale

En AltaRica, chaque objet est considéré comme une boîte qui a des interfaces accessibles (des flux) et dont l'état interne est encapsulé. La relation entre les interfaces d'un composant est en premier lieu contrainte par les clauses assert du composant et dans le fragment AltaRica DataFlow, chaque composant définit comment calculer les valeurs de ses variables de sortie en fonction de son état interne et des valeurs de ses variables d'entrée. Ce qui provoque les changements d'état du système, ce sont les changements "spontanés" d'états internes de composants, liés à des phénomènes aléatoires (défaillances, réparations...) ou déterministes (fin d'une temporisation...).
L'avantage du caractère local des descriptions est qu'il est généralement plus facile d'appréhender un modèle complexe avec cette philosophie. En effet, l'effort du concepteur porte sur l'analyse de chaque type de composant, en tentant d'imaginer tous les cas qui se présentent localement. Puis il pourra s'assurer que les descriptions locales permettent bien d'activer les schémas de propagation globaux qu'il a en tête. Enfin, grâce aux outils il pourra examiner la combinatoire des cas locaux et éventuellement découvrir à l'aide des outils des chemins de propagation de défaillances qui n'avaient pas été imaginés initialement lors de la conception. Cette intuition a d'ailleurs été corroborée par plusieurs retour d'expérience [9].
Cependant la contrainte "toute description doit être locale" peut introduire des lourdeurs ou des éléments peu intuitifs dans la modélisation. Considérons par exemple la destruction par influence mutuelle des deux sources de notre exemple. En AltaRica, un objet n'a pas le droit de modifier directement les variables d'état des sources pour les déclarer détruites (ce que fait l'objet chemin_1 du modèle FIGARO).

Pour modéliser ceci, deux solutions sont envisageables. La première consiste à introduire deux événements synchronisés, situés dans les deux sources. Ceci présuppose que le modélisateur sait qu'il existe un chemin susceptible de mettre en relation les deux sources. Aussi cette solution a-t-elle été écartée.
La seconde, retenue dans l'exemple, consiste à véhiculer l'information par les chemins de flux existants. Les ports électriques des composants ont donc été systématiquement dédoublés en fils d'entrée et fils de sortie pour pouvoir recevoir des informations sur les états des autres composants et propager leur propre statut. Ainsi, lorsqu'une source détecte une tension venant de l'extérieur, elle mémorise d'abord cette détection via la variable interne detect_conflit puis s'auto-détruit puisque l'événement prioritaire conflit_sources devient tirable. Un relecteur de ce type de modèle pourra s'interroger sur l'opportunité d'avoir ces deux transitions au lieu d'une seule. En fait, la destruction est réalisée en deux temps pour mémoriser le conflit dans les deux sources. Autrement, le conflit apparaîtrait dans une première source, qui s'autodétruirait et ferait ainsi disparaître le conflit pour la source restante. Ce type de motivation doit être tracé. En outre, il faut noter que cette solution n'est possible qu'à cause de la structure très simple du système. Il suffirait qu'il contienne une boucle topologique pour qu'une source ne soit pas capable de distinguer si une tension qu'elle reçoit vient d'une autre source, ou bien d'elle même (ce qui ne provoquerait pas de conflit).
Dans le modèle FIGARO, le rôle de l'objet chemin est limpide, car il est décrit par une règle unique. Il suffit de supprimer l'objet chemin_1 du modèle, pour changer les hypothèses et considérer que si les sources sont mises en relation, rien ne se passe (ce serait le cas en courant continu). En fait, la solution prise en FIGARO équivaut à la solution des transitions synchronisées en AltaRica, mais en plus flexible.

Utilisation des quantificateurs

Grâce aux quantificateurs, on atteint un meilleur niveau d'abstraction, ainsi que nous l'avons expliqué dans [14]. Cela a des conséquences sur les représentations graphiques, que nous mentionnerons plus loin. Par ailleurs, grâce à l'utilisation combinée des quantificateurs et de l'héritage, on optimise le nombre de déclarations de variables dans une base de connaissances. Ainsi dans notre exemple, il y a 36 déclarations de variables d'état dans le modèle AltaRica, et seulement 12 dans la base de connaissances FIGARO.

Réutilisation

La question de la réutilisation peut être envisagée sous deux angles : réutilisation dans un modèle AltaRica ou FIGARO du modèle du fonctionnement nominal d'un système, et réutilisation de modèles AltaRica ou FIGARO précédemment écrits.

Réutilisation d'un modèle de fonctionnement nominal de système : réalité ou mirage ?

Les modèles FIGARO et AltaRica présentés ici ont été spécialement construits pour raisonner sur la propagation de défaillances au sein d'un système. Ils font abstraction des grandeurs physiques en jeu (volts ou ampères) pour mettre en évidence les bons et les mauvais comportements de manière plus qualitative. Or, il peut exister d'autres modèles axés sur les spécifications du comportement nominal du système. Peut-on étendre ces modèles pour construire des modèles de propagation de défaillances ? Ces idées ont été explorées dans [10] et [15] notamment. L'intérêt de l'approche dépend résolument des objectifs poursuivis. Ainsi, s'il s'agit d'effectuer une démonstration de sécurité à partir du modèle étendu, l'utilisateur rencontrera vraisemblablement des difficultés. En effet, les modèles fonctionnels peuvent être très détaillés et si complexes à analyser que les outils ne suivent pas. En outre, et c'est beaucoup plus gênant, il devient difficile de caractériser précisément et complètement les modes de défaillance. En conséquence, les analyses peuvent être difficiles à interpréter, voire infaisables pour cause d'explosion combinatoire. En revanche, en phase de conception, la simulation de ce type de modèle permet d'injecter des fautes et d'en analyser les effets locaux ou plus globaux.

La réutilisation de modèles FIGARO

Comme nous l'avons signalé en introduisant le langage FIGARO, ce dernier a bénéficié d'une grande stabilité. Constamment utilisé à EDF pendant plus de 15 ans, il n'a subi que deux évolutions majeures : l'introduction (en 1993) de la possibilité d'écrire des systèmes d'équations linéaires pour les besoins de l'outil TOPASE, et (en 2003) l'enrichissement de la notion de PANNE par l'ajout de modèles fiabilistes, modèles que l'on retrouve dans les feuilles des arbres de défaillances générés à partir d'un modèle FIGARO. Notre petit exemple, qui ne fait appel à aucun de ces concepts, aurait pu être écrit tel quel en 1990 ! Cette grande stabilité a permis de ne rien perdre de tous les développements de bases de connaissances réalisés à EDF. On ne part donc pratiquement jamais d'une "page blanche" pour créer de nouveaux modèles, mais plutôt en examinant quelle est, parmi la trentaine de bases de connaissances disponibles "sur étagère", la mieux adaptée au problème que l'on doit résoudre. Le facteur qui favorise le plus la réutilisation de parties de bases de connaissances FIGARO est le haut degré de généricité des règles, ce qui limite les adaptations que l'on doit faire quand on change de contexte, ainsi que le nombre de candidats que l'on doit passer en revue pour en trouver un réutilisable ; à l'opposé, ce qui peut la freiner est le fait qu'un objet peut interagir avec d'autres en modifiant leurs variables d'état, ou en voyant ses propres variables modifiées par les objets extérieurs. A cause de ce fait, il n'est pas prudent de remplacer sans précaution un type par un autre, sous prétexte que les noms et les types de leurs interfaces coïncident.

La réutilisation de modèles AltaRica

Le langage AltaRica étant plus récent, le retour sur la réutilisation de modèles de ce type est plus limité. Nous mentionnerons néanmoins une expérience réalisée dans le cadre du projet européens ESACS. Dans le cadre de ce projet, l'ONERA a développé deux bibliothèques pour modéliser deux systèmes de génération et distribution, l'un de puissance hydraulique, l'autre de puissance électrique, inspirés des systèmes embarqués dans les Airbus A320 et A330. Un utilisateur qui n'avait pas participé au développement des bibliothèques initiales a tenté de les réutiliser pour modéliser les systèmes électrique et hydraulique de l'A380. Ceci n'a pas été possible pour la bibliothèque hydraulique car la technologie support avait changé et introduisait de nouveaux modes de défaillances. Par contre, la bibliothèque électrique a pu être reprise et étendue de manière cohérente directement par le nouvel utilisateur. Nous pensons que ceci est dû aux points suivants. Tout d'abord, AltaRica est défini par peu de concepts. D'autre part, les composants ont le plus souvent une taille réduite et sont en nombre limité. Tout ceci favorise l'appropriation et la réutilisation des modèles.

Liberté de style

La richesse syntaxique du langage FIGARO ainsi que l'absence de contraintes telles que la nécessité de se cantonner à des descriptions locales font qu'il autorise une bien plus grande liberté de style qu'AltaRica dans l'écriture de modèles. Ce point peut être vu comme un avantage ou un inconvénient. C'est un inconvénient surtout si plusieurs personnes doivent collaborer à l'écriture d'une base de connaissances.

Robustesse

Par robustesse, nous entendons la capacité à détecter des incohérences d'une part, et d'autre part à aider l'utilisateur à débusquer de petites erreurs de modélisation. En effet, l'expérience prouve qu'en modélisation comme en programmation, il faut compter 50% du temps pour écrire une première version et 50% pour faire les quelques ajustements qui lui permettent de fonctionner correctement. Au-delà des erreurs syntaxiques, la principale source de difficulté est le non-déterminisme non souhaité du comportement des modèles. Il peut provenir principalement de deux sources : une boucle dans la définition de variables ou bien une incomplétude dans la définition des variables. Examinons comment ces différents problèmes sont pris en compte dans les deux langages.

Robustesse du langage AltaRica

Examinons le cas de la boucle. Supposons que nous ayons à saisir la topologie de la Figure 4, en prenant les deux interrupteurs conformes au type défini dans la base de connaissances AltaRica. Cette topologie ne comporte aucune source ni récepteur. Or, ces types de composants ont l'avantage de "couper les boucles".

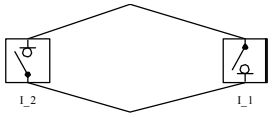

Figure 4 : un système conduisant à une indétermination

Ce modèle est indéterminé, car deux solutions sont possibles pour les variables de tension, correspondant à des valeurs toutes à Vrai, ou toutes à Faux. Il en est de même pour les variables de court-circuit. Cela va poser des difficultés aux outils d'analyse et complexifier aussi l'interprétation des résultats : ce type de cas doit être évité ou traité par des outils ad-hoc.

AltaRica étant inspiré par les langages formels, il hérite naturellement des techniques applicables dans ce domaine. La notion de boucle d'inter-définition de variables est un concept bien connu pour les langages réactifs à flots de données. Il existe d'ailleurs des algorithmes pour détecter statiquement des boucles potentielles et avertir l'utilisateur. En revanche, il n'existe pas d'outil permettant de couper automatiquement les boucles.

Une autre source d'incertitude peut provenir du fait que l'utilisateur pose des contraintes contradictoires sur les variables d'un composant, c'est-à-dire des assertions locales ou globales non satisfiables.

Enfin, il se peut que les définitions soient incomplètes car un cas d'entrées n'a pas été prévu. Lorsque les flux sont orientés, la complétude peut-être testée (voir [16]) ou mieux, résulter de la construction comme dans le fragment AltaRica DataFlow. Ainsi, pour ce fragment le test de complétude est ramené à un test de respect de la syntaxe.

Robustesse du langage FIGARO

Nous avons consacré un article complet [17] à la question de la détection d'incohérences dans les modèles en FIGARO. Nous y avons montré que la structure du langage permet à un vérificateur automatique de détecter de manière exhaustive toute incohérence d'une base de connaissances, qui pourrait apparaître lors d'une instanciation de système à l'ordre 0.

Les tests effectués par ce vérificateur, qui sont un préalable à toute utilisation d'une base de connaissances, permettent d'éliminer la grande majorité des fautes d'étourderie. Il reste donc à éliminer les fonctionnements indésirables qui apparaissent lorsqu'on assemble plusieurs composants pour créer un modèle de système. Dans l'article [17], nous avons répertorié ces fonctionnements indésirables et nous avons montré comment il est possible, par quelques "règles de style" simples, de s'en affranchir. Il s'agit essentiellement de l'utilisation de l'inférence monotone portant sur des variables déduites dans les règles d'interaction, qui assure qu'en **toutes situations**, la base de règles converge bien vers une solution unique (même en cas de topologie bouclée). Cette règle de style peut aisément être vérifiée par un outil. Nous l'avons bien sûr appliquée dans notre petit exemple de système électrique. Par exemple, la première étape sert à calculer (et ce ne sera jamais remis en question par les étapes suivantes) l'effet[4] "isolant" des composants. Dans les étapes suivantes, cet effet peut donc être considéré comme une donnée d'entrée figée, ce qui permet d'avoir des conditions du type "NON isolant" tout en restant dans le cadre de l'inférence monotone. Dans l'exemple de la Figure 4 ci-dessus, toutes les variables de flux étant définies par des effets (au sens FIGARO) , il n'y a qu'une solution, avec toutes les valeurs à Faux, ce qui correspond bien à l'intuition.

[4] Un effet est une variable déduite booléenne systématiquement remise à faux en début d'application des règles d'interaction.

Un autre avantage du langage FIGARO en termes de robustesse est le caractère générique des règles : il permet de faire rapidement des tests sur des topologies très diverses de systèmes. Par exemple, une fois que les règles de propagation des flux du type nœud ont été testées sur quelques assemblages de composants, il suffit d'en hériter pour les avoir à disposition, validées, dans tous les composants d'une base de connaissances, quelle que soit sa taille. En AltaRica, il faudrait tester les 6 règles de la jonction en T, mais aussi les 8 règles d'une jonction à 4 entrées, etc. : on voit que les risques d'erreur sont grands pour une base de connaissances ayant un minimum d'envergure.

Représentations graphiques

Association entre liens graphiques et objets du modèle

Les représentations graphiques associées au langage FIGARO exploitent fortement le fait qu'un lien aussi bien qu'un noeud peut correspondre à un objet FIGARO. C'est ainsi qu'il est possible de représenter de manière naturelle un câble électrique ou un tuyau possédant des caractéristiques propres telles qu'une impédance, un taux de défaillance etc. avec un lien.
On aurait très bien pu imaginer d'associer de la même manière un lien à un type AltaRica avec une entrée et une sortie uniques, correspondant respectivement au point de départ et au point d'arrivée du lien. Mais en fait dans tous les outils graphiques fondés sur AltaRica, les liens ont été cantonnés à un rôle très simple : faire apparaître visuellement l'identité entre une sortie donnée d'un objet et une entrée donnée d'un autre objet. Cela oblige à avoir trois objets graphiques là où un seul suffirait.

Variabilité des entrées-sorties

L'absence des quantificateurs dans le langage AltaRica fait que la seule souplesse dont on dispose quand on relie graphiquement des objets entre eux est la possibilité de relier une sortie à des entrées situées sur un ou plusieurs composants. En revanche, on doit avoir une bijection entre les points d'entrée et les liens arrivant sur un objet. Ainsi, pour décrire un circuit électronique de type voteur avec une logique en k/N, avec N = 2, 3 ou 4 entrées :

- avec le langage FIGARO, on a besoin d'un seul type paramétré par un entier k, et on peut brancher un nombre arbitraire d'entrées sur un objet de ce type,
- avec le langage AltaRica, on peut soit définir uniquement les portes ET et OU à deux entrées et représenter les différents voteurs par des assemblages de ce type de portes, soit définir autant de types de voteurs qu'il y a de couples (k, N) différents.

Les deux points mentionnés ci-dessus font que l'aspect graphique d'un modèle AltaRica peut apparaître bien plus chargé que celui d'un modèle FIGARO équivalent. Dans notre exemple, on a 10 objets graphiques dans le modèle FIGARO et 19 dans le modèle AltaRica.
Par ailleurs, à cause du dernier point ci-dessus, seule une interface graphique spécifiquement dédiée à cette tâche permettrait la saisie de modèles abstraits tels que les réseaux de Petri en vue de leur traduction en AltaRica. Au contraire, l'IHM de l'outil de modélisation KB3 fondé sur le langage FIGARO est polyvalente, ce qui a rendu possible la construction d'une base de connaissances "hybride" qui permet de décrire à la fois le schéma physique d'un système hydraulique et ses procédures d'exploitation sous la forme d'un réseau de Petri [14]. Grâce à ces outils, il a également été possible de concrétiser les concepts des BDMP (Boolean logic Driven Markov Process) ® à très peu de frais [21].

Aptitude à la transformation en d'autres langages

Possibilités de traduction automatique entre AltaRica et FIGARO

Voici ci-après deux tableaux qui présentent les caractéristiques du langage AltaRica sans équivalent en FIGARO et réciproquement, avec la façon éventuelle de contourner l'absence d'équivalent direct dans une optique de traduction automatique.

Caractéristique AltaRica sans équivalent en FIGARO	**Palliatif pour une traduction automatique AltaRica -> FIGARO**
Niveaux de détail	"Mise à plat" du modèle avant traduction
Règles avec des If Then Else imbriqués	Eclatement en plusieurs règles
Données composites (ex : l'association d'un booléen, d'un énuméré et d'un réel)	Eclatement en plusieurs attributs séparés
Priorités pour les transitions instantanées	Création d'un attribut global "priorité" et mise en place d'un mécanisme qui déclenche d'abord les règles d'occurrence contenant des transitions de priorité 1 puis celles de priorité 2 etc.
Synchronisation entre événements	Regroupement dans une règle d'occurrence unique des conditions des événements à synchroniser, et de l'ensemble des actions associées à ces événements.

Caractéristique FIGARO sans équivalent en AltaRica	**Palliatif pour une traduction automatique FIGARO -> AltaRica**
Hiérarchie des types et héritage	Instanciation en FIGARO d'ordre 0 avant la traduction.
Quantificateurs et possibilité de brancher un nombre non prévu à l'avance de liens sur un point de connexion unique. Contrôle sur la cardinalité.	Instanciation en FIGARO d'ordre 0 avant la traduction.
Définition d'un système d'équations linéaires sur un ensemble d'attributs réels. Cette notion permet de modéliser par exemple les lois de Kirchoff dans des systèmes électriques maillés.	Aucun pour une traduction en AltaRica Dataflow. En AltaRica général, on pourrait imaginer de faire l'équivalent, mais aucun outil n'existe pour traiter un tel modèle.
Groupes de règles, permettant d'avoir plusieurs variantes de comportement associées à un système saisi une seule fois.	Instanciation en FIGARO d'ordre 0 en sélectionnant les groupes que l'on souhaite avant la traduction.
Etapes dans les règles d'interaction (cf. plus haut la définition de la sémantique de FIGARO0)	Il n'y a pas de solution de complexité "raisonnable". Il faudrait transformer la majorité des règles d'interaction FIGARO en transitions spécifiques. Les dépendances temporelles entre ces différentes transitions pourraient être établies soit en introduisant des variables d'états n'ayant pour but que de séquencer les transitions, soit en utilisant des priorités ou des synchronisations, voire en mixant les trois types de constructions selon les cas. Ceci nous éloignerait considérablement de la philosophie du langage AltaRica et créerait un modèle parfaitement illisible et sans doute peu performant.
Possibilité de définir des topologies bouclées sans création d'incomplétude (grâce au principe de l'inférence monotone, cf. [17])	Aucun
Possibilité pour une règle contenue dans un objet X de modifier un attribut d'un objet Y différent de X (cf. exemples en langage FIGARO d'ordre 1 dans [6])	Mise en place d'un mécanisme tel que celui décrit dans le § "Intérêts et contraintes liées à une description locale", avec des variables intermédiaires, ou utilisation de transitions synchronisées.

Ces deux tableaux montrent qu'il est possible de traduire automatiquement tout modèle AltaRica en FIGARO, mais que la traduction en sens inverse suppose le respect de certaines restrictions par le modèle FIGARO de départ.
Un traducteur (gratuit) AltaRica DataFlow → FIGARO0 est mis à disposition sur le site internet [18].

Possibilités de traduction automatique vers des modèles formels de systèmes dynamiques

Ainsi que cela transparait dans le titre de la thèse [7], la définition d'AltaRica a intégré dès le départ un objectif de traduction aisée vers les langages formels traditionnellement utilisés pour traiter des systèmes

dynamiques à événement discrets tels que les automates à états. Un intérêt de ces langages est de servir d'entrée à des outils de model-checking qui garantissent une exploration exhaustive du graphe d'états du système [19]. Cette exigence ne pouvait pas exister à l'époque de la création de FIGARO, car les techniques de model-checking n'étaient alors qu'embryonnaires.
Il en résulte qu'il est relativement aisé de faire du model-checking à partir de modèles AltaRica, soit directement avec des outils dédiés à ce langage [20], soit après traduction automatique, par exemple en syntaxe SMV [13].
En revanche, il est beaucoup plus difficile de traduire un modèle FIGARO en un modèle acceptable par un model-checker, à cause des même points que ceux signalés dans le tableau concernant la traduction FIGARO vers AltaRica. Il serait malgré tout envisageable de développer un outil capable de traduire automatiquement des modèles FIGARO respectant certaines restrictions en modèles écrits pour des model-checkers. Un prototype de traducteur FIGARO vers SMV a d'ailleurs été développé dans le cadre d'une collaboration entre EDF et l'ONERA.

Types de résultats que l'on peut obtenir

Avec les outils qui ont été spécifiquement développés pour interpréter les langages AltaRica (à condition de se limiter à la version DataFlow) ou FIGARO, il est possible à partir de ces deux types de modèles, de faire les traitements suivants :

- génération d'arbres de défaillances (sous réserve que le modèle satisfasse certaines conditions d'indépendance des événements associés aux pannes),
- simulation interactive,
- simulation de Monte-Carlo,
- génération du graphe de tous les états atteignables à partir de l'état initial,
- quantification probabiliste du graphe si toutes ses transitions correspondent à des événements associés à des lois exponentielles ou instantanées (modèle Markovien),
- recherche (et éventuellement quantification) de séquences menant à des états possédant certaines caractéristiques…

A cette liste commune s'ajoutent des traitements plus spécifiques : model-checking pour AltaRica et génération d'objets déduits pour FIGARO.

Autres langages de modélisation pour les études de sûreté de fonctionnement

Cette section nous permet de revenir sur les points partagés par les deux langages.
En effet, les modèles FIGARO et AltaRica se distinguent assez naturellement des modèles traditionnels pour la SdF comme les arbres de défaillances statiques ou dynamiques pour au moins une même raison. Les modèles traditionnels ont pour but de capturer des chaînes spécifiques de causalité liant un événement redouté à des événements élémentaires, alors que les modèles AltaRica ou FIGARO tentent d'offrir un modèle plus proche de l'architecture complète du système, susceptible de couvrir toutes les chaînes de causalité. Les deux langages favorisent alors la gestion des modifications des systèmes complexes car il est plus simple de répercuter une modification sur un unique modèle du système que sur une forêt d'arbres.

Comme FIGARO et AltaRica visent à modéliser aussi bien des systèmes statiques que dynamiques, les modèles théoriques sous-jacents les plus simples sont des automates, des processus ou des chaînes de Markov. Ce type de modèles peut être généré à partir de notations de plus haut niveau comme les Réseaux de Petri, les Statecharts, les langages réactifs à flots de données étendus pour prendre en compte les aspects probabilistes. Néanmoins, ces langages n'ont pas été créés pour la SdF, et s'il est possible de réaliser des modèles équivalents du point de vue des comportements générés, leur réalisation peut s'avérer nettement plus lourde.
En effet, comme nous l'avons vu dans les modèles présentés, les éléments clés à modéliser sont

- les états d'erreur ou de fonctionnement, qui qualifient l'aptitude d'un composant à se comporter selon certaines modalités,
- les événements provoquant un changement d'état interne, tels qu'une défaillance ou un changement de mode de fonctionnement,
- les fonctions assurées dans un état normal ou défaillant.

Or, si la notion d'événement, de changement d'état appartient au domaine des automates, la notion de fonction impose d'identifier des interfaces et des relations entre entrées et sorties. Ainsi, les concepts de variables d'état essentielles et déduites, de règles d'occurrence et d'interaction du langage FIGARO se retrouvent en AltaRica, respectivement sous la forme des variables dites d'état et de flux (state et flow), de clauses trans et assert.
De plus, pour modéliser naturellement la propagation des défaillances, nous avons vu l'utilité des mécanismes permettant de donner une priorité à certains changements d'état sur d'autres (étapes Figaro ou priorités AltaRica) ou de pouvoir modifier simultanément l'état de plusieurs objets.
La réunion de toutes ces caractéristiques, qui apparaissent donc incontournables, n'appartient, à notre connaissance, qu'à ces deux langages.
Il existe différents environnements offrant potentiellement les mêmes outils de modélisation et de calcul, comme par exemple l'environnement RODON®. Nous n'avons pas pu consulter à ce jour la documentation de ce type d'outils. Nous ne savons donc pas si ces outils reposent sur un langage public comme FIGARO ou AltaRica ou sur des formats propriétaires, pas nécessairement accessibles ni formalisés.

Conclusion

Lorsque nous avons commencé ce travail de comparaison, nous pensions que nous pourrions le conclure en donnant le cahier des charges d'un nouveau langage susceptible de remplacer à terme à la fois le langage FIGARO et le langage AltaRica. Mais la comparaison détaillée que nous avons effectuée a montré qu'au delà de différences syntaxiques secondaires telles que le choix des mots-clés équivalents, les langages FIGARO et AltaRica diffèrent profondément dans leur philosophie et leur mode d'utilisation. Nous pensons donc que les futurs utilisateurs de ces langages devront se poser très tôt les questions que nous avons abordées dans cet article, pour éviter d'avoir à réécrire dans un langage des modèles qu'ils auront saisis dans l'autre.
En résumé, le langage FIGARO est mieux adapté pour **qui veut concevoir des outils** de modélisation associés à des langages graphiques évolués, permettant de saisir soit des schémas de principe très proches de ceux dont les concepteurs de systèmes ont l'habitude, soit des modèles abstraits tels que les réseaux de Petri ou les BDMP [21], soit enfin des modèles "hybrides" mélangeant les deux. Son utilisation est destinée à un petit nombre de personnes, formées pour être capables de créer des outils clés en main pour les utilisateurs qui réaliseront des études. Ces derniers n'auront alors qu'à manipuler une interface entièrement graphique et n'auront pas une ligne de langage FIGARO à écrire.
Le langage AltaRica, plus simple, peut être utilisé directement par les personnes **qui font les études de systèmes**. On peut même dire qu'il **doit** être utilisé par ces personnes, car il ne permet pas d'écrire des bibliothèques de composants aussi génériques que celles que l'on peut développer en langage FIGARO. En contrepartie de cet inconvénient, les modèles AltaRica sont facilement "compilables" dans d'autres formalismes, en particulier ceux qui servent de point d'entrée aux outils de model-checking. C'est donc le langage AltaRica qui donne accès à la plus large palette d'outils de traitement, y compris ceux qui ont été conçus pour le langage FIGARO grâce à un traducteur automatique permettant de transformer tout modèle AltaRica en un modèle strictement équivalent en langage FIGARO d'ordre 0.

Références

[1] S. V. Rice, A. Marjanski, The SIMSCRIPT III programming language for modular object-oriented simulation, Proceedings of the 2005 Winter Simulation Conference.
[2] M. Bouissou, H. Bouhadana, M. Bannelier, Un moyen de réaliser l'unification de diverses modélisations pour les études probabilistes : le langage FIGARO, Note interne EDF HT-53/90-42A Juillet 1990.
[3] C. ANCELIN, M. BANNELIER, H. BOUHADANA, M. BOUISSOU, J.Y. LUCAS, L. MAGNE, N. VILLATTE, Poste de travail basé sur l'intelligence artificielle pour les études de fiabilité, Revue Française de Mécanique, Numéro spécial "les systèmes experts et la mécanique" N°1990-2, ISSN 0373-6601.
[4] M. BOUISSOU, Recherche et quantification automatiques de séquences accidentelles pour un système réparable, 5ème congrès de fiabilité et maintenabilité, Biarritz, octobre 1986.
[5] M. Bouissou, H. Bouhadana, M. Bannelier, N. Villatte. Knowledge modelling and reliability processing : presentation of the FIGARO language and associated tools Safecomp'91, Trondheim (Norvège), novembre 1991.
[6] M. BOUISSOU, S. HUMBERT, S. MUFFAT, N. VILLATTE, KB3 tool: feedback on knowledge bases, ESREL 2002, Lyon (France), March 2002.

[7] Gérald Point, AltaRica : Contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement. Thèse, LaBRI, Université Bordeaux I, Janvier 2000.
[8] A. Arnold, A. Griffault, G. Point, and A. Rauzy. The AltaRica formalism for describing concurrent systems, Fundamenta Informaticae, 40:109-124, 2000.
[9] P. Bieber, C. Bougnol, C. Castel, J.-P. Heckmann, C. Kehren, S. Metge, C. Seguin, Safety Assessment with AltaRica - Lessons learnt based on two aircraft system studies, 18th IFIP World Computer Congress, Topical Day on New Methods for Avionics Certification, Toulouse (France), August 26th, 2004.
[10] O. Akerlund, P. Bieber, E. Boede, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, J. Gauthier, O. Lisagor, A. Lüdtke, S. Metge, C. Papadopoulos, T. Peikenkamp, L. Sagaspe, C. Seguin, H. Trivedi et L. Valacca, ISAAC, a framework for integrated safety analysis of functional, geometrical and human aspects, proceedings Embedded Real Time Software, 2006, Toulouse (France), Janvier 2006.
[11] Claire Pagetti, Extension Temps Réel d'AltaRica, Thèse de doctorat, Ecole Centrale de Nantes, Université de Nantes, Avril 2004.
[12] Christophe Kehren, Motifs d'architecture de sûreté de fonctionnement, thèse de doctorat, Ecole Nationale Supérieure de l'Aéronautique, Toulouse, Déc. 2005.
[13] C. Kehren, C. Seguin, P. Bieber; C. Castel, C. Bougnol, J.-P. Heckmann et S. Metge, Advanced Multi-System Simulation Capabilities with AltaRica, proceedings of Safety Critial System Conference, Rhodes Island, USA, August 2004.
[14] Compte-rendu de la Journée du 9 mai 2005 de démonstrations d'outils d'analyse système. Organisée par l'IMdR SdF, groupe de travail et de réflexion "Recherche méthodologique".
[15] A. Joshi et M. Heimdahl, Model-based safety analysis of Simulink models using SCADE Design Verifier, proceeding Safecomp 2005, Fredrikstad (Norway), Septembre 2005, vol 3688 LNCS, Springer.
[16] C. Castel et C. Seguin, Modèles formels pour l'évaluation de la sûreté de fonctionnement des architectures logicielles d'avionique modulaire intégrée, AFADL, 2001.
[17] M. Bouissou, J.C. Houdebine. Inconsistency detection in KB3 models, ESREL 2002, Lyon (France), March 2002.
[18] Téléchargement de KB3 et diverses bases de connaissances (dont BDMP) à l'adresse http://rdsoft.edf.fr. Version de démonstration gratuite sans limite de durée.
[19] Edmund M. Clarke, Orna Grumberg, Doron A. Peled, Model Checking. The MIT Press.
[20] Aymeric Vincent, Conception et réalisation d'un vérificateur de modèles AltaRica, Thèse, LaBRI, Université Bordeaux I, Déc. 2003.
[21] M. BOUISSOU, J.L. BON, A new formalism that combines advantages of fault-trees and Markov models: Boolean logic Driven Markov Processes, Reliability Engineering and System Safety, Vol. 82, Iss. 2, 149-163, Nov. 2003.
[22] E. Bourgade, T. Desmas, Substation reliability evaluation: a specific tool for designers. Reliability Engineering & System Safety, Volume 46, Issue 1, 1994, Pages 63-73.
[23] http://altarica.labri.fr/ site dédié aux travaux sur le langage AltaRica, ses variantes et les outils associés.